It's Not You, It's Me: Reevaluating the Relationship between Concolic Execution and Fuzzing

Lukas Dresel
UC Santa Barbara
Santa Barbara, USA
lukas.dresel@cs.ucsb.edu

Stijn Pletinckx
UC Santa Barbara
Santa Barbara, USA
stijn@ucsb.edu

Fabio Gritti
UC Santa Barbara
Santa Barbara, USA
degrigis@ucsb.edu

Mathias Payer
EPFL
Lausanne, Switzerland
mathias.payer@nebelwelt.net

Giovanni Vigna
UC Santa Barbara
Santa Barbara, USA
vigna@cs.ucsb.edu

Christopher Kruegel
UC Santa Barbara
Santa Barbara, USA
chris@cs.ucsb.edu

***Abstract*—Concolic execution is a well-established program analysis technique for discovering vulnerabilities in software systems. However, recent concolic executors have been primarily designed and optimized for hybrid fuzzing scenarios, where they complement fuzzers by solving constraints to reach program paths with complex conditions. This scenario has led to architectural decisions that prioritize integration with fuzzers over the inherent strengths of concolic execution itself, namely systematic path exploration in the presence of complex constraints. We argue that the design choices borrowed from fuzzing, such as shared coverage metrics and corpus management strategies, may actually limit the effectiveness of concolic execution when used independently.**

In this paper, we develop SYMCTS, a concolic executor that breaks from the hybrid fuzzing paradigm to revisit the potential of modern standalone concolic execution design. SYMCTS introduces a novel coverage metric, edge-dependence coverage, and scheduling algorithm that prioritizes under-explored regions of code. These design choices enable SYMCTS to efficiently explore complex program behaviors independently without relying on an accompanying fuzzer.

We evaluate SYMCTS against state-of-the-art concolic executors on well-known benchmark targets from the FuzzBench and Unibench benchmarks. Our results show that SYMCTS can significantly improve coverage when using our coverage metric compared to existing concolic execution approaches and our scheduler improves coverage reached when using an empty input corpus. Notably, standalone concolic execution using SYMCTS performs on-par with or even outperforms established greybox and hybrid fuzzers on a few targets, motivating further research into the capabilities of concolic execution itself.

***Index Terms*—concolic execution, symbolic execution, fuzzing, hybrid fuzzing, software testing**

1. Introduction

Finding software faults efficiently has been an active research area in the computer security community for many years. Discovering bugs before the software reaches production enables timely patches, prevents end users from being affected by vulnerabilities, and reduces the risk of security incidents that can lead to disastrous economic losses [55].

Researchers developed a plethora of techniques to automatically discover software bugs, including fuzzing [45], symbolic execution [5], bounded model checking [7], static analysis [20], and formal verification [36]. While every technique comes with its own benefits and limitations [61], the past decade of research has mainly focused on fuzzing, with some efforts to improve it through symbolic or concolic exploration. This combined approach is called *hybrid fuzzing*, a technique that couples concolic execution [30] and fuzzing [45] to analyze a target program.

Modern hybrid-fuzzing systems [17], [29], [56], [57] have demonstrated that they can successfully explore deeper program logic in a time span achievable by neither fuzzing nor concolic execution alone. However, while this technique seems to be the panacea for efficient program exploration, we observe that the community-accepted blueprint has a hidden caveat: To scale to real-world programs, *all* modern hybrid fuzzing systems are increasingly tailored towards (and dominated by) the design of the accompanying fuzzer. For instance, in LibAFL [27], the concolic execution component employs the same coverage feedback and scheduling logic used by (and optimized for) the fuzzer. This design entails several implicit limitations that, to our knowledge, have not been addressed in the literature before.

A notable issue is the contrasting throughput dynamics between the fuzzer and the concolic executor: Fuzzers operate quickly and generate substantial volumes of inputs, whereas concolic executors, due to their solver component, are considerably slower. The prolific generation of mutated inputs from fuzzing necessitates the adoption of coarse-grained, high-throughput coverage metrics to distinguish between pertinent and irrelevant inputs. Consequently, many generated inputs are dismissed, particularly those that merely traverse previously explored code regions (basic blocks). Intuitively, while this strategy

proves effective for fuzzer-generated inputs, it becomes suboptimal when applied to inputs generated by concolic executors: High-relevance inputs, for example, those that reach previously explored code but via a different path, are at risk of early dismissal as they may not meet the criteria set by fuzzing (e.g., if the fuzzer's metric only considers inputs discovering new code or edges). In turn, this can limit program exploration, thereby reducing the odds of finding interesting bugs.

In this paper, we argue that concolic executors designed for (and tailored to) hybrid fuzzing systems are fundamentally limited by a mismatch in the characteristics of the underlying components. Instead, we propose to revisit the design of *independent* concolic executors, using their own, more fine-grained, coverage metrics and scheduling mechanisms. In doing so, the concolic executor is able to to play to its strengths with a more precise understanding of the state space of a target program's inputs.

To support our observations, we design SYMCTS, a standalone concolic executor, with a novel coverage metric (edge-dependency coverage) and a specialized input-selection mechanism. We compare SYMCTS across various configurations and against a baseline concolic execution system used in hybrid fuzzing (SymCC), which our tool is based on, on the well-known UniBench benchmark. Our evaluation demonstrates that edge-dependence coverage significantly improves the concolic executor's ability to reach and discover new code. Interestingly, we also find that the under-explored branch scheduling improves reached coverage when used with an empty seed corpus but reduces in utility when used with larger input corpora. When equipped with an empty seed corpus, SYMCTS improves the reached coverage by a median of around 20% across all targets when compared to the baseline coverage metric and scheduling. On a saturated corpus, SYMCTS improves reached coverage by up to 9.7%, with a median of 2% newly reached edges across all targets. We also evaluate SYMCTS against a recent state-of-the-art concolic execution system, Marco (based on SymSan), and find that it improves reached coverage on 10/12 FuzzBench targets.

Finally, when evaluating SYMCTS against widely used grey-box and hybrid fuzzers on FuzzBench, we also found that, for some targets, standalone concolic execution with SYMCTS can outperform existing hybrid solutions, motivating further research into the capabilities of modern concolic executors.

In summary, we make the following contributions:

- We investigate state-of-the-art concolic executors used in hybrid fuzzing approaches and argue that the current practice of closely tailoring the concolic execution to a fuzzer feedback mechanism limits its utility.
- We propose a *standalone concolic execution* approach with independent coverage metrics and a scheduling approach optimized for the strengths of concolic execution. To this end, we implement a standalone concolic executor, called SYMCTS, that uses a novel fine-grained coverage metric, *edge-dependence coverage*, and custom input scheduling mechanism.
- We evaluated SYMCTS on the well-known UniBench benchmark in various configurations and demonstrate that our novel coverage metric significantly outperforms the ones commonly used in modern concolic executors for hybrid fuzzing whereas our scheduling logic improves exploration when starting from an empty input corpus.
- We evaluate SYMCTS against state-of-the-art concolic execution, grey-box fuzzing and hybrid fuzzing approaches in separate, long-running evaluations using FuzzBench. We find that SYMCTS can outperform concolic execution, and, for a few targets, even outperform fuzzers in terms of coverage reached.

2. Background

2.1. Fuzzing

Fuzzing executes a target program with repeatedly mutated inputs to discover crashes, bugs, and security vulnerabilities. The most popular fuzzing technique is coverage-guided ("grey-box") fuzzing, where a set of inputs form the *corpus*, which is periodically inspected by a *scheduler*. The scheduler selects some of the inputs to be modified by a series of *mutation* strategies, producing a set of mutant inputs. Finally, an *executor* continuously runs an instrumented version of the target program under a mutant input to extract *observations* (in most cases, a measure of code coverage). The observations are then used by a *feedback* mechanism to decide if the (mutant) input should be added to the *corpus* for further mutation. Fuzzers such as AFL++ [26], HonggFuzz [34], and LibFuzzer [1] have seen industry-wide deployment, including large-scale fuzzing campaigns to test widely used open-source software [35].

The effectiveness of fuzzers largely stems from their ability to rapidly test the target program with variants of existing inputs to discover new program behaviors. A fuzzer's relatively simple mutations (e.g., flipping a small number of bits) are often individually unlikely to cause behavior changes, and fuzzing relies on high-throughput testing of such inputs to ensure timely exploration of program behaviors. However, if the input constraints are too complex, fuzzers generally fail to reach certain program areas when the feedback they receive is not granular enough to make incremental progress.

2.2. Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) is a dynamic testing strategy that uses symbolic variables to represent possible program inputs. A symbolic executor tracks the computations performed by the program to collect the constraints on symbolic variables and the symbolic values computed from them. When a conditional branch depends on symbolic values, the symbolic executor uses constraints to represent the conditions of the taken branch that must be satisfied to follow the given path. The collection of such constraints up to the current point in the program is called the *path constraint set*. Such path constraints are solved using an SMT solver [6], [24], which generates concrete

input solutions that can drive the execution of the program along a designated path. In theory, symbolic execution could explore *all* possible paths of a program. In practice, this is infeasible due to (1) path explosion [15] and (2) the increasing difficulty of solving constraints when reaching deeper states of the code, which quickly leads to resource exhaustion. Despite its limitations, symbolic execution remains a valuable tool in security testing and has been successfully employed in a range of applications.

2.3. Concolic Execution

Concolic Execution combines *conc*rete execution with symb*olic* execution. In particular, given a concrete input to a target program, concolic execution will symbolically trace the program's execution path and collect the path constraints associated with that input. The concolic executor then derives new inputs that deviate from the traced path by incrementally negating the path constraints and using an SMT-solver to produce satisfiable solutions. Concolic execution can also fall back to concrete values when symbolic handling of complex constraints is not possible. By re-executing the target with the given input to recover symbolic analysis results for one path, concolic execution avoids the overhead of maintaining multiple program states in memory for other program paths. In this regard, concolic execution more closely resembles a fuzzer and is also referred to as *white-box fuzzing* [31].

2.4. Hybrid Fuzzing

Lastly, hybrid fuzzing combines fuzzing and concolic execution to harness the advantages of both approaches. While well-designed concolic executors have been proven to be effective at discovering bugs [12] when deployed correctly, they require careful setup and configuration. They also complement fuzzing approaches by systematically exploring the state space, unlike fuzzing, which uses stochastic exploration.

As such, more recently, *hybrid fuzzing* [17], [56], [57], [63] combines fuzzing and concolic execution to overcome these hurdles. Specifically, in hybrid fuzzing, a fuzzer and a concolic executor run alongside each other and share newly discovered inputs back and forth. This approach promises to harness the benefits of both techniques: the fuzzer quickly covers easily reachable parts of the program, while the concolic executor provides inputs that reach areas that require complex reasoning. In its typical setup [61], the concolic executor attempts to discover new program paths by concolically mutating the inputs from the fuzzer's input corpus. These inputs are provided back to the fuzzer for coverage evaluation, and, *according to the fuzzer's coverage metrics*, "interesting" inputs are added to the fuzzer's corpus.

In theory, the strength of this approach lies in the synergy between the different mutation strategies employed by traditional fuzzers and concolic executors. However, we argue that, in practice, the currently popular approach of "coupling" the concolic executor to the fuzzer's coverage metric and scheduling logic limits the effectiveness of hybrid fuzzing. Specifically, fuzzers rely on very fast, broad mutations (e.g., splicing two inputs together) and require very coarse-grained coverage metrics (e.g., only keeping inputs that discover new branches) to handle the high throughput of inputs. Concolic execution, instead, carefully satisfies one constraint after another, leading to more precise mutations. This requires a larger number of intermediate inputs and mutations to reach a given program location. However, these additional inputs do not necessarily cover new branches or basic blocks, and as a result, the fuzzer's coarse-grained coverage metrics often discard them prematurely.

3. Motivation

To better illustrate the limitations of existing analysis methodologies, we provide a code example in Listing 1, which is considered hard to explore with current program analysis techniques. The program is a simplified parser for Executable and Linkable Format (ELF) files. The code works as follows: First, the program checks whether the ELF header contains the magic bits of an ELF file (Line 2). Then, depending on whether the file is 32-bit or 64-bit (checked at Lines 7 and 11), one of two parsing methods (i.e., `parse_elf32` or `parse_elf64`) is used to parse the file (Lines 8, 12). Finally, after the section headers are validated (Line 18), the ELF file is disassembled and printed (Line 22). For the sake of this motivating example, we assume that in this last step (in `print_disasm`), there exists a vulnerability *only* when a 64-bit ELF file is provided (Line 29). In the following, we describe how existing program analysis techniques (described in Section 2) fail to detect this vulnerability effectively.

```
1  bool disass_elf(char *data, int size) {
2    if (memcmp(data, "\x7fELF", 4)) {
3        return false;
4    }
5    ELF* parsed = NULL;
6    bool is32bit = false;
7    if (data[4] == ELFCLASS32) {
8      parsed = parse_elf32(data, size);
9      is32bit = true;
10   }
11   else if (data[4] == ELFCLASS64) {
12     parsed = parse_elf64(data, size);
13   }
14   else {
15     return false;
16   }
17
18   if (!validate_section_headers(parsed)) {
19     return false;
20   }
21
22   print_disasm(is32bit, parsed->code);
23 }
24
25 void print_disasm(bool is32bit, char *code) {
26   if (is32bit) {
27     do_stuff();
28   } else {
29     bug();
30   }
31 }
```

Listing 1: Example of an ELF parser.

