Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing

Stefan Nagy
Virginia Tech
snagy2@vt.edu

Matthew Hicks
Virginia Tech
mdhicks2@vt.edu

***Abstract*—Coverage-guided fuzzing is one of the most successful approaches for discovering software bugs and security vulnerabilities. Of its three main components: (1) test case generation, (2) code coverage tracing, and (3) crash triage, code coverage tracing is a dominant source of overhead. Coverage-guided fuzzers trace every test case's code coverage through either static or dynamic binary instrumentation, or more recently, using hardware support. Unfortunately, tracing *all* test cases incurs significant performance penalties—even when the overwhelming majority of test cases and their coverage information are discarded because they do *not* increase code coverage.**

To eliminate needless tracing by coverage-guided fuzzers, we introduce the notion of coverage-guided tracing. Coverage-guided tracing leverages two observations: (1) only a fraction of generated test cases increase coverage, and thus require tracing; and (2) coverage-increasing test cases become less frequent over time. Coverage-guided tracing encodes the current frontier of coverage in the target binary so that it self-reports when a test case produces new coverage—without tracing. This acts as a filter for tracing; restricting the expense of tracing to only coverage-increasing test cases. Thus, coverage-guided tracing trades increased time handling coverage-increasing test cases for decreased time handling non-coverage-increasing test cases.

To show the potential of coverage-guided tracing, we create an implementation based on the static binary instrumentor Dyninst called UnTracer. We evaluate UnTracer using eight real-world binaries commonly used by the fuzzing community. Experiments show that after only an hour of fuzzing, UnTracer's average overhead is below 1%, and after 24-hours of fuzzing, UnTracer approaches 0% overhead, while tracing every test case with popular white- and black-box-binary tracers AFL-Clang, AFL-QEMU, and AFL-Dyninst incurs overheads of 36%, 612%, and 518%, respectively. We further integrate UnTracer with the state-of-the-art hybrid fuzzer QSYM and show that in 24-hours of fuzzing, QSYM-UnTracer executes 79% and 616% more test cases than QSYM-Clang and QSYM-QEMU, respectively.

***Keywords*—Fuzzing, software security, code coverage.**

I. Introduction

Software vulnerabilities remain one of the most significant threats facing computer and information security [1]. Real-world usage of weaponized software exploits by nation-states and independent hackers continues to expose the susceptibility of society's infrastructure to devastating cyber attacks. For defenders, existing memory corruption and control-flow safeguards offer incomplete protection. For software developers, manual code analysis does not scale to large programs. Fuzzing, an automated software testing technique, is a popular approach for discovering software vulnerabilities due to its speed, simplicity, and effectiveness [2], [3], [4], [5].

At a high level, fuzzing consists of (1) generating test cases, (2) monitoring their effect on the target binary's execution, and (3) triaging bug-exposing and crash-producing test cases. State-of-the-art fuzzing efforts center on coverage-guided fuzzing [5], [4], [6], [7], [8], [9], which augments execution with control-flow tracking apparatuses to trace test cases' code coverage (the exact code regions they execute). Tracing enables coverage-guided fuzzers to focus mutation on a small set of unique test cases (those that reach previously-unseen code regions). The goal being complete exploration of the target binary's code.

Code coverage is an abstract term that takes on three concrete forms in fuzzing literature: basic blocks, basic block edges, and basic block paths. For white-box (source-available) binaries, code coverage is measured through instrumentation inserted at compile-time [4], [5], [6]. For black-box (source-unavailable) binaries, it is generally measured through instrumentation inserted dynamically [5], [7] or statically through binary rewriting [10], or through instrumentation-free hardware-assisted tracing [11], [12], [4].

Tracing code coverage is costly—the largest source of time spent for most fuzzers—and the resulting coverage information is commonly discarded, as most test cases do *not* increase coverage. As our results in Section VI show, AFL [5]—one of the most popular fuzzers—faces tracing overheads as high as 1300% for black-box binaries and as high as 70% for white-box binaries. These overheads are significant because, as experiments in Section III-B show, over 90% of the time spent fuzzing involves executing and tracing test cases. The problem with spending all this effort on coverage tracing is that most test cases and their coverage information are discarded; because, for most benchmarks in our evaluation, **less than 1 in 10,000** of all test cases are coverage-increasing. Thus, the current practice of blindly tracing the coverage of every test case is incredibly wasteful.

This paper introduces the idea of coverage-guided tracing, and its associated implementation UnTracer, targeted at reducing the overheads of coverage-guided fuzzers. Coverage-guided tracing's goal is to restrict tracing to test cases *guaranteed* to increase code coverage. It accomplishes this by transforming the target binary so that it self-reports when a test case increases coverage. We call such modified binaries *interest oracles*. Interest oracles execute at native speeds because they eliminate the need for coverage tracing. In the event that the interest oracle reports a test case is coverage-increasing, the test case is marked as coverage-increasing and conventional tracing is used to collect code coverage. Portions of the interest oracle are then unmodified to reflect the additional coverage and the fuzzing process continues. By doing this, coverage-guided tracing pays a high cost for handling coverage-increasing test cases (about 2x the cost of tracing alone in our experiments), for the ability to run all test

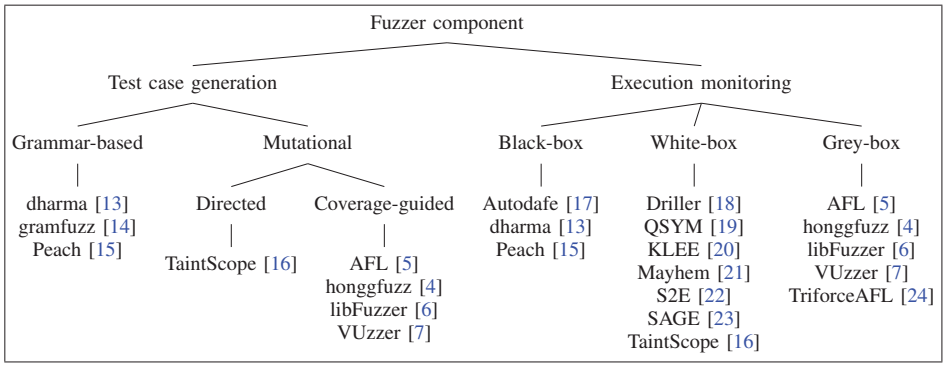

Fig. 1. A taxonomy of popular fuzzers by test case generation and program analysis approaches.

cases (initially) at native speed. To validate coverage-guided tracing and explore its tradeoffs on real-world software, we implement UnTracer. UnTracer leverages the black-box binary instrumentor Dyninst [25] to construct the interest oracle and tracing infrastructure.

We evaluate UnTracer alongside several coverage tracers used with the popular fuzzer AFL [5]. For tracing black-box binaries, we compare against the dynamic binary rewriter AFL-QEMU [5], and the static binary rewriter AFL-Dyninst [25]. For tracing white-box binaries, we compare against AFL-Clang [5]. To capture a variety of target binary and tracer behaviors, we employ a set of eight real-world programs of varying class and complexity (e.g., cryptography and image processing) that are common to the fuzzing community. In keeping with previous work, we perform evaluations for a 24-hour period and use 5 test case datasets per benchmark to expose the effects of randomness. Our results show UnTracer outperforms blindly tracing all test cases: UnTracer has an average run time overhead of 0.3% across all benchmarks, while AFL-QEMU averages 612% overhead, AFL-Dyninst averages 518% overhead, and AFL-Clang averages 36% overhead. Experimental results also show that the rate of coverage-increasing test cases rapidly approaches zero over time and would need to increase four orders-of-magnitude to ameliorate the need for UnTracer—even in a white-box tracing scenarios. We further integrate UnTracer with the state-of-the-art hybrid fuzzer QSYM [19]. Results show that QSYM-UnTracer averages 79% and 616% more executed test cases than QSYM-Clang and QSYM-QEMU, respectively.

In summary, this paper makes the following contributions:

- We introduce coverage-guided tracing: an approach for reducing fuzzing overhead by restricting tracing to coverage-increasing test cases.
- We quantify the infrequency of coverage-increasing test cases across eight real-world applications.
- We show that, for two coverage-guided fuzzers of different type: AFL ("blind" test case generation) and Driller ("smart" test case generation), they spend a majority of their time on tracing test cases.
- We implement and evaluate UnTracer; UnTracer is our coverage-guided tracer based on the Dyninst black-box binary instrumentor. We evaluate UnTracer against three popular, state-of-the-art white- and black-box binary fuzzing tracing approaches: AFL-Clang (white-box), AFL-QEMU (black-box, dynamic binary rewriting), and AFL-Dyninst (black-box, static binary rewriting), using eight real-world applications.
- We integrate UnTracer with the state-of-the-art hybrid fuzzer QSYM, and show that QSYM-UnTracer outperforms QSYM-Clang and QSYM-QEMU.
- We open-source our evaluation benchmarks [26], experimental infrastructure [27], and an AFL-based implementation of UnTracer [28].

II. Background

In this section, we first discuss fuzzers' defining characteristics, and how they relate to UnTracer. Second, we provide a detailed overview of coverage-guided fuzzing and how current fuzzers measure code coverage. Third, we discuss related work on the performance of coverage tracing for fuzzing. We conclude with our guiding research questions and principles.

A. An Overview of Fuzzing

Fuzzing is one of the most efficient and effective techniques for discovering software bugs and vulnerabilities. Its simplicity and scalability have led to its widespread adoption among both bug hunters [5], [4] and the software industry [2], [3]. Fundamentally, fuzzers operate by generating enormous amounts of test cases, monitoring their effect on target binary execution behavior, and identifying test cases responsible for bugs and crashes. Fuzzers are often classified by the approaches they use for test case generation and execution monitoring (Figure 1).

Fuzzers generate test cases using one of two approaches: grammar-based [29], [13], [14], [15] or mutational [30], [5], [4], [7], [6]. Grammar-based generation creates test cases constrained by some pre-defined input grammar for the target binary. Mutational generation creates test cases using other test cases; in the first iteration, by mutating some valid "seed" input accepted by the target binary; and in subsequent iterations, by mutating prior iterations' test cases. For large applications, input grammar complexity can be burdensome, and for proprietary applications, input grammars are seldom available. For these reasons, most popular fuzzers are mutational. Thus, coverage-guided tracing focuses on mutational fuzzing.

Most mutational fuzzers leverage program analysis to strategize which test cases to mutate. Directed fuzzers [31], [32] aim to reach specific locations in the target binary; thus they prioritize mutating test cases that seem to make progress toward those locations. Coverage-guided fuzzers [5], [4], [7], [6] aim to explore the entirety of the target binary's code; thus they favor mutating test cases that reach new code regions. As applications of directed fuzzing are generally niche, such as taint tracking [16] or patch testing [31], coverage-guided

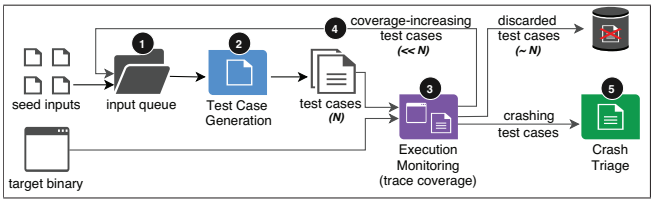

Fig. 2. High-level architecture of a coverage-guided mutational fuzzer.

fuzzing's wider scope makes it more popular among the fuzzing community [5], [6], [4], [3]. Coverage-guided tracing is designed to enhance coverage-guided fuzzers.

