Practical Feedback and Instrumentation Enhancements for Performant Security Testing of Closed-source Executables

Stefan Nagy

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Matthew Hicks, Chair
Danfeng (Daphne) Yao
Na Meng
Gang Wang
Taesoo Kim

April 15, 2022
Blacksburg, Virginia

Keywords: Fuzzing, Binaries, Instrumentation, Coverage, Testing, Tracing, Security
Copyright 2022, Stefan Nagy

Practical Feedback and Instrumentation Enhancements for Performant Security Testing of Closed-source Executables

Stefan Nagy

(ABSTRACT)

The Department of Homeland Security reports that over 90% of cyberattacks stem from security vulnerabilities in software, costing the U.S. $109 billion dollars in damages in 2016 alone according to The White House. As NIST estimates that today's software contains 25 bugs for every 1,000 lines of code, the prompt discovery of security flaws is now vital to mitigating the next major cyberattack. Over the last decade, the software industry has overwhelmingly turned to a lightweight defect discovery approach known as *fuzzing*: automated testing that uncovers program bugs through repeated injection of randomly-mutated test cases. Academic and industry efforts have long exploited the semantic richness of *open-source* software to enhance fuzzing with *fast and fine-grained code coverage feedback*, as well as *fuzzing-enhancing code transformations* facilitated through lightweight compiler-based instrumentation. However, the world's increasing reliance on *closed-source* software (i.e., commercial, proprietary, and legacy software) demands analogous advances in automated security vetting beyond open-source contexts.

Unfortunately, the semantic gaps between source code and opaque *binary* code leave fuzzing nowhere near as effective on closed-source targets. The difficulty of balancing coverage feedback speed and precision in binary executables leaves fuzzers frequently bottlenecked and orders-of-magnitude slower at uncovering security vulnerabilities in closed-source software. Moreover, the challenges of analyzing and modifying binary executables at scale leaves closed-source software fuzzing unable to fully leverage the sophisticated enhancements that have long accelerated open-source software vulnerability discovery. As the U.S. Cybersecu-

rity and Infrastructure Security Agency reports that closed-source software makes up over 80% of the top routinely exploited software today, combating the ever-growing threat of cyberattacks demands new *practical*, *precise*, and *performant* fuzzing techniques *unrestricted by the availability of source code.*

This thesis answers the following research questions toward enabling fast, effective fuzzing of closed-source software:

1. Can common-case fuzzing insights be exploited to more achieve low-overhead, fine-grained code coverage feedback irrespective of access to source code?

2. What properties of binary instrumentation are needed to extend performant fuzzing-enhancing program transformation to closed-source software fuzzing?

In answering these questions, this thesis produces the following key innovations:

A. The first code coverage techniques to enable fuzzing speed and code coverage greater than source-level fuzzing for closed-source software targets. (chapter 3)

B. The first instrumentation platform to extend both compiler-quality code transformation and compiler-level speed to closed-source fuzzing contexts (chapter 4)

Practical Feedback and Instrumentation Enhancements for Performant Security Testing of Closed-source Executables

Stefan Nagy

(GENERAL AUDIENCE ABSTRACT)

The Department of Homeland Security reports that over 90% of cyberattacks stem from security vulnerabilities in software, costing the U.S. $109 billion dollars in damages in 2016 alone according to The White House. As NIST estimates that today's software contains 25 bugs for every 1,000 lines of code, the prompt discovery of security flaws is now vital to mitigating the next major cyberattack. Over the last decade, the software industry has overwhelmingly turned to lightweight defect discovery through *automated testing*, uncovering program bugs through the repeated injection of randomly-mutated test cases. Academic and industry efforts have long exploited the semantic richness of *open-source* software (i.e., software whose full internals are publicly available, interpretable, and changeable) to enhance testing with *fast and fine-grained exploration feedback*; as well as *testing-enhancing program transformations* facilitated during the process by which program executables are generated. However, the world's increasing reliance on *closed-source* software (i.e., software whose internals are opaque to anyone but its original developer) like commercial, proprietary, and legacy programs demands analogous advances in automated security vetting beyond open-source contexts.

Unfortunately, the challenges of understanding programs without their full source information leaves testing nowhere near as effective on closed-source programs. The difficulty of balancing exploration feedback speed and precision in program executables leaves testing frequently bottlenecked and orders-of-magnitude slower at uncovering security vulnerabilities in closed-source software. Moreover, the challenges of analyzing and modifying program

executables at scale leaves closed-source software testing unable to fully leverage the sophisticated enhancements that have long accelerated open-source software vulnerability discovery. As the U.S. Cybersecurity and Infrastructure Security Agency reports that closed-source software makes up over 80% of the top routinely exploited software today, combating the ever-growing threat of cyberattacks demands new *practical*, *precise*, and *performant* software testing techniques *unrestricted by the availability of programs' source code.*

This thesis answers the following research questions toward enabling fast, effective fuzzing of closed-source software:

1. Can common-case testing insights be exploited to more achieve low-overhead, fine-grained exploration feedback irrespective of access to programs' source code?

2. What properties of program modification techniques are needed to extend performant testing-enhancing program transformations to closed-source programs?

In answering these questions, this thesis produces the following key innovations:

A. The first techniques enabling testing of closed-source programs with speed and exploration higher than on open-source programs. (chapter 3)

B. The first platform to extend high-speed program transformations from open-source programs to closed-source ones (chapter 4)

Dedication

To my loving parents Lidia and Gabriel, my brother Gabe, and my late grandmother Tamara, for your unrelenting love and support.

Acknowledgments

I am forever thankful to my loving parents Lidia Nagy and Dr. Gabriel Nagy, my brother Dr. Gabe Nagy, and my late grandmother Tamara Nagy, for instilling in me the importance of an education, and for their unrelenting love, patience, and sacrifices for me throughout my childhood, and undergraduate and graduate educational journeys.

I would not dream of being where I am today were it not for my advisor, Dr. Matthew Hicks, taking a chance hiring me as his first Ph.D. student back in November of 2017. Being your student and apprentice has been one of the greatest blessings of my life, and I am forever grateful for all of the mentorship, support, patience, and friendship that you've shown me throughout these last five years. You created something out of nothing in me, and if I am able to do this just half as well as you, I will consider myself a success.

I owe a tremendous thanks to my amazing collaborators and mentors: Dr. Jason Hiser, Dr. Anh Nguyen-Tuong, and Dr. Jack Davidson from the University of Virginia; Peter Goodman from Trail of Bits; Tim Leek from MIT Lincoln Lab; Prashast Srivastava and Dr. Mathias Payer from Purdue University and EPFL; Dr. Sibin Mohan from Oregon State University; and Dr. Gang Wang and Dr. Roy Campbell from UIUC. I've learned so much from each and every one of you, and I am forever grateful for all of the time you've spent patiently helping me grow as a student and researcher.

I feel incredibly blessed to have befriended so many amazing people throughout my time at Virginia Tech: my labmates Jubayer Mahmod, Harrison Williams, Ian Paterson, Dr. Michael

Moukarzel, and the other members of the FoRTE-Research group, for their never-ending support, friendship, and passionate ping-pong rivalries; Dr. Thomas Lux, Dr. Ayaan Kazerouni, Dr. James Davis, Dr. Tyler Chang, and the many other friends I've made around the Department of Computer Science; the Oddos, the Wilkersons, the Holy Trinity Greek Orthodox Church community, and the Roanoke Romanian community; and the endless list of people that have welcomed me into Blacksburg and the New River Valley with open arms and treated me like family. The memories, laughs, and relationships that I've been blessed to have here have made me certain of one thing: *there is no place as special as Blacksburg, Virginia.* I will sorely miss this community and everyone that has made it my home.

I am thankful for my other close friends Leroy Ekechukwu, Stephen Umunna, and everyone else that I've befriended during college at UIUC. I cherish my friendships with each of you, and I'm grateful for your unwavering encouragement throughout my time in grad school.

Lastly, I want to thank Erin Lucy, who has been my greatest rock and cheerleader over the last few years. I love you, and I'm beyond excited about the future we'll build together.

Funding Acknowledgment: The work in this dissertation was supported in part by the Defense Advanced Research Projects Agency under Contract No. W911NF-18-C-0019, and the National Science Foundation under Grant No. 1650540.

Declaration of Collaboration: In addition to my advisor, Dr. Matthew Hicks, the research presented in this dissertation benefited from several collaborators from the University of Virginia: Dr. Jason Hiser, Dr. Anh Nguyen-Tuong, and Dr. Jack Davidson.

Publications from this Thesis

This thesis is derived from the following coauthored publications:

1. **Stefan Nagy** and Matthew Hicks. *Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing.* In Proceedings of the 2019 IEEE Symposium on Security and Privacy (**Oakland'19**). May 2019.

2. **Stefan Nagy**, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. *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 2021.

3. **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 Proceedings of the 2021 USENIX Security Symposium (**USENIX'21**). August 2021.

All figures used in this document are of my own creation.

Contents

List of Figures **xvi**

List of Tables **xx**

1 Introduction **1**

1.1 Problem Definition 2

1.2 Contributions of This Thesis 5

2 Literature Review **6**

2.1 An Overview of Fuzzing 6

2.1.1 Coverage-Guided Fuzzing 8

2.2 Fuzzing's Code Coverage Metrics 10

2.3 Coverage Tracing Performance 12

3 Fast and Fine-grained Coverage for Closed-source Software Fuzzing **13**

3.1 Introduction 13

3.1.1 Coverage-guided Tracing 14

3.1.2 Coverage-*preserving* Coverage-guided Tracing 16

3.2 Impact of Discarded Test Cases 18

3.2.1 Experimental Setup 20
3.2.2 Results 20
3.3 Coverage-guided Tracing 23
3.3.1 Overview 23
3.3.2 The Interest Oracle 24
3.3.3 Tracing 25
3.3.4 Unmodifying 25
3.3.5 Theoretical Performance Impact 26
3.4 Implementation: UnTracer 27
3.4.1 UnTracer Overview 27
3.4.2 Forkserver Instrumentation 30
3.4.3 Interest Oracle Binary 31
3.4.4 Tracer Binary 31
3.4.5 Unmodifying the Oracle 32
3.5 Tracing-only Evaluation 33
3.5.1 Evaluation Overview 34
3.5.2 Experiment Infrastructure 35
3.5.3 Benchmarks 35
3.5.4 Timeouts 38
3.5.5 UnTracer versus Coverage-agnostic Tracing 39

3.5.6 Dissecting UnTracer's Overhead 44
3.5.7 Overhead versus Rate of Coverage-increasing Test Cases 47
3.6 Hybrid Fuzzing Evaluation 50
3.6.1 Evaluation Overview 51
3.6.2 Performance of UnTracer-based Hybrid Fuzzing 53
3.7 A Coverage-*preserving* Coverage-guided Tracing 53
3.7.1 Supporting Edge Coverage 55
3.7.2 Supporting Hit Counts 59
3.8 Implementation: HeXcite 62
3.8.1 Architectural Overview 62
3.8.2 Implementing Jump Mistargeting 64
3.8.3 Implementing Bucketed Unrolling 67
3.9 Evaluation 68
3.9.1 Experiment Setup 69
3.9.2 Q1: Coverage Evaluation 71
3.9.3 Q2: Performance Evaluation 77
3.9.4 Q3: Bug-finding Evaluation 80
3.10 Discussion 84
3.10.1 Indirect Critical Edges 86
3.10.2 Trade-offs of Hit Count Coverage 87

3.10.3 Improving Performance 88

3.10.4 Supporting Other Software & Platforms 89

3.11 Related Work 89

3.11.1 Faster Instrumentation 89

3.11.2 Less Instrumentation 90

3.11.3 Faster Execution 90

3.12 Conclusion 91

4 Compiler-quality Code Transformation for Closed-source Fuzzing 92

4.1 Introduction 92

4.2 Compiler-based Fuzzing Enhancements 95

4.2.1 Instrumentation Pruning 96

4.2.2 Instrumentation Downgrading 96

4.2.3 Sub-instruction Profiling 97

4.2.4 Extra-coverage Behavior Tracking 97

4.3 Binary-only Fuzzing: the Bad & the Ugly 97

4.3.1 Limitations of Existing Platforms 98

4.3.2 Fundamental Design Considerations 101

4.4 The Zafl Platform 104

4.4.1 Design Overview 104

4.4.2 The Zax Transformation Architecture 107

4.5 Extending Compiler-quality Transforms to Binary-only Fuzzing 108

4.5.1 Performance-enhancing Transformations 109

4.5.2 Feedback-enhancing Transformations 111

4.6 Evaluation 112

4.6.1 Evaluation-wide Instrumenter Setup 113

4.6.2 LAVA-M Benchmarking 114

4.6.3 Fuzzing Real-world Software 117

4.6.4 Fuzzing Closed-source Binaries 124

4.6.5 Scalability and Precision 127

4.7 Limitations 131

4.7.1 Improving Baseline Performance 132

4.7.2 Supporting New Architectures, Formats, and Platforms 132

4.7.3 Static Rewriting's Limitations 133

4.8 Related Work 134

4.8.1 Static Binary Rewriting 134

4.8.2 Improving Fuzzing Test Case Generation 134

4.8.3 Hybrid Fuzzing 135

4.8.4 Emergent Fuzzing Transformations 135

4.9 Conclusion 136

5 Conclusion and Future Work **140**

5.1 Future Work: Improving *Scalability* of Closed-source Software Fuzzing Program Transformation 140

5.1.1 Tackling Asymmetries in Program Analysis 141

5.1.2 Tackling Asymmetries in Automated Testing 143

Bibliography **145**

List of Figures

2.1 A taxonomy of popular fuzzers by test case generation and program analysis approaches. 7

2.2 High-level architecture of a coverage-guided mutational fuzzer. 9

2.3 An example of basic blocks in C code. 11

3.1 Visualization of how Coverage-guided Tracing augments the workflow of a conventional coverage-guided grey-box fuzzer (e.g., AFL [146]). Coverage-guided Tracing can also be similarly adapted into coverage-guided white-box fuzzers (e.g., Driller [117]). 24

3.2 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. 26

3.3 UnTracer's workflow. Not shown is test case generation, or starting/stopping forkservers. 28

3.4 Per-benchmark relative overheads of UnTracer versus black-box binary tracers AFL-QEMU and AFL-Dyninst. 41

3.5 Per-benchmark relative overheads of UnTracer versus white-box binary tracer AFL-Clang. 43

3.6 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. 44

3.7 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. 45

3.8 Visualization of the overheads per UnTracer's four components related to coverage-increasing test case processing for each benchmark. 46

3.9 The rates of coverage-increasing test cases encountered over the total number of test cases processed, per benchmark. 48

3.10 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 my evaluations (4.92E-3). 49

3.11 Per-benchmark relative average executions in 24 hours of QSYM-UnTracer versus QSYM-QEMU and QSYM-Clang. 52

3.12 Visualization of the proportion of critical edges by transfer type encountered throughout fuzzing. 54

3.13 A visualization of jump mistargeting via embedded (left) and zero-address interrupts (right). 58

3.14 Bucketed unrolling applied to a simple loop. 61

3.15 HeXcite's fuzzer-side test case handling logic. Like UnTracer, I discard timeout-producing test cases; however, I 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). 65

3.16 HeXcite's mean code coverage over time relative to all supported tracing approaches per benchmark. I log-scale the trial duration (24 hours) to more clearly show the end-of-fuzzing coverage divergence. 72

3.17 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. 75

3.18 HeXcite's mean throughput relative to conventional coverage tracers. I 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 I omit incompatible tracers (denoted by a colored ✗). *All comparisons to* HeXcite *yield a statistically significant difference* (i.e., Mann-Whitney U test $p < 0.05$). 78

3.19 HeXcite's mean unique bugs over time relative to all supported tracing approaches per benchmark. 82

3.20 HeXcite's total unique bugs found versus the fastest conventional always-on tracers RetroWrite (binary-only) and AFL-Clang (source-level). 83

4.1 A high-level depiction of the Zafl platform architecture and its four Zax transformation and instrumentation phases. 105

4.2 Real-world software fuzzing unique triaged crashes averaged over 8×24-hour trials. 119

4.3 Compiler, assembler, AFL-Dyninst, AFL-QEMU, and Zafl fuzzing instrumentation performance relative to baseline (higher is better). 122

4.4 Closed-source binary fuzzing unique triaged crashes averaged over 5×24-hour trials. 125

4.5 A comparison of Zafl's runtime overhead with and without register liveness-aware instrumentation optimization (lower is better). 129

List of Tables

2.1 A survey of recent coverage-guided fuzzers and their coverage metrics (edges/blocks and hit counts). Key: ➤ (edges), ■ (blocks). 10

3.1 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). I run each fuzzer for one hour per benchmark. 21

3.2 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). 22

3.3 Information on the eight benchmarks used in my evaluation in section 3.5 and section 3.6 and averages over 5 24-hour datasets for each benchmark. 36

3.4 Proportion of critical edges in eight real-world programs. 55

3.5 Examples of x86 critical edge instructions by transfer type. 56

3.6 Proportion of encountered critical edges by transfer type. 56

3.7 Fuzzing coverage tracers evaluated alongside HeXcite; and their *type*, *level*, and *coverage* metric. 69

3.8 My evaluation benchmark corpora. 70

3.9 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. 73

3.10 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. 76

3.11 Performance trade-offs of different CGT coverage granularities. I compute mean throughputs for three **HeXcite** coverage granularities (edge, full, and the best of both) relative to UnTracer's *block*-only granularity. 79

3.12 HeXcite's mean crashes and bugs relative to UnTracer, QEMU, Dyninst, RetroWrite, and AFL-Clang. I 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. 81

3.13 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. 85

4.1 Popular compiler-based fuzzing-enhancing program transformations, listed by category and effect. 95

4.2 A qualitative comparison of the leading coverage-tracing methodologies currently used in binary-only coverage-guided fuzzing, alongside compiler instrumentation (LLVM). No existing approaches are able to support compiler-quality transformation at compiler-level speed and generalizability. 100

4.3 A catalog of Zafl-implemented compiler-quality fuzzing-enhancing program transformations and their compiler-based origins. 109

4.4 Zafl's LAVA-M mean bugs and total/queued test cases relative to AFL-Dyninst and AFL-QEMU. I report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance). ✗ = Zafl finds crashes while competitor finds zero. 116

4.5 Zafl's real-world software mean triaged crashes and total/queued test cases rel. to AFL-Dyninst and AFL-QEMU. I report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance). 120

4.6 Zafl's closed-source binary mean triaged crashes and total/queued test cases relative to AFL-Dyninst and AFL-QEMU. I report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance). ✗ = Zafl finds crashes while competitor finds zero. 126

4.7 Mean time-to-discovery of closed-source binary bugs found for AFL-Dyninst, AFL-QEMU, and Zafl over 5×24-hour fuzzing trials. ✗ = bug is not reached in any trials for that instrumenter configuration. 127

4.8 Open-source binaries tested successfully with Zafl. L 137

4.9 Closed-source binaries tested successfully with Zafl. L/W = Linux/Windows; D/I = position-dependent/independent; *Sym* = binary is non-stripped; *Opt* = whether register liveness-aware optimization succeeds. 138

4.10 Instruction recovery statistics for IDA Pro, Binary Ninja, and Zafl, with ground-truth disassembly from LLVM-10's objdump. *Reached* = mean unrecovered instructions reached by fuzzing (hence, erroneously-unrecovered); *FalseNeg* = erroneously-unrecovered instructions over total. 139

4.11 Zafl's fuzzing code coverage true positive and true negative rates, and accuracy with respect to the LLVM compiler over 5×24-hour trials. 139

Chapter 1

Introduction

Software vulnerabilities represent one of the greatest threats to the connected world, costing the global economy hundreds of billions of dollars in damages each year [124]. As the software ecosystem continues expand at a rapid pace, vetting the security of software has become more and more tedious; today's increasingly large and complex software has rendered previous-generation approaches like manual analysis and formal verification infeasible. Instead, software developers and bug-hunters are overwhelmingly embracing a lighter-weight automated testing strategy known as coverage-guided grey-box *fuzzing*.

