UNIVERSITY OF CALIFORNIA
RIVERSIDE

Improving the Efficacy of Automated Test Generation

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jinghan Wang

December 2022

Dissertation Committee:

Dr. Chengyu Song, Chairperson
Dr. Heng Yin
Dr. Rajiv Gupta
Dr. Zhiyun Qian

The Dissertation of Jinghan Wang is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First of all, I would like to express my extreme gratitude to my advisor Prof. Chengyu Song. I could not have completed my research work without his immense knowledge, insightful advice, persistent encouragement, and constant support. I am also sincere grateful to my co-advisor Prof. Heng Yin. He brought me into the frontier of system security research. His sharp comments and suggestions inspired me in all the time of research. I also want to express my sincere respect to the academic enthusiasm and professional dedication of my advisors.

Besides, I would like to thank the rest of my dissertation committee: Prof. Rajiv Gupta and Prof. Zhiyun Qian, for willing to serve on my dissertation committee, but also for the insightful comments and suggestions.

I would like to thank my fellow labmates: Mu, Aravind, Qian, Andrew, Xunchao, Yue, Rundong, Brain, Wei, Minghua, Sina, Lian, Jie, Zhenxiao, Jue, Lei, Yaowen, Pan, Jianlei, Xuezixiang, Yu, Sheng, Lutfor, Ju, Chuan-Yu, Haochen, and Mingjun, for the stimulating discussions, for their help during my hard times, and for all the fun we have had in the last eight years.

Last but not least, I would like to give special thanks to my family for their love and support. I am proud to acknowledge my parents Qifa Wang, Bangrong Wang for their continuous support and encouragement throughout my doctoral study. Also, I want to take this opportunity to express my deep love and overwhelming thank to my fiancée, Qing Yu. This dissertation will not be possible without your love and encouragement.

This dissertation includes previously published materials: "Be Sensitive and collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing" published in the International Symposium of Research in Attacks, Intrusions and Defences, September 2019, and "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing" published in the Network and Distributed System Security Symposium, February 2021.

To my parents and my fiancée for all the support.

ABSTRACT OF THE DISSERTATION

Improving the Efficacy of Automated Test Generation

by

Jinghan Wang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2022
Dr. Chengyu Song, Chairperson

Exploring the execution space is essential to many program analysis tasks such as finding vulnerabilities in the program under test. Starting from a corpus of initial inputs (a.k.a. seeds), automated test generation aims to find more and more inputs (or testcases) that exercise new program states, and hopefully, some inputs will reach interesting states, e.g., triggering a vulnerability. Based on the observation that under-tested code is more likely to have bugs, coverage-guided testing, which tries to maximize the code coverage, works very well in practice.

Notably, coverage-guided testing usually involves three stages: seed selection, seed schedule, and new input generation. This dissertation addresses three core problems in each stage and advances the state-of-the-art on automated test generation.

First, we conduct the first systematic study on the impact of coverage metrics. In particular, we formally define and discuss the concept of sensitivity to compare different coverage metrics. We show that certain program states (e.g., vulnerabilities) cannot be reached without enough sensitivity. We then selectively present several metrics with different

sensitivities, and evaluate them on a large set of programs. Results show that each metric has its unique merit in terms of vulnerability finding. However, there is no grand slam one that defeats all the others when the computational resources are limited. Specifically, a high sensitive coverage metric can select too many seeds that overwhelm the current scheduling algorithms and lead to poor performance.

Second, we aim to address the seed explosion problem caused by a sensitive coverage metric. To this end, we model this problem as a trade-off between exploration and exploitation. Then, we design a novel multi-level coverage metric that incorporates sensitive coverage metrics in a novel way. Combined with a reinforcement-learning-based hierarchical seed scheduler, our approach not only can trigger more bugs and achieve higher code coverage, but also can achieve the same coverage faster than existing approaches. However, we also discover that with a large amount of seeds, the input generation stage will become a bottleneck. That is, the likelihood of a (randomly generated) input being selected as a new seed decreases when the input size and the corpus size increase.

One way to improve the efficiency of input generation is to replace random mutation with path constraints solving, e.g., by leveraging concolic execution (CE). Ideally, each input generated by a concolic execution engine should visit a new state and be selected as a new seed. However, our study finds that a considerable amount (as high as 50%) inputs actually failed to reach new states thus are not selected, especially when handling format inputs. To address this problem, in the last piece of this dissertation, we propose format-aware solving that leverages path constraints to infer input format information, and leverages inferred format information to guide the constructing and solving of path constraints. Evaluation

shows that our approach can negate significantly more branches, lead to deeper new paths, and cover more code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Program testing is the act of exploring execution space to find vulnerabilities in the program under test. Automated test generation is the core technique in this literature. Starting from a corpus of initial inputs (a.k.a. seeds), it is expected to quickly find more and more inputs (or testcases) that exercise new program states, and hopefully, some inputs will reach interesting states, e.g., triggering a vulnerability. Based on the observation that under-tested code is more likely to have bugs, coverage-guided testing, which aims to maximize the code coverage, works very well in practice.

In particular, coverage-guided testing keeps track of the test generation via a feedback loop. Take greybox fuzzing [1, 51, 52, 66, 188], which is a representative technique in this literature, for instance, the loop involves three major stages in each iteration. (1) *Seed scheduling*: a seed is picked from the seed corpus according to the scheduling criteria. (2) *Seed mutation*: within a limited time budget, new inputs are generated by performing various random mutations on the scheduled seed. (3) *Seed selection*: each generated input is

fed to the program under test and evaluated based on the coverage metric; if the input leads to new coverage, it will be selected as a new seed and added back to the seed corpus. As this feedback loop continues, more coverage will be reached, and hopefully, bugs will be triggered.

Notably, the coverage metric is an essential parameter of coverage-guided testing, as it decides how to select new seeds. And this relies on its ability to preserve intermediate waypoints [120]. To better illustrate this, consider flipping a magic number check `a = 0xdeadbeef` as an example. If a fuzzer only considers edge coverage, then the probability of generating the correct `a` with random mutations is 2^{32}. However, if the fuzzer can preserve important waypoints, e.g., by breaking the 32-bit magic number into four 8-bit number [89], then solving this checking will be much more efficient since the answer can be generated from a sequence as `0xef`, `0xbeef`, `0xadbeef`, and `0xdeadbeef`. This check can also be solved faster by understanding distances between current value of `a` and the target value [32,33,38,54,150]. More importantly, recent research has shown that many program states *cannot* be reached without saving critical waypoints [108, 156]. In summary, coverage metrics determine how fast a specific program state, e.g., flipping a condition check and triggering a bug, can be reached and even whether it can be reached eventually.

Due to the importance of coverage metrics, a number of different coverage metrics have been developed and proposed [1, 32, 92, 120, 127, 134, 165, 188]. However, there lacks a formal way to define the differences among them uniformly. And little is known about how these differences will actually affect the fuzzing results in practice. More importantly, it is unclear whether there exists one coverage metric that is superior to all the other ones.

As the coverage metric selects more and more seeds, the limited processing capability of testing makes it essential for the scheduling criteria to prioritize some seeds over others in order to maximize the coverage. For example, AFL [188] prefers seeds with small sizes and short execution time to achieve a higher fuzzing throughput. Furthermore, it maintains a minimum set of seeds that stress all the code coverage so far, and focus on fuzzing them (i.e., prefers exploitation). AFLFAST [17] models greybox fuzzing as a Markov chain and prefers seeds exercising paths that are rarely exercised, as high-frequency paths tend to be covered by invalid test cases. LIBFUZZER [141] prefers seeds generated later in a fuzzing campaign. Entropic [15] prefers seeds with higher information gains. Notably, although existing seed schedulers work well on a normally sized seed corpus, they can hardly handle cases where seed explosion occurs as the coverage metric selects too many seeds.

Furthermore, a large amount of seeds, as well as increase input size, will make the input generation stage a bottleneck. That is, the likelihood of a (randomly generated) input being selected as a new seed decreases when the input size and the corpus size increase. Notably, many research efforts have proposed to utilize various techniques including taint analysis [6,32,33,38,54,94,134,161], static analysis [98], and deep learning [133,142] to locate key input bytes to mutate as well as proper values they are mutated to, aiming to improve the mutation efficiency. MOPT [104] and EMS [105] propose to find the optimal byte-level mutation strategies in order to generate inputs triggering unique paths. Angora [32] utilizes gradient descent search to improve the efficiency of mutations. In addition, to generate valid inputs when the program under test works on structured input format, various format-aware fuzzing techniques have been proposed. In particular, if the input format is known,

they utilize that to parse inputs into a tree-like structure, where structural mutations can be performed [4, 129, 159]; otherwise they infer an approximate input format to guide the mutation based on the insight that variances in program states caused by modifying certain input bytes can imply the underlying input format that is expected [6, 50, 96, 101, 183, 184].

Besides greybox fuzzing, concolic execution (CE) [19, 62, 130, 131, 149, 187] is a popular alternative regarding automated test generation, which performs exploration in a more systematic manner. More specifically, it executes the program under test with a concrete input, collects symbolic path constraints along the concrete execution path, and selectively negates symbolic branches to generate new test inputs. In consequence, CE can easily solve the aforementioned magic number check, which is challenging for greybox fuzzing. Notably, as CE is also driven by concrete inputs, a CE engine can be integrated into a coverage-guided testing scheme, where it acts as a special seed mutator to generate new inputs while sharing similar seed selection and scheduling approaches as greybox fuzzing [52, 187].

Input generation in CE is achieved via solving the negated path constraints, e.g., by consulting a satisfiability modulo theories (SMT) solver [10, 41]). Conceptually, this approach should be more efficient than greybox fuzzing, because the newly generated test input is expected to follow the same execution path prefix (as the input it is derived from) until the target branch the CE engine aims to negate, go to the opposite branch direction, and reach a new path. However, the solver may fail to find a satisfying input if the path constraints are too restrictive or too complex. At the same time, if the path constraints miss important conditions, the newly generated test input may fail to negate the target branch.

The execution path may diverge earlier and never reach the target branch. It is also possible that the execution reaches the target branch with a different set of constraints (due to an earlier deviation from the path prefix), thus invalidating the solution returned by the solver. This problem is further exacerbated when the program under test handles highly formatted inputs.

1.1 Thesis Statement

This thesis aims to *advance the state-of-the-art on automated test generation by (1) improving the efficacy of seed selection, (2) improving the efficiency of seed scheduling in greybox fuzzing, and (3) improving the efficiency of input generation in concolic execution.* We hope our work would stimulate developing a coverage-guided testing schemes where seeds are generated, via either a random mutator or a concolic execution engine, selected, and scheduled effectively.

We start with conducting the first systematic study on the impact of coverage metrics in coverage-guided greybox fuzzing [156]. In particular, we formally define and discuss the concept of *sensitivity* to distinguish different coverage metrics. Based on the different levels of sensitivity, we then present multiple representative coverage metrics with their variants. We conduct a study on these metrics with the DARPA CGC dataset [27], the LAVA-M dataset [44], and a set of real-world applications (a total of 221 binaries). We find that because a fuzzing instance has limited resources (time and computation power), (1) each metric has its unique merit in terms of flipping certain types of branches (thus

vulnerability finding), (2) there is no grand slam metric that defeats all the others, and (3) a combination of these different metrics can often achieve even better performance.

During the first study, we observe that a sensitive coverage will select many seeds that may cause seed explosion and exceed the fuzzer's capability to schedule. To address this problem, we have developed a novel hierarchical seed scheduler [157]. More specifically, we present a new coverage metric design called multi-level coverage metric, where we cluster seeds selected by more-sensitive metrics into a hierarchical tree using less-sensitive metrics. Next we model seed scheduling as a multi-armed bandit (MAB) problem [177] and design a reinforcement-learning-based hierarchical seed scheduling algorithm to balance between seed exploration (trying out other fresh seeds) and exploitation (keep fuzzing a few interesting seeds to trigger a breakthrough).

During the second study, we observe that random-mutation-based input generation becomes bottleneck when the seed size and the corpus size increase. Generally, in terms of generating specific inputs to satisfy hard branches, CE that is based on solving path constraints is more efficient than greybox fuzzing that is based on random mutations. However, when the program under test handles formatted inputs that consist of fields and chunks, it becomes more challenging for CE as the path constraints may miss important data dependencies that are implicitly indicated by the input format and not assessed in any conditional branches. Motivated by this, we have developed FORMATLY that aims to improve the efficacy of input generation for CE via taking input format into account. In particular, FORMATLY first infers input format information including field boundary, hierarchy, and type automatically via parsing the path constraints that are augmented with

runtime information. Afterwards, it utilizes the inferred format information to guide the path constraints constructing and solving.

1.2 Thesis Outline

The rest of the thesis is organized as follows: Chapter 2 provides more detail on greybox fuzzing, concolic execution, hybrid fuzzing and MAB model. Chapter 3 presents the study we conduct on coverage metrics. Chapter 4 presents the hierarchical seed scheduler to address the seed explosion problem. Chapter 5 presents the format-aware input generation for more effectivee concolic execution. Finally Chapter 6 concludes this thesis and discusses open problems for future work.

Chapter 2

Background

2.1 Greybox Fuzzing

Algorithm 1 illustrates the greybox fuzzing process. Given a program to fuzz and a set of initial seeds, the fuzzing process consists of a sequence of loops named rounds. Each round starts with selecting the next seed for fuzzing from the seed corpus according to the scheduling criteria. The scheduled seed is assigned to a certain amount of power that determines how many new inputs (or test cases) will be generated in this round. Next, inputs are generated through random mutation and crossover based on the scheduled seed. Compared to blackbox and whitebox fuzzing, the most distinctive step of greybox fuzzing is that, when executing a newly generated input, the fuzzer uses lightweight instrumentation to capture runtime features and expose them to the fitness function to measure the "quality" of a generated test case. Inputs with good quality will then be saved as a new seed into the seed corpus. This step allows a greybox to gradually evolve towards a target (e.g., more coverage).

From the description above, we can clearly see that there exist three key stages that drive greybox fuzzing: *seed scheduling*, *seed mutation*, and *seed selection*, as illustrated in Figure 2.1. Below, we discuss each of them in detail.

Algorithm 1: Greybox Fuzzing Algorithm

```
Input: target program P, set of initial seeds S^0
Output: unique seed set S*,
        bug-triggering seed set S^v
Data: seed s and test case I
1  Function Main(P, S^0):
2      S* ← S^0
3      S^v ← ∅
4      while true do
5          s ← SelectNextSeedToFuzz(S*)
6          s.power ← AssignPower()
7          while s.power > 0 do
8              s.power ← s.power − 1
9              I ← MutateSeed(s)
10             status ← RunAndEval(I)
11             if status is Bug then
12                 S^v ← S^v ∪ {I}
13             else if status is NewCovExplored then
14                 S* ← S* ∪ {I}
15             else
16                 continue // drop I
17             end
18         end
19         PayReward(s)
20     end
21 End
```

2.1.1 Seed Scheduling

This stage determines which seed will be scheduled in the next iteration and how much fuzzing time will be allocated for it. Various factors can be taken into account to determine the priority of each seed. AFL prefers faster execution and smaller seeds in order to ensure high throughput. AFLFast [17] prioritizes seeds exercising paths that are rarely

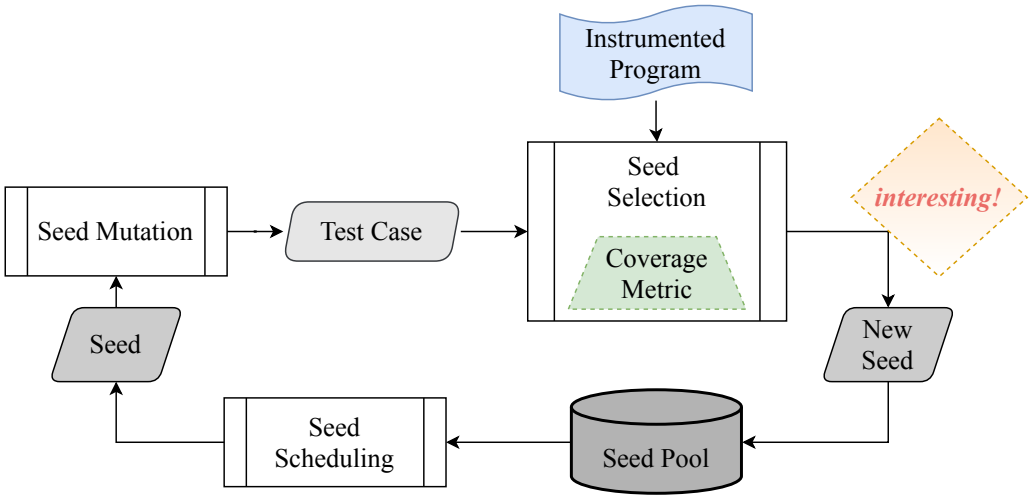


Figure 2.1: The workflow of greybox fuzzing.

covered by other testcases (implying that the path has not been fuzzed enough yet), and gives them exponentially more fuzzing time, aiming for better code coverage in a shorter time period. VUzzer [134] selects seeds that exercise deeper and rarer paths and avoids testcases that trigger error-handling, aiming for paths that are hard-to-reach and more useful. LIBFUZZER [141] prefers seeds generated later in a fuzzing campaign. Entropic [15] prefers seeds with higher information gains.

Seed scheduling can also be used to achieve goals other than code coverage. For example, AFLGo [16] performs effective *directed* fuzzing towards a given set of program locations by employing a simulated annealing based power scheduler that prioritizes seeds that are "closer" to the target locations and gives them more fuzzing time. The "distance" between a seed and the set of target locations is measured based on the distances from the code blocks exercised by the seed to the target code blocks, which are computed at compile time. In consequence, the generated testcases are getting increasingly closer to the set of target program locations. SlowFuzz [127] aims to discover algorithmic complexity vulnera-

bilities by prioritizing seeds that utilizes more system resources such as CPU, memory, and energy.

The pitfall in this step is that if the scheduling algorithms also rely on coverage information (e.g., AFLFast [17]), then imprecise coverage information could lead to sub-optimal decisions and miss potentially discoverable bugs.

2.1.2 Seed Mutation

This stage mutates a given seed to generate new testcases. A mutation strategy is concerned with (1) where to mutate and (2) how to mutate. AFL and libFUzzer utilize two kinds of mutations: deterministic and random mutations. Deterministic mutations include flipping bits in various granularities, as well as adding, subtracting, and inserting a set of predetermined integer values (e.g., 0, 1 and INT_MAX). Random mutations include inserting random byte chunks, deleting and modifying existing byte chunks at random offsets, as well as randomly splicing multiple seeds.

Several research efforts aim to improve this stage. For instance, VUzzer [134] leverages dynamic taint analysis to discover input bytes that will affect the control-flow, especially branches that compare the input bytes with magic numbers; then mutates these bytes to match the target constant. TaintScope [161] also utilizes dynamic taint analysis to discover input bytes that can affect sensitive operations (e.g., memory allocation size), and then mutates the bytes to trigger integer overflow. It also uses dynamic taint analysis to identify input bytes that are used to calculate checksum then patches the checksum function to avoid failure of integrity check. Lin et.al. [98] propose using static lineage analysis to identify sensitive bytes to mutate. Rajpal et.al. [133] from Microsoft propose using deep

neural network (DNN) to learn promising locations in inputs from a given set of seeds and then focusing on mutating these locations. MOPT [104] and EMS [105] propose to prioritize byte-level mutation strategies to make a good exploration of testcases triggering unique paths based on the insight that a program owns a certain distribution of mutation operators that can perform the best. SYMFUZZ [29] obtains deeper insights about seed mutation and designs an algorithm to adapt the mutation ratio (the rate between the number of modified bits and the number of total bits of a seed) to maximize the bug finding via detecting dependencies among the bit positions of the seed. In addition, SemFuzz [185] proposes a semantic-based approach to optimize seed mutation for generating testcases, which are in the form of sequences of system calls, as Proof-of-Concept exploits crashing Linux kernels. In more detail, first semantic information related to bugs is retrieved from public reports such as CVE descriptions and software patch logs. The information then is used to guide mutating a certain seed as how to change the order and parameters of the system calls.

2.1.3 Seed Selection

A seed selection strategy determines the trend and speed of the evolution of the fuzzing process. Essentially, a good seed selection strategy needs to solve two essential problems: (1) how to collect coverage information and (2) how to measure the quality of test cases.

Coverage Information Collection. AFL instruments the program under test to collect and compute the coverage. There are two instrumentation approaches. When the source code of the program under test is available, a modified `Clang` compiler is used to insert the

coverage calculation logic into the compiled executable at assembly level (normal mode) or intermediate representation level (fast mode). When the source code is not available, a modified user-mode QEMU is used to run the binary code of the tested program directly, and the coverage calculation logic is inserted during the binary translation phase. VUzzer [134] uses PIN [166] to perform binary instrumentation to collect the information. HonggFuzz [165] and kAFL [138] use hardware branch tracers like Intel Process Tracing (PT) to collect coverage information and DigTool [122] uses a hypervisor to collect coverage information from OS kernels.

Test Case Measurement. The quality of test cases is measured by leveraging coverage metrics. HonggFuzz [165] and Vuzzer [134] use basic block coverage metric that tracks visits of basic blocks. AFL [163] uses an improved branch coverage metric that could differentiate the visits to the same block from different preceding blocks. LibFuzzer [168] can use either block coverage or branch coverage. A more recent work Angora [32] extends the branch coverage metric with a calling context. MemFuzz [39] involves memory accesses, when calculating edge coverage to explore program states more pervasively. In addition, some research focus on finding domain-specific bugs via specifically designed coverage metrics [92, 120, 127]. Another important aspect is how the metric is really measured. Since coverage is measured during the execution of each test case, fuzzers usually prefer simpler implementations to improve the fuzzing throughput. For example, AFL identifies a branch using a simple hash function (Equation 3.1). Unfortunately, this approximation could reduce the effective sensitivity of a coverage metric due to hash collisions [55].

2.1.4 Format-Aware Greybox Fuzzing

Greybox fuzzing can generate a large number of new inputs in a short time via randomly mutating existing seeds, which are provided by users or from the procedure that inputs leading to new coverage are saved as new seeds. However, most of the generated inputs are invalid ones that will be dropped early by the program under test, especially when the program works on structured input formats. To mitigate this issue, various format-aware fuzzing techniques have been proposed [4, 6, 13, 50, 81, 96, 111, 129, 159, 183, 184].

Specifically, NAUTILUS [4] and SUPERION [159] utilize language grammars to parse inputs into abstract syntax trees (ASTs), where structural mutations such as deleting, adding, or splicing tree nodes can be performed. AFLSMART [129] proposes a similar approach but parses and represents inputs using a lightweight yet generic tree-like virtual structure.

A limitation of format-aware fuzzers is that they assume the format of the inputs is known to users, which often does not hold in practice. To solve this problem, some recent researches propose to infer an approximate format of inputs to guide the mutation based on the insight that the variances in program states caused by modifying certain input bytes can imply the underlying input format that is expected. For instance, REDQUEEN [6] mutates an input to maximize its entropy while keeping the coverage the same in order to identify input bytes that are directly copied into one comparison operand of conditional branches. These bytes are further marked as magic number or checksum according to the values they are compared against. PROFUZZER [184] mutates each individual byte in sequence with enumerating its values and observes the execution variations regarding edge coverage and

hit counts in response to the mutations. Afterwards, it groups consecutive bytes with similar variations as a field, and further classifies the field to a type according to the exposed pattern of the variations. Finally, specialized mutations are developed for each type of fields. Weizz [50] flips individual bits in each byte of an input for execution, and detects dependent conditional branches of which comparison operands vary but the hit count keeps the same. Then, it recovers plausible boundaries of chunks as well as fields based on heuristics about the dependencies to perform field- and chunk-level mutations. SLF [183] detects dependencies between input bytes and conditional branches in a similar way, but it recognizes only fields, and further classifies checks done by conditional branches into different types based on some rough heuristics, enabling field-level, type-specific mutations. TensileFuzz [101] follows the same way of identifying fields, and aims to improve seed growth in mutation-based fuzzing via modeling and solving string constraints. PATA [96] distinguishes between multiple occurrences of the same conditional branch when detecting dependencies between input bytes and conditional branches. But instead of enabling field- or chunk-level mutations, it aims to find path-aware byte-level mutations that can generate inputs passing certain hard branches.

2.2 Concolic Execution

2.2.1 Symbolic and Concolic Execution

The key idea of classic symbolic execution (SE) [8, 139] is to execute the program under test on symbolic rather than concrete inputs. To this end, SE maintains, as parts of the execution state, (1) a symbolic store that associates program variables with their

corresponding symbolic expressions, and (2) a set of symbolic path constraints from executed conditional branches. When meeting a conditional branch whose predicate is symbolic, SE tries to visit both branching targets (`taken` and `not-taken`) by forking the execution state and updating path constraints. A forked state can then be proceeded concurrently [23, 36] or sequentially [113, 144] with others, if its path constraint is assessed to be satisfiable by a SMT solver. As a result, SE can explore multiple paths at once to discover new coverage. But it may meet scalability problem while there are too many states being forked (i.e., path explosion).

Concolic execution (CE), as a modern variant of SE, strikes a balance between the scalability and exploration capability. In particular, starting with a concrete input, CE executes the program both concretely and symbolically. At each conditional branch, instead of forking, CE always follows the same branch direction taken by the concrete execution. To visit the opposite branching target, CE generate a new concrete input by asking a path constraint solver for a satisfying solution for the constraint $\pi \wedge \neg b$, where π is the constraint of the path prefix prior to the branch and $\neg b$ is the negated condition of the branch. If a solution is found, the newly generated input is expected to follow the original path until reaching the target branch, and then take the opposite one. By repeating this process, a wide variety of concrete inputs can be generated to drive CE to explore different paths.

2.2.2 Input Generation for Concolic Execution

Because CE relies on the newly generated concrete inputs to explore different execution paths, it is expected that each input can indeed reach the target branch, visit the

opposite direction, and explore a new path. Therefore, a fundamental research question in CE is: *how to construct the path constraints* π?

On one extreme, one can include all the constraints prior the target branch. While this approach guarantees that the newly generated input can *reach* the target branch, it has two main drawbacks. First, π could be *over-constrained*, meaning that it is impossible to negate the target branch following the same path prefix. Second, when the path constraints are complex, SMT solvers could fail to find a satisfying solution.

On the other extreme, one can use an empty π and only consider the negated branch condition $\neg b$ (a.k.a., optimistic solving). Apparently, because this approach is *under-constrained*, the newly generated input may never reach the target branch (i.e., early path divergence), or fail to visit the opposite direction, as the branch condition could change under a different path prefix.

Since it is impractical to brute-force all possible combinations of path constraints, we have to resort to various heuristics. Recent concolic executors like QSYM [187] leverage a two-tier strategy to construct path constraints. First, it considers prior branch constraints on which the target branch to negate has *direct data-dependency*. Thus, this strategy is named *nested solving*. Specifically, QSYM builds a dependency graph recursively by (1) finding all the input bytes involved in the current set of path constraints, and (2) adding all the branch constraints that involve the current set of input bytes; until the set stops growing. If nested solving fails, due to over-constrained or too complex (i.e., time out), QSYM switches to *optimistic solving*, which only considers the last negated condition $\neg b$. While an input generated by optimistic solving is expected to cause early path divergence

thus fails to negate the target branch, an input generated by nested solving is expected to negate the branch. In fact, once nested solving finds a solution, QSYM will mark the target branch as solved, and never try to negate it again, regardless of whether the new input can actually negate the branch or not. Unfortunately, our investigation revealed that nearly half of inputs generated by nested solving failed to negate the target branch.

2.3 Hybrid Fuzzing

The combination of blackbox/greybox fuzzing and concolic execution results in hybrid fuzzing. Notably, in most existing hybrid fuzzing frameworks, fuzzing and concolic execution run in independent instances that will synchronize seeds periodically or on demand, while each instance is with its own seed selection and scheduling approaches. Pak's master thesis [121] first uses symbolic execution to discover frontier nodes representing unique paths and then launches blackbox fuzzing to explore deeper code along the paths from these nodes. Stephens et al. [149] develop Driller that launches selective symbolic execution to generate new seed inputs when the greybox fuzzing could not make any new progress due to complex constraints in program branches. Furthermore, Shoshitaishvili et al. [145] extend Driller to incorporate human knowledge. DigFuzz [189] proposes a novel Monte Carlo based probabilistic model to prioritize paths for concolic execution in hybrid fuzzing. QSYM [187] designs a fast concolic execution engine that integrates symbolic execution tightly with the native execution to support hybrid fuzzing.

