Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing

Stefan Nagy
Virginia Tech
Blacksburg, Virginia
snagy2@vt.edu

Anh Nguyen-Tuong
University of Virginia
Charlottesville, Virginia
nguyen@virginia.edu

Jason D. Hiser
University of Virginia
Charlottesville, Virginia
hiser@virginia.edu

Jack W. Davidson
University of Virginia
Charlottesville, Virginia
jwd@virginia.edu

Matthew Hicks
Virginia Tech
Blacksburg, Virginia
mdhicks2@vt.edu

ABSTRACT

Coverage-guided fuzzing's aggressive, high-volume testing has helped reveal tens of thousands of software security flaws. While executing billions of test cases mandates fast code coverage tracing, the nature of *binary-only* targets leads to reduced tracing performance. A recent advancement in binary fuzzing performance is *Coverage-guided Tracing* (CGT), which brings orders-of-magnitude gains in throughput by restricting the expense of coverage tracing to only when new coverage is guaranteed. Unfortunately, CGT suits only a basic block coverage granularity—yet most fuzzers require finer-grain coverage metrics: *edge coverage* and *hit counts*. It is this limitation which prohibits nearly all of today's state-of-the-art fuzzers from attaining the performance benefits of CGT.

This paper tackles the challenges of adapting CGT to fuzzing's most ubiquitous coverage metrics. We introduce and implement a suite of enhancements that expand CGT's introspection to fuzzing's most common code coverage metrics, while maintaining its orders-of-magnitude speedup over conventional always-on coverage tracing. We evaluate their trade-offs with respect to fuzzing performance and effectiveness across **12** diverse real-world binaries (**8** open- and **4** closed-source). On average, our *coverage-preserving* CGT attains **near-identical** speed to the present *block-coverage-only* CGT, UnTracer; and outperforms leading binary- *and* source-level coverage tracers QEMU, Dyninst, RetroWrite, and AFL-Clang by **2–24**×, finding more bugs in less time.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Fuzzing, Binaries, Code Coverage

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484787

ACM Reference Format: Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages.
https://doi.org/10.1145/3460120.3484787

1 INTRODUCTION

Coverage-guided fuzzing has become one of the most popular and successful techniques for software security auditing. Its aggressive, high-volume testing strategy has revealed countless security vulnerabilities in software, and helped proactively secure many of the world's most popular codebases [8]. Today, software projects of all sizes rely on fuzzing to root out bugs and vulnerabilities both throughout and beyond the software development cycle.

Fuzzing consists of three main steps: (1) *test case generation*, (2) *code coverage tracing*, and (3) *test case triage*. Many works improve fuzzing at the generation level by incorporating input grammars [21], path prioritization [34], better mutators [36], or constraint solving [3]; while others focus on refining triage with sanitizers [15] or other heuristics. However, given fuzzing's core goal of producing—and eventually executing—a large volume of test cases, maintaining high-performance test case execution is critical to effective fuzzing. Recent work shows both "dumb" and "smart" fuzzers spend the majority of their time executing test cases and collecting their coverage traces [37]. However, in binary-only fuzzing contexts, the semantically-poor and opaque nature of a binary prevents the tight integration of coverage-tracing routines that is possible in source-available contexts. This inflates the tracing overhead by up to two orders of magnitude compared to compiler-based instrumentation of source code. Even in an ideal world where black-box instrumenters approach compiler-level performance, recent work shows that coverage tracing increases test case execution time by roughly 30% [15]. To address this performance gap and the time wasted by needless coverage tracing, many binary-only fuzzing efforts [18, 23, 30, 51] are eschewing conventional always-on coverage tracing for an on-demand tracing strategy known as *Coverage-guided Tracing* (CGT) [37]. CGT restricts the expense of tracing to only when new coverage is guaranteed (roughly 1 test case in every 10,000), thereby increasing fuzzing throughput by 500–600% over the leading binary-only tracers.

While some practitioners are leveraging the idea of CGT [18, 23, 30], we are not aware of any fuzzer that has adopted it as a replacement for always-on tracing. Our survey of 27 fuzzers reveals why such performance benefits sit unrealized: CGT only supports *basic block* coverage (instruction sequences ending in control-flow transfer), but most fuzzers rely on finer-grained coverage metrics. Specifically, our study shows **25/27** adopt *edges* (transitions between blocks), and **26/27** further track block or edge *hit counts* (execution frequencies). This lack of support for the most common coverage metrics inhibits CGT's adoption in nearly all fuzzers. While CGT's near-0% tracing overhead is ideal for fuzzing's high-throughput needs, its coverage deficiencies force today's state-of-the-art fuzzers to instead rely on orders-of-magnitude slower, always-on tracing—leaving their full performance potential unrealized.

This paper tackles the challenge of extending CGT to fuzzing's most ubiquitous coverage metrics—edges and hit counts—making high-performance tracing available for all existing (and future) fuzzers. At the core of our efforts are binary-level and fuzzing enhancements that broaden CGT's coverage while maintaining its orders-of-magnitude speedup: for edge coverage, we introduce a zero-overhead strategy called *jump mistargeting* that addresses the most common (statically and dynamically) form of critical edges while keeping control flow intact. To maintain completeness of edge coverage, we back jump mistargeting with a low-overhead binary-only implementation of a control-flow transformation that eliminates critical edges through block insertion called branch splitting (e.g., LLVM's SanitizerCoverage [50]). To extend CGT to hit count coverage, we exploit the observation that execution frequency changes are highly localized to loops, devising a *bucketed unrolling* strategy to encode them with a minimally-invasive hit count tracking mechanism congruent with current fuzzers [18, 59].

We implement our *coverage-preserving* Coverage-guided Tracing, *HeXcite*, and evaluate it against the current block-coverage-only CGT implementation UnTracer [37]; the leading binary-only fuzzing coverage tracers QEMU [59], Dyninst [26], and RetroWrite [15]; and the popular source-level coverage tracing via AFL-Clang [59]. In evaluations across **12** diverse real-world binaries (**8** open- and **4** closed-source), HeXcite attains a throughput near identical to UnTracer's; **3–24**× that of conventional *always-on* binary-only tracers QEMU, Dyninst, and RetroWrite; and **2.8**× that of source-level tracing with AFL-Clang. HeXcite's coverage-preserving transformations further enable it to find **12–749%** more unique bugs than UnTracer as well as always-on binary- *and* source-level tracers in standard coverage-guided grey-box fuzzing integrations—while finding **16** known bugs and vulnerabilities in **32–52%** less time.

Through the following contributions, **this paper enables the use of the fastest tracing approach in fuzzing—*Coverage-guided Tracing*—by the majority of today's fuzzers**:

- We introduce *jump mistargeting*: a control-flow redirection strategy which alters the common-case of edge instructions such that they self-report edge coverage at native speed.
- We introduce *bucketed unrolling*: a technique which clones loop conditions at discrete intervals, enabling the self-reporting of loop hit-count coverage at near-native speed.
- We demonstrate that with these techniques, our *coverage-preserving* CGT eclipses block-only CGT—and conventional always-on binary- *and* source-level tracers—in edge coverage, loop coverage, and bug-finding fuzzing effectiveness.
- We show that coverage-preserving CGT's speed is nearly indistinguishable from that of block-only CGT, and—despite being a binary-only technique—is >2× the speed of even source-level tracing approaches.
- We open-source HeXcite, our implementation of binary-only coverage-preserving CGT, and our evaluation benchmarks at: `https://github.com/FoRTE-Research/HeXcite`.

2 BACKGROUND

To understand our improvements to Coverage-guided Tracing, it is crucial to understand the core details of coverage-guided fuzzing, its code coverage metrics, and the high-performance tracing strategy known as Coverage-guided Tracing.

2.1 Software Fuzzing

Software fuzzing broadly represents one of today's most popular software quality assurance approaches. Unlike other forms of software testing that vet functionality (e.g., unit testing, mutational testing), fuzzing's primary focus is security auditing; test cases are generated and fed to the target program with their effects monitored for signs of security violations. Many software vulnerabilities have been (and continue to be) uncovered via fuzzing, and its use among developers large and small continues to grow each year [18].

Fuzzing encompasses a variety of techniques accommodating specific use cases, with the most common distinction being search strategy; *directed* fuzzers constrain testing to specific code or paths (e.g., newly-patched [6] or likely-vulnerable code [11]), while *guided* fuzzers aim to maximize the program's state space along some prespecified metric (e.g., memory accesses or code coverage). By far the most common and successful form of fuzzing is *coverage-guided fuzzing* [59] which, as the name implies, aims to maximize test cases' code coverage to uncover hidden program bugs.

2.2 Coverage-guided Fuzzing

Coverage-guided fuzzing's scalability, easy adoption, and time-tested effectiveness have made it widely popular among both developers and security practitioners. As shown in Figure 1, given a target program, a typical coverage-guided fuzzing workflow consists of the following recurring steps:

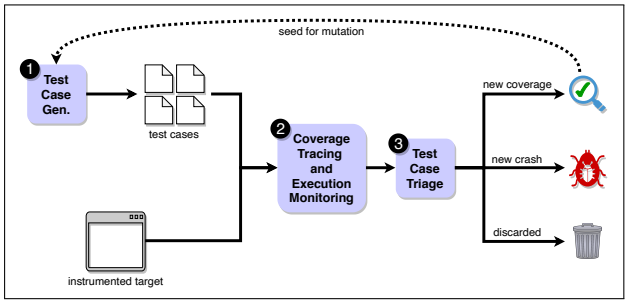


Figure 1: The high-level steps of coverage-guided fuzzing.

(1) ***Generation***. Genetic algorithms (typically a mix of random and deterministic byte mutations) create batches of candidate test cases from one or more ancestors.

(2) ***Coverage Tracing & Execution Monitoring***. Lightweight statically- or dynamically-inserted instrumentation captures each test case's runtime code coverage given some pre-specified coverage metric(s), while monitoring their other execution behavior (e.g., terminating signal).
(3) ***Triage***. Candidates are grouped based on observed execution behavior; those increasing coverage are preserved for future mutation, while those triggering crashes are deduplicated in anticipation of manual bug analysis.

Coverage-guided fuzzing's balance of feedback-guided auditing and aggressive, high-volume testing continues to reign supreme over other automated security testing methodologies; its effectiveness is evidenced by the deep (and ever-growing) vulnerability trophy cases held by prominent fuzzers such as Google's AFL [59], honggFuzz [49], and libFuzzer [45]; and its fundamental principles form the core of today's most state-of-the-art fuzzing efforts.

2.3 Fuzzing's Code Coverage Metrics

To maximally vet the target application, coverage-guided fuzzing collects a test case's dynamic code coverage and subsequently mutates only those which attain new coverage. In our efforts to understand fuzzing's current coverage landscape, we survey 27 of today's state-of-the-art coverage-guided fuzzers (Table 1) and identify three universal coverage metrics: *basic blocks*, *edges*, and *hit counts*. We discuss these coverage metrics in detail below.