Fuzzing uncovers software flaws through the aggressive, high-volume injection of randomly-generated test cases. Having exposed tens of thousands of critical vulnerabilities over the last two decades, fuzzing has become the de-facto standard technique for security auditing both throughout and beyond the software development life cycle. Today, software companies dedicate entire server farms to fuzzing at cloud scale, with Google's having found over 16,000 bugs in the Chrome browser alone [49] and Microsoft's revealing one third of all critical Windows 7 vulnerabilities [17]. Moreover, a number of community-driven efforts continue to make fuzzing more powerful and accessible to all [41, 107, 108, 119].

Though developers have the ability to audit their own software's source code, today's computing world is increasingly *closed-source*. Modern commodity software is either fully or partially proprietary, with many formerly open-source projects becoming closed-source for intellectual property reasons [35, 39, 42]. Legacy software also poses a challenge as it was

written in esoteric languages by defunct companies, leaving its source code seldom available. Lastly, many of the software platforms that society depends on (e.g., Microsoft Windows and Apple MacOS) have long remained closed-source since their inceptions decades ago. As industry research shows that closed-source software makes up **65%** of *all* software [58]—and contains **nearly 20%** more security vulnerabilities [80]—*mitigating this high-value attack surface demands fast and effective automated security testing.*

Unfortunately, fuzzing has not yet caught up in closed-source use cases. Binary executables are significantly harder to analyze due to their opaque semantics, restricting closed-source software fuzzers to orders-of-magnitude slower techniques for obtaining the code coverage feedback that is critical to effective fuzzing; and worse yet, efforts to introduce compiler-based fuzzing enhancements to closed-source fuzzing contexts are overshadowed by the enormous overheads of current binary instrumenters, leaving vulnerability discovery on closed-source codebases no better than previous-generation techniques. Without resolving the *feedback* and *instrumentation* asymmetries that impede fuzzing in closed-source domains, the fuzzing community's many advancements will stay confined to the open-source ecosystem, leaving security practitioners powerless at fighting the critical, high-value vulnerabilities in closed-source commercial, proprietary, and legacy code that wreak havoc across cyberspace.

1.1 Problem Definition

My work in this thesis is motivated by the following four feedback and instrumentation asymmetries that uniquely hinder the effectiveness of closed-source software fuzzing:

Asymmetry 1: Low-overhead Coverage Guidance. The ability to retrieve runtime feedback of the program being fuzzed is critical to effective fuzzing. Most fuzzers

today adopt a *coverage-guided* search policy: monitoring each program execution to extract the code coverage of every generated test case, with the goal being to maximize exploration of the program by focusing mutation on only the subset of test cases which exercise previously-unexplored program functionality. While open-source fuzzing contexts have long benefited from low-overhead code coverage collection through the use of efficient, compiler-based instrumentation, compiler instrumentation *cannot* be applied to binary executables (i.e., pre-compiled software), **restricting closed-source software fuzzing to techniques incurring upwards of 10,000% more overhead**.

Asymmetry 2: Fine-grained Coverage Guidance. The success of code coverage as a feedback for guiding fuzzers' exploration has led it to become an essential feature of modern fuzzer design. Previous-generation fuzzers tracked code coverage at the granularity of *basic blocks*: the nodes of a program's control-flow graph representation. While basic block coverage benefits from the fastest coverage collection speed, it is no longer a desired coverage metric today. Most current fuzzers adopt *edge* coverage—the transitions between basic blocks—to capture interesting paths that would otherwise be missed by a basic-block-level coverage granularity; as well as *hit counts*—the execution frequencies of edges or basic blocks—to track progress made on exploring a program's loops or cycles. Unfortunately, tracking these finer-grained coverage metrics requires more invasive program monitoring than basic block coverage, and thus brings far higher coverage-collection overhead per program execution. While this overhead has little impact on open-source fuzzing due to its already-fast compiler instrumentation, **closed-source fuzzing's only options for retrieving fine-grained code coverage are burdened by orders-of-magnitude more overhead**.

Asymmetry 3: Low-overhead Program Transformation. Recent innovations in open-source software fuzzing make extensive use of compilers' code transformation abilities

to modify programs and make them *easier* or *faster* to fuzz. Yet, while many instrumentation tools for binary executables offer similar program transformation capabilities, their orders-of-magnitude higher baseline overheads smother any potential benefits of porting fuzzing-enhancing program transformations to closed-source fuzzing contexts. Thus, existing instrumenters' **inability to match compiler-quality transformation capabilities *and* compiler-level speed leaves fuzzing significantly less effective at uncovering vulnerabilities in closed-source software**.

Asymmetry 4: Scalable Program Transformation. Modern compiler toolchains such as GCC and LLVM support an ever-growing variety of software languages (e.g., C, C++), platforms (e.g., MacOS, Linux, Windows), and architectures (e.g., x86-64, ARM), making it easy to extend compiler-based fuzzing enhancements from one open-source ecosystem to another. Yet, because binary instrumenters are restricted to only *pre-compiled* binary executables, they face significant challenges in maintaining scalability to multiple software formats. For example, Linux and Windows are both x86 platforms, yet their binary executable formats—ELF and PE32, respectively—vary greatly in semantics (e.g., sections, code/data interleaving, etc.), making it non-trivial to port fuzzing-enhancing program transformations from Linux- to Windows-based closed-source software. Beyond this, many closed-source software developers adopt countermeasures to make third-party analysis of their software as difficult as possible (e.g., debug information stripping, code obfuscation, software anti-tamper), creating further roadblocks for closed-source software fuzzing enhancements. **Without scalable, robust, *and* format-agnostic binary instrumentation, effective fuzzing will only be possible on a *subset* of the closed-source software ecosystem.**

1.2 Contributions of This Thesis

Through the following contributions, my work empowers security practitioners to uncover vulnerabilities hidden in today's complex, opaque software more efficiently and effectively than ever before—enabling the outperforming of even current open-source fuzzing techniques—while operating purely from the semantically-limited granularity of binary-only code:

A. UnTracer and HeXcite (chapter 3): A suite of practical techniques to optimize fuzzing's core coverage feedback strategy, enabling closed-source fuzzing to attain *speed* and *code coverage* higher than coverage techniques for open-source software fuzzing.

B. Zafl (chapter 4): A binary instrumentation platform attaining compiler-quality code transformation *and* compiler-level instrumentation speed, enabling porting of *compiler-based fuzzing enhancements* to closed-source fuzzing contexts at minimal cost.

Chapter 2

Literature Review

This section provides an overview of the topics related to the contributions of this thesis: fuzzing, code coverage, and tracing performance.

2.1 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 [119, 146] and the software industry [17, 108]. 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 2.1).

Fuzzers generate test cases using one of two approaches: grammar-based [36, 46, 68, 85] or mutational [104, 107, 118, 119, 146]. 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,

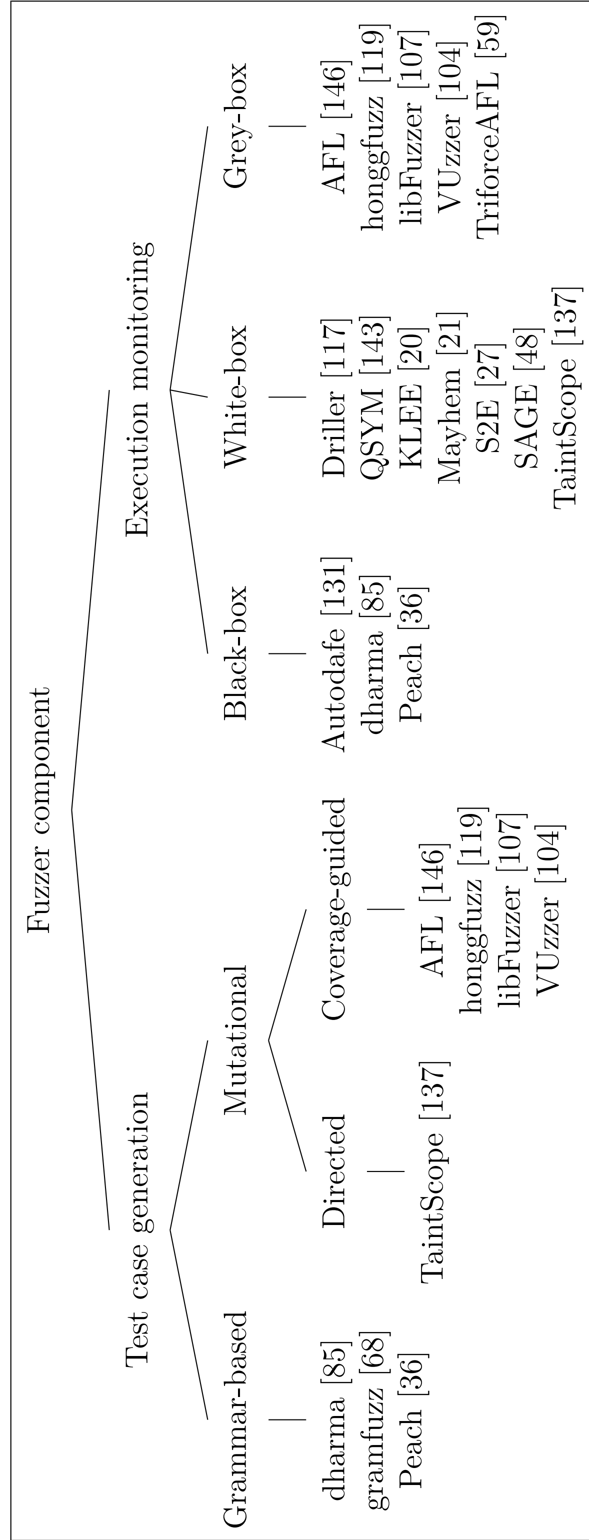


Figure 2.1: A taxonomy of popular fuzzers by test case generation and program analysis approaches.

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 [15, 44] 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 [104, 107, 119, 146] 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 [137] or patch testing [15], coverage-guided fuzzing's wider scope makes it more popular among the fuzzing community [107, 108, 119, 146]. 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 [36, 85, 131] only monitor input/output execution behavior (e.g., crashes). White-box fuzzers [20, 21, 27, 47, 48, 117, 137] use heavy-weight program analysis for fine-grained execution path monitoring and constraint solving. Grey-box fuzzers [14, 15, 59, 104, 107, 119, 146] are a trade-off 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 [104], Google's libFuzzer [107], honggfuzz [119], and AFL [146].

2.1.1 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 2.2 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:

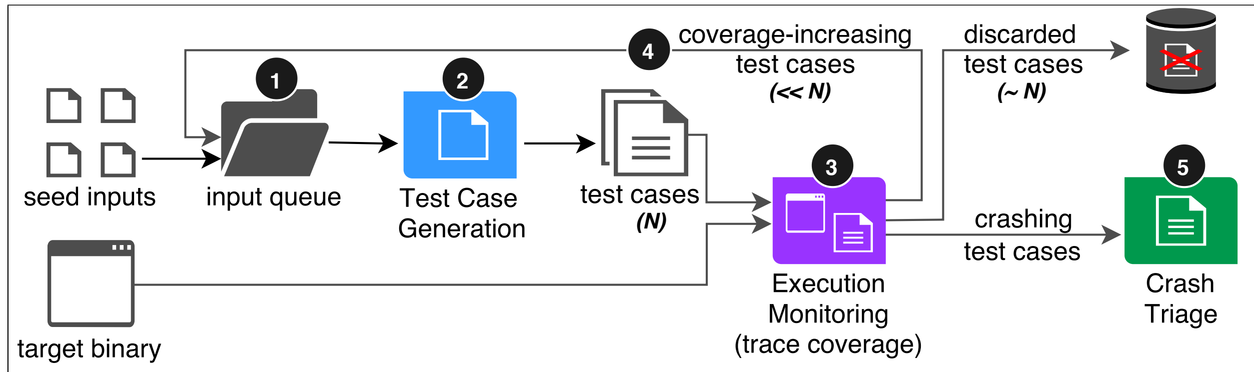


Figure 2.2: High-level architecture of a coverage-guided mutational fuzzer.

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 [107, 119, 146], system emulation [59, 105, 146], or hardware-assisted mechanisms [105, 119, 147]. 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.

[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.

2.2 Fuzzing's Code Coverage Metrics

In my efforts to understand fuzzing's current coverage landscape, I survey 27 of today's state-of-the-art coverage-guided fuzzers (Table 2.1) and identify three universal coverage metrics: *basic blocks*, *edges*, and *hit counts*. I discuss these coverage metrics in detail below.

Name	**Cov**	**Hit**	**Name**	**Cov**	**Hit**	**Name**	**Cov**	**Hit**
AFL [146]	➤	✔	EnFuzz [26]	➤	✔	ProFuzzer [142]	➤	✔
AFL++ [41]	➤	✔	FairFuzz [77]	➤	✔	QSYM [143]	➤	✔
AFLFast [14]	➤	✔	honggFuzz [119]	➤	✗	REDQUEEN [7]	➤	✔
AFLSmart [99]	➤	✔	GRIMORE [13]	➤	✔	SAVIOR [25]	➤	✔
Angora [22]	➤	✔	lafIntel [1]	➤	✔	SLF [141]	➤	✔
CollAFL [43]	➤	✔	libFuzzer [107]	➤	✔	Steelix [79]	➤	✔
DigFuzz [148]	➤	✔	Matryoshka [23]	➤	✔	Superion [133]	➤	✔
Driller [117]	➤	✔	MOpt [82]	➤	✔	TIFF [67]	■	✔
Eclipser [28]	➤	✔	NEUZZ [110]	➤	✔	VUzzer [104]	■	✔

Table 2.1: 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 [103] or bitmap [146]). Two modern fuzzers that employ basic block coverage are VUzzer [104] and its successor TIFF [67].

Edge Coverage: Edges refer to block-to-block transitions, and offer a finer-grained approximation of paths taken. As Table 2.1 shows, most fuzzers rely on edge coverage; AFL [146] and its many derivatives [41] record edges as hashes of their start/end block tuples in a bitmap data structure; while LLVM SanitizerCoverage-based [123] fuzzers honggFuzz and

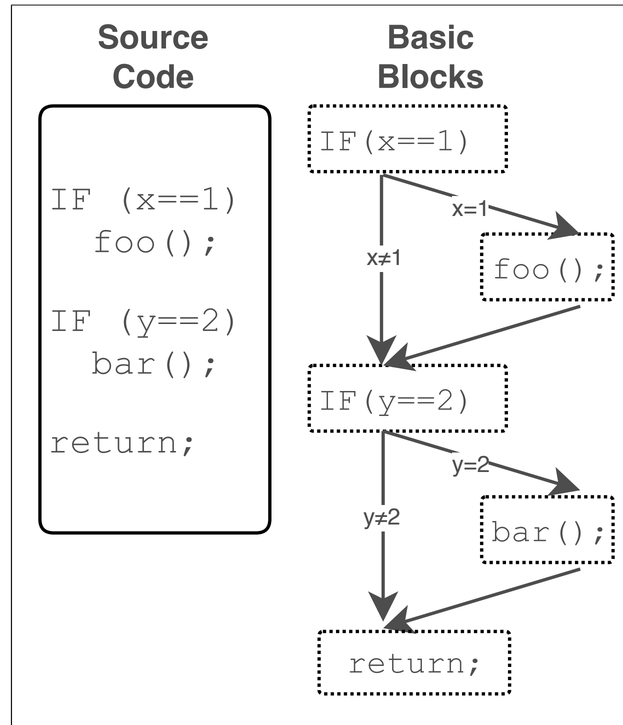


Figure 2.3: An example of basic blocks in C code.

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 over-saturating the fuzzer seed pool with needless test cases (e.g., one per new loop iteration), and is hence seldom used.

2.3 Coverage Tracing Performance

Contingent on the ability to instrument a target program from source, fuzzing is divided into two distinct worlds: *compiler-based* and *binary-only*. Most modern fuzzers turn to compiler instrumentation as its low runtime overhead supports high fuzzing throughput. AFL accomplishes this through custom GCC and Clang wrappers. honggfuzz and libFuzzer also provide their own Clang wrappers. More recent state-of-the-art efforts also leverage compilers' ability to apply complex program transformations. Researchers have shown that such transformations improve fuzzing effectiveness by enhancing performance [43, 65] or introspection [1, 22, 41, 70].

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 [104] uses PIN [81] to dynamically (during run-time) instrument black-box binaries. AFL's QEMU user-mode emulation also instruments dynamically, but as my experiments show (section 3.5), 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 [120]. While previous work shows AFL-Dyninst significantly outperforms AFL-QEMU on select binaries [93], results in section 3.5 suggest that the performance gap is much narrower.

Chapter 3

Fast and Fine-grained Coverage for Closed-source Software Fuzzing

3.1 Introduction

Software vulnerabilities remain one of the most significant threats facing computer and information security [94]. 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 [17, 108, 119, 146].

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 [14, 79, 104, 107, 119, 146], 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 [107, 119, 146]. For black-box (source-unavailable) binaries, it is generally measured through instrumentation inserted dynamically [104, 146] or statically through binary rewriting [120], or through instrumentation-free hardware-assisted tracing [105, 119, 147].

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 my results in section 3.5 show, AFL [146]—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 3.2 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 my 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.

3.1.1 Coverage-guided Tracing

This work 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. I 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 my experiments), for the ability to run all test cases (initially) at native speed. To validate Coverage-guided Tracing and explore its tradeoffs on real-world software, I implement UnTracer. UnTracer leverages the black-box binary instrumenter Dyninst [96] to construct the interest oracle and tracing infrastructure.