2.4 Multi-Armed Bandit Model

The multi-armed bandit model offers a fundamental framework for algorithms that learn optimal resource allocation policies over time under uncertainty. The term "bandit" comes from a gambling scenario where the player faces a row of slot machines (also known as one-armed bandits) yielding random payoffs and seeks the best strategy of playing these machines to gain the highest long-term payoffs.

In the basic formulation, a multi-armed bandit problem is defined as a tuple $(\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ is a known set of K arms (or actions) and $\mathcal{R}^a(r) = \mathbb{P}[r|a]$ is an unknown but fixed probability distribution over rewards. At each time step t the agent selects an arm a_t, and observes a reward $r_t \sim \mathcal{R}^{a_t}$. The objective is to maximize the cumulative rewards $\sum_{t=1}^{T} r_t$.

Initially, the agent has no information about which arm is expected to have the highest reward, so it tries some randomly and observes the rewards. Then the agent has more information than before. However, it has to face the trade-off between "exploitation" of the arm that is with the highest expected reward so far, and "exploration" to obtain more information about the expected rewards of the other arms so that it does not miss out on a valuable one by simply not trying it enough times.

Various algorithms are proposed to make the optimal trade-off between exploitation and exploration of arms. Upper Confidence Bound (UCB) algorithms [7] are a family of bandit algorithms that perform impressively. Specifically, they construct a confidence interval to estimate each arm's true reward, and select the arm with the highest UCB each time. Notably, the confidence interval is designed to shrink when the arm with its reward is sampled more. As a result, while the algorithm tends to select arms with high average

rewards, it will periodically try less explored arms since their estimated rewards have wider confidence intervals.

Take UCB1 [2], which is almost the most fundamental one, as an example. It starts with selecting each arm once to obtain an initial reward. Then at each time step, it selects arm a that maximizes $Q(a) + C \times \sqrt{\frac{log(N)}{n_a}}$ where $Q(a)$ is the average reward obtained from arm a, C is a predefined constant that is usually set to $\sqrt{2}$, N is the overall number of selections done so far, and n_a is the number of times arm a has been selected.

Particularly, seed scheduling can be modeled as a multi-armed bandit problem where seeds are regarded as arms [177, 186]. However, to make the fuzzer benefit from this model, such as maximizing the code coverage, we need to design the reward of scheduling a seed carefully.

Chapter 3

Systematic Study on the Impact of Coverage Metrics in Greybox Fuzzing

3.1 Introduction

Greybox fuzzing is a state-of-the-art program testing technique that has been widely adopted by both mainstream companies such as Google [170] and Adobe [172], and small startups (e.g., Trail of Bits [173]). In the DARPA Cyber Grand Challenge (CGC), greybox fuzzing has been demonstrated to be more effective compared to other alternatives such as symbolic execution and static analysis [36, 63, 147, 161, 164].

Greybox fuzzing generally contains three major stages: seed scheduling, seed mutation, and seed selection. From a set of seed inputs, the seed scheduler picks the next seed for testing. Then, more test cases are generated based on the scheduled seeds through mutation and crossover in the seed mutation stage. Finally, test cases of good quality are

selected as new seeds to generate more test cases in the future rounds of fuzzing. Among these stages, seed selection is the most important one as it differentiates greybox fuzzing from blackbox fuzzing and determines the goal of the fuzzer. For example, when the goal is to improve coverage, we use a coverage metric to evaluate the quality of a test case, and when the goal is to reach a particular code point, we can use distance to evaluate the quality of a test case [16]. Note that although previous studies [59, 79] have shown that better coverage of test suite is not directly related to a better quality of the tested software, the observation that under-tested code is more likely to have bugs still holds. For this reason, coverage-guided greybox fuzzing still works very well in practice.

Although various techniques have been proposed to improve greybox fuzzing at the seed scheduling stage [16, 17, 127, 134] and the seed mutation stage [98, 133, 134, 161, 185], very few efforts focus on improving seed selection. HonggFuzz [165] only counts the number of basic blocks visited. AFL [163] utilizes an improved branch coverage that also counts how many times a branch is visited. Angora [32] further extends the branch coverage to be context-sensitive. More importantly, many critical questions about coverage metrics remain unanswered.

First, *how do we uniformly define the differences among different coverage metrics?* Coverage metrics can be categorized into two major categories: code coverage and data coverage. Code coverage metrics evaluate the uniqueness among test cases at the code level, such as line coverage, basic block coverage, branch/edge coverage, and path coverage. Data coverage metrics, on the other hand, try to distinguish test cases from a data accessing perspective, such as memory addresses, access type (read or write), and access sequences.

While many new metrics have been proposed individually in recent works, there is no systematic and uniform way to characterize the differences among them. Apparently, different coverage metrics have very distinct capability of differentiating test cases, which we refer to as *sensitivity*. For example, block coverage could not tell the difference between visits to the same basic block from different preceding blocks, while branch coverage can. Therefore, branch coverage is more sensitive than block coverage. A systematic and formal definition of *sensitivity* is essential as it can not only tell the differences among current metrics but also guide future research to propose more metrics.

Second, *is there an optimal coverage metric that outperforms all the others in coverage-guided fuzzing?* Although sensitivity provides us a way to compare the capability of two coverage metrics in discovering interesting inputs, a more sensitive coverage metric does not always lead to better fuzzing performance. More specifically, fuzzing can be modeled as a multi-armed bandit (MAB) problem [177] where each stage (seed selection, scheduling, and mutation) has multiple choices, and the ultimate goal is to *find more bugs* with a limited time budget. A more sensitive coverage metric may select more inputs as seeds, but the fuzzer may not have enough time budget to schedule all the seeds or mutate them sufficiently. Implementation details such as how coverage is actually measured can further complicate this problem. For instance, a previous study [55] has shown that hash collisions could reduce the actual sensitivity of a coverage metric. A systematic evaluation is essential to understand the relationship between sensitivity and fuzzing performance better.

Third, *is it a good idea to combine different metrics during fuzzing?* Hypothetically, if different coverage metrics have their own merits during fuzzing, then it would make sense

to combine them so that different metrics could contribute differently. This question is also crucial as it motivates different thinking and may lead to strategies for improving fuzzing.

To answer the questions mentioned above, we conduct the first systematic study on the impact of coverage metrics on the performance of coverage-guided fuzzing. In particular, we formally define and discuss the concept of *sensitivity* to distinguish different coverage metrics. Based on the different levels of sensitivity, we then present several representative coverage metrics, namely "basic branch coverage," "context-sensitive branch coverage," "n-gram branch coverage," and "memory-access-aware branch coverage," as well as their variants. Finally, we implement six coverage metrics in a widely-used greybox fuzzing tool, AFL [163], and evaluate them with large datasets, including the DARPA CGC dataset [27], the LAVA-M dataset [167], and a set of real-world binaries. The highlighted findings are:

- Many of these more sensitive coverage metrics indeed lead to finding more bugs as well as finding them significantly faster.
- Different coverage metrics often result in finding different sets of bugs. Moreover, at different times of the whole fuzzing process, the best performer may vary. As a result, there is no grand slam coverage metric that can beat others.
- A combination of these different metrics can help find more bugs and find them faster. Notably, using less computing resources, a combination of fuzzers with different coverage metrics is able to find at least the same amount of bugs in the CGC dataset as Driller, a hybrid fuzzer augmented AFL with concolic execution did [149].

To facilitate further research on this topic, we have made the source code and dataset available at `https://github.com/bitsecurerlab/afl-sensitive`.

3.2 Sensitivity and Coverage Metrics

In this section, we formally define and discuss the concept of the sensitivity of a coverage metric. Accordingly, we present several coverage metrics that have different sensitivities.

3.2.1 Formal Definition of Sensitivity

When comparing different coverage metrics, a central question is "is metric A better than metric B?" To answer this question, we need to take a look at how a mutation-based greybox fuzzer finds a bug. In mutation-based greybox fuzzing, a bug triggering test case is reached via a chain of mutated test cases. In this process, if an intermediate test case is deemed "uninteresting" by a coverage metric, the chain will break and the bug triggering input may not be reached. Based on this observation, we decide to define *sensitivity* as a coverage metric's ability to preserve such mutation chains.

To formally describe this concept, we first need to define a coverage metric as a function $\mathcal{C} : (\mathcal{P} \times \mathcal{I}) \rightarrow \mathcal{M}$, which produces a measurement $M \in \mathcal{M}$ when running a program $P \in \mathcal{P}$ with an input $I \in \mathcal{I}$. Given two coverage metrics C_i and C_j, C_i is "more sensitive" than C_j, denoted as $C_i \succ C_j$, if

(i) $\forall P \in \mathcal{P},\ \forall I_1, I_2 \in \mathcal{I},\ C_i(P, I_1) = C_i(P, I_2) \rightarrow C_j(P, I_1) = C_j(P, I_2)$, and

(ii) $\exists P \in \mathcal{P},\ \exists I_1, I_2 \in \mathcal{I},\ C_j(P, I_1) = C_j(P, I_2) \wedge C_i(P, I_1) \neq C_i(P, I_2)$

The first condition means, for any program P, if any two inputs I_1 and I_2 produce the same coverage measurement using C_i; then they must produce the same measurement

using C_j, i.e., C_j is always not more discriminative than C_i. The second condition means, there exists at least a program P such that two inputs I_1 and I_2 would produce the same measurement using C_j but different measurements using C_i, i.e., C_i can be more discriminative than C_j.

3.2.2 Coverage Metrics

In this subsection, we introduce several coverage metrics and their approximated measurement. Then we compare their sensitivity.

Branch Coverage Branch coverage is a straightforward yet effective enhancement over block coverage, which is the most basic one that can only tell which code block is visited. By involving the code block preceding the currently visited one, branch coverage can differentiate the visits of the same code block from different predecessors. Branch here means an edge from one code block to another one.

Ideally, branch coverage should be measured as a tuple ($prev\_block$, cur_block), where $prev\_block$ and cur_block stand for the previous block ID and the current block ID, respectively. In practice, branch coverage is usually measured by hashing this tuple (as key) into a hash table (e.g., a `hit_count` map). For example, the state-of-the-art fuzzing tool AFL identifies a branch as:

$$block\_trans = (prev\_block << 1) \ \oplus \ cur\_block \tag{3.1}$$

where branch ID is calculated as its runtime address. The $block\_trans$ is then used as the key to index into a hash map to access the `hit_count` of the branch, which records how many times the branch has been taken. After a test case finishes its execution, its coverage information is compared with the global coverage information (i.e., a global `hit_count` map). If the current test case has new coverage, it will be selected as a new seed.

Although branch coverage is widely used in mainstream fuzzers, its sensitivity is low. For instance, considering a branch within a function that is frequently called by the program (e.g., `strcmp`). When the branch is visited under different calling contexts, branch coverage will not be able to distinguish them.

N-Gram Branch Coverage After incorporating one preceding block in branch coverage, it is intuitive to incorporate more preceding basic blocks as history into the current basic block. We refer to this coverage metric as *n-gram branch coverage*, where n is a configurable parameter that indicates how many continuous branches are considered as one unit, and any changes of them will be distinguished. When $n = 0$, n-gram branch coverage is reduced to block coverage. On the opposite extreme, when $n \to \infty$, n-gram branch coverage is equivalent to path coverage because it incorporates all preceding branches into the context and any change in the execution path will be treated differently.

Ideally, n-gram branch coverage should be measured as a tuple $(block_1, \ldots, block_{n+1})$. For efficiency, we propose to hash the tuple as a key into the `hit_count` map as $(prev\_block\_trans << 1) \oplus curr\_block\_trans$, where

$$prev\_block\_trans = (block\_trans_1 \oplus \cdots \oplus block\_trans_{n-1}) \tag{3.2}$$

In other words, we record the previous $n-1$ block transitions (calculated as in Equation 3.1) and XOR them together, left shift 1 bit, and then XOR with the current block transition.

Now an interesting question is: *what is the best value for n?* If n is too small, it might be almost the same as branch coverage. If n is too large, it may cause seed explosion (a similar phenomenon as path explosion). Fuzzing progress would be even slower due to the enormous amount of seeds.

To answer this question empirically, we adapt AFLFast to n-gram branch coverage where n is set to 2, 4, and 8. We will evaluate these settings in §3.3.

Context-Sensitive Branch Coverage A function lies between a basic block and a path with respect to the granularity of code. Therefore, calling context is another important piece of information that can be incorporated as part of the coverage metric, which allows a fuzzer to distinguish the same code executed with different data. We refer to this coverage metrics as "context-sensitive coverage metric."

Ideally, context-sensitive branch coverage metric should be measured as a tuple ($call\_stack$, $prev\_block$, $curr\_block$). For efficiency, we define a calling context $call\_ctx$ as a sequence of program locations where function calls are made in order:

$$call\_ctx = \begin{cases} 0 & \text{initial value} \\ call\_ctx \oplus call\_next\_insn & \text{if call} \\ call\_ctx \oplus ret\_to\_insn & \text{if ret} \end{cases} \tag{3.3}$$

As a result, the key-value pair stored in the bitmap can be calculated as $call\_ctx \oplus block\_trans$.

Initially, the calling context value $call\_ctx$ is set to 0. Then during the program execution, when encountering a `call` instruction, we XOR the current $call\_ctx$ with the instruction's position immediately next to the call instruction and store the result in $call\_ctx$. Similarly, when encountering a `ret` instruction, we XOR the current `call_ctx` with the return address. In this way, a small value $call\_ctx$ efficiently accumulates function calls made in sequence and eliminates function calls that have returned.

Memory-Access-Aware Branch Coverage In addition to leveraging extra control flow information as stated above, data flow information also deserves to be considered. Based on the intuition that a primary focus of fuzzing is to detect memory-corruption vulnerabilities, memory access information can be of great help in measuring coverage. Fundamentally, memory corruption exhibits an erroneous memory access behavior. Therefore, it makes sense to select seeds that exhibit distinct memory access patterns.

In general, this memory-access aware coverage metric is more sensitive than branch coverage. Because if a new test case reaches a branch that has been covered by prior test cases, but at least one new memory location is accessed, this test case will still be considered as "interesting" in memory-access aware coverage metric and kept as a seed.

There can be many ways to characterize memory access patterns. In this chapter, we investigate one design option. We instrument memory access operations of the program under test, and define each memory access as a tuple ($type$, $addr$, $block\_trans$), where

type represents access type (read or write), *addr* is the accessed memory location, and *block_trans* means after which branch this memory access is performed.

For efficiency, we propose to calculate the hash key as $(block\_trans \oplus mem\_ac\_ptn)$, where

$$mem\_ac\_ptn = \begin{cases} mem\_addr & \text{if read} \\ mem\_addr + half\_map\_size & \text{if write} \end{cases} \tag{3.4}$$

Note that reads are distinguished from writes by allocating their keys to different half regions of the map.

Since memory corruption is mainly caused by memory writes, it is meaningful to investigate a variant of memory access coverage: "memory-write-aware branch coverage." That is, we only instrument and record memory writes, but not reads, making it less sensitive.

3.2.3 Sensitivity Lattice

Obviously, $\succ$ is a strict partial order, because it is asymmetric (if $C_1 \succ C_2$, by no means $C_2 \succ C_1$), transitive (if $C_1 \succ C_2$ and $C_2 \succ C_3$, then $C_1 \succ C_3$), and irreflexive ($C_i \succ C_i$ is not possible). However, it is not a total order, because it is possible that two metrics are not comparable.

As a result, we can draw a sensitivity lattice for the coverage metrics discussed above. Figure 3.1 shows this lattice. Block coverage is the least sensitive metric, compared to the rest, so it appears on the top. Immediately below is branch coverage. It is more

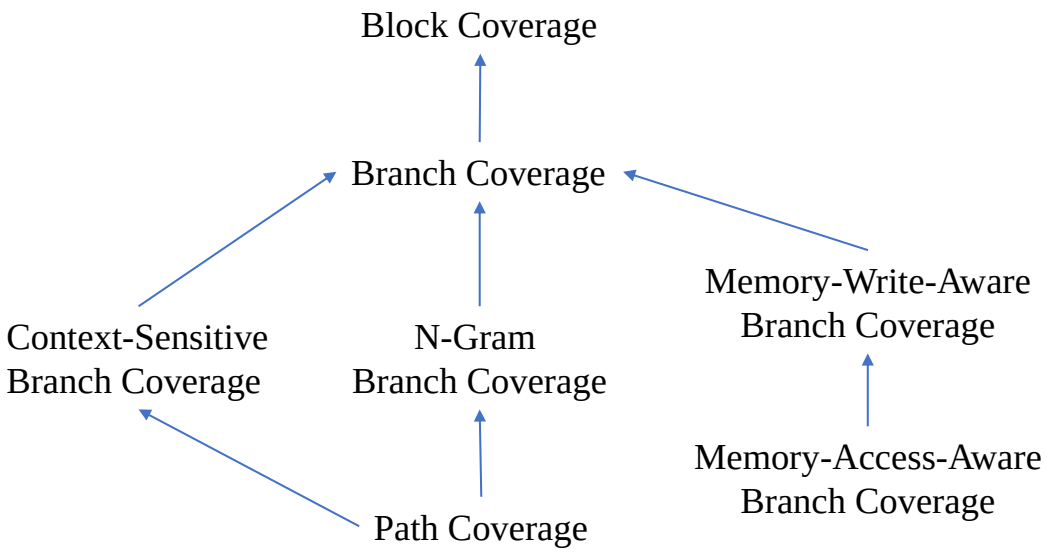


Figure 3.1: Sensitivity lattice for coverage metrics

sensitive than block coverage. Then below branch coverage are the three coverage metrics that incorporate different extra information on top of branches.

However, there is no direct comparison among these three coverage metrics, because each of them extends branch coverage in different dimensions: context-sensitive branch coverage incorporates calling context, n-gram branch coverage integrates n-1 preceding block transitions, and memory-access-aware branch coverage includes memory accesses. We can always construct a program and two inputs, such that the same coverage measurement is produced for one metric, but two different coverage measurements are produced for another.

For different values of n in n-gram branch coverage, i-gram is more sensitive than j-gram if $i > j$. Ultimately, path coverage is more sensitive than n-gram branch coverage and context-sensitive branch coverage.

Interestingly enough, we cannot compare path coverage with either memory-access-aware branch coverage or memory-write-aware branch coverage. Path coverage is not nec-

essarily more sensitive because two inputs may follow the same path but exhibit different memory access patterns.

It is noteworthy that the coverage metrics presented here are a few representative ones but are by no means complete. We hope this work can stimulate research on developing more coverage metrics and obtaining a deeper understanding of their impact.

3.3 Evaluation

To answer the research questions raised in §3.1, we implemented all the coverage metrics mentioned in §3.2 except the basic branch coverage, which is already implemented in AFL. We then conducted comprehensive experiments to evaluate the performance of different coverage metrics. Moreover, to better understand how different coverage metrics working together could affect fuzzing; we also evaluate the combination of them.

3.3.1 Implementation

In this study, since our primary goal is to fuzz binaries without source code, we choose to add our instrumentation based on user-mode QEMU. For instance, for context-sensitive branch coverage, we instrument `call` and `ret` instructions to calculate calling context, and for memory-access-aware branch coverage, we instrument memory reads and writes. For n-gram branch coverage, we use a circular buffer to store the last n-block transitions, for efficient n-gram calculation.

For convenience, in the remainder of this chapter, we use the following abbreviations to represent different metrics: `bc` represents the existing branch coverage in AFL,

`ct` represents context-sensitive branch coverage, `mw` is short for memory-write-aware branch coverage, and `ma` represents memory-access-aware branch coverage. For n-gram branch coverage, we choose to implement three versions: 2-gram, 4-gram and 8-gram, and use `n2`, `n4`, and `n8` for their abbreviations.

Furthermore, we adopted the seed scheduling of AFLFast [17] in our implementation. Since AFLFast inclines to allocate more fuzzing time on newly generated seeds, different coverage metrics will make a greater impact on fuzzing performance.

3.3.2 Experiment Setup

Dataset

We collect binaries from DARPA Cyber Grand Challenge (CGC) [27]. There are 131 binaries from CGC Qualifying Event (CQE) and 74 binaries from CGC Final Event (CFE), and thus 205 ones in total. These binaries are carefully crafted by security experts to utilize different kinds of techniques (e.g., complex I/O protocols and input checksums) and embed vulnerabilities in various ways (e.g., buffer overflow, integer overflow, and use-after-free) to comprehensively evaluate various vulnerability discovery techniques.

We also choose the LAVA-M dataset [44, 167], which consists of four GNU coreutils programs (`base64`, `md5sum`, `uniq`, and `who`) for evaluation. Each of these binaries is injected with a large number of specific vulnerabilities. As a result, we treat these injected vulnerabilities as ground truth and use them to evaluate different coverage metrics.

Table 3.1: Real-world applications used in evaluation.

Applications	**Version**	**Applications**	**Version**
objdump+binutils	2.29	readelf+binutils	2.29
strings+binutils	2.29	nm+binutils	2.29
size+binutils	2.29	file	5.32
gzip	1.8	tiffset+tiff	4.0.9
tiff2pdf+tiff	4.0.9	gif2png	2.5.11
info2cap+ncurses	6.0	jhead	3.0

In addition to the two datasets above, we also manage to collect 12 real-world applications with their latest versions (Table 3.1) and assess the performance of different coverage metrics in practice with them.

Evaluation Metrics

To answer the question of whether there is an optimal coverage metric, we propose three metrics to quantify the experimental results and evaluate the performance of the presented coverage metrics:

- **Unique Crashes**. A unique crash during fuzzing implies that a potential bug of the binary has been found. For the CGC dataset, each binary is designed to have a single vulnerability, so we did not perform any crash deduplication. For the LAVA-M dataset, each bug is assigned with a unique ID which is used for crash deduplication. For the real-world dataset, we utilize the hash of each crash's backtrace for deduplication.

- **Time to Crash**. This metric indicates how fast a given binary can be crashed by a fuzzer and is mainly for the CGC dataset. Because a CGC binary only has one

vulnerability, this metric can be used to measure the efficiency of fuzzing with different coverage metrics.

- **Seed Count**. A more sensitive coverage metric is more likely to convert a testcase into a seed, and thus the number of unique seeds may be larger. Therefore, this metric quantifies the sensitivity of each coverage metric in a practical sense.

Computing Resources

Our experiments are conducted on a private cluster consisting of a pool of virtual machines. Each virtual machine has a Ubuntu 14.04.1 operating system equipped with 2.3 GHz Intel Xeon processor (24 cores) and 30GB of RAM. As fuzzing is a random process, we followed the recommendations from [88] and performed each evaluation several times for a sufficiently long period.

Settings

The tests are mainly focused on the CGC dataset. Specifically, each coverage metric is tested with every binary of the CGC dataset in the dataset using two fuzzing instances for 6 hours (i.e., similar to one instance running 12 hours). We chose this fuzzing time because almost all of the bugs found by fuzzer in CQE and CFE were reported within the first six hours. Moreover, in order to take the randomness of fuzzing into account, each test is performed ten times. The total evaluation time is around 60 days. For binaries with initial sample inputs, we utilized them as initial seeds; otherwise, we used an empty seed.

For the LAVA-M dataset, we tested each coverage metric separately for 24 hours and three times. We used the seed inputs provided by this benchmark and dictionaries of

constants extracted from the binary as suggested in [169]. For the real-world dataset, we tested each coverage metric for 48 hours, with two fuzzing instances, and for six times. We used the example inputs from AFL as seeds whenever possible; otherwise with an empty seed.

3.3.3 Study Findings

Comparison of Unique Crashes

We first compare the number of unique crashes that have been found with different coverage metrics.

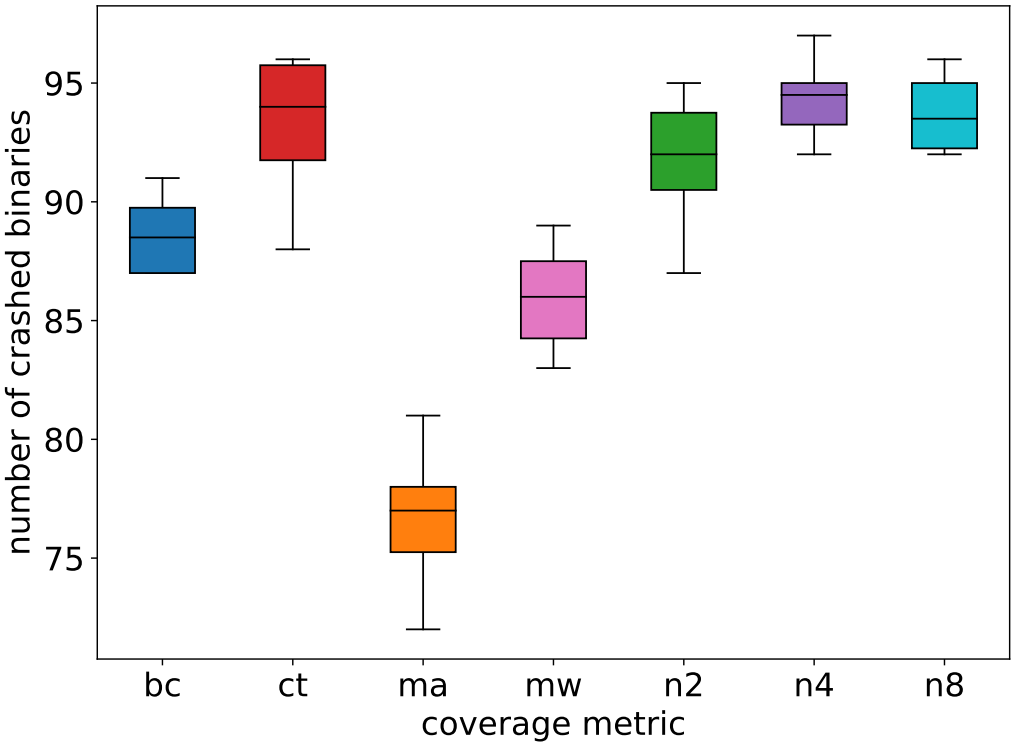


Figure 3.2: Number of crashed CGC binaries.

CGC Dataset Figure 3.2 summarizes the number of crashed CGC binaries for each coverage metric across ten rounds of trials. Note that because each binary only has one vulnerability, this number is equivalent to the total number of unique crashes. Overall, the baseline metric `bc` crashed about 89 binaries on average and 91 binaries at most. Except for `ma` and `mw`, all other more sensitive coverage metrics (`ct, n2, n4, n8`) outperform `bc`. This result is encouraging: sensitivity does play an important role in finding crashes. However, as demonstrated by `mw` and `ma`, too much sensitivity could also have a negative impact on fuzzing performance. The reason is, more sensitive metrics will select more test cases as seeds (§3.3.3); when the time budget is limited, each seed will get less time to mutate or not get scheduled at all.