3.1. Fuzzing

A grey-box fuzzer, such as AFL++ [26], would quickly generate a large amount of different inputs for the arguments `data` and `size`. However, producing a fully validated ELF file requires the fuzzer to progressively synthesize multiple correct data structures. In fact, the fuzzer needs to generate an input that: (1) contains *all* the correct headers (i.e., `ELF header`, `Section headers`, `Program headers`), and (2) contains valid code to be disassembled.

Consider the header encoding for the ELF file's computing architecture (32-bit vs. 64-bit). Lines 7 and 11 check this encoding to determine which parsing function to invoke. The fuzzer will try many different inputs, including some that represent a valid 32-bit architecture and others that represent a valid 64-bit architecture. If the fuzzer generates a valid 32-bit ELF first, this will improve code coverage and reach the header validation (Line 18) and printing (Line 22) sections of the code example. Since the bug in our `print_disasm` function applies only to 64-bit architectures, the generated input containing a 32-bit ELF will not trigger the bug. While the fuzzer can still generate a valid 64-bit ELF later in the process (and, hence, successfully explore Line 12 of the program), the code beyond Line 17 will be considered "explored" by the coverage metric, leading the fuzzer to discard the 64-bit ELF input for further exploration. This is a core difficulty in fuzzing: the fuzzer essentially "locks on" to a particular program path, making it challenging to diverge from it. As such, if the fuzzer is "unlucky" and first generates a 32-bit ELF, the chances are slim that it will discover the bug on Line 29 with a 64-bit ELF, due to the coarse-grained coverage feedback. While improvements like Redqueen [3] allow the fuzzer to perform individual mutations more effectively (e.g., to produce valid section header constants), the coverage metric will not recognize the newly generated inputs as novel, discarding them all the same.

3.2. Symbolic Execution

While, in theory, the exhaustive path exploration of a symbolic executor can fully explore the program above, the exponential number of paths through the program makes this intractable in practice. Consider, for instance, the parsing of the section headers in an ELF file. The ELF file format specifies at least seven different section header types [22], and the number of headers present is controlled by the ELF header. Even if a given program has only three program headers, this would already yield at least 7^3 distinct paths to explore. It is worth noting that a real-world ELF parser is significantly more complex than the small example presented in Listing 1. Thus, the resulting explosion of paths will likely overwhelm symbolic execution systems, causing them to get stuck at Line 18.

3.3. Concolic Execution

Concolic execution attempts to trace and systematically negate path conditions for the current path to produce new inputs for further exploration. In our motivating example, a concolic executor would still have to deal with path explosion (both in the number of constraints per input and in the number of inputs themselves). Concolic execution, therefore, often uses heuristics to prioritize paths for exploration (e.g., SAGE [31] selects the input with the most unexplored coverage). However, the sheer number of paths discovered can still often overwhelm these systems, therefore drastically reducing the probability of finding our architecture-dependent example bug. As a consequence, modern concolic execution engines, especially those designed with hybrid fuzzing in mind, use coverage metrics similar to those used by fuzzers to prune the search space and reduce the total number of paths to consider.

Consequently, all inputs produced by the concolic executor must be considered "interesting" by the coverage metric to be considered for further mutation. Coarse-grained coverage metrics aggressively prune necessary stepping-stone inputs prematurely, preventing exploration of later stages with diverse sets of states.

Concretely, even if the concolic executor mutates an input in our motivating example to represent a valid 64-bit ELF, if edge-coverage is used to determine the interestingness of new inputs, this input will not cover any new branches in the program (because we explored it earlier with a 32-bit ELF). Thus, the resulting input will not be added to the corpus, preventing further mutations of this input that are necessary to trigger the architecture-dependent bug.

3.4. Hybrid Fuzzing

A hybrid fuzzing approach seems ideal for solving the shortcomings mentioned above. The concolic executor provides a variety of high-quality inputs (e.g., by mutating inputs for all possible program header types), which the fuzzer can use to fuzz the data structures encoded in them. However, modern hybrid fuzzing configurations generally either explicitly use the fuzzer's coverage metric to evaluate the inputs/paths produced by concolic execution [17], [56], [57], [63], or directly schedule inputs for symbolic mutation from the accompanying fuzzer's queue, directly tying the concolic executor to the fuzzer's coverage metric. This inhibits the complementary strength of path-based exploration that concolic execution offers, limiting the exploration of code regions with complex path constraints and dependencies.

4. Standalone Concolic Execution

Motivated by advancements in concolic execution for hybrid fuzzing, we propose revisiting *standalone concolic execution*. Our goal is to design and develop a fully independent concolic executor that eliminates misalignment between the performance characteristics of fuzzing and concolic mutation. This approach empowers the concolic executor to employ coverage metrics, seed scheduling, and corpus management approaches explicitly designed to optimize the concolic execution process. By emphasizing the unique strengths of concolic execution, we can substantially improve the exploration of code with complex path conditions.

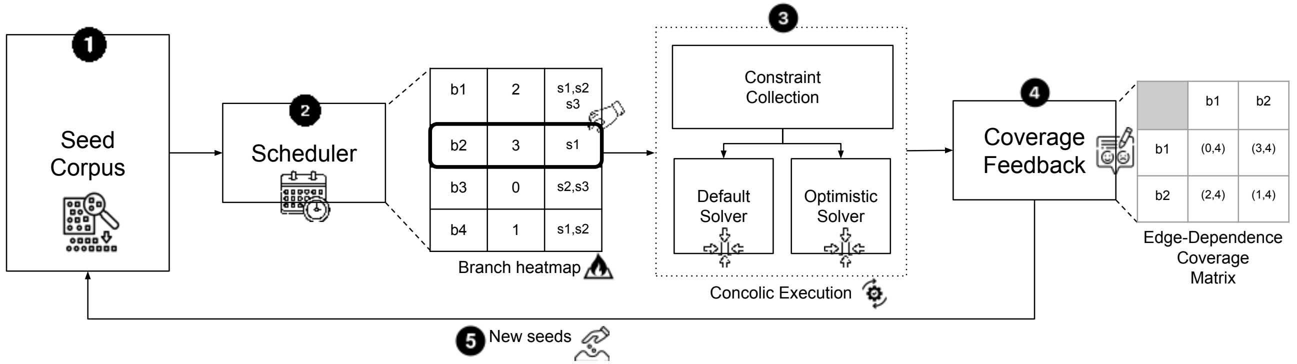


Figure 1: SYMCTS Overview. The system begins with a seed corpus ❶ provided to the scheduler. In ❷, the scheduler selects the most promising seed according to the heatmap-based prioritization strategy. Then, in ❸, the concolic executor runs the chosen seed, collecting path constraints and solving them to generate new inputs. Finally, in ❹, the coverage feedback from these executions is aggregated and stored in a custom coverage matrix. The new seeds ❺ are now part of the corpus.

In this section, we present the design of a novel concolic executor, called SYMCTS, designed as a standalone tool for systematic program exploration. We highlight the key advantages over existing concolic executors used in hybrid fuzzing, as explained in Section 3, and describe how it improves the exploration of our motivating example.

Fig. 1 depicts a schematic overview of our decoupled concolic executor design. To outline this design, we start by explaining how we modify the concolic execution component compared to traditional implementations. We implemented our concolic executor SYMCTS based on LibAFL's [27] concolic execution support, using the instrumentation of SymCC [56].

4.1. Design

In Section 2, we highlighted how naive concolic execution is prone to state explosion and, hence, not efficient for program exploration. On the other hand, we also highlighted that concolic executors for fuzzing use overly coarse coverage metrics, which inhibit exploration progress. Instead, we propose two novel components for concolic execution that strike a unique balance in these considerations: (1) a more fine-grained coverage metric able to preserve significantly more inputs for further mutation and (2) a scheduler that tracks under-explored regions of code to prioritize promising inputs in the resulting larger corpus effectively.

4.1.1. Coverage Metric. Compared to fuzzing, constraint-based input mutation produces significantly fewer, yet higher-quality, inputs. As discussed before, this does not align with the high throughput of the fuzzer's mutation strategy in classic hybrid fuzzing settings. When designing a standalone concolic executor independent of fuzzing constraints, we create room for a more fine-grained coverage metric that allows the concolic executor to leverage its strengths. However, existing metrics (such as full-path coverage) would likely cause the state of a concolic executor to explode, leading to severe performance issues. We therefore propose a new coverage metric, called *edge-dependence coverage*, that aims to track high-level, program-wide dependencies between branches while maintaining a bounded exploration. This was designed to mitigate the problems introduced by concretization and program values that are concrete but are control-dependent on the symbolic input. By allowing the concolic executor to trace and analyze multiple distinct paths for each given program location, the control-dependence relationships between program locations can be explore more broadly. The metric works as follows: Instead of storing information on a per-branch basis, we focus on *pairs* of branches. For each branch pair (b_1, b_2) in a program, we store the minimum and maximum number of times that branch b_2 gets executed in an input that also executes b_1. For this, we use a matrix called the *edge-dependence coverage map*. For example, if we have two inputs, I_1 and I_2, that both execute branch b_1, but input I_1 executes branch b_2 twice (e.g., due to running inside a loop), and I_2 executes branch b_2 four times, we store in our coverage map at location (b_1, b_2) the pair $(2, 4)$. Fig. 2 gives a visual representation of this structure. Note that each row in the matrix represents a branch of our program under test. If a cell in the matrix is updated (either its minimum or its maximum), we consider the triggering input interesting and add it to the corpus of the corresponding branch. We give a concrete example of this process when we revisit the application of SYMCTS to the motivating example in Section 4.2

Moreover, instead of maintaining a global corpus for program exploration, we maintain a corpus of interesting inputs for each branch. This allows us to construct a finer-grained scheduling mechanism, which we explain next.

4.1.2. Scheduler. Program behavior is rarely uniform, meaning that paths leading to some program points can disproportionately outnumber the paths reaching other program points. In SYMCTS, the scheduler is designed and built around the intuition that under-explored regions of code harbor the most significant potential for the discovery of new program behavior and should, therefore, be prioritized for further analysis. Previous work has shown the effectiveness of similar approaches in the context of fuzzing [8], [9].

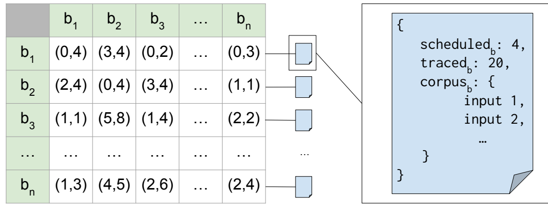


Figure 2: Simplified illustration of SYMCTS's internal data structure for tracking edge-dependent branch coverage. The values in the square matrix represent the min/max counts used to determine novel inputs, while the per-row sampling/tracing counts are used for scheduling.

To implement our scheduler, we augment our edge-dependence coverage matrix with the following fields for each branch: (1) a value representing the number of times an input triggering the given branch was scheduled for symbolic mutation via constraint solving ($scheduled_b$), and (2) a value representing the number of times an input triggering the given branch was encountered during symbolic mutation ($traced_b$). Together with the corpus, we maintain both values for each branch. This is illustrated on the right-hand side of Fig. 2. The scheduler uses these metrics to decide which input to mutate first. In particular, we calculate a score for every branch using the following formula:

$$score_b = traced_b + (100 \cdot scheduled_b)^2$$

The lower $score_b$ is, the higher the priority we give to branch b. Using our priority score, we favor mutating inputs of less-discovered paths by putting a disproportionate weight of 100 on the $scheduled_b$ variable. The weight value was selected to approximate the instrumentation overhead incurred by symbolic tracing. This choice is grounded in prior measurements, which indicate that the overhead generally falls between 60× and 250× [17]. At each iteration, the scheduler selects the branch with the highest priority and schedules one of its inputs for mutation.

4.1.3. Mutation Strategy. The symbolic mutation component in SYMCTS largely follows the mechanism proposed in QSym [63], which was later adopted by SymCC [56] and SymQEMU [57]. However, we make two modifications:

First, QSym implements two solving strategies: *fully constrained solving*, where every constraint is considered, and *optimistic solving* where only the latest constraint is considered. QSym implements optimistic solving as a fallback mechanism in case the fully constrained solving times out. In SYMCTS, we attempt optimistic solving first to filter out cases where the latest constraint alone introduces complexities that cannot be solved, thereby avoiding fully constrained solving in those situations. SYMCTS only attempts fully constrained solving when the optimistic solving succeeded.