I evaluate UnTracer alongside several coverage tracers used with the popular fuzzer AFL [146]. For tracing black-box binaries, I compare against the dynamic binary rewriter AFL-QEMU [146], and the static binary rewriter AFL-Dyninst [96]. For tracing white-box binaries, I compare against AFL-Clang [146]. To capture a variety of target binary and tracer behaviors, I 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, I perform evaluations for a 24-hour period and use 5 test case datasets per benchmark to expose the effects of randomness. My 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. I further integrate UnTracer with the state-of-the-art hybrid fuzzer QSYM [143]. Results show that QSYM-UnTracer averages 79% and 616% more executed test cases than QSYM-Clang and QSYM-QEMU, respectively.

In summary, this work makes the following contributions:

- I introduce Coverage-guided Tracing: an approach for reducing fuzzing overhead by restricting tracing to coverage-increasing test cases.
- I quantify the infrequency of coverage-increasing test cases across eight real-world applications.
- I 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.
- I implement and evaluate UnTracer; UnTracer is my coverage-guided tracer based on the Dyninst black-box binary instrumenter. I 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.
- I integrate UnTracer with the state-of-the-art hybrid fuzzer QSYM, and show that QSYM-UnTracer outperforms QSYM-Clang and QSYM-QEMU.
- I open-source my evaluation benchmarks [88], experimental infrastructure [87], and an AFL-based implementation of UnTracer [90].

3.1.2 Coverage-*preserving* Coverage-guided Tracing

I further tackle the challenge of extending CGT to fuzzing's most ubiquitous coverage metrics—edges and hit counts—making *high-performance, fine-grained* code coverage tracing available for all existing (and future) fuzzers. At the core of my efforts are binary-level

and fuzzing enhancements that broaden CGT's coverage while maintaining its orders-of-magnitude speedup: for edge coverage, I 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, I 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 [123]). To extend CGT to hit count coverage, I 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 [41, 146].

I implement my *coverage-preserving* Coverage-guided Tracing, **HeXcite**, and evaluate it against the current block-coverage-only CGT implementation UnTracer [89]; the leading binary-only fuzzing coverage tracers QEMU [146], Dyninst [61], and RetroWrite [32]; and the popular source-level coverage tracing via AFL-Clang [146]. 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 work enables the use of the fastest tracing approach in fuzzing—*Coverage-guided Tracing*—by the majority of today's fuzzers:

- I 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.

- I 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.

- I demonstrate that with these techniques, my *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.

- I 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.

- I open-source HEXCITE, my implementation of binary-only coverage-preserving CGT, and my evaluation benchmarks at: `https://github.com/FoRTE-Research/HeXcite`.

3.2 Impact of Discarded Test Cases

Traditional coverage-guided fuzzers (e.g., AFL [146], libFuzzer [107], and honggfuzz [119]) 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 [117], program state [79], and taint tracking [22, 104]) 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, I 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) [146] and Driller (smart test case generation) [117]. I 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, I highlight the most relevant implementation details of both fuzzers regarding test case generation and coverage tracing, and my experimental setup.

AFL

AFL [146] is a "blind" fuzzer as it relies on random mutation to produce coverage-increasing (coverage-increasing) test cases, which are then used during mutation.[1] AFL traces test case coverage using either QEMU-based dynamic instrumentation for black-box binaries or assembly/compile-time instrumentation for white-box binaries. I cover both options by evaluating AFL-QEMU and AFL-Clang.

Driller

Driller [117] 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. I evaluate Driller-AFL (aka ShellPhuzz [111]). Like AFL, Driller-AFL also

[1] 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 [146].

utilizes QEMU for black-box binary coverage tracing.

3.2.1 Experimental Setup

For AFL-Clang and AFL-QEMU I use the eight benchmarks from my evaluation in section 3.5. As Driller currently only supports benchmarks from the DARPA Cyber Grand Challenge (CGC) [30], I evaluate Driller-AFL on eight pre-compiled [112] CGC binaries. I run all experiments on the same setup as my performance evaluation (section 3.5).

To measure each fuzzer's execution/tracing time, I insert timing code in AFL's test case execution function (`run_target()`). As timing occurs per-execution, this allows me to also log the total number of test cases generated. I 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.

3.2.2 Results

As shown in Table 3.1, 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 3.2 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.*

	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 3.1: 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). I 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 3.2: 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).

3.3 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 3.2, I 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.

3.3.1 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 3.1, 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

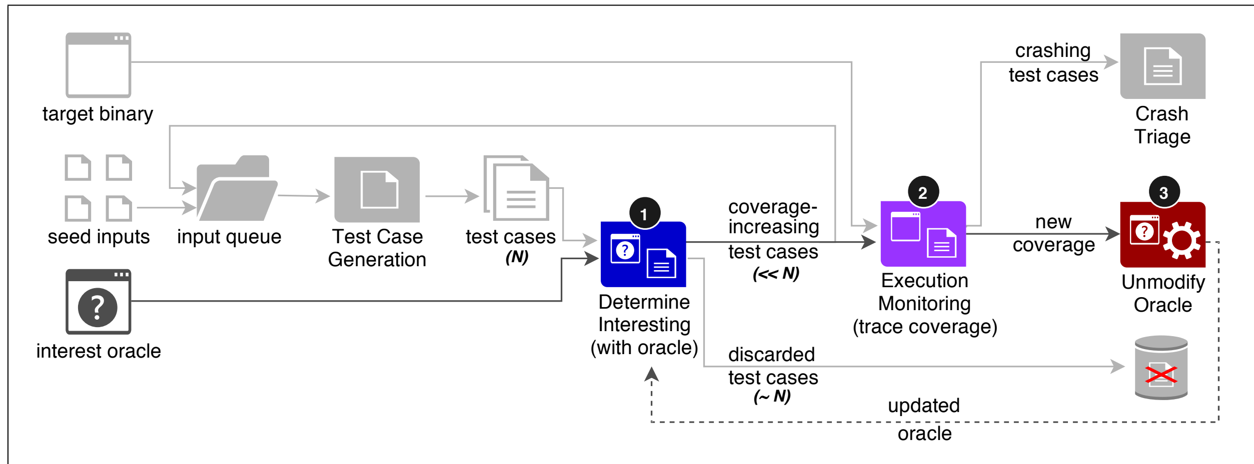


Figure 3.1: Visualization of how Coverage-guided Tracing augments the workflow of a conventional coverage-guided grey-box fuzzer (e.g., AFL [146]). Coverage-guided Tracing can also be similarly adapted into coverage-guided white-box fuzzers (e.g., Driller [117]).

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.

3.3.2 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 [71, 74, 125], and tools like angr [113] and Dyninst [96] 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).

3.3.3 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 [104], or—if all critical edges are mitigated—edge coverage [107, 119]. Thus, Coverage-guided Tracing is compatible with most existing tracing approaches.

3.3.4 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.

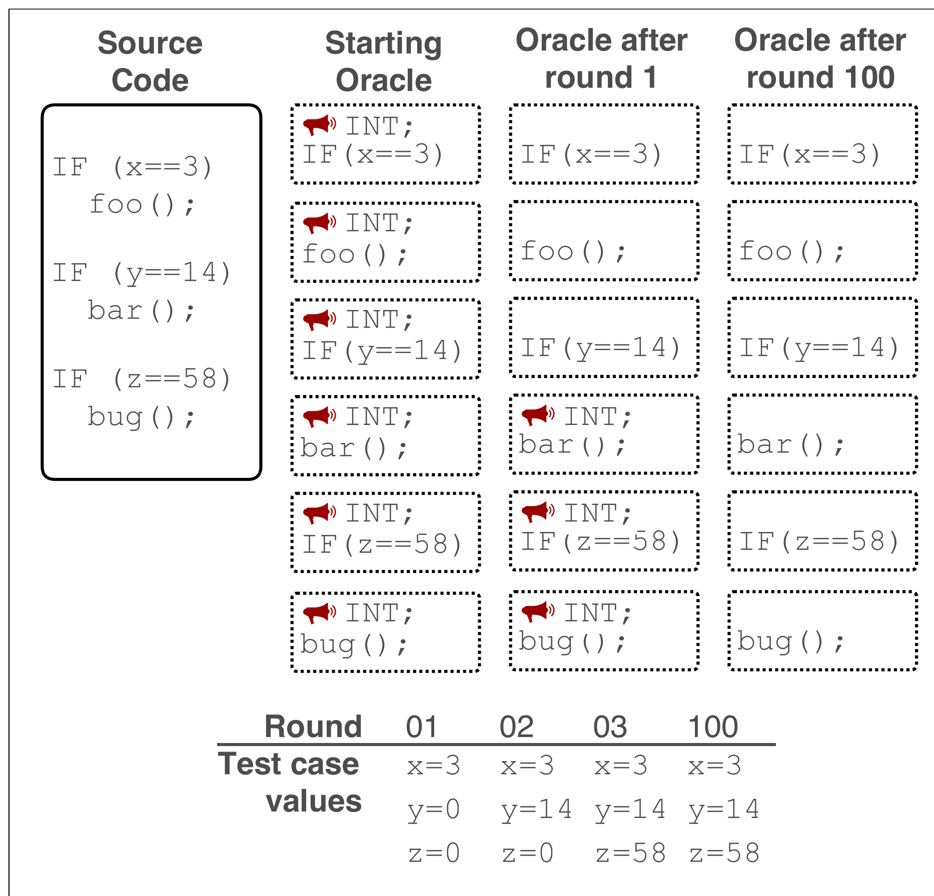


Figure 3.2: An example of the expected evolution of a Coverage-guided Tracing interest oracle's basic blocks alongside its original source code. Here, `INT` denotes an oracle interrupt. For simplicity, this diagram depicts interrupts as inserted; however, in Coverage-guided Tracing, the interrupts instead overwrite the start of each basic block. Unmodifying basic blocks consists of resetting their interrupt-overwritten byte(s) to their original values.

3.3.5 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 3.2)—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.

3.4 Implementation: UnTracer

Here I introduce UnTracer, my implementation of Coverage-guided Tracing. Below, I offer an overview of UnTracer's algorithm and discuss its core components in detail.

3.4.1 UnTracer Overview

UnTracer is built atop a modified version of the coverage-guided grey-box fuzzer, AFL 2.52b [146], which I selected due to both its popularity in the fuzzing literature [14, 15, 59, 78, 79, 98, 117, 132] and its open-source availability. My implementation consists of 1200 lines of C and C++ 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 [144], I 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),

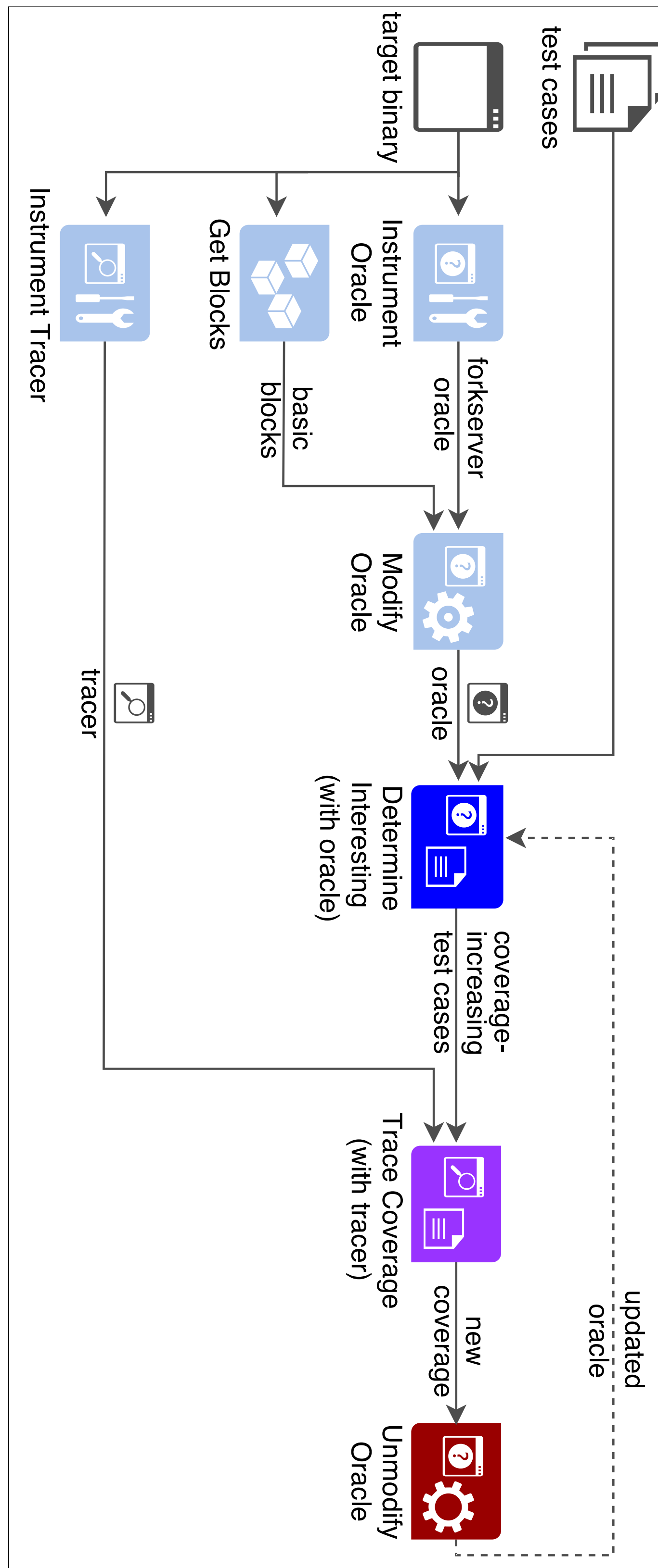


Figure 3.3: UNTRACER's workflow. Not shown is test case generation, or starting/stopping forkservers.