Name	Cov	Hit	Name	Cov	Hit	Name	Cov	Hit
AFL [59]	➤	✔	EnFuzz [12]	➤	✔	ProFuzzer [57]	➤	✔
AFL++ [18]	➤	✔	FairFuzz [34]	➤	✔	QSYM [58]	➤	✔
AFLFast [7]	➤	✔	honggFuzz [49]	➤	✗	REDQUEEN [3]	➤	✔
AFLSmart [41]	➤	✔	GRIMORE [5]	➤	✔	SAVIOR [11]	➤	✔
Angora [9]	➤	✔	lafIntel [1]	➤	✔	SLF [56]	➤	✔
CollAFL [19]	➤	✔	libFuzzer [45]	➤	✔	Steelix [35]	➤	✔
DigFuzz [60]	➤	✔	Matryoshka [10]	➤	✔	Superion [53]	➤	✔
Driller [48]	➤	✔	MOpt [36]	➤	✔	TIFF [29]	■	✔
Eclipser [13]	➤	✔	NEUZZ [46]	➤	✔	VUzzer [43]	■	✔

Table 1: A survey of recent coverage-guided fuzzers and their coverage metrics (edges/blocks and hit counts). Key: ➤ (edges), ■ (blocks).

Basic Block Coverage: Basic blocks refer to straight-line (i.e., single entry and exit) instruction sequences beginning and ending in control-flow transfer (i.e., jumps, calls, or returns), and comprise the nodes of a program's control-flow graph. Tracking basic block coverage necessitates instrumenting each to record their execution in some data structure (e.g., an array [44] or bitmap [59]). Two modern fuzzers that employ basic block coverage are VUzzer [43] and its successor TIFF [29].

Edge Coverage: Edges refer to block-to-block transitions, and offer a finer-grained approximation of paths taken. As Table 1 shows, most fuzzers rely on edge coverage; AFL [59] and its many derivatives [18] record edges as hashes of their start/end block tuples in a bitmap data structure; while LLVM SanitizerCoverage-based [50] fuzzers honggFuzz and libFuzzer track edges from the block level by splitting *critical edges* (edges whose start/end blocks have at least two outgoing/incoming edges, respectively).

Hit Count Coverage: Hit counts refer to block/edge execution frequencies, and are commonly tracked to reveal progress in state exploration (e.g., iterating on a loop). libFuzzer, AFL, and AFL derivatives approximate hit counts using 8-bit "buckets", with each bit representing one of eight ranges (0–1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+); test cases are seeded if they jump from one bucket to another for any block/edge. While tracking *exact* hit counts (e.g., VUzzer and TIFF) reveals finer-grained state changes, it risks oversaturating the fuzzer seed pool with needless test cases (e.g., one per new loop iteration), and is hence seldom used.

2.4 Coverage-guided Tracing

Many recent works improve fuzzing with smarter test case generation or triage. But despite these advancements, the maximal performance of both standard and state-of-the-art coverage-guided fuzzers is subject to a key constraint: code coverage is traced for *all* test cases, yet *less than 1 in 10,000* actually increase coverage [37]. While this has little impact in use cases where tracing instrumentation is already fast (i.e., open-source software), it is the principal bottleneck for those where tracing is costly—i.e., closed-source software. For this reason, a number of binary-only fuzzing efforts [18, 23, 30, 51] are instead adopting a lighter-weight strategy called *Coverage-guided Tracing* (CGT), which restricts the expense of tracing to only the $< 0.01\%$ of test cases that increase coverage.

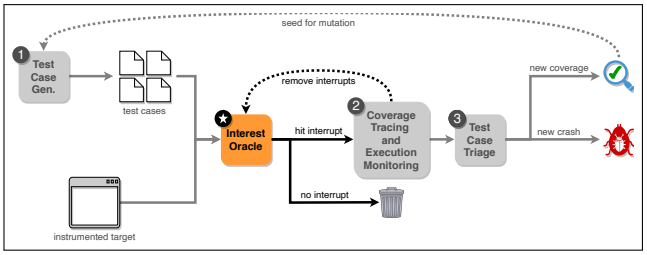


Figure 2: Coverage-guided Tracing's core workflow.

Given a target binary, CGT constructs two versions: a *coverage oracle* with an interrupt (e.g., `0xCC`) inserted at every basic block, and a *tracer* instrumented for conventional fuzzing coverage tracing. As shown in Figure 2, CGT runs each test case first on the oracle; if an interrupt is hit, the test case's full coverage is then captured with the tracer, and all visited blocks' have their corresponding oracle interrupts removed; and if no interrupt was hit, the test case is simply discarded following its run on the oracle. Most test cases ($> 99.9\%$ [37]) revisit already-seen coverage and thus will not trigger interrupts, sparing them of tracing; and because the oracle's mechanism of reporting new coverage is just interrupts (and not instrumentation callbacks) this majority of test cases are run at speed equivalent to the original binary's—giving CGT a near-native runtime overhead of 0.3%, and 500–600% higher test case throughput over the conventional *always-on* tracing used in binary-only fuzzing like AFL-Dyninst [26] and AFL-QEMU [59].

The Code Coverage Dilemma: Though CGT enables orders-of-magnitude higher binary-only fuzzing throughput, it is currently incompatible with **all** of the state-of-the-art coverage-guided fuzzers we surveyed in Table 1: whereas CGT presently supports only a basic block coverage level, **25/27** fuzzers instead rely on edge coverage, and **26/27** further track hit counts. Allowing the broad spectrum of coverage-guided fuzzers to obtain the performance benefits of CGT necessitates an answer to this disparity in code coverage metrics.

3 A COVERAGE-PRESERVING CGT

Coverage-guided Tracing (CGT) accelerates binary-only fuzzing by restricting the expense of code coverage tracing to only the few test cases that reach new coverage. Unfortunately, CGT's lack of support for fuzzing's most common coverage metrics, edges and hit counts, leaves its performance benefits untapped for nearly all of today's state-of-the-art fuzzers.

To address this incompatibility, we observe how CGT achieves lightweight coverage tracking at the control-flow level; and devise two new techniques exploiting this paradigm to facilitate finer-grained coverage—*jump mistargeting* (for edge coverage) and *bucketed unrolling* (for hit counts)—without compromising CGT's minimally-invasive nature. Below we discuss the inner workings of jump mistargeting and bucketed unrolling, and the underlying insights and observations that motivate them.

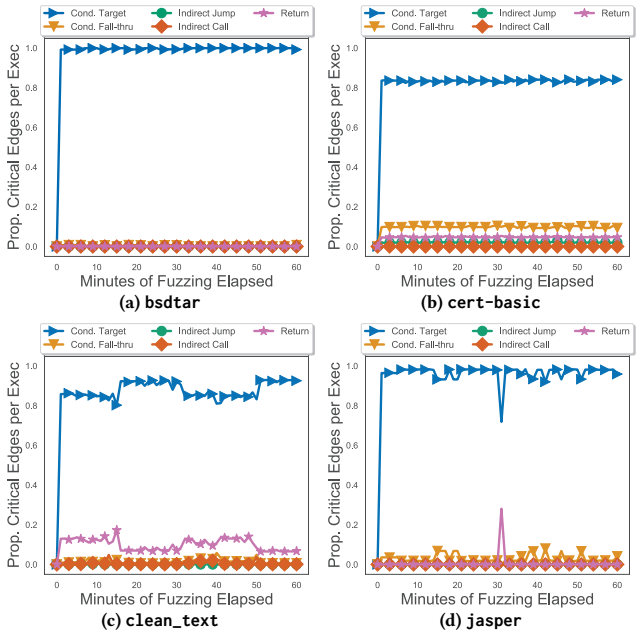


Figure 3: Visualization of the proportion of critical edges by transfer type encountered throughout fuzzing.

3.1 Supporting Edge Coverage

AFL and its derivatives utilize *hash-based* edge coverage, instrumenting each basic block to dynamically record edges as hashes of their start/end blocks. However, as CGT's key speedup comes from replacing per-block instrumentation with far cheaper interrupts, it is thus incompatible with AFL-style hash-based edge coverage.

libFuzzer and honggFuzz track edges using LLVM's SanitizerCoverage instrumentation, which forgoes hashing to instead infer edges from the set of covered blocks. For example, given a control-flow graph with edges $\overrightarrow{ab}$ and $\overrightarrow{bc}$, covering blocks a and b implies covering edge $\overrightarrow{ab}$; and subsequently covering c implies $\overrightarrow{bc}$. However, such *block-centric* edge coverage does not suffice if there exists a third edge $\overrightarrow{ac}$. In this case, covering blocks a, b, and c implies edges $\overrightarrow{ab}$ and $\overrightarrow{bc}$; but since c has *already* been covered, there is no way to detect $\overrightarrow{ac}$. Formally, these problematic edges are referred to as *critical edges*: edges whose start/end blocks have two or more incoming/outgoing edges, respectively [50].

Program	Total Edges	Crit. Edges	Prop.
bsdtar	42911	9867	0.23
cert-basic	7544	1642	0.22
clean_text	8762	1592	0.18
jasper	21637	5878	0.27
readelf	30959	7301	0.24
sfconvert	8358	2022	0.24
tcpdump	36200	7312	0.20
unrtf	2505	465	0.19
		Mean	**22%**

Table 2: Proportion of critical edges in eight real-world programs.

1. **Conditional target** (e.g., `jle 0x100`'s *True* branch)
2. **Conditional fall-through** (e.g., `jle 0x100`'s *False* branch)
3. **Indirect jump** (e.g., `jmp %eax`)
4. **Indirect call** (e.g., `call %eax`)
5. **Return** (e.g., `ret`)

Table 3: Examples of x86 critical edge instructions by transfer type.

Program	CndTarg	CndFall	IndJmp	IndCall	Ret
bsdtar	1.00	0.00	0.00	0.00	0.00
cert-basic	0.84	0.10	0.02	0.00	0.05
clean_text	0.87	0.02	0.00	0.01	0.10
jasper	0.97	0.03	0.00	0.00	0.00
readelf	0.70	0.03	0.01	0.12	0.14
sfconvert	0.84	0.02	0.00	0.00	0.13
tcpdump	0.98	0.01	0.00	0.00	0.01
unrtf	0.94	0.03	0.00	0.00	0.02
Mean	**89.3%**	**2.9%**	**0.4%**	**1.6%**	**5.7%**

Table 4: Proportion of encountered critical edges by transfer type.

Diving deeper into critical edges: Supporting block-centric edge coverage requires resolving all critical edges. LLVM's SanitizerCoverage achieves this by *splitting* each critical edge with a "dummy" block, creating two new edges. Continuing example § 3.1, dummy d will split critical edge $\overrightarrow{ac}$ into $\overrightarrow{ad}$ and $\overrightarrow{dc}$, thus permitting path $\overrightarrow{adc}$ to be differentiated from $\overrightarrow{abc}$. But while such approach is indeed compatible with CGT's block-centric, interrupt-driven coverage, our analysis of eight real-world binaries shows **over 1 in 5** edges are critical (Table 2), revealing that splitting *every* critical edge with a new block leaves a significant control-flow footprint—and inevitably, a higher baseline binary fuzzing overhead.