Second, we implement *Generational Search* as described by SAGE [31] to avoid re-solving the same constraints. Traditional hybrid fuzzing passes the inputs generated from concolic execution to the fuzzer for evaluation, and new inputs are scheduled for concolic mutation from the fuzzer corpus, losing track of the association between the inputs and the constraints that were solved to create them. As a standalone concolic executor, SYMCTS fully controls each input in its corpus and tracks (and re-uses) the knowledge of the branch whose constraints lead to a given input being produced, therefore avoiding redundant constraint solving.

4.2. Revisiting the Motivating Example

We illustrate the effectiveness of our standalone concolic execution approach on our motivating example in Listing 1. Recall that the bug we want to discover resides in the disassembly logic of a 64-bit ELF architecture. If we want to trigger this bug, we need to reach Line 22 in Listing 1 with a 64-bit ELF input, eventually triggering the bug on Line 29. In traditional hybrid fuzzing, if the fuzzer reaches Line 18 first with a 32-bit architecture, the code (coverage) in `validate_section_headers` will likely be explored almost exclusively with 32-bit ELF files. This means that there is very little chance that a 64-bit ELF would be used as a future input for testing the following code, therefore failing to discover the bug in `print_disasm`. With our standalone concolic execution approach, however, our coverage metric preserves a set of inputs at each program point, allowing both 32-bit *and* 64-bit ELF files to propagate past Line 18. This allows us to explore regions of code *within the context of a specific branch taken at an earlier stage*. Concretely, reaching the branch at Line 22 after executing the branch at Line 7 (with a 32-bit ELF file) is considered different from reaching Line 22 after executing the branch at Line 11 (with a 64-bit ELF file). In our coverage map, this corresponds to updating cell (b_7, b_{22}) after running the 32-bit ELF, and updating cell (b_{11}, b_{22}) after running the 64-bit ELF, showing a clear distinction between the two paths. Our coverage metric thus has a much broader view of the program space and the relationships between branches, allowing it to distinguish when a region of code is explored within a different context (via a different path). Second, if a 32-bit ELF input gets prioritized initially, our *scheduler* will eventually favor less-explored branches and pick a 64-bit ELF for further program exploration. This ensures that Line 22 is reached with a 64-bit ELF input, eventually triggering the bug in Line 29.

5. Evaluation

To examine the rationale and impact of SYMCTS's core design choices, we consider the following research questions:

RQ1 What is the impact of edge-dependence coverage on the concolic executor's ability to discover new code?

RQ2 Does the under-explored branch scheduling accelerate the discovery of new code coverage?

RQ3 How do the proposed scheduling logic and coverage metric complement each other?

RQ4 How does SYMCTS compare against state-of-the-art concolic execution?

RQ5 How does concolic execution using SYMCTS compare against existing grey-box and hybrid fuzzing approaches?

To address research questions **RQ1**, **RQ2**, and **RQ3**, we compare several configurations of SYMCTS against the default setup and against SymCC as the implementation base of our tool. Each configuration disables a specific component, allowing us to isolate and quantify its contribution to the overall performance. To address **RQ4** and **RQ5**, we compare SYMCTS first against a state-of-the-art concolic executor and secondly against widely used grey-box fuzzers (AFL++) and hybrid fuzzers (SymCC+AFL++, SymSan+AFL++). While both SYMCTS and our baseline SymCC implement optimistic solving, we perform one additional minor evaluation to evaluate the impact of optimistic solving on our results as well as the impact of our modification to the ordering. For this purpose, we compare SYMCTS in three settings: SYMCTS's strategy of optimistic solving first, SymCC's strategy of applying full solving second, and finally a version with optimistic solving disabled and using only full solving.

5.1. Experiment Setup

Dataset. To answer **RQ1**, **RQ2**, and **RQ3**, as well as for the optimistic solving evaluation, we leverage the UniBench dataset, a benchmark suite by UNIFUZZ [44] and built around real-world command-line programs spanning image, audio, video, text, binary, and network processing. The targets in UniBench are diverse and have historically been known to contain a wide range of vulnerability classes. In contrast, prior benchmark sets – such as CGC [63], LAVA-M [13], and BugBench [48] – are now considered outdated, synthetic, low in complexity, or fully saturated by the last decade of fuzzing research [19], [37], making them less suitable for a meaningful evaluation. To answer **RQ4** and **RQ5**, we evaluate SYMCTS against a state-of-the-art concolic executor (MARCO [38]) and against grey-box and hybrid fuzzers in the FuzzBench [33] benchmark.

Fuzzing Infrastructure. We ran our UniBench experiments across 9 AWS EC2 c6a.24xlarge instances with 96 vCPUs and 192 GB of RAM, configured with 64 GB of swap space. Our evaluation includes 19 of the 20 targets in UniBench, but excludes gdk-pixbuf-data due to compatibility issues with Ubuntu 20.04. Each experiment is repeated three times, with all trials for the same target running on the same physical machine to ensure consistency. We run each experiment for 24 hours. To ensure a fair comparison, each concolic executor instance is limited to 32 GB of RAM and pinned to one dedicated CPU core.

Corpus Configurations. A key factor when evaluating dynamic program analysis systems is the seed corpus used to bootstrap execution. While some use cases provide a target-specific corpus, others must begin from scratch and rely on the analysis system to build an effective corpus over time. For our evaluations in the UniBench framework, we leverage three different seed corpora: *Empty Corpus*, *Unibench Corpus*, and *Saturated corpus*.

(1) Empty Corpus. First, we use a minimal corpus containing only repetitions of the byte "A" in sizes 4, 8, 16, 32, 64, 128, 256, 512, and 1,024 bytes. This configuration tests the concolic executor's ability to explore programs from scratch without domain-specific guidance.

(2) Unibench Corpus. In this configuration, we use the initial seed corpus provided by the UNIFUZZ framework [44] itself. This configuration represents a realistic starting point for real-world fuzzing campaigns using a seed corpus that a practitioner might manually provide.

(3) Saturated Corpus. Lastly, we use a fully saturated corpus for each corresponding input format to capture our advantage on well-fuzzed targets with near-saturated coverage. To collect this set, we combine seed inputs from two sources: A publicly available saturated corpus from OSS-Fuzz [32] and the UniBench corpus (discussed above). Concretely, if a target is also present in the OSS-Fuzz framework, we download and extract the public corpus from OSS-Fuzz, and augment it with UniBench seeds (prefixed with `unibench_` to avoid filename collisions). For targets not present in the OSS-Fuzz framework, we use a saturated corpus of a similar target that processes the same input file format as the target application. Table 7 in Appendix A shows the exact mapping of UniBench targets to the OSS-Fuzz project and fuzzer used to construct the saturated input corpus. Note that for four targets (flvmeta, infotocap, jq, ffmpeg) we did not identify a directly applicable OSS-Fuzz corpus. The saturated corpus for these targets consists of just the UniBench corpus. Lastly, we filter out seeds larger than 1 MB to accommodate SymCC since it is designed to run alongside AFL, which has this maximum file-size restriction.

SYMCTS Configurations. To assess the individual contributions of SYMCTS's components (**RQ1–RQ3**), we isolate each component using four reduced configurations of SYMCTS: SYMCTS-*L*, SYMCTS-*S*, SYMCTS-*LS*, and the default SYMCTS.

(1) SYMCTS-*L.* To study the effect of edge-dependence coverage tracking, we implement a version of SYMCTS that uses a traditional context-insensitive branch coverage metric with hitcount buckets, e.g., as implemented by AFL++. Concretely, we reduce our edge-dependence matrix to its diagonal entries only. We call this "linearized" coverage. All other coverage tracking and aggregation remain unchanged, allowing us to isolate the effect of edge-dependence.

(2) SYMCTS-*S.* This configuration uses edge-dependence coverage to assess a seed novelty, but adopts a different scheduling strategy, which is loosely inspired by SymCC's internal scheduling logic [56]. This scheduler prioritizes seeds first by whether they discover new coverage, then by whether they originate from the initial corpus, followed by their input length (shorter seeds are favored), and finally by their discovery order, with more recently discovered seeds receiving higher priority. We use this configuration to isolate the contributions of our own scheduling approach.

(3) SYMCTS-*LS.* In this configuration, we combined SYMCTS-L and SYMCTS-S to resemble more closely the design of SymCC, while still preserving the advantages of generational search and the mutation logic of SYMCTS. This allows us to not only capture the effects of both design decisions in isolation, but also of their symbiosis.

(4) Default SYMCTS. Lastly, we use the full implementation of SYMCTS. This configuration deploys edge-dependence coverage alongside the under-explored branch-scheduling algorithm, generational search, and

mutation logic, as described in Section 4. Since SYMCTS is built on top of SymCC, we also include a baseline of concolic execution using SymCC alongside AFL, modified not to perform any mutations, therefore measuring only the exploration ability of the concolic execution itself. This highlights the differences in exploration capabilities afforded by a fully independent concolic executor that does not delegate its responsibilities to a fuzzer.

6. Experiment Results

Tables 1, 2 and 3 summarize our results for each input corpus category. In what follows, we discuss the impact of each of our components when used in these different input corpus scenarios.

6.1. Impact of Edge-Dependence Coverage

Empty Corpus. In total, SYMCTS reaches the highest coverage on 15/19 targets. For the other four targets, SYMCTS reaches the second-highest coverage, trailing only SYMCTS-S, which also uses edge-dependence coverage. Thus, we can conclude that, when given an empty corpus, edge-dependence is the most effective coverage metric in our experiments.

Unibench Corpus. Configurations using edge-dependence coverage reach the highest coverage on 18/19 targets (SYMCTS on 11, SYMCTS-S for the remaining 7) with a median improvement of 1.1% and a mean improvement of 5.2%. In contrast, we see that the non-edge-dependence configurations, SYMCTS-L and SYMCTS-LS, trail SYMCTS by median differences of 22.6% and 20.6%, respectively, across all targets. As such, we see that edge-dependence coverage improves concolic execution also for the UniBench seed corpus.

Saturated Corpus. For the saturated corpus, edge-dependence configurations reach the highest coverage across 14/19 targets. We note that for four of the remaining five targets, SYMCTS-S, which *does* use edge-dependence, reaches only 1% fewer edges than the best performing configuration without edge-dependence.

Summary. As shown in Table 4, across all corpus configurations and corpus settings, our experiments clearly show that instances with edge-dependence coverage significantly outperform the respective configurations without. This answers **RQ1**: Concolic execution significantly benefits from edge-dependence coverage universally across various targets and initial corpus settings.

6.2. Impact of Under-Explored Branch Scheduling

Empty corpus. For the empty corpus, our under-explored branch scheduler provides a clear advantage over using SymCC's scheduler. On 14/19 targets, configurations with our scheduler achieve higher coverage than those without it. An example of the impact that under-explored branch scheduling can make can be seen in the `objdump` target in Fig. 3, where both instances of SYMCTS using our scheduler significantly outperform the instances that do not.

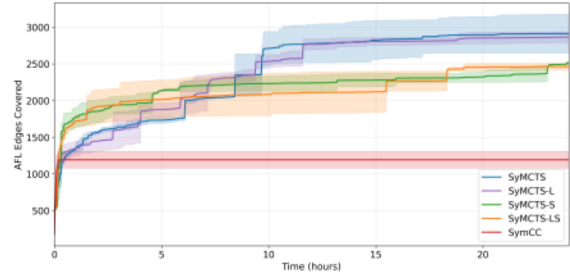


Figure 3: Coverage over time for the `objdump` target using an empty initial corpus.

Removing our edge-dependence coverage diminishes this effect while still achieving better results than SymCC's scheduler. Thus, when given an empty corpus, we conclude that using our under-explored branch scheduler yields greater coverage.

Unibench corpus. For exploration with the initial UniBench corpus, we see neither advantage nor disadvantage of using our under-explored branch scheduler. That is, we observe a median difference of 0.0% and 0.1% in the experiment comparison, as seen in Table 4.

Saturated corpus. For the saturated corpus, our scheduler generally under-performs against SymCC's scheduler in both comparisons.

We hypothesize that this difference stems from SymCC's scheduler prioritizing the exploration of all inputs that originate from the initial seed corpus. When starting with a rich corpus of well-formed inputs that have already undergone extensive fuzzing, most reachable regions of the program are likely already covered by the input corpus. Since our scheduler does not apply special consideration to the original corpus inputs, we believe it focuses its exploration on relatively small areas of the code that are not yet fully covered, rather than fully exploring these high-quality seeds to discover new such regions.

Summary. In our evaluation to answer **RQ2** we find that under-explored branch scheduling effectively improves program exploration in settings with limited input corpora. However, the most effective scheduling strategy appears to vary significantly based on the type of input corpus available for the target under test. We have therefore shown that practitioners should not rely solely on a one-size-fits-all scheduling solution.

6.3. Combined Improvements

Empty corpus. For the empty corpus experiment, we observe that the scheduler accelerates the performance of edge-dependence. For example, in Fig. 4, we depict the median coverage over time for `exiv2`, and see that the median coverage reached by SYMCTS-S after 24 hours is already reached by SYMCTS in merely 4 hours.