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)
11 // Main fuzzing loop
12 while 1 do
13 |  i ← AFL_WriteTotest case()
14 |  if P_O(i) → INTERRUPT then
   |  |  // The test case is coverage-increasing
15 |  |  Φ.add(i)
   |  |  // Trace test case's new coverage
16 |  |  B_trace ← getTrace(P_T(i))
   |  |  // Kill oracle before unmodifying
17 |  |  stopForkserver(P_O)
   |  |  // Unmodify test case's new coverage
18 |  |  for b ∈ B_trace do
19 |  |  |  unmodifyOracle(b)
20 |  |  end
   |  |  // Restart oracle before continuing
21 |  |  startForkserver(P_O)
22 |  |  AFL_HandleCoverageIncreasing()
23 |  end
24 end
```

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). I 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, I 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 3.3 depicts UnTracer's architecture.

3.4.2 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 coverage. In implementing UnTracer, I aim to optimize execution speeds of both binaries. Like other AFL tracers, UnTracer incorporates a *forkserver* execution model [144] 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, I already use Dyninst's static binary rewriting for black-box binary instrumentation, so I use that same technique for the forkserver.

For the oracle binary, my initial approach was to instrument it using Dyninst. Unfortunately, preliminary evaluations revealed several performance problems.[2] Since the oracle executes

[2] I made Dyninst developers aware of several performance issues—specifically, excessive function calls (e.g.,

every test case, it is performance critical. To avoid Dyninst's limitations, I 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.

3.4.3 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 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. I 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, I do not consider functions executed prior to the forkserver callback in `<main>` (e.g., `<_start>, <_libc_start_main>, <_init>, <frame_dummy>`).

I use `SIGTRAP` for my interrupt for two reasons: (1) it has long been used for fine-grain execution control and analysis (e.g., `gdb` [18, 116] and kernel-probing [63, 72]); and (2) its binary representation—`0xCC`—is one byte long, making it possible to overwrite basic blocks of all sizes.

3.4.4 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

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.

version of the target binary. I 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, I 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, I optimize tracing to only record *uniquely*-covered basic blocks as follows: in the tracer forkserver, I 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.

3.4.5 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.

I 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, I 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.

3.5 Tracing-only Evaluation

My evaluation compares UnTracer against tracing all test cases with three widely used white- and black-box binary fuzzing tracing approaches—AFL-Clang (white-box) [146], AFL-QEMU (black-box dynamically-instrumented) [146], and AFL-Dyninst (black-box statically-instrumented) [120] on eight real-world benchmarks of different type.

My 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?

3.5.1 Evaluation Overview

I 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 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. My evaluations examine each tracer's overhead on eight real-world, open-source benchmarks of different type, common to the fuzzing community. Table 3.3 provides benchmark details. To smooth the effects of randomness and ensure the most fair comparison of performance, I 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 my results show UNTRACER has less than 1% overhead after one hour of fuzzing, I extend all evaluations to 24 hours to better match previous fuzzing evaluations.

I configure AFL to run with 500ms timeouts and leave all other parameters at their defaults. I 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, I perform eight trials of each dataset. For each set of trials per dataset, I apply trimmed-mean de-noising [6] 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. I allocate each virtual machine 6GB of RAM.[3]

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

3.5.2 Experiment Infrastructure

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

My second binary—`TestTrace`—forms the backbone of my evaluation infrastructure. I 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, I include both the initial full-speed execution and any time spent handling coverage-increasing test cases.

3.5.3 Benchmarks

My benchmark selection is based on popularity in the fuzzing community and benchmark type. I first identify candidate benchmarks from several popular fuzzers' trophy cases[4] and public benchmark repositories [50, 102, 108, 119, 146]. To maximize benchmark variety, I 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 I filter out several candidate benchmarks based on incompat-

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

Package	Benchmark	Version	Class	Basic Blocks	Test Cases ($\cdot 10^6$)	Coverage-increasing Ratio	500ms Time-outs
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 3.3: Information on the eight benchmarks used in my evaluation in section 3.5 and section 3.6 and averages over 5 24-hour datasets for each benchmark.

ibility with my tracers (e.g., Dyninst-based instrumentation crashes on `openssl`), I 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, I 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 [96]. For counting timeouts, I examined the statistics reported by `afl-fuzz-saveinputs` during dataset generation; using my specified timeout value (500ms), I then averaged the number of timeouts per benchmark among its datasets. Lastly, for each benchmark, I counted and averaged the number of test cases generated in all of its 24-hour datasets.

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

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 [144]. As each fuzzing tracer in my evaluation leverages forkserver-based execution, I design my "ground-truth" benchmark execution models to represent the fastest known execution speeds: a statically-instrumented forkserver without any coverage tracing. I use a modified copy of AFL's assembler (`afl-as`) to instrument *baseline* (forkserver-only) benchmark versions. In each benchmark trial, I use its baseline execution speeds as the basis for comparing each fuzzing tracers' overhead.

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

AFL-QEMU: I only need to provide it the original uninstrumented target binary of each benchmark in my evaluation.

AFL-Dyninst: For my AFL-Dyninst evaluations, I instrument each binary using AFL-Dyninst's instrumenter with configuration parameters:

- `bpatch.setDelayedParsing` set to True;
- `bpatch.setLivenessAnalysis` set to False; and
- `bpatch.setMergeTramp` set to False.

I leave all other configuration parameters at their default settings.

3.5.4 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), I observed that, for some datasets, my 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, I 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. I 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).

3.5.5 UnTracer versus Coverage-agnostic Tracing

I examine my evaluation results to identify each fuzzing tracer's overhead per benchmark. For each tracer's set of trials per benchmark dataset, I employ trimmed-mean de-noising (shown to better reveal median tendency [6]) at test case level—removing the top and bottom 33% outliers—to reduce impact of system interference on execution speeds. I then take the resulting five trimmed-mean dataset overheads for each tracer-benchmark combination and average them to obtain tracer-benchmark overheads. Lastly, I 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, I compare the performance of UnTracer to three popular *coverage-agnostic* tracing approaches. I first explore the performance of two black-box binary fuzzing tracers: AFL-QEMU (dynamic) and AFL-Dyninst (static). Secondly, I compare UnTracer's performance against that of the white-box binary fuzzing tracer AFL-Clang (static assembler-instrumented tracing).

Black-box binary tracing