To understand the impact of critical edges on fuzzing, we instrument the same eight real-world binaries and dynamically record their instruction traces.[1] In conjunction with the statically-generated control-flow graphs, we analyze each trace to measure the occurrences of critical edges; and further quantify them by *transfer type*, which on the x86 ISA takes on one of five forms (shown in Table 3).[2]

As shown in Figure 3 and Table 4, our findings reveal that *conditional jump target* branches make up an average of **89%** of all dynamically-encountered critical edges.

> **Observation 1:** Conditional jump target branches make up the vast majority of critical edges encountered during fuzzing.

[1] We limit instruction tracing to one hour of fuzzing due to the massive size of the resulting trace data (ranging from 200GB to 7TB per benchmark).

[2] As critical edges are, by definition, one of at least two outgoing edges from their starting block, transfers with at most one destination (direct jumps/calls and unconditional fall-throughs) can *never* be critical edges.

3.1.1 ***Jump Mistargeting.*** Splitting critical edges with dummy blocks adds a significant number of new instructions to each execution, and with it, more runtime overhead—slowing binary-only fuzzing down even further. For the common case of critical edges (conditional jump target branches), we observe that the edge's destination address is encoded within the jump instruction itself, and thus can be statically altered to direct the edge elsewhere. Our approach, *jump mistargeting*, exploits this phenomena to "mistarget" the jump's destination so that it resolves into a CGT-style interrupt—permitting a signaling of the critical edge's coverage *without* any need for a dummy block (i.e., identifying edge $\overrightarrow{ac}$ in § 3.1's example without the additional dummy block `d`).

An overview of jump addressing: The x86 ISA has three types of jumps: *short*, *near* (or long), and *far*. Short and near jumps achieve intra-segment transfer via program counter (PC)-*relative addressing*: short jumps use 8-bit signed displacements, and thus can reach up to +127/-128 bytes relative to the PC; while near jumps use much larger 16–32-bit signed displacements. In contrast, far jumps achieve inter-segment transfer via *absolute addressing* (i.e., to a fixed location irrespective of the PC). All three jumps share the common instruction layout of an *opcode* followed by a 1–4 byte *destination operand* (an encoding of the relative/absolute address). Since the adoption of position-independent layouts, most x86/x86-64 code utilizes relative addressing.

Redirecting jumps to interrupts: Jump mistargeting alters conditional jump target critical edges to trigger interrupts when taken. When used in CGT, its effect is identical to combining interrupts with conventional (yet more invasive) edge splitting—while avoiding the associated cost of inserting new blocks. We envision two possible jump mistargeting strategies (Figure 4): one leveraging *embedded* interrupts, and another with *zero-address* interrupts.

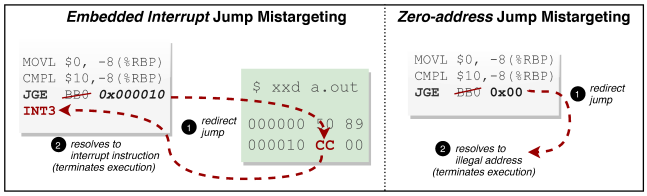


Figure 4: A visualization of jump mistargeting via embedded (left) and zero-address interrupts (right).

(1) ***Embedded Interrupts.*** The simplest mistargeting approach is to replace each jump's destination with a garbage address, ideally resolving to an illegal instruction (thus interrupting the program). However, as many instructions have *one-byte* opcodes, a carelessly-chosen destination may very well initiate an erroneous sequence of instructions.
A more complete strategy is to instead redirect the jump to a location where an interrupt opcode is embedded. For example, the byte sequence `[00 CC]` at address `0x405500` normally resolves to instruction `[add %cl,%ah]`; but as `0xCC` is itself an opcode for interrupt `int3`, it suffices to redirect the target critical edge jump to `0x405501`, which subsequently fetches and executes `0xCC`, thus triggering the interrupt instruction. A key challenge (and bottleneck) of this approach is scanning the bytespace in the jump's displacement range to pinpoint embedded interrupts.

(2) ***Zero-address Interrupts.*** As nearly all x86/x86-64 code is position-independent and hence uses PC-relative addressing, an alternative and less analysis-intensive mistargeting approach is to interrupt the program by resolving the jump's displacement to the zero address (i.e., `0x00`). For example, taking the conditional jump represented by byte sequence `[0F 8F 7C 00 00 00]` at address `0x400400` normally branches to address `0x400400+6+0x0000007C` (i.e., the PC + instruction length + displacement); but to resolve it to the zero address merely requires the displacement be rewritten to `0xFFBFFB7E` (i.e., the negative sum of the PC and instruction length). As 8–16 bit displacements do not provide enough "room" to cover the large virtual address space of modern programs, zero-address mistargeting is generally restricted to jumps with 32-bit displacements, however, most x86-64 branches fit this mold.

> **Technique 1:** Jumps' self-encoded targets can be rewritten to resolve to addresses that result in interrupts, enabling binary-level CGT edge coverage at native speed (i.e., without needing to insert additional basic blocks).

3.2 Supporting Hit Counts

Most fuzzers today adopt AFL-style [59] bucketed hit count coverage, which coarsely tracks changes in block/edge execution frequencies over a set of eight ranges: 0–1, 2, 3, 4–7, 8–15, 16–31, 32–127, and 128+. Unfortunately, CGT's interrupt-driven coverage currently only supports a *binarized* notion of coverage (i.e., taken/not taken), and thus requires a fundamentally new approach to support finer-grained frequencies.

Diving deeper into hit counts: In exploring the importance of hit counts, we observe that most new hit count coverage is localized to *loops* (e.g., `for()`, `while()`). As Rawat and Mounier [43] demonstrate that as many as **42%** of binary code loops induce buffer overflows (e.g., by iterating over user-provided input with `strcpy()`), it is imperative to track hit counts as a means of assessing—and prioritizing—fuzzer "progress" toward higher loop iterations. However, inferring a loop's iteration count is achievable purely from monitoring its induction variable—eliminating the expense of tracking hit counts for *every* loop block (as AFL and libFuzzer do).

> **Observation 2:** Hit counts provide fuzzing a notion of loop exploration progress, but need only be tracked once per loop iteration.

3.2.1 ***Bucketed Unrolling.*** AFL-style [59] hit count tracking adds counters to each block/edge to dynamically update their respective hit counts in a shared memory coverage bitmap. However, this approach is fundamentally incompatible with the binarized nature of CGT's block-centric, interrupt-driven coverage. While a naive solution is to instead add CGT's interrupts following the application of a *loop peeling* transformation—making several copies of the loop's body and stitching them together with direct jumps (e.g., $\texttt{head} \rightarrow \texttt{body}_1 \rightarrow \ldots \rightarrow \texttt{body}_n \rightarrow \texttt{tail}$)—the resulting binary will be exceedingly space inefficient due to excessive code duplication—especially for nested loops.

In search of a more performant solution, we develop *bucketed unrolling*—drawing from compiler loop unrolling principles to encode the functionality of AFL-style bucketed hit counts as a series of binarized range comparisons.

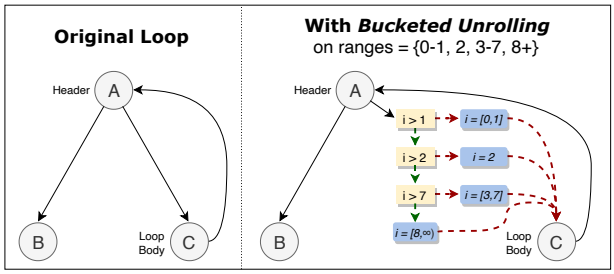


Figure 5: Bucketed unrolling applied to a simple loop.

As shown in Figure 5, bucketed unrolling augments each loop header with a series of sequential conditional statements weighing the loop induction variable against the desired hit count bucket ranges (e.g., AFL's eight). To support CGT, each conditional's fall-through block is assigned an interrupt; taking any conditional's target branch jumps directly to the loop's body, indicating no change from the current bucket range; and taking the fall-through triggers the next sequential interrupt, thus signaling an advancement to the next bucket. The resulting code replicates the functionality of AFL-style hit count tracking—but obtains much higher performance by doing so at just one instrumentation location per loop.

Technique 2: Encoding conventional bucketed hit count tracking as a series of sequential, binarized range checks enables CGT to capture binary-level loop exploration progress—while upholding its fast, interrupt-driven coverage-tracing strategy.

4 IMPLEMENTATION: HEXCITE

In this section we introduce HeXcite—***H**igh-**E**fficiency e**X**panded **C**overage for **I**mproved **T**esting of **E**xecutables*—our implementation of binary-only *coverage-preserving* Coverage-guided Tracing. Below we discuss HeXcite's core architecture, and our design decisions in realizing jump mistargeting and bucketed unrolling.

4.1 Architectural Overview

HeXcite consists of three main components: **(1) binary generation**, **(2) control-flow mapping**, and **(3) the fuzzer**. We implement components 1–2 as a set of analysis and transformation passes atop the ZAFL static rewriting platform [38], and component 3 atop the industry-standard fuzzer AFL [59]. Below we briefly discuss each and their synergy in facilitating coverage-preserving CGT.

Binary Generation: HeXcite's workflow is similar in nature to UnTracer's (Figure 2); i.e., we generate two versions of the original target binary: (1) an *oracle* (run for every test case) with interrupts added to each basic block; and (2) a *tracer* (run only for coverage-increasing test cases) equipped with conventional tracing instrumentation. While many fuzzers embrace compiler instrumentation for its speed and soundness (i.e., LLVM [33]), there are by now a number of static binary rewriters with comparable qualities. We examine several popular and/or emerging security-oriented binary rewriters—Dyninst [40], McSema [14], RetroWrite [15], and ZAFL [38]—and distill a set of properties we feel are best-suited supporting jump mistargeting and bucketed unrolling: (1) *a modifiable control-flow* representation; (2) *dominator flow analysis* [2]); and (3) *sound code transformation and generation*. We select ZAFL as the basis for HeXcite as it is the highest performance rewriter that possesses the above three properties in addition to an LLVM-like transformation API. We expect that with additional engineering effort, our findings apply to the other rewriters listed.

Like most static binary rewriters, ZAFL operates by first disassembling and lifting the input binary to an intermediate representation;[3] and performing all code transformation at this IR level (e.g., injecting bucketed unrolling's range checks § 4.3), adjusting the binary's layout as necessary before reconstituting the final executable. While relocating direct (i.e., absolute and PC-relative) control flow is generally trivial, attempting so for *indirect* transfers is *undecidable* and risks corrupting the resulting binary, as their respective targets cannot be identified with any generalizable accuracy [39, 54]. ZAFL addresses this challenge conservatively via *address pinning* [25, 27], which "pins" any *unmovable* items (including but not limited to: indirectly-called function entries, callee-to-caller return targets, data, or items that cannot be precisely disambiguated as being either code or data) to their *original* addresses;[4] while safely relocating the remaining *movable* items around these pins (often via chained jumps). Though address pinning will likely over-approximate the set of unmovable items at slight cost to binary performance and/or space efficiency (particularly for exceedingly-complex binaries with an abundance of jump tables, handwritten assembly, or data-in-code), its *general-purpose soundness, speed, and scalability* [38] *makes it promising for facilitating coverage-preserving CGT.* Our current prototype, HeXcite, supports binary fuzzing of x86-64 Linux C and C++ executables.