Even in cases where, at the end of the experiment, no clear advantage exists for our under-explored branch scheduling, the scheduler is still more effective at selecting mutation targets that maximize the exploration of coverage much earlier in the experiment than other schedulers. For example, on the `infotocap` target, the median coverage reached by SYMCTS and SYMCTS-S after 24 hours is

TABLE 1: Coverage Summary for empty corpus: Median edges covered across all targets. The #Edges column shows the total number of instrumented edges in each target. Percentage values show difference relative to SYMCTS (positive means the respective approach outperformed SYMCTS).

Target	#Edges	SyMCTS	SyMCTS-L	SyMCTS-LS	SyMCTS-S	SymCC
cflow	4952	838	805 (-3.9%)	812 (-3.1%)	**849 (+1.3%)**	627 (-25.2%)
exiv2	50570	**4832**	2807 (-41.9%)	3981 (-17.6%)	4236 (-12.3%)	1004 (-79.2%)
ffmpeg	380488	**2266**	**2266 (+0.0%)**	**2266 (+0.0%)**	**2266 (+0.0%)**	—
flvmeta	4582	**219**	153 (-30.1%)	158 (-27.9%)	179 (-18.3%)	26 (-88.1%)
imginfo	13005	**1193**	1085 (-9.1%)	912 (-23.6%)	832 (-30.3%)	417 (-65.0%)
infotocap	6765	1037	956 (-7.8%)	955 (-7.9%)	**1043 (+0.6%)**	523 (-49.6%)
jhead	1210	**536**	285 (-46.8%)	243 (-54.7%)	534 (-0.4%)	28 (-94.8%)
jq	6067	**1763**	1543 (-12.5%)	1640 (-7.0%)	1713 (-2.8%)	1200 (-31.9%)
lame	11068	**2944**	1779 (-39.6%)	1779 (-39.6%)	2187 (-25.7%)	93 (-96.8%)
mp3gain	2088	**881**	271 (-69.2%)	261 (-70.4%)	735 (-16.6%)	147 (-83.3%)
mp42aac	14102	**1915**	1721 (-10.1%)	1691 (-11.7%)	1824 (-4.8%)	257 (-86.6%)
mujs	14754	3354	2345 (-30.1%)	2478 (-26.1%)	**3419 (+1.9%)**	1655 (-50.7%)
nm	32220	1352	1337 (-1.1%)	1309 (-3.2%)	**2023 (+49.6%)**	578 (-57.2%)
objdump	45432	**2912**	2865 (-1.6%)	2462 (-15.5%)	2511 (-13.8%)	1192 (-59.1%)
pdftotext	29265	**873**	852 (-2.4%)	843 (-3.4%)	838 (-4.0%)	456 (-47.8%)
sqlite3	36007	**606**	602 (-0.7%)	602 (-0.7%)	602 (-0.7%)	602 (-0.7%)
tcpdump	28989	**10877**	10545 (-3.1%)	10388 (-4.5%)	9546 (-12.2%)	37 (-99.7%)
tiffsplit	5890	**87**	**87 (+0.0%)**	86 (-1.1%)	**87 (+0.0%)**	**87 (+0.0%)**
wav2swf	3913	**93**	79 (-15.1%)	79 (-15.1%)	**93 (+0.0%)**	44 (-52.7%)
Mean Diff		0.0%	-17.1%	-17.5%	-4.6%	-59.4%
Median Diff		0.0%	-9.1%	-11.7%	-2.8%	-58.2%

TABLE 2: Coverage Summary for unibench corpus: Median edges covered across all targets. The #Edges column shows the total number of instrumented edges in each target. Percentage values show difference relative to SYMCTS (positive means the respective approach outperformed SYMCTS).

Target	#Edges	SyMCTS	SyMCTS-L	SyMCTS-LS	SyMCTS-S	SymCC
cflow	4952	**1064**	1063 (-0.1%)	1061 (-0.3%)	1063 (-0.1%)	1060 (-0.4%)
exiv2	50570	**5167**	2901 (-43.9%)	4092 (-20.8%)	4336 (-16.1%)	1929 (-62.7%)
ffmpeg	380488	**9689**	9688 (-0.0%)	**9689 (+0.0%)**	**9689 (+0.0%)**	—
flvmeta	4582	217	205 (-5.5%)	204 (-6.0%)	**218 (+0.5%)**	182 (-16.1%)
imginfo	13005	1469	1590 (+8.2%)	1590 (+8.2%)	**1622 (+10.4%)**	1290 (-12.2%)
infotocap	6765	**1119**	1019 (-8.9%)	1032 (-7.8%)	1073 (-4.1%)	590 (-47.3%)
jhead	1210	519	429 (-17.3%)	430 (-17.1%)	**531 (+2.3%)**	203 (-60.9%)
jq	6067	**1847**	1765 (-4.4%)	1828 (-1.0%)	1835 (-0.6%)	1710 (-7.4%)
lame	11068	**2964**	2931 (-1.1%)	2935 (-1.0%)	2938 (-0.9%)	2928 (-1.2%)
mp3gain	2088	893	868 (-2.8%)	877 (-1.8%)	**895 (+0.2%)**	798 (-10.6%)
mp42aac	14102	2557	**2649 (+3.6%)**	2531 (-1.0%)	2562 (+0.2%)	1301 (-49.1%)
mujs	14754	3735	3042 (-18.6%)	3136 (-16.0%)	**3892 (+4.2%)**	2736 (-26.7%)
nm	32220	2669	2646 (-0.9%)	2910 (+9.0%)	**2962 (+11.0%)**	2629 (-1.5%)
objdump	45432	3339	3162 (-5.3%)	4125 (+23.5%)	**4221 (+26.4%)**	3905 (+17.0%)
pdftotext	29265	**5816**	5804 (-0.2%)	4531 (-22.1%)	4388 (-24.6%)	4765 (-18.1%)
sqlite3	36007	**606**	**606 (+0.0%)**	**606 (+0.0%)**	**606 (+0.0%)**	**606 (+0.0%)**
tcpdump	28989	**11586**	10245 (-11.6%)	10321 (-10.9%)	10030 (-13.4%)	7908 (-31.7%)
tiffsplit	5890	**435**	434 (-0.2%)	434 (-0.2%)	**435 (+0.0%)**	432 (-0.7%)
wav2swf	3913	**97**	84 (-13.4%)	84 (-13.4%)	93 (-4.1%)	81 (-16.5%)
Mean Diff		0.0%	-6.4%	-4.1%	-0.5%	-19.2%
Median Diff		0.0%	-2.8%	-1.0%	+0.0%	-14.2%

similar, yet SYMCTS reached this stage 12 hours before SYMCTS-S.

We conclude that, for practitioners in need of a rapid exploration of a program without an available input corpus, the combination of under-explored branch scheduling together with edge-dependence coverage can provide a significant speedup compared to other configurations.

Unibench corpus. We see the trend continue for these three targets when the manually created initial Unibench corpus is used. For `exiv2`, again, SYMCTS discovers the final coverage reached by SYMCTS-S in around 2-3 hours. For `infotocap`, SYMCTS discovers the coverage reached by SYMCTS-S in 24 hours in only 10 hours. We have elided the full plots for brevity, but they can be found in Appendix B.

Saturated corpus. For the saturated corpus, we no longer see our under-explored branch scheduler having a positive effect on coverage reached in the target, with SYMCTS-S now reaching the highest level of coverage on 10/19 targets and achieving the second-highest coverage on eight of the remaining nine targets. On the last remaining target (`mp3gain`), however, all SYMCTS variants are within 0.2% of the highest coverage.

We believe this is mainly due to the agnosticism of our under-explored branch scheduler regarding the treatment of the initial corpus and the resulting differences in the underlying exploration distribution, as described previously.

Summary. While the under-explored branch scheduler demonstrates limited improvements on its own, it synergizes well with our edge-dependence coverage to accelerate and stabilize the discovery of new coverage within a

TABLE 3: Coverage Summary for saturated corpus: Median edges covered across all targets. The #Edges column shows the total number of instrumented edges in each target. Percentage values show difference relative to SYMCTS (positive means the respective approach outperformed SYMCTS).

Target	#Edges	SyMCTS	SyMCTS-L	SyMCTS-LS	SyMCTS-S	SymCC
cflow	4952	**1082**	1076 (-0.6%)	1079 (-0.3%)	1080 (-0.2%)	949 (-12.3%)
exiv2	50570	8562	8453 (-1.3%)	8469 (-1.1%)	**8658 (+1.1%)**	7353 (-14.1%)
ffmpeg	380488	**9693**	**9693 (+0.0%)**	**9693 (+0.0%)**	9690 (-0.0%)	—
flvmeta	4582	217	205 (-5.5%)	206 (-5.1%)	**218 (+0.5%)**	177 (-18.4%)
imginfo	13005	1463	1442 (-1.4%)	1606 (+9.8%)	**1627 (+11.2%)**	1176 (-19.6%)
infotocap	6765	**1097**	1055 (-3.8%)	1050 (-4.3%)	1060 (-3.4%)	578 (-47.3%)
jhead	1210	530	501 (-5.5%)	483 (-8.9%)	**534 (+0.8%)**	431 (-18.7%)
jq	6067	1825	1741 (-4.6%)	1789 (-2.0%)	**1841 (+0.9%)**	1703 (-6.7%)
lame	11068	**2959**	2939 (-0.7%)	2930 (-1.0%)	2956 (-0.1%)	2899 (-2.0%)
mp3gain	2088	882	893 (+1.2%)	**894 (+1.4%)**	892 (+1.1%)	813 (-7.8%)
mp42aac	14102	2517	**2543 (+1.0%)**	2505 (-0.5%)	2524 (+0.3%)	1608 (-36.1%)
mujs	14754	4807	4454 (-7.3%)	4682 (-2.6%)	**5271 (+9.7%)**	4780 (-0.6%)
nm	32220	3879	3875 (-0.1%)	3882 (+0.1%)	**3914 (+0.9%)**	3164 (-18.4%)
objdump	45432	5478	5459 (-0.3%)	5520 (+0.8%)	5535 (+1.0%)	**5558 (+1.5%)**
pdftotext	29265	11723	12013 (+2.5%)	7954 (-32.2%)	**12127 (+3.4%)**	11308 (-3.5%)
sqlite3	36007	606	603 (-0.5%)	603 (-0.5%)	606 (+0.0%)	**607 (+0.2%)**
tcpdump	28989	9331	9853 (+5.6%)	**11000 (+17.9%)**	10292 (+10.3%)	6832 (-26.8%)
tiffsplit	5890	**1317**	1313 (-0.3%)	1313 (-0.3%)	**1317 (+0.0%)**	483 (-63.3%)
wav2swf	3913	**101**	97 (-4.0%)	97 (-4.0%)	**101 (+0.0%)**	94 (-6.9%)
Mean Diff		0.0%	-1.3%	-1.7%	+2.0%	-16.7%
Median Diff		0.0%	-0.6%	-0.5%	+0.8%	-13.2%

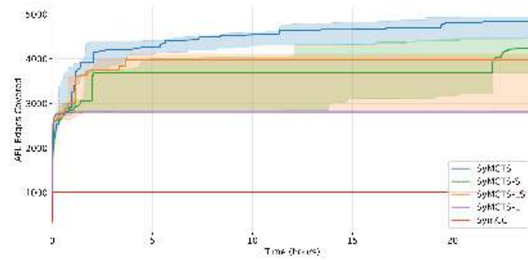
TABLE 4: Summary of performance benefit for Edge-Dependence coverage and our Scheduler. For each input corpus, we show the mean/median percentage of improvement achieved by each design decision across all targets.

	Empty	Unibench	Saturated
Edge-Dependence			
SyMCTS vs -L	+32.2/+10.0%	+9.0/+2.9%	+1.5/+0.6%
SyMCTS-S vs -LS	+24.5/+6.4%	+4.2/+1.8%	+4.7/+0.9%
Scheduler			
SyMCTS vs -S	+7.4/+2.9%	+1.6/+0.0%	-1.8/-0.7%
SyMCTS-L vs -LS	+1.0/+0.1%	-2.0/-0.1%	+1.4/-0.1%

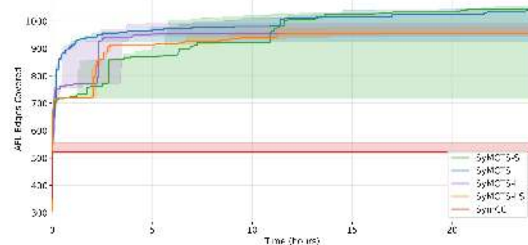
given time frame, especially when using sparser corpora.

6.4. Comparison against State-of-the-Art Concolic Execution

In this section, we evaluate against Marco, a state-of-the-art concolic executor developed by Hu et al. [38]. Marco is a concolic executor, based on SymSan, which implements a novel branch-scheduling strategy (referred to as a branch-flipping policy in their paper) based on Thompson Sampling [16]. "Branch-scheduling" in this case refers to an algorithm that selects which constraints to solve and produce new derivative inputs for. The authors recognize that solving for previously-covered branches can be beneficial due to the problem of "divergences" in concolic execution (when the input derived from solving a given symbolic constraint does not actually lead to the target branch being taken). Their approach models the branch-scheduling problem as a Markov process and proposes a branch-scheduler to optimize the expected "reward" of the coverage reached via Thompson Sampling [16]. This provides a model for the probability of discovering new coverage when solving the constraints of a given branch, prioritizing constraint-solving for the most promising candidates.