Fuzzers are further differentiated based on the degree of program analysis they employ. Black-box fuzzers [17], [13], [15] only monitor input/output execution behavior (e.g., crashes). White-box fuzzers [33], [23], [21], [18], [16], [20], [22] use heavy-weight program analysis for fine-grained execution path monitoring and constraint solving. Grey-box fuzzers [5], [4], [7], [6], [24], [31], [8] are a tradeoff between both—utilizing lightweight program analysis (e.g., code coverage tracing). Coverage-guided grey-box fuzzers are widely used in practice today; examples include VUzzer [7], Google's libFuzzer [6], honggfuzz [4], and AFL [5]. Our implementation of coverage-guided tracing (UnTracer) is built atop the coverage-guided grey-box fuzzer AFL [5].

B. Coverage-Guided Fuzzing

Coverage guided fuzzing aims to explore the entirety of the target binary's code by maximizing generated test cases' code coverage. Figure 2 highlights the high-level architecture of a coverage-guided mutational fuzzer. Given a target binary and some initial set of input seeds, S, fuzzing works as follows:

1) Queue all initial seeds[1] $s \in S$ for mutation.
2) **test case generation:** Select a queued seed and mutate it many times, producing test case set T.
3) **Execution monitoring:** For all test cases $t \in T$, trace their code coverage and look for crashes.
4) If a test case is *coverage-increasing*, queue it as a seed, and prioritize it for the next round of mutation. Otherwise, discard it.
5) **Crash triage:** Report any crashing test cases.
6) Return to step 2 and repeat.

Coverage-guided fuzzers trace code coverage during execution via binary instrumentation [5], [6], [4], system emulation [5], [11], [24], or hardware-assisted mechanisms [11], [4], [12]. All coverage-guided fuzzers are based on one of three metrics of code coverage: *basic blocks*, basic block *edges*, or basic block *paths*. Basic blocks (Figure 3) refer to straight-lined sequences of code terminating in a control-flow transfer instruction (e.g., jumps or returns); they form the nodes of a program's control-flow graph.

A basic block edge represents the actual control-flow transfer. It is possible to represent edge coverage as a set of `(src , dest)` tuples, where `src` and `dest` are basic blocks. Representing edge coverage this way (i.e., solely of basic blocks) allows edge coverage to be inferred from block coverage. The caveat is that this requires prior elimination of all critical edges, i.e., edges whose starting/ending basic blocks have multiple outgoing/incoming edges, respectively (details in Section VIII-B). honggFuzz [4], libFuzzer [6], and AFL [5] are fuzzers that track coverage at edge granularity. honggFuzz and libFuzzer track edge coverage indirectly using block coverage, while AFL tracks edge coverage directly (although it stores the information approximately in a 64KB hash table [34]).

[1] *Seeds* refers to test cases used as the basis for mutation. In the first iteration, the seeds are generally several small inputs accepted by the target binary.

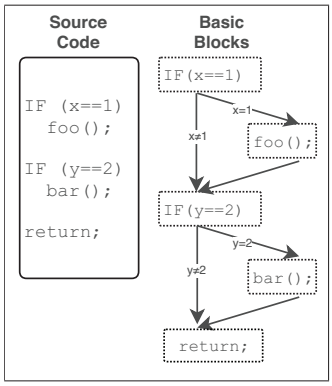

Fig. 3. An example of basic blocks in C code.

To date, no fuzzers that we are aware of track coverage at path granularity, however, we can imagine future approaches leveraging Intel Processor Trace's [35] ability to make tracking path coverage tractable. Thus, coverage-guided tracing complements coverage-guided fuzzers that trace block or edge coverage at block granularity.

C. Coverage Tracing Performance

Coverage-guided fuzzing of *white-box (source-available) binaries* typically uses instrumentation inserted at compile/assembly-time [5], [6], [4], allowing for fast identification and modification of basic blocks from source. AFL accomplishes this through custom GCC and Clang wrappers. honggfuzz and libFuzzer also provide their own Clang wrappers. Fuzzing *black-box (source-unavailable) binaries* is far more challenging, as having no access to source code requires costly reconstruction of binary control-flow. VUzzer [7] uses PIN [36] to dynamically (during run-time) instrument black-box binaries. AFL's QEMU user-mode emulation also instruments dynamically, but as our experiments show (Section VI), it incurs overheads as high as 1000% compared to native execution. To address the weakness of dynamic rewriting having to translate basic blocks in real-time—potentially multiple times—Cisco-Talos provides a static binary rewriter AFL-Dyninst [10]. While previous work shows AFL-Dyninst significantly outperforms AFL-QEMU on select binaries [37], results in Section VI suggest that the performance gap is much narrower.

D. Focus of this Paper

A characteristic of coverage-guided fuzzing is the coverage tracing of *all* generated test cases. Though "smarter" fuzzing efforts generate coverage-increasing test cases with higher frequency, results in Section III show that only a small percentage of all test cases are coverage-increasing. We draw inspiration from Amdahl's Law [38], realizing that the common case—the tracing of non-coverage-increasing test cases—presents an opportunity to substantially improve the performance of coverage-guided fuzzing. Thus we present coverage-guided tracing, which restricts tracing to only coverage-increasing test

	bsdtar	cert-basic	cjson	djpeg	pdftohtml	readelf	sfconvert	tcpdump	**avg.**
AFL-Clang	89.4	91.9	86.0	94.7	98.4	86.9	99.2	88.3	91.8
AFL-QEMU	95.7	98.9	95.7	97.8	99.5	96.5	98.6	95.8	97.3

	CADET_1	CADET_3	CROMU_1	CROMU_2	CROMU_3	CROMU_4	CROMU_5	CROMU_6	**avg.**
Driller-AFL	97.6	97.1	96.0	94.9	96.0	93.1	97.5	94.9	95.9

TABLE I. PER-BENCHMARK PERCENTAGES OF TOTAL FUZZING RUNTIME SPENT ON TEST CASE EXECUTION AND COVERAGE TRACING BY AFL-CLANG AND AFL-QEMU ("BLIND" FUZZING), AND DRILLER-AFL ("SMART" FUZZING). WE RUN EACH FUZZER FOR ONE HOUR PER BENCHMARK.

	bsdtar	cert-basic	cjson	djpeg	pdftohtml	readelf	sfconvert	tcpdump	**avg.**
AFL-Clang	1.63E−5	4.47E−5	2.78E−6	4.30E−5	1.42E−4	7.43E−5	8.77E−5	8.55E−5	6.20E−5
AFL-QEMU	3.34E−5	4.20E−4	1.41E−5	1.09E−4	6.74E−4	2.28E−4	4.25E−4	1.55E−4	2.57E−4

	CADET_1	CADET_3	CROMU_1	CROMU_2	CROMU_3	CROMU_4	CROMU_5	CROMU_6	**avg.**
Driller-AFL	2.70E−5	4.00E−4	2.06E−5	2.67E−5	2.33E−5	8.65E−7	1.61E−5	8.45E−6	6.53E−5

TABLE II. PER-BENCHMARK RATES OF COVERAGE-INCREASING TEST CASES OUT OF ALL TEST CASES GENERATED IN ONE HOUR BY AFL-CLANG AND AFL-QEMU ("BLIND" FUZZING), AND DRILLER-AFL ("SMART" FUZZING).

cases. Our implementation, UnTracer, is a coverage-guided tracing framework for coverage-guided fuzzers.

III. IMPACT OF DISCARDED TEST CASES

Traditional coverage-guided fuzzers (e.g., AFL [5], libFuzzer [6], and honggfuzz [4]) rely on "blind" (random mutation-based) test case generation; coverage-increasing test cases are preserved and prioritized for future mutation, while the overwhelming majority are non-coverage-increasing and discarded along with their coverage information. To reduce rates of non-coverage-increasing test cases, several white-box and grey-box fuzzers employ "smart" test case generation. Smart mutation leverages source analysis (e.g., symbolic execution [18], program state [9], and taint tracking [39], [7]) to generate a higher proportion of coverage-increasing test cases. However, it is unclear if such fuzzers spend significantly more time on test case generation than on test case execution/coverage tracing or how effective smart mutation is at increasing the rate of coverage-increasing test cases.

In this section, we investigate the performance impact of executing/tracing non-coverage-increasing test cases in two popular state-of-the-art fuzzers—AFL (blind test case generation) [5] and Driller (smart test case generation) [18]. We measure the runtime spent by both AFL and Driller on executing/tracing test cases across eight binaries, for one hour each, and their corresponding rates of coverage-increasing test cases. Below, we highlight the most relevant implementation details of both fuzzers regarding test case generation and coverage tracing, and our experimental setup.

AFL: AFL [5] is a "blind" fuzzer as it relies on random mutation to produce coverage-increasing (coverage-increasing) test cases, which are then used during mutation.[2] AFL traces test case coverage using either QEMU-based dynamic instrumentation for black-box binaries or assembly/compile-time instrumentation for white-box binaries. We cover both options by evaluating AFL-QEMU and AFL-Clang.

Driller: Driller [18] achieves "smart" test case generation by augmenting blind mutation with selective concolic execution—solving path constraints symbolically (instead of by brute-force). Intuitively, Driller aims to outperform blind fuzzers by producing fewer non-coverage-increasing test cases; its concolic execution enables penetration of path constraints where blind fuzzers normally stall. We evaluate Driller-AFL (aka ShellPhuzz [40]). Like AFL, Driller-AFL also utilizes QEMU for black-box binary coverage tracing.

[2] A second, less-relevant factor influencing AFL's test case mutation priority is test case size. For two test cases exhibiting identical code coverage, AFL will prioritize the test case with smaller filesize [5].

A. Experimental Setup

For AFL-Clang and AFL-QEMU we use the eight benchmarks from our evaluation in Section VI. As Driller currently only supports benchmarks from the DARPA Cyber Grand Challenge (CGC) [41], we evaluate Driller-AFL on eight precompiled [42] CGC binaries. We run all experiments on the same setup as our performance evaluation (Section VI).

To measure each fuzzer's execution/tracing time, we insert timing code in AFL's test case execution function (`run_target()`). As timing occurs per-execution, this allows us to also log the total number of test cases generated. We count each fuzzer's coverage-increasing test cases by examining its AFL queue directory and counting all saved test cases AFL appends with tag `+cov`—its indicator that the test case increases code coverage.