Control-flow Mapping: A key requirement of CGT is a mapping of each oracle basic block's address (i.e., where an interrupt is added) to its corresponding tracer binary trace-block ID; when a coverage-increasing test case is found, the tracer is invoked to capture the test case's full coverage, for which all interrupts are subsequently removed at their addresses in the oracle. To generate this mapping, we save the original and rewritten control-flow graphs for both the oracle and tracer binaries. We then parse the pair of original control-flow graphs to find their corresponding matches, and subsequently map each to their oracle and tracer binary counterparts (i.e., `(cfgBB,oracleBB)` → `(cfgBB,tracerBB)`). From there, we generate the necessary `(oracleAddr,tracerID,interruptBytes)` mapping for each block (e.g., `(0x400400,30,0xCC)`). If mapping should fail (e.g., a tracer block with no corresponding oracle block), we omit the block to avoid problematic interrupts; we observe this generally amounts to no more than a handful of instances per binary, and does not impact HeXcite's overall coverage (§ 5.2.1–§ 5.2.2).

The Fuzzer: Like UnTracer, we implement HeXcite atop the industry-standard fuzzer AFL [59] 2.52b with several changes in

[3] ZAFL's disassembly supports mixing-and-matching of recursive descent and linear sweep. The current tools utilized are based on IDA Pro [24] and GNU objdump [20].
[4] To support address pinning, ZAFL conservatively scans for addresses *likely* targeted by indirect control flow; generally this is achieved via rudimentary heuristics (e.g., post-call instructions, jump table entries, etc.). Additionally, ZAFL pins all data items.

test case handling logic (Figure 6). We default to conventional tracing for any executions where coverage is required (e.g., calibration and trimming), while not re-executing or saving timeout-producing test cases. As jump mistargeting triggers signals that might otherwise appear as valid crashes (e.g., `SIGSEGV`), we alter HeXcite's fuzzer-side crash-handling logic as follows: if a test case crashes the oracle, we re-run it on the tracer to verify whether it is a *true* or a *mistargeted* crash; if it does not crash the tracer, we conclude it is the result of taking a mistargeted critical edge (i.e., a `SIGSEGV` from jumping to the zero address), and save it to the fuzzer queue. We note that the core principles of coverage-preserving CGT scale to any fuzzer (e.g., honggFuzz), as evidenced by emerging CGT-based efforts within the fuzzing community [18, 23, 30].

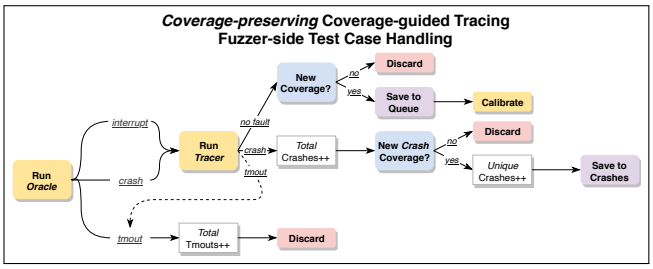


Figure 6: HeXcite's fuzzer-side test case handling logic. Like UnTracer, we discard timeout-producing test cases; however, we re-run crashing test cases to determine whether they are a *true* crash (i.e., occurring on both the oracle and tracer) or the result of hitting an oracle *mistargeted* edge (generally triggering a `SIGSEGV` from the jump being redirected to the zero address).

4.2 Implementing Jump Mistargeting

We implement *zero-address* jump mistargeting for the common-case of critical edges, conditional jump target branches (§ 3.1), as follows. To statically identify critical edges we first enumerate all control-flow edges, and mark an edge as *critical* if at least two edges both precede and succeed it. We subsequently parse each critical edge and categorize it by type by examining its starting block's last instruction (Table 3). Lastly, we update an offline record of each critical edge by type (e.g., "conditional jump target") and its respective starting/ending basic block addresses.

We enumerate all conditional jump target critical edges; as x86-64 conditional jumps are 6-bytes in length and encoded with a 32-bit PC-relative displacement, we compute the sum of the instruction's address and its length, and determine the 2's complement (i.e., negative binary representation). Using basic file I/O we then statically overwrite the jump's displacement operand with the little-endian encoding of the zero-address-mistargeted displacement, and update our oracle-to-tracer mapping accordingly (e.g., `(0x400400,30,0x7C000000)` for the example in § 3.1).

If a critical edge cannot accommodate zero-address mistargeting (e.g., from having a <32-bit displacement), we attempt to fall-back to conventional SanitizerCoverage-style [50] edge splitting, inserting a dummy block and connecting it to the edge's end block. Conditional *fall-through* critical edges require careful handling, as accommodating the transfer from the edge's starting block to the dummy requires the dummy be placed immediately after the starting block (i.e., the next sequential address). However, splitting *indirect* critical edges remains a universal problem even for robust compilers like LLVM (§ 6.1). While recent work [31] reveals the possibility that indirect edges may be modeled at the binary level, such approaches are still too imprecise to be realistically deployed; hence, we conservatively omit indirect critical edges as we observe they have little overall significance on dynamically-seen control-flow (Figure 3).

4.3 Implementing Bucketed Unrolling

We implement bucketed unrolling to replicate AFL-style loop hit count tracking, beginning with an analysis pass to retrieve all code loops from the target binary based on the classic *dominance-based* loop detection [42]: given the control-flow graph and dominator tree (generally available in any off-the-shelf static rewriter's API), we mark a set of blocks S as a loop if (1) there exists a header block h that dominates all blocks in S; and (2) there exists a *backward* edge $\overrightarrow{bh}$ from some block $b \in S$ such that h dominates b.[5] Though binary-level loop head/body detection is difficult—particularly around complex optimizations like Loop-invariant Code Motion—we observe that the standard dominance-based algorithm is sufficient; and while HeXcite attains the highest loop coverage in our evaluation (§ 5.2.2), we expect that future advances in optimized-binary loop detection will only improve these capabilities.

As pinpointing a loop's induction variable (the target of bucketed unrolling's discrete range checks) is itself semantically challenging at the binary level, we opt for a simpler approach and instead add a "fake" loop counter before each loop header; and augment the header with an instruction to increment this counter per iteration (e.g., x86's `incl`). *Where* the increment is inserted in the header ultimately depends on the static rewriter of choice; Dyninst [40] prefers to conservatively insert new code at basic block entrypoints to avoid clobbering occupied registers; while RetroWrite [15] and ZAFL [38] analyze register liveness to more tightly weave code with the original instructions. Either style is supportive of HeXcite, though tight code insertion is preferable for higher runtime speed.

We implement bucketed unrolling's sequential range checks (per AFL's 8-bucket hit count scheme) as a transformation pass directly before the loop's first body block; and connect each to the first body block via direct jumps, and to each other via fall-throughs. The resulting assembly resembles the following (shown in Intel syntax):

```
1  _loop_head:
2      incl  rdx
3      cmpl  rdx, 1
4      jle    _loop_body
5      cmpl  rdx, 2
6      jle    _loop_body
7      ...
8  _loop_body:
```

To facilitate signaling of a range change, we flag the start of each sequential range check (e.g., lines 3 and 5 above) with the one-byte `0xCC` interrupt. To maintain control-flow congruence, we apply this transformation to both the oracle and tracer binaries.

5 EVALUATION

Our evaluation of the effectiveness of *coverage-preserving Coverage-guided Tracing* is motivated by three key questions:

[5] In compiler and graph theory, a basic block a is said to *dominate* basic block b if and only if every path through b also covers a. [2]

Q1: Do jump mistargeting and bucketed unrolling improve coverage over basic-block-only CGT?
Q2: What are the performance impacts of expanding CGT to finer-grained code coverage metrics?
Q3: How do the benefits of coverage-preserving CGT impact fuzzing bug-finding effectiveness?

5.1 Experiment Setup

Below we provide expanded detail on our evaluation: the coverage-tracing approaches we are testing, our benchmark selection, and our experimental infrastructure and analysis procedures.

Competing Tracing Approaches: Table 5 lists the fuzzing coverage-tracing approaches tested in our evaluation. We evaluate our binary-only coverage-preserving CGT implementation, HeXcite, alongside the current *block-coverage-only* CGT approach **UnTracer** [37].[6] To test HeXcite's fidelity against the conventional *always-on* coverage tracing in binary fuzzing, we also evaluate the leading binary tracers **QEMU** (AFL [59] and honggFuzz's [49] default approach for fuzzing binary-only targets); **Dyninst** (a popular static-rewriting-based alternative [26]); and **RetroWrite** [15] (a recent static-rewriting-based instrumenter). Lastly, we replicate UnTracer's evaluation for open-source targets by further comparing against **AFL-Clang** (AFL's [59] source-level always-on tracing) [37]. We report HeXcite's best-performing coverage configuration (edge coverage or edge+count coverage) in all experiments.

Approach	Tracing Type	Level	Coverage
HeXcite	coverage-guided	binary	edge + counts
UnTracer [37]	coverage-guided	binary	block
QEMU [59]	always-on	binary	edge + counts
Dyninst [26]	always-on	binary	edge + counts
RetroWrite [15]	always-on	binary	edge + counts
Clang [59]	always-on	source	edge + counts

Table 5: Fuzzing coverage tracers evaluated alongside HeXcite; and their *type*, *level*, and *coverage* metric.

Binary	Package	Source	Input File
jasper	jasper-1.701.0	✔	JPG
mjs	mjs-1.20.1	✔	JS
nasm	nasm-2.10	✔	ASM
sam2p	sam2p-0.49.3	✔	BMP
sfconvert	audiofile-0.2.7	✔	WAV
tcpdump	tcpdump-4.5.1	✔	PCAP
unrtf	unrtf-0.20.0	✔	RTF
yara	yara-3.2.0	✔	YAR
lzturbo	lzturbo-1.2	✗	LZT
pngout	Mar 19 2015	✗	PNG
rar	rarlinux-4.0.0	✗	RAR
unrar	rarlinux-4.0.0	✗	RAR

Table 6: Our evaluation benchmark corpora.

Benchmark Selection: Our benchmark selection (Table 6) follows the current standard in the fuzzing literature, consisting of eight binaries from popular open-source applications varying by input file format (e.g., images, audio, video) and characteristics. Furthermore, as CGT's most popular usage to date [18, 23, 30] is in accelerating *binary-only* fuzzing, we also incorporate a set of four closed-source binary benchmarks distributed as free software. All benchmarks are selected from versions with well-known bugs to ensure a *self-evident* comparison in our bug-finding evaluation.