Table 3.2: Pairwise comparisons (row vs. column) of uniquely crashed CGC binaries.

	bc	ct	ma	mw	n2	n4	n8	others
bc	0/0	**0/6**	**0/15**	**0/11**	**0/6**	**0/6**	**0/5**	**0/2**
ct	**9/13**	0/0	9/23	10/15	6/12	3/6	4/8	1/3
ma	**2/3**	3/4	0/0	2/3	4/6	4/5	2/3	1/1
mw	**6/8**	2/5	0/12	0/0	3/8	2/7	3/5	0/2
n2	**4/4**	0/3	7/16	4/9	0/0	0/2	0/2	0/0
n4	**9/12**	3/5	12/23	8/16	8/10	0/0	0/5	0/1
n8	**9/10**	6/6	13/20	10/13	7/9	2/4	0/0	0/0
all	**19/21**	10/14	20/33	19/24	18/23	11/15	9/16	110

Next, we investigated each coverage metric's ability to trigger individual bug/crash – is there any bug that is only triggered by one or a subset of the evaluated metrics but not the rest? To answer this question, we conducted a pairwise comparison on crashed binaries (Table 3.2). For each pair of coverage metrics i (in the row) and j (in the column), we first count the number of binaries that were only crashed by i but not by j, denoted as the number after the "/". Since such differences could be caused by randomness, we conducted a second experiment focusing on the impact of sensitivity. Specifically, during fuzzing, we

recorded the chain of seeds that led to each crashing test case. Each chain starts with the initial seed and ends with the crashing test case. Afterward, for each pair of coverage metrics (i, j), we checked whether each seed along the chain selected by i would also be selected by j as seed, without any additional mutation (i.e., fuzzing). In this process, we also discarded additional sensitivity (non-binary `hit_count`) and insensitivity (key collision) introduced in implementation. The result is denoted as the number before the "/" in each cell of Table 3.2. For example, entry (`ct`, `bc`) indicates that there were 13 binaries crashed by `ct` but not by `bc`, within which 9 crashes have at least one seed along the crashing chains that will be dropped by `bc`. Similarly, entry (`bc`, `ct`) indicates that 6 binaries crashed by `bc` are not crashed by `ct`, of which however none of the seeds along the crashing chain will be dropped by `ct`. Besides, for a metric k, entry (`all`, k) indicates the number of binaries crashed by at least one of the other coverage metrics but not by k and entry (k, `others`) indicates the number of binaries only crashed by k but not by any other coverage metrics. Finally, entry (`all`, `others`) indicates the number of binaries crashed by at least one of all the seven coverage metrics.

We can see that the difference between any two coverage metrics is considerable. More importantly, there is no single winner that beats everyone else. Even for `ma`, although it crashes the smallest amount of binaries in total, it contributes 2 unique crashed binaries beyond `bc`, and 3, 2, 4, 4, and 2 unique crashed binaries beyond `ct`, `mw`, `n2`, `n4`, and `n8` respectively, of which the crashes have at least one seed along the crashing chains that will be dropped by the other metric. In other words, every coverage metric can make its own

and unique contribution. This observation further motivates us to study the combination of different coverage metrics. We will discuss more in §3.3.3.

Table 3.3: Number of unique bugs found by different coverage metrics on the LAVA-M dataset

	bc	ct	ma	mw	n2	n4	n8	Listed
base64	45	45	44	45	45	45	45	44
md5sum	54	58	35	43	59	58	51	57
uniq	29	29	29	20	29	29	29	28
who	261	255	301	231	166	159	299	2136

LAVA-M Dataset Table 3.3 summarizes the bugs found on LAVA-M dataset by different coverage metrics, while the last column represents the number of bugs listed by LAVA authors. Compare to the CGC dataset, the LAVA-M dataset is not very suitable for our goal. In particular, most injected bugs are protected by a magic number, which is very hard to be solved by random mutation and cannot reflect unique abilities of different coverage metrics. Although we have followed the suggestions from [169] and used dictionaries of constant (magic) numbers extracted from the binary, we still cannot rule out the differences caused by not being able to solve the magic number. For binary `base64`, `md5sum`, and `uniq`, the difference between different coverage metrics is small, except for `ma` in `md5sum` and `mw` in `uniq`. For binary `who`, it is surprising that in addition to `n8`, `ma` also finds much more unique bugs than `bc` and other three metrics, despite its poor performance on the CGC dataset.

Real-World Dataset There are many crashes found for binaries in the real-world dataset. We use the open-source tool `afl-collect` [174] to de-duplicate these crashes and identify unique crashes. Overall, we have successfully found unique bugs in 5 real-world binaries as

Table 3.4: Number of unique crashes found by different coverage metrics in the real-world dataset.

	bc	ct	ma	mw	n2	n4	n8
gif2png	4	4	3	4	5	4	4
info2cap	1446	1063	481	99	568	933	943
objdump	–	–	–	–	1	1	–
size	–	1	–	–	1	1	1
nm	–	1	–	1	–	–	1

listed in Table 3.4. It is worth noting that for binary `objdump`, `size`, and `nm`, only our newly proposed coverage metrics find unique bugs.

Comparison of Time to Crash

CGC Dataset Since most CGC binaries only contain one bug, we then measure the time to first crash (TFC) for different coverage metrics across the ten rounds of trials. The accumulated number within a 95% confidence of binaries crashed over time is shown in Figure 3.3, where the x-axis presents in seconds and the y-axis shows the number of binaries whose TFC (time-to-first-crash) were within that time. The x-axis presents time in seconds while the y-axis shows the accumulated number of binaries crashed. For example, we can see that `n4` almost manages to crash more binaries than other coverage metrics in the first hour (3600 seconds) and `ma` performs the worst among them. We also see that all of the proposed coverage metrics other than `ma` and `mw` can help find crashes in binaries more quickly than the original AFL (`bc`). Moreover, although `n4` does not find the most crashes, it is the best one during the early stage (30 to 90 minutes). After 90 minutes, `ct` surpasses it and becomes one of the best performers.

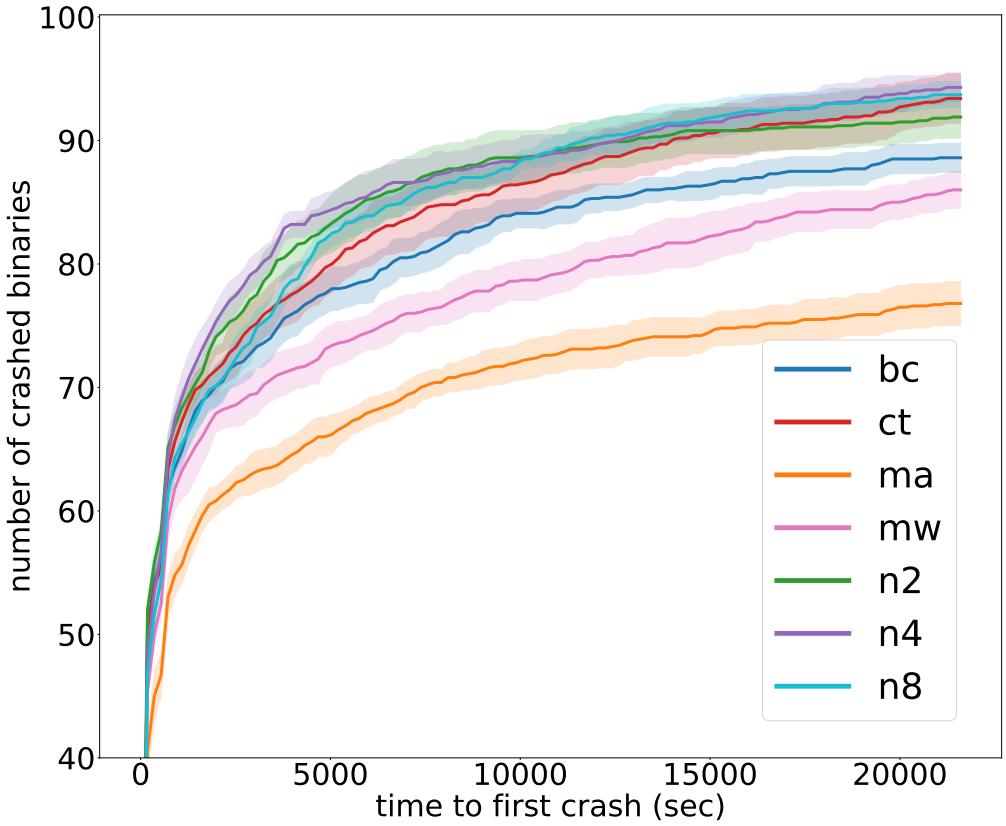


Figure 3.3: Number of binaries crashed over time during fuzzing on the CGC dataset.

LAVA-M Dataset Figure 3.4 presents the number of unique bugs found over time by different coverage metrics on the four binaries. We can see that the newly proposed coverage metrics outperform `bc` on all four binaries. Although `ma` is slower than others, it finally finds the same number of unique bugs on binary `base64` and `unique`. On binary `who`, `ma` even finds quite more unique bugs. Moreover, `ct` and `n8` perform stably well across four binaries, and the latter one performs extremely well on binary `who`: it finds the largest number of unique bugs and much faster than the rest.

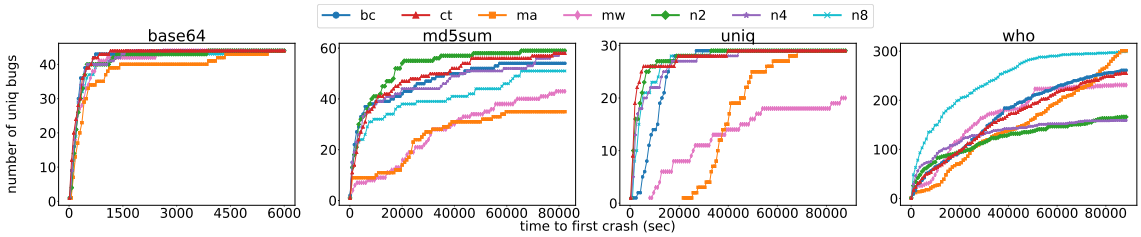


Figure 3.4: Number of unique bugs found over time during fuzzing on the LAVA-M dataset.

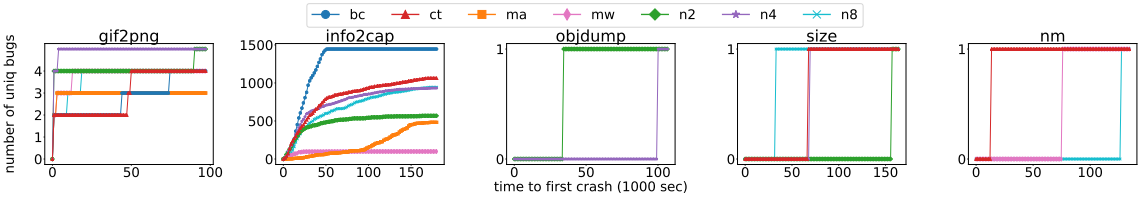


Figure 3.5: Number of unique crashes found over time on real-world dataset.

Real-World Dataset Similarly, Figure 3.5, where the x-axis presents TFC in 1000 seconds, shows The number of unique bugs over time found by different coverage metrics on the five crashed binaries in the real-world dataset. We can see that except for `info2cap`, `bc` either finds unique bugs much more slowly than others or does not find any bugs at all. In addition, there is no global trend about which coverage metric is the fastest one to find bugs across the five binaries.

Comparison of Seed Count

CGC Dataset We collect the number of seeds selected for each binary using different coverage metrics and report the mean number within a 95% confidence among the ten runs. Figure 3.6 displays the cumulative distribution of the numbers of generated seeds. A curve closer to the top left in the figure implies that in general fewer seeds are generated for binaries with the corresponding coverage metric.

We had several observations from the result. First, `ma` was significantly more sensitive than the rest coverage metrics. It selects several orders of magnitude more seeds than the others. While most of these seeds are stepping stones for more meaningful mutations that lead to final crashes, too many of them would hurt the fuzzing performance because the differences among most of the seeds are so tiny that they are unlikely to result in any new bug. Second, for n-gram branch coverage, as n increases from 1 (`bc`) to 8, the number of seeds increases correspondingly, although the lines for `bc` and `n2` are too close to each other. This phenomenon meets our expectation, as $n8 \succ n4 \succ n2 \succ bc$. Third, while in theory, we cannot compare `ct` with n-gram regarding their sensitivities, we observe that the seed count distribution for `ct` is between `n4` and `n8`, at least for the CGC dataset. Fourth, in theory, $ma \succ mw \succ bc$. We indeed observe these relations in the form of seed counts for `ma`, `mw`, and `bc`.

Table 3.5: The numbers of seeds generated by different coverage metrics on the LAVA-M dataset.

	bc	ct	ma	mw	n2	n4	n8
base64	208	170	16372	200	196	273	425
md5sum	706	497	75323	71131	474	719	4958
uniq	104	52	43928	50178	77	92	153
who	223	144	14183	16511	190	271	470

LAVA-M Dataset Table 3.5 lists seed counts generated by each coverage metric on the four binaries in the LAVA-M dataset. We can see that the observations for the CGC dataset still hold in general, with some outliers. For instance, the seed counts of `ct` on all four

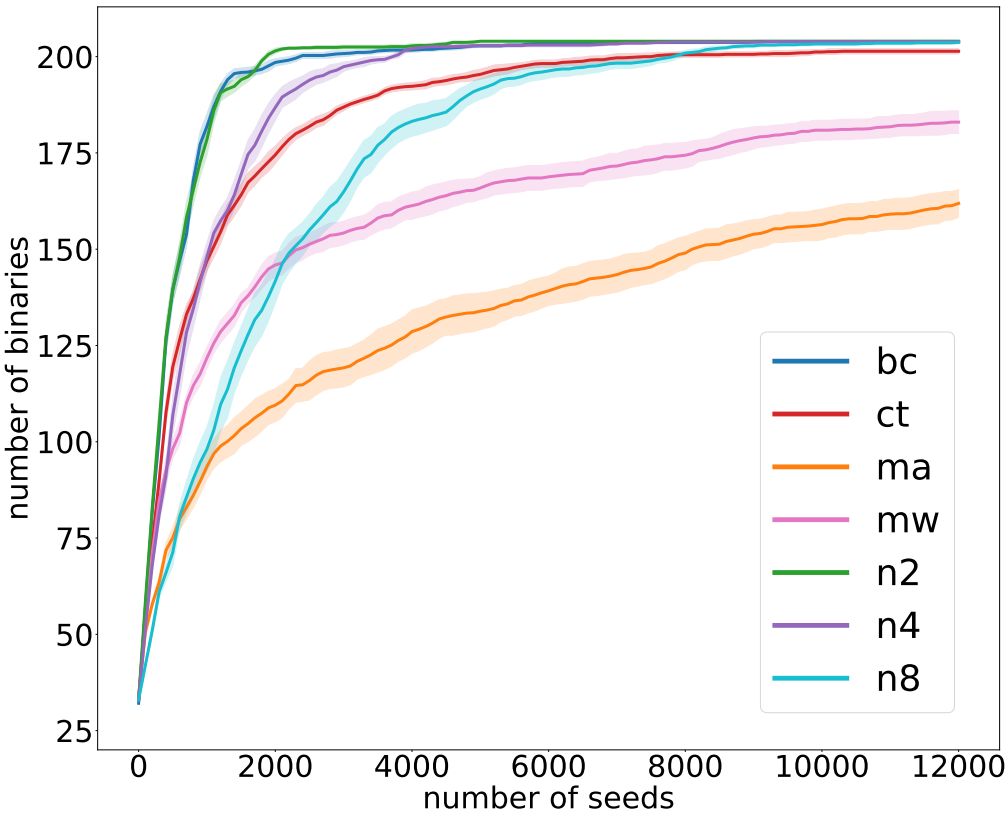


Figure 3.6: Partial CDFs of seeds generated by different coverage metrics on the CGC dataset.

binaries are smaller than those of `bc`. These numbers are not statistically significant, given such a small-scale dataset.

Real-World Dataset Table 3.6 lists seed counts generated by each coverage metric on the 12 real-world binaries. We can draw similar observations as on the CGC and LAVA-M datasets with some exceptions: the seed count distribution for `ct` is no longer between `n4` and `n8` in general.

Table 3.6: The numbers of seeds generated by different coverage metrics on the real-world dataset.

	bc	ct	ma	mw	n2	n4	n8
file	38	38	38	19462	38	38	38
gif2png	1039	2037	151008	29606	804	1665	3840
gzip	1305	1340	124253	65035	1002	1875	5446
info2cap	4966	12555	76048	30136	4802	8831	17104
objdump	6015	42625	49401	126578	4978	8756	22914
readelf	8461	15317	91982	63009	8758	15425	35429
strings	61	62	1619	59	69	68	131
tiff2pdf	834	883	143902	2841	724	1108	2395
tiffset	2	2	2	2	2	2	2
size	2117	4860	111978	143693	1605	3003	10278
nm	12566	49307	133460	73386	5947	10174	23322
jhead	384	284	75328	29229	362	576	1376

Combination of Coverage Metrics

From the evaluation results above, we observe that each coverage metric has its unique characteristics in terms of crashes found and crashing times. This observation leads us to wonder whether combining fuzzers with different coverage metrics together would find more crashes and find them faster. To answer this question, we consider two options for combination: (1) fuzzers with different coverage metrics are running in parallel and synchronizing seeds across all metrics periodically (i.e., cross-seeding); and (2) fuzzers with different coverage metrics are running in parallel but independently, as the baseline to show whether cross-seeding really helps.

To study these two options, we create three configurations of 14 fuzzing instances: (a) all 14 fuzzing instances with `bc` and seed synchronization; (b) 2 fuzzing instances for each of the 7 different coverage metrics with seed synchronization only within the same metric; and (c) 2 fuzzers for each of the 7 different coverage metrics with seed synchronization across all metrics (i.e., cross-seeding).

CGC Dataset We run the three configurations each for six hours, and for three times to get median results on the CGC dataset. Figure 3.7 illustrates the number of binaries crashed over time for the three configurations. We can make the following observations. First, both combination options outperform the baseline by large margins, with respect to both the number of crashed binaries and crash times. The combination without cross-seeding (configuration b) crashes 78 CQE binaries, 31 CFE binaries, and 109 binaries in total. The one with cross-seeding (configuration c) crashes 77 CQE binaries, 33 CFE binaries, and 110 in total. Meanwhile, the baseline only crashes 64 CQE binaries, 30 CFE binaries, and 94 in total. It is a notable achievement: the hybrid fuzzer Driller [149] was able to crash 77 CQE binaries after 24 hours with the help of concolic execution, where each binary is assigned to four fuzzing instances and all binaries share a pool of 64 CPU cores for concolic execution, using totally 12,640 CPU hours ($131\ binaries \times 4\ cores \times 24\ hours + 60\ cores \times 24\ hours$). Compared with Driller, *we can achieve the same or even better results by pure fuzzing with less computing resources (*$131\ binaries \times 14\ cores \times 6\ hours = 11,004$ *CPU hours totally)!*

Second, the blue line and the red line cross at around 3 hours. At this cross point, 105 binaries have been crashed for both configurations. It implies that the combination with cross-seeding is able to crash 105 binaries much earlier than the one without cross-seeding.

LAVA-M and Real-World Datasets We also run the three configurations each for 24 hours on LAVA-M dataset, and each for 48 hours on the real-world dataset. Figure 3.8 and Figure 3.9 present the results. We observed that the combination without cross-seeding always outperforms the baseline (14 fuzzers with `bc` only) by large margins. On the other

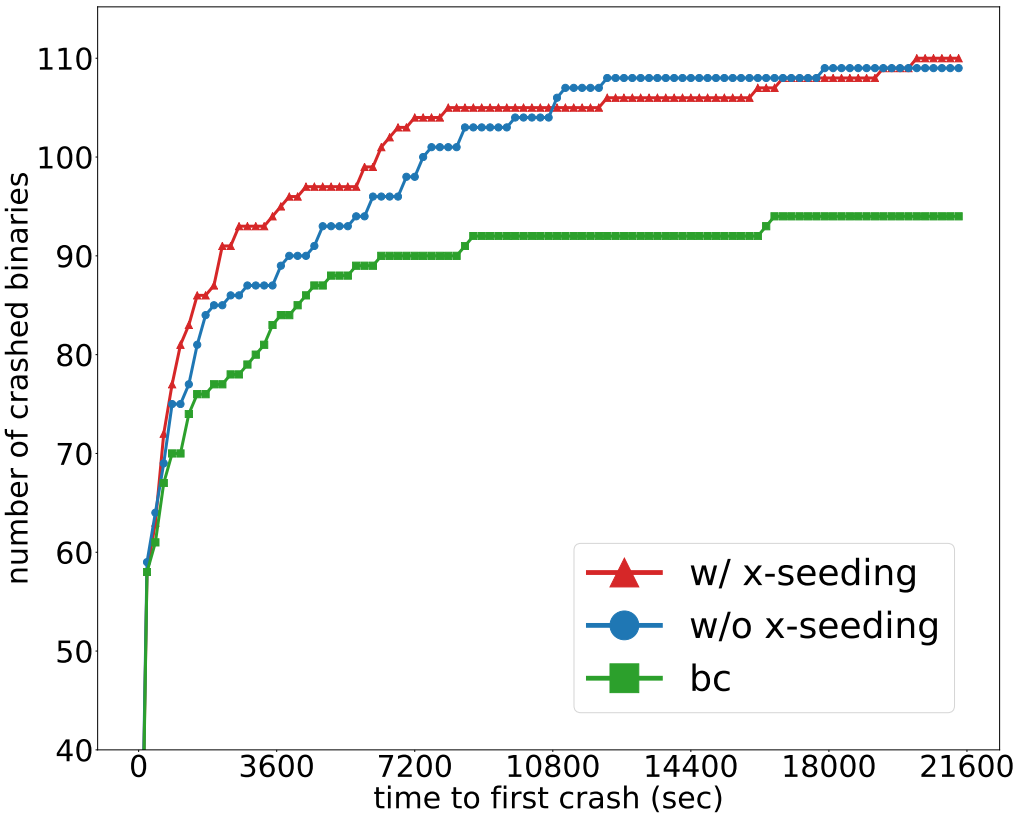


Figure 3.7: Number of binaries crashed during fuzzing tests by combining different coverage metrics on the CGC dataset.

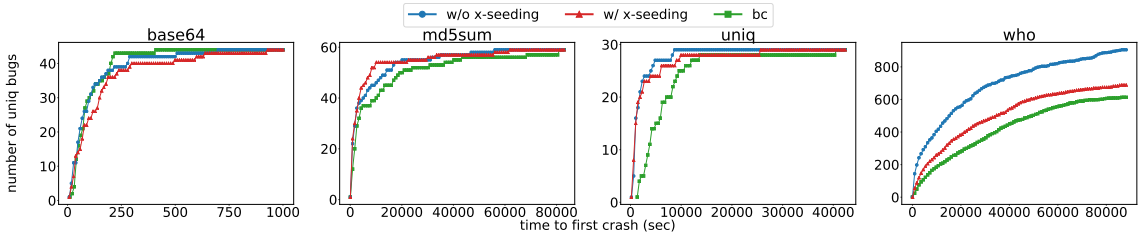


Figure 3.8: Number of unique bugs found over time by combining different coverage metrics on the LAVA-M dataset.

hand, the combination with cross-seeding has inconsistent performance across these nine binaries. In some cases, it is even worse than the baseline. Unlike the result for the CGC dataset, this result is not statistically significant. However, it does indicate that sometimes, the overhead of cross-seeding may outweigh its benefits. Xu et al. [180] have shown that

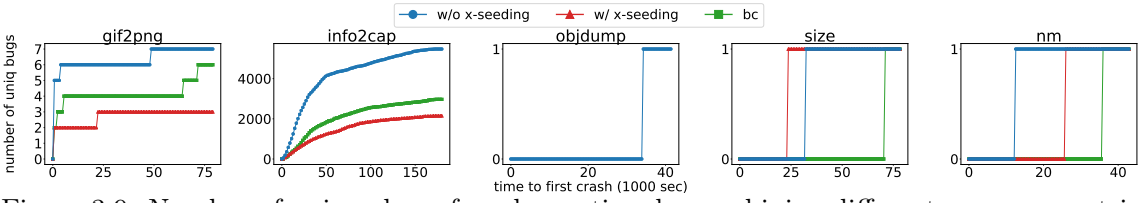


Figure 3.9: Number of unique bugs found over time by combining different coverage metrics for crashed real-world binaries.

cross-seeding overhead is significant in parallel fuzzing and propose OS-level modifications for improving fuzzing performance. It would be interesting to re-evaluate the performance of the combination with cross-seeding with these OS-level modifications. We leave it as future work.

In summary, *it is better to combine different coverage metrics with or without cross-seeding, which can help find more bugs and find them faster.*

3.4 Summary

In this chapter, we present the first systematic study on the impact of coverage metrics on greybox fuzzing with the DARPA CGC dataset, the LAVA-M dataset, and real-world binaries. To this end, we formally define the concept of sensitivity when comparing two coverage metrics, and selectively discuss several metrics that have different sensitivities. Our study has revealed that each coverage metric leads to find different sets of vulnerabilities, indicating there is no grand slam that can beat others. We also showed a combination of different metrics helps find more crashes and find them faster. We hope our study would stimulate research on developing more coverage metrics for greybox fuzzing.

Chapter 4

Reinforcement Learning-Based Hierarchical Seed Scheduling for Greybox Fuzzing

4.1 Introduction

Greybox fuzzing is a state-of-the-art testing technique that has been widely adopted by the industry and has successfully found tens of thousands of vulnerabilities in widely used software. For example, the OSS-Fuzz [68] project has found more than 10,000 bugs in popular open-sourced projects like OpenSSL since its launch in December 2016.

Greybox fuzzing can be modeled as a genetic process where new inputs are generated through mutation and crossover/splice. The generated inputs are selected according to

a fitness function. Selected inputs are then added back to the seed pool for future mutation. Unlike natural evolution, due to the limited processing capability, only a few inputs from the seed pool will be scheduled to generate the next batch of inputs. For example, a single fuzzer instance can only schedule one seed at a time.

The most common fitness function used by off-the-shelf fuzzers like American Fuzzy Lop (AFL) [188] is edge coverage, i.e., inputs that cover new branch(es) will be added to the seed pool, as its goal is to achieve higher edge coverage of the code. While most fuzzers are coverage-guided (i.e., use new coverage as the fitness function), recent research has shown that the genetic process can also be used to discover a diversity of program properties by using a variety of fitness functions [92, 120, 127].

An important property of a fitness function (e.g., a coverage metric) is its ability to preserve intermediate waypoints [120]. To better illustrate this, consider flipping a magic number check `a = 0xdeadbeef` as an example. If a fuzzer only considers edge coverage, then the probability of generating the correct `a` with random mutations is 2^{32}. However, if the fuzzer can preserve important waypoints, e.g., by breaking the 32-bit magic number into four 8-bit number [89], then solving this checking will be much more efficient since the answer can be generated from a sequence as `0xef`, `0xbeef`, `0xadbeef`, and `0xdeadbeef`. This check can also be solved faster by understanding distances between current value of `a` and the target value [32, 33, 38, 54, 150]. More importantly, recent research has shown that many program states *cannot* be reached without saving critical waypoints [108, 156].

We have formalized the ability to preserve intermediate waypoints as the *sensitivity* of a coverage metric [156]. Conceptually, a more sensitive metric would lead to more program

states (e.g., code coverage). However, the empirical evaluation of [156] shows that this is not always the case. The reason is that, a more sensitive coverage metric will select more seeds, which could cause seed explosion and exceed the fuzzer's ability to schedule. As a result, many seeds may never be scheduled or be scheduled without enough time/power to make a breakthrough [17].

In this work, we aim to address the seed explosion problem with a *hierarchical scheduler*. Specifically, fuzzing can be modeled as a multi-armed bandit (MAB) problem [177], where the scheduler needs to balance between *exploration* and *exploitation*. With a more sensitive coverage metric like branch distance, exploitation can be considered as focusing on solving a hard branch (e.g., magic number check), and exploration can be considered as exercising an entirely different function. Our crucial observation is that when a coverage metric C_j is more sensitive than C_i, we can use C_j to save all the intermediate waypoints without losing the ability to discover more program states; but at the same time, we can use C_i to cluster seeds into a representative node and schedule at node level to achieve better exploration. More specifically, the scheduler will choose a node first, and then choose a seed in that node. Based on this observation, we propose to organize the seed pool as a multi-level tree where leaf nodes are real seeds and internal nodes are less sensitive coverage measurements. The closer a node is to the leaf, the more sensitive the corresponding coverage measurement is. Then we can utilize the existing MAB algorithms to further balance between exploitation and exploration.

To validate our idea, we implemented two prototypes: one AFL-HIER based on AFL and the other AFL++-HIER based on AFL++. We performed extensive evaluation

on the DARPA Cyber Grand Challenge (CGC) dataset [27] and Google FuzzBench [70] benchmarks. Compared to AflFast [17], Afl-Hier can find more bugs in CGC (77 vs. 61). Afl-Hier also achieved better coverage in about 83 of 180 challenges and the same coverage on 60 challenges. More importantly, Afl-Hier can find the same amount of bugs and achieve the same coverage faster than AflFast. On FuzzBench, Afl++-Hier achieved higher coverage on 10 out of 20 projects than Afl++ (Qemu).