B. Results

As shown in Table I, both AFL and Driller spend the majority of their runtimes on test case execution/coverage tracing across all benchmarks: AFL-Clang and AFL-QEMU average 91.8% and 97.3% of each hour, respectively, while Driller-AFL averages 95.9% of each hour. Table II shows each fuzzer's rate of coverage-increasing test cases across all one-hour trials. On average, AFL-Clang and AFL-QEMU have .0062% and .0257% coverage-increasing test cases out of all test cases generated in one hour, respectively. Driller-AFL averages .00653% coverage-increasing test cases out of all test cases in each one hour trial. These results show that coverage-guided fuzzers AFL (blind) and Driller (smart)—despite adopting different test case generation methodologies—*both spend the majority of their time executing and tracing the coverage of non-coverage-increasing test cases.*

IV. COVERAGE-GUIDED TRACING

Current coverage-guided fuzzers trace *all* generated test cases to compare their individual code coverage to some accumulated *global* coverage. Test cases with *new* coverage are retained for mutation and test cases without new coverage are discarded along with their coverage information. In Section III,

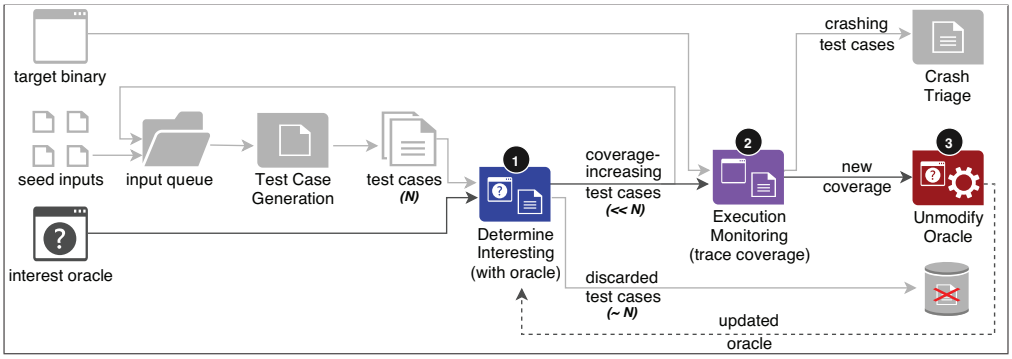

Fig. 4. Visualization of how coverage-guided tracing augments the workflow of a conventional coverage-guided grey-box fuzzer (e.g., AFL [5]). Coverage-guided tracing can also be similarly adapted into coverage-guided white-box fuzzers (e.g., Driller [18]).

we show that two coverage-guided fuzzers of different type—AFL ("blind") and Driller ("smart")—both spend the majority of their time executing/tracing non-coverage-increasing test cases. *Coverage-guided tracing* aims to trace *fewer* test cases by restricting tracing to *only* coverage-increasing test cases.

A. Overview

Coverage-guided tracing introduces an intermediate step between test case generation and code coverage tracing: the *interest oracle*. An interest oracle is a modified version of the target binary, where a pre-selected software interrupt is inserted via overwriting at the start of each uncovered basic block. Interest oracles restrict tracing to only coverage-increasing test cases as follows: test cases that trigger the oracle's interrupt are marked coverage-increasing, and then traced. As new basic blocks are recorded, their corresponding interrupts are removed from the oracle binary (*unmodifying*)—making it increasingly mirror the original target. As this process repeats, only test cases exercising *new* coverage trigger the interrupt—thus signaling them as coverage-increasing.

As shown in Figure 4, coverage-guided tracing augments conventional coverage-guided fuzzing by doing the following:

1) **Determine Interesting:** Execute a generated test case against the *interest oracle*. If the test case triggers the interrupt, mark it as coverage-increasing. Otherwise, return to step 1.
2) **Full Tracing:** For every coverage-increasing test case, trace its full code coverage.
3) **Unmodify Oracle:** For every *newly-visited* basic block in the test case's coverage, remove its corresponding interrupt from the interest oracle.
4) Return to step 1.

B. The Interest Oracle

In coverage-guided tracing, interest oracles sit between test case generation and coverage tracing—acting as a mechanism for filtering-out non-coverage-increasing test cases from being traced. Given a target binary, an interest oracle represents a modified binary copy with a software interrupt signal overwriting the start of each basic block. A test case is marked *coverage-increasing* if it triggers the interrupt—meaning it has entered some previously-uncovered basic block. Coverage-increasing test cases are then traced for their *full* coverage, and their newly-covered basic blocks are *unmodified* (interrupt removed) in the interest oracle.

Interest oracle construction requires prior identification of the target binary's basic block addresses. Several approaches for this exist in literature [43], [44], [45], and tools like angr [46] and Dyninst [25] can also accomplish this via static analysis. Inserting interrupts is trivial, but bears two caveats: first, while any interrupt signal can be used, it should avoid conflicts with other signals central to fuzzing (e.g., those related to crashes or bugs); second, interrupt instruction size must not exceed any candidate basic block's size (e.g., one-byte blocks cannot accommodate two-byte interrupts).

C. Tracing

Coverage-guided tracing derives coverage-increasing test cases' *full* coverage through a separate, tracing-only version of the target. As interest oracles rely on block-level binary modifications, code coverage tracing must also operate at block-level. Currently, block-level tracing can support either block coverage [7], or—if all critical edges are mitigated—edge coverage [4], [6]. Thus, coverage-guided tracing is compatible with most existing tracing approaches.

D. Unmodifying

Coverage-guided tracing's unmodify routine removes oracle interrupts in newly-covered basic blocks. Given a target binary, an interest oracle, and a list of newly-covered basic blocks, unmodifying overwrites each block's interrupt with the instructions from the original target binary.

E. Theoretical Performance Impact

Over time, a growing number of coverage-increasing test cases causes more of the oracle's basic blocks to be unmodified (Figure 5)—thus reducing the dissimilarity between oracle and target binaries. As the oracle more closely resembles the target, it becomes less likely that a test case will be coverage-increasing (and subsequently traced). Given that non-coverage-increasing test cases execute at the same speed for both the original and the oracle binaries, as fuzzing continues, coverage-guided tracing's overall performance approaches 0% overhead.

V. Implementation: UnTracer

Here we introduce *UnTracer*, our implementation of coverage-guided tracing. Below, we offer an overview of UnTracer's algorithm and discuss its core components in detail.

A. UnTracer Overview

UnTracer is built atop a modified version of the coverage-guided grey-box fuzzer, AFL 2.52b [5], which we selected due to both its popularity in the fuzzing literature [47], [18], [48], [8], [31], [24], [9], [49] and its open-source availability. Our implementation consists of 1200 lines of C and C++

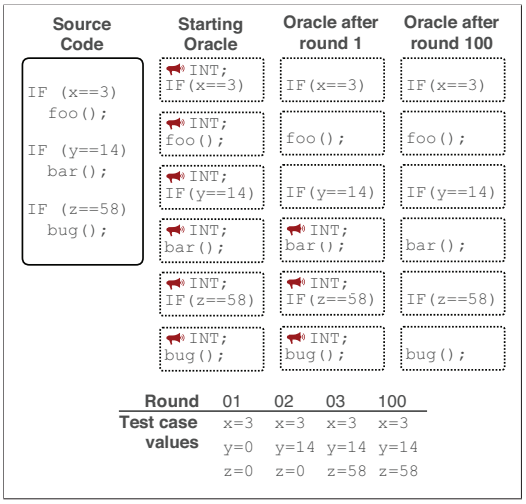

Fig. 5. An example of the expected evolution of a coverage-guided tracing interest oracle's basic blocks alongside its original source code. Here, `INT` denotes an oracle interrupt. For simplicity, this diagram depicts interrupts as inserted; however, in coverage-guided tracing, the interrupts instead overwrite the start of each basic block. Unmodifying basic blocks consists of resetting their interrupt-overwritten byte(s) to their original values.

code. UnTracer instruments two separate versions of the target binary—an *interest oracle* for identifying coverage-increasing test cases, and a *tracer* for identifying new coverage. As AFL utilizes a forkserver execution model [50], we incorporate this in both UnTracer's oracle and tracer binaries.

Algorithm 1 shows the steps UnTracer takes, as integrated with AFL. After AFL completes its initial setup routines (e.g., creating working directories and file descriptors) (line 1), UnTracer instruments both the oracle and tracer binaries (lines 2–3); the oracle binary gets a forkserver while the tracer binary gets a forkserver and basic block-level instrumentation for coverage tracing. As the oracle relies on block-level software interrupts for identifying coverage-increasing test cases, UnTracer first identifies all basic blocks using static analysis (line 5); then, UnTracer inserts the interrupt at the start of every basic block in the oracle binary (lines 6–8). To initialize both the oracle and tracer binaries for fuzzing, UnTracer starts their respective forkservers (lines 9–10). During AFL's main fuzzing loop (lines 11–23), UnTracer executes every AFL-generated test case (line 12) on the oracle binary (line 13). If any test case triggers an interrupt, UnTracer marks it as coverage-increasing (line 14) and uses the tracer binary to collect its coverage (line 15). We then stop the forkserver (line 16) to unmodify every newly-covered basic block (lines 17-19)—removing their corresponding oracle interrupts; this ensures only future test cases with new coverage will be correctly identified as coverage-increasing. After all newly-covered blocks have been unmodified, we restart the updated oracle's forkserver (line 20). Finally, AFL completes its coverage-increasing test case handling routines (e.g., queueing and prioritizing for mutation) (line 21) and fuzzing moves onto the next test case (line 12). Figure 6 depicts UnTracer's architecture.

B. Forkserver Instrumentation

During fuzzing, both UnTracer's oracle and tracer binaries are executed many times; the oracle executes all test cases to determine which are coverage-increasing and the tracer executes all coverage-increasing test cases to identify new

Algorithm 1: The UnTracer algorithm integrated in AFL.

Input: P: the target program
Data: b: a basic block
B: a set of basic blocks
i: an AFL-generated test case
Φ: the set of all coverage-increasing test cases

```
1  AFL_SETUP()
   // Instrument oracle and tracer binaries
2  P_O ← INSTORACLE(P)
3  P_T ← INSTTRACER(P)
   // Find and modify all of oracle's blocks
4  B = ∅
5  B ← GETBASICBLOCKS(P)
6  for b ∈ B do
7  |  MODIFYORACLE(b)
8  end
   // Start oracle and tracer forkservers
9  STARTFORKSERVER(P_O)
10 STARTFORKSERVER(P_T)
   // Main fuzzing loop
11 while 1 do
12 |  i ← AFL_WRITETOTEST CASE()
13 |  if P_O(i) → INTERRUPT then
   |  |  // The test case is coverage-increasing
14 |  |  Φ.ADD(i)
   |  |  // Trace test case's new coverage
15 |  |  B_trace ← GETTRACE(P_T(i))
   |  |  // Kill oracle before unmodifying
16 |  |  STOPFORKSERVER(P_O)
   |  |  // Unmodify test case's new coverage
17 |  |  for b ∈ B_trace do
18 |  |  |  UNMODIFYORACLE(b)
19 |  |  end
   |  |  // Restart oracle before continuing
20 |  |  STARTFORKSERVER(P_O)
21 |  |  AFL_HANDLECOVERAGEINCREASING()
22 |  end
23 end
```

coverage. In implementing UnTracer, we aim to optimize execution speeds of both binaries. Like other AFL tracers, UnTracer incorporates a *forkserver* execution model [50] in its tracer binary, as well as in its oracle binary. By launching new processes via `fork()`, the forkserver avoids repetitive process initialization—achieving significantly faster speeds than traditional `execve()`-based execution. Typically, instrumentation first inserts a forkserver function in a binary's `.text` region, and then links to it a callback in the first basic block of function `<main>`. In the tracer binary, we already use Dyninst's static binary rewriting for black-box binary instrumentation, so we use that same technique for the forkserver.