For each tracing approach we omit benchmarks that are unsupported or fail: `sam2p` and `sfconvert` for QEMU (due to repeated deadlock); `lzturbo`, `pngout`, `rar`, and `unrar` for Dyninst (due to its inability to support closed-source, stripped binaries [38]); `jasper`, `nasm`, `sam2p`, `lzturbo`, `pngout`, `rar`, and `unrar` for RetroWrite (due to crashes on startup and/or being position-dependent/stripped); and `lzturbo`, `pngout`, `rar`, and `unrar` for AFL-Clang (due to it only supporting open-source targets).

Infrastructure: We carry out all evaluations on the Microsoft Azure cloud infrastructure. Each fuzzing trial is issued its own isolated Ubuntu 16.04 x86-64 virtual machine. Following Klees et al.'s [32] standard we run 16×24-hour trials per benchmark for each of the coverage-tracing approaches listed in Table 5, amounting to over 2.4 years' of total compute time across our entire evaluation. All benchmarks are instrumented on an Ubuntu 16.04 x86-64 desktop with a 6-core 3.50GHz Intel Core i7-7800x CPU and 64GB memory. We repurpose the same system for all data post-processing.

5.2 Q1: Coverage Evaluation

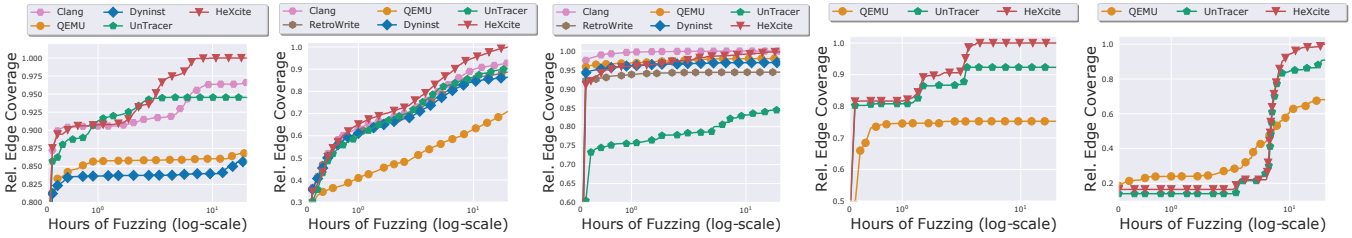
To understand the trade-offs of adapting CGT to finer-grained coverage metrics, we first evaluate HeXcite's code and loop coverage against the block-coverage-only Coverage-guided Tracer UnTracer; as well as conventional always-on coverage-tracing approaches QEMU, Dyninst, RetroWrite, and AFL-Clang. We detail our experimental setup and results below.

5.2.1 ***Code Coverage.*** We compare the code coverage of all tracing approaches in Table 5. We utilize AFL++'s Link Time Optimization (LTO) instrumentation [18] to build *collision-free* edge-tracking versions of each binary; the same technique is applied to our four closed-source benchmarks (Table 6) with the help of the industry-standard binary-to-LLVM lifting tool McSema [14]. We measure each trial's code coverage by replaying its test cases on the LTO binary using AFL's `afl-showmap` [59] utility and compute the average across all 16 trials. Table 7 reports the average across all benchmark–tracer pairs as well as Mann-Whitney U significance scores at the $p = 0.05$ significance level; and Figure 7 shows the relative edge coverage over 24-hours for several benchmarks.

Versus UnTracer: As Table 7 shows, HeXcite surpasses UnTracer in total coverage across all benchmarks by **1–18%** for a mean improvement of **6.2%**, with statistically higher coverage on 10 of 12 benchmarks. The impact of coverage granularity on CGT is significant; besides seeing the worst coverage on `unrtf` (Figure 7c) and `sfconvert`, block-only coverage UnTracer is bested by AFL-Clang on all 8 open-source benchmarks, demonstrating that sheer speed is not enough to overcome a sacrifice in code coverage—whereas HeXcite's *coverage-preserving* CGT averages the highest overall code coverage in our entire evaluation.

Versus binary-only always-on tracing: We see that HeXcite achieves a mean **23.1%**, **18.1%**, and **6.3%** higher code coverage over binary-only always-on tracers QSYM, Dyninst, and RetroWrite (respectively), with statistically significant improvements on all but one binary per comparison (`yara` for QEMU, and `sfconvert` for Dyninst and RetroWrite). For `sfconvert` in particular, we find that all tracers' runs are dominated by timeout-inducing inputs,

[6] As UnTracer is partially reliant on AFL's *source-level* instrumentation and is hence impossible to use on binary-only targets in its original form, we implement a *fully binary-only* version suitable across all 12 of our evaluation benchmarks.



(a) `nasm` (b) `tcpdump` (c) `unrtf` (d) `pngout` (e) `unrar`

Figure 7: HeXcite's mean code coverage over time relative to all supported tracing approaches per benchmark. We log-scale the trial duration (24 hours) to more clearly show the end-of-fuzzing coverage divergence.

Binary	vs. *Coverage-guided Tracing*		vs. Binary- and Source-level *Always-on Tracing*							
	HeXcite / UnTracer		HeXcite / QEMU		HeXcite / Dyninst		HeXcite / RetroWrite		HeXcite / Clang	
	Rel. Cov	MWU	Rel. Cov	MWU	Rel. Cov	MWU	Rel. Cov	MWU	Rel. Cov	MWU
`jasper`	1.04	0.403	1.71	**<0.001**	1.77	**<0.001**	✗	✗	1.01	0.209
`mjs`	1.05	**0.002**	1.07	**<0.001**	1.09	**<0.001**	1.04	**0.001**	1.01	0.231
`nasm`	1.06	**<0.001**	1.15	**<0.001**	1.17	**<0.001**	✗	✗	1.03	**<0.001**
`sam2p`	1.03	**0.003**	✗	✗	1.12	**<0.001**	✗	✗	1.02	0.292
`sfconvert`	1.04	**<0.001**	✗	✗	1.00	0.057	1.00	0.492	0.99	0.031
`tcpdump`	1.11	**<0.001**	1.41	**<0.001**	1.16	**<0.001**	1.13	**<0.001**	1.08	**0.002**
`unrtf`	1.18	**0.002**	1.02	0.168	1.03	**0.041**	1.06	**0.002**	1.00	0.440
`yara`	1.03	0.057	1.08	**0.028**	1.12	**0.034**	1.09	**0.034**	0.95	0.061
`lzturbo`	1.01	**<0.001**	1.06	**<0.001**	✗	✗	✗	✗	✗	✗
`pngout`	1.08	**0.001**	1.33	**<0.001**	✗	✗	✗	✗	✗	✗
`rar`	1.02	**0.004**	1.02	**0.026**	✗	✗	✗	✗	✗	✗
`unrar`	1.10	**0.005**	1.47	**<0.001**	✗	✗	✗	✗	✗	✗
Mean Increase	**+6.2%**		**+23.1%**		**+18.1%**		**+6.3%**		**+1.1%**	

Table 7: HeXcite's mean code coverage relative to UnTracer, QEMU, Dyninst, Retrowrite, and AFL-Clang. ✗ = the competing tracer is incompatible with the respective benchmark and hence omitted. Statistically significant improvements for HeXcite (i.e., Mann-Whitney U test $p < 0.05$) are bolded.

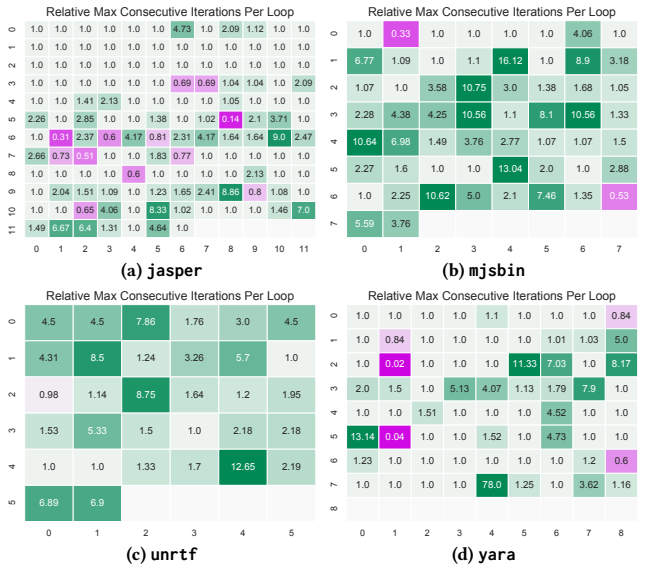
causing each to see roughly equal execution speeds, and hence, code coverage. While we expect that timeout-laden binaries are less likely to see benefit from CGT in general, overall, HeXcite's balance of fine-grained coverage *and* speed easily rank it the highest-coverage binary-only tracer.

Versus source-level always-on tracing: Across all eight open-source benchmarks HeXcite averages **1.1%** higher coverage than AFL's source-level tracing, AFL-Clang. Despite having statistically worse coverage on `sfconvert` (due to its heavy timeouts), HeXcite's coverage is statistically better or identical to AFL-Clang's on 7/8 benchmarks, confirming that coverage-preserving CGT brings coverage tracing *at least as effective as* source-level tracing—to binary-only fuzzing use cases.

Binary	HeXcite / UnTracer Rel. LoopCov	HeXcite / Clang Rel. LoopCov
`jasper`	1.56	1.14
`mjs`	3.61	1.06
`nasm`	2.54	1.85
`sam2p`	1.05	1.19
`sfconvert`	1.89	2.56
`tcpdump`	1.21	1.39
`unrtf`	3.54	0.73
`yara`	2.98	0.95
Mean Increase	**+130%**	**+36%**

Table 8: HeXcite's mean loop coverage (i.e., average maximum consecutive iterations capped at 128) relative to block-only CGT UnTracer and the source-level conventional tracer AFL-Clang.

5.2.2 ***Loop Coverage.*** To determine if coverage-preserving CGT is more effective at covering code loops, we develop a custom LLVM instrumentation pass to report the maximum consecutive iterations per loop per trial. Despite our success in lifting our closed-source benchmarks to add edge-tracking instrumentation (§ 5.2.1),



(a) `jasper` (b) `mjsbin` (c) `unrtf` (d) `yara`

Figure 8: HeXcite's mean loop coverage relative to UnTracer. Each box represents a mutually-covered loop, with values indicating the mean maximum consecutive iterations (capped at 128 total iterations to match AFL) over all 16 trials. *Green* and *pink* shading indicate a higher relative loop coverage for HeXcite and UnTracer (respectively), while *grey* indicates no change.

none of our binary-to-LLVM lifters (McSema, rev.ng, RetDec, reopt, llvm-mctoll, or Ghidra-to-LLVM) succeeded in recovering the loop metadata necessary for our LLVM loop transformation to work; thus our loop analysis is restricted to our eight open-source benchmarks.