Contributions. This works makes the following contributions:

- We propose multi-level coverage metrics that bring a novel approach to incorporate sensitive coverage metrics in greybox fuzzing.
- We design a hierarchical seed scheduling algorithm to support the multi-level coverage metric based on the multi-armed bandits model.
- We implement our approach as an extension to Afl and Afl++ and release the source code at `https://github.com/bitsecurerlab/aflplusplus-hier`.
- We evaluate our prototypes on DARPA CGC and Google FuzzBench. The results show that our approach not only can trigger more bugs and achieve higher code coverage, but also can achieve the same coverage faster than existing approaches.

4.2 Multi-Level Coverage Metrics

In this section, we discuss what are multi-level coverage metrics and why they are useful for greybox fuzzing.

4.2.1 Sensitivity of Coverage Metrics

Given a mutation-based greybox fuzzer, a fuzzing campaign starts with a set of initial seeds. As the fuzzing goes on, more seeds are added into the seed pool through mutating the existing seeds. By tracking the evolution of the seed pool, we can see how each seed can be traced back to an initial seed via a mutation chain, in which each seed is generated from mutating its immediate predecessor. If we consider a bug triggering test case as the *end* of a chain and the corresponding initial seed as the *start*, those internal seeds between them serve as waypoints that allow the fuzzer to gradually reduce the search space to find the bug [120].

The coverage metric used by a fuzzer plays a vital role in creating such chains, from two main aspects. First, if the chain terminates earlier before reaching the bug triggering test case, then the bug may never be discovered by the fuzzer. Our work [156] formally model this ability to preserve critical waypoints in seed chains as the *sensitivity* of a coverage metric. For example, consider the maze game in Listing 4.1, which is widely used to demonstrate the capability of symbolic execution of exploring program states. In this game, a player needs to navigate the maze via the pair of (x, y) that determines a location for each step. In order to win the game, a fuzzer has to try as many sequences of (x, y) pairs as possible to find the right route from the starting location to the crashing location. This simple program is very challenging for fuzzers using edge coverage as the fitness function, because there are only four branches related to every pair of (x, y), each checking against a relatively simple condition that can be satisfied quite easily. For instance, five different inputs: "`a`," "`u`," "`d`," "`l`," and "`r`" are enough to cover all branches/cases of the `switch` statement. After this,

```
1  char maze[7][11] = {
2          "+-+---+---+",
3          "| |       |#|",
4          "| | --+ | | |",
5          "| |   | | | |",
6          "| +-- | | | |",
7          "|     |   |  |",
8          "+-----+---+"};
9  int x = 1, y = 1;
10 for(int i = 0; i < MAX_STEPS; i++){
11     switch(steps[i]){
12         case 'u': y--; break;
13         case 'd': y++; break;
14         case 'l': x--; break;
15         case 'r': x++; break;
16         default:
17             printf("Bad step!"); return 1;
18     }
19     if (maze[y][x] == '#'){
20         printf("You win!");
21         return 0;
22     }
23     if (maze[y][x] != ' '){
24         printf("You lose.");
25         return 1;
26     }
27 }
28 return 1;
```

Listing 4.1: A Simple Maze Game

even if the fuzzer can generate new interesting inputs that indeed advance the program's state towards the goal (e.g., "`dd`"), these inputs will *not be selected as new seeds* because they do not provide new edge coverage. As a result, it is extremely hard, if not impossible, for fuzzers that use the edge coverage to win the game [5].

On the contrary, as we will show in §4.4.3, if a fuzzer can measure the different combinations of x and y (e.g., by tracking different memory accesses via $*(maze + y + x)$ at line 10), then reaching the winning point will be much easier [5, 156]. Similarly, researchers

have also observed that the orderless of branch coverage and hash collisions can cause a fuzzer to drop critical waypoints hence prevent certain code/bugs from being discovered [55,93,108].

The second impact of a coverage metric has on creating seed chains is the *stride* between a pair of seeds in a chain. Specifically, the sensitivity of a coverage metric also determines how likely (i.e., the probability) a newly generated test case will be saved as a new seed. For instance, it is easier for a fuzzer that uses edge coverage to discover a new seed than a fuzzer that uses block coverage. Similarly, as we have discussed in §4.1, it is much easier to find a match for an 8-bit integer than a 32-bit integer. Böhme et al. [17] model the minimum effort to discover a neighbouring seed as the required *power* (i.e., mutations). Based on this modeling, a more sensitive coverage metric requires less power to make progress, i.e., a shorter stride between two seeds. Although each seed only carries a small step of progress, the accumulation of them can narrow the search space faster.

While the above discussion seems to suggest that a more sensitive coverage metric would allow fuzzers to detect more bugs, the empirical results from [156] showed this is not always the case. For instance, while memory access coverage would allow a fuzzer to win the maze game (Listing 4.1), it did not perform very well on many of the DARPA CGC challenges. The reason is that, a more sensitive coverage metric will also create a larger seed pool. As a result, the seed scheduler needs to examine more candidates each time when choosing the next seed to fuzz. In addition to the increased workload of the scheduler, a larger seed pool also increases the difficulty of seed exploration, i.e., trying as many fresh seeds as possible. Since the time of a fuzzing campaign is fixed, more abundant seeds also imply that the average fuzzing time of each seed could be decreased, which could

negatively affect seed exploitation, i.e., not fuzzing interesting seeds enough time to find critical waypoints.

Overall, a more sensitive coverage metric boosts the capability (i.e., upper bound) of a fuzzer to explore deeper program states. Nevertheless, in order to effectively utilize its power and mitigate the side effects of the resulting excessive seeds, the coverage metric and the corresponding seed scheduler should be carefully crafted to strike a balance between exploration and exploitation.

4.2.2 Seed Clustering via Multi-Level Coverage Metrics

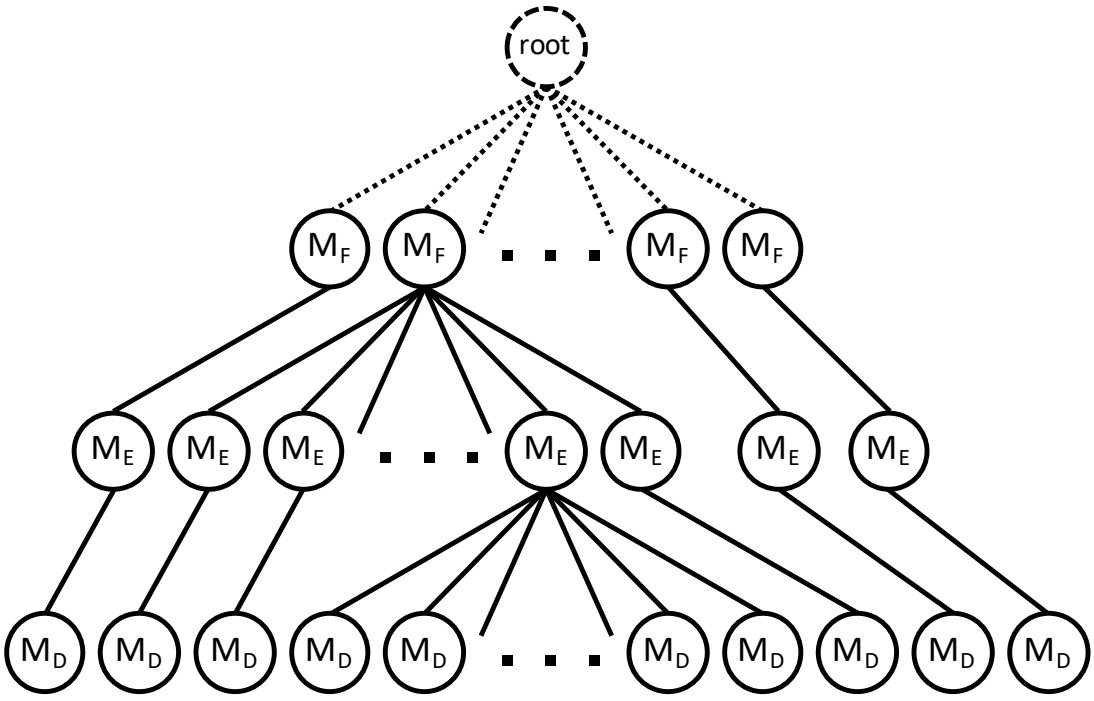


Figure 4.1: A multi-level coverage metric that measures function coverage at top-level, edge coverage at mid-level, and hamming distance of comparison operands at leaf-level. The root node is a virtual node only used by the scheduler.

The similarity and diversity of seeds, which can be measured in terms of the exercised coverage, drive the seed exploration and exploitation in a fuzzing campaign. In general, a set of similar seeds gains less information about the program under test than a set of diverse seeds. When a coverage metric measures more fine-grained coverage information (e.g., edge), it can dim the coarse-grained diversity (e.g., block) among different seeds. First, it encourages smaller variances between seeds. Second, it loses the awareness of the potential larger variance between seeds that can be detected by a more coarse-grained metric. For instance, a metric measuring edge coverage is unaware of whether two seeds exercise two different sets of basic blocks or the same set of basic blocks but through different edges. Therefore, it is necessary to illuminate seed similarity and diversity when using a more sensitive coverage metric.

Clustering is a technique commonly used in data analysis to group a set of similar objects. Objects in the same cluster are more similar to each other than to those in a different cluster. Inspired by this technique, we propose to perform seed clustering so that seeds in the same cluster are similar while seeds in different clusters are more diverse. In other words, these clusters offer another perspective that allows a scheduler to zoom in the similarity and diversity among seeds.

Based on the observation that the sensitivity of most coverage metrics for greybox fuzzing can be directly compared (i.e., the more sensitive coverage metric can subsume the less sensitive one), we propose an intuitive way to cluster seeds—using a coarse-grained coverage measurement to cluster seeds selected by a fine-grained metric. That is, seeds in the same cluster will have the same coarse-grained coverage measurement. Moreover, we can

use more than one level of clustering to provide more abstraction at the top level and more fidelity at the bottom level. To this end, the coverage metric should allow the co-existence of multiple coverage measurements. We name such a coverage metric a *multi-level coverage metric*.

4.2.3 Incremental Seed Clustering

With the multi-level coverage metric in place, if a test case is assessed as exercising a new coverage (feature) by any of the measurements, it will be retained as a new seed and put in a proper cluster as described in algorithm 2. Generally, except for the top-level measurement M_1 that directly classifies all seeds into different clusters, the following lower-level measurement M_i $(i = 2, \cdots, n)$ works on each of the clusters generated by M_{i-1} separately, classifying seeds in it into smaller sub-clusters, which is named incremental seed clustering.

In more detail, given a multi-level coverage metric as shown in Figure 4.1 , a test case exercising any new function, edge, or distance coverage will be assessed as a new seed. Then the root node starts the seed clustering. It will find from its child nodes an existing M_F node that covers the same functions as the new seed, or create a new M_F node if the desired node does not exist. Next, the seed clustering continues in a similar way that puts the new seed into a M_E node with the same edge coverage. Finally, a child M_D node of the M_E node is selected to save the new seed according to its distance coverage.

Algorithm 2: Seed Selection Algorithm

Input: test case I

Output: return a status code indicating whether I triggers a bug or covers new features

Data: program being fuzzed P,
existing seed set S^*,
existing feature set M^*,
current working cluster cc,
map from feature sets to sub clusters $cc.map$,
coverage metric $C^n \sim \langle C_1, \ldots, C_n \rangle$,
coverage measurements $\langle M_1, \cdots, M_n \rangle$

Result: put I in a proper cluster if it is a new seed

```
1  Function RunAndEval(I):
2      ⟨M_1, ..., M_n⟩ ← RunWithInstrument(P, I, C^n)
3      if bug triggered then
4          return Bug
5      end
6      M^t ← M_1 ∪ ··· ∪ M_n
7      if M^t ⊆ M^* then
8          return Known
9      else
10         M^* ← M^* ∪ M^t
11         foreach i ∈ {1, ..., n} do
12             next_cc ← cc.map[M_i]
13             if next_cc = NULL then
14                 next_cc ← new_cluster()
15             end
16             move I into next_cc
17             cc.map[M_i] ← next_cc
18             cc ← next_cc
19         end
20         return NewCovExplored
21     end
22 End
```

Terms used in the algorithm are defined as follows.

Definition 4.2.1. *A coverage space* Γ *defines the set of enumerable features we pay attention to that can be covered by executing a program.*

Some typical coverage spaces are:

- Γ_F is the set of all program functions.
- Γ_B is the set of all program blocks.
- Γ_E is the set of all program edges. Note that an edge is a transition from one block to the next.

It is worth mentioning that in real-world fuzzers such as AFL, the coverage information is recorded via well-crafted `hit_count` maps. Consequently, the features are signified by entries of the maps.

Definition 4.2.2. *A coverage metric* $C : (\mathcal{P} \times \mathcal{I}) \to \Gamma^*$ *measures the execution of a program* $P \in \mathcal{P}$ *with an input* $I \in \mathcal{I}$*, and produces a set of features that are exercised by it at least once, denoted as* $M \in \Gamma^*$ *.*

Since coverage metric is mainly characterized by the coverage space Γ, it can be simplified with the coverage space. Some typical coverage metrics are:

- C_F measures the functions that are exercised by an execution.
- C_B measures the blocks that are exercised by an execution.
- C_E measures the edges that are exercised by an execution.

Finally, we give the definition of a multi-level coverage metric.

Definition 4.2.3. *A coverage metric* $C^n : (\mathcal{P} \times \mathcal{I}) \to \langle \Gamma_1^*, \ldots, \Gamma_n^* \rangle$ *consists of a sequence of coverage metrics* $\langle C_1, \ldots, C_n \rangle$*. It measures the execution of a program* $P \in \mathcal{P}$ *with an input* $I \in \mathcal{I}$*, and produces a sequence of measurements* $\langle M_1, \ldots, M_n \rangle$*.*

A multi-level coverage metric combines multiple metrics at different levels to assess a seed. As a result, it relies on lower-level coverage measurements to preserve minor variances among seeds so that there will be more abundant seeds in a chain. This helps to reduce the search space of finding bug triggering test cases. Meanwhile, it allows a scheduler to use upper-level measurements to detect major differences among seeds. Note that when $n = 1$, it is reduced to a traditional single level coverage metric.

4.2.4 Principles and Examples of Multi-level Coverage Metrics

To further illustrate how a multi-level coverage metric works, we propose some representative examples. We first discuss some principles for developing an effective multi-level coverage metric $C^n \sim \langle C_1, \ldots, C_n \rangle$ for fuzzing a program P.

Principles

Through the incremental seed clustering, all seeds are put into a hierarchical tree that lays the foundation of our hierarchical seed scheduling algorithm, which will be described in §4.3. However, the scheduling makes sense only when a node at an upper level can have multiple child nodes at lower levels. This indicates that the cases where if a set of seeds are assessed to be with the same coverage measurement M_i, all following measures

$M_{i+1}, \ldots, M_n$ will also be the same should be excluded. Motivated by this fundamental requirement, the main principle is that measurements generated by a less sensitive metric should always cluster seeds prior to more sensitive ones. Here, we use the same definition of sensitivity between two coverage metrics as in §3.2.

Specifically, take the multi-level metric in Figure 4.1 as an example. Seeds in the same M_F clusters must have the same function coverage. However, since M_E is more sensitive than M_F, these seeds are likely to have different edge coverage, resulting in multiple different sub-clusters. However, if we use M_E to cluster seeds prior to M_F, since seeds with the same edge coverage must also have the same function coverage, it is impossible further to put them into different sub- M_F clusters. As a result, each M_E node will have only a single M_F node, making the clustering useless.

As discussed in [156], $\succ_s$ is a partial order, so it is possible that two metrics are not comparable. To solve this problem, we propose a weaker principle: given two non-comparable coverage metrics, we should cluster a seed with the metric that will select fewer seeds before the one that will select more seeds.

Examples

Following the above principles, we propose two multi-level coverage metrics as examples. Both examples use three-level clustering that works well in our evaluation.

The top-level metric in both examples is C_F, which measures the function coverage. The middle-level metric is edge coverage C_E. Functions invoked are essential runtime features that are commonly used to characterize an execution, and edge coverage is widely used in fuzzers such as AFL and LIBFUZZER. Notably $C_E \succ_s C_F$.

The most important one is the bottom-level metric, which is the most sensitive one. In this work, we mainly evaluated a bottom-level metric called *distance* metric C_D. It traces conditional jumps (i.e., edges) of a program execution, calculates the hamming distances of the two arguments of the conditions as covered features, and treats each observed new distance of a conditional jump as new coverage. Unlike C_E or C_F that traces control flow features, C_D focuses on data-flow features and actively accumulates progress made in passing condition checks for fuzzing.

To understand whether our approach can support different coverage metrics (fitness functions), we also evaluated another coverage metric called *memory access* metric C_A. As the name implies, this metric traces all memory reads and writes, captures continuous access addresses as array indices, and treats each new index of a memory access as new coverage. C_A pays attention to data flow features and accumulates progress made in accessing arrays that might be long. To distinguish memory accesses that happen at different program locations, the measurement also includes the address of the last branch. However, since not all basic blocks contain memory accesses, C_A is not directly comparable to C_E using sensitivity. However, we observe that C_A can generate much more seeds than C_E, so C_A comes after C_E and its measurement M_A stays at the bottom level.

4.3 Hierarchical Seed Scheduling

This section discusses how to schedule seeds against hierarchical clusters generated by a multi-level coverage metric.

4.3.1 Scheduling Against A Tree of Seeds

Conceptually, a multi-level coverage metric $C^n \sim \langle C_1 \cdots C_n \rangle$ organizes coverage measurements (M_i) and seeds as a tree, where each node at layer (or depth) $i \in \{1, \cdots, n\}$ represents a cluster represented by M_i and its child nodes at layer $i+1$ represent sub-clusters represented by M_{i+1}. At leaf-level, each node is associated with real seeds. Additionally, at layer 0 is a virtual root node representing the whole tree. To schedule a seed, the scheduler needs to seek a path from the root to a leaf node.

Exploration vs Exploitation The main challenge a seed scheduler faces is the trade-off between seed exploration (trying out other fresh seeds) and exploitation (keep fuzzing a few interesting seeds to trigger a breakthrough). On the one hand, fresh seeds that have rarely been fuzzed may lead to surprisingly new coverage. On the other hand, a few valuable seeds that have led to significantly more new coverage than others in recent rounds encourage to focus on fuzzing them.

Organizing seeds in a tree with hierarchical clusters facilitates a more flexible control over the seed exploration and exploitation. Specifically, fuzzers can focus on a single cluster in which seeds cover the same functions at the first layer and then try out many (sub-)clusters with seeds exercising different edges at the second layer. Alternatively, fuzzers can also try out seeds exercising different groups of functions, then only pick seeds covering some specific edges.

In this work, we explore the feasibility of modeling the fuzzing process as a multi-armed bandit (MAB) problem [177, 186] and using the existing MAB algorithms to balance between exploitation and exploration. After trying several different MAB algorithms, we

decide to adopt the UCB1 algorithm [2, 7] to schedule seeds, since it works the best empirically, despite being one of the simplest MAB algorithms. As illustrated by function `SelectNextSeedToFuzz()` in algorithm 3, starting from the root node, our scheduling algorithm selects the child node with the highest score, which is calculated based on the coverage measurements, until reaching the last layer to select among leaf nodes that are associated with real seeds. Because all seeds have the same coverage at the leaf level, we scheduling them with round robin for simplicity.

Algorithm 3: Seed Scheduling Algorithm

```
    Input: seed set S
    Output: return the seed to fuzz
    Data: the tree T with n layers
          current working tree node cx
 1  Function SelectNextSeedToFuzz(S):
 2      T ← S.tree
 3      cx ← T.root
 4      foreach i ∈ {1, ··· , n} do
 5          children ← cx.child_nodes
 6          cx ← argmax_{x∈children} Score(x)
 7      end
 8      s ← cx.next_seed()
 9      return s
10  End
11
```

At the end of each round of fuzzing, nodes along the scheduled path will be rewarded based on how much progress the current seed has made in this round, e.g., whether there are new coverage features exercised by all the generated test cases. In this way, seeds that perform well are expected to have increased scores for competing in the following rounds, while seeds making little progress will be de-prioritized.

Note that a traditional MAB problem assumes a fixed number of arms (nodes in our case) so that all arms can get an estimation of their rewards at the beginning. However, our setup breaks this assumption since the number of nodes grows as more and more seeds are generated. To address this issue, we introduce a rareness score of a node, so that each new node will have an initial score to differentiate itself from other new nodes. We will discuss seed scoring in more detail later in §4.3.2.

It is also worth mentioning that a recent work Ecofuzz [186] proposed using a variant of the adversarial multi-armed bandit (AMAB) model to perform seed scheduling. However, it can not solve the seed exploration problem caused by more sensitive coverage metrics, as it attempts to explore all existing seeds at least once. Moreover, we have also experimented with the EXP3 algorithm that aims to solve the AMAB problem; but it performed worse than UCB1 in our setup.

4.3.2 Seed Scoring

How to score seeds directly affect the trade-off between exploration and exploitation. First, for exploitation, seeds that have performed well *recently* should have high scores as they are expected to make more progress. Second, for exploration, the scoring system should also consider the uncertainty of rarely explored seeds. We extended the UCB1 algorithm [2, 7] to achieve a balance between exploitation and exploration. From a high level, our scoring method considers three aspects of a seed: (1) its own rareness, (2) easiness to discover new seeds from this seed, and (3) uncertainty.

In order to discuss this in more detail, let us first define some terms more formally. First, we define the hit count of a feature $F \in \Gamma_l$ at level l as the number of test cases ever generated that cover the feature.

Definition 4.3.1. *Let P be the program under fuzzing, $\mathcal{I}$ be the set of all test cases that have been generated so far. The hit count of a feature F is $num\_hits[F] = |\{I \in \mathcal{I} : F \in C_l(P, I)\}|$.*

As observed in [17], features that are rarely exercised by test cases deserve more attention because they are not likely to be exercised by valid inputs. The rareness of a feature describes how rarely it is hit, which is the inverse of the hit count.

Definition 4.3.2. *The rareness of a feature F is $rareness[F] = \dfrac{1}{num\_hits[F]}$*

Before describing how we calculate the reward of a round of fuzzing, we first define the feature coverage of fuzzing seed s at round t.

Definition 4.3.3. *Let P be the program under fuzzing, $\mathcal{I}_{s,t}$ be the set of test cases generated at round t via fuzzing seed s. We denote the feature coverage at level C_l, $l \in \{1, \cdots, n\}$ as $fcov[s, l, t] = \{F : F \in C(P, I)\ \forall I \in I_{s,t}\}$*

Next, we describe how we calculate the reward to the seed just fuzzed after a round of fuzzing. An intuitive way is to count the number of new features covered as the reward. However, we quickly noticed that this does not work well. As the fuzzing campaign goes on, the probability of exercising new coverage is dramatically decreased, indicating that a seed can hardly obtain new rewards. Consequently, the mean reward of seeds may quickly decrease to zero. When we have many seeds with minor variances near zero, the UCB

algorithm cannot properly prioritize seeds. Moreover, under the common observation that infrequent coverage features deserve more exploration than others, seeds that can lead to inputs that exercise rare features are definitely more valuable, even if they do not cover new features. Motivated by these observations, we take the rareness of the rarest feature that is exercised by all generated inputs as the reward to the schedule seed. Formally, for a seed s that is fuzzed at round t, its fuzzing reward w.r.t. coverage metric C_l is

$$SeedReward(s, l, t) = \max_{F \in fcov[s,l,t]} (rareness[F]) \tag{4.1}$$

Based on the seed reward, we compute the reward to a cluster by propagating seed rewards to clusters scheduled at upper levels. More formally, let $\langle a^1, \ldots, a^n, a^{n+1} \rangle$ be the sequence of nodes (in the seed tree) selected at round t, where a^{n+1} is the seed node for s and a_i is coverage measurements for the corresponding clusters. Since scheduling node a^l affects the following scheduling of nodes $a^{l+1}, \cdots, a^n$ at lower layers, the reward of node a^l as feedback consists of the seed reward regarding coverage levels $l, l+1, \cdots, n$ as illustrated in Equation 4.2. Note that we use the geometric mean here since it can handle different scalars of the involved values with ease.

$$Reward(a^l, t) = \sqrt[n-l+1]{\prod_{l \le k \le n} SeedReward(s, k, t)} \tag{4.2}$$

Right now, we are able to estimate the expected performance of fuzzing a node using the formula of UCB1 [2, 7]. Formally, the fuzzing performance of a node a is estimated as

$$FuzzPerf(a) = Q(a) + U(a) \tag{4.3}$$

$Q(a)$ is the empirical average of fuzzing rewards that a obtains so far, and $U(a)$ is radius of the upper confidence interval.

Unlike UCB1 which calculates $Q(a)$ using the arithmetic mean of the rewards that node a obtains so far, we use the weighted arithmetic mean instead. More specifically, during the fuzzing, the rareness of a feature is decreasing as it is exercised by more and more test cases. As a result, even the same fuzzing coverage can lead to different fuzzing rewards for mutating a seed: the reward of an earlier round might be significantly higher than that of a later round. To address this issue, we introduce a discount factor as weight in order to favor newer rewards rather than older ones. More formally, given a node a that is selected for round t, we update its weighted mean at the end of round t in such a way that we progressively decrease the weight to the previous mean reward in order to give higher weights to newer rewards as illustrated in Equation 4.4

$$Q(a,t) = \frac{Reward(a,t) + w \times Q(a,t') \times \sum_{p=0}^{N[a,t]-1} w^p}{1 + w \times \sum_{p=0}^{N[a,t]-1} w^p} \tag{4.4}$$

$N[a,t]$ denotes the number of times that node a has been selected so far at the end of round t, t' is the last round at which node a was selected, and w is the discount factor. Note that the smaller w is, the more we ignore the past rewards. When w is set to 0, all the past rewards are ignored. To study how w affects the fuzzing performance, we conduct

an empirical experiment (§4.4.3). Based on the results (see Table 4.5 and Table 4.6), we empirically set w to 0.5 in our evaluation.

$U(a)$ is the estimated radius factoring in the number of times a has been selected. In addition, we also consider the number of seeds that a contains based on the insight that nodes with more seeds should be scheduled more for seed exploration. More formally, given a seed a and its parent a', we calculate $U(a)$ as

$$U(a) = C \times \sqrt{\frac{Y[a]}{Y[a']}} \times \sqrt{\frac{\log N[a']}{N[a]}} \tag{4.5}$$

$Y[a]$ denotes the number of seeds in the cluster of node a, and $N[a]$ denotes the times a has been selected so far. C is a pre-defined parameter that configures the relative strength of exploration and exploitation. In particular, a larger C results in a relatively wider radius in Equation 4.3, which encourages exploring fresh nodes that have been fuzzed fewer times. This can help a fuzzer get out of code regions that are too hard to solve. On the contrary, a smaller C indicates that the empirical average of fuzzing rewards gets weighted more, thus promoting nodes that have recently led to good progress. As a result, the fuzzer will focus on these nodes and is expected to reach more new code coverage. To further demonstrate how it affects the fuzzing performance, we fuzz the CGC benchmark with different values of C and show the results in §4.4.3. Based on the results, we set C to 1.4 in our evaluation.

The fuzzing performance estimated by Equation 4.3 based on fuzzing coverage is limited by what can be observed. This limitation can impact seeds that have never been scheduled and seeds that exercise rare features themselves but usually lead to inputs

that exercise high-frequency features (e.g., for a program with rigorous input syntax checks, random mutations usually lead to invalid paths, hence lowering the reward). To mitigate this limitation, when evaluating a seed, we also consider features that it exercises. Particularly, we calculate the rareness of a seed via aggregating the rareness of features that it covers. More formally, let P be the program under fuzzing, given a seed s, its rareness regarding M_l, $l \in \{1, \cdots, n\}$ is

$$SeedRareness(s, l) = \sqrt{\frac{\sum_{F \in C_l(P,s)} rareness^2[F]}{|\{F : F \in C_l(P, s)\}|}} \tag{4.6}$$

Note that here we take quadratic mean rather than, e.g., arithmetic mean because it preserves more data diversity. The rareness of a node a^l measured by M_l is completely decided by its child seeds as they share the same coverage regarding M_l. Let $\langle a^1, \cdots, a^n, a^{n+1} \rangle$ be the sequence of nodes selected at round t, where a^{n+1} is the leaf node representing a real seed s, then at the end of round t the rareness of node a^l is updated as

$$Rareness(a^l) = SeedRareness(s, l) \tag{4.7}$$

Notably, we update the rareness score of seeds and nodes lazily for two reasons. First, it reduces the performance overhead. Second, it can lead to overestimating the rareness of a node that has not been fuzzed for a long time, so that seed is more likely to be scheduled.