For the oracle binary, our initial approach was to instrument it using Dyninst. Unfortunately, preliminary evaluations revealed several performance problems.[3] Since the oracle executes every test case, it is performance critical. To avoid Dyninst's limitations, we leverage AFL's assembly-time instrumentation to insert the forkserver in the oracle binary, since it closely mimics the outcome of black-box binary rewriters.

C. Interest Oracle Binary

The oracle is a modified version of the target binary that adds the ability to self-report coverage-increasing test cases through the insertion of software interrupts at the start of

[3]We made Dyninst developers aware of several performance issues—specifically, excessive function calls (e.g., to `__dl_relocate_object`) after exiting the forkserver function. While they confirmed that this behavior is unexpected, they were unable to remedy these issues before publication.

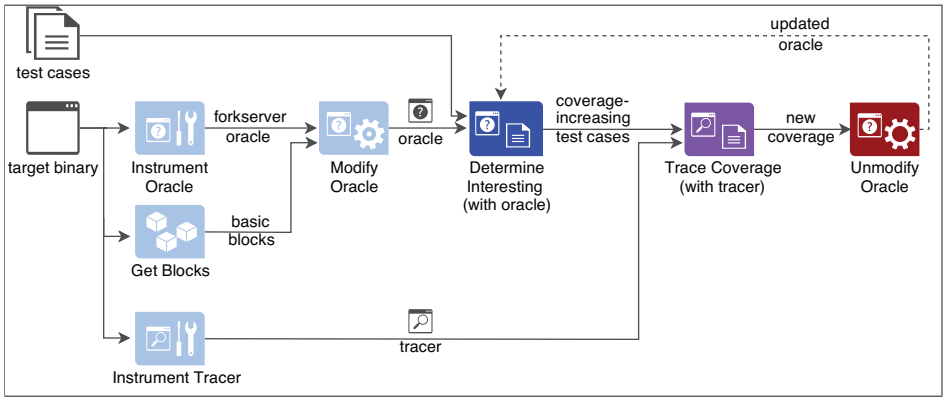

Fig. 6. UnTracer's workflow. Not shown is test case generation, or starting/stopping forkservers.

each *uncovered* basic block. Thus, if a test case triggers the interrupt, it has exercised some *new* basic block and is marked as coverage-increasing. Oracle binary construction requires prior knowledge of the target binary's basic block addresses. We leverage Dyninst's static control-flow analysis to output a list of basic blocks, then iterate through that list in using binary file IO to insert the interrupts. To prevent interrupts triggering before forkserver initialization, we do not consider functions executed prior to the forkserver callback in `<main>` (e.g., `<_start>`, `<__libc_start_main>`, `<_init>`, `<frame_dummy>`).

We use `SIGTRAP` for our interrupt for two reasons: (1) it has long been used for fine-grain execution control and analysis (e.g., `gdb` [51], [52] and kernel-probing [53], [54]); and (2) its binary representation—`0xCC`—is one byte long, making it possible to overwrite basic blocks of all sizes.

D. Tracer Binary

If the oracle determines a test case to be coverage-increasing, UnTracer extracts its new code coverage by executing it on a separate *tracer* binary—a coverage tracing-instrumented version of the target binary. We utilize Dyninst to statically instrument the tracer with a forkserver for fast execution, and coverage callbacks inserted in each of its basic blocks for coverage tracing. Upon execution, a basic block's callback appends its corresponding basic block address to a trace file located in UnTracer's working directory.

In an early version of UnTracer, we observed that coverage traces containing repeatedly-executing basic blocks add significant overhead in two ways: first, recording individual blocks multiple times—a common occurrence for binaries exhibiting looping behavior—slowed UnTracer's trace *writing* operations; second, trace *reading* is also penalized, as subsequent block-level unmodification operations are forced to process any repeatedly-executing basic blocks. To mitigate such overhead, we optimize tracing to only record *uniquely*-covered basic blocks as follows: in the tracer forkserver, we initialize a global hashmap data structure to track all covered basic blocks unique to each trace; as each tracing child is forked, it inherits the initial hashmap; upon a basic block's execution, its callback utilizes hashmap lookup to determine if the block has been previously covered in the current execution; if not, the callback updates the current trace log and updates the hashmap. With this optimization, for each coverage-increasing test case, UnTracer records a set of all uniquely-covered basic blocks, thus reducing the overhead resulting from logging, reading, and processing the same basic block multiple times.

E. Unmodifying the Oracle

When a test case triggers the oracle's software interrupt, it is marked as coverage-increasing and UnTracer removes its interrupts from its newly-covered basic blocks to ensure no future test case with the non-new coverage is marked coverage-increasing. For each newly-covered basic block reported in an coverage-increasing test case's trace log, UnTracer replaces the inserted interrupt with the original byte found in the target binary—effectively resetting it to its pre-modified state. Doing so means any future test cases executing this basic block will no longer trigger the interrupt and subsequently not be misidentified as coverage-increasing.

We observe that even coverage-increasing test cases often have significant overlaps in coverage. This causes UnTracer to attempt unmodifying many already-unmodified basic blocks, resulting in high overhead. To mitigate this, we introduce a hashmap data structure for tracking *global* coverage. Much like the hashmap used for per-trace redundant basic block filtering, before unmodifying any basic block from the trace log, UnTracer determines if the block has been seen in any previous trace via hashmap lookup. If so, the basic block is skipped. If not, its interrupt is removed, and the basic block is added to the hashmap. Thus, global coverage tracking ensures that only newly-covered basic blocks are processed.

VI. Tracing-only Evaluation

Our evaluation compares UnTracer against tracing all test cases with three widely used white- and black-box binary fuzzing tracing approaches—AFL-Clang (white-box) [5], AFL-QEMU (black-box dynamically-instrumented) [5], and AFL-Dyninst (black-box statically-instrumented) [10] on eight real-world benchmarks of different type.

Our experiments answer the following questions:

1) How does UnTracer (*coverage-guided tracing*) perform compared to tracing all test cases?
2) What factors contribute to UnTracer's overhead?
3) How is UnTracer's overhead impacted by the rate of coverage-increasing test cases?

A. Evaluation Overview

We compare UnTracer's performance versus popular white- and black-box fuzzing tracing approaches: AFL-Clang, AFL-QEMU, and AFL-Dyninst. These tracers all work with the

Package	Benchmark	Version	Class	Basic Blocks	Test Cases ($\cdot 10^6$)	Coverage-increasing Ratio	500ms Timeouts
`libarchive`	`bsdtar`	3.3.2	archiv	31379	21.06	1.47E−5	0
`libksba`	`cert-basic`	1.3.5	crypto	9958	10.73	1.50E−5	0
`cjson`	`cjson`	1.7.7	web	1447	25.62	1.48E−5	0
`libjpeg`	`djpeg`	9c	image	4844	14.53	1.33E−5	12133
`poppler`	`pdftohtml`	0.22.5	doc	54596	1.21	7.85E−5	0
`binutils`	`readelf`	2.30	dev	21249	14.89	8.98E−5	0
`audiofile`	`sfconvert`	0.2.7	audio	5603	10.17	3.91E−2	1137609
`tcpdump`	`tcpdump`	4.9.2	net	33743	27.14	3.73E−5	0

TABLE III. INFORMATION ON THE EIGHT BENCHMARKS USED IN OUR EVALUATION IN SECTIONS VI AND VII AND AVERAGES OVER 5 24-HOUR DATASETS FOR EACH BENCHMARK.

same fuzzer, AFL, and they cover the tracing design space including working with white- and black-box binaries as well as static and dynamic binary rewriting. Our evaluations examine each tracer's overhead on eight real-world, open-source benchmarks of different type, common to the fuzzing community. Table III provides benchmark details. To smooth the effects of randomness and ensure the most fair comparison of performance, we evaluate tracers on the same five input datasets per benchmark. Each dataset contains the test cases generated by fuzzing that benchmark with AFL-QEMU for 24 hours. Though our results show UnTracer has less than 1% overhead after one hour of fuzzing, we extend all evaluations to 24 hours to better match previous fuzzing evaluations.

We configure AFL to run with 500ms timeouts and leave all other parameters at their defaults. We modify AFL so that all non-tracing functionality is removed (e.g., progress reports) and instrument its `run_target()` function to collect per-test case timing. To address noise from the operating system and other sources, we perform eight trials of each dataset. For each set of trials per dataset, we apply trimmed-mean denoising [55] on each test case's tracing times; the resulting times represent each test case's median tracing performance.

All trials are distributed across two workstations—each with five single-core virtual machines. Both host systems run Ubuntu 16.04 x86_64 operating systems, with six-core Intel Core i7-7800X CPU @ 3.50GHz, and 64GB RAM. All 10 virtual machines run Ubuntu x86_64 18.04 using VirtualBox. We allocate each virtual machine 6GB of RAM.[4]

B. Experiment Infrastructure

To narrow our focus to tracing overhead, we only record time spent executing/tracing test cases. To maintain fairness, we run all tracers on the same five pre-generated test case datasets for each benchmark. For dataset generation, we implement a modified version of AFL that dumps its generated test cases to file. In our evaluations, we use QEMU as the baseline tracer (since our focus is black-box tracing) to generate the five datasets for each benchmark.

Our second binary—`TestTrace`—forms the backbone of our evaluation infrastructure. We implement this using a modified version of AFL—eliminating components irrelevant to tracing (e.g., test case generation and execution monitoring). Given a benchmark, pre-generated dataset, and tracing mode (i.e., AFL-Clang, AFL-QEMU, AFL-Dyninst, or none (a.k.a. *baseline*)), `TestTrace`: (1) reproduces the dataset's test cases one-by-one, (2) measures time spent tracing each test case's coverage, and (3) logs each trace time to file. For UnTracer, we include both the initial full-speed execution and any time spent handling coverage-increasing test cases.

C. Benchmarks

Our benchmark selection is based on popularity in the fuzzing community and benchmark type. We first identify candidate benchmarks from several popular fuzzers' trophy cases[5] and public benchmark repositories [5], [56], [4], [3], [57]. To maximize benchmark variety, we further partition candidates by their overall type—software **dev**elopment, **image** processing, data **archiv**ing, **net**work utilities, **audio** processing, **doc**ument processing, **crypto**graphy, and **web** development. After we filter out several candidate benchmarks based on incompatibility with our tracers (e.g., Dyninst-based instrumentation crashes on `openssl`), we select one benchmark per category: `bsdtar` (archiv), `cert-basic` (crypto), `cjson` (web), `djpeg` (image), `pdftohtml` (doc), `readelf` (dev), `sfconvert` (audio), and `tcpdump` (net).

For each benchmark, we measure several other metrics with potential effects on tracing overhead: number of basic blocks; and average number of generated test cases, average rate of coverage-increasing test cases, and average number of 500ms timeouts in 24 hours. Benchmark basic block totals are computed by enumerating all basic blocks statically using Dyninst [25]. For counting timeouts, we examined the statistics reported by `afl-fuzz-saveinputs` during dataset generation; using our specified timeout value (500ms), we then averaged the number of timeouts per benchmark among its datasets. Lastly, for each benchmark, we counted and averaged the number of test cases generated in all of its 24-hour datasets.

We compile each benchmark using Clang/LLVM, with all compiler options set to their respective benchmark-specific defaults. Below, we detail our additional tracer-specific benchmark configurations.