As shown in Figure 3.4, I compare UNTRACER's performance to two popular black-box binary fuzzing tracers—AFL's dynamically-instrumented tracing via QEMU user-mode emulation (AFL-QEMU) [145], and Dyninst-based static binary rewriting-instrumented tracing (AFL-Dyninst) [120]. 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`, I observe a clear trend between higher number of timeouts and lower tracing overheads across all tracers (Table 3.3). In my 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.), I am unable to identify which benchmark characteristics are optimal for AFL-Dyninst's performance. Across all benchmarks, UNTRACER achieves an average 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, my 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 [75] recommendation, I 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 I compute all per-dataset execution times for each benchmark[5] and tracer combination; then for each benchmark dataset I apply

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

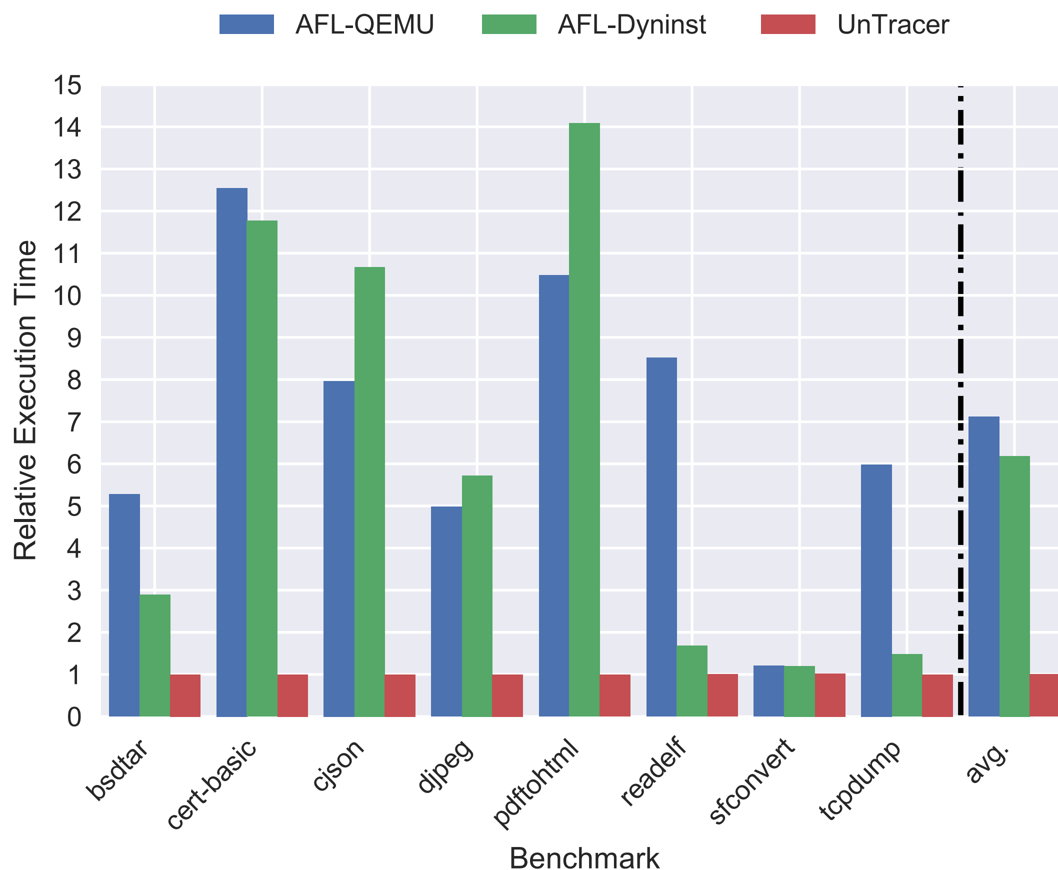


Figure 3.4: Per-benchmark relative overheads of UnTracer versus black-box binary tracers AFL-QEMU and AFL-Dyninst.

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, I 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, I apply Vargha and Delaney's $\hat{A}_{12}$

statistical test [129]. 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 I conclude that the difference in execution times between UnTracer versus either black-box tracer is statistically large.

White-box binary tracing

In Figure 3.5, I 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 I 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, I conclude UnTracer's execution time convincingly outperforms AFL-Clang's.

Figure 3.6 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.

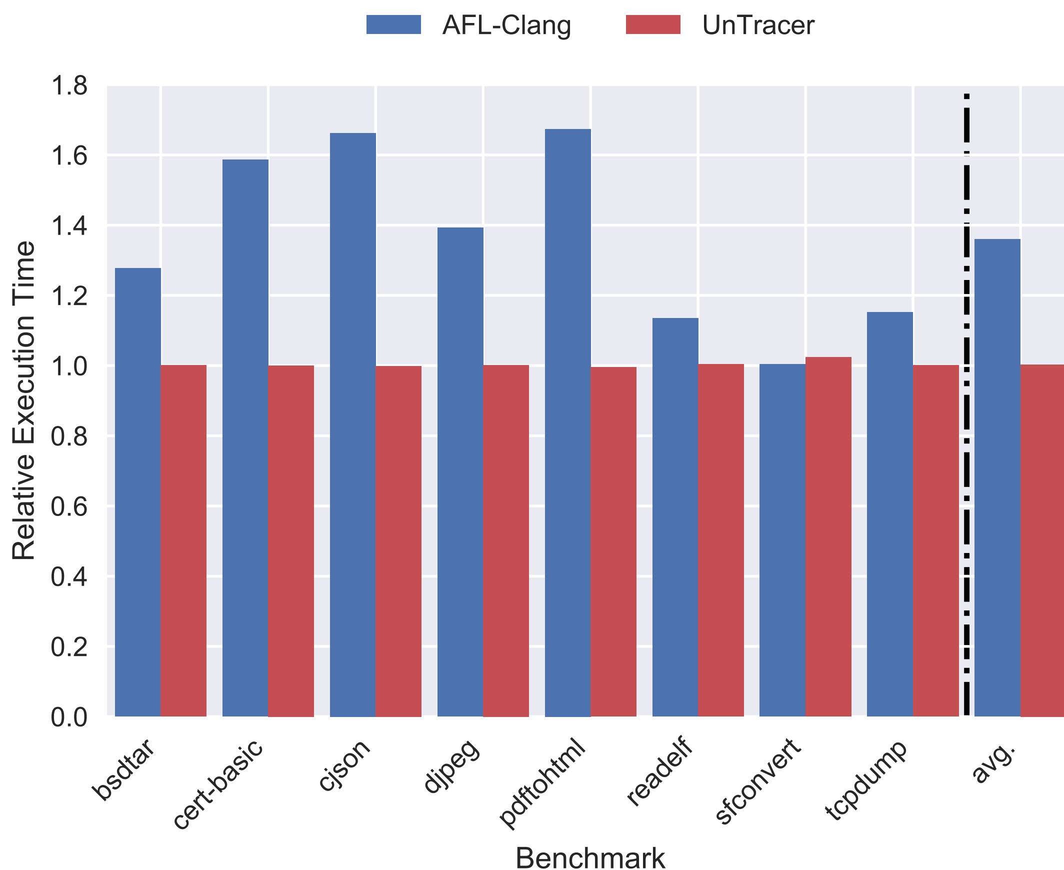


Figure 3.5: Per-benchmark relative overheads of **UnTracer** versus white-box binary tracer AFL-Clang.

Figure 3.7 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.

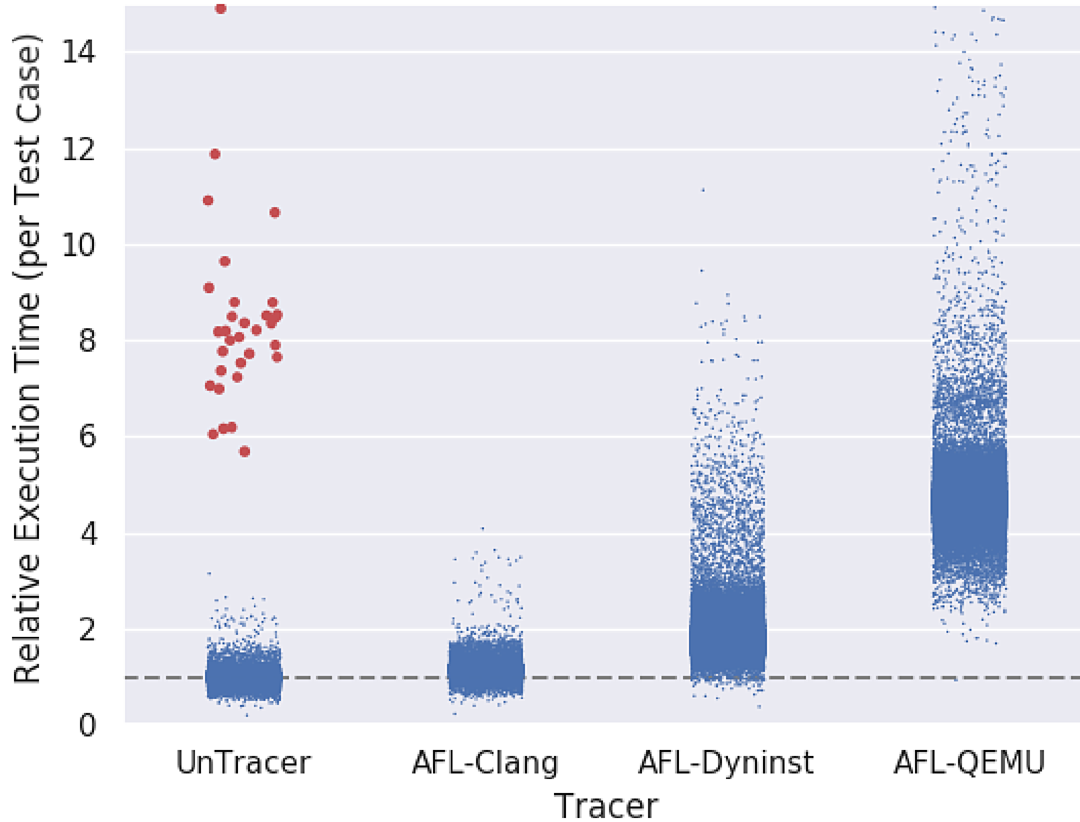


Figure 3.6: Distribution of each tracer's relative execution time averaged per-test case for one 24-hour `cjson` dataset. The horizontal grey dashed line represents the average baseline execution speed. Red dots represent coverage-increasing test cases identified by UNTRACER.

3.5.6 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 test cases' code coverage by executing them on the tracer;

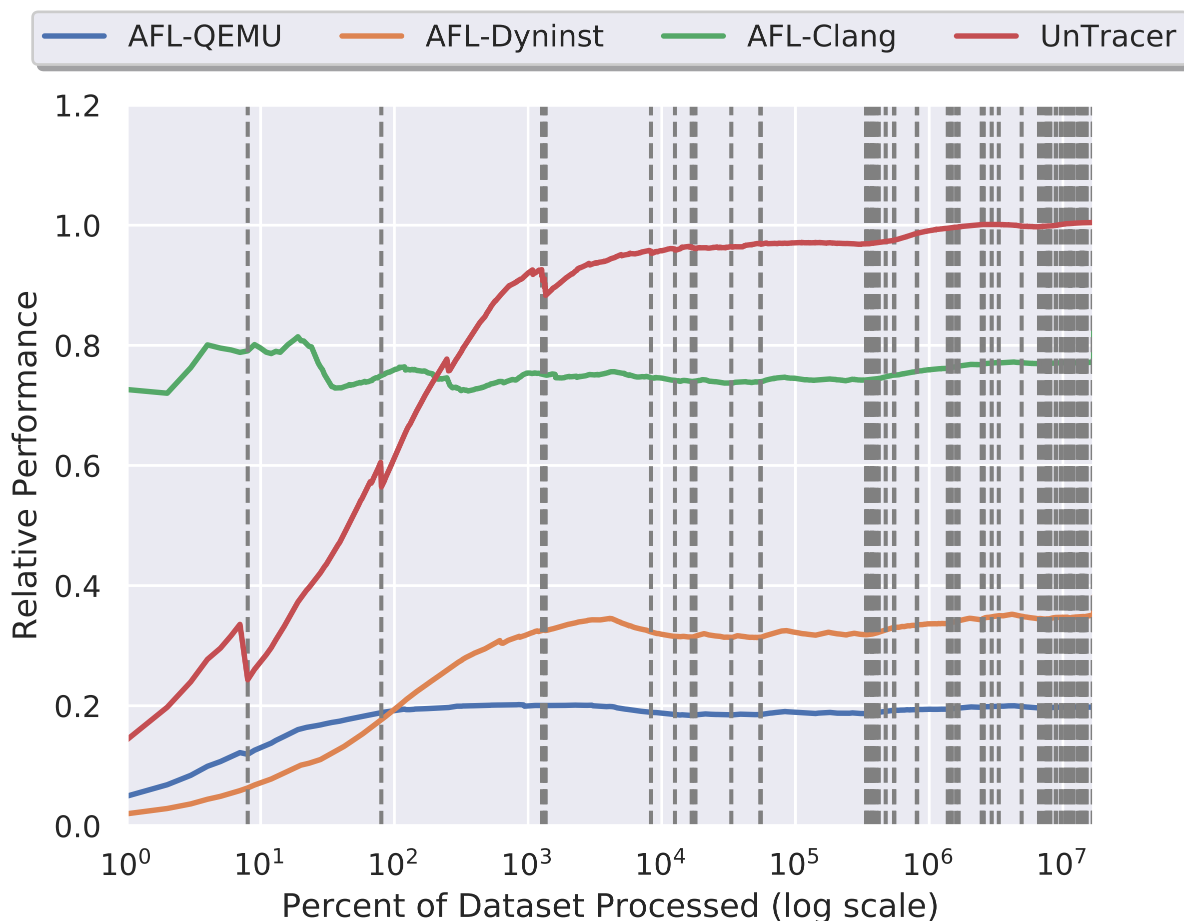


Figure 3.7: Averaged relative performance of all tracers over the percentage of test cases processed for one 24-hour `bsdtar` dataset. Here, 1.0 refers to baseline (maximum) performance. Each grey dashed vertical line represents a coverage-increasing test case.

(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 my evaluation of UnTracer's overhead, I 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`). I average all components' measured execution times across all coverage-increasing test cases, and calculate their respective proportions of UnTracer's total overhead. Figure 3.8 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.

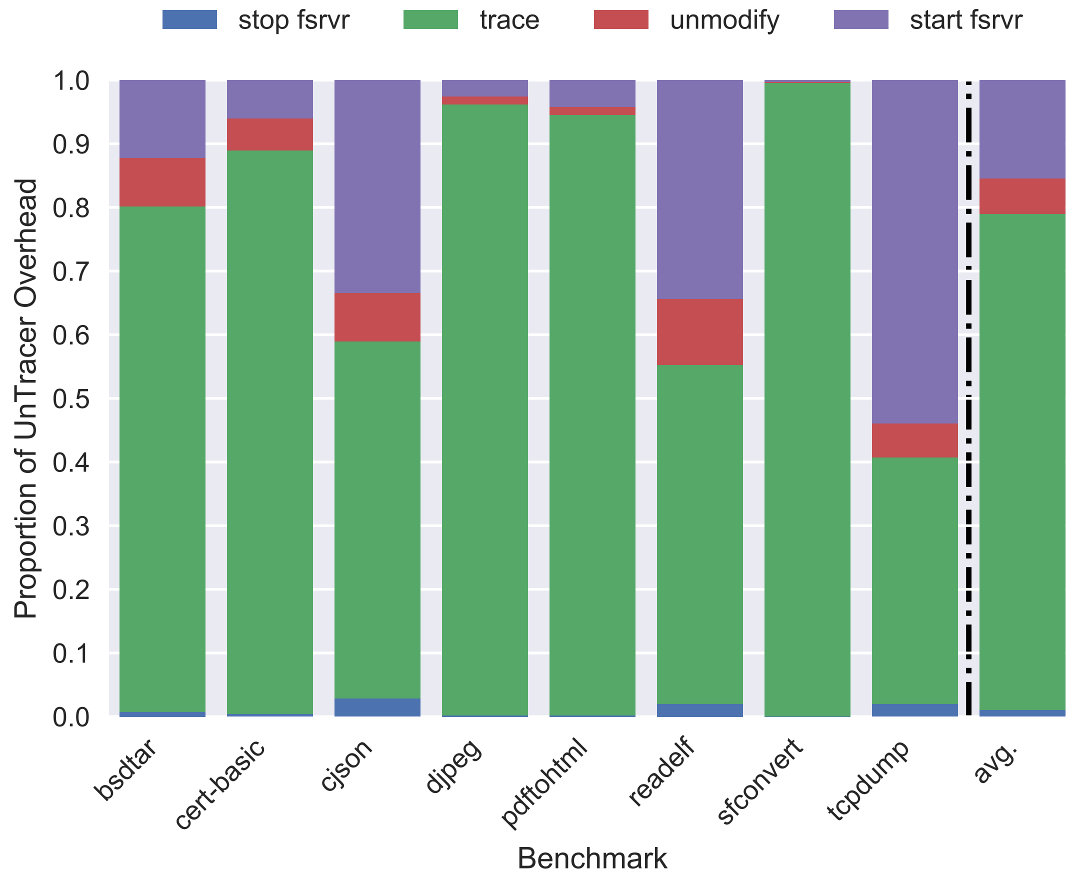


Figure 3.8: Visualization of the overheads per UnTracer's four components related to coverage-increasing test case processing for each benchmark.

Tracing: Unsurprisingly, coverage tracing (`trace`) contributes to the almost 80% of Un-

Tracer's overhead across all benchmarks. My implementation relies on Dyninst-based static binary rewriting-instrumented black-box binary tracing. As my evaluation results (Figure 3.4) 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 [139], but given UnTracer's low overhead in my evaluation, further optimization adds little performance improvement. However, I can imagine niche contexts where such approaches would yield meaningful performance improvements.

3.5.7 Overhead versus Rate of Coverage-increasing Test Cases

Below, I discuss the potential performance advantage of a hybrid approach combining coverage-guided and coverage-agnostic tracing (e.g., AFL [146], libFuzzer [107], honggFuzz [119]). 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 3.9). 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 my evaluations (Table 3.3), 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 [22, 79, 104]), my approach may yield less benefit.

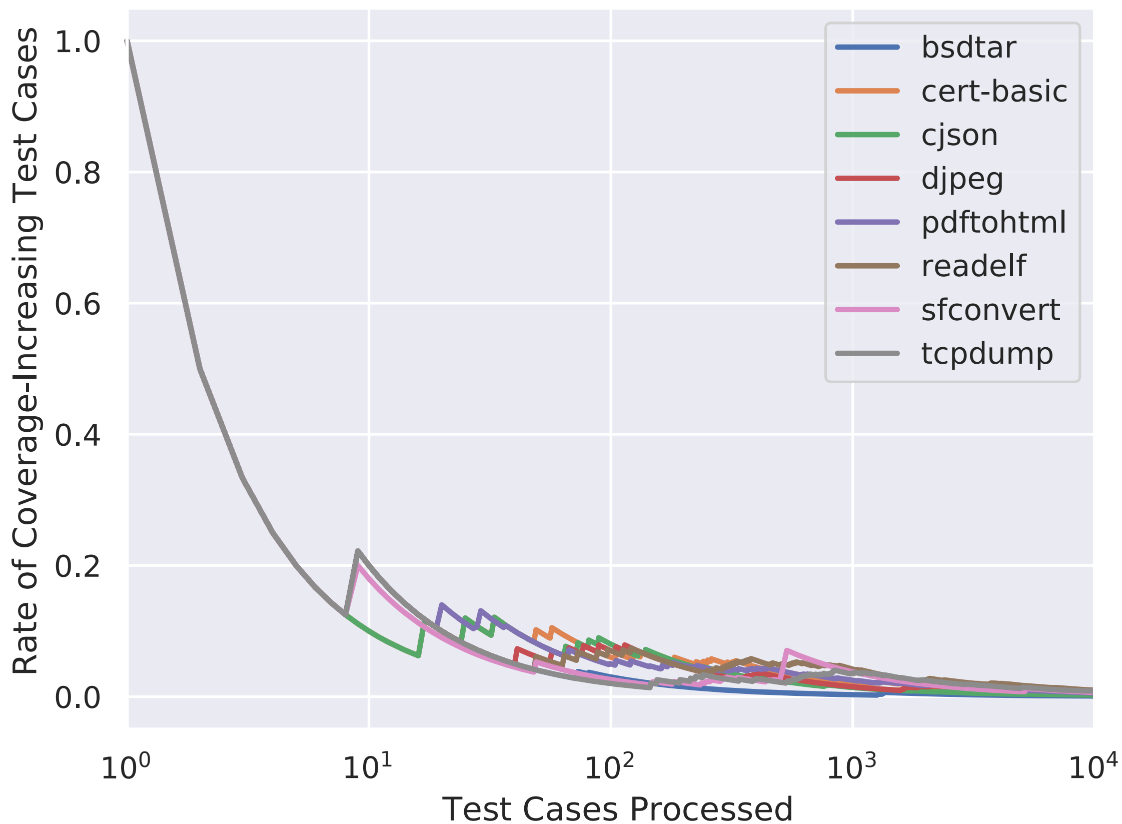


Figure 3.9: The rates of coverage-increasing test cases encountered over the total number of test cases processed, per benchmark.

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. I 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.

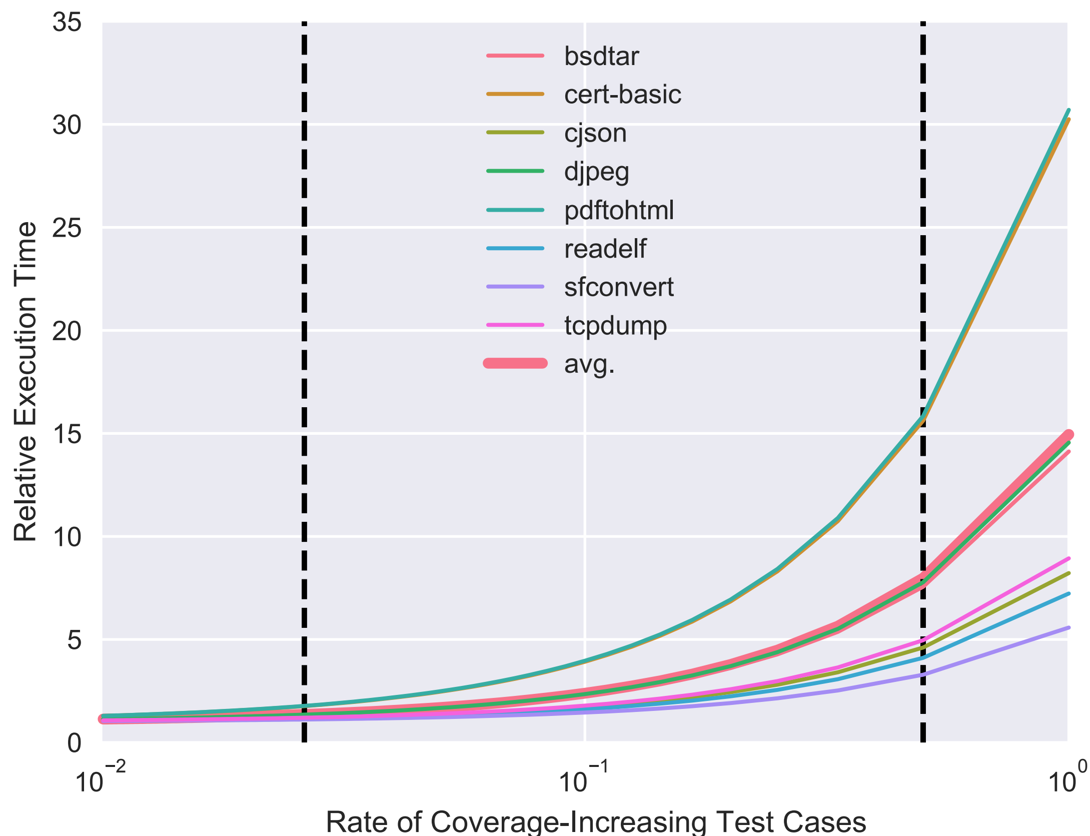


Figure 3.10: Model of the relationship between coverage-increasing test case rate and UnTracer's overhead per test case. For all rates left of the leftmost dashed vertical line, UnTracer's overhead per test case is less than AFL-Clang's. Likewise, for all rates left of the rightmost dashed vertical line, it is less than AFL-QEMU's and AFL-Dyninst's. Not shown is the average rate of coverage-increasing test cases observed during my evaluations (4.92E-3).

To develop a universally-applicable threshold for all tracing approaches, I average the overheads of coverage-increasing test cases across all trials in my tracer-benchmark evaluations. I then model overhead as a function of the rate of coverage-increasing test cases; I 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 3.10, 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).

3.6 Hybrid Fuzzing Evaluation

State-of-the-art hybrid fuzzers (e.g., Driller [117] and QSYM [143]) combine program-directed mutation (e.g., via concolic execution) with traditional blind mutation (e.g., AFL [146]). Hybrid approaches offer significant gains in code coverage at the cost of reduced test case execution rate. In this section, I compare UnTracer, Clang [146] (white-box tracing), and QEMU [146] (black-box dynamically-instrumented tracing) implementations of the state-of-the-art hybrid fuzzer QSYM on seven of my eight benchmarks.[6] 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 I provide an estimate of the impact hybrid fuzzing has on Coverage-guided Tracing's value in section 3.2, this section provides concrete data on the impact to UnTracer of a recent hybrid fuzzer.

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

Implementing QSYM-UnTracer

I implemented [90] 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 I briefly discuss design choices specific to each.

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 I 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 3.3)), I configure UnTracer-AFL, AFL-Clang, and AFL-QEMU to only track *total* timeouts.

For all other coverage contexts I implement the UnTracer interest oracle and tracer execution model as described in section 3.4.

3.6.1 Evaluation Overview

To identify the performance impact from using UnTracer in hybrid fuzzing I incorporate it in the state-of-the-art hybrid fuzzer QSYM and evaluate its against existing Clang- [146] and QEMU-based [146] QSYM implementations. My experiments compare the number of test cases executed for all three hybrid fuzzer variants for seven of the eight benchmarks

from section 3.5 (Table 3.3) with 100ms timeouts. To account for randomness, I 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, I compute a per-variant geometric mean.

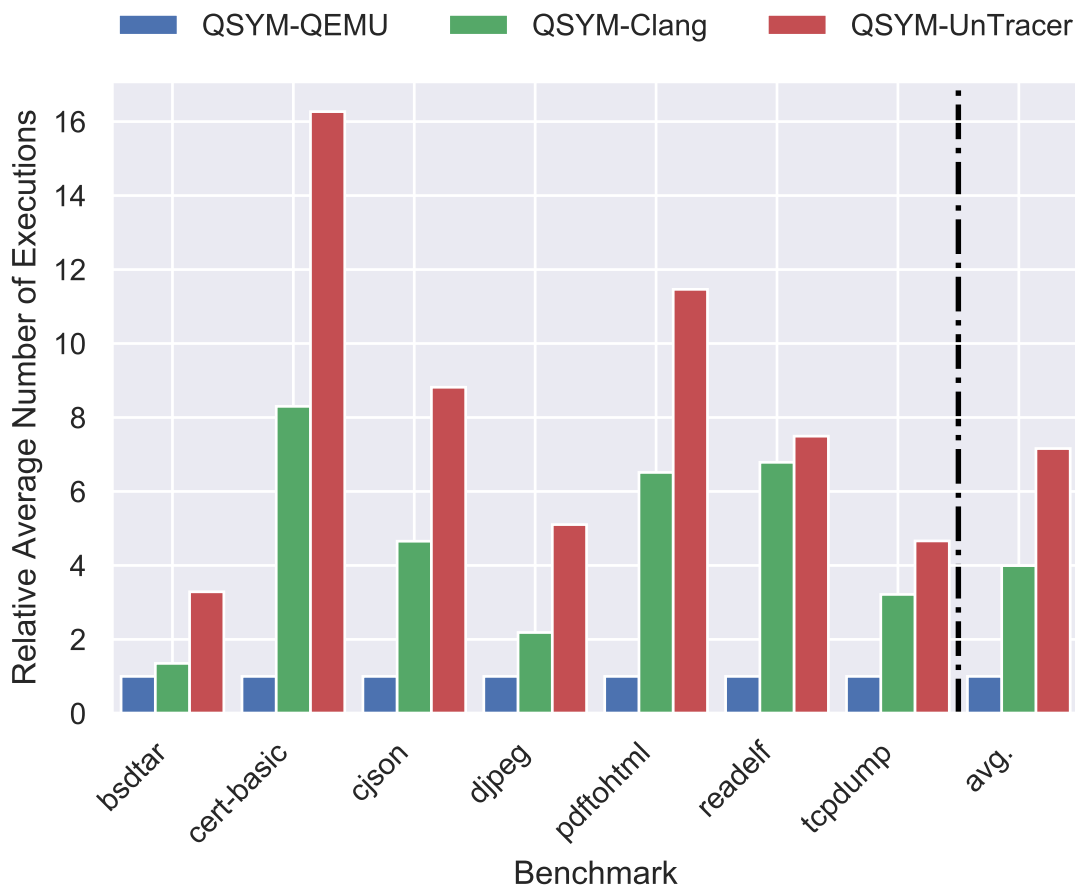


Figure 3.11: Per-benchmark relative average executions in 24 hours of QSYM-UnTracer versus QSYM-QEMU and QSYM-Clang.

I 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 3.11 presents the results

for each benchmark and the geometric mean across all benchmarks scaled to my baseline of the number of test cases executed by QSYM-QEMU.

3.6.2 Performance of UnTracer-based Hybrid Fuzzing

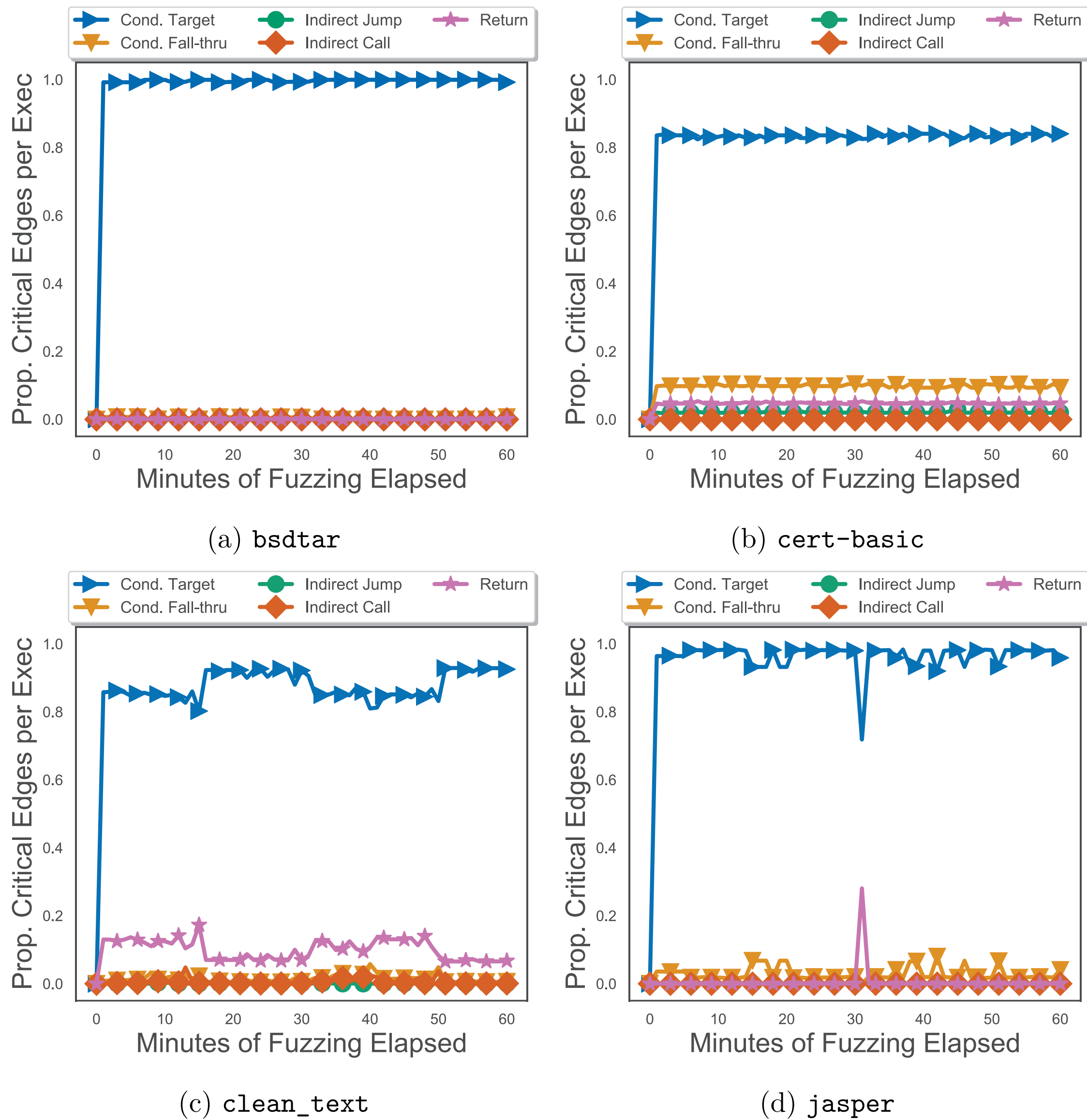
As shown in Figure 3.11, on average, QSYM-UnTracer achieves 616% and 79% more test case executions than QSYM-QEMU and QSYM-Clang, respectively. A potential problem I considered was the overhead resulting from excessive test case trimming and calibration. Since my implementation of QSYM-UnTracer defaults to the slow tracer binary for test case trimming and calibration, an initial problem I considered was the potential overhead resulting from either operation. However, my 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.

3.7 A Coverage-*preserving* Coverage-guided Tracing

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 I surveyed in Table 2.1: 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.

To address this incompatibility, I 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 I discuss the inner workings of jump mistargeting and bucketed unrolling, and the underlying insights and observations that motivate them.



(a) `bsdtar` (b) `cert-basic`

(c) `clean_text` (d) `jasper`

Figure 3.12: Visualization of the proportion of critical edges by transfer type encountered throughout fuzzing.