In addition to updating the rareness of a node picked in the past round, we also calculate the rareness of each new node similarly. As discussed previously, this makes each

new node have an initial score to differentiate itself from other new nodes before its reward is estimated.

Finally, we have the score of a node a via multiplying its rareness and estimated fuzzing performance together as shown in Equation 4.8. This score is the one used in algorithm 3 to determine which nodes will be picked and which seed will be fuzzed next.

$$Score(a) = Rareness(a) \times FuzzPerf(a) \tag{4.8}$$

4.4 Evaluation

Our main hypothesis is that our multi-level coverage metric and hierarchical seed scheduling algorithm driven by the MAB model can achieve a good balance between exploitation and exploration, thus boosting the fuzzing performance. To validate our hypothesis, we implemented two prototypes AFL-HIER and AFL++-HIER, one based on AFL [188] and the other based on AFL++ [51], and evaluated them on various benchmarks aiming to answer the following research questions.

- **RQ1.** Can AFL-HIER/AFL++-HIER detect more bugs than the baseline?
- **RQ2.** Can AFL-HIER/AFL++-HIER achieve higher coverage than the baseline?
- **RQ3.** How much overhead does our technique impose on the fuzzing throughput?
- **RQ4.** How well does our hierarchical seed scheduling mitigate the seed explosion problem caused by high sensitive of coverage metrics?

- **RQ5.** How do the hyper-parameters affect the performance of our hierarchical seed scheduling algorithm?

- **RQ6.** How flexible is our framework to integrate other coverage metrics?

4.4.1 Implementations

For evaluation over the CGC dataset, we used a prototype built on top of the code open-sourced in our former work [156], which is based on AFL QEMU-mode, for its support for binary-only targets and its emulation of CGC system calls. For evaluation over the FuzzBench dataset, we used a prototype built upon the AFL++ project [51] (QEMU-mode only), for its support of persistent mode and higher fuzzing throughput.

4.4.2 Experiment Setup

Benchmarks

The first set of programs are from DARPA Cyber Grand Challenge (CGC) [27]. These programs are carefully crafted by security experts that embed different kinds of technical challenges (e.g., complex I/O protocols and input checksums) and vulnerabilities (e.g., buffer overflow, integer overflow, and use-after-free) to comprehensively evaluate automated vulnerability discovery techniques. There are 131 programs from CGC Qualifying Event (CQE) and 74 programs from CGC Final Event (CFE), a total of 205. CGC programs are designed to run on a special kernel with seven essential system calls so that competitors can focus on vulnerability discovery techniques. In order to run those programs within a normal Linux environment, we use QEMU to emulate the special system calls. Unfortu-

nately, due to imperfect simulation, some CGC programs fail to be run correctly. We also cannot handle programs that consist of multiple binaries, which communicate with each other through pre-defined inter-process communication (IPC) channels. As a result, we can only successfully fuzz 180 CGC programs (or binaries, in other words). We fuzz each binary for two hours and repeat each experiment 10 times to mitigate the effects of randomness. Each fuzzing starts with a single seed "`123\n456\n789\n`". We chose this initial because the initial seed affects the fuzzing progress a lot: seeds that are too good may make most code covered at the beginning if the program is not complex, while poor ones may make the fuzzing get stuck before reaching the core code of the program. These two cases both will make the fuzzing reach the plateau early, and fail to show the performance differences between our approach and other fuzzers. The seed we chose showed a good capability to reveal performance differences between fuzzers.

The second benchmark set is the Google FuzzBench [70] that offers a standard set of tests for evaluating fuzzer performance. These tests are derived from real-world open-sourced projects (e.g., libxml, openssl, and freetype) that are widely used in file parsers, protocols, and font operations. For this dataset, we used the standard automation script to run the benchmarks, so each benchmark uses the seeds provided by Google.

Baseline Fuzzers

For AFL-based prototype, we choose three fuzzers as the baseline for comparison: the original AFL [188], AFLFAST [17], and AFL-FLAT [156]. AFL-FLAT is configured with edge sensitivity C_E and distance sensitivity C_D (see §4.2.4 for more details), but uses the power scheduler from AFLFAST instead of our hierarchical scheduler. As discussed in §2.1,

the performance of greybox fuzzing is mainly affected by four factors: seed selection, seed scheduling, mutation strategies, and fuzzing throughput. We made all fuzzers use the same mutation strategy to reflect the benefit of our approach, and ran the experiments ten times to minimize the impact of randomness [88]. We also ran all fuzzers in the QEMU mode so they can have similar fuzzing throughput, which also makes it easier to assess AFL-HIER's performance overhead. Comparisons with AFL and AFLFAST aim to show the overall performance improvement of AFL-HIER; and comparison with AFL-FLAT aims to show the necessity/benefit of our scheduler (i.e., increasing the sensitivity of the coverage metric alone is not enough).

For AFL++-based prototype, we choose two fuzzers as the baseline: the original AFL++[1] [51] and AFL++-FLAT. We ran all fuzzers in the QEMU-mode and enabled persistent mode for better throughput

Computing Resources

All the experiments are conducted on a 64-bit machine with 48 cores (2 Intel(R) Xeon(R) Platinum 8260 @2.40GHz), 375GB of RAM, and Ubuntu 18.04. Each fuzzing instance is bound to a core to avoid interference.

[1]The version is 2.68c which our prototype is built on.

4.4.3 Evaluation Results

RQ 1. Bug Detection

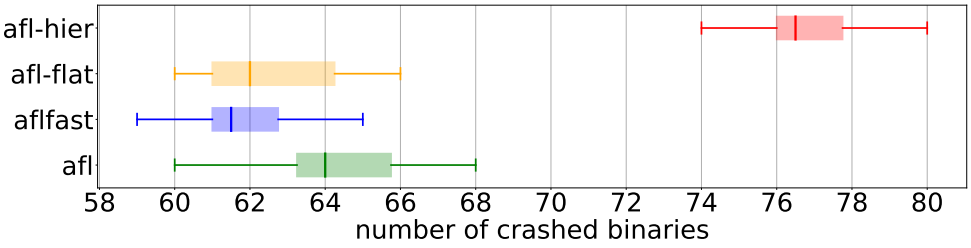
⋆ In experiments with CGC benchmarks, AFL-HIER crashes more binaries and faster. Especially, it crashes the same number of binaries in 30 minutes, that AFLFAST crashes in 2 hours.

In this experiment, we evaluate fuzzers' capability of detecting known bugs embedded in the CGC binaries. Figure 4.2a shows the number of crashed CGC binaries across ten rounds of trials. Note that since each binary supposedly only has one vulnerability, this number equals the total number of unique crashes. On average, AFL crashed 64 binaries, AFLFAST crashed 61 binaries, and AFL-FLAT crashed 62 binaries. In contrast, AFL-HIER crashes about 77 binaries on average, which is about 20% more binaries in the 2-hour fuzzing campaign. AFL-HIER also performed much better when we look at the lower and upper bound: its lower bound of crashes (74) is always higher than the upper bound of all other fuzzers. Notably, these vulnerabilities are carefully designed by security experts to highly mimic real-world security-critical vulnerabilities.

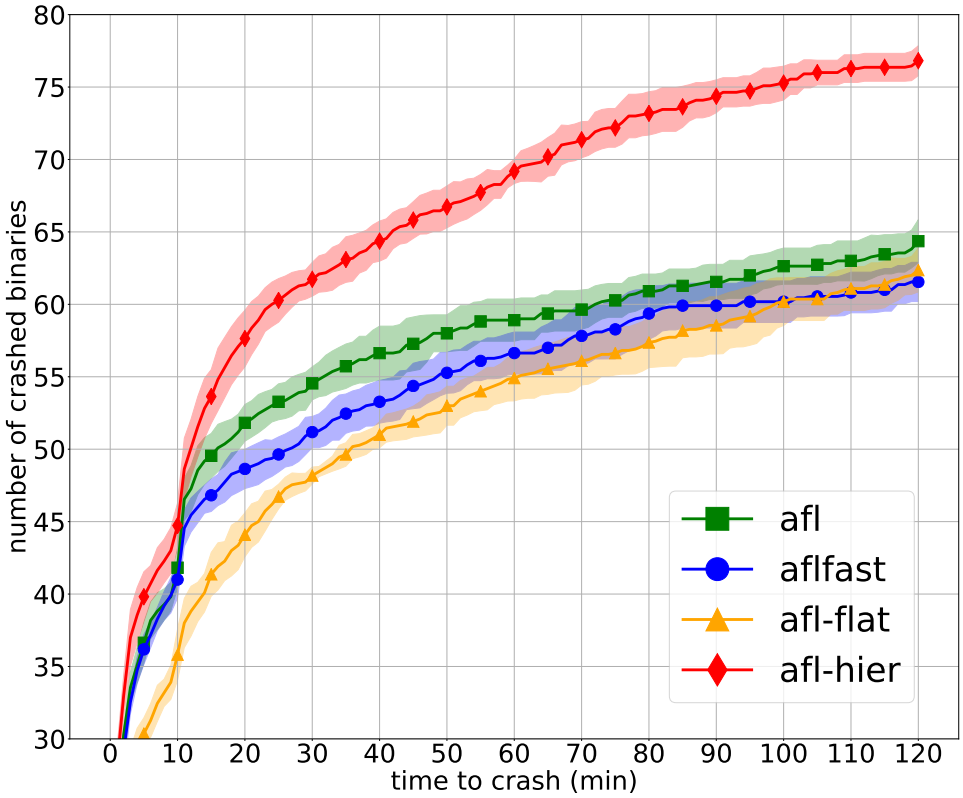
Table 4.1: Pairwise comparisons (row vs. column) of uniquely crashed on CGC benchmark.

	AFL	AFLFAST	AFL-FLAT	AFL-HIER
AFL	-	8	16	5
AFLFAST	3	-	13	5
AFL-FLAT	11	13	-	1
AFL-HIER	**17**	**22**	**18**	-

Table 4.1 shows the pairwise comparisons of CGC binaries uniquely crashed by a fuzzer across ten rounds of trails. As we can see, the added (distance) sensitivity C_D allows



(a) Number of crashed CGC binaries.



(b) Number of CGC binaries crashed over time.

Figure 4.2: Crash detection on CGC benchmarks.

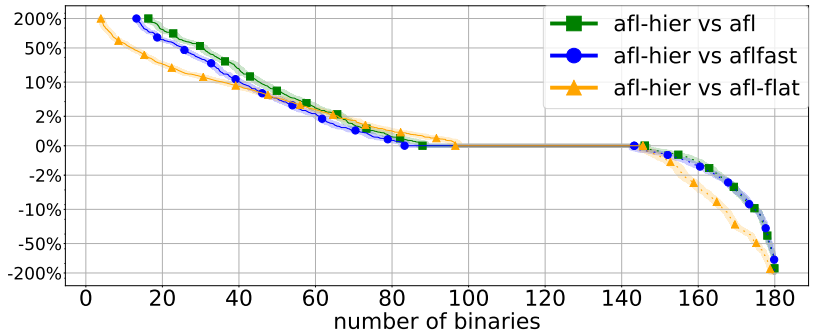
AFL-FLAT and AFL-HIER to crash a considerable amount of binaries that edge sensitivity (i.e., AFL and AFLFAST) cannot crash. However, due to the seed explosion problem, AFL-FLAT could not efficiently explore the seed pool; so it also missed many bugs AFL and AFLFAST can trigger. In contrast, AFL-HIER can achieve a good balance between exploration and exploitation: it crashed more unique binaries and missed much less.

Next, we measured the time to first crash (TFC) and show the accumulated number within a 95% confidence of binaries crashed over time in Figure 4.2b. As shown in recent studies [14, 74], TFC is a good metric to measure the performance of fuzzers. The x-axis presents the time in minutes, and the y-axis shows the number of crashed binaries. As shown in the graph, AFL-HIER stably crashed about 20% more binaries than other fuzzers from the beginning to the end. Notably, AFL-HIER crashed the same number of binaries in 30 minutes as AFLFAST did in 120 minutes; and crashed the same number of binaries in 40 minutes as AFL did in 120 minutes. In contrast, AFLFAST was lagging behind AFL and AFLFAST in most of the time and only surpassed AFLFAST after 100 minutes. This result showed that our hierarchical scheduler not only can find many unique bugs but also can efficiently explore the search space.

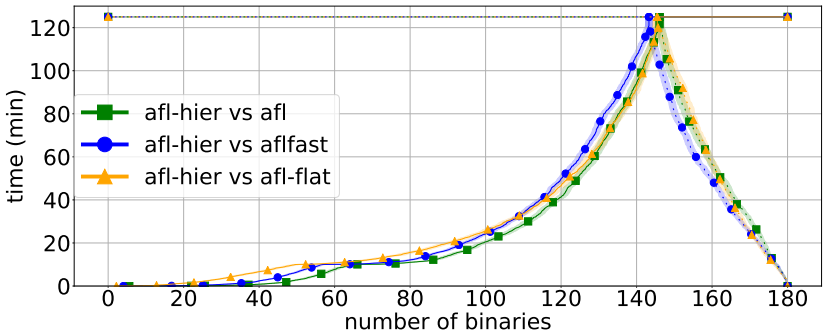
RQ 2. Code Coverage

⋆ Results on CGC binaries demonstrate that AFL-HIER generally achieved more code coverage and achieved the same coverage faster. Specifically, AFL-HIER increases the coverage by more than 100% for 20 binaries, and achieves the same coverage in 15 minutes that AFLFAST achieves in 120 minutes for about half of the binaries. On FuzzBench, AFL++-HIER achieved higher coverage on 10 out of 20 projects.

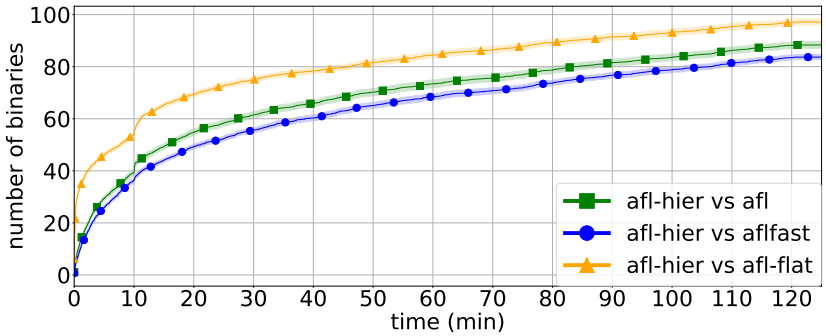
CGC Benchmark In this experiment, we first measured the edge coverage achieved by fuzzers using QEMU (i.e., captured during binary translation) on CGC binaries.



(a) **Mean coverage increase.** For X binaries, AFL-HIER achieves at least Y% more coverage than other fuzzers. A curve towards upper-right indicates that AFL-HIER outperforms the other more significantly.



(b) **Time to coverage.** For X binaries, AFL-HIER achieved the same coverage in Y minutes, as the other fuzzer achieved in 2 hours (solid line). A curve in solid line towards the lower right with its counterpart in dashed line towards lower left indicates a more statistical significance of that AFL-HIER achieve coverage faster than the opponent.



(c) **Better coverage.** After X minutes of fuzzing, AFL-HIER achieves more coverage than other fuzzers for Y binaries. An curve towards upper left indicates that AFL-HIER achieves better coverage than the opponent more significantly.

Figure 4.3: Coverage improvement on the CGC benchmarks.

Figure 4.3a illustrates the mean code coverage increase of AFL-HIER over other fuzzers for the 180 CGC binaries, after 2 hours of fuzzing. The curve above 0% means AFL-HIER covered more and the curse below 0% means AFL-HIER covered less. The x-axis presents the accumulated number of binaries within a 95% confidence, and the y-axis shows the increased coverage in *logarithmic* scale. For example, there are about 20 binaries for which the code coverage is increased by at least 100%, and about 45 binaries for which the code coverage is increased by at least 10%. After 2 hours of fuzzing, AFL-HIER achieved more coverage for about 90 binaries than other fuzzers and achieved the same coverage for 50 binaries. Among about 30 binaries on which AFL-HIER achieves less coverage, on half of them the difference is lower than 2%; and only on five of them the difference is greater than 10%. This result shows that our approach can cover more or similar code on most binaries besides detecting more bugs.

Figure 4.3b illustrates how fast AFL-HIER can achieve *the same* coverage as other fuzzers in two hours. The dashed lines (on the right-hand-side after hitting 120 min) show for the cases where baseline fuzzers achieved more final coverage in two hours. The x-axis shows the accumulated number of binaries within a 95% confidence, while the y-axis shows the time in minutes. We can see that for about half of the total 180 binaries, AFL-HIER achieved the same coverage in 15 minutes as baseline fuzzers did in 2 hours. Moreover, for about 110 binaries, AFL-HIER achieves the same coverage in half an hour; and for about 130 binaries, AFL-HIER achieves the same coverage in one hour. Similar to TFC (time to first crash), this result also shows that our approach can achieve the same coverage faster, indicating it can balance exploration and exploitation well.

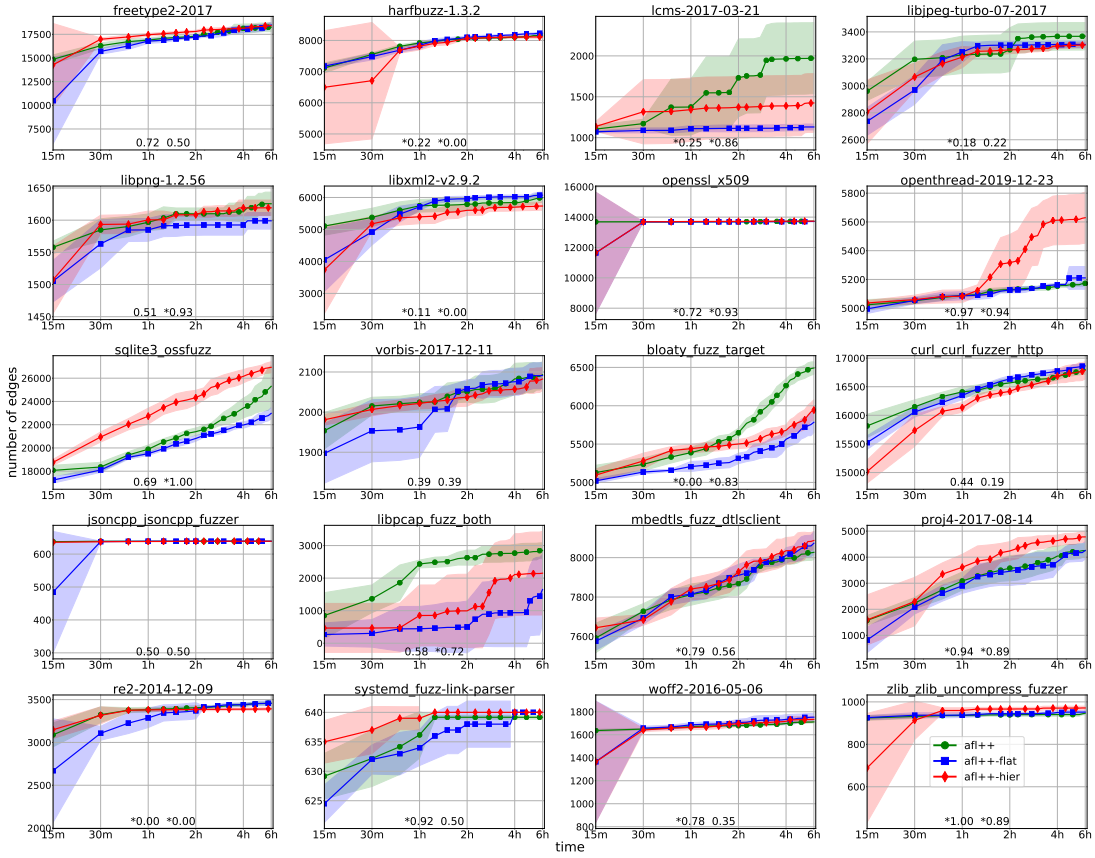


Figure 4.4: Mean coverage in a 6 hour fuzzing campaign on FuzzBech benchmarks.

Figure 4.3c shows the number of binaries for which AFL-HIER achieved *more* coverage than other fuzzers over time. The x-axis represents the time in minutes and the y-axis shows the accumulated number of binaries within a 95% confidence that AFL-HIER won on coverage. We can observe that after 10 minutes, AFL-HIER already won for about 40 binaries over AFL and AFLFAST. After 1 hour, it further increased the gap by winning for more than 70 binaries. Overall, AFL-HIER steadily won for more and more binaries throughout the process of the 2-hour fuzzing campaign. This indicates that AFL-HIER can continuously make breakthroughs in new coverage for binaries when other fuzzers plateaued.

FuzzBench Next, we compare AFL++-HIER with two baseline fuzzers (AFL++ and AFL++-FLAT) on Google FuzzBench benchmarks. Figure 4.4 shows the mean coverage (with confidence intervals) over time during 6-hour fuzzing campaigns[2]. The y-axis presents the number of covered edges and the x-axis represents time. Please note that the x-axis is in *logarithmic* scale, as recent work suggests the required efforts to achieve more coverage grow exponentially [14]. Meanwhile, the Vargha-Delaney [153] effect size $\hat{A}_{12}$ is shown at the bottom of each sub-figure, where the left one is of between AFL++-HIER over AFL++ (Qemu) and the right one is of between AFL++-HIER and AFL++-FLAT, respectively. A value above 0.5735, 0.665, 0.737 (or below 0.4265, 0.335, 0.263) indicates a small, medium, large effect size. More intuitively, a larger value above 0.5 indicates a higher probability of that AFL++-HIER will cover more edges than AFL++ (Qemu) or AFL++-FLAT in a fuzzing campaign. Moreover, a value starting with a star indicates a statistical significance tested by Wilcoxon signed-rank test ($p < 0.05$). Overall, AFL++-HIER could beat AFL++ (Qemu) and AFL++-FLAT on about ten projects, and achieved significantly more coverage on projects openthread, sqlite3, and proj4.

Table 4.2 shows the unique edge coverage, which is union over different runs, of AFL++ (Qemu) and AFL++-HIER. The results indicate even on programs where AFL++-HIER has lower mean coverage than AFL++, it still can cover some unique edges AFL++ does not cover. Note that here we union edge coverage across different runs, so for some benchmarks like lcms and libpcap, though the mean coverage differences are large, the unique coverage differences are much smaller.

[2]We are working with Google to provide a 23-hour run that compares with more fuzzers.

Compared to the results on the CGC benchmarks, we observe that our performance is not significantly better than AFL++ on most of the FuzzBench benchmarks. We suspect the reason is that our UCB1-based scheduler and the hyper-parameters we used in the evaluation prefer exploitation over exploration. As a result, when the program under test is relatively smaller (e.g., CGC benchmarks), our scheduler can discover more bugs without sacrificing the overall coverage by too much. But on FuzzBench programs, breaking through some unique edges (Table 4.2) can be overshadowed by not exploring other easier to cover edges.

Table 4.2: Unique edge coverage between afl++ (Qemu) and afl++-hier (Hier) on FuzzBench benchmarks.

Benchmark	**Total**	**Hier - Qemu**	**Qemu - Hier**
bloaty_fuzz_target	102417	24	674
curl_curl_fuzzer_http	143182	203	114
freetype2-2017	56114	774	1227
harfbuzz-1.3.2	13073	58	124
jsoncpp_jsoncpp_fuzzer	2583	0	0
lcms-2017-03-21	12817	36	33
libjpeg-turbo-07-2017	18486	0	237
libpcap_fuzz_both	11800	141	195
libpng-1.2.56	5944	6	54
libxml2-v2.9.2	89852	52	210
mbedtls_fuzz_dtlsclient	32046	142	102
openssl_x509	115381	26	14
openthread-2019-12-23	42901	344	0
proj4-2017-08-14	10434	109	67
re2-2014-12-09	5904	2	100
sqlite3_ossfuzz	48181	1880	965
systemd_fuzz-link-parser	4167	0	0
vorbis-2017-12-11	6372	8	4
woff2-2016-05-06	6401	54	8
zlib_zlib_uncompress	1664	24	0

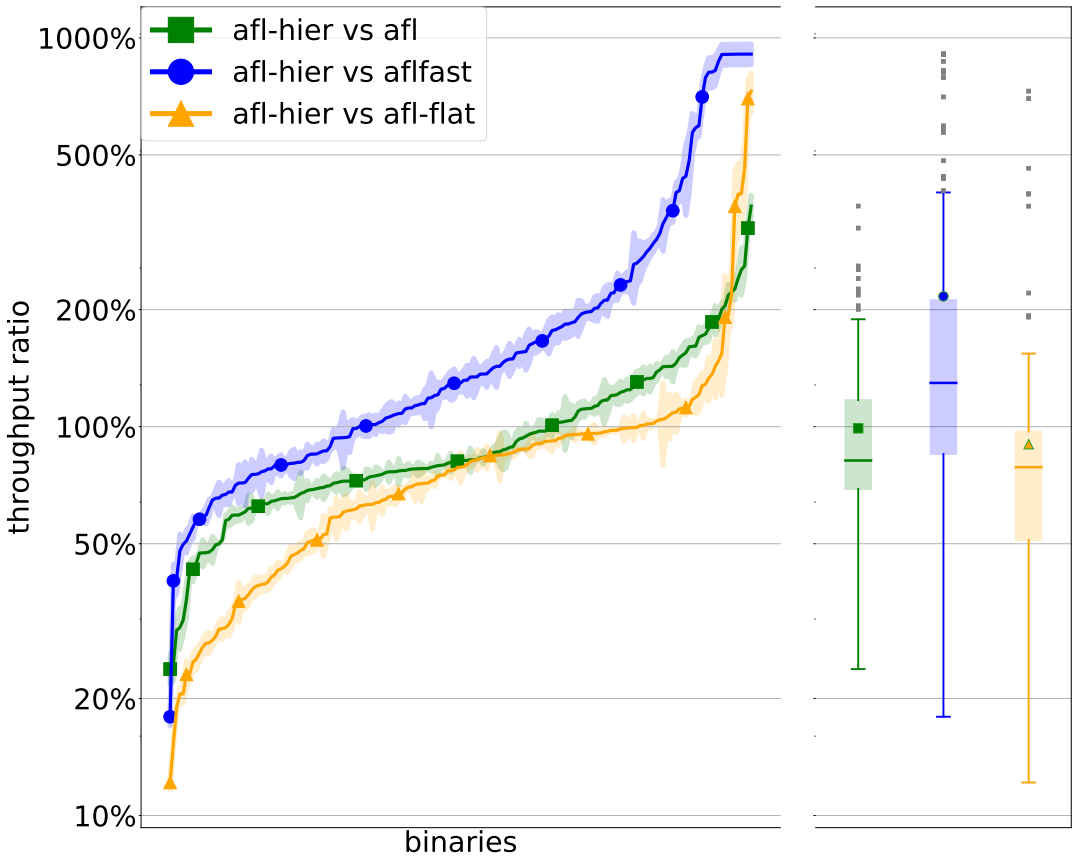


Figure 4.5: Comparison between throughput of AFL-HIER, AFL, AFLFAST and AFL-FLAT on CGC benchmarks.