(a) `exiv2` (empty corpus)



(b) `infotocap` (empty corpus)

Figure 4: Coverage over time for the exiv2 and infotocap experiments using an empty initial corpus.

Unfortunately, Marco was not directly compatible with our main evaluation due to its dependency on an outdated *clang* version (version 6.0) in the end-of-life Ubuntu 18.04 distribution. Instead, we integrated SYMCTS into the Docker setup of the publicly released artifact from the Marco paper. We use targets from the FuzzBench framework adapted to fit the versioning of Marco for this evaluation. After corresponding with the authors of Marco, we successfully compiled and evaluated 12/22 of the FuzzBench targets. The other 10 targets exhibited issues during compilation, caused either by an out-of-date distribution or by missing Data-flow Sanitizer (DFSAN) instrumentation lists, as required by Marco. Note that SYMCTS does not require all external functions in the

target program to be annotated and therefore does not have the same issue.

We evaluate all targets on two physical machines, each equipped with 48 physical cores, 96 virtual cores, Intel Xeon Gold CPUs at 2.10GHz, and 378GB of RAM. The experiments were run for 4 days (96 hours), and we evaluated the edge (branch) coverage reached by both approaches. Similarly to the empty corpus setting of our evaluation in Section 5, we evaluate the tools without using project-specific seed corpora to assess their ability to discover program behaviors on their own. Finally, we retrace all discovered inputs to evaluate the coverage reached by each approach.

Table 5 shows the median coverage for all evaluated FuzzBench targets. Except for `libxml2`, SYMCTS outperforms Marco on all of the targets. On 66% (8) of the targets, SYMCTS reaches a higher coverage after 12 hours than Marco reached after 96 hours. Moreover, on 50% (6) of the targets, the worst-performing run of SYMCTS still achieves a higher coverage than the best-performing run of Marco (see Table 5). Overall, SYMCTS outperforms Marco with respect to minimum/maximum/median/mean coverage on 10/12 targets and ties for another (*jsoncpp*).

TABLE 5: Comparison of Marco and SymCTS results across different benchmarks

	Marco		SymCTS	
	min.	max.	min.	max.
`freetype2`	1,692	2,963	**2,385**	**4,650**
`jsoncpp`	664	666	664	666
`lcms`	260	453	**1,326**	**1,385**
`libpng`	867	910	**885**	**913**
`libxml2`	**5,218**	**5,572**	4,581	5,422
`openh264`	375	1,479	**1,377**	**6,174**
`openthread`	1,757	1,757	**2,332**	**2,349**
`re2`	2,253	2,591	**2,704**	**2,721**
`sqlite3`	4,845	5,682	**6,604**	**7,802**
`vorbis`	64	164	**253**	**261**
`woff`	808	1,033	**1,052**	**1,077**
`zlib`	325	328	**327**	**332**

6.5. Comparison against Grey-box and Hybrid Fuzzers

To evaluate how standalone concolic execution compares to state-of-the-art fuzzing techniques, we ran SYMCTS against pure grey-box fuzzers (AFL++) and hybrid fuzzers (SymCC alongside AFL++, SymSan alongside AFL++) on the FuzzBench benchmark suite. These experiments ran for 96 hours to account for concolic execution's higher overhead compared to traditional fuzzing. The full results are presented in Appendix D.

While the comparison makes clear that standalone concolic execution still faces substantial limitations on many real-world programs, with grey-box and hybrid fuzzers achieving higher coverage on the majority of FuzzBench targets, some targets exhibit surprising behavior that invites further research (Fig. 5). On `lcms_cms_transform_fuzzer`, SYMCTS achieves higher coverage than AFL++ throughout the experiment. On `openthread_ot-ip6-send-fuzzer`, SYMCTS surpasses all evaluated approaches after approximately 80 hours. On `stb_stbi_read_fuzzer`, SYMCTS temporarily leads both SymSan+AFL++ and AFL++ around the 30-hour mark before the fuzzers catch up (Fig. 10r). While these remain exceptions rather than the rule, they suggest that specific program characteristics may be more amenable to standalone concolic execution than current hybrid approaches. We believe that the limitations encountered by concolic execution most likely stem from research challenges rather than scalability or performance issues: indirect control flow complicating taint tracking, operations requiring concretization that lose precision, and complex constraints.

6.6. Impact of optimistic solving

We evaluated the optimistic solving strategies across all UniBench with an empty input corpus. The median result of three trials after 12 hours for each target can be seen in Table 6. We see that SYMCTS-full-first (SymCC-style) discovers a median coverage improvement of 0.53% fewer edges across all targets, full-solve-only discovers a median of around 19% fewer edges than both across all targets.

TABLE 6: Comparison of median edges reached of SYMCTS, SYMCTS-full-first (-ff) and SYMCTS-full-only (-fo) across three trials in 12 hour evaluation on all UniBench targets with an empty input corpus.

	SYMCTS	SYMCTS-ff	SYMCTS-fo
`cflow`	849	819 (-3.53%)	691 (-18.61%)
`exiv2`	4800	4701 (-2.06%)	2887 (-39.85%)
`ffmpeg`	2266	2266 (0.00%)	2266 (0.00%)
`flvmeta`	219	180 (-17.81%)	169 (-22.83%)
`imginfo`	881	723 (-17.93%)	827 (-6.13%)
`infotocap`	1038	1011 (-2.60%)	709 (-31.70%)
`jhead`	532	488 (-8.27%)	295 (-44.55%)
`jq`	1712	1703 (-0.53%)	1672 (-2.34%)
`lame`	2937	2935 (-0.07%)	1648 (-43.89%)
`mp3gain`	830	869 (+4.70%)	308 (-62.89%)
`mp42aac`	1878	1875 (-0.16%)	1630 (-13.21%)
`mujs`	3268	3343 (+2.29%)	2640 (-19.22%)
`nm`	1354	1348 (-0.44%)	1241 (-8.35%)
`objdump`	2518	2285 (-9.25%)	1798 (-28.59%)
`pdftotext`	866	868 (+0.23%)	508 (-41.34%)
`sqlite3`	602	606 (+0.66%)	602 (0.00%)
`tcpdump`	9160	8444 (-7.82%)	5819 (-36.47%)
`tiffsplit`	87	87 (0.00%)	87 (0.00%)
`wav2swf`	93	91 (-2.15%)	93 (0.00%)
Median %diff	–	-0.53%	-19.22%

7. Discussion

In the following paragraphs, we discuss the further-reaching consequences of our approach on the design and implementation of concolic executors. We highlight the advantages of building a self-contained concolic executor with regards to resource constraints and its theoretical exploration capabilities, and argue that concolic execution should receive a broader research focus than it currently does in the context of hybrid fuzzing.

7.1. Concolic Tracing vs. Constraint Solving

Recent work focused on advancing the *performance* of concolic tracing components in hybrid fuzzers [17], [56],

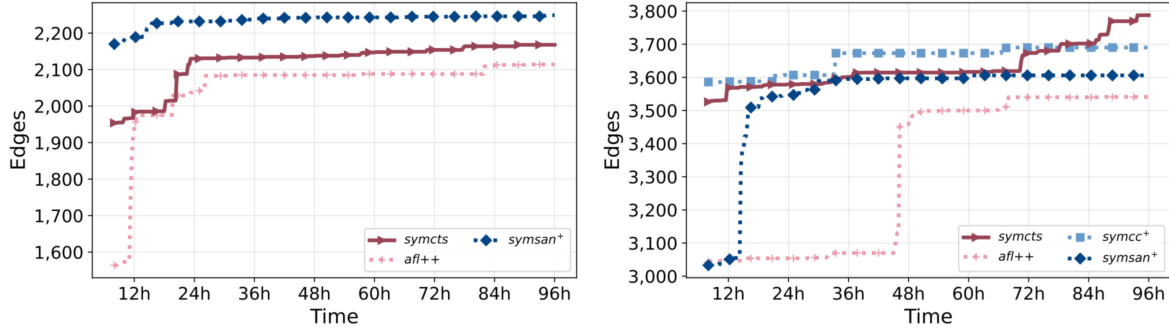


Figure 5: Coverage over time on FuzzBench targets where SYMCTS demonstrates competitive performance. Left: `lcms_cms_transform_fuzzer` showing SYMCTS outperforming AFL++. Right: `openthread_ot-ip6-send-fuzzer` showing SYMCTS surpassing all approaches after 80 hours.

[57], [63]. For example, SymSan and SymCC made significant improvements to the performance of the constraint collection mechanism of their hybrid fuzzing solutions [17], [56]. This focus is reasonable in the current coupled hybrid fuzzing paradigm, where the concolic executor is designed to solve only a small set of constraints to improve the *fuzzer's* coverage metric. As a result, the performance of the concolic tracing component needs to "keep up" with the fuzzer's high throughput, making it a bottleneck for current hybrid fuzzing approaches.

SYMCTS, however, makes liberal use of the constraint solver as its main mutation component, solving for all divergent branches from a given input. While this might suggest a slower performance of SYMCTS compared to current concolic executors in hybrid fuzzing, our experiments show relatively little evidence of this, as illustrated by our comparison against SymCC. In fact, our concolic executor consistently achieves greater coverage faster than a more "restrained" concolic executor like SymCC, which restricts constraint solving to cases guaranteed to surface more coverage, showcasing the overlooked potential of concolic executors as an independent driver of program exploration.

Hence, we argue that the current research trend of focusing mainly on improving the performance of concolic tracing to collect constraints for concolic executors used in hybrid solutions should be reconsidered. Instead, further research into optimizing constraint-solving approaches and addressing additional constraint-solving opportunities, such as indirect taint, could yield greater benefits.

7.2. Coverage Metrics

In SYMCTS, we use edge-dependence coverage, which, unlike traditional coverage metrics, scales quadratically with respect to the size of the target program. While this provides a clearer picture of the relationship between branches during program execution, it also suggests potential scalability issues that could degrade the tool's performance. Contrary to current dogma, our results show that this compromise in performance does not hinder the concolic executor in effectively exploring the program space, but in fact leads to more effective program exploration.

7.3. Limitations

As a concolic executor, the limitations of SYMCTS generally fall into two categories: limitations inherited from the concolic execution paradigm and limitations introduced by the design choices of our edge-dependence coverage and under-explored branch sampling. In the first category, the main limitations of SYMCTS are inherited from the SymCC concolic instrumentation framework it relies on: It does not support mutations that change the length of the symbolic input, it does not track control-flow dependencies and, as such, can lose relationships with control-dependent variables, the symbolic memory model concretizes addresses on first access, and, finally, the performance and capabilities of the underlying constraint solving engines. While the tracking of multiple paths by our edge-dependence coverage is able to mitigate some common cases of these issues by exploring the control-dependent as well as memory access locations in various contexts and paths, it does not provide a general solution to these problems, and further research in this field is required.

The second category contains limitations that are directly linked to our design choices. First, our evaluation reveals that the under-explored branch scheduler, while effective for empty corpus exploration, degrades performance on saturated corpora where SymCC's default scheduling outperforms it. We attribute this to SymCC's higher weighting of starting-corpus inputs. A second limitation lies in the increased overhead incurred by the NxN edge-dependence matrix that needs to be stored, updated and evaluated for each produced input. While this trade-off is effective for concolic execution due to the high-quality input generation and slower throughput requirements of concolic executors, it likely remains infeasible for higher-throughput dynamic analysis approaches such as fuzzing or for extremely large pieces of software like the Linux kernel or the Chrome browser due to quadratic scaling with respect to the number of branches in the target.

7.4. Future Work

Potential for Hybrid Integration. While SYMCTS is designed as a standalone concolic executor, future work could explore hybrid designs that combine SYMCTS with fuzzers in a complementary fashion. In particular,

SYMCTS could support more informed and defensive synchronization between the two systems: the fuzzer's broad input exploration can feed into SYMCTS's structured symbolic reasoning, while SYMCTS can refine, prioritize, or recover hard-to-reach paths for the fuzzer. Such coordinated synchronization could lead to a symbiotic, SYMCTS-driven hybrid approach that may further improve both coverage and efficiency.

Potential Synchronization Strategy. In hybrid fuzzing, input synchronization is considered a beneficial strategy. It allows constituent tools to analyze their own inputs and those generated by other techniques, thereby expanding the overall reach of the combined analysis. However, current solutions disproportionally favor synchronizing input from the fuzzer's side, which we believe can actually degrade the performance of a concolic executor like SYMCTS. Concretely, if the fuzzer finds an input for a branch pair, that pair will be considered explored by our coverage metric. If, however, this input is difficult to mutate, we lose the opportunity to find better inputs stemming from this branch-pair coverage. Ideally, we want the concolic executor to generate an input for each branch pair, as this increases the likelihood that those inputs will later be mutated. Nonetheless, if our concolic executor "gets stuck," we do want to synchronize inputs with the fuzzer to continue program exploration.