3.7.1 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 $\overrightarrow{\texttt{ab}}$ and $\overrightarrow{\texttt{bc}}$, covering blocks `a` and `b` implies covering edge $\overrightarrow{\texttt{ab}}$; and subsequently covering `c` implies $\overrightarrow{\texttt{bc}}$. However, such *block-centric* edge coverage does not suffice if there exists a third edge $\overrightarrow{\texttt{ac}}$. In this case, covering blocks `a`, `b`, and `c` implies edges $\overrightarrow{\texttt{ab}}$ and $\overrightarrow{\texttt{bc}}$; but since `c` has *already* been covered, there is no way to detect $\overrightarrow{\texttt{ac}}$. Formally, these problematic edges are referred to as *critical edges*: edges whose start/end blocks have two or more incoming/outgoing edges, respectively [123].

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

Diving deeper into critical edges: Supporting block-centric edge coverage requires resolving all critical edges. LLVM's SanitizerCoverage achieves this by *splitting* each critical

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

Table 3.5: 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 3.6: Proportion of encountered critical edges by transfer type.

edge with a "dummy" block, creating two new edges. Continuing example subsection 3.7.1, dummy `d` will split critical edge $\overrightarrow{\texttt{ac}}$ into $\overrightarrow{\texttt{ad}}$ and $\overrightarrow{\texttt{dc}}$, thus permitting path $\overrightarrow{\texttt{adc}}$ to be differentiated from $\overrightarrow{\texttt{abc}}$. But while such approach is indeed compatible with CGT's block-centric, interrupt-driven coverage, my analysis of eight real-world binaries shows **over 1 in 5** edges are critical (Table 3.4), 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, I instrument the same eight real-world binaries and dynamically record their instruction traces.[7] In conjunction with the statically-

[7] I 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).

generated control-flow graphs, I 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 3.5).[8]

As shown in Figure 3.12 and Table 3.6, my 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.

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), I 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. My 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 $\overrightarrow{\texttt{ac}}$ in subsection 3.7.1'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

[8] 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.

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. I envision two possible jump mistargeting strategies (Figure 3.13): one leveraging *embedded* interrupts, and another with *zero-address* interrupts.

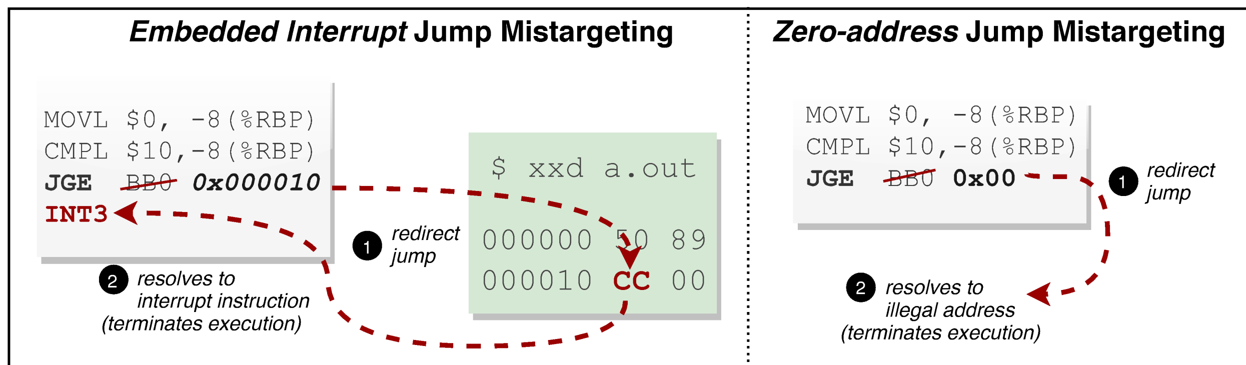


Figure 3.13: A visualization of jump mistargeting via embedded (left) and zero-address interrupts (right).

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 `0xFFBFFBFA` (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).

3.7.2 Supporting Hit Counts

Most fuzzers today adopt AFL-style [146] 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, I observe that most new hit count coverage is localized to *loops* (e.g., `for()`, `while()`). As Rawat and Mounier [104] 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.

Bucketed Unrolling

AFL-style [146] 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., `head` $\rightarrow$ $\texttt{body}_1$ $\rightarrow$ ... $\rightarrow$ $\texttt{body}_n$ $\rightarrow$ `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, I 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.

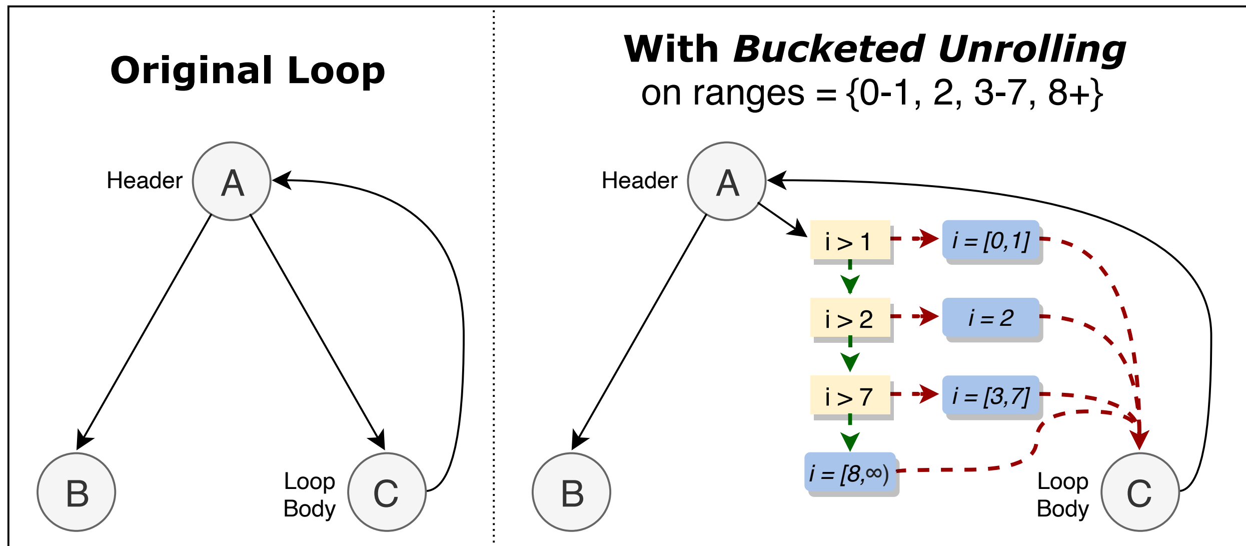


Figure 3.14: Bucketed unrolling applied to a simple loop.

As shown in Figure 3.14, 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.

3.8 Implementation: HeXcite

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

3.8.1 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 [91], and component 3 atop the industry-standard fuzzer AFL [146]. Below I briefly discuss each and their synergy in facilitating coverage-preserving CGT.

Binary Generation: HeXcite's workflow is similar in nature to UnTracer's (Figure 3.1); i.e., I 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 [76]), there are by now a number of static binary rewriters with comparable qualities. I examine several popular and/or emerging security-oriented binary rewriters—Dyninst [96], McSema [31], RetroWrite [32], and ZAFL [91]—and distill a set of properties I 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.* I 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. I expect that with

additional engineering effort, my 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;[9] and performing all code transformation at this IR level (e.g., injecting bucketed unrolling's range checks subsection 3.8.3), 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 [95, 138]. ZAFL addresses this challenge conservatively via *address pinning* [56, 64], 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;[10] 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* [91] *makes it promising for facilitating coverage-preserving CGT.* My 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 ad-

---

[9] ZAFL's disassembly supports mixing-and-matching of recursive descent and linear sweep. The current tools utilized are based on IDA Pro [54] and GNU objdump [45].

[10] 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.

dresses in the oracle. To generate this mapping, I save the original and rewritten control-flow graphs for both the oracle and tracer binaries. I 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, I 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), I omit the block to avoid problematic interrupts; I observe this generally amounts to no more than a handful of instances per binary, and does not impact HeXcite's overall coverage (section 3.9.2–section 3.9.2).

The Fuzzer: Like UnTracer, I implement HeXcite atop the industry-standard fuzzer AFL [146] 2.52b with several changes in test case handling logic (Figure 3.15). I 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`), I alter HeXcite's fuzzer-side crash-handling logic as follows: if a test case crashes the oracle, I re-run it on the tracer to verify whether it is a *true* or a *mistargeted* crash; if it does not crash the tracer, I 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. I 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 [41, 53, 69].

3.8.2 Implementing Jump Mistargeting

We implement *zero-address* jump mistargeting for the common-case of critical edges, conditional jump target branches (subsection 3.7.1), as follows. To statically identify critical

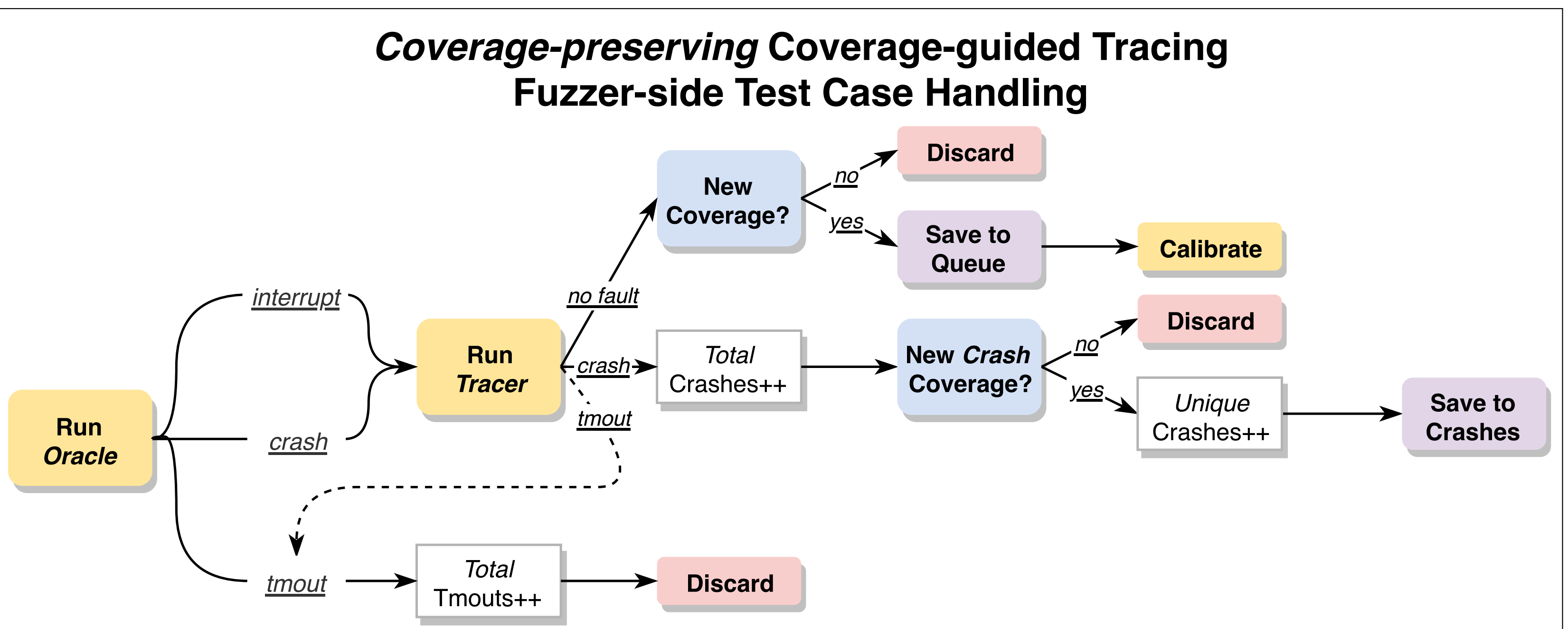


Figure 3.15: **HeXcite**'s fuzzer-side test case handling logic. Like UnTracer, I discard timeout-producing test cases; however, I re-run crashing test cases to determine whether they are a *true* crash (i.e., occurring on both the oracle and tracer) or the result of hitting an oracle *mistargeted* edge (generally triggering a `SIGSEGV` from the jump being redirected to the zero address).

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 3.5). 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.

I 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, I 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 I then statically overwrite the jump's displacement operand with the little-endian encoding of the zero-address-mistargeted displacement, and update my oracle-to-tracer mapping accordingly (e.g., `(0x400400,30,0x7C000000)` for the example in subsection 3.7.1).

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 [123] 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 (subsection 3.10.1). While recent work [73] reveals the possibility that indirect edges may be modeled at the binary level, such approaches are still too imprecise to be realistically deployed; hence, I conservatively omit indirect critical edges as I observe they have little overall significance on dynamically-seen control-flow (Figure 3.12).

3.8.3 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 [101]: 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 $\overrightarrow{\texttt{bh}}$ from some block $\texttt{b} \in \texttt{S}$ such that `h` dominates `b`.[11] Though binary-level loop head/body detection is difficult—particularly around complex optimizations like Loop-invariant Code Motion—I observe that the standard dominance-based algorithm is sufficient; and while **HEXCITE** attains the highest loop coverage in my evaluation (section 3.9.2), I 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, I 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 [96] prefers to conservatively insert new code at basic block entrypoints to avoid clobbering occupied registers; while RetroWrite [32] and ZAFL [91] 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

[11] 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]

resulting assembly resembles the following (shown in Intel syntax):