We compare HeXcite to UnTracer and AFL-Clang as they support all eight open-source benchmarks (and hence omit QEMU, Dyninst and RetroWrite which only support a few). We compute each loop's mean from the maximum consecutive iterations for all trials per benchmark–tracer pair, capping iterations at 128 as AFL

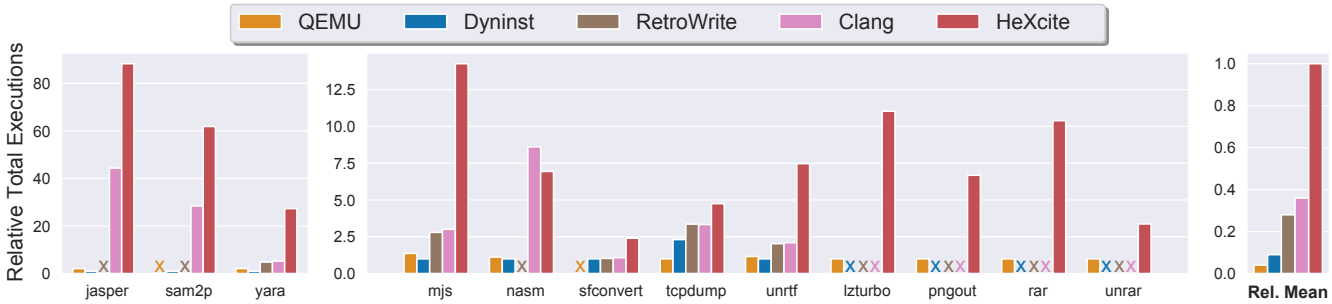


Figure 9: HeXcite's mean throughput relative to conventional coverage tracers. We normalize throughput to the worst-performing tracer per benchmark, and compute each tracer's mean performance relative to HeXcite's across all benchmarks (shown in the rightmost plot). For each benchmark we omit incompatible tracers (denoted by a colored ✗). *All comparisons to HeXcite yield a statistically significant difference* (i.e., Mann-Whitney U test $p < 0.05$).

omits hit counts beyond this range. Table 8 reports HeXcite's mean coverage across all loops for each binary relative to UnTracer and AFL-Clang; and Figure 8 shows a heatmap of HeXcite's per-loop coverage relative to UnTracer's for several benchmarks.

Versus UnTracer: As Table 8 shows, HeXcite's bucketed unrolling brings **130%** higher loop penetration coverage over UnTracer. We see that UnTracer beats HeXcite on a minutia of loops per benchmark (Figure 8)—expectedly—as its inability to track loop progress inevitably constrains fuzzing to exploring the same few loops trial after trial. We find that HeXcite queues over **2**× as many test cases, thus showing that its loop-progress-aware coverage leads fuzzing to sacrifice focusing on the same few loops in favor of a *higher diversity* of loops per binary.

Versus source-level always-on tracing: We see that, on average, HeXcite attains a **36%** higher loop coverage than source-level always-on tracing with AFL-Clang. Though this improvement is modest, these results show that bucketed unrolling outperforms conventional coverage tracing's exhaustive (i.e., on every basic block) hit count tracking—yet only instruments loop headers. While we posit that bucketed unrolling has further optimization potential (e.g., halving the number of buckets, selective insertion, etc.), we leave exploring this trade-off space to future work.

Q1: Jump mistargeting and bucketed unrolling enable Coverage-preserving CGT to achieve the highest overall coverage versus *block-only* CGT—as well as conventional binary *and* source-level tracing.

5.3 Q2: Performance Evaluation

To measure the impacts of finer-grained coverage on CGT performance, we perform a piece-wise evaluation of the fuzzing test case throughput (i.e., mean total test cases processed in 24-hours) of HeXcite's *edge* (via jump mistargeting) and *full* (jump mistargeting + bucketed unrolling) coverage versus UnTracer's block-only coverage, shown in Table 9. To ascertain where coverage-preserving CGT's performance stands with respect to always-on tracing, we further evaluate HeXcite's best-case throughput alongside the leading binary- and source-level coverage tracers QEMU, Dyninst, RetroWrite, and AFL-Clang, shown in Figure 9.

Versus UnTracer: As Table 9 shows, incorporating edge coverage in CGT incurs a mean throughput slowdown of **3%**, while supporting full coverage (i.e., edges *and* counts) sees a slightly higher slowdown of **8%**. However, as the experiments in § 5.2.1 and

Binary	Edge / Block		Full / Block		Best / Block	
	Rel. Perf	MWU	Rel. Perf	MWU	Rel. Perf	MWU
jasper	0.52	<0.001	0.54	<0.001	0.54	<0.001
mjs	0.93	0.046	0.65	<0.001	0.93	0.046
nasm	1.46	<0.001	2.61	<0.001	2.61	<0.001
sam2p	0.99	0.433	1.07	0.090	1.07	0.090
sfconvert	1.06	<0.001	1.24	<0.001	1.24	<0.001
tcpdump	0.96	0.150	0.64	<0.001	0.96	0.150
unrtf	1.04	0.332	0.78	<0.001	1.04	0.332
yara	0.97	0.125	0.18	<0.001	0.97	0.125
lzturbo	0.74	0.292	0.82	0.448	0.82	0.448
pngout	1.02	0.002	0.99	0.332	1.02	0.002
rar	1.01	0.492	0.68	<0.001	1.01	0.492
unrar	0.97	0.188	0.90	0.047	0.97	0.188
Mean Rel. Perf.	**97%**		**92%**		**110%**	

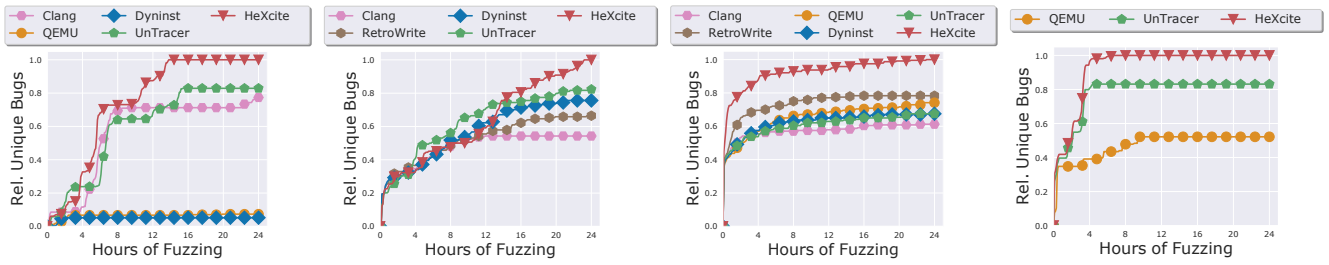
Table 9: Performance trade-offs of different CGT coverage granularities. We compute mean throughputs for three HeXcite coverage granularities (edge, full, and the best of both) relative to UnTracer's *block*-only granularity.

§ 5.2.2 show, coverage-preserving CGT attains the highest edge and loop coverage of all tracers in our evaluation—offsetting the performance deficits expected of finer-grained coverage (e.g., from spending more time covering more loops). Furthermore, as column 3 in Table 9 shows, HeXcite's best-case performance is **nearly indistinguishable** from UnTracer's, with performance statistically improved or identical on all but two benchmarks.

Versus binary-only always-on tracing: As Figure 9 shows, HeXcite averages **11.4**×, **24.1**×, and **3.6**× the throughput of always-on binary-only tracers QEMU, Dyninst, and RetroWrite, respectively. Furthermore, we observe that **all 23** comparisons to HeXcite yield a statistically significant improvement in HeXcite's speed over these competing binary-only tracers.

Versus source-level always-on tracing: HeXcite averages **2.8**× the throughput of AFL's main source-level coverage tracer AFL-Clang. In only one case (`nasm`) does HeXcite face lower a throughput of around 19%; however, the remaining seven open-source benchmarks see HeXcite attaining a statistically higher throughput. Thus, we conclude that HeXcite's coverage-preserving CGT indeed upholds the speed advantages of CGT—outperforming even the ordinarily-fast source-level tracing.

Q2: Coverage-preserving CGT trades-off a negligible amount of speed to attain the highest binary-only code and loop coverage—and still outperforms conventional always-on binary- *and* source-level tracing with over 2–24× the test case throughput.



(a) `nasm` (b) `sfconvert` (c) `unrtf` (d) `pngout`

Figure 10: HeXcite's mean unique bugs over time relative to all supported tracing approaches per benchmark.

Binary	vs. *Coverage-guided Tracing*			vs. Binary- and Source-level *Always-on Tracing*											
	HeXcite / UnTracer			**HeXcite / QEMU**			**HeXcite / Dyninst**			**HeXcite / RetroWrite**			**HeXcite / Clang**		
	Rel. Crash	Rel. Bugs	MWU	Rel. Crash	Rel. Bugs	MWU	Rel. Crash	Rel. Bugs	MWU	Rel. Crash	Rel. Bugs	MWU	Rel. Crash	Rel. Bugs	MWU
`jasper`	1.40	0.97	0.241	25.32	19.92	**<0.001**	42.50	37.00	**<0.001**	✗	✗	✗	1.31	1.12	0.216
`mjs`	1.37	1.02	0.462	17.33	6.71	**<0.001**	12.27	3.38	**<0.001**	5.92	1.84	**<0.001**	5.22	1.82	**<0.001**
`nasm`	1.99	1.21	**<0.001**	20.03	13.63	**<0.001**	18.93	19.24	**<0.001**	✗	✗	✗	1.74	1.27	**<0.001**
`sam2p`	1.43	1.05	0.447	✗	✗	✗	2.24	1.36	**<0.001**	✗	✗	✗	1.32	1.21	**0.018**
`sfconvert`	1.52	1.23	**<0.001**	✗	✗	✗	1.42	1.35	**<0.001**	1.56	1.53	**<0.001**	1.78	1.88	**<0.001**
`tcpdump`	1.28	1.04	0.212	2.43	1.64	**<0.001**	1.91	1.27	**<0.001**	1.29	1.09	**0.048**	1.01	1.05	0.084
`unrtf`	1.88	1.48	**<0.001**	1.37	1.35	**0.001**	1.67	1.46	**<0.001**	1.18	1.28	**0.001**	1.10	1.63	**<0.001**
`yara`	0.72	1.02	0.215	16.80	2.34	**<0.001**	22.49	2.89	**<0.001**	12.58	2.05	**<0.001**	10.47	1.72	**<0.001**
`pngout`	1.27	1.36	**<0.001**	2.49	2.17	**<0.001**	✗	✗	✗	✗	✗	✗	✗	✗	✗
`unrar`	1.25	0.80	0.279	2.00	2.00	**0.039**	✗	✗	✗	✗	✗	✗	✗	✗	✗
Mean Increase	**+41%**	**+12%**		**+997%**	**+521%**		**+1193%**	**+749%**		**+350%**	**+56%**		**+199%**	**+46%**	

Table 10: HeXcite's mean crashes and bugs relative to UnTracer, QEMU, Dyninst, RetroWrite, and AFL-Clang. We omit `lzturbo` and `rar` as none trigger any crashes for them. ✗ = the tracer is incompatible with the respective benchmark and hence omitted. Statistically significant improvements in *mean bugs found* for HeXcite (i.e., Mann-Whitney U test $p < 0.05$) are bolded.