RQ 3. Fuzzing Throughput

⋆ **Results on CGC benchmarks show that AFL-HIER has a competitive throughput as AFL and AFLFAST. Moreover, even built on the faster fuzzer AFL++, AFL++-HIER still has a comparable throughput as shown by the results on FuzzBench benchmarks.**

A multi-level coverage metric requires collecting more coverage measurements during runtime and performing more operations to insert a seed into the seed tree. Similarly, our hierarchical scheduler also requires more steps than the power scheduler of AFL and

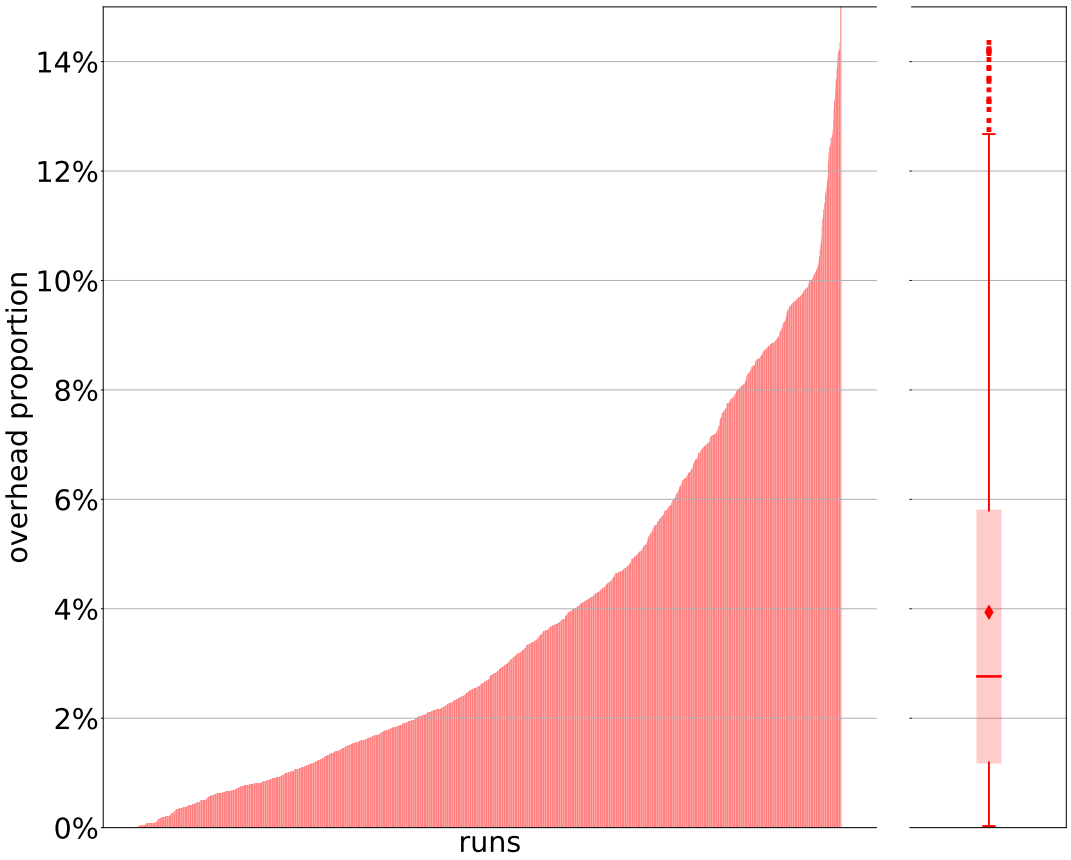


Figure 4.6: Overhead of AFL-HIER scheduler on CGC benchmarks.

AFLFAST. Therefore, we expect our approach to have a negative impact on fuzzing throughput. Moreover, the multi-level coverage metric is sensitive to minor variances of test cases and execution paths; consequently, it is more likely to schedule larger and more complex seeds leading to longer execution time.

To quantify the impact on fuzzing throughput, we first investigated the proportion of the time that AFL-HIER spends in scheduling, which involves maintaining the incidence frequencies and the tree of seeds and choosing the next seed to fuzz. The results on CGC benchmarks are shown in Figure 4.6, where the x-axis represents individual runs (in total $10 \times 180 = 1800$) and the y-axis shows the portion of time spent on scheduling. We can

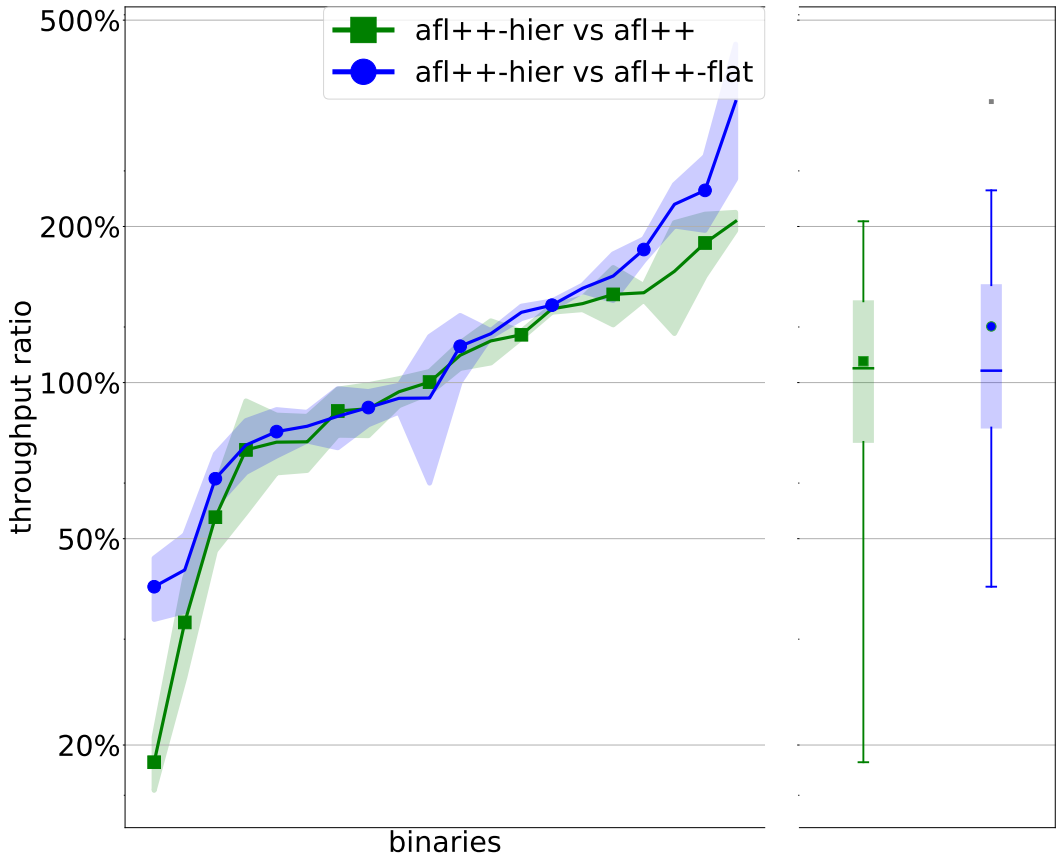


Figure 4.7: Comparison between throughput of AFL++-HIER, AFL++, and AFL++-FLAT on FuzzBench benchmarks.

see that the median overhead is as low as 3%, and most overhead is lower than 10%. On AFL++-based prototype, we observed lower performance overhead, as shown in Figure 4.8.

Next, we measured the throughput of AFL-HIER versus AFL and AFLFAST on CGC benchmarks. Figure 4.5 shows the ratio of AFL-HIER's throughput over AFL and AFLFAST in an ascending order. The x-axis represents different CGC binaries while the y-axis shows the ratio within a 95% confidence in *logarithmic* scale. Surprisingly, AFL-HIER only leads to a lower throughput for about a quarter of the binaries; and for another quarter of the binaries, AFL-HIER's throughput is at least twice as AFLFAST's. This indicates that the

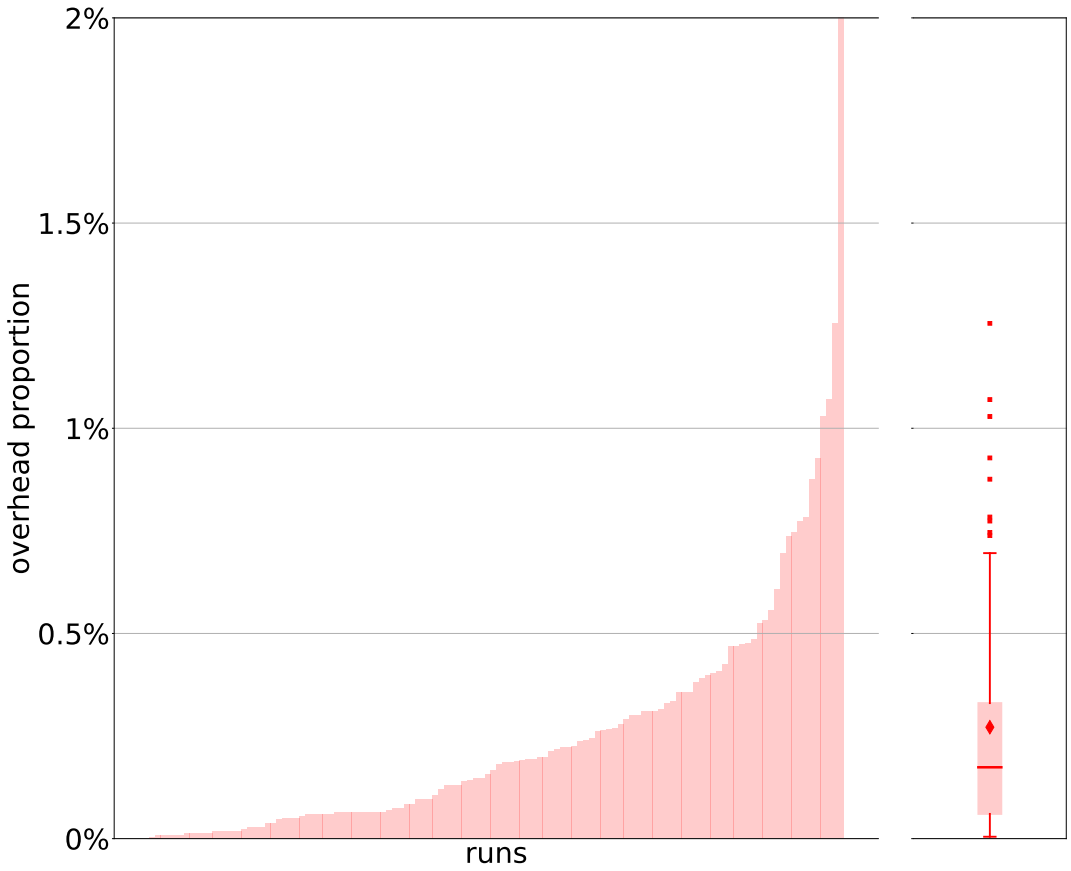


Figure 4.8: Overhead of AFL++-HIER scheduler on FuzzBench benchmarks.

specific optimizations for AFL-HIER act very well. A similar trend is also observed on the AFL++-based prototype, as shown in Figure 4.7.

RQ 4. Performance Boost via Hierarchical Seed Scheduling

⋆ **Experiment results on CGC and FuzzBench benchmarks demonstrate that our hierarchical seed scheduler dramatically reduces the number of candidates to be examined.**

Previous experiments already show that our hierarchical seed scheduler is more suitable for highly sensitive coverage metrics, as AFL-HIER can achieve higher coverage

faster than AFL-FLAT and find more bugs. In this evaluation, we investigate the number of seeds generated by each fuzzer to validate that such improvement is indeed caused by the scheduler. Figure 4.9 shows the number of seeds generated by each fuzzer on the left side, as well as the number of nodes at different levels of the tree in AFL-HIER on the right side. The y-axis is in *logarithmic* scale. We can observe that due to the increased sensitivity of distance metric C_D, both AFL-HIER and AFL-FLAT selected one magnitude more seeds than AFL and AFLFAST, which uses edge coverage with hit count. However, by clustering the seeds in a hierarchical structure, AFL-HIER dramatically reduced the number of candidates to examine when scheduling. Specifically, on average there are about $21 + 1102/21 + 2350/1102 + 2608/2350 \approx 77$ examinations to perform for each scheduling, which is significantly less than examining 2608 seeds. As a result, even with the most number of seeds (more than AFL-FLAT), AFL-HIER can still balance exploration and exploitation and achieve better fuzzing performance (in terms of coverage and detected bugs) than baseline fuzzers.

On FuzzBench benchmarks, we also observed a similar level of reduction, as shown in Figure 4.10. More importantly, we can see that our scheduling algorithm can scale to larger programs with significantly more edges and more saved seeds. As shown in Table 4.2, all the benchmarks have at least thousands of edges in total, and some even contain more than one hundred thousand edges.

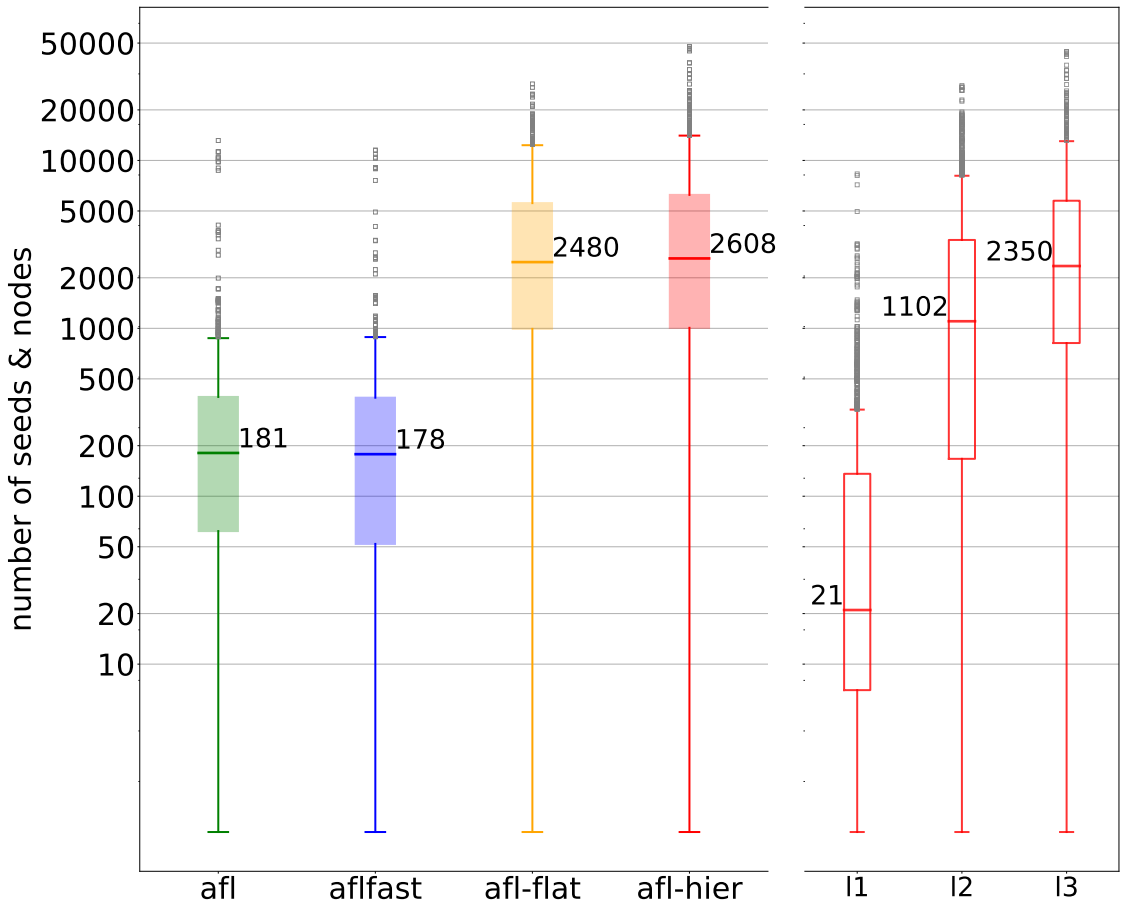


Figure 4.9: Number of seeds and nodes on CGC benchmarks.

RQ 5. Hyper-Parameters

⋆ **Experiment results on CGC benchmarks demonstrate that the hyper-parameters will affect the performance in terms of crashes and edge coverage.**

As discussed in §4.3.2, the seed scoring involves two hyper-parameters. One is w in Equation 4.4 that determines how much we will decrease weights to old rewards when calculating the mean reward. The other one is C in Equation 4.5 that controls the trade-off between seed exploration and exploitation. In this evaluation, we investigate when they are set to different values, how the fuzzing performance will vary. Table 4.3 and Table 4.5 show

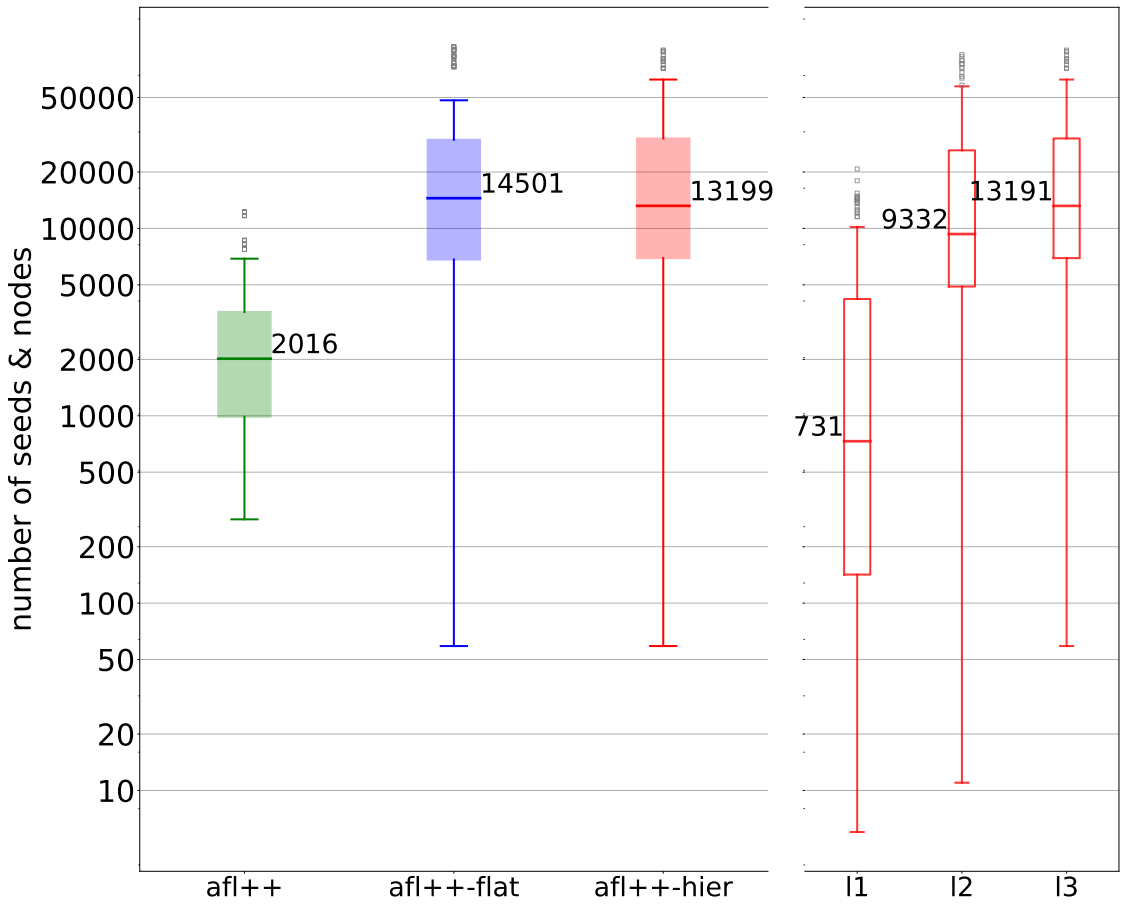


Figure 4.10: Number of seeds and nodes on FuzzBech benchmarks.

the average number of crashed binaries and covered edges with different values of C and w, respectively. In addition, we also investigate the number of binaries uniquely crashed as shown in Table 4.4 and Table 4.6, where each cell represents the number of binaries that have been crashed by the setting of the row once but never by the setting of the column. We can observe that different settings will lead to different results.

Notably, when C is set to 0, which extremely encourages exploitation, it uniquely crashes the most binaries, but on average, it crashes the least. This indicates that although keeping exploitation may help to trigger a crash at the end of a seed chain in one run, it

also takes the risk of being trapped in fuzzing other seeds that previously have led to rarely explored coverage, thus missing the crash in other runs. In other words, high exploitation may do better in crash triggering than crash reproducing. Meanwhile, the result of edge coverage indicates that exploring more coverage may not be closely related to bug detection as expected when under different configurations of the relative strength of exploration and exploitation. For example, setting C to 0.014 will lead to significantly less coverage, but it crashes almost the same number of binaries as others.

In terms of the hyper-parameter w, note that a larger w makes old rewards more weighted, thus encourages seed exploitation rather than exploration. We can observe that setting w either too high (as 1.0) or too low (as 0.5) will lead to worse coverage, while setting w to 0.5 will lead to significantly more unique crashes.

Overall, we can observe that when setting C to 1.4, w to 0.5, they perform reasonably well in average crashes, unique crashes, and mean edge coverage. Thus we adapt these settings in our current implementation.

Table 4.3: Average number of crashed CGC binaries and mean edge coverage with different values of hyper-parameter C.

Value of C	0	0.014	0.14	**1.4**	14
Crash	74	75	75	76	75
Edge Cov	776	667	748	727	746

Table 4.4: Pairwise comparisons (row vs. column) of uniquely crashed on CGC benchmarks with different values of hyper-parameter C.

Value of C	0	0.014	0.14	**1.4**	14
0	-	8	7	4	9
0.014	3	-	2	3	4
0.14	4	4	-	5	5
1.4	3	7	7	-	9
14	3	3	2	4	-

Table 4.5: Average number of crashed CGC binaries and mean edge coverage with different values of hyper-parameter W.

Value of W	0.10	0.25	**0.50**	0.75	0.90	1.00
Crash	74	73	76	72	75	75
Edge Cov	698	758	727	666	739	660

Table 4.6: Pairwise comparisons (row vs. column) of uniquely crashed on CGC benchmarks with different values of hyper-parameter W.

Value of W	0.10	0.25	**0.50**	0.75	0.90	1.00
0.10	-	5	2	3	4	3
0.25	1	-	1	1	1	1
0.50	8	11	-	9	9	9
0.75	2	4	2	-	3	3
0.90	4	5	3	4	-	4
1.00	4	6	4	5	5	-

RQ 6. Ability to Support Other Coverage Metrics

⋆ Experiment results on the maze problem show that our hierarchical scheduler can also improve the fuzzing performance when using other sensitive coverage metrics.

As discussed in §4.2.1, it is very hard, if not impossible, to use edge or even distance coverage to solve the maze problem (Listing 4.1). However, it is possible to solve it using memory sensitivity C_A (see §4.2.4 for details). In this experiment, we investigate whether our hierarchical scheduler can also boost the performance of coverage metrics other than code-related coverage. Specifically, we configured AFL-FLAT and AFL-HIER to use memory access metric C_A instead of distance metric C_D and evaluate the two fuzzers on the maze problem. Table 4.7 shows the results. As we can see, compared to the power scheduler used by AFL-FLAT, our hierarchical scheduler allows AFL-HIER to solve the maze problem

Table 4.7: Average solving time for the maze problem (Listing 4.1).

Fuzzer	AFL-FLAT	AFL-HIER
Time (sec)	383 ± 92	180 ± 36

much faster. This empirical result suggests that our scheduler is flexible to support different coverage metrics.

4.5 Summary

Fine-grained coverage metrics like distances between operands of comparison operations and memory access allow greybox fuzzers to detect bugs that cannot be triggered by traditional edge coverage. However, existing seed scheduling algorithms cannot efficiently handle the increased number of seeds. In this chapter, we present a new coverage metrics design called multi-level coverage metric where we cluster seeds selected by fine-grained metrics using coarse-grained metrics. Combining with a reinforcement-learning-based hierarchical scheduler, our approach significantly outperform existing edge-coverage-based fuzzers on DARPA CGC challenges.

Chapter 5

Format-Aware Input Generation for More Effective Concolic Execution

5.1 Introduction

Concolic execution (CE) [19, 30, 62, 130, 131, 149, 187], as a modern variant of symbolic execution (SE) [23, 24, 28, 36, 61, 63, 113, 139, 140, 144], is a well-known approach for program analysis and software testing, due to its ability to systematically explore program states. In particular, it executes the program under test with a concrete input, collects symbolic path constraints along the concrete execution path, and selectively negates symbolic branches to generate new test inputs.

Input generation in CE is achieved via solving the negated path constraints (e.g., by consulting a satisfiability modulo theories (SMT) solver [10, 41]). Ideally, the newly generated test input is expected to negate the target branch and reach a new path. The

challenge, however, is *what to be included in the path constraints*. On one extreme end, we can force the new input follow the same execution path prefix (as the input it is derived from) until the target branch the CE engine aims to negate by including all constraints prior the target branch. However, following the same path prefix (i.e., including all path constraints prior the target branch) can be too restrictive that can make the negated branch constraints not feasible. For example, consider an implementation of `atoi` that uses a switch statement to parse string characters to integers. In this case, following the same path will always lead to the same output, making it impossible to generate a different output. Including all prior path constraints can also make the constraints too complex for the solver to find a solution.

On the other extreme, we can only consider the constraints from the target branch. However, inputs generated using this approach usually miss important conditions. As a result, the execution path may diverge earlier and never reach the target branch. It is also possible that the execution reaches the target branch with a different set of constraints (due to an earlier deviation from the path prefix), thus invalidating the solution returned by the solver.

To avoid these problems, modern CE engines [30, 33, 187] use a two-tier strategy to construct path constraints. They first try *nested solving*, which includes prior constraints over which the target branch has direct data-dependency. If nested solving failed (e.g., too complex or too restrictive), then they try *optimistic solving*, which only includes the target branch's constraints. While it is expected that inputs generated with optimistic solving are likely to fail to negate the target branch, our evaluation shows that a considerable portion of solutions generated by nested solving also failed to negate the target branch.

This problem is especially severe when the program under test handles highly formatted inputs. A formatted input usually consists of fields and chunks, where fields have *implicit* dependencies on other fields or chunks. For instance, some bytes as a field could represent the `size` of another chunk; so, simply mutating the values of these bytes when negating a branch without shrinking or expanding the corresponding chunk accordingly will lead to format inconsistency and invalid inputs.

In addition, our investigation also reveals that, even if the new input successfully negates the target branch, it may inadvertently affect some important branches afterwards, thus making the execution terminate too early without reaching deep code regions, since these conditions are not considered when constructing the path constraints. This issue makes it especially hard to generate valid formatted inputs. Take the JSON input `{"key":"abc","command":"root"}` as an example, where each field is a string. If we replace a string in-place (e.g., from `"abc"` to `"deadbeef"`), without shifting the following filed, then the new input (e.g., `{"key":"deadbeef"mand":"root"}`) will become invalid. Although it is possible to gradually fix the invalid inputs through additional rounds of mutations, such intermediate inputs may be dropped by the CE engine (e.g., due to no new coverage) [156].

In this work, we aim to improve the efficacy of input generation for CE by taking input format into account. To this end, we start with inferring the format for each executed input based on the key insight that *path constraints already contain rich information about how a program handles its input thus can also be used to infer the input format.* We first recover plausible field boundaries and hierarchies (i.e., the structure), based on how input bytes are used in branch conditions. For instance, contiguous bytes probably belong to one

field if they are assessed in the same static branch, and contiguous fields probably belong to one chunk if they are assessed under the same calling context. Next, we try to infer the possible semantic types for certain fields (e.g., `size`, `checksum`, `string`), based on specific patterns in the path constraints.

After inferring the input format, we apply three strategies to improve the efficacy of input generation. First, when constructing the path constraints that are used to negate a branch, we prefer to include branch conditions where fields have been found, expecting that new inputs would not break the structure. Secondly, during the solving process, in addition to querying an SMT solver, we search for chunks that already satisfy the negated branch condition in existing seeds, and adopt them as additional solutions. Notably, this methods can save redundant efforts of negating and solving path constraints to re-generate the chunks. Lastly, for each solution raised by the solver, if we find an involved fields of recognized types, we will adjust the solution accordingly to satisfy the constraints imposed by the types.

To validate our idea, we implement a prototype FORMATLY based on SYMSAN [30], a state-of-the-art concolic executor. Our evaluation with a set of popular real-world programs shows that, compared to nested-then-optimistic solving, our format-aware solving can (1) negate more symbolic branches, (2) lead to deeper new paths, and (3) unlock more code coverage. End-to-end hybrid fuzzing results also show that format-aware solving can lead to better edge coverage.