```
_loop_head:
    incl  rdx
    cmpl  rdx, 1
    jle    _loop_body
    cmpl  rdx, 2
    jle    _loop_body
    ...
_loop_body:
```

To facilitate signaling of a range change, I 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, I apply this transformation to both the oracle and tracer binaries.

3.9 Evaluation

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

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?

3.9.1 Experiment Setup

Below I provide expanded detail on my evaluation: the coverage-tracing approaches I am testing, my benchmark selection, and my experimental infrastructure and analysis procedures.

Competing Tracing Approaches: Table 3.7 lists the fuzzing coverage-tracing approaches tested in my evaluation. I evaluate my binary-only coverage-preserving CGT implementation, HeXcite, alongside the current *block-coverage-only* CGT approach **UnTracer** [89].[12] To test HeXcite's fidelity against the conventional *always-on* coverage tracing in binary fuzzing, I also evaluate the leading binary tracers **QEMU** (AFL [146] and honggFuzz's [119] default approach for fuzzing binary-only targets); **Dyninst** (a popular static-rewriting-based alternative [61]); and **RetroWrite** [32] (a recent static-rewriting-based instrumenter). Lastly, I replicate UnTracer's evaluation for open-source targets by further comparing against **AFL-Clang** (AFL's [146] source-level always-on tracing) [89]. I 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 [89]	coverage-guided	binary	block
QEMU [146]	always-on	binary	edge + counts
Dyninst [61]	always-on	binary	edge + counts
RetroWrite [32]	always-on	binary	edge + counts
Clang [146]	always-on	source	edge + counts

Table 3.7: Fuzzing coverage tracers evaluated alongside HeXcite; and their *type*, *level*, and *coverage* metric.

Benchmark Selection: My benchmark selection (Table 3.8) follows the current standard

[12] 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, I implement a *fully binary-only* version suitable across all 12 of my evaluation benchmarks.

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 3.8: My evaluation benchmark corpora.

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 [41, 53, 69] is in accelerating *binary-only* fuzzing, I 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 my bug-finding evaluation.

For each tracing approach I 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 [91]); `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: I 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 [75] standard I run 16×24-hour trials per benchmark for each of the coverage-tracing approaches listed in Table 3.7, amounting to over 2.4 years' of total compute time across my 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. I repurpose the same system for all data post-processing.

3.9.2 Q1: Coverage Evaluation

To understand the trade-offs of adapting CGT to finer-grained coverage metrics, I 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. I detail my experimental setup and results below.

Code Coverage

I compare the code coverage of all tracing approaches in Table 3.7. I utilize AFL++'s Link Time Optimization (LTO) instrumentation [41] to build *collision-free* edge-tracking versions of each binary; the same technique is applied to my four closed-source benchmarks (Table 3.8) with the help of the industry-standard binary-to-LLVM lifting tool McSema [31]. I measure each trial's code coverage by replaying its test cases on the LTO binary using AFL's `afl-showmap` [146] utility and compute the average across all 16 trials. Table 3.9 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 3.16 shows the relative edge coverage over 24-hours for several benchmarks.

Versus UnTracer: As Table 3.9 shows, HeXcite surpasses UnTracer in total coverage

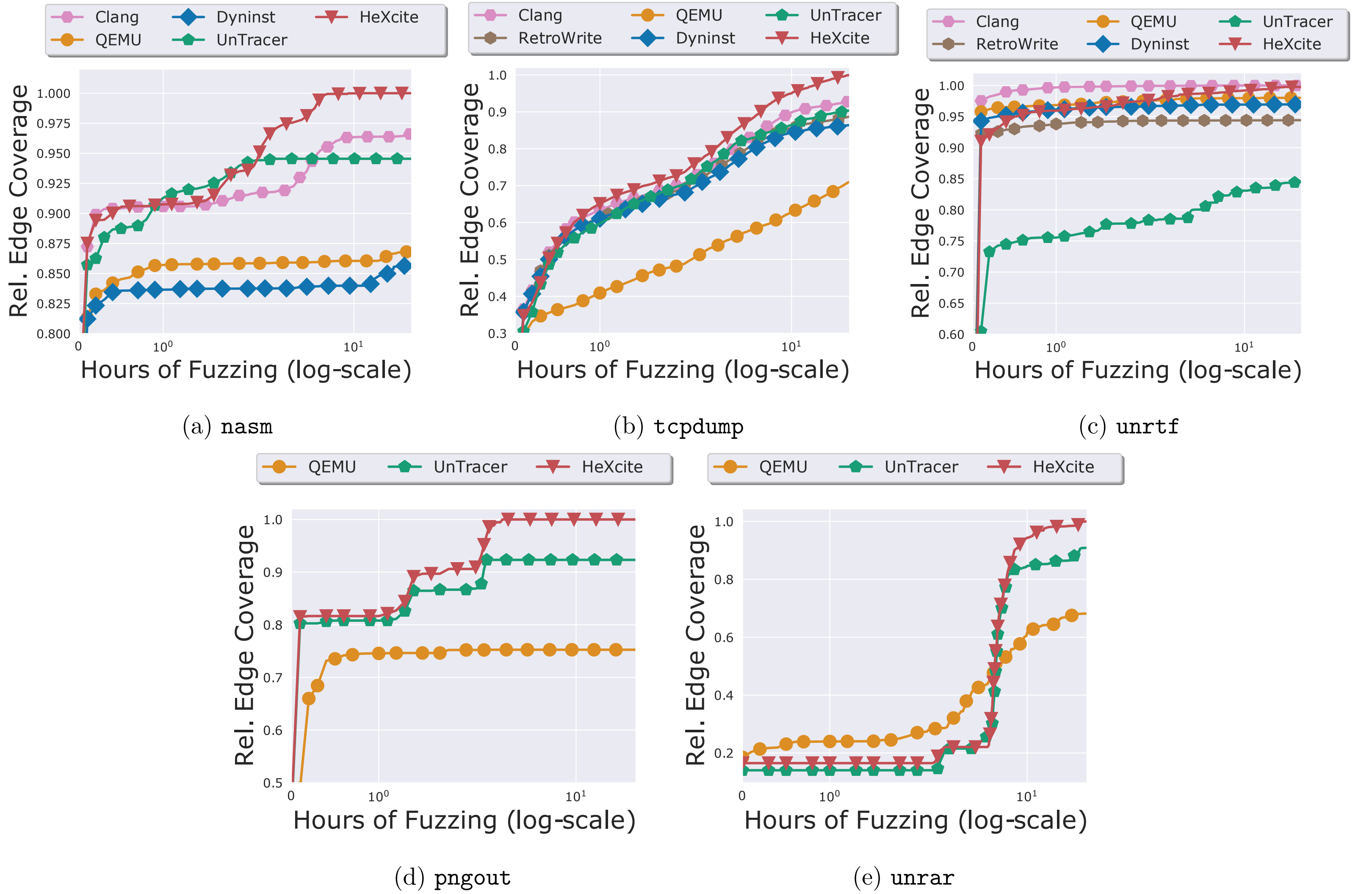


Figure 3.16: HeXcite's mean code coverage over time relative to all supported tracing approaches per benchmark. I log-scale the trial duration (24 hours) to more clearly show the end-of-fuzzing coverage divergence.

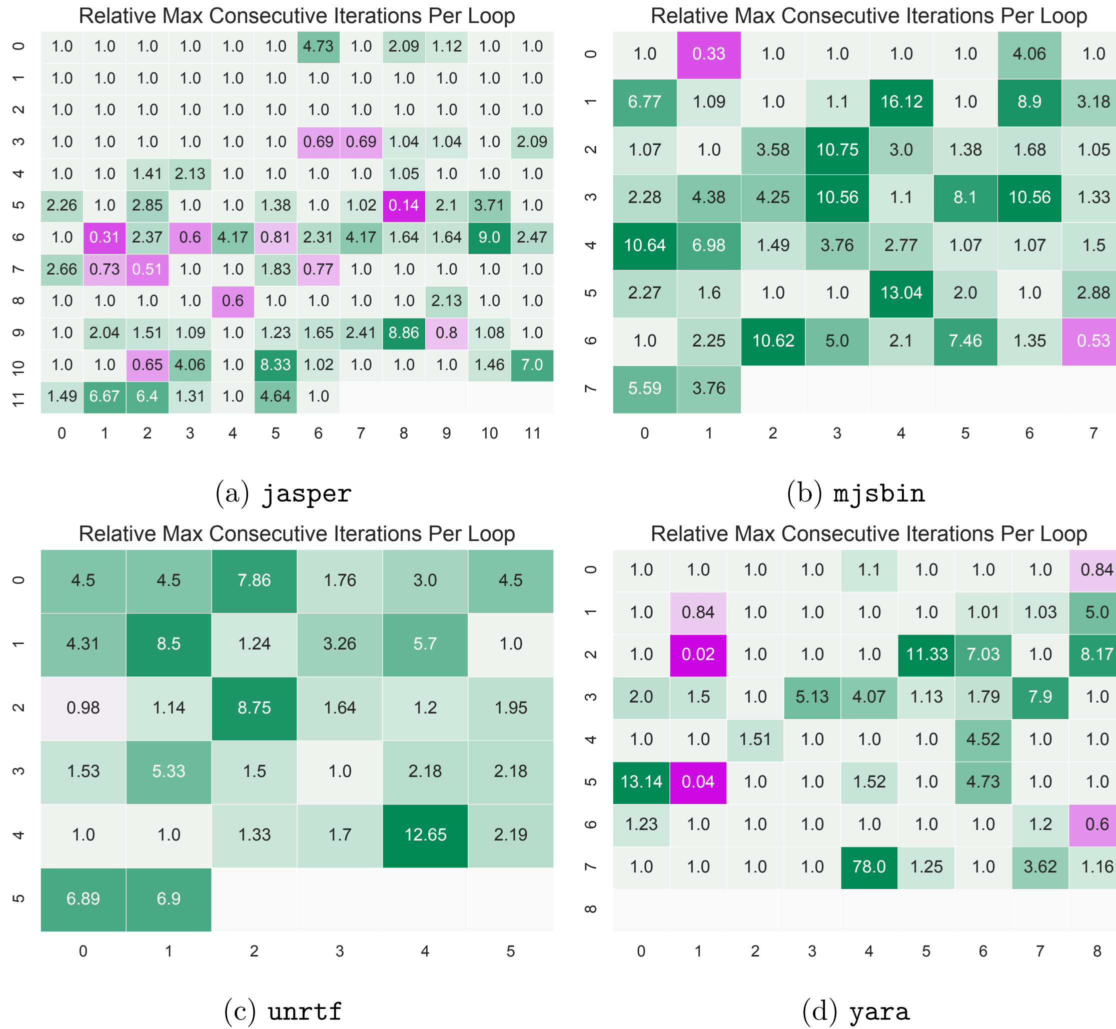
Binary	vs. *Coverage-guided Tracing*		vs. Binary- and Source-level *Always-on Tracing*							
	HeXcite / **UnTracer**		HeXcite / **QEMU**		HeXcite / **Dyninst**		HeXcite / **RetroWrite**		HeXcite / **Clang**	
	Rel. Cov	MWU	Rel. Cov	MWU	Rel. Cov	MWU	Rel. Cov	MWU	Rel. Cov	MWU
`jasper`	1.04	0.403	1.71	**<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 3.9: **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.

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 3.16c) 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 my entire evaluation.

Versus binary-only always-on tracing: I 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, I find that all tracers' runs are dominated by timeout-inducing inputs, causing each to see roughly equal execution speeds, and hence, code coverage. While I 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.



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

Figure 3.17: **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.

Loop Coverage

To determine if coverage-preserving CGT is more effective at covering code loops, I develop a custom LLVM instrumentation pass to report the maximum consecutive iterations per loop per trial. Despite my success in lifting my closed-source benchmarks to add edge-tracking

Binary	**HeXcite/ UnTracer**	**HeXcite/ Clang**
	Rel. LoopCov	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 3.10: 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.

instrumentation (section 3.9.2), none of my binary-to-LLVM lifters (McSema, rev.ng, RetDec, reopt, llvm-mctoll, or Ghidra-to-LLVM) succeeded in recovering the loop metadata necessary for my LLVM loop transformation to work; thus my loop analysis is restricted to my eight open-source benchmarks.

I 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). I compute each loop's mean from the maximum consecutive iterations for all trials per benchmark–tracer pair, capping iterations at 128 as AFL omits hit counts beyond this range. Table 3.10 reports HeXcite's mean coverage across all loops for each binary relative to UnTracer and AFL-Clang; and Figure 3.17 shows a heatmap of HeXcite's per-loop coverage relative to UnTracer's for several benchmarks.

Versus UnTracer: As Table 3.10 shows, HeXcite's bucketed unrolling brings **130%** higher loop penetration coverage over UnTracer. I see that UnTracer beats HeXcite on a minutia

of loops per benchmark (Figure 3.17)—expectedly—as its inability to track loop progress inevitably constrains fuzzing to exploring the same few loops trial after trial. I 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: I 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 I posit that bucketed unrolling has further optimization potential (e.g., halving the number of buckets, selective insertion, etc.), I 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.

3.9.3 Q2: Performance Evaluation

To measure the impacts of finer-grained coverage on CGT performance, I perform a piecewise 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 3.11. To ascertain where coverage-preserving CGT's performance stands with respect to always-on tracing, I 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 3.18.

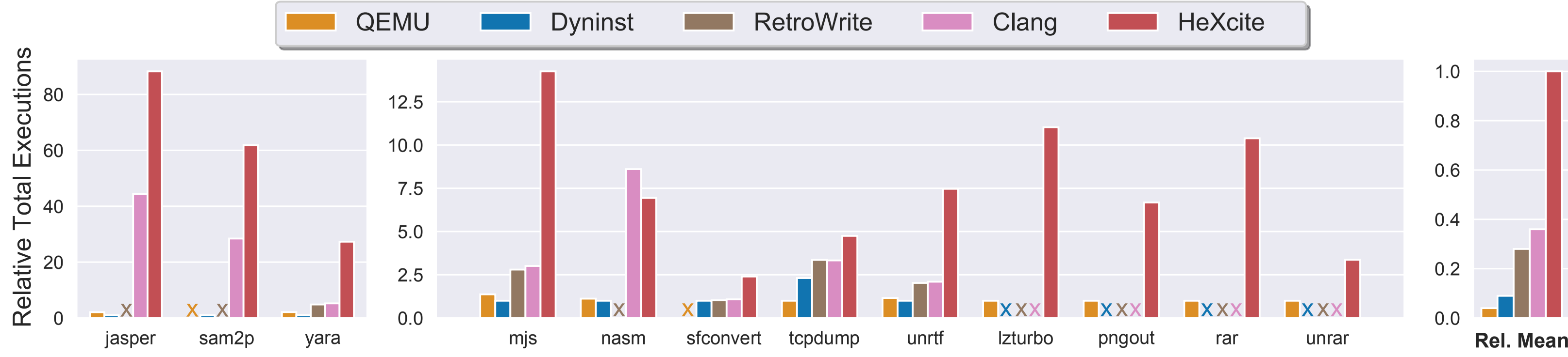


Figure 3.18: HeXcite's mean throughput relative to conventional coverage tracers. I normalize throughput to the worst-performing tracer per benchmark, and compute each tracer's mean performance relative to HeXcite's across all benchmarks (shown in the rightmost plot). For each benchmark I omit incompatible tracers (denoted by a colored ✗). *All comparisons to* HeXcite *yield a statistically significant difference* (i.e., Mann-Whitney U test $p < 0.05$).

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

Versus UnTracer: As Table 3.11 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 section 3.9.2 and section 3.9.2 show, coverage-preserving CGT attains the highest edge and loop coverage of all tracers in my 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 3.11 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 3.18 shows, HeXcite averages **11.4**×, **24.1**×, and **3.6**× the throughput of always-on binary-only tracers QEMU, Dyninst, and

RetroWrite, respectively. Furthermore, I 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, I 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.

3.9.4 Q3: Bug-finding Evaluation

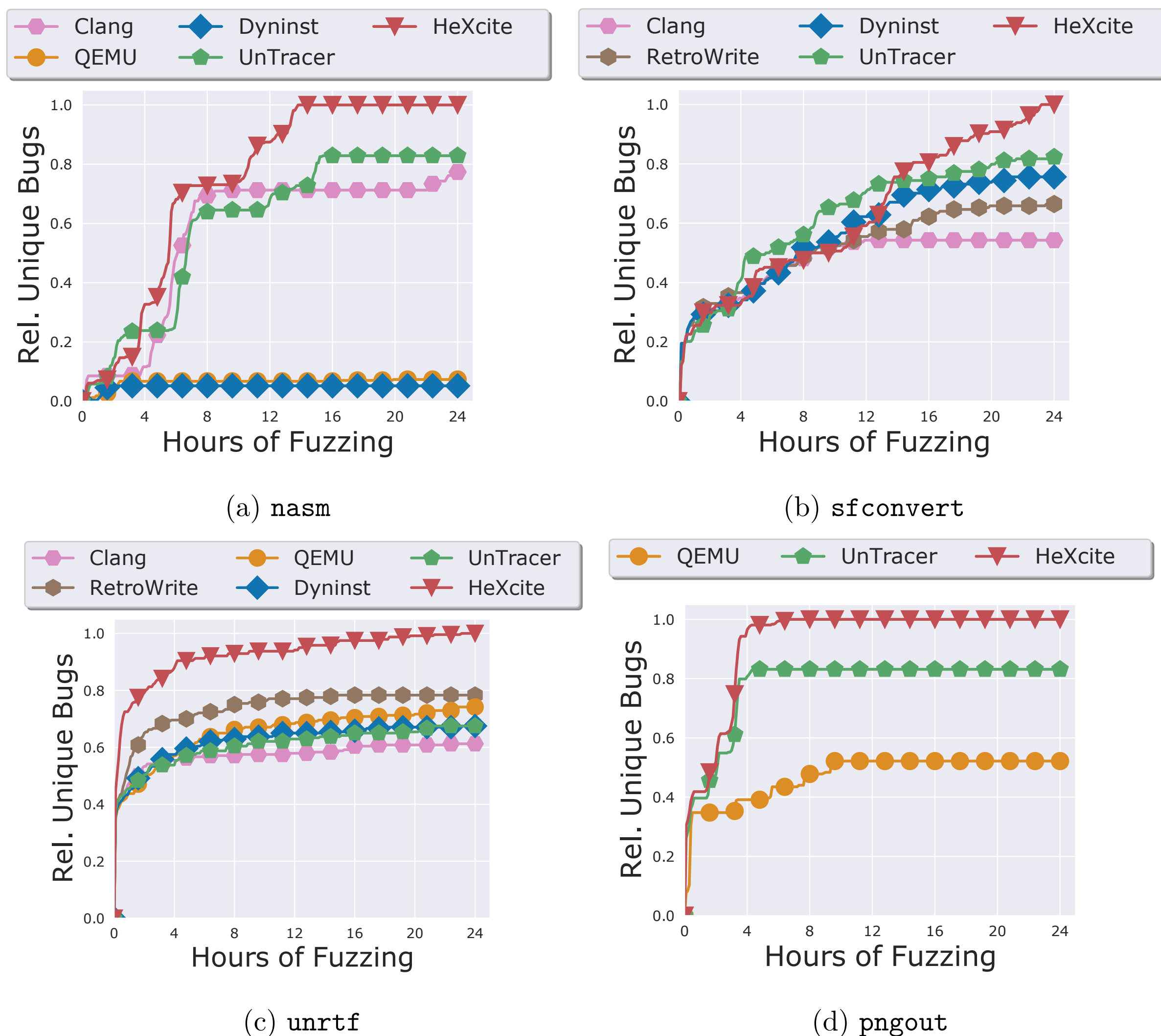
I evaluate the crash- and bug-finding effectiveness of coverage-preserving CGT across my 12 benchmarks. To triage raw crashes into bugs, I apply the popular "fuzzy stack hashing" methodology, trimming stack traces to their top-6 entries, and hash each with their corresponding fault address and reported error. I make use of the binary-only AddressSanitizer implementation QASan [40] to extract crash stack traces and errors.

Unique Bugs and Crashes

Table 3.12 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

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 3.12: **HeXcite**'s mean crashes and bugs relative to UnTracer, QEMU, Dyninst, RetroWrite, and AFL-Clang. I 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.



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

Figure 3.19: HEXCITE's mean unique bugs over time relative to all supported tracing approaches per benchmark.

AFL-Clang. Figure 3.19 shows the mean unique crashes over time for several benchmarks. I omit `lzturbo` and `rar` as no fuzzing run found crashes in them.

Versus UnTracer: As Table 3.12 shows, HEXCITE exposes a mean **12%** more bugs than UnTracer. In conjunction with the plots shown in Figure 3.19, I 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 my performance experiments (subsection 3.9.3), **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.

Bug Diversity

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

(a) HEXCITE **vs. RetroWrite** (b) HEXCITE **vs. AFL-Clang**

Figure 3.20: HEXCITE's total unique bugs found versus the fastest conventional always-on tracers RetroWrite (binary-only) and AFL-Clang (source-level).

To determine whether coverage-preserving CGT effectively reveals *many* bugs, or is merely constrained to the same few time after time, I 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 3.20 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.

Bug Time-to-Exposure

I 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 3.13 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 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.

3.10 Discussion

Below I discuss several limitations of coverage-preserving CGT and my prototype implementation, HeXcite.

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 3.13: **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.

3.10.1 Indirect Critical Edges

While resolving *direct* critical edges is straightforward through jump mistargeting or edge splitting (subsection 3.7.1), *indirect* critical edges (i.e., indirect jumps/calls/returns) remain a universal problem even for source-level solutions like LLVM's SanitizerCoverage [123]. Below I 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 $\overrightarrow{\mathtt{ib}}$ (with `i` having outgoing indirect edges to some other blocks `x` and `y`) and direct edge $\overrightarrow{\mathtt{ab}}$, SanitizerCoverage first cuts block `b`'s header from its body into two copies, $\mathtt{b}_{0i}$ and $\mathtt{b}_{0a}$. Second, as the indirect transfer's destination is resolved dynamically and thus cannot be statically moved, $\mathtt{b}_{0i}$'s location must be pinned to that of the original block `b`. Finally, the twin header blocks ($\mathtt{b}_{0i}$ and $\mathtt{b}_{0a}$) are appended with a direct jump to `b`'s body, $\mathtt{b}_1$—effectively splitting the original indirect critical edge $\overrightarrow{\mathtt{ib}}$ with edges $\overrightarrow{\mathtt{ib}_{0i}}$ and $\overrightarrow{\mathtt{b}_{0i}\mathtt{b}_1}$; and direct edge $\overrightarrow{\mathtt{ab}}$ with $\overrightarrow{\mathtt{ab}_{0a}}$ and $\overrightarrow{\mathtt{b}_{0a}\mathtt{b}_1}$. 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 ($\overrightarrow{\mathtt{i}_1\mathtt{b}}$ and $\overrightarrow{\mathtt{i}_2\mathtt{b}}$), *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 (section 2.2). 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). I 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.

3.10.2 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. I discuss both of these below.

Bucket Granularity: Our current implementation of bucketed unrolling (subsection 3.8.3) 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, I 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 my analysis of the bugs exclusively found by exhaustive hit counts (Figure 3.20b) reveals that none are outside the reach of HEXCITE, I 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.

3.10.3 Improving Performance

The fuzzing-oriented binary transformation platform currently utilized in HEXCITE, ZAFL [91], adopts a code layout algorithm that rewrites all direct jumps to have 32-bit PC-relative signed displacements. While this is well-suited to my implementation of zero-address jump mistargeting (subsection 3.8.2)—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, I thus envision potential for faster "hybrid" mistargeting schemes that coalesce both zero-address *and* embedded interrupt styles.

3.10.4 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, I expect that HeXcite will capitalize on future engineering improvements in static rewriting to bring accelerated fuzzing to the broader software ecosystem.

3.11 Related Work

I 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.

3.11.1 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 [146], DrAFL [115], UnicornAFL [130]) generally focus on simplifying or expanding the caching of translated code [11]; while those using static rewriting (e.g., ZAFL [91], Dyninst [96], RetroWrite [32]) tackle various challenges related to generated code performance. Though my coverage-preserving CGT prototype, HeXcite, currently leverages the ZAFL rewriter, I believe that future advances in binary instrumentation will enable it to achieve performance even closer to native speed.

3.11.2 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 [65], CollAFL [43]), their principles are well-suited to binary-only fuzzing. A recent fork of AFL-Dyninst [61] 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, I 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.

3.11.3 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 [146]). Xu et al. [139] 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 [51], 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 [146] and honggFuzz [119] support parallelization out-of-the-box, and recent work by Falk [38] achieves even faster speed by leveraging vector-

ized instruction sets. As execution and coverage tracking work hand-in-hand during fuzzing, I view such accelerated execution mechanisms as complementary to HeXcite's accelerated coverage tracking.

3.12 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. I 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.

My 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. I envision a new era in software fuzzing, where synergistic and target-tailored approaches will maximize *common-case* performance with *infrequent-case* precision.

Chapter 4

Compiler-quality Code Transformation for Closed-source Fuzzing

4.1 Introduction

Software vulnerabilities represent a persistent threat to cybersecurity. Identifying these bugs in both modern and legacy software is a tedious task; manual analysis is unrealistic, and heavyweight program analysis techniques like symbolic execution are unscalable due to the sheer size of real-world applications. Instead, developers and bug-hunters alike have largely adopted a software testing strategy known as fuzzing.

Fuzzing consists of mutationally generating massive amounts of test cases and observing their effects on the target program, with the end goal of identifying those triggering bugs. The most successful of these approaches is *coverage-guided grey-box fuzzing*, which adds a feedback loop to keep and mutate only the few test cases reaching new code coverage; the intuition being that exhaustively exploring target code reveals more bugs. Coverage is collected via instrumentation inserted in the target program's basic blocks. Widely successful coverage-guided grey-box fuzzers include AFL [146], libFuzzer [107], and honggFuzz [119].

Most modern fuzzers require access to the target's source code, embracing *compiler instrumentation*'s low overhead for high fuzzing throughput [107, 119, 146] and increased crash finding. State-of-the-art fuzzers further use compilers to apply *fuzzing-enhancing program transformation* that improves target speed [43, 65], makes code easier-to-penetrate [1], or tracks interesting behavior [22]. Yet, compiler instrumentation is impossible on closed-source targets (e.g., proprietary or commercial software). In such instances fuzzers are restricted to *binary instrumentation* (e.g., Dyninst [96], PIN [81], and QEMU [9]). But while binary instrumentation succeeds in many non-fuzzing domains (e.g., program analysis, emulation, and profiling), available options for *binary-only fuzzing* are simply unable to uphold both the speed and transformation of their compiler counterparts—limiting fuzzing effectiveness. Despite advances in general-purpose binary instrumentation [10, 56, 64, 134, 135], it remains an open question whether compiler-quality instrumentation capabilities *and* performance are within reach for binary-only fuzzing.

To address this challenge I scrutinize the field of binary instrumentation, identifying key characteristics for achieving performant and general-purpose binary-only fuzzing instrumentation. We apply this standard in designing ZAFL: an instrumentation platform bringing *compiler-quality* capabilities and speed to x86-64 binary-only fuzzing. I demonstrate how ZAFL facilitates powerful fuzzing enhancements with a suite of five transformations, ported from compiler-based fuzzing contexts. I show how ZAFL's capabilities improve binary-only fuzzing bug-finding: among evaluations on the LAVA-M corpus and eight real-world binaries, ZAFL finds an average of **26–96%** more unique crashes than the static rewriter AFL-Dyninst; and **37–131%** more than the dynamic translator AFL-QEMU. I further show that ZAFL achieves compiler-quality overhead of **27%** and increases fuzzing throughput by **48–78%** and **131–203%** over AFL-Dyninst and AFL-QEMU, respectively. Lastly, I show that ZAFL scales to real-world software—successfully instrumenting 56 binaries of varying type