5.4 Q3: Bug-finding Evaluation

We evaluate the crash- and bug-finding effectiveness of coverage-preserving CGT across our 12 benchmarks. To triage raw crashes into bugs, we apply the popular "fuzzy stack hashing" methodology, trimming stack traces to their top-6 entries, and hash each with their corresponding fault address and reported error. We make use of the binary-only AddressSanitizer implementation QASan [17] to extract crash stack traces and errors.

5.4.1 ***Unique Bugs and Crashes.*** Table 10 shows the HeXcite's mean crash- and bug-finding relative to block-coverage-only CGT UnTracer; and always-on fuzzing coverage tracers QEMU, Dyninst, RetroWrite, and AFL-Clang. Figure 10 shows the mean unique crashes over time for several benchmarks. We omit `lzturbo` and `rar` as no fuzzing run found crashes in them.

Versus UnTracer: As Table 10 shows, HeXcite exposes a mean **12%** more bugs than UnTracer. In conjunction with the plots shown in Figure 10, we see that coverage-preserving CGT's small sacrifice in speed is completely offset by the much higher number of bugs and crashes found—attaining effectiveness statistically better than or identical to UnTracer on all 12 benchmarks.

Versus binary-only always-on tracing: As expected, HeXcite's coverage-preserving CGT attains a mean improvement of **521%**, **1193%**, and **56%** in fuzzing bug-finding over always-on binary-only tracers QEMU, Dyninst, and RetroWrite (respectively). Just as in our performance experiments (§ 5.3), **all 21** comparisons yield a statistically significant improvement for HeXcite.

Versus source-level always-on tracing: Across all eight open-source benchmarks, HeXcite achieves a **46%** higher bug-finding effectiveness than source-level tracer AFL-Clang, with statistically improved and statistically identical bug-finding on 6/8 and 2/8 binaries (respectively). Overall, beating even source-level tracers highlights HeXcite's value at *binary-only* coverage.

5.4.2 ***Bug Diversity.*** Following additional triage to map discovered crashes to previously-reported vulnerabilities and bugs, we conduct several case studies to further examine HeXcite's practicality in real-world bug-finding versus existing tracers.

To determine whether coverage-preserving CGT effectively reveals *many* bugs, or is merely constrained to the same few time after time, we compare the total bugs found by HeXcite to the best-performing always-on coverage-tracers, RetroWrite (binary-only) and AFL-Clang (source-level). As Figure 11 shows, despite some overlap, HeXcite reveals **1.4×** the unique bugs as RetroWrite and AFL-Clang—with a higher number of bugs that *only* HeXcite successfully reveals—confirming that coverage-preserving CGT is practical for real-world bug-finding.

(a) HeXcite vs. RetroWrite (b) HeXcite vs. AFL-Clang

Figure 11: HeXcite's total unique bugs found versus the fastest conventional always-on tracers RetroWrite (binary-only) and AFL-Clang (source-level).

5.4.3 ***Bug Time-to-Exposure.*** We further compare HeXcite's mean time-to-exposure for 16 previously-reported bugs versus block-only CGT UnTracer; and always-on coverage tracers QEMU, Dyninst, RetroWrite, and AFL-Clang. As Table 11 shows, HeXcite accelerates bug discovery by **52.4%**, **48.9%**, **41.2%**, **43.5%**, and **32.3%** over UnTracer, QEMU, Dyninst, RetroWrite, and AFL-Clang (respectively). While HeXcite is not the fastest on every bug, its

Identifier	Category	Binary	*Coverage-guided Tracing*		Binary- and Source-level *Always-on Tracing*			
			HeXcite	UnTracer	QEMU	Dyninst	RetroWrite	Clang
CVE-2011-4517	heap overflow	`jasper`	13.1 hrs	18.2 hrs	✗	✗	✗	8.70 hrs
GitHub issue #58-1	stack overflow	`mjs`	13.3 hrs	19.0 hrs	✗	✗	15.30 hrs	✗
GitHub issue #58-2	stack overflow	`mjs`	13.6 hrs	16.4 hrs	✗	22.6 hrs	✗	15.70 hrs
GitHub issue #58-3	stack overflow	`mjs`	5.88 hrs	6.80 hrs	✗	14.7 hrs	✗	✗
GitHub issue #58-4	stack overflow	`mjs`	8.60 hrs	10.7 hrs	✗	20.1 hrs	19.6 hrs	✗
GitHub issue #136	stack overflow	`mjs`	1.30 hrs	7.50 hrs	✗	1.30 hrs	✗	✗
Bugzilla #3392519	null pointer deref	`nasm`	12.1 hrs	13.5 hrs	✗	✗	✗	✗
CVE-2018-8881	heap overflow	`nasm`	5.06 hrs	14.6 hrs	✗	✗	✗	13.9 hrs
CVE-2017-17814	use-after-free	`nasm`	3.54 hrs	6.31 hrs	✗	✗	✗	5.91 hrs
CVE-2017-10686	use-after-free	`nasm`	3.84 hrs	5.40 hrs	✗	✗	✗	4.70 hrs
Bugzilla #3392423	illegal address	`nasm`	8.17 hrs	14.2 hrs	✗	✗	✗	✗
CVE-2008-5824	heap overflow	`sfconvert`	13.1 hrs	14.8 hrs	✗	14.3 hrs	15.4 hrs	✗
CVE-2017-13002	stack over-read	`tcpdump`	8.34 hrs	12.5 hrs	✗	13.5 hrs	11.5 hrs	8.04 hrs
CVE-2017-5923	heap over-read	`yara`	3.24 hrs	5.67 hrs	1.87 hrs	✗	9.33 hrs	6.19 hrs
CVE-2020-29384	integer overflow	`pngout`	5.40 min	34.5 min	18.0 min	✗	✗	✗
CVE-2007-0855	stack overflow	`unrar`	10.7 hrs	17.6 hrs	✗	✗	✗	✗
HeXcite's Mean Relative Speedup				**52.4%**	**48.9%**	**41.2%**	**43.5%**	**32.3%**

Table 11: HeXcite's mean bug time-to-exposure relative to block-coverage-only CGT UnTracer; and conventional always-on coverage tracers QEMU, Dyninst, RetroWrite, and AFL-Clang. ✗ = the competing tracer is incompatible with the benchmark or does not uncover the bug.

overall improvement over competing tracers further substantiates the improved fuzzing effectiveness of coverage-preserving CGT.

> **Q3:** Coverage-preserving CGT's balance of speed and coverage improves fuzzing effectiveness, revealing more bugs than alternative tracing approaches—in less time.

6 DISCUSSION

Below we discuss several limitations of coverage-preserving CGT and our prototype implementation, HeXcite.

6.1 Indirect Critical Edges

While resolving *direct* critical edges is straightforward through jump mistargeting or edge splitting (§ 3.1), *indirect* critical edges (i.e., indirect jumps/calls/returns) remain a universal problem even for source-level solutions like LLVM's SanitizerCoverage [50]. Below we discuss several emerging and/or promising techniques for resolving indirect critical edges, and their trade-offs with respect to supporting a binary-level coverage-preserving CGT.

Block Header Splitting: LLVM's SanitizerCoverage supports resolving indirect critical edges whose end blocks have one or more incoming *direct* edges. For example, given a CFG with indirect critical edge $\overrightarrow{\texttt{ib}}$ (with `i` having outgoing indirect edges to some other blocks `x` and `y`) and direct edge $\overrightarrow{\texttt{ab}}$, SanitizerCoverage first cuts block `b`'s header from its body into two copies, $\texttt{b}_{0i}$ and $\texttt{b}_{0a}$. Second, as the indirect transfer's destination is resolved dynamically and thus cannot be statically moved, $\texttt{b}_{0i}$'s location must be pinned to that of the original block `b`. Finally, the twin header blocks ($\texttt{b}_{0i}$ and $\texttt{b}_{0a}$) are appended with a direct jump to `b`'s body, $\texttt{b}_1$—effectively splitting the original indirect critical edge $\overrightarrow{\texttt{ib}}$ with edges $\overrightarrow{\texttt{ib}_{0i}}$ and $\overrightarrow{\texttt{b}_{0i}\texttt{b}_1}$; and direct edge $\overrightarrow{\texttt{ab}}$ with $\overrightarrow{\texttt{ab}_{0a}}$ and $\overrightarrow{\texttt{b}_{0a}\texttt{b}_1}$. However, the inability to statically alter indirect transfer destinations makes this approach only applicable for indirect critical edges that are the *sole* indirect edge to their end block; i.e., should there be multiple indirect critical edges ($\overrightarrow{\texttt{i}_1\texttt{b}}$ and $\overrightarrow{\texttt{i}_2\texttt{b}}$), *at most one* can be split.

Indirect Branch Promotion: Originally designed as a mitigation for branch target prediction attacks, indirect branch promotion aims to "rewrite" indirect transfers as direct: at runtime, each dynamically-resolved indirect branch target is compared to several statically-encoded candidates, with a conditional jump to each should the comparison match (e.g., `if(%eax == foo): jump foo`). While promotion is applicable to nearly all indirect branches (and hence indirect critical edges), branch target prediction accuracy is *never* guaranteed. Existing approaches attempt to maximize precision by profiling indirect branches in advance for their "most probable" targets, however, fuzzing may expose (and prioritize) new targets previously considered unlikely by profiling.

Hybrid Instrumentation: A third possibility for indirect critical edges is to default back to AFL-style hashing-based edge coverage (§ 2.3). While it is impossible to identify each indirect edge's targets accurately, a conservative approach is to instead instrument the set of *all* potential indirect branch targets, as their heuristics are generally well-known (e.g., function entrypoints for indirect calls, and post-call blocks for returns). We can thus imagine future *target-tailored* CGT approaches balancing fast speed for common-case critical edges with more precise handling (e.g., header splitting, promotion, and hybrid instrumentation) of infrequent ones.

6.2 Trade-offs of Hit Count Coverage

Hit counts measure fuzzing exploration progress in loops and cycles, but as with any coverage metric, their implementation must carefully balance precision and speed to support effective bug-finding. Two considerations central to hit count coverage implementations are (1) the size and number of bucket ranges; and (2) the frequency at which hit counts are tracked. We discuss both of these below.

Bucket Granularity: Our current implementation of bucketed unrolling (§ 4.3) mimics the hit count tracking of conventional fuzzers by injecting conditional checks against eight bucket ranges (0–1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+). However, *these eight* bucket ranges are merely an artifact of AFL's original implementation (each hashed edge is mapped to an 8-bit index in its coverage bitmap). Adding *more* buckets makes it possible to track more subtle changes in loop iteration counts, while using *fewer* buckets trades-off this level of introspection for higher fuzzing throughput. While it is unclear which bucket ranges achieve the *best* balance of speed and coverage with respect to bug-finding, we expect that future research will address these unanswered questions and more.