We believe that a future hybrid fuzzing solution using SYMCTS would likely benefit from a more "defensive" synchronization strategy that leaves more room for the concolic executor to independently find inputs.

Indirect Taint. One roadblock we identified during the development of SYMCTS stems from concolic execution's usage of dynamic taint tracking for constraint collection to precisely track data-flow dependencies of symbolic input values. Crucially, this does not account for the control dependencies of these values. The concolic executor then treats variables that are indirectly controlled by the input as concrete, and loses the ability to derive new inputs from them. Although our edge-dependence coverage provides an initial step in mitigating this issue by tracking a diverse corpus of inputs per branch-pair, this methodology offers no guarantee in programs with more complex control dependencies. Future work on dynamic taint tracking can address this limitation, helping the concolic executor generate test cases that can handle deeper levels of control dependencies.

Saturated Corpus Scheduling. As discussed in our evaluation, our scheduling generally improves exploration of an empty corpus (from scratch) but does not perform as well for concolic exploration of saturated fuzzing corpora. In theory, a concolic executor could explore every possible state discovered by its coverage metric until no new states are found. However, in practice, this quickly becomes intractable due to the large number of paths and the computational overhead of constraint solving. As such, concolic executors have to make trade-offs about which paths to explore first within a given time period to identify the most interesting behaviors.

Based on our evaluation results, it is apparent that scheduling promising inputs for further mutation clearly requires different solutions depending on the initial corpus setting. As fuzzing and concolic execution continue to improve, we believe that the scheduling problem in the case of a saturated corpus presents an increasingly relevant avenue for future research. This is especially relevant in the context of the increasing adoption of continuous fuzzing solutions like OSS-Fuzz, OneFuzz [53], or CI-Fuzz, in which a given application is fuzzed repeatedly over months as its codebase evolves. In such a scenario, building analyses that operate successfully when considering near-saturated corpora is imperative to maximally utilize the available information over time.

8. Related Work

8.1. Concolic Execution

The initial ideas underpinning concolic execution were first described in 2005 by Godefroid et al. in DART [30]. Sen et al. proposed an approach to apply similar concepts in a unit-testing setting, with support for modeling complex data structures in CUTE [59], and first coined the term "concolic testing." Since then, concolic execution has been used in various forms in tools like SAGE [31], KLEE [14], S^2E [21], Triton [58], and angr [60]. Notably, SAGE in 2008 represents the to-date most successful deployment of pure concolic execution for vulnerability detection in the wild.

SymFusion [23] proposed a combined tracing mechanism that uses source-code instrumentation when available, but falls back to binary instrumentation for libraries and binary-only code. SymSan [17] optimized the constraint-tracking and creation mechanisms for concolic execution by modeling them as dataflow analyses and implementing them atop Clang's data-flow sanitizers.

JIGSAW [18] proposes a faster method of constraint solving based on the gradient descent algorithm of Angora [19] and just-in-time compilation of constraints. Similarly, Borzacchiello et al. proposed Fuzzy-SAT [10], a fast approximate solver that uses fuzzing-like techniques to solve the constraints generated by concolic engines.

Mi et al. identify the input bytes that may influence any target branches and symbolize only those bytes, thus reducing the constraints collected during concolic execution [51]. "Branch-scheduling" was proposed by Hu et al. and restricts the branches for which divergent inputs are computed to the ones that will further increase fuzzer coverage, as well as selecting mutation for inputs that contain more such branches [39]. Geretto et al. propose the use of a prefix-trie structure to optimize constraint solving via improved model-caching and constraint simplification [28].

While recent improvements to the performance of constraint solving are incredibly promising and applicable to SYMCTS, approaches that reduce or minimize the usage of the constraint solver are incompatible with SYMCTS's design as a fully independent concolic executor. The proposed edge-dependency coverage and scheduling algorithm for SYMCTS offers a novel trade-off between full-path exploration (as implemented by SAGE [31]) and coarse-grained coverage metrics of modern hybrid fuzzers.

8.2. Grey-box Fuzzing

Since the release of AFL [64], the landscape of fuzzing research has flourished with a multitude of innovative systems designed to amplify execution speed, increase target exploration, and enhance vulnerability discovery [46], [54], [62], [64]. In recent years, Fioraldi et al. proposed AFLPlusPlus [26], a re-vamped version of AFL incorporating multiple critical improvements suggested by the security community, and LibAFL [27], a framework to build modular and reusable fuzzers. One notable area of enhancement is *stateful fuzzing*, which focuses on maintaining and leveraging the internal state of the target application during fuzz testing [2]–[4]. Another recent advancement is *invariant-driven fuzzing*, which utilizes program invariants to guide the fuzzing process to target specific locations of a program [25], [41]. Other popular techniques include *Directed fuzzing*: a fuzzing strategy to efficiently reach specific parts of the code [40], [43], [47], and, more recently, *LLM-aided fuzzing*, which leverages Large Language Models to guide the generation of input data, enhancing the effectiveness of fuzzing campaigns [50]. While grey-box fuzzing has made great strides in its independent program exploration capabilities, we believe that a concolic executor like SYMCTS offers unique benefits for systematically exploring complex paths and constraint dependencies. While combining concolic executors with grey-box fuzzers will likely remain beneficial, concolic execution as a separate research field can highlight differences in approaches and how they can be improved.

8.3. Hybrid Fuzzing

Majumdar and Sen introduced the concept of hybrid fuzzing in 2007 [49], but the technique was popularized by Driller [61] in 2016, which showcased the potential of hybrid fuzzing during the DARPA Cyber Grand Challenge. Since then, many other solutions followed. One such solution is DigFuzz [65], which uses a Monte Carlo execution tree to quantify the cost of exploring a given path and decide whether to cover it with fuzzing or concolic execution. QSYM [63] is a concolic execution engine optimized for hybrid fuzzing. It detects and prunes repetitive basic blocks to reduce the pressure on the constraint solver. It significantly improved the constraint tracing speed of concolic execution and introduced improvements to constraint-solving approaches (e.g., optimistic solving), which we also use in SYMCTS. Poeplau and Francillon introduced SymCC [56], which embedded symbolic execution capabilities directly into the binary at compile time, effectively avoiding the overhead of IR-based symbolic execution systems and the implementation complexity of IR-less systems [58], [63], while still achieving better performance. As a follow-up, the authors showed how a similar technique can be applied to closed-source binaries by instrumenting QEMU's intermediate representation [57]. Mi et al. designed a per-branch hit-count collection and use it to prioritize branches which have the "most complex branches to be solved" via dynamic taint analysis [52]. Fuzzolic [11] is a SymQEMU-like tracer for binary targets that uses fuzzing for symbolic solving. Li and Zhang developed SILK [42], a constraint-guided hybrid fuzzer that assesses the difficulty of path exploration in code and uses this assessment to guide the fuzzer during program testing. Recently, Hu et al. proposed Marco [38], a concolic executor that improves on the branch-scheduling policy of SymSan by tracking mutation results globally across all executions, and predicting for each branch the "path divergence probability" to gauge its potential for discovering new code.

Hybrid fuzzing re-popularized the idea of concolic execution and, in the process, improved the performance of the key components needed to handle programs at the complexity of real-world software. SYMCTS showcases the potential benefits of standalone concolic execution by implementing fully independent concolic execution with its own fine-grained coverage metrics and scheduling, demonstrating that this approach can lead to significantly more effective program exploration.

9. Conclusion

Combining the fast throughput of a fuzzer with the precise input generation of a concolic executor has lead to the successful adoption of hybrid fuzzing solutions. A fundamental shortcoming of this approach, however, is that its design often suffers from a bias towards the fuzzer's coverage metric, leaving the concolic executor severely underused. In this paper, we proposed decoupling the design of the concolic executor from the fuzzer, which allows us to leverage custom coverage metrics and scheduling techniques tailored to concolic execution. To illustrate this point, we implemented an independent concolic executor, called SYMCTS, together with a novel coverage metric and scheduling algorithm that better leverages the strengths of concolic execution.

From our evaluation, we found that our newly designed coverage metric improves program exploration regardless of the available input corpus, and that our new scheduling algorithm can amplify this effect in specific scenarios. Moreover, we have shown that scheduling algorithms are not a one-size-fits-all solution, and that practitioners require different scheduling strategies depending on the available input corpus.

More importantly, our results show that the strengths of concolic execution are indeed underexplored and, therefore, motivate further research into concolic execution as a viable alternative to grey-box fuzzing. The source code for SYMCTS and our experiments is available under the MIT License at https://github.com/ucsb-seclab/symcts. We additionally provide the Docker containers of our experiments at https://hub.docker.com/symcts.

Acknowledgment

This material is based on research sponsored by DARPA under agreement number N66001-22-2-4037 and HR0011259E022 (with Kudu Dynamics). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or

endorsements, either expressed or implied, of DARPA, the U.S. Government, or Kudu Dynamics, Inc.

References

[1] LibFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2024. Accessed: 2024-02-18.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[4] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[5] 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.

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[7] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Handbook of satisfiability*, 185(99):457–481, 2009.

[8] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. *Commun. ACM*, 66(11):89–97, oct 2023.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, 2021.

[11] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. FUZZOLIC: mixing fuzzing and concolic execution. *Computers & Security*, 2021.

[12] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 122–131. IEEE, 2013.

[13] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. Evaluating synthetic bugs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 716–730, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

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

[16] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, page 2249–2257, Red Hook, NY, USA, 2011. Curran Associates Inc.

[17] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. {SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2531–2548, 2022.

[18] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1531–1531. IEEE Computer Society, 2022.

[19] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[20] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, mar 2011.

[22] TIS Committee. Tool interface standard (tis) executable and linking format (elf) specification. https://refspecs.linuxfoundation.org/elf/elf.pdf, 1995. Accessed: 2024-04-22.

[23] Emilio Coppa, Heng Yin, and Camil Demetrescu. SymFusion: Hybrid Instrumentation for Concolic Execution. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, 2022.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[25] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846, 2021.

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, pages 10–10, 2020.

[27] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1051–1065, 2022.

[28] Elia Geretto, Julius Hohnerlein, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus v. Gleissenthall. Triereme: Speeding up hybrid fuzzing through efficient query scheduling. In *Proceedings of the 39th Annual Computer Security Applications Conference*, ACSAC '23, page 56–70, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 206–215, New York, NY, USA, 2008. Association for Computing Machinery.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[31] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[32] Google. Oss-fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz, 2016. Accessed: 2025-11-20.

[33] Google. Fuzzbench integration: Symcc, test-case minimization. https://github.com/google/fuzzbench/blob/ae21be9dff84936e64b2525ab4ea1411a0b81529/fuzzers/symcc_aflplusplus/fuzzer.py#L111, 2022.

[34] Google. Honggfuzz - security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz, 2024. Accessed: 2024-02-18.

[35] Google. Oss-fuzz - continuous fuzzing for open source software. https://github.com/google/oss-fuzz, 2024. Accessed: 2024-04-22.

[36] Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI Global, 2015.

[37] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[38] Jie Hu, Yue Duan, and Heng Yin. Marco: A stochastic asynchronous concolic explorer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[39] Qi Hu, Weijia Chen, Zhi Wang, Shuaibing Lu, Yuanping Nie, Xiang Li, and Xiaohui Kuang. Bsfuzz: Branch-state guided hybrid fuzzing. *Electronics*, 12(19), 2023.

[40] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.

[41] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 142–142. IEEE Computer Society, 2024.

[42] Junhao Li and Yujian Zhang. Silk: Constraint-guided hybrid fuzzing. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 607–616, 2023.

[43] Penghui Li, Wei Meng, and Chao Zhang. Sdfuzz: Target states driven directed fuzzing. In *Proceedings of the 33rd USENIX Security Symposium (Security). Philadelphia, PA, USA*, 2024.

[44] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[45] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[46] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 18(6):2675–2688, 2019.

[47] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, and Kai Lu. Deepgo: Predictive directed greybox fuzzing.

[48] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5. Chicago, Illinois, 2005.

[49] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[50] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[51] Xianya Mi, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. Leansym: Efficient hybrid fuzzing through conservative constraint debloating. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 62–77, 2021.

[52] Xianya Mi, Baosheng Wang, Yong Tang, Pengfei Wang, and Bo Yu. Shfuzz: Selective hybrid fuzzing with branch scheduling based on binary instrumentation. *Applied Sciences*, 10(16), 2020.

[53] Microsoft. Onefuzz - a self-hosted fuzzing-as-a-service platform. https://github.com/microsoft/onefuzz, 2020. Accessed: 2025-09-19.

[54] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[55] NIST. Software errors cost u.s. economy 59.5 billion usd annually nist assesses technical needs of industry to improve software-testing. https://www.abeacha.com/NIST_press_release_bugs_cost.html, 2017.

[56] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[57] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *NDSS*, 2021.

[58] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*, pages 31–54, 2015.

[59] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[60] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

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

[62] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *2023 IEEE symposium on security and privacy (SP)*, pages 2658–2675. IEEE, 2023.