1) Baseline: AFL's forkserver-based execution model (also used by UnTracer's interest oracle and tracer binaries) adds a substantial performance improvement over `execve()`-based execution [50]. As each fuzzing tracer in our evaluation leverages forkserver-based execution, we design our "ground-truth" benchmark execution models to represent the fastest known execution speeds: a statically-instrumented forkserver without any coverage tracing. We use a modified copy of AFL's assembler (`afl-as`) to instrument *baseline* (forkserver-only) benchmark versions. In each benchmark trial, we use

[4] Across all trials, we saw no benchmarks exceeding 2GB of RAM usage.

[5] A fuzzer's "trophy case" refers to a collection of bugs/vulnerabilities reportedly discovered with that fuzzer.

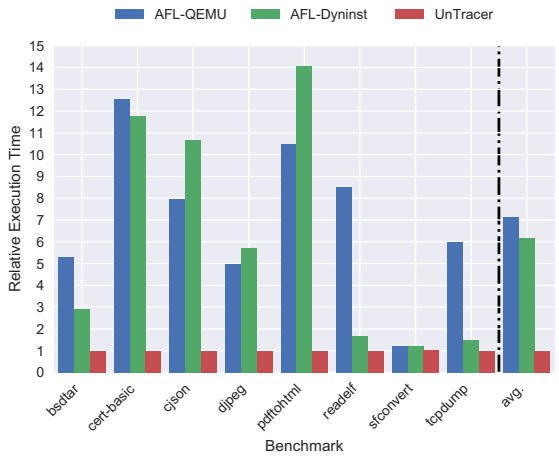

Fig. 7. Per-benchmark relative overheads of UnTracer versus black-box binary tracers AFL-QEMU and AFL-Dyninst.

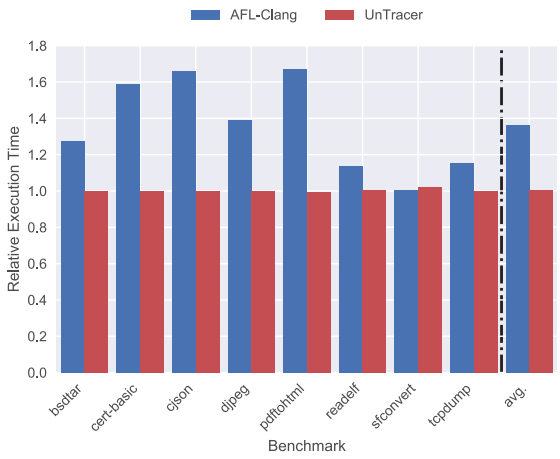

Fig. 8. Per-benchmark relative overheads of UnTracer versus white-box binary tracer AFL-Clang.

its baseline execution speeds as the basis for comparing each fuzzing tracers' overhead.

2) AFL-Clang: As compiling with AFL-GCC failed for some binaries due to changes in GCC, we instead use AFL-Clang.

3) AFL-QEMU: We only need to provide it the original uninstrumented target binary of each benchmark in our evaluation.

4) AFL-Dyninst: For our AFL-Dyninst evaluations, we instrument each binary using AFL-Dyninst's instrumenter with configuration parameters `bpatch.setDelayedParsing` set to true; `bpatch.setLivenessAnalysis` and `bpatch.setMergeTramp` false; and leave all other configuration parameters at their default settings.

D. Timeouts

Coverage tracing is affected by pre-defined execution *timeout* values. Timeouts act as a "hard limit"—terminating a test case's tracing if its duration exceeds the timeout's value. Though timeouts are necessary for halting infinitely-looping test cases, small timeouts prematurely terminate tracing. For long-running test cases, this results in missed coverage information. In cases where missed coverage causes coverage-increasing test cases to be misidentified as non-coverage-increasing, this will have cascading effects on test case generation. As coverage-guided fuzzers explore the target binary by mutating coverage-increasing test cases, exclusion of timed-out—but otherwise coverage-increasing—test cases results in a higher likelihood of generated test cases being non-coverage-increasing, and thus, slowing coverage indefinitely.

Small timeouts, when hit frequently, distort tracers' overheads, making their performance appear closer to each others'. In early experiments with timeouts of 100ms (AFL's default), we observed that, for some datasets, our worst-performing tracers (e.g., AFL-Dyninst, AFL-QEMU) had similar performance to otherwise faster white-box-based tracing (i.e., AFL-Clang). Upon investigating each tracer's logs, we found that all were timing-out on a significant percentage of the test cases. This was striking given that the baseline (forkserver-only) benchmark versions had significantly fewer timeouts. Thus, a 100ms timeout was too restrictive. We explored the effect of several different timeout values, with the goal of making each tracer's number of timeouts close to the baseline's (assumed ground truth).

E. UnTracer versus Coverage-agnostic Tracing

We examine our evaluation results to identify each fuzzing tracer's overhead per benchmark. For each tracer's set of trials per benchmark dataset, we employ trimmed-mean de-noising (shown to better reveal median tendency [55]) at test case level—removing the top and bottom 33% outliers—to reduce impact of system interference on execution speeds. We then take the resulting five trimmed-mean dataset overheads for each tracer-benchmark combination and average them to obtain tracer-benchmark overheads. Lastly, we convert all averaged tracer-benchmark overheads to *relative execution times* with respect to baseline (e.g., a relative execution time of 1.5 equates to 50% overhead).

In the following sections, we compare the performance of UnTracer to three popular *coverage-agnostic* tracing approaches. We first explore the performance of two black-box binary fuzzing tracers: AFL-QEMU (dynamic) and AFL-Dyninst (static). Secondly, we compare UnTracer's performance against that of the white-box binary fuzzing tracer AFL-Clang (static assembler-instrumented tracing).

1) Black-box binary tracing: As shown in Figure 7, we compare UnTracer's performance to two popular black-box binary fuzzing tracers—AFL's dynamically-instrumented tracing via QEMU user-mode emulation (AFL-QEMU) [58], and Dyninst-based static binary rewriting-instrumented tracing (AFL-Dyninst) [10]. For one benchmark (`sfconvert`), AFL-QEMU and AFL-Dyninst have similar relative execution times (1.2 and 1.22, respectively) to UnTracer (1.0); however; by looking at the different datasets for `sfconvert`, we observe a clear trend between higher number of timeouts and lower tracing overheads across all tracers (Table III). In our evaluations, a 500ms test case timeout significantly overshadows a typical test case execution of 0.1–1.0ms.

AFL-Dyninst outperforms AFL-QEMU in three benchmarks (`bsdtar, readelf, tcpdump`), but as these benchmarks all vary in complexity (e.g., number of basic blocks, execution times, etc.), we are unable to identify which benchmark characteristics are optimal for AFL-Dyninst's performance. Across all benchmarks, UnTracer achieves an aver-

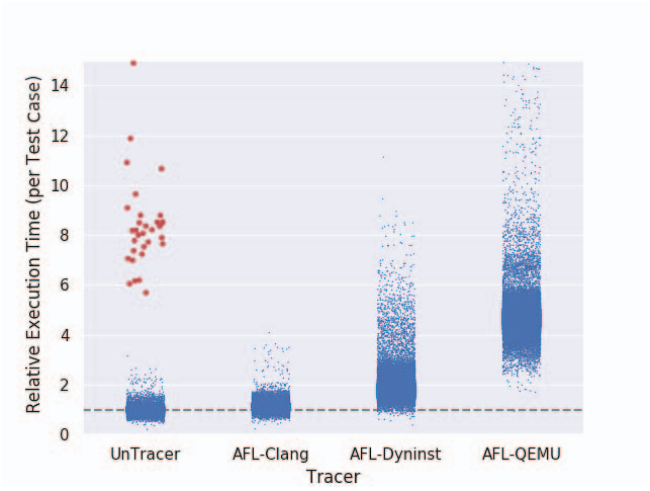

Fig. 9. Distribution of each tracer's relative execution time averaged per-test case for one 24-hour `cjson` dataset. The horizontal grey dashed line represents the average baseline execution speed. Red dots represent coverage-increasing test cases identified by UnTracer.

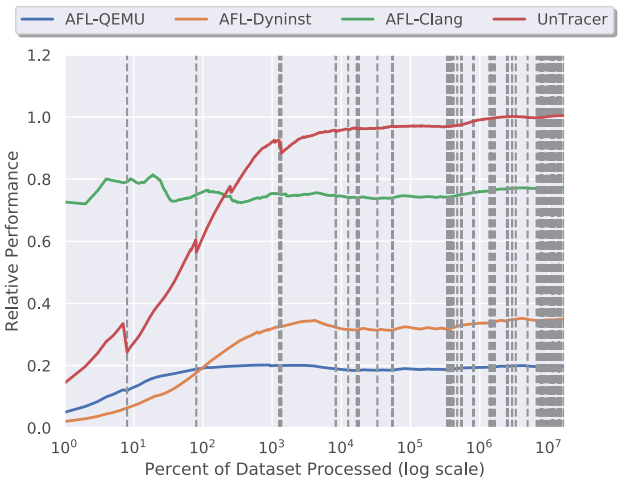

Fig. 10. Averaged relative performance of all tracers over the percentage of test cases processed for one 24-hour `bsdtar` dataset. Here, 1.0 refers to baseline (maximum) performance. Each grey dashed vertical line represents a coverage-increasing test case.

age relative execution time of 1.003 (0.3% overhead), while AFL-QEMU and AFL-Dyninst average relative execution times of 7.12 (612% overhead) and 6.18 (518% overhead), respectively. The average Relative Standard Deviation (RSD) for each tracer was less than 4%. In general, our results show UnTracer reduces the overhead of tracing black-box binaries by up to four orders of magnitude.

Mann Whitney U-test scoring: Following Klees et al.'s [59] recommendation, we utilize the Mann Whitney U-test to determine if UnTracer's execution overhead is stochastically smaller than AFL-QEMU's and AFL-Dyninst's. First we compute all per-dataset execution times for each benchmark[6] and tracer combination; then for each benchmark dataset we apply the Mann Whitney U-test with 0.05 significance level on execution times of UnTracer versus AFL-QEMU and UnTracer versus AFL-Dyninst. Averaging the resulting p-values for each benchmark and tracer combination is less than .0005 for UnTracer compared (pair-wise) to AFL-QEMU and AFL-Dyninst. Given that these p-values are much smaller than the 0.05 significance level, we conclude there exists a statistically significant difference in the median execution times of UnTracer versus AFL-QEMU and AFL-Dyninst.

Vargha and Delaney $\hat{A}_{12}$ scoring: To determine the *extent* to which UnTracer's execution time outperforms AFL-QEMU's and AFL-Dyninst's, we apply Vargha and Delaney's $\hat{A}_{12}$ statistical test [60]. For all comparisons among benchmark trials the resulting $\hat{A}_{12}$ statistic is 1.0—exceeding the conventionally large effect size of 0.71. Thus we conclude that the difference in execution times between UnTracer versus either black-box tracer is statistically large.

2) White-box binary tracing: In Figure 8, we show the benchmark overheads of UnTracer, and AFL's white-box binary (static assembly-time instrumented) tracer AFL-Clang. AFL-Clang averages a relative execution time of 1.36 (36% overhead) across all eight benchmarks, while UnTracer averages 1.003 (0.3% overhead) (average RSD for each tracer was less than 4%). As is the case for black-box binary tracers AFL-QEMU and AFL-Dyninst, in one benchmark with a large number of timeouts—`sfconvert`—AFL-Clang's performance is closest to baseline (nearly matching UnTracer's).