Contributions: In summary, this work makes the following contributions:

- We show that the nested-then-optimistic solving strategy fail to negate many symbolic branches.
- We propose a novel format-aware solving strategy that leverages path constraints to infer input format information, and leverages inferred format information to construct better path constraints.
- We implement a prototype FORMATLY and will release the source code.
- We evaluate FORMATLY with a set of real-world applications. The results show that it can significantly improve the performance of input generation, which enables better performance in end-to-end fuzzing.

5.2 Motivation

Our investigation indicates that when the program under test handles highly formatted inputs that consist of various fields and chunks, inputs generated by existing solving strategies (e.g., nested solving) could fail to negate the target branch due to failing to capture *implicit dependencies*.

First, it is common that some fields have impacts on other fields or chunks. For instance, considering an input where a field acts as an index to access another chunk, simply modifying the value of the offset field when negating a branch without adjusting the element to access in the chunk will lead to inconsistent inputs. Listing 1 shows such a case from the application `openssl`. Notably, `len` that is from the input is used as an index to access `p`

that is an array on the input at line 3, and the leading bit of the accessed element must be zero to avoid the early return at line 5. Consequently, when negating the conditional branch at line 10 to explore the code after the *taken* direction, we need to either keep `len`, which is assigned to `length` at line 8, intact and only modify `ret->length`, or adjust the array `p` accordingly if `len` has to be modified. Otherwise, the execution with the new input will terminate early before reaching the target branch. However, because there is no pure data-dependency between line 3 and line 10, nested solving will ignore the additional constraints and generate invalid inputs.

```
1  ASN1_OBJECT *ossl_c2i_ASN1_OBJECT(...) {
2      ...
3      if (... || p[len - 1] & 0x80) {
4          ...
5          return NULL;
6      }
7      ...
8      length = (int)len;
9      ...
10     if (... || (ret->length < length)) {
11         ...
12     }
13     ...
14 }
```

Listing 1: A code snippet from *openssl*

Second, arguably a more interesting case we found during our investigation is that path constraints *after* the target branch can also be important. Specifically, a new input with inconsistent format may negate the target branch but the execution would terminate prematurely without reaching deeper paths. Take a PNG file that consist of various types of data chunks as an example. Listing 2 displays the code snippet from the library `libpng` that identifies the type of a chunk and dispatches the execution to the corresponding handler. It first invokes the function `png_memcmp()` to compare the chunk name and determine the

chunk type. As the function compares the two arguments byte by byte, CE needs to negate a sequence of unmatched comparisons until reaching a valid chunk type. Furthermore, even after constructing a valid type (e.g., `png_IHDR`), there are additional checks on other fields (e.g., the `length == 13`) of the chunk. As a result, CE needs to further negate a sequence of *failed* checks to eventually generate a valid `IHDR` chunk. The whole process requires a sequence of execution paths, which is hard and time-consuming. More importantly, this process can be terminated prematurely due to path scheduling and filtering (e.g., QSYM uses a bitmap to track branches it has tried to negate and will not negate the same branch again).

```
1  void PNGAPI png_read_info(...) {
2    ...
3    if (!png_memcmp(chunk_name, png_IHDR, 4))
4      png_handle_IHDR(png_ptr, info_ptr, length);
5    else if (!png_memcmp(chunk_name, png_IEND, 4))
6      png_handle_IEND(png_ptr, info_ptr, length);
7    ...
8  }
9
10 void png_handle_IHDR(...) {
11   ...
12   if (length != 13)
13       png_error(png_ptr, "Invalid IHDR chunk");
14   ...
15 }
```

Listing 2: A code snippet from *libpng*

These examples motivate us to leverage format information when constructing the path constraints.

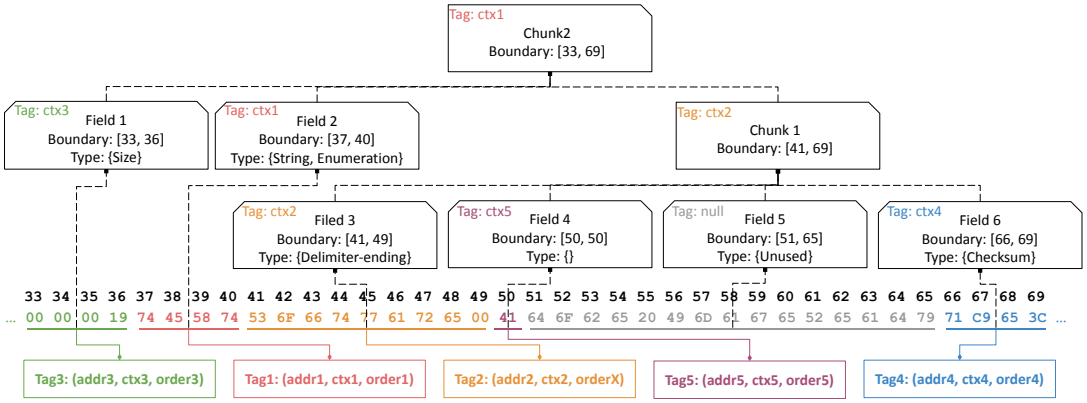
5.3 Input Format Inference

In this project, we do *not* assume the availability of the input format specification or grammar. Therefore, we need to recover the format information first.

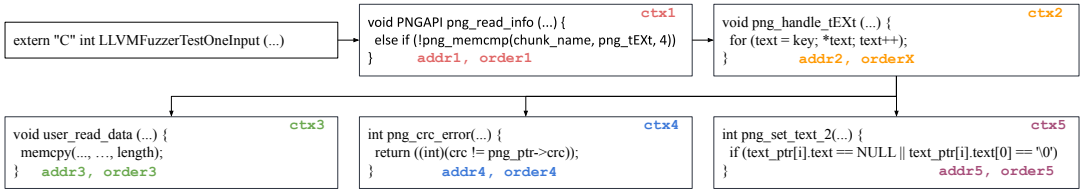
Generally, a formatted input consists of a sequence of disjoint fields that are assessed individually in certain conditional branches during the program execution. These fields can further be coupled into various chunks that are dealt separately with different handling functions. In consequence, an input can be virtually represented as a hierarchical tree, where the root is the whole input, internal nodes are chunks, and leaf nodes are fields. In addition, fields are usually used to describe various chunk attributes (e.g., chunk type and size). Based on these observations, we aim to (1) locate the boundaries of these fields, (2) recover the hierarchical structure derived from them, and (3) infer the types of attributes they describe.

5.3.1 Field Boundary and Hierarchy

We leverage the rich runtime information contained in path constraints to recover field boundaries, and their hierarchy. Figure 5.1 shows an example of inferred formation information from the `not_kitty.png` testcase. Specifically, based on the observation that *input bytes belong to the same field are usually used in the same branch*, we group input bytes into different fields. For example, the `chunk_name` field of a PNG chunk is checked using the same equality check inside `png_memcmp`. Similarly, based on the observation that *input fields belong to the same chunk are usually processed under the same calling context*, we group



(a) A snippet for the `tEXt` chunk and the inferred format for it



(b) A simplified function call graph about handling the `tEXt` chunk

Figure 5.1: Format inferred from the `not_kitty.png` testcase.

fields into chunks. For instance, a `IHDR` chunk of a PNG file is handled by `png_handle_IHDR`. Finally, the hierarchy of nested chunks is inferred based on the hierarchy of the calling context.

Notably, some format-aware greybox fuzzers like WEIZZ [50] also leverage similar observations to recover some format information like fields. The main difference is that for grey-box fuzzers, because they only collect lightweight runtime information (e.g., edge coverage), many necessary metadata has to be inferred. On the contrary, because concolic executors already collect rich runtime information, such metadata is already available. For example, to group input bytes into a field, we need to track which input bytes are used in which branch(es). To do so, grey-box fuzzers like WEIZZ need to enumerate a number of distinct values for each input byte to generate a set of new inputs, execute the program

with these inputs, and then observe which operands of comparison instructions change (i.e., which branches are influenced by which input byte). This process can be expensive for large inputs and could introduce imprecision [96]. However, in concolic execution, dependencies between input bytes and conditional branches have already been precisely captured by path constraints, in a path-sensitive manner. In addition, the rich information contained in the path constraints also allows us to infer semantic types of fields and the hierarchy of nested chunks.

Byte Tagging

At each conditional branch, we parse symbolic expression of the branch condition to extract input bytes that control this branch, and tag each input byte with branch information. In our current design, we identify each branch as a tuple $\langle addr,\ ctx,\ order,\ direction \rangle$, where *addr* is the static address of the branch, *ctx* is the calling context, *order* is the branch's position in the trace (e.g., 3 mean the 3rd symbolic branch), and *direction* is the branching target (taken, non-taken). Note that many concolic executors already maintain this mapping so as to find data-dependencies when constructing path constraints, but it is not being used to infer the input structure. In Figure 5.1, we omitted the *direction* to save space.

Tag Ranking

When the same input byte is used in different branch conditions, it will be tagged with multiple branches, which may lead to conflicted results (e.g., overlapping fields). To address this issue, we heuristically rank the branches and use the top-ranked branch for field identification. The priorities are (in a descending order): (1) branches that can be

used to infer the type of a field (see §5.3.2), (2) branches that test for equality or distinct, (3) branches that use fewer input bytes, and (4) branches that appear earlier in the trace. Figure 5.1a shows the top tag for each tagged input byte. Note that `orderX` in `Tag2` is to simply represent that those bytes are tagged with different but contiguous `order`.

Byte Grouping

In general, a field is either an individual unit that is assessed as a whole, or in the form of an array (e.g., a string is an array of characters), whose element is assessed separately. For the former case, we simply group contiguous input bytes tagged with the same branch (i.e., the same pair of $\langle addr,\ ctx \rangle$) into one field. Note that branches here are context-sensitive so that we can distinguish fields handled by the same function but under different calling contexts. For the array case, we expect each element to be assessed consecutively. Therefore, we group contiguous bytes into one field if their tags have consecutive *order* and the same ctx. During group, we allow skipping a single byte without any tag, which could happen when not all bits in a field is used (e.g., a flag field). However, contiguous bytes without tag will be grouped into a special `unused` field. Figure 5.1a shows the resultant fields after byte grouping. In particular, `Field3` is from contiguous tags, `Field5` is an `unused` one.

Field Grouping

After determining the boundaries of fields, we leverage the calling context ctx to recover the input hierarchy in a bottom-to-up manner. Recall that a unique calling context ctx corresponds a path in the dynamic call graph. To recover the input hierarchy, (1) we

clone the dynamic call graph such that in the new graph, each chunk node represents a unique context, and chunks that share the same parent node share the same calling context prefix. (2) we create nodes for fields and create an edge between a chunk and a field if they share the same context. (3) we recursively split chunk nodes to make sure all fields connected to them are *consecutive* in the input. (4) we connect `unused` fields (i.e., fields without context) to the chunk immediately before it. (5) we merge a chunk node with its parent if it has no sibling node and remove chunk node that does not contain any field. Figure 5.1a shows the resultant two chunks after field grouping. In particular, `Chunk1` includes `Field3` to `Field6`, and `Chunk2` includes `Field1`, `Field2`, and `Chunk1`.

5.3.2 Field Type

In addition to recovering the input structure, we also try to infer the semantic types of fields. These semantic types impose important constraints that need to be included when constructing the path constraints. Notably, some grey-box fuzzers like ProFuzz [184] also try to recover semantics of bytes. The main difference is that instead of leveraging indirect feedback like the coverage bitmap, we directly use the rich runtime information contained in symbolic path constraints.

Checksum

The first type that we aim to identify is checksum, which is a well known challenge for automated testing [126, 134, 161]. We use the following patterns to identify checksum checks: (1) the conditional branch checks for equality or distinct; (2) one operand of the comparison is a direct copy from the input (based on the endian); (3) the other operand is

derived from a sequence of logical or arithmetic calculations, or the output of well-known checksum functions (e.g., `crc32`); and (4) the two operands involve disjoint sets of input bytes. We mark such directly copied bytes in one operand as a `checksum` field, and record the concrete value of the other operand as the desired checksum. Meanwhile, we also mark input bytes involved in the checksum calculation. In Figure 5.1a, `Field6` is a special `checksum` field, where bytes from offset 51 to offset 65 are involved in the checksum calculation (via `crc32`).

Delimiter-Ending

Many input formats, especially string-based like JSON use variable-sized fields, whose ending are marked by specific delimiters. To parse such inputs, the program usually contain a conditional branch that compares each element of the field against the delimiter to decide whether the field ends or not. In consequence, negating such a branch will shrink or expand the current field, thus affect the consistency of subsequent fields. To identify this type of fields, we search the path constraints for following patterns: (1) a sequence of contiguous input bytes are tagged by the same *addr* and *ctx*, but different *order*; (2) the conditional branch compares the input with a constant value; and (3) all the previous branching directions are the same except the last one. When finding such bytes, we mark them as a `delimiter-ending` field, and record the concrete value as the delimiter. Notably, `Field3` in Figure 5.1a is such a field ending with the delimiter '\0'.

String

String field is a special type delimiter-end field. We treat string field different because they are frequently used in string operations like `strcmp` and `strlen` that could be hard to express and solve as bitvector constraints. For this reason, we try to infer string field so we can leverage string theory in SMT solvers or customer solver to speed up the solving. In our current design, we identify string fields when it is used in well-known string operations like `strcmp`. We leave the inference of inlined or custom string operations for future work. Note that `Field2` in Figure 5.1a is a `string` field that we obtain from the function `png_memcmp`, as shown in Figure 5.1b.

Enumeration

Some fields can only have multiple valid values (e.g., `chunk_name` in PNG file), and each value leads to a distinct type of the corresponding chunk. When mutating such a field to negate a branch, leaving other parts of the chunk intact will lead to inconsistency. To mitigate this issue, we aim to identify this type of fields and apply custom solving. We use the following patterns to identify `enumeration` field: (1) the same field is checked by a sequence of different branches under the same context; and (2) each branch checks for equal or distinct against a different concrete values. In case the concolic execution is performed on languages that contain `switch` statements (e.g., LLVM IR), a field used in a `switch` statement is also marked as an `enumeration` field. Once we identified an `enumeration` field, we record all the values of the concrete operands as candidate values. Meanwhile, we find the parent chunk of the enumeration field according to the recovered field hierarchy and

associate the chunk with the enumeration. In Figure 5.1a, `Field2` is an `enumeration` field, as it is compared against various chunk names.

Size

Besides delimiter-ending, another popular way to implement variable-length field-/chunk is to use a dedicated `size` field, which indicates how many elements are in the corresponding field/chunk. We identify `size` field based on the observation that this field usually determines how many times a loop will be executed, where each iteration will access some input bytes. Based on this observation, we first find loop controlling branches during concolic execution, then search for following patterns: (1) a sequence of the same loop controlling branch (i.e., the same *addr* and *ctx*, but different *order*); (2) these branches all compare a symbolic value with a concrete one, where the symbolic operand is controlled by the same filed; (3) one operand of the comparison is fixed while the other operand's concrete value is changed by a fixed step between two occurrences of the branch; (4) the branching direction only changes on the last occurrence of the branch. Once we find such a sequence, we mark the filed in (2) as a `size` field, and the fixed step in (3) as the loop step. Meanwhile, we also associate fields/chunks involved in path constraints that between the first and last occurrences of the loop controlling branch with the size field. In addition, we also leverage well-known library functions that accept a size argument to identify size field, such as `fread` and `memcpy`. In Figure 5.1, `Field1` is found to be a `size` field, as it is used to invoke `memcpy`.

Offset

As shown in Listing 1, another important type of field is `offset`, which controls what input bytes will be used in future path constraints. We use two rules to identify `offset` fields. (1) If a field is used in pointer-arithmetic (e.g., `GetElementPtr` instruction in LLVM). (2) If a field is used in `lseek`-like functions that adjust the current offset of the input file. Notably, recent work like TensileFuzz [101] also uses the second rule. However, this rule alone cannot handle common cases where the input file is read as a whole into memory, then parsed later. The first rule allows us to overcome this limitation.

Unused

For contiguous input bytes that are not tagged with branches, we group into a `unused` field. Although these bytes are not used by any conditional branches, they may still affect the program states but the dependencies could be lost during concolic execution (e.g., due to lack of support of floating-point operations or incomplete modeling of external library calls). To overcome this limitation, we apply fuzzing-like mutations to such fields. In Figure 5.1a, `Field5` is an `unused` field.

5.4 Format-Aware Solving

After recovering the input format including field boundary, hierarchy, and field semantics, we utilize the format information to improve the input generation of CE. Our hypothesis is that, *if a newly generated test input is well-formatted, then it is more likely to avoid early path divergence, be able to negate the target branch, and reach a deep path.*

To achieve this goal, we focus on preserving two types of format constraints: (1) structure consistency and (2) dependencies imposed by semantic types.

5.4.1 Structure-Aware Solving

When negating a branch, we aim to make the newly generated input follow the input structure (i.e., field boundary and hierarchy), without breaking its integrity. To this end, we develop two new solving strategies: field-driven solving and chunk cache. The first strategy aims to preserve the tags we used to infer the structural information, while the later aims to reuse known-good solutions.

Field-Driven Solving

Recall that we infer structure information based on tagging the input bytes with the tuple $\langle addr,\ ctx,\ order,\ direction \rangle$. Therefore, as long as the tag relationships among the input bytes do not change (e.g., they still share the same pair of $\langle addr,\ ctx \rangle$), the inferred input structure (i.e., boundaries and hierarchy) will also not change. Based on this observation, we select path constraints as follows. We first apply optimistic solving to ensure that the target branch can be negated without any additional constraints. Next, we find direct data-dependencies using the dependency forest maintained by nested solving. Note that here we do not try to include all bytes in dependent fields because unmodified bytes that do not have direct data-dependency with the target branch are unlikely to break the metadata consistency. For implicit dependency, we will include them in our type-driven solving (§5.4.2). Once we have identified input bytes that need to be mutated together, instead of adding all related constraints, we *only* add the top-ranked constraints (i.e., the

one used to infer the structure §5.3.1) to the path constraints. Because each input byte has at most one constraint, the constructed path constraints are less likely to be over-constrained. At the same time, because the constraints used to derive structural metadata is preserved, the generated input is also likely to preserve the input structure.

Chunk Caching

The core idea of chunk caching is similar to caching solutions returned by solvers. The main difference, however, is that instead of reusing satisfying assignments to individual input bytes, we will reuse a whole *chunk* of input bytes (based on the inferred structural information) that satisfy the path constraints. This is based on the insight that reusing a whole chunk is more likely to preserve the input structure. Furthermore, as shown in Listing 2, negating a branch may lead the execution to a path where a different chunk containing different fields is expected. Hence, by reusing a whole chunk, these constraints will be readily satisfies, without going through a lengthy and fragile chain of inputs.

To this end, we maintain a global mapping that records an input chunk that satisfy a conditional branch. During input generation, we simply replace the existing chunk with the saved one that satisfies the negated constraints. It is worthwhile mentioning that there may exist multiple chunks corresponds to the same branch, we decide to save the minimal one for simplicity and efficiency.

5.4.2 Type-Aware Solving

As discussed in §2.2.2, one limitation of existing solving strategies like nested solving is that it only considers direct data-dependencies but ignores implicit dependencies,

therefore a satisfying solution could still fail to negate the target branch. We mitigate this problem by considering constraints imposed by *field semantics*. Specifically, for each solution generated by format-aware solving (§5.4.1), if we find an involved field has a recognized type, we will adjust the solution accordingly.

Checksum

A checksum check compares one symbolic operand to another symbolic operand. Because one operand is usually very complex, solving such constraints with SMT solvers is not efficient. Therefore, we leverage a simple heuristic to generate the solution: replace an identified `checksum` field with the concrete checksum calculated from the corresponding bytes [161].

Delimiter-Ending

A `delimiter-ending` field must end with a specific constant value. Therefore, if the last byte of a delimiter-ending field changes to a different value (i.e., the field grows), an additional delimiter is appended to the end of the field, and the next field is shifted accordingly. If a byte prior to the existing delimiter becomes a delimiter (i.e., the field shrinks), other bytes after the new delimiter will be removed, and the next field is also shifted.

String

We apply custom solving for certain constraints over string fields. Specifically, when a string field is used to invoke a `strcmp` family function with a concrete string, we use

the concrete string as the solution. We plan to include support for other string constraints in the future work. Moreover, when the size of a string field changed, we will shift subsequent bytes after the field accordingly to generate a consistent solution.

Enumeration

We use custom solving for mutating enumeration field. First, we only choose known-good values identified during CE. Second and more importantly, every enumeration is likely to correspond to a specific type of chunk (e.g., an `IHDR` chunk in PNG files), so simply change the field without adjusting the rest of the chunk is likely to result in invalid inputs. Similar to chunk caching (§5.4.1), in order to preserve other consistency constraints, we maintain a global table that maps each enumeration to a minimum valid chunk; then when using a new enumeration, we replace the whole chunk with a known-good one.

Size

When a `size` field is modified to a different value, we have to adjust the corresponding field to maintain the consistency. In more detail, for symbolic loop counter, we try to infer how many bytes are accessed in each loop iteration (§5.3.2), and adjust the corresponding the field accordingly by adding or deleting bytes. For `size` argument in library calls like `fread`, we use the extracted symbolic expression to calculate the new size and adjust the corresponding field accordingly.

Offset

An `offset` field determines where a chunk starts (i.e., where to start reading the input bytes). To keep such implicit data dependencies, we first attempt to not change any offset field when negating branches. However, if an offset field has to be changed, we adjust the input bytes to access accordingly. If the `offset` becomes smaller, we will shrink the chunk before the corresponding field by deleting bytes from the old offset. If the `offset` becomes larger, we will insert random bytes at the old offset to make sure that the same bytes will be accessed at the new offset. Notably, this actually will move all bytes after the old offset thus may modify the whole input dramatically.

Unused

These fields are not used in path constraints, hence will never be modified by a concolic executor. To explore potential control- and data-dependencies, we uniformly sample one field and perform deterministic mutations that flip bits one by one to generate new inputs.

5.5 Implementation

To demonstrate the effectiveness of our approaches, we develop a prototype namely FORMATLY on top of SYMSAN [30], a state-of-the-art concolic executor based on the data-flow sanitizer of LLVM. The architecture is shown in Figure 5.2. It mainly consists of three important components: the exploring engine, the analyzing engine, and the grading engine.

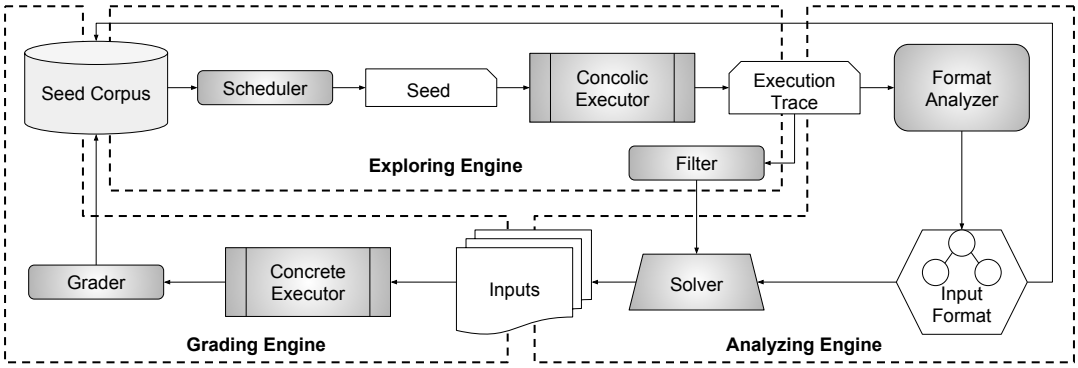


Figure 5.2: Overview of FORMATLY

- **Exploring Engine.** This is to select a seed from the corpus, feed it into the program under test to launch the concolic execution, collect the execution trace including path constraints, and decide which branches to negate next. In particular, we implement a simple FIFO scheduler that fetches fresh seeds, which have not been tracked yet, following the order they are added into the corpus. And we utilize the branch filter from QSYM [187] to decide whether a branch will be negated. Notably, we further adopt the data-flow sanitizer in SYMSAN to collect runtime information of conditional branches and specific function calls for format inference during the execution.

- **Analyzing Engine.** This is to analyze the execution trace collected during the concolic execution to generate the format of the current input as well as new inputs. Specifically, first it infers the input format as described in Section 5.3. Secondly, it queries the SMT solver Z3 for solutions of path constraints when negating branches. And it further utilizes the recovered input format to improve the solutions as described in Section 5.4. Eventually, new inputs are generated based on these solutions. It is worthwhile mentioning that the timeout set for Z3 is 10 seconds, which is the same setting used by other concolic execution engines [130, 131, 187].

- **Grading Engine.** This is to grade those newly generated inputs regarding the code coverage they have exercised. Each input leading to new code coverage will be added into the corpus as a new seed for further testing. To this end, we utilize the input grader from ANGORA [32] that adopts a calling-context sensitive edge metric to measure code coverage.

5.6 Evaluation

We evaluate FORMATLY on a set of real-world programs aiming to answer the following research questions:

- **RQ1.** Can format-aware solving improve the performance of branch negating?
- **RQ2.** Can format-aware solving lead to more code coverage in CE?
- **RQ3.** Can format-aware solving improve the performance of end-to-end hybrid fuzzing?

5.6.1 Experiment Setup

Dataset We evaluate 16 real world programs from Google FuzzBench [70], as shown in Table 5.1. For better reproducibility, we use the corpus from SYMSAN[1] in CE testing. To save time while not losing statistic significance, we randomly pick $n = max(N/10,\ 50)$ inputs from the corpus for each program, where N is the number of total seeds in the corpus. For end-to-end fuzzing, we used seeds offered by FuzzBench.

Computing Resources All the experiments are conducted on a 64-bit workstation with 48 cores (2 Intel(R) Xeon(R) Platinum 8260 @2.40GHz), 1.5TB of RAM, and Ubuntu 18.04.

[1] `https://jigsaw.cs.ucr.edu/seeds.tar.gz`

5.6.2 CE Testing

To answer RQ1 and RQ2, we conduct CE testing where FORMATLY is set to negate all symbolic branches along execution traces of inputs in the corpus. As argued in [131], this setting removes the variables from different corpus and different branch scheduling and filtering strategies, so the results can better reflect the performance of different strategies in path constraints construction.

Table 5.1: Results of branch negating.

Program	Input	Branch	Base	Fmt	Struct	Type	F-Deep	S-Deep	T-Deep
curl	135	2877	2349	2396 (2%)	2390 (2%)	2392 (2%)	210	180	132
freetype	132	30843	16492	20312 (23%)	20160 (22%)	17829 (8%)	5223	4884	1045
harfbuzz	296	12359	2756	3759 (36%)	3711 (35%)	3168 (15%)	1173	992	606
json	50	7996	460	599 (30%)	599 (30%)	460 (0%)	143	133	31
lcms	50	633	370	458 (24%)	455 (23%)	391 (6%)	227	184	54
libjpeg	85	15052	10011	11090 (11%)	10985 (10%)	10283 (3%)	2720	2399	1032
libpng	50	1115	388	581 (50%)	473 (22%)	513 (32%)	178	66	129
mbedtls	50	1972	903	1188 (32%)	1174 (30%)	1131 (25%)	124	106	56
openssl	158	10174	3956	5375 (36%)	5031 (27%)	4824 (22%)	646	499	369
openthread	50	995	497	633 (27%)	630 (27%)	606 (22%)	99	94	45
proj	78	2367	1139	1307 (15%)	1299 (14%)	1153 (1%)	316	297	29
re2	108	7205	1935	2681 (39%)	2645 (37%)	2017 (4%)	703	600	156
sqlite	522	27978	709	1580 (123%)	1563 (120%)	1039 (47%)	430	363	226
vorbis	50	4755	1774	2399 (35%)	2373 (34%)	1928 (9%)	968	663	580
woff2	52	1223	288	544 (89%)	529 (84%)	469 (63%)	65	46	53
xml	196	65885	40874	43458 (6%)	43409 (6%)	41675 (2%)	9026	8438	1589

RQ1. Performance in Branch Negating

As discussed in §5.4, our hypothesis is that format-aware solving is expected to generate better inputs that can avoid early path divergences and succeed in negating the target branches. To verify this, we measure the number of symbolic branches that the state-of-the-art nested-then-optimistic solving (§2.2.2) failed to negate, but our format-aware solving succeeded. Besides, we also hypothesized that input generated with format-aware

solving are can reach deeper paths after a successful negation. To verify this, we measure two metrics as an approximation to the real execution depth: (1) the number of executed symbolic branches after a successful negation and (2) the new branch coverage after the successful negation.