Frequency of Tracking: How often hit counts are tracked further influences fuzzing exploration and bug-finding. Conventional *exhaustive* (per-edge) hit counts shed light on frequencies of cycle subpaths (e.g., how many times a loop `break` is taken), but risk

saturating a fuzzer's search space with redundant or noisy paths. Bucketed unrolling instead trades-off coverage exhaustiveness for speed by restricting hit count tracking to only a *subset* of the program state (e.g., loop iteration counters). While our analysis of the bugs exclusively found by exhaustive hit counts (Figure 11b) reveals that none are outside the reach of HeXcite, we expect that future work will explore adapting *selective* and *synergistic* hit count schemes to better cover complex loops, cycles, and compiler optimizations at high speed.

6.3 Improving Performance

The fuzzing-oriented binary transformation platform currently utilized in HeXcite, ZAFL [38], adopts a code layout algorithm that rewrites all direct jumps to have 32-bit PC-relative signed displacements. While this is well-suited to our implementation of zero-address jump mistargeting (§ 4.2)—enabling virtually every conditional jump in the program's address space to be mistargeted to `0x00`—32-bit displacements accumulate more runtime overhead over 8–16-bit displacements. As ZAFL has experimental code layouts that instead prioritize smaller displacements, we thus envision potential for faster "hybrid" mistargeting schemes that coalesce both zero-address *and* embedded interrupt styles.

6.4 Supporting Other Software & Platforms

Our current coverage-preserving CGT prototype, HeXcite, supports 64-bit Linux C and C++ binaries. Extending support to other software characteristics (e.g., 32-bit) or platforms (e.g., Windows) requires retooling of its underlying static binary rewriting engine. However, as this component is orthogonal to the fundamental principles of coverage-preserving CGT, we expect that HeXcite will capitalize on future engineering improvements in static rewriting to bring accelerated fuzzing to the broader software ecosystem.

7 RELATED WORK

We discuss recent efforts to improve binary-only fuzzing performance that are orthogonal to coverage-preserving CGT: (1) faster instrumentation, (2) less instrumentation, and (3) faster execution.

7.1 Faster Instrumentation

As binary fuzzing effectiveness depends heavily on maintaining fast coverage tracking, a growing body of research is targeting instrumentation-side optimizations. Efforts to improve dynamic translation-based instrumentation (e.g., AFL-QEMU [59], DrAFL [47], UnicornAFL [52]) generally focus on simplifying or expanding the caching of translated code [4]; while those using static rewriting (e.g., ZAFL [38], Dyninst [40], RetroWrite [15]) tackle various challenges related to generated code performance. Though our coverage-preserving CGT prototype, HeXcite, currently leverages the ZAFL rewriter, we believe that future advances in binary instrumentation will enable it to achieve performance even closer to native speed.

7.2 Less Instrumentation

Another way to reduce the footprint of coverage tracking is to eliminate needless instrumentation from the program under test. While most other control-flow-centric approaches only exist in compiler instrumentation-based implementation (e.g., dominator trees [2], INSTRIM [28], CollAFL [19]), their principles are well-suited to binary-only fuzzing. A recent fork of AFL-Dyninst [26] omits instrumentation from blocks preceded by unconditional direct transfer, as their coverage is directly implied by their ancestor's. In addition to accelerating execution of HeXcite's tracer binary, we see the potential for such control-flow-centric analyses to help determine how HeXcite's control-flow-altering transformations (e.g., bucketed unrolling) should optimally be applied.

7.3 Faster Execution

Besides instrumentation, execution is itself a bottleneck to fuzzing, as faster execution enables more test cases to be run on the target program in less time. Most modern binary-only fuzzing efforts have abandoned slow process creation-based execution for faster snapshotting, leveraging cheap copy-on-write cloning to rapidly initiate target execution from a pre-initialized state (e.g., AFL's forkserver [59]). Xu et al. [55] achieve even faster snapshotting through fuzzing-optimized Linux kernel extensions. The recent technique of persistent/in-memory execution offers higher speed by restricting execution to only a pre-specified target program code region (essentially interposing a loop), and is gaining support among popular binary-only fuzzing toolchains (e.g., WinAFL [22], AFL-QEMU, UnicornAFL). Many efforts are also exploring the benefits of amortizing fuzzing execution speed through parallelization; off-the-shelf binary-only fuzzers like AFL [59] and honggFuzz [49] support parallelization out-of-the-box, and recent work by Falk [16] achieves even faster speed by leveraging vectorized instruction sets. As execution and coverage tracking work hand-in-hand during fuzzing, we view such accelerated execution mechanisms as complementary to HeXcite's accelerated coverage tracking.

8 CONCLUSION

Coverage-preserving Coverage-guided Tracing extends the principles behind CGT's performance-maximizing, waste-eliminating tracing strategy to the finer-gained coverage metrics it is not naturally supportive of: edge coverage and hit counts. We introduce program transformations that enhance CGT's introspection capabilities while upholding its minimally-invasive nature; and show how these techniques improve binary-only fuzzing effectiveness over conventional CGT, while keeping an orders-of-magnitude performance advantage over the leading binary-only coverage tracers.

Our results reveal it is finally possible for today's state-of-the-art coverage-guided fuzzers to embrace the acceleration of CGT—without sacrificing coverage. We envision a new era in software fuzzing, where synergistic and target-tailored approaches will maximize *common-case* performance with *infrequent-case* precision.

ACKNOWLEDGEMENT

We thank our shepherd Jun Xu and our reviewers for helping us improve the paper. We also thank Peter Goodman and Trail of Bits for assisting us with binary-to-LLVM lifting. This material is based upon work supported by the Defense Advanced Research Projects Agency under Contract No. W911NF-18-C-0019 and the National Science Foundation under Grant No. 1650540.

REFERENCES

[1] laf-intel: Circumventing Fuzzing Roadblocks with Compiler Transformations, 2016. URL: https://lafintel.wordpress.com/.

[2] Hiralal Agrawal. Dominators, Super Blocks, and Program Coverage. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 1994.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Network and Distributed System Security Symposium*, NDSS, 2018.

[4] Andrea Biondo. Improving AFL's QEMU mode performance, 2018. URL: https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[5] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, USENIX, 2019.

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2016.

[8] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. Technical report, 2012. URL: https://www.microsoft.com/en-us/research/publication/billions-and-billions-of-constraints-whitebox-fuzz-testing-in-production/.

[9] Peng Chen and Hao Chen. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, Oakland, 2018.

[10] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing Deeply Nested Branches. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2019.

[11] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Taowei, and Long Lu. SAVIOR: Towards Bug-Driven Hybrid Testing. In *IEEE Symposium on Security and Privacy*, Oakland, 2020. arXiv: 1906.07327.

[12] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*, USENIX, 2019.

[13] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box Concolic Testing on Binary Code. In *International Conference on Software Engineering*, ICSE, 2019.

[14] Artem Dinaburg and Andrew Ruef. McSema: Static Translation of X86 Instructions to LLVM, 2014. URL: https://github.com/trailofbits/mcsema.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy*, Oakland, 2020.

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

[17] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *IEEE Secure Development Conference*, SecDev, 2020.

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*, WOOT, 2020.

[19] 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*, Oakland, 2018.

[20] GNU Project. GNU gprof, 2018. URL: https://sourceware.org/binutils/docs/gprof/.

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

[22] Google Project Zero. WinAFL, 2016. URL: https://github.com/googleprojectzero/winafl.

[23] Samuel Groß and Google Project Zero. Fuzzing ImageIO, 2020. URL: https://googleprojectzero.blogspot.com/2020/04/fuzzing-imageio.html.

[24] Ilfak Guilfanov and Hex-Rays. IDA, 2019. URL: https://www.hex-rays.com/products/ida/.

[25] William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. Zipr: Efficient Static Binary Rewriting for Security. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2017.

[26] Marc Heuse. AFL-Dyninst, 2018. URL: https://github.com/vanhauser-thc/afl-dyninst.

[27] Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, and Jack Davidson. Zipr++: Exceptional Binary Rewriting. In *Workshop on Forming an Ecosystem Around Software Transformation*, FEAST, 2017.

[28] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *NDSS Workshop on Binary Analysis Research*, BAR, 2018.

[29] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. TIFF: Using Input Type Inference To Improve Fuzzing. In *Annual Computer Security Applications Conference*, ACSAC, 2018.

[30] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Network and Distributed System Security Symposium*, NDSS, 2021.

[31] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining Indirect Call Targets at the Binary Level. In *Network and Distributed System Security Symposium*, NDSS, 2021.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[33] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, CGO, 2004.

[34] Caroline Lemieux and Koushik Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *ACM/IEEE International Conference on Automated Software Engineering*, ASE, 2018.

[35] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *ACM Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, 2017.

[36] Chenyang Lv, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimize Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, USENIX, 2019.

[37] Stefan Nagy and Matthew Hicks. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *IEEE Symposium on Security and Privacy*, Oakland, 2019.

[38] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing. In *USENIX Security Symposium*, USENIX, 2021.

[39] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *IEEE Symposium on Security and Privacy*, Oakland, 2021.

[40] Paradyn Tools Project. Dyninst API, 2018. URL: https://dyninst.org/dyninst.

[41] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[42] Ganesan Ramalingam. On Loops, Dominators, and Dominance Frontiers. *ACM transactions on Programming Languages and Systems*, page 22, 2002.

[43] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium*, NDSS, 2017.

[44] Sanjay Rawat and Laurent Mounier. Finding Buffer Overflow Inducing Loops in Binary Executables. In *IEEE International Conference on Software Security and Reliability*, SERE, 2012.

[45] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *IEEE Cybersecurity Development Conference*, SecDev, 2016.

[46] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *IEEE Symposium on Security and Privacy*, Oakland, 2019.

[47] Maksim Shudrak and Battelle. drAFL, 2019. URL: https://github.com/mxmssh/drAFL.

[48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Network and Distributed System Security Symposium*, NDSS, 2016.

[49] Robert Swiecki. honggfuzz, 2018. URL: http://honggfuzz.com/.

[50] The Clang Team. SanitizerCoverage, 2019. URL: https://clang.llvm.org/docs/SanitizerCoverage.html.

[51] Fabian Toepfer and Dominik Maier. BSOD: Binary-only Scalable fuzzing Of device Drivers. In *International Symposium on Research in Attacks, Intrusions and Defenses*, RAID, 2021.

[52] Nathan Voss and Battelle. AFL-Unicorn, 2019. URL: https://github.com/Battelle/afl-unicorn.

[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-Aware Greybox Fuzzing. In *International Conference on Software Engineering*, ICSE, 2019. arXiv: 1812.01197.

[54] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From Hack to Elaborate Technique—A Survey on Binary Rewriting. *ACM Computing Surveys*, 52(3), 2019.

[55] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[56] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. SLF: Fuzzing without Valid Seed Inputs. In *International Conference on Software Engineering*, ICSE, 2019.

[57] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery. In *IEEE Symposium on Security and Privacy*, Oakland,

2019.

[58] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, USENIX, 2018.

[59] Michal Zalewski. American fuzzy lop, 2017. URL: http://lcamtuf.coredump.cx/afl/.

[60] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Network and Distributed System Security Symposium*, NDSS, 2019.