Mann Whitney U-test scoring: On average per dataset, the resulting p-values ranged from .00047 to .015—though only in one instance did the p-value exceed .0005. Thus we conclude that there is a statistically significant difference in median execution times of UnTracer versus AFL-Clang.

Vargha and Delaney $\hat{A}_{12}$ scoring: Among all trials the resulting $\hat{A}_{12}$ statistics range from 0.76 to 1.0. As the minimum of this range exceeds 0.71, we conclude UnTracer's execution time convincingly outperforms AFL-Clang's.

Figure 9 shows the distributions of overheads for each tracer on one dataset of the `cjson` benchmark. The coverage-increasing test cases (red dots) are clearly separable from the non-coverage-increasing test cases for UnTracer, with the coverage-increasing test cases incurring double the overhead of tracing with AFL-Dyninst alone.

Figure 10 shows how UnTracer's overhead evolves over time and coverage-increasing test cases. Very early in the fuzzing process, the rate of coverage-increasing test cases is high enough to degrade UnTracer's performance. As time progresses, the impact of a single coverage-increasing test case is inconsequential and UnTracer gradually approaches 0% overhead. In fact, by 1000 test cases, UnTracer has 90% of the native binary's performance. This result also shows that there is an opportunity for a hybrid coverage-guided tracing model, where initial test cases are always traced until the rate of coverage-increasing test cases diminishes to the point where UnTracer becomes beneficial.

F. Dissecting UnTracer's Overhead

While Untracer achieves significantly lower overhead compared to conventional coverage-agnostic tracers (i.e., AFL-QEMU, AFL-Dyninst, AFL-Clang), it remains unclear which operations are the most performance-taxing. As shown in Algorithm 1, UnTracer's high-level workflow comprises the following: (1) starting the interest oracle and tracer binary forkservers; (2) identifying coverage-increasing test cases by executing them on the oracle; (3) tracing coverage-increasing

[6]We ignore `sfconvert` in all statistical evaluations as its high number of timeouts results in all tracers having similar overhead.

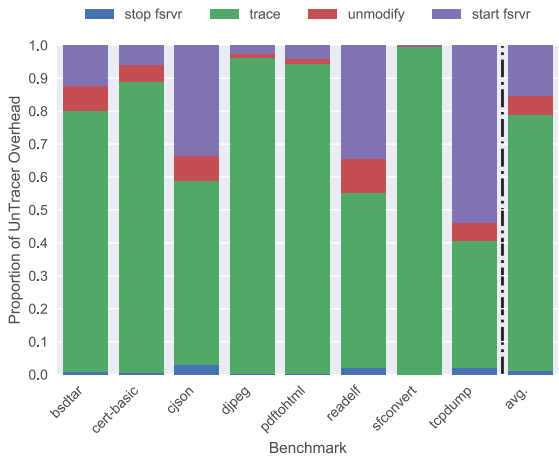

Fig. 11. Visualization of the overheads per UnTracer's four components related to coverage-increasing test case processing for each benchmark.

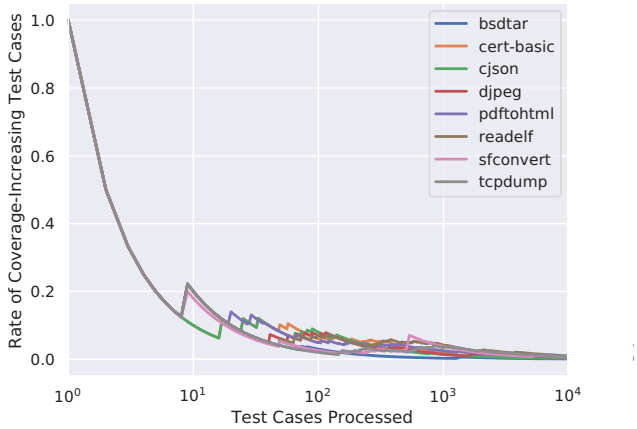

Fig. 12. The rates of coverage-increasing test cases encountered over the total number of test cases processed, per benchmark.

test cases' code coverage by executing them on the tracer; (4) stopping the oracle's forkserver; (5) unmodifying (removing interrupts from) basic blocks in the oracle; and (6) restarting the oracle's forkserver. Since UnTracer identifies coverage-increasing test cases as those which trigger the oracle's interrupt, non-coverage-increasing test cases—the overwhelming majority—exit the oracle cleanly without triggering any interrupts. Thus, executing non-coverage-increasing test cases on the oracle is equivalent to executing them on the original (baseline) binary. Based on this, UnTracer's only overhead is due to processing coverage-increasing test cases.

In our evaluation of UnTracer's overhead, we add timing code around each component run for every coverage-increasing test case: coverage tracing with the tracer (`trace`), stopping the oracle's forkserver (`stop fsrvr`), unmodifying the oracle (`unmodify`), and restarting the oracle (`start fsrvr`). We average all components' measured execution times across all coverage-increasing test cases, and calculate their respective proportions of UnTracer's total overhead. Figure 11 shows the breakdown of all four components' execution time relative to total overhead. The graph shows that the two largest components of UnTracer's overhead are coverage tracing and forkserver restarting.

Tracing: Unsurprisingly, coverage tracing (`trace`) contributes to the almost 80% of UnTracer's overhead across all benchmarks. Our implementation relies on Dyninst-based static binary rewriting-instrumented black-box binary tracing. As our evaluation results (Figure 7) show, in most cases, Dyninst adds a significant amount of overhead. Given UnTracer's compatibility with other binary tracers, there is an opportunity to take advantage of faster tracing (e.g., AFL-Clang in a white-box binary tracing scenario) to lower UnTracer's total overhead.

Forkserver restarting: Restarting the oracle's forkserver (`start fsrvr`) is the component with second-highest overhead. In binaries with shorter test case execution times (e.g., `cjson`, `readelf`, and `tcpdump`), the proportion of tracing time decreases, causing more overhead to be spent on forkserver restarting. Additionally, in comparison to UnTracer's constant-time forkserver-*stopping* operation (`stop fsrvr`), forkserver-restarting relies on costly process creation (e.g., `fork()`, `execve()`) and inter-process communication (e.g., `pipe()`, `read()`, `write()`). Previous work looks at optimizing these system calls for fuzzing [61], but given UnTracer's low overhead in our evaluation, further optimization adds little performance improvement. However, we can imagine niche contexts where such approaches would yield meaningful performance improvements.

G. Overhead versus Rate of Coverage-increasing test cases

Below, we discuss the potential performance advantage of a hybrid approach combining coverage-guided and coverage-agnostic tracing (e.g., AFL [5], libFuzzer [6], honggFuzz [4]). In contrast to existing fuzzing tracers, which face high overhead due to tracing *all* generated test cases, UnTracer achieves near-zero overhead by tracing only *coverage-increasing* test cases—the rate of which decreases over time for all benchmarks (Figure 12). Compared to AFL, UnTracer's coverage tracing is slower on average—largely due to its trace reading/writing relying on slow file input/output operations. Thus, as is the case in our evaluations (Table III), coverage-guided tracing offers significant performance gains when *few* generated test cases are coverage-increasing. For scenarios where a higher percentage of test cases are coverage-increasing (e.g., fuzzers with "smarter" test case generation [7], [39], [9]), our approach may yield less benefit.

In such cases, overhead may be minimized using a *hybrid* fuzzing approach that switches between coverage-guided and coverage-agnostic tracing, based on the observed rate of coverage-increasing test cases. We first identify a *crossover threshold*—the rate of coverage-increasing test cases at which coverage-guided tracing's overhead exceeds coverage-agnostic tracing's. During fuzzing, if the rate of coverage-increasing test cases drops below the threshold, coverage-guided tracing becomes the optimal tracing approach; its only overhead is from tracing the few coverage-increasing test cases. Conversely, if the rate of coverage-increasing test cases exceeds the threshold, coverage-agnostic tracing (e.g., AFL-Clang, AFL-QEMU, AFL-Dyninst) is optimal.

To develop a universally-applicable threshold for all tracing approaches, we average the overheads of coverage-increasing test cases across all trials in our tracer-benchmark evaluations. We then model overhead as a function of the rate of coverage-

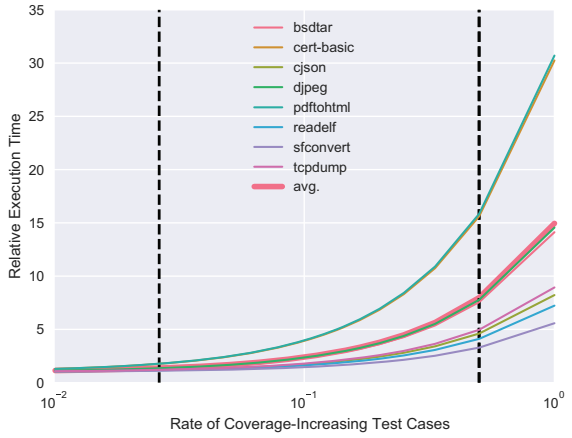

Fig. 13. Model of the relationship between coverage-increasing test case rate and UnTracer's overhead per test case. For all rates left of the leftmost dashed vertical line, UnTracer's overhead per test case is less than AFL-Clang's. Likewise, for all rates left of the rightmost dashed vertical line, it is less than AFL-QEMU's and AFL-Dyninst's. Not shown is the average rate of coverage-increasing test cases observed during our evaluations (4.92E-3).

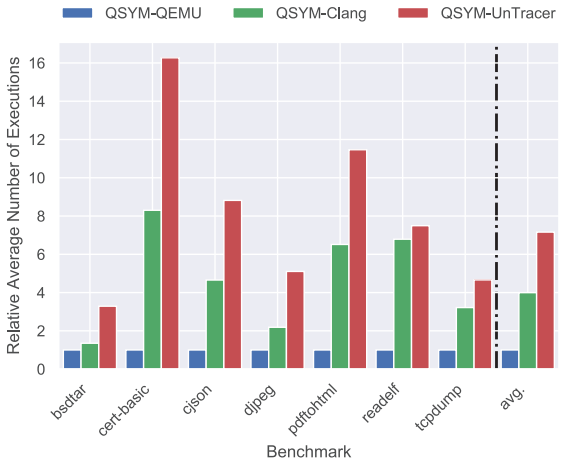

Fig. 14. Per-benchmark relative average executions in 24 hours of QSYM-UnTracer versus QSYM-QEMU and QSYM-Clang.

increasing test cases; we apply this model to identify the coverage-increasing test case rates where UnTracer's overhead exceeds AFL-Clang's, and AFL-QEMU's and AFL-Dyninst's. As shown in Figure 13, for all rates of coverage-increasing test cases below 2% (the leftmost dashed vertical line), UnTracer's overhead per test case is less than AFL-Clang's. Similarly, UnTracer's overhead per test case is less than AFL-QEMU's and AFL-Dyninst's for all rates less than 50% (the rightmost vertical dashed line).