[63] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[64] Michał Zalewski. American fuzzy lop -whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016.

[65] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

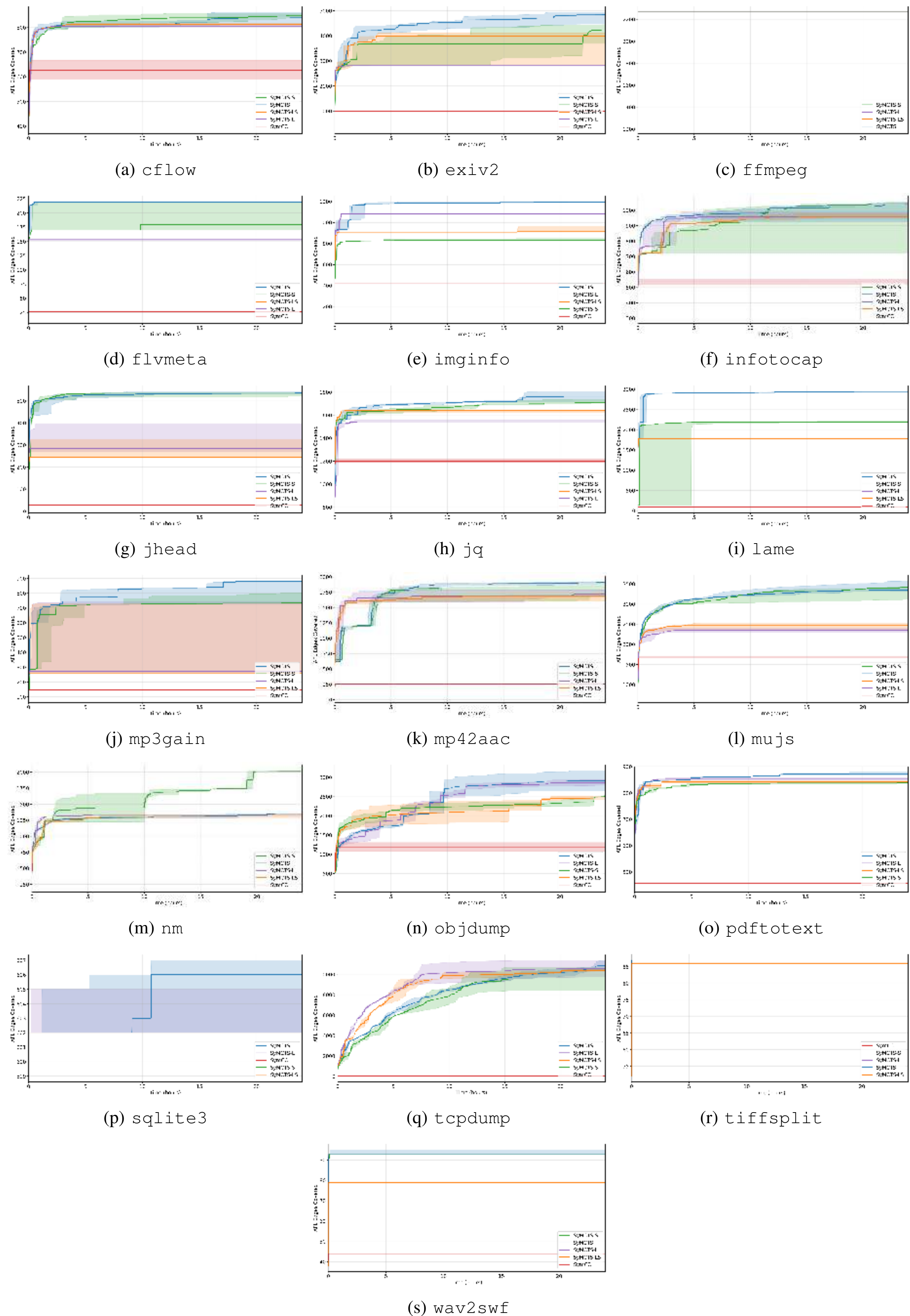
Appendix A.
Unibench saturated OSS-Fuzz corpus mapping

TABLE 7: OSS-Fuzz corpus mapping for saturated corpus configuration. Targets that are not listed use only UniBench seeds.

UniBench Target	OSS-Fuzz Project	OSS-Fuzz Fuzzer
cflow	llvm	clang-fuzzer
exiv2	exiv2	fuzz-read-print-write
imginfo	libjpeg-turbo	libjpeg_turbo_fuzzer
jhead	libjpeg-turbo	libjpeg_turbo_fuzzer
lame	lame	fuzzer-encoder
mp3gain	mpg123	read_fuzzer
mp42aac	ffmpeg	DEMUXER_fuzzer
mujs	quickjs	fuzz_eval
nm	binutils	fuzz_nm
objdump	binutils	fuzz_objdump
pdftotext	poppler	pdf_fuzzer
sqlite3	sqlite3	ossfuzz
tcpdump	libpcap	fuzz_both
tiffsplit	libtiff	tiff_read_rgba_fuzzer
wav2swf	ffmpeg	DEMUXER_fuzzer

Appendix B.
UniBench Evaluation - Full Results

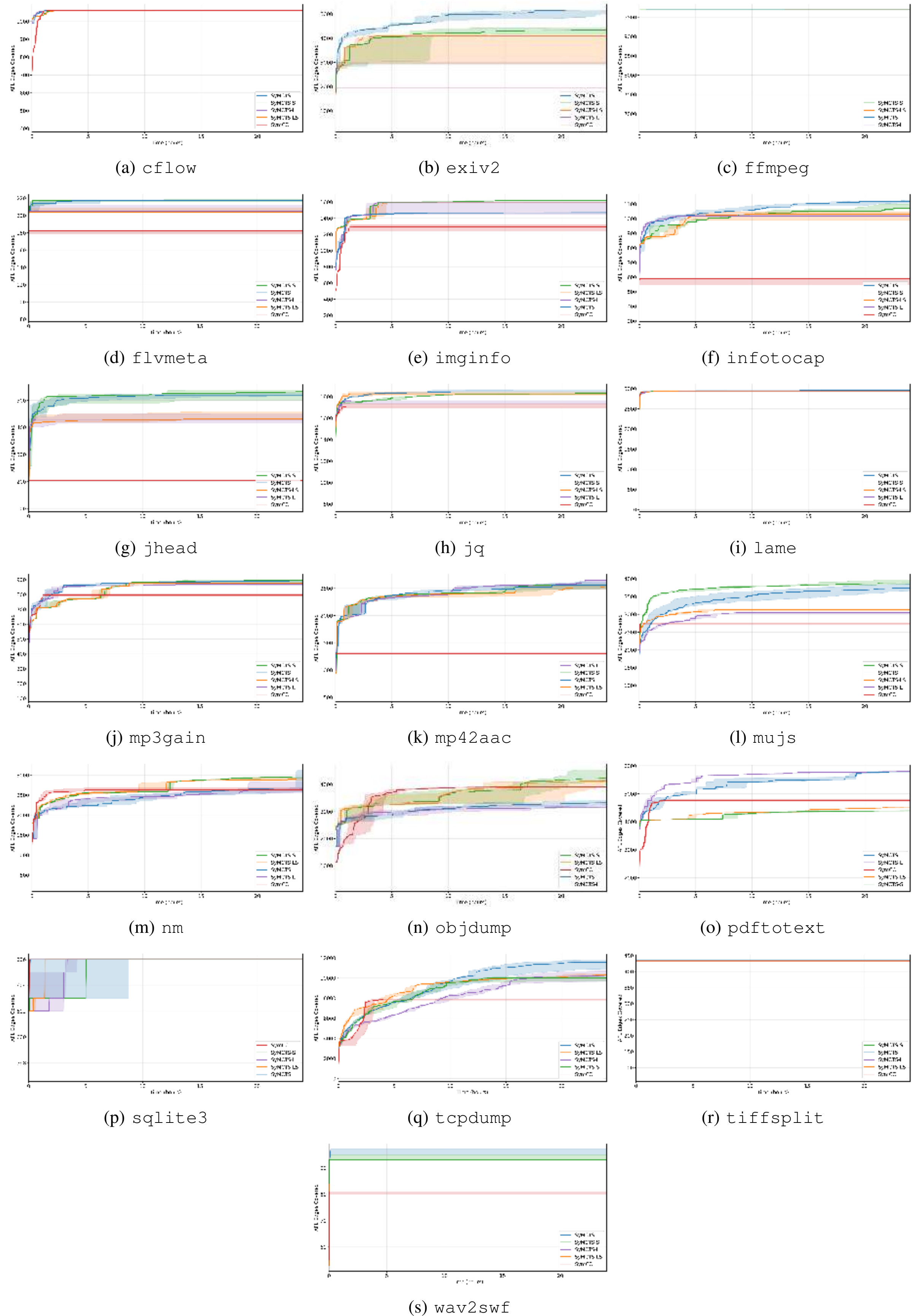
B.1. Empty Corpus



(a) cflow (b) exiv2 (c) ffmpeg
(d) flvmeta (e) imginfo (f) infotocap
(g) jhead (h) jq (i) lame
(j) mp3gain (k) mp42aac (l) mujs
(m) nm (n) objdump (o) pdftotext
(p) sqlite3 (q) tcpdump (r) tiffsplit
(s) wav2swf

Figure 6: Coverage for the UniBench evaluation using a empty corpus.

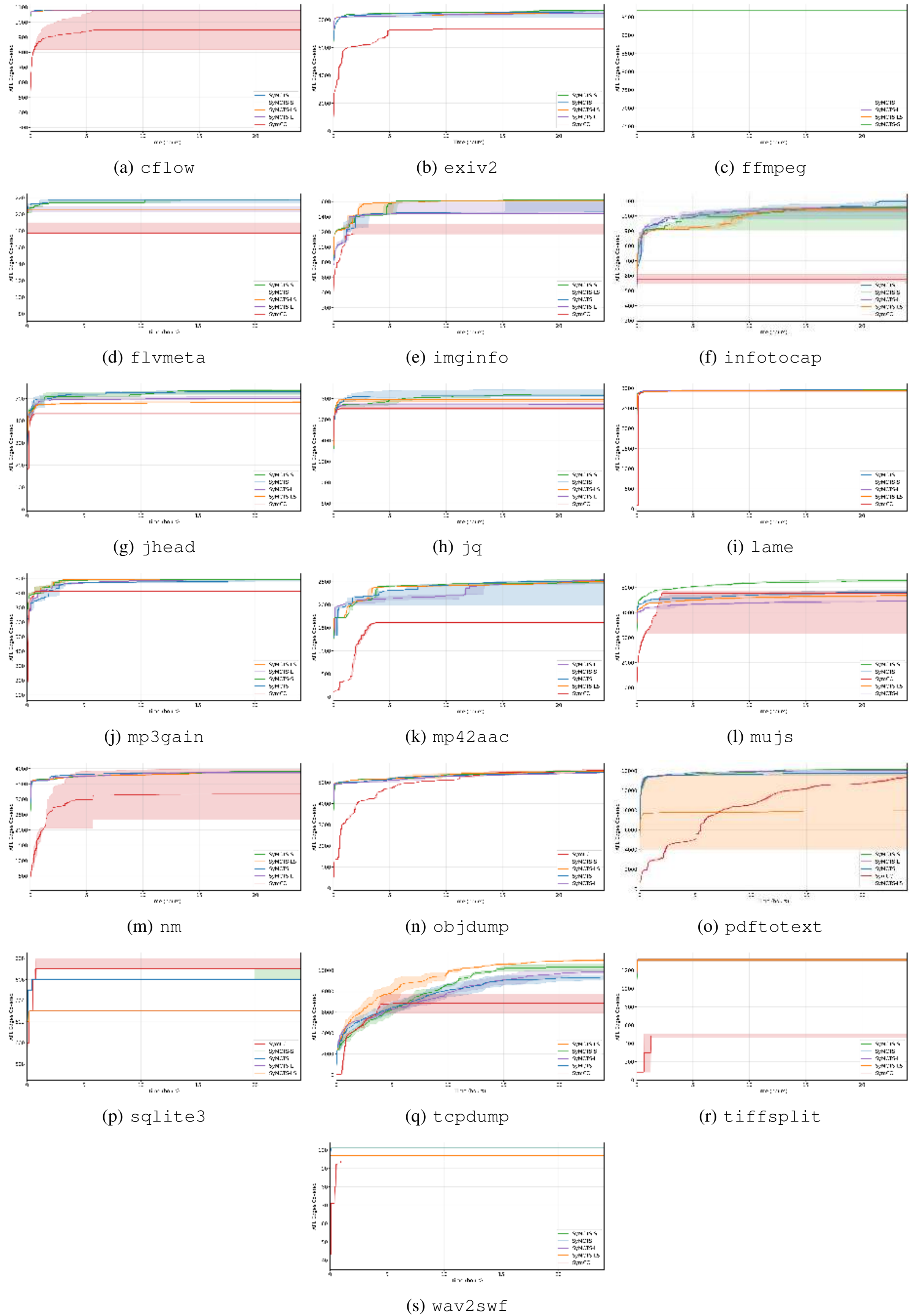
B.2. Unibench Corpus



(a) cflow (b) exiv2 (c) ffmpeg (d) flvmeta (e) imginfo (f) infotocap (g) jhead (h) jq (i) lame (j) mp3gain (k) mp42aac (l) mujs (m) nm (n) objdump (o) pdftotext (p) sqlite3 (q) tcpdump (r) tiffsplit (s) wav2swf

Figure 7: Coverage for the UniBench evaluation using a unibench corpus.