Table 5.1 shows the summary of branch negating results. Note that *Input* is the number of tested inputs, *Branch* is the number of symbolic branches that we tried to negate, *Base* is the number of symbolic branches the baseline strategy successfully negated, *Fmt* (*F* for short) is the number of branches FORMATLY successfully negated, and the improvements over baseline. *F-Deep* is the number of branches that both FORMATLY and the baseline negate, but FORMATLY-generated inputs visit *deeper* paths, *Strut* (*S* for short) and *Type* (*T* for short) break down the improvements by structural- and type-aware solving. First of all, we can observe that a considerable portion of inputs generated by the baseline strategy failed to negate the target branch. To confirm this problem is general and not a specific issue of SYMSAN, we have tested other concolic executors and the results (in the supplementary materials) are similar. Among the failed cases, 50.18% are caused by over-constrained (i.e., the solver returns `unsat` or time-out), 49.82% are caused by under-constrained.

With format-aware solutions, at least 10% more symbolic branches can be successfully negated on 14 out of 16 programs, and at least 30% more symbolic branches can be negated on about nine programs. In addition, among symbolic branches that the baseline strategy successfully negate, there are at least 10% and 20% ones that format-aware solutions lead to deeper new paths, on 15 and 12 out of 16 programs, respectively. It is worthwhile mentioning that results where *Fmt-Deeper* shows higher numbers than *Fmt* indicate fixing

path divergence and negating branches in general is more challenging than leading to deeper paths.

Contribution breakdown. Table 5.1 also shows the contribution of structure-aware solving (§5.4.1) and type-aware solving (§5.4.2). We can observe that both solving strategies contributed uniquely to negating more symbolic branches, and to deeper new paths, which indicates that both types of format information are helpful. Structure information generally contributes more to the improvements. We think this is because that (1) currently we only infer a limited set of semantic types, and (2) it is harder to infer and utilize type information precisely. However, on specific programs like `libpng` where various types like `size`, `checksum`, `keyword` are recognized, type information becomes more helpful.

The answer to RQ1 is yes. Inputs generated by format-aware solving can negate more target branches, and lead to deeper new paths.

RQ2. Effectiveness on Code Coverage

One of the most popular application scenario of concolic execution is coverage-guided test generation, where the primary goal is to cover more code. Therefore, we also evaluate how much format-aware solving can improve in term of code coverage. We use the tool `SanitizerCoverage` to measure the new edges covered by inputs generated during CE. Specifically, for each input in the corpus, we measure (1) Cov_{init}: its initial edge coverage, (2) Cov_{base}: the new edge coverage (over Cov_{init}) from the input generated by the baseline strategy, and (3) Cov_{fmt}: the new edge coverage (over Cov_{base} and Cov_{init}) from the input generated by format-aware solving. We then calculate the improvement over the baseline as

$Inc = |Cov_{fmt}| \div |Cov_{base}|$. After getting the per input improvement, we calculate the mean values within a 95% confidence, and show the statistical results in Table 5.2, where *Init* is the mean of edges exercised by each input in the corpus, *Base* is the mean of *new* edges exercised by inputs generated by the baseline solving strategy, *Inc* is the mean improvement of format-aware solving over the baseline. Note that all mean values are presented within a 95% confidence, *Inc-Struct* and *Inc-Type* break down the improvement by structure-aware and type-aware solving.

Table 5.2: Edge coverage improvement per input.

Program	**Init**	**Base**	**Inc (%)**	**Inc-Struct (%)**	**Inc-Type (%)**
curl	6087 ± 186	1565 ± 72	62 ± 10	30 ± 8	61 ± 10
freetype	2898 ± 144	1715 ± 126	52 ± 16	51 ± 16	7 ± 3
harfbuzz	2567 ± 60	963 ± 36	53 ± 5	52 ± 5	4 ± 1
json	388 ± 15	68 ± 13	471 ± 247	462 ± 248	16 ± 10
lcms	847 ± 123	182 ± 54	60 ± 53	57 ± 53	6 ± 3
libjpeg	1057 ± 81	508 ± 44	16 ± 3	13 ± 3	10 ± 3
libpng	670 ± 36	246 ± 19	31 ± 9	26 ± 8	13 ± 7
mbedtls	2683 ± 84	358 ± 68	11 ± 6	10 ± 6	9 ± 5
openssl	3157 ± 117	286 ± 26	472 ± 65	469 ± 65	5 ± 1
openthread	4323 ± 52	308 ± 50	7 ± 6	7 ± 6	1 ± 1
proj	495 ± 25	180 ± 26	314 ± 70	300 ± 71	46 ± 24
re2	1571 ± 156	911 ± 127	160 ± 35	159 ± 35	6 ± 5
sqlite	8154 ± 295	186 ± 43	17395 ± 2120	17393 ± 2120	23 ± 15
vorbis	757 ± 105	91 ± 15	48 ± 29	46 ± 29	9 ± 6
woff2	1257 ± 140	178 ± 26	26 ± 9	24 ± 9	20 ± 9
xml	1705 ± 47	677 ± 31	310 ± 21	294 ± 21	122 ± 18

We can observe from Table 5.2 that format-aware solving leads to significant improvements for almost all programs. It achieved at least 50% higher edge coverage over more than half of the programs, and achieved at least 100% higher edge coverage over six programs. In `sqlite`, FORMATLY even have orders of magnitudes higher edge coverage than the baseline. Considering that each newly generated input only aims to negate a single target branch, the significant improvement on edge coverage indicates that negating key

symbolic branches or leading to specific deep paths can make a huge breakthrough in terms of code coverage.

Contribution breakdown. We also evaluate how much structure and type information contribute to the improvement. In general, both types of format information contribute uniquely. However, similar to branch negation, structure information contributes more.

The answer to RQ2 is yes. Format-aware solving can lead to significant improvement on code coverage.

5.6.3 End-to-End Fuzzing

To answer RQ3, we conduct the evaluation on end-to-end fuzzing. We use three configurations, (1) AFL++ (commit 8fc249 with the default build and fuzz options) solely, (2) AFL++ paired with FORMATLY via periodical seed synchronization, named as FORMATLY for short, and (3) AFL++ paired with FORMATLY that discards the format inference and format-aware solving, namely UNFORMATLY.

Figure 5.3 shows the coverage growth curves. We observe a mix results. On `mbedtls`, `libpng`, `lcms`, `freetype`, FORMATLY significantly outperformed AFL++ and UNFORMATLY. Note that we fail to compile `mbedtls` with AFL++, so we can only compare FORMATLY and UNFORMATLY without AFL++. On `sqlite` and `xml`, AFL++ by itself performed better than FORMATLY and UNFORMATLY, though FORMATLY still performed better than UNFORMATLY. This indicates that the potential improvements led by CE cannot pay for the additional computing resources it takes. On `harfuzz`, FORMATLY performs

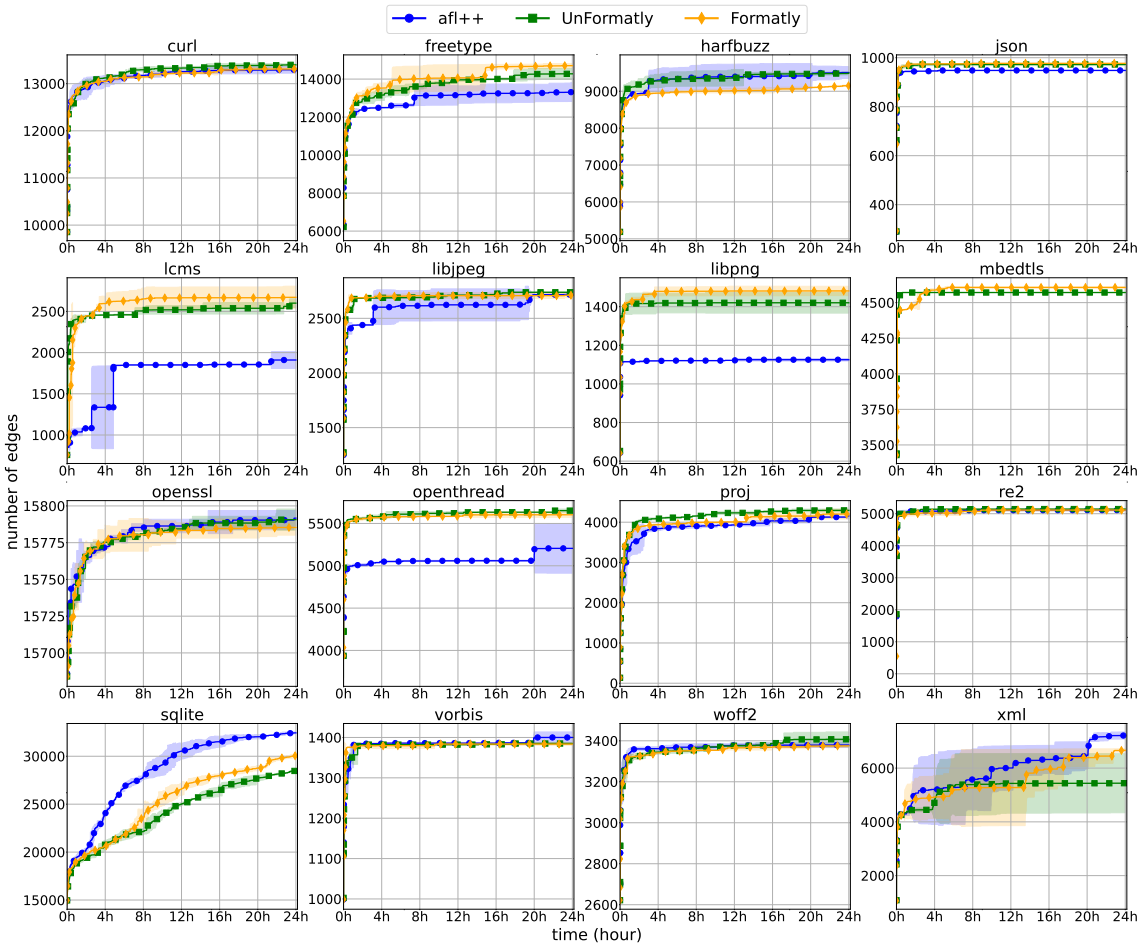


Figure 5.3: Edge coverage growth over time for hybrid fuzzing.

significantly worse than UnFormatly. Our investigation showed that Formatly processed significantly fewer seeds. This indicates that the format inference and format-aware solving introduced too much overhead on these programs, thus decreasing the throughput and affecting the overall performance. On the rest 9 programs, there is no significant difference among all three configurations.

The answer to RQ3 is generally positive: format-aware solving can help cover more code in hybrid fuzzing; however, the overhead caused by format inference and format-aware solving demands a more efficient collaboration design than simple seed synchronization.

5.6.4 Threats of Validity

There are three major threats to the validity of our evaluation. First, although we tried to use a large and diverse set of popular real-world programs, many of which are from the standard benchmark, the evaluation results may not reflect the actual performance improvements on new target programs. Second, the performance of a hybrid fuzzing campaign depends on many aspects, including randomness. Although we have fixed many factors (e.g., the initial corpus), performed each experiments several times, and used statistic tools, a new round of fuzzing campaign can still generate different results. Finally, although we have used many regression tests, our prototype implementation could still have bugs that can affect the evaluation results

5.7 Summary

In this chapter we showed that path constraints constructed by state-of-the-art concolic executors can be over-constrained and under-constrained. Consequently, a considerable portion of generated inputs fail to negate target symbolic branches. To address this problem, we propose a novel format-aware approach to generate inputs, which infers input structures and semantics based on path constraints, then leverages the inferred format information to guide the input generation.

Chapter 6

Conclusion

6.1 Summary

Greybox fuzzing and concolic execution are two popular approaches of automated test generation. In this dissertation, we sought to advance the state-of-the-art on automated test generation via improving the seed selection, scheduling, and generation which greybox fuzzing and concolic execution are built on top of.

First, we proposed the first systematic study on the the first systematic study on the impact of different coverage metrics in greybox fuzzing. To this end, we formally defined and discussed the concept of *sensitivity*, which can be used to theoretically compare different coverage metrics. We then presented several coverage metrics based on the different levels of sensitivity. We conducted the study on these metrics with the DARPA CGC dataset, the LAVA-M dataset, and a set of real-world applications (a total of 221 binaries). The study has revealed that each coverage metric leads to find different set of vulnerabilities, indicating

there is no grand slam that can beat others. In addition, combing different coverage metrics through cross-seeds helps find more crashes and find them faster.

Second, we presented a hierarchical seed scheduler in order to address the seed explosion problem where the increased number of seeds that are selected by a sensitive coverage metric exceed the fuzzer's capability to schedule. To this end, we first designed a novel multi-level coverage metric where we cluster seeds selected by fine-grained metrics using coarse-grained metrics. Consequently, the seed pool can be organized into a multi-level tree where leaf nodes are real seeds and internal nodes are less sensitive coverage measurements. Next we designed a hierarchical seed scheduling algorithm to support the multi-level coverage metric based on the multi-armed bandits model (MAB). We implemented our approach as an extension to AFL and AFL++, and evaluated them on DARPA CGC and Google FuzzBench. The results showed that our approach not only can trigger more bugs and achieve higher code coverage, but also can achieve the same coverage faster than existing approaches.

Third and last, we showed that path constraints constructed by state-of-the-art concolic executors can be over-constrained and under-constrained. Consequently, a considerable portion of generated inputs fail to negate target symbolic branches. To address this problem, we proposed FORMATLY that generates input in a novel format-aware manner. In particular, FORMATLY first infers input structures and semantics based on path constraints that are augmented with rich runtime information. Afterwards, it leverages the inferred format information to guide the path constraint constructing and solving in order to generate better inputs. Results of the evaluating FORMATLY on real-world programs showd

that it can significantly improve the performance of input generation, which enables better performance in end-to-end fuzzing.

6.2 Discussion

Application-Aware Coverage Metric Selection and Resource Allocation. In Figure 3.1, we can see the presented coverage metrics are not in a total order in terms of sensitivity. This means different coverage metrics have either unique strength in breaking through a specific pattern of code like loops. From the evaluation results presented in §3.3, we also observe that (1) there is no "grand slam" metric that beats all other metrics; and (2) even for metrics whose sensitivities are in total order (e.g., `bc, n2, n4, n8`), the most sensitive on is not always better. We explored a simple combination of them and allocated computing resources equally among them in §3. Because fuzzing can be modeled as a multi-armed bandit (MAB) problem [177] that aims to find more bugs with a limited time budget, previous work has shown how to improve the performance of fuzzing through adaptive mutation ratio [29]. Similarly, it might be possible to conduct static or dynamic analysis on each tested program to determine which coverage metric is more suitable. This decision may also change over time, so a resource allocation scheme might be useful to allocate computing resources among different coverage metrics dynamically.

Branch Scheduling for Concolic Execution While the hierarchical seed scheduler proposed in §4 targets on greybox fuzzing, it can be easily applied to CE that is also driven by concrete inputs. Furthermore, considering that CE generates new inputs via selectively negating branches, the scheduler can work in a more fine-grained manner that it

decides which branch will be negated next. To this end, we need to adapt the multi-level coverage metric accordingly. In particular, we can maintain a branch pool that is organized similarly into a multi-level tree where leaf nodes are symbolic branches that CE has exercised so far and intermediate nodes are coverage measurements. One opening problem here is whether we should distinguish the same branch, which can be identified using the tuple discussed in §5.3.1, that are exercised by different inputs. If the answer is yes, it indicates that the scheduler further needs to decide which input to pick. In addition, note that the reinforcement learning algorithm should be re-designed, since the scheduler in general should avoid picking a symbolic branch that has been negated successfully.

Collaboration of Greybox Fuzzing and Concolic Execution. Results of end-to-end fuzzing (see §5.6.3) suggests a more efficient collaboration design than simple seed synchronization for combining greybox fuzzing and concolic execution, as the potential improvements led by CE cannot pay for the additional computing resources it takes. One promising approach is to plugin a CE engine into a greybox fuzzing scheme as a special mutator. When solving normal branches, the fuzzing scheme utilizes the lightweight random mutator; but if the branch is found to be too hard, it switches to the CE engine to launch format-aware solving that is more heavyweight yet much more powerful. This approach can lead to several advantages. First, it avoids the overhead introduced by seed synchronization, which is not negligible according to [181]. Second, CE is launched only when necessary, thus the computing resources it takes is minimized. Third, as the seed scheduler owns a collaborative view on both the fuzzing (i.e., random mutator) side and the CE side, it is capable of striking a better balance between seed exploitation and exploration.

Bibliography

[1] Libfuzzer: a library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`.

[2] Rajeev Agrawal. Sample mean based index policies with o (log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, pages 1054–1078, 1995.

[3] Dave Aitel. An introduction to spike, the fuzzer creation kit. *presentation slides*, 1, 2002.

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[5] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[7] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[8] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[9] C. Barrett, P. Fontaine, and C. Tinelli. Sage2 benchmarks. `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/Sage2`, 2016.

[10] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification (CAV)*, pages 171–177. Springer, 2011.

[11] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[12] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[13] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. Grimoire: Synthesizing structure while fuzzing. In *USENIX Security Symposium (Security)*, 2019.

[14] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2020.

[15] Marcel Böhme, Valentin Manes, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2020.

[16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[18] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing symbolic expressions. In *International Conference on Software Engineering (ICSE)*, 2021.

[19] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzolic: mixing fuzzing and concolic execution. *Computers & Security*, page 102368, 2021.

[20] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *International Conference on Software Engineering (ICSE)*, 2013.

[21] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. Jvm fuzzing for jit-induced side-channel detection. In *International Conference on Software Engineering (ICSE)*, 2020.

[22] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *International Conference on Software Engineering (ICSE)*, 2020.

[23] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[24] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[25] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *International Conference on Software Engineering (ICSE)*, 2011.

[26] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[27] DARPA CGC. Darpa cyber grand challenge binaries. `https://github.com/CyberGrandChallenge`, 2014.

[28] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[29] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[30] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, and Chengyu Song. Symsan: Time and space efficient concolic execution via dynamic data-flow analysis. 2022.

[31] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2022.

[32] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[33] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing deeply nested branches. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[34] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, and Long Lu. Savior: towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[35] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *USENIX Security Symposium (Security)*, 2019.

[36] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[37] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. Intriguer: Field-level constraint solving for hybrid fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[38] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *International Conference on Software Engineering (ICSE)*, 2019.

[39] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. Memfuzz: Using memory accesses to guide fuzzing. In *IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019.

[40] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[41] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[42] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *USENIX Security Symposium (Security)*, 2015.

[43] S Dinesh, N Burow, D Xu, and M Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[44] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[45] M Eddington. Peach fuzzer. *url: http://www. peachfuzzer. com/(visited on 06/21/2018)*, 2008.

[46] Michael Eddington. Peach fuzzer platform. `http://www.peachfuzzer.com/products/peach-platform/`, 2011.

[47] Shawn Embleton, Sherri Sparks, and Ryan Cunningham. Sidewinder: An evolutionary guidance system for malicious input crafting. In *BlackHat*, 2006.

[48] Brandon Falk. How conditional branches work in vectorized emulation. `https://gamozolabs.github.io/fuzzing/2019/10/07/vectorized_emulation_condbranch.html`, 2018.

[49] Brandon Falk. Vectorized emulation: Hardware accelerated taint tracking at 2 trillion instructions per second. `https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html`, 2018.

[50] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. Weizz: Automatic grey-box fuzzing for structured binary formats. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[51] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[52] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, November 2022.

[53] Andreas Fröhlich, Armin Biere, Christoph M Wintersteiger, and Youssef Hamadi. Stochastic local search for satisfiability modulo theories. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2015.

[54] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *USENIX Security Symposium (Security)*, 2019.

[55] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[56] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009.

[57] Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *International Conference on Software Engineering (ICSE)*, 2020.

[58] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, 2015.

[59] Gregory Gay, Matt Staats, Michael Whalen, and Mats PE Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015.

[60] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[61] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[62] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[63] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated whitebox fuzz testing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[64] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[65] Google. Tcmalloc. `https://github.com/google/tcmalloc`.

[66] Google. honggfuzz. `https://github.com/google/honggfuzz`, 2010.

[67] Google. Fuzzing for security. `https://blog.chromium.org/2012/04/fuzzing-for-security.html`, 2012.

[68] Google. OSS-Fuzz - continuous fuzzing of open source software. `https://github.com/google/oss-fuzz`, 2016.

[69] Google. fuzzer-test-suite. `https://github.com/google/fuzzer-test-suite`, 2017.

[70] Google. Fuzzbench: Fuzzer benchmarking as a service. `https://google.github.io/fuzzbench/`, 2020.

[71] Rahul Gopinath and Andreas Zeller. Building fast fuzzers. *arXiv preprint arXiv:1911.07707*, 2019.

[72] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *USENIX Security Symposium (Security)*, 2013.

[73] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[74] Ahmad Hazimeh, ADRIAN HERRERA, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In *ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*, 2021.

[75] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[76] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, and Stephen McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*, 43(2):164–184, 2017.

[77] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: automatically learning the x86-64 instruction set. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[78] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[79] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering (ICSE)*. ACM, 2014.

[80] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. Fuzzgen: Automatic fuzzer generation. In *USENIX Security Symposium (Security)*, 2020.

[81] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. Tiff: using input type inference to improve fuzzing. In *Annual Computer Security Applications Conference (ACSAC)*, pages 505–517, 2018.

[82] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[83] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. Towards efficient heap overflow discovery. In *USENIX Security Symposium (Security)*, 2017.

[84] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using context-sensitive software fault injection. In *USENIX Security Symposium (Security)*, 2020.

[85] Ulf Kargén and Nahid Shahmehri. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[86] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening pandora's box through atfuzzer: dynamic analysis of at interface for android smartphones. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2019.

[87] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[88] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[89] lafintel. Circumventing fuzzing roadblocks with compiler transformations. `https://lafintel.wordpress.com/`, 2016.

[90] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54, 2012.

[91] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *USENIX Security Symposium (Security)*, 2020.

[92] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[93] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[94] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2017.

[95] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[96] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In *IEEE Symposium on Security and Privacy (Oakland)*, 2022.

[97] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[98] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2008.

[99] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. Fans: Fuzzing android native system services via automated interface analysis. In *USENIX Security Symposium (Security)*, 2020.

[100] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. A comparative study of incremental constraint solving approaches in symbolic execution. In *Haifa Verification Conference*, 2014.

[101] Xuwei Liu, Wei You, Zhuo Zhang, and Xiangyu Zhang. Tensilefuzz: facilitating seed input generation in fuzzing via string constraint solving. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 391–403, 2022.

[102] LLVM. The LLVM compiler infrastructure project. `llvm.org`, 2018.

[103] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[104] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *USENIX Security Symposium (Security)*, 2019.

[105] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. Ems: History-driven mutation for coverage-based fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2022.

[106] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, 2011.

[107] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *International Conference on Software Engineering (ICSE)*, 2007.

[108] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *International Conference on Software Engineering (ICSE)*, 2020.

[109] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. Compiler fuzzing: how much does it matter? In *Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019.

[110] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. Parser-directed fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 548–560, 2019.

[111] Björn Mathis, Rahul Gopinath, and Andreas Zeller. Learning input tokens for effective fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 27–37, 2020.

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

[113] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[114] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[115] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *International Conference on Software Engineering (ICSE)*, 2020.

[116] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. Hydiff: Hybrid differential software analysis. In *International Conference on Software Engineering (ICSE)*, 2020.

[117] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *USENIX Security Symposium (Security)*, 2020.

[118] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *USENIX Security Symposium (Security)*, 2020.

[119] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[120] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: domain-specific fuzzing with waypoints. In *Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019.

[121] Brian S Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *In Master's thesis, School of Computer Science Carnegie Mellon University*, 2012.

[122] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *Proceedings of the 26th USENIX Security Symposium*. USENIX, 2017.

[123] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. Deferred concretization in symbolic execution via fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[124] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[125] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing usb drivers by device emulation. In *USENIX Security Symposium (Security)*, 2020.

[126] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[127] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[128] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *IEEE International Conference on Software Testing, Verification and Validation (Testing Tools Track)*, 2020.

[129] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[130] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don‘t interpret, compile! In *USENIX Security Symposium (Security)*, 2020.

[131] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[132] LLVM Project. LLVM language reference manual. /url-https://llvm.org/docs/LangRef.html.

[133] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[134] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[135] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium (Security)*, 2014.

[136] Jesse Ruderman. Introducing jsfunfuzz. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/`, 2007.

[137] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-cube: High-dimensional hypervisor fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[138] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium (Security)*, 2017.

[139] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[140] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[141] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *IEEE Cybersecurity Development (SecDev)*. IEEE, 2016.

[142] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[143] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[144] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[145] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[146] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[147] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. *Information Systems Security*, pages 1–25, 2008.

[148] Suhwan Song, Chengyu Song, Yeongjin Jang, and Byoungyoung Lee. Crfuzz: Fuzzing multi-purpose programs through input validation. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2020.

[149] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[150] László Szekeres. *Memory corruption mitigation via software hardening and bug-finding*. PhD thesis, Stony Brook University, 2017.

[151] the Clang team. Dataflowsanitizer design document. `https://clang.llvm.org/docs/DataFlowSanitizerDesign.html`, 2018.

[152] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

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

[154] Dmitry Vyukov. Syzkaller: an unsupervised, coverage-guided kernel fuzzer, 2019.

[155] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *International Conference on Software Engineering (ICSE)*, 2020.

[156] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[157] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. 2021.

[158] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2017.

[159] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *International Conference on Software Engineering (ICSE)*, pages 724–735, 2019.

[160] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[161] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[162] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[163] Website. American fuzzy lop (afl) fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2018. Accessed: 2018-04.

[164] Website. Angr: a framework for analyzing binaries. `https://angr.io/`, 2018. Accessed: 2018-04.

[165] Website. honggfuzz. `http://honggfuzz.com/`, 2018. Accessed: 2018-04.

[166] Website. Intel pin tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`, 2018. Accessed: 2018-04.

[167] Website. The lava synthetic bug corpora. `https://moyix.blogspot.com/2016/10/the-lava-synthetic-bug-corpora.html/`, 2018. Accessed: 2018-04.

[168] Website. libfuzzer. `https://llvm.org/docs/LibFuzzer.html`, 2018. Accessed: 2018-04.

[169] Website. Of bugs and baselines. `https://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html`, 2018. Accessed: 2018-04.

[170] Website. Oss fuzz. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2018. Accessed: 2018-04.

[171] Website. Peach fuzzer. https://www.peach.tech/, 2018. Accessed: 2018-04.

[172] Website. Security @ adobe. https://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe.html, 2018. Accessed: 2018-04.

[173] Website. Trail of bits blog. https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again, 2018. Accessed: 2018-04.

[174] Website. Utilities for automated crash sample processing/analysis. https://github.com/rc0r/afl-utils, 2018. Accessed: 2018-04.

[175] Website. Zzuf: multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf, 2018. Accessed: 2018-04.

[176] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *International Conference on Software Engineering (ICSE)*, 2020.

[177] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[178] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *International Conference on Software Engineering (ICSE)*, 2020.

[179] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: A coverage-guided fuzz testing framework for deep neural networks. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[180] Wen Xu, Sanidhya Kashyap, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[181] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[182] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[183] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. Slf: fuzzing without valid seed inputs. In *International Conference on Software Engineering (ICSE)*, pages 712–723, 2019.

[184] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[185] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[186] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *USENIX Security Symposium (Security)*, 2020.

[187] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium (Security)*, 2018.

[188] Michal Zalewski. American fuzzy lop.(2014). `http://lcamtuf.coredump.cx/afl`, 2014.

[189] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[190] Meng Xu Sanidhya Kashyap Hanqing Zhao and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[191] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium (Security)*, 2019.

[192] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security Symposium (Security)*, 2020.