VII. Hybrid Fuzzing Evaluation

State-of-the-art hybrid fuzzers (e.g., Driller [18] and QSYM [19]) combine program-directed mutation (e.g., via concolic execution) with traditional blind mutation (e.g., AFL [5]). Hybrid approaches offer significant gains in code coverage at the cost of reduced test case execution rate. In this section, we compare UnTracer, Clang [5] (white-box tracing), and QEMU [5] (black-box dynamically-instrumented tracing) implementations of the state-of-the-art hybrid fuzzer QSYM on seven of our eight benchmarks.[7] Exploring the benefit of UnTracer in a hybrid fuzzing scenario is important as hybrid fuzzers make a fundamental choice to spend less time executing test cases (hence tracing) and more time on mutation. While we provide an estimate of the impact hybrid fuzzing has on coverage-guided tracing's value in Section III, this section provides concrete data on the impact to UnTracer of a recent hybrid fuzzer.

1) Implementing QSYM-UnTracer: We implemented [28] QSYM-UnTracer in QSYM's core AFL-based fuzzer, which tracks coverage (invoked by `run_target()`) in several contexts: test case trimming (`trim_case()`), test case calibration (`calibrate_case()`), test case saving (`save_if_interesting()`), hybrid fuzzing syncing (`sync_fuzzers()`), and the "common" context used for most test cases (`common_fuzz_stuff()`). Below we briefly discuss design choices specific to each.

[7]We exclude `sfconvert` from this evaluation since the QEMU-based variant of QSYM crashes on all eight experimental trials.

Trimming and calibration: test case trimming and calibration must be able to identify changes in a priori coverage. Thus the interest oracle is unsuitable since it only identifies *new* coverage, and we instead utilize only the tracer binary.

Saving timeouts: A sub-procedure of test case saving involves identifying unique timeout-producing and unique hang-producing test cases by tracing and comparing their coverage to a global timeout coverage. Since AFL only tracks this information for reporting purposes (i.e., timeouts and hangs are not queued), and using an interest oracle or tracer would ultimately add unwanted overhead for binaries with many timeouts (e.g., `djpeg` (Table III)), we configure UnTracer-AFL, AFL-Clang, and AFL-QEMU to only track *total* timeouts.

For all other coverage contexts we implement the UnTracer interest oracle and tracer execution model as described in Section V.

A. Evaluation Overview

To identify the performance impact from using UnTracer in hybrid fuzzing we incorporate it in the state-of-the-art hybrid fuzzer QSYM and evaluate its against existing Clang- [5] and QEMU-based [5] QSYM implementations. Our experiments compare the number of test cases executed for all three hybrid fuzzer variants for seven of the eight benchmarks from Section VI (Table III) with 100ms timeouts. To account for randomness, we average the number of test cases executed from 8, 24-hour trials for each variant/benchmark combination. To form an average result for each variant across all benchmarks, we compute a per-variant geometric mean.

We distribute all trials across eight virtual machines among four workstations. Each host is a six-core Intel Core i7-7800X CPU @ 3.50GHz with 64GB of RAM that runs two, two-CPU 6GB virtual machines. All eight virtual machines run Ubuntu 16.04 x86_64 (as opposed to 18.04 for previous experiments due to QSYM requirements). Figure 14 presents the results for each benchmark and the geometric mean across all benchmarks scaled to our baseline of the number of test cases executed by QSYM-QEMU.

B. Performance of UnTracer-based Hybrid Fuzzing

As shown in Figure 14, on average, QSYM-UnTracer achieves 616% and 79% more test case executions than

QSYM-QEMU and QSYM-Clang, respectively. A potential problem we considered was the overhead resulting from excessive test case trimming and calibration. Since our implementation of QSYM-UnTracer defaults to the slow tracer binary for test case trimming and calibration, an initial problem we considered was the potential overhead resulting from either operation. However, our results show that the performance advantage of interest oracle-based execution (i.e., the "common case") far outweighs the performance deficit from trimming and calibration tracing.

VIII. Discussion

Here we consider several topics related to our evaluation and implementation. First, we discuss the emergence of hardware-assisted coverage tracing, offering a literature-based estimation of its performance with and without coverage-guided tracing. Second, we detail the modifications required to add basic block edge coverage support to UnTracer and the likely performance impact of moving to edge-based coverage. Lastly, we highlight the engineering needed to make UnTracer fully support black-box binaries.

A. UnTracer and Intel Processor Trace

Recent work proposes leveraging hardware support for more efficient coverage tracing. kAFL [11], PTfuzz [12], and honggFuzz [4] adapt Intel Processor Trace (IPT) [35] for black-box binary coverage tracing. IPT saves the control-flow behavior of a program to a reserved portion of memory as it executes. After execution, the log of control-flow information is used in conjunction with an abstract version of the program to generate coverage information. Because monitoring occurs at the hardware-level, it is possible to completely capture a program's dynamic coverage at the basic block, edge, or path level incurring modest run time overheads. The three main limitations of IPT are its requirement of a supporting processor, time-consuming control-flow log decoding, and its compatability with only x86 binaries.

Despite these limitations, it is important to understand how IPT impacts coverage-guided tracing. From a high level, coverage-guided tracing works with IPT because it is orthogonal to the tracing mechanism. Thus, an IPT variant of UnTracer would approach 0% overhead sooner than our Dyninst-based implementation due to IPT's much lower tracing overhead. From a lower level, the question arises as to the value of coverage-guided tracing with relatively cheap black-box binary coverage tracing. To estimate IPT's overhead in the context of our evaluation, we look to previous work. Zhang et al. [12] present a fuzzing-oriented analysis of IPT that shows it averaging around 7% overhead relative to AFL-Clang-fast. Although we cannot use this overhead result directly as we compile all benchmarks with AFL-Clang, according to AFL's author, AFL-Clang is 10–100% slower than AFL-Clang-fast [5]. By applying these overheads to the average overhead of 36% of AFL-Clang from our evaluation, AFL-Clang-fast's projected overhead is between 18–32% and IPT's projected overhead is between 19–35%.

B. Incorporating Edge Coverage Tracking

As discussed in Section II-B, two coverage metrics dominate the fuzzing literature: basic blocks and basic block edges. UnTracer, our implementation of coverage-guided tracing, uses basic block coverage. Alternatively, many popular fuzzers (e.g., AFL [5], libFuzzer [6], honggFuzz [4]) use edge coverage. While the trade-offs between basic block and edge coverage metrics have yet to be studied with respect to fuzzing outcomes, we believe that it is important to consider coverage-guided tracing's applicability to edge coverage metrics.

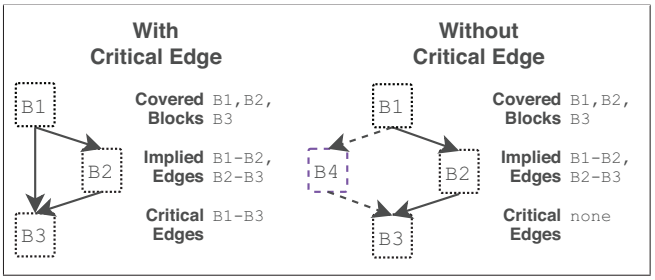

Fig. 15. An example of the *critical edge problem* (left) and its solution (right). To remove the critical edge `B1-B3`, an empty "dummy" block (`B4`) is inserted to introduce two new edges, `B1-B4` and `B4-B3`. Such approach is widely used by software compilers to optimize flow analyses [62].

The first point to understand is that most fuzzers that use edge coverage metrics actually rely on basic block-level tracing [63]. Key to enabling accurate edge coverage while only tracing basic blocks is the removal of critical edges. A critical edge is an edge in the control-flow graph whose starting/ending basic blocks have multiple outgoing/incoming edges, respectively [62]. Critical edges make it impossible to identify which edges are covered from knowing only the basic blocks seen during execution. This inflates coverage and causes the fuzzer to erroneously discard coverage-increasing inputs.

The solution to the critical edge problem is to split each by inserting an intermediate basic block, as shown in Figure 15. The inserted "dummy" basic block consists of a direct control-flow transfer to the original destination basic block. For white-box binaries, edge-tracking fuzzers honggFuzz [4] and libFuzzer [6] fix critical edges during compilation [63]. This approach works for white-box use cases of coverage-guided tracing as well. Unfortunately, how to adapt this approach to black-box binaries is an open technical challenge.

With respect to performance, the impact of moving from basic block coverage to edge coverage is less clear. It is clear that, given that edge coverage is a super-set of basic block coverage, the rate of coverage-increasing test cases will increase. To determine if the increase in the rate of coverage-increasing test cases is significant enough to disrupt the asymmetry that gives coverage-guided tracing its performance advantage, we reference the results in Figure 13 and Table II. Given that seven out of eight of our benchmarks have rates of coverage-increasing test cases below 1 in 100,000 and Figure 13 shows that UnTracer provides benefit for rates below 1 in 50, moving to edge-based coverage needs to induce a 4-orders-of-magnitude increase in the rate of coverage-increasing test cases to undermine UnTracer's value. Such an increase is unlikely given Table II, which shows that even for fuzzers using edge coverage, the rate of coverage-increasing test cases is in line with the rates in our evaluation. Thus, given UnTracer's near-0% overhead, we expect that any increase in the rate of coverage-increasing test cases due to moving to edge coverage will *not* change the high-level result of this paper.

C. Comprehensive Black-Box Binary Support

Niche fuzzing efforts desire support for black-box (source-unavailable) binary coverage tracing. Currently, UnTracer relies on a mix of black- and white-box binary instrumentation

for constructing its two versions of the target binary. For tracer binaries, we use Dyninst-based black-box binary rewriting [25] to insert the forkserver and tracing infrastructure; for oracles, we re-purpose AFL's assembler front-end (`afl-as`) [5] to insert the forkserver. As discussed in Section V-B, our initial implementation used Dyninst to instrument the oracle binary, but we had to switch at `afl-as` due to unresolved performance issues. Though instrumenting the oracle's forkserver at assembly-time requires assembly code access, we expect that inserting the forkserver is not a technical challenge for modern black-box binary rewriters [64], [65], [66], [67] or through function hooking (e.g., via `LD_PRELOAD` [68]).

IX. Related Work

Two research areas orthogonal, but, closely related to coverage-guided tracing are improving test case generation, because improvements here increase the rate of coverage-increasing test cases and system optimizations, because they share the net outcome of improving overall fuzzer performance. We overview recent work in each area and relate those results back to coverage-guided tracing.

A. Improving Test Case Generation

Coverage-guided grey-box fuzzers like AFL [5] and libFuzzer [6] generally employ "blind" test case generation—relying on random mutation, prioritizing coverage-increasing test cases. A drawback of this strategy is stalled coverage, e.g., when mutation fails to produce test cases matching a target binary's *magic bytes* (multi-byte strings or numbers) comparison operations. Research approaches this problem from several directions: Driller [18] and QSYM [19] use concolic execution (i.e., a mix of concrete and symbolic execution) to attempt to solve magic byte comparisons via symbolic path constraints. As is common with symbolic execution, exponential path growth becomes a limiting factor as target binary complexity increases. honggFuzz [4] and VUzzer [7] both leverage static and dynamic analysis to identify locations and values of magic bytes in target binaries. Steelix [9] improves coverage by inferring magic bytes from lighter-weight static analysis and static instrumentation. Angora [39] incorporates byte-level taint tracking, outperforming Steelix's coverage on the synthetic LAVA datasets [69]. However, despite seeing higher rates of coverage-increasing test cases, these fuzzers still face the overhead of tracing all generated test cases.