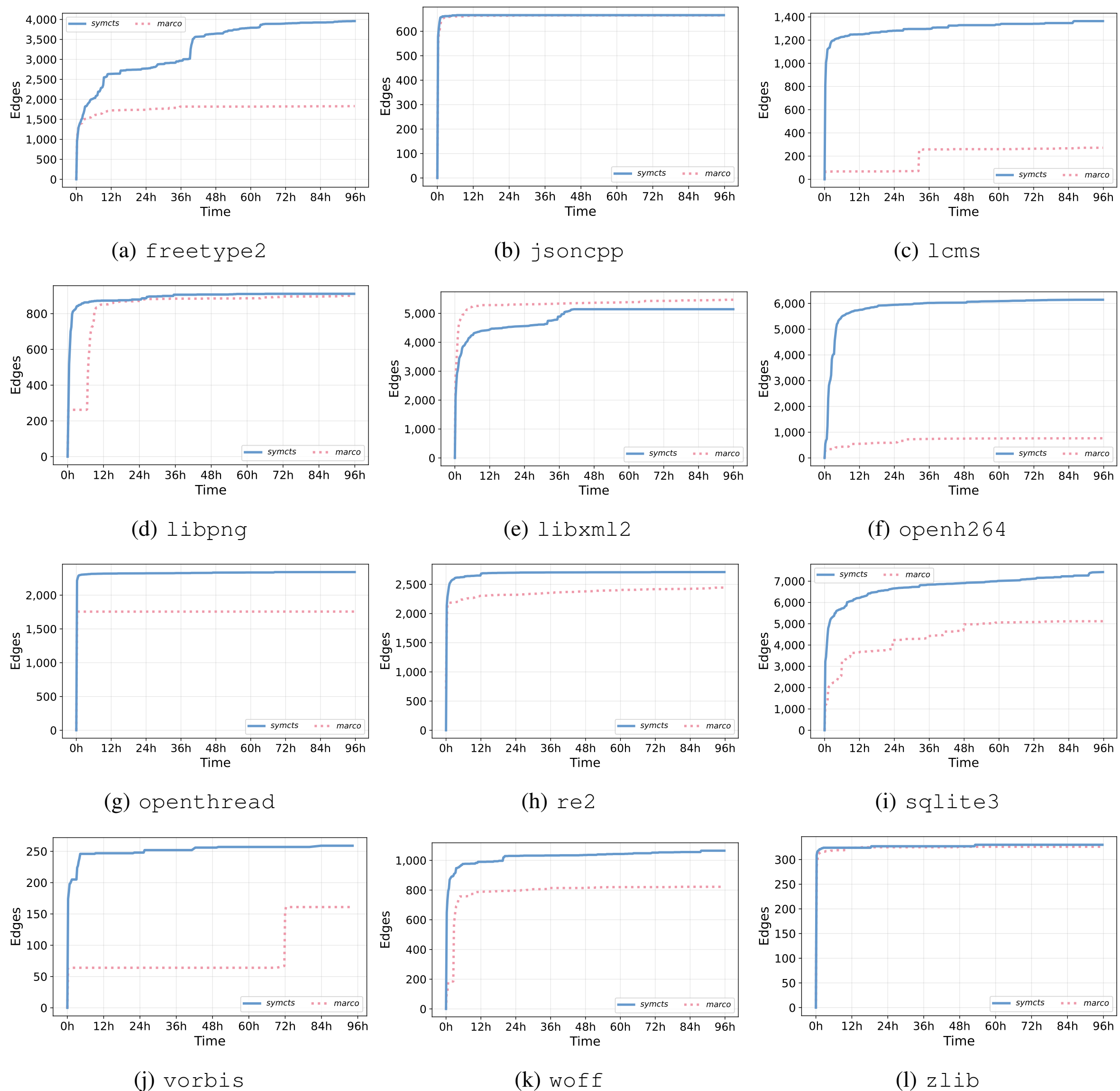
B.3. Saturated Corpus



(a) cflow (b) exiv2 (c) ffmpeg (d) flvmeta (e) imginfo (f) infotocap (g) jhead (h) jq (i) lame (j) mp3gain (k) mp42aac (l) mujs (m) nm (n) objdump (o) pdftotext (p) sqlite3 (q) tcpdump (r) tiffsplit (s) wav2swf

Figure 8: Coverage for the UniBench evaluation using a saturated corpus.

Appendix C.
Marco Evaluation



(a) `freetype2` (b) `jsoncpp` (c) `lcms`

(d) `libpng` (e) `libxml2` (f) `openh264`

(g) `openthread` (h) `re2` (i) `sqlite3`

(j) `vorbis` (k) `woff` (l) `zlib`

Figure 9: Median coverage results of the evaluation against Marco on 12 FuzzBench targets.

Appendix D.
FuzzBench Full Results

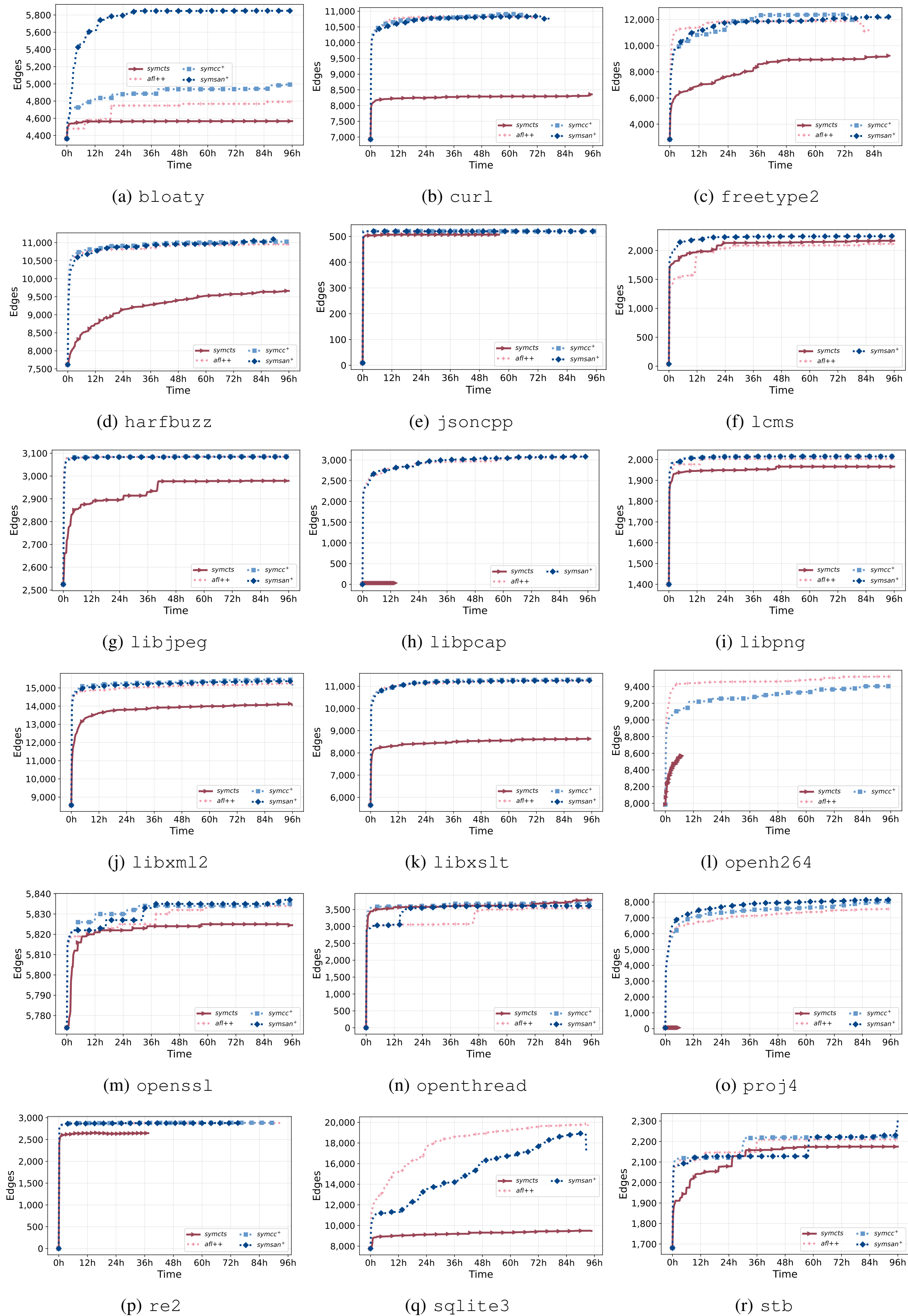
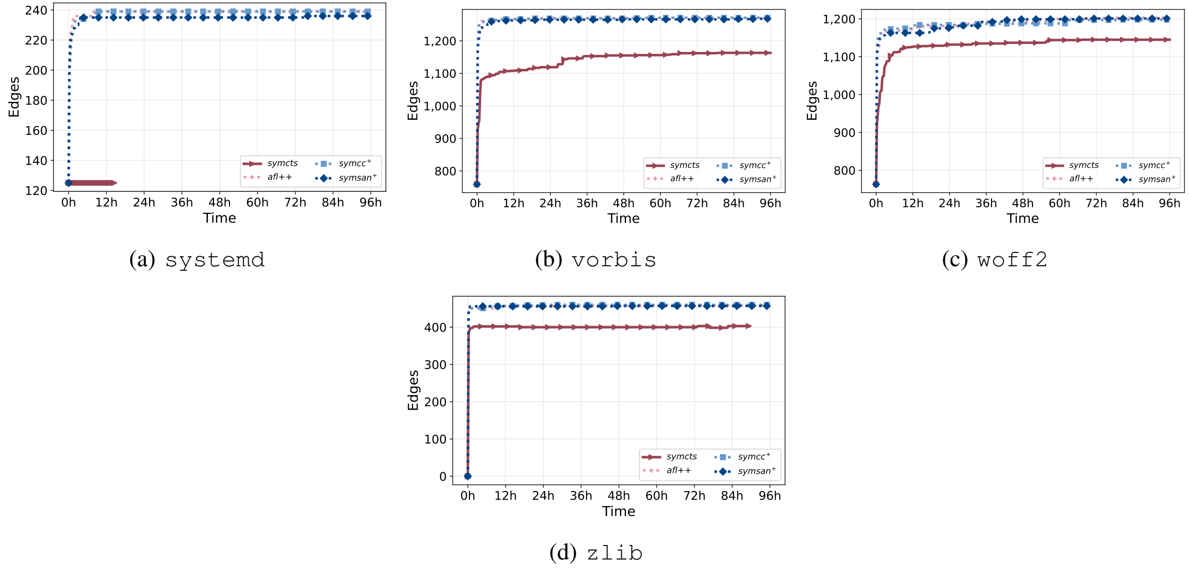


Figure 10: Median coverage results in FuzzBench



(a) systemd (b) vorbis (c) woff2

(d) zlib

Figure 11: Median coverage in FuzzBench (cont.)

	bloaty	curl	freetype2	harfbuzz	jsoncpp	lcms	libjpeg	libpcap
SYMCTS	4567.0	8357.0	9226.5	9662.0	507.0	2168.0	2979.0	31.0
SymCC+	4992.0	**10885.0**	11977.0	11018.0	**520.0**	/	**3086.0**	/
SymSAN+	**5849.0**	10766.0	**12197.0**	**11098.0**	**520.0**	**2249.0**	3084.0	3084.0
AFL++	4799.0	10806.0	11176.0	10926.0	**520.0**	2114.0	3085.0	**3087.0**

	libpng	libxml2	libxslt	openh264	openssl	openthread	proj4	re2
SYMCTS	1966.0	14066.0	8635.0	8570.0	5824.5	**3787.5**	58.0	2647.0
SymCC+	**2015.0**	**15445.0**	**11298.0**	9405.0	5835.0	3690.0	8012.0	**2884.0**
SymSAN+	**2015.0**	15149.0	11259.0	/	**5837.0**	3606.0	**8139.0**	2880.5
AFL++	2003.0	15251.0	11249.0	**9520.0**	5834.0	3541.0	7562.0	2880.0

	sqlite3	stb	systemd	vorbis	woff2	zlib
SYMCTS	9467.0	2175.0	125.0	1163.0	1145.0	403.0
SymCC+	/	2224.0	**239.0**	**1271.0**	1198.0	**460.0**
SymSAN+	17059.0	**2305.0**	236.0	1268.0	1201.0	457.0
AFL++	**19659.0**	2210.0	**239.0**	**1271.0**	**1203.0**	458.0

TABLE 8: Median coverage in FuzzBench.