Instead of attempting to focus mutation on match magic byte comparisons all at once, an alternative set of approaches uses program transformation to make matching more tractable. AFL-lafIntel [70] unrolls magic bytes into single comparisons at compile-time, but currently only supports white-box binaries. MutaGen [71] utilizes mutated "input-producing" code from the target binary for test case generation, but it relies on input-producing code availability, and faces slow execution speed due to dynamic instrumentation. T-Fuzz [47] attempts to strip target binaries of coverage-stalling code, but suffers "transformational explosion" on complex binaries.

Changes in test case mutation schemes have also offered potential workarounds to stalled coverage. FidgetyAFL [58], AFLFast [8], and VUzzer all prioritize mutating test cases exercising rare basic blocks. Ultimately, coverage-guided fuzzers identify coverage-increasing test cases by tracing the coverage of *all* test cases. While such approaches decrease the number of test cases required to create a coverage-increasing test case, their rates of discarded test cases mean that coverage-guided tracing represents a performance improvement.

B. System Scalability

System scalability represents an additional focus of research on improving fuzzing. AFL's execution monitoring component avoids overhead from repetitive `execve()` calls by instead using a fork-server execution model [50]. Xu et al. [61] further improve AFL and libFuzzer's performance by developing several fuzzer-agnostic operating primitives. Distributed fuzzing has also gained popularity; Google's ClusterFuzz [72] (the backbone of OSS-Fuzz [3]) allocates more resources to fuzzing by parallelizing across thousands of virtual machines. As these efforts aim to improve performance of *all* fuzzers, they serve as complements to other fuzzing optimizations (e.g., coverage-guided tracing).

X. Conclusion

Coverage-guided tracing leverages the fact that coverage-increasing test cases are the overwhelmingly uncommon case in fuzzing by modifying target binaries so that they self-report when a test case produces new coverage. While our results show that the additional steps involved in coverage-guided tracing (namely, running the modified binary, tracing, and unmodifying based on new coverage) are twice as expensive as tracing alone, the ability to execute test cases at native speed, combined with the low rate of coverage-increasing test cases, yields overhead reductions of as much as 1300% and 70% for black- and white-box binaries, respectively. Applying coverage-guided tracing in hybrid fuzzing achieves 616% and 79% more test case executions than black- and white-box tracing-based hybrid fuzzing, respectively. Thus, given that tracing consumes over 90% of the total time spent fuzzing—even for fuzzers that focus on test case generation—reductions in tracing time carry over to fuzzing as a whole;

From a higher level, our results highlight the potential advantages of identifying and leveraging asymmetries inherent to fuzzing. Fuzzing relies on executing many test cases in the hopes of finding a small subset that are coverage-increasing or crash-producing. Even given recent attempts to reduce the number of discarded test cases, they are still the common case. Another opportunity is that most of the code itself is uninteresting, but must be executed to reach the interesting code. Thus, we envision a future where faster than full-speed execution is possible by finding ways to skip other "uninteresting" but common aspects of fuzzing.

Acknowledgment

We would like to thank our reviewers for helping us improve the paper. We also thank Xiaozhu Meng from the Dyninst project and Insu Yun from the QSYM project for graciously assisting us in utilizing their software in our implementations. Lastly, we thank Michal Zalewski for providing guidance on the inner workings of AFL. This material is based upon work supported by the National Science Foundation under Grant No. 1650540.

References

[1] "CVE Details: The ultimate security vulnerability datasource," Tech. Rep., 2018. [Online]. Available: https://www.cvedetails.com/vulnerabilities-by-types.php

[2] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," Tech. Rep., 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/billions-and-billions-of-constraints-whitebox-fuzz-testing-in-production/

[3] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software," in *USENIX Security Symposium*, ser. USENIX, 2017.

[4] R. Swiecki, "honggfuzz," 2018. [Online]. Available: http://honggfuzz.com/

[5] M. Zalewski, "American fuzzy lop," 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[6] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *IEEE Cybersecurity Development Conference*, ser. SecDev, 2016, pp. 157–157.

[7] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Network and Distributed System Security Symposium*, ser. NDSS, 2017.

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing As Markov Chain," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2016, pp. 1032–1043.

[9] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state Based Binary Fuzzing," in *ACM Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, 2017, pp. 627–637.

[10] talos vulndev, "AFL-Dyninst," 2018. [Online]. Available: https://github.com/talos-vulndev/afl-dyninst

[11] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, ser. USENIX, 2017, pp. 167–182.

[12] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided Fuzzing with Processor Trace Feedback," *IEEE Access*, vol. 6, pp. 37 302–37 313, 2018.

[13] M. Security, "Dharma: A generation-based, context-free grammar fuzzer." 2018. [Online]. Available: https://github.com/MozillaSecurity/dharma

[14] J. Johnson, "gramfuzz," 2018. [Online]. Available: https://github.com/d0c-s4vage/gramfuzz

[15] M. Eddington, "Peach fuzzing platform," 2018. [Online]. Available: https://www.peach.tech/products/peach-fuzzer/

[16] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2010, pp. 497–512.

[17] M. Vuagnoux, "Autodafe, an Act of Software Torture," 2006. [Online]. Available: http://autodafe.sourceforge.net/

[18] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Network and Distributed System Security Symposium*, ser. NDSS, 2016, pp. 2–16.

[19] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *USENIX Security Symposium*, ser. USENIX, 2018.

[20] C. Cadar, D. Dunbar, D. R. Engler, and others, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2008, pp. 209–224.

[21] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2012, pp. 380–394.

[22] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," in *ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2011, pp. 265–278.

[23] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[24] J. Hertz and T. Newsham, "ProjectTriforce: AFL/QEMU fuzzing with full-system emulation." 2017. [Online]. Available: https://github.com/nccgroup/TriforceAFL

[25] "Dyninst API," 2018. [Online]. Available: https://dyninst.org/dyninst

[26] S. Nagy and M. Hicks, "FoRTE-FuzzBench: FoRTE-Research's fuzzing benchmarks," 2019. [Online]. Available: https://github.com/FoRTE-Research/FoRTE-FuzzBench

[27] ——, "afl-fid: A suite of AFL modifications for fixed input dataset experiments," 2019. [Online]. Available: https://github.com/FoRTE-Research/afl-fid

[28] ——, "UnTracer-AFL: An AFL implementation with UnTracer (our coverage-guided tracer)," 2019. [Online]. Available: https://github.com/FoRTE-Research/UnTracer-AFL

[29] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2008, pp. 206–215.

[30] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[31] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2017, pp. 2329–2344.

[32] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *International Conference on Software Engineering*, ser. ICSE, 2009, pp. 474–484.

[33] P. Godefroid, M. Y. Levin, D. A. Molnar, and others, "Automated whitebox fuzz testing." in *Network and Distributed System Security Symposium*, ser. NDSS, 2008, pp. 151–166.

[34] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2018, pp. 660–677.

[35] Intel, "Intel Processor Trace Tools," 2017. [Online]. Available: https://software.intel.com/en-us/node/721535

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2005, pp. 190–200.

[37] A. Nikolic, "Guided Fuzzing And Binary Blobs," Information Security Symposium (Fsec), 2016. [Online]. Available: https://www.youtube.com/watch?v=zQb-QT7tiFQ

[38] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[39] P. Chen and H. Chen, "Angora: efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2018.

[40] Shellphish, "ShellPhuzz," 2018. [Online]. Available: https://github.com/shellphish/fuzzer

[41] "DARPA Cyber Grand Challenge," 2018. [Online]. Available: https://github.com/cybergrandchallenge

[42] Y. Shoshitaishvili, "CGC Binaries: Compiled CGC binaries for experimentation porpoises." 2017. [Online]. Available: https://github.com/zardus/cgc-bins

[43] J. Kinder, F. Zuleger, and H. Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI, 2009, pp. 214–228.

[44] H. Theiling, "Extracting safe and precise control flow from binaries," in *IEEE International Conference on Real-Time Systems and Applications*, ser. RCTSA, 2000, pp. 23–30.

[45] D. Kästner and S. Wilhelm, "Generic control flow reconstruction from assembly code," in *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES, 2002, pp. 46–55.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2016.

[47] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2018.

[48] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2018, p. 12.

[49] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-Driven Seed Generation for Fuzzing," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2017.

[50] M. Zalewski, "Fuzzing random programs without execve()," 2014. [Online]. Available: http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

[51] R. Stallman, R. Pesch, S. Shebs, and others, "Debugging with GDB," *Free Software Foundation*, vol. 675, 1988.

[52] A. Brown and G. Wilson, "The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks," vol. 1, 2012.

[53] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu, "Kernel probes (kprobes)," *Documentation provided with the Linux kernel sources (v2. 6.29)*, 2016.

[54] M. Hiramatsu and S. Oshima, "Djprobe–Kernel probing with the smallest overhead," in *Linux Symposium*, ser. Linux Symposium, 2007, p. 189.

[55] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. Stillwell, and others, "SCONE: Secure Linux Containers with Intel SGX." in *USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2016, pp. 689–703.

[56] M. Rash, "afl-cve: A collection of vulnerabilities discovered by the AFL fuzzer (afl-fuzz)," 2017. [Online]. Available: https://github.com/mrash/afl-cve

[57] Google, "fuzzer-test-suite: Set of tests for fuzzing engines," 2018. [Online]. Available: https://github.com/google/fuzzer-test-suite

[58] M. Zalewski, "afl-users > Re: "FidgetyAFL" implemented in 2.31b," 2016. [Online]. Available: goo.gl/zmcvZf

[59] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2018.

[60] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[61] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2017.

[62] S. S. Muchnick, *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[63] "SanitizerCoverage: Clang 7 documentation," 2018. [Online]. Available: https://clang.llvm.org/docs/SanitizerCoverage.html

[64] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson, "Zipr: Efficient Static Binary Rewriting for Security," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN, 2017.

[65] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Network and Distributed System Security Symposium*, ser. NDSS, 2017, pp. 2–15.

[66] S. Wang, P. Wang, and D. Wu, "Reassembleable Disassembling," in *USENIX Security Symposium*, ser. USENIX Sec, 2015, pp. 627–642. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai

[67] A. R. Bernat and B. P. Miller, "Anywhere, Any-time Binary Instrumentation," in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, ser. PASTE, 2011, pp. 9–16.

[68] J. Lopez, L. Babun, H. Aksu, and A. S. Uluagac, "A Survey on Function and System Call Hooking Approaches," *Journal of Hardware and Systems Security*, vol. 1, no. 2, pp. 114–136, 2017.

[69] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2016, pp. 110–121.

[70] "laf-intel: Circumventing Fuzzing Roadblocks with Compiler Transformations," 2016. [Online]. Available: https://lafintel.wordpress.com/

[71] U. Kargén and N. Shahmehri, "Turning Programs Against Each Other: High Coverage Fuzz-testing Using Binary-code Mutation and Dynamic Slicing," in *ACM Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, 2015, pp. 782–792.

[72] Google, "ClusterFuzz," 2018. [Online]. Available: https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md