(**33** open- and **23** closed-source), size (**10K–100MB**), complexity (**100–1,000,000** basic blocks), and platform (**30** Linux and **12** Windows).

In summary, this work contributes the following:

- I examine the challenges of achieving compiler-quality instrumentation in binary-only fuzzing, developing a criteria for success, and highlighting where popular binary-only instrumenters fit with respect to my criteria.
- We apply this criteria in designing Zafl: a platform for state-of-the-art compiler-quality instrumentation—and speed—in binary-only fuzzing. Zafl's architectural focus on fine-grained instrumentation facilitates complex fuzzing-enhancing transformations in a performant manner.
- We show that it is possible to achieve fuzzing-enhancing program transformation in a performant manner for binary-only contexts by implementing five of such transformations derived from existing compiler-based implementations in Zafl, and evaluating runtime overhead.
- I demonstrate how Zafl improves fuzzing effectiveness; on average Zafl's performant, fuzzing-enhancing program transformations enable fuzzers to find more unique crashes than the leading binary-only fuzzing instrumenters AFL-Dyninst and AFL-QEMU across both LAVA-M and real-world benchmarks.
- I show that Zafl supports real-world binaries of varying characteristics, size, complexity, and platform—even those binaries not supported by other instrumenters.
- We open-source Zafl and all benchmark corpora at: `https://git.zephyr-software.com/opensrc/zafl`.

4.2 Compiler-based Fuzzing Enhancements

Coverage-guided fuzzing spans two distinct domains: compiler-based and binary-only, with both using program instrumentation to track test case code coverage. Much of fuzzing's success is due to the high throughput made possible by fast compiler instrumentation [123, 146]. Though advanced fuzzers introduce more heavyweight analyses [7, 22, 117, 143], the core of these approaches remains the standard coverage-guided fuzzing loop (Figure 2.2)—amounting to over 90% of their execution time [89]; recent feedback enhancements (e.g., context sensitivity) only increase the proportion of time spent tracing execution. Thus, my focus is *performant fuzzing-enhancing transformations* in the absence of source code.

Focus	Category	Effect on Fuzzing
Performance	*Instrumentation Pruning*	Overhead reduction from fewer blocks instrumented
	Instrumentation Downgrading	Overhead reduction from lighter-weight instrumentation
Feedback	*Sub-instruction Profiling*	Incremental coverage to guide code penetration
	Extra-coverage Behavior	Ability to consider finer-grained execution behavior

Table 4.1: Popular compiler-based fuzzing-enhancing program transformations, listed by category and effect.

State-of-the-art fuzzers leverage compiler instrumentation to add transformations that improve fuzzing *performance* and *feedback* (e.g., AFL++ [41], Angora [22], CollAFL [43], honggFuzz [119], INSTRIM [65], libFuzzer [107]). Performance-enhancing transformation helps alleviate the runtime cost of coverage tracing and other feedback sources. Feedback-enhancing transformations reveal finer-grained program progress, beyond traditional code coverage metrics. I broadly examine popular fuzzers and identify four categories of fuzzing-enhancing transformation that target the core coverage-guided loop (Table 4.1): (1) *instru-*

mentation pruning, (2) *instrumentation downgrading*, (3) *sub-instruction profiling*, and (4) *extra-coverage behavior tracking*. Below I detail each transformation.

4.2.1 Instrumentation Pruning

Graph reducibility techniques [57, 121] are used in fuzzing to elide instrumenting some target basic blocks, thus lowering overall runtime overhead. AFL's [146] compiler instrumentation permits a "ratio": 100 instruments all blocks; 0 only function entries; and values in between form a probability to arbitrarily skip blocks. Clearly, culling random blocks risks coverage blind-spots. More rigorous *CFG-aware* analyses [41, 65] prune blocks implicitly covered by others: formally, for N blocks and M unique paths over N, it is possible to select a subset $N' \in N$ such that the M' unique paths over N' equals M. INSTRIM [65] only instruments blocks targeted by backward edges and tracks loops either by entry or pre-entry blocks (the latter forgoing loop iteration tracking).

4.2.2 Instrumentation Downgrading

The majority of today's fuzzers track coverage in the form of edges (i.e., branches between basic blocks). Edges are typically recorded as hashes of their start and end blocks (computed in the body of the end block's instrumentation), as popularized by the fuzzer AFL [146]. Edge hashing requires several instructions (two index fetches, a hash, array update, and an `XOR`); but given that blocks themselves are small, maintaining speed requires inserting as few instructions as necessary. CollAFL [43]'s compiler instrumentation optimizes single-predecessor blocks by downgrading them to fewer-instruction block coverage (i.e., $cov(A \rightarrow B) \equiv cov(B)$).

4.2.3 Sub-instruction Profiling

Fuzzers struggle to penetrate code guarded by complex predicates like "magic bytes" [104], nested checksums [7], and switch cases [1]. Most fuzzers track edge/block coverage and hence are oblivious to "incremental" predicate progress. Recent compiler-based efforts apply sub-instruction profiling—decomposing multi-byte conditionals into single-byte comparisons (e.g., CmpCov [70], honggFuzz [119], laf-Intel [1]). Such splitting of roadblocks into smaller, simpler problems facilitates greater fuzzing code coverage.

4.2.4 Extra-coverage Behavior Tracking

An area of current research in fuzzing is the inclusion of execution behavior beyond traditional code coverage. Although I foresee future work considering metrics such as register or memory usage, the existing body of work on extra-coverage behavior tracking focuses on context sensitivity. Context-sensitive coverage tracks edges along with their preceding calling context. For example, given two paths over the same set of edges, $A \rightarrow B \rightarrow C$ and $B \rightarrow A \rightarrow C$, context-insensitive coverage misses the second path as it offers no new edges; however context-sensitive coverage reveals two distinct calls: $B \rightarrow C$ and $A \rightarrow C$. Several LLVM implementations exist for both function- and callsite-level context sensitivity [22, 41].

4.3 Binary-only Fuzzing: the Bad & the Ugly

Program transformation has become ubiquitous in compiler-based fuzzers (e.g., AFL++ [41], CollAFL [43], laf-Intel [1]), and for good reason: it makes fuzzing significantly more powerful. Despite these advantages there is no platform that adapts such transformation to binaries in an effective manner—severely impeding efforts to fuzz closed-source software.

This section examines existing binary instrumenters and their limitations that prevent them from attaining effective binary-only fuzzing instrumentation. I follow this exploration with an identification of the key instrumenter design attributes necessary to support *compiler-quality* fuzzing-enhancing program transformation and speed.

4.3.1 Limitations of Existing Platforms

Coverage-guided fuzzers trace test case code coverage via fast compiler instrumentation; and state-of-the-art efforts further leverage compilers to apply fuzzing-enhancing program transformation. In binary-only fuzzing, code coverage is traced by one of three mechanisms: (1) hardware-assisted tracing, (2) dynamic binary translation, or (3) static binary rewriting. Below I briefly detail each, and weigh their implications with respect to supporting the extension of compiler-quality transformation to binary-only fuzzing.

- **Hardware-assisted Tracing.** Newer processors are offering mechanisms that facilitate binary code coverage (e.g., Intel PT [66]). Fuzzing implementations are burdened by the need for costly trace post-processing, which reportedly incurs overheads as high as **50%** over compilers [7, 24]; but despite some optimistic performance improvements [51], hardware-assisted tracing currently remains incapable of modifying programs—and hence fails to support fuzzing-enhancing program transformation.

- **Dynamic Binary Translators.** Dynamic translators apply coverage-tracing on-the-fly as the target is executing (e.g., DynamoRIO [60], PIN [81], and QEMU [9]). Translators generally support many architectures and binary characteristics; and offer deep introspection that simplifies analysis and transformation [41, 146]. However, existing dynamic translators attain the worst-known fuzzing performance: recent work shows AFL-QEMU's average overhead is well over **600%** [89], and AFL-DynamoRIO [60]

and AFL-PIN [62] report overheads of up to **10x** and **100x** higher, respectively.

- **Static Binary Rewriters.** Static rewriting improves performance by modifying binaries prior to runtime (e.g., Dyninst [61]). Unfortunately, static rewriting options for binary-only fuzzing are limited. AFL-Dyninst is the most popular, but sees prohibitively-high fuzzing overheads of over **500%** [89] and is restricted to Linux programs. RetroWrite suggests reassembleable-assembly is more performant and viable, but it relies on AFL's assembly-time instrumentation which is both unsupportive of transformation and reportedly **10–100%** slower than compile-time instrumentation [146]; and moreover, it does not overcome the generalizability challenges of prior attempts at reassembleable-assembly (e.g., Uroboros [135], Ramblr [134]), and is hence limited to position-independent Linux C programs. Neither scale well to stripped binaries.

As summarized in Table 4.2, the prevailing binary-only fuzzing coverage-tracing approaches are limited in achieving compiler-quality fuzzing instrumentation. Hardware-assisted tracing (Intel PT) is *incompatible* with program instrumentation/transformation and adds post-processing overhead. Dynamic translators (DynamoRIO, PIN, and QEMU) all face *orders-of-magnitude* worse overheads. Static rewriters (Dyninst and RetroWrite) fail to uphold both performance *and* transformation and are unsupportive of Windows software (the most popular being PE32+), common binary characteristics (e.g., position-dependent code), or the simplest obfuscation techniques (i.e., stripped binaries).

These limitations make fuzzing-enhancing transformations scarce in binary-only fuzzing. To my knowledge the only two such implementations exist atop of AFL-Dyninst (instruction pruning [61]) and AFL-PIN (context sensitivity [143])—both suffering from the central flaw that any of their potential benefits are outweighed by the steep overheads of their respective

Name	Fuzzing Appearances	Fuzzing Overhead	Supports Xform	Instrumentation			Supported Programs			
				type	invoked	liveness	PIC & PDC	C & C++	stripped	PE32+
LLVM	[1, 8, 15, 22, 25, 41, 43, 65, 107, 119, 146]	18–32%	✓	static	inline	✓	N/A	✓	N/A	N/A
Intel PT	[7, 13, 24, 51, 119]	19–48%	✗	hardware	replay	✗	✓	✓	✓	✓
DynamoRIO	[51, 60, 114]	>1,000%	✓	dynamic	inline	✓	✓	✓	✓	✓
PIN	[62, 67, 92, 104, 143]	>10,000%	✓	dynamic	inline	✓	✓	✓	✓	✓
QEMU	[28, 41, 141, 146]	>600%	✓	dynamic	inline	✓	✓	✓	✓	✓
Dyninst	[61, 79, 89, 120]	>500%	✓	static	tramp.	✗	✓	✓	✗	✗
RetroWrite	[32]	20–64%	✗	static	tramp.	✓	✗	✗	✗	✗

Table 4.2: A qualitative comparison of the leading coverage-tracing methodologies currently used in binary-only coverage-guided fuzzing, alongside compiler instrumentation (LLVM). No existing approaches are able to support compiler-quality transformation at compiler-level speed and generalizability.

binary instrumenters (over **500%** and **10,000%**, respectively [62, 89]).

> **Impetus:** Current binary instrumenters are fundamentally ill-equipped to support compiler-quality fuzzing instrumentation. I envision a world where binary-only and compiler-based fuzzing are not segregated by capabilities; thus we design a binary-only fuzzing instrumentation platform capable of performant compiler-quality transformation.

4.3.2 Fundamental Design Considerations

My analysis of how compilers support performant program transformations reveals four critical design decisions: **(1) rewriting versus translation**, **(2) inlining versus trampolining**, **(3) register allocation**, and **(4) real-world scalability**. Below I discuss the significance of each, and build a criteria of the instrumenter characteristics *best-suited* to compiler-quality instrumentation.

- **Consideration 1: Rewriting versus Translation.** Dynamic translation processes a target binary's source instruction stream as it is executed, generally by means of emulation [9]. Unfortunately, this requires heavy-lifting to interpret target instructions to the host architecture; and incurs significant runtime overhead, as evidenced by the poor performance of AFL-DynamoRIO/PIN/QEMU [60, 62, 146]. While translation does facilitate transformations like sub-instruction profiling [41], static binary rewriting is a more viable approach for fuzzing due to its significantly lower overhead. **Like compilers**, static binary rewriting performs all analyses (e.g., control-flow recovery, code/data disambiguation, instrumentation) *prior* to target execution, avoiding the costly runtime effort of dynamic translation. Thus, static rewriting is the most compatible with achieving compiler-quality speed in binary-only fuzzing.

Criterion 1: Instrumentation added via static rewriting.

- **Consideration 2: Inlining versus Trampolining.** A second concern is how instrumentation code (e.g., coverage-tracing) is invoked. Instrumenters generally adopt one of two techniques: *trampolining* or *inlining*. Trampolining refers to invocation via jumping to a separate payload function containing the instrumentation. This requires two transfers: one to the payload, and another back to the callee. However, the total instructions needed to accommodate this redirection is significant relative to a basic block's size; and their overhead accumulation quickly becomes problematic for fuzzing. **Modern compilers inline**, injecting instrumentation directly within target basic blocks. Inlining offers the least-invasive invocation as instrumentation is launched via contiguous instruction execution rather than through redirection. I thus believe that inlining is essential to minimize fuzzing instrumentation's runtime overhead and achieve compiler-quality speed in binary-only fuzzing.

Criterion 2: Instrumentation is invoked via inlining.

- **Consideration 3: Register Allocation.** Memory access is a persistent bottleneck to performance. On architectures with a finite set of CPU registers (e.g., x86), generating fast code necessitates meticulous register allocation to avoid clobbering occupied registers. Condition code registers (e.g., x86's `eflags`) are particularly critical as it is common to modify them; but saving/restoring them to their original state requires pushing to the stack and is thus ∼**10x** slower than for other registers. **Compilers track register liveness** to avoid saving/restoring dead (untouched) condition code registers as much as possible. Smart register allocation is thus imperative to attaining compiler-quality binary instrumentation speed.

> **Criterion 3:** Must facilitate register liveness tracking.

- **Consideration 4: Real-world Scalability.** Modern compilers support a variety of compiled languages, binary characteristics, and platforms. While dynamic translators (e.g., DynamoRIO, QEMU, PIN) are comparably flexible because of their reliance on emulation techniques, existing static rewriters have proven far less reliable: some require binaries be written in C despite the fact that developers are increasingly turning to C++ [32, 134, 135]. others apply to only position-independent (i.e., relocatable) code and neglect the bulk of software that remains position-dependent [32]; many presume access to debugging symbols (i.e., non-stripped) but this seldom holds true when fuzzing proprietary software [61]; and most are only Linux-compatible, leaving some of the world's most popular commodity software (Windows 64-bit PE32+) unsupported [32, 61, 134, 135]. A compiler-quality binary-only fuzzing instrumenter must therefore support these garden-variety closed-source binary characteristics and formats.

> **Criterion 4:** Support common binary formats and platforms.

While binary instrumenters have properties useful to many non-fuzzing domains (e.g., analysis, emulation, and profiling), attaining compiler-quality fuzzing instrumentation hinges on satisfying four core design criteria: **(C1) static rewriting**, **(C2) inlining**, **(C3) register liveness**, and **(C4) broad binary support**. Hardware-assisted tracing cannot modify programs and hence *violates* criteria (**C1**)–(**C3**). DynamoRIO, PIN, and QEMU adopt dynamic translation ($\overline{\textbf{C1}}$) and thus incur orders-of-magnitude performance penalties—before applying any feedback-enhancing transformation. Dyninst and RetroWrite embrace static rewriting but both rely on costlier trampoline-based invocation ($\overline{\textbf{C2}}$) and fail to support commodity binary formats and characteristics ($\overline{\textbf{C4}}$); and moreover, Dyninst's liveness-aware instrumen-

tation failed on my evaluation benchmarks ($\overline{\mathbf{C3}}$). Thus, compiler-quality instrumentation in a binary-only context demands a new approach that satisfies all four criteria.

4.4 The ZAFL Platform

Fuzzing effectiveness severely declines on closed-source targets. Recent efforts capitalize on compiler instrumentation to apply state-of-the-art fuzzing-enhancing program transformations; however, current binary-only fuzzing instrumenters are ineffective at this. As practitioners are often restricted to binary-only fuzzing for proprietary or commercial software, any hope of advancing binary-only fuzzing beseeches efforts to bridge the gap between source-available and binary-only fuzzing instrumentation.

To combat this disparity we introduce ZAFL: a compiler-quality instrumenter for x86-64 binary fuzzing. ZAFL extends the rich capabilities of compiler-style instrumentation—with compiler-level throughput—to closed-source fuzzing targets of any size and complexity. Inspired by recent compiler-based fuzzing advancements (section 4.2), ZAFL streamlines instrumentation through four extensible phases, facilitating intuitive implementation and layering of state-of-the-art fuzzing-enhancing program transformations. Below I detail ZAFL's internal architecture and guiding design principles.

4.4.1 Design Overview

As shown in Figure 4.1, ZAFL consists of two primary components (1) a static rewriting engine and (2) ZAX: our four IR-modifying phases for integrating compiler-quality instrumentation and fuzzing enhancements. Given a target binary, ZAFL operates as follows:

1. **IR Extraction**. From our (or any compatible) binary rewriter, ZAFL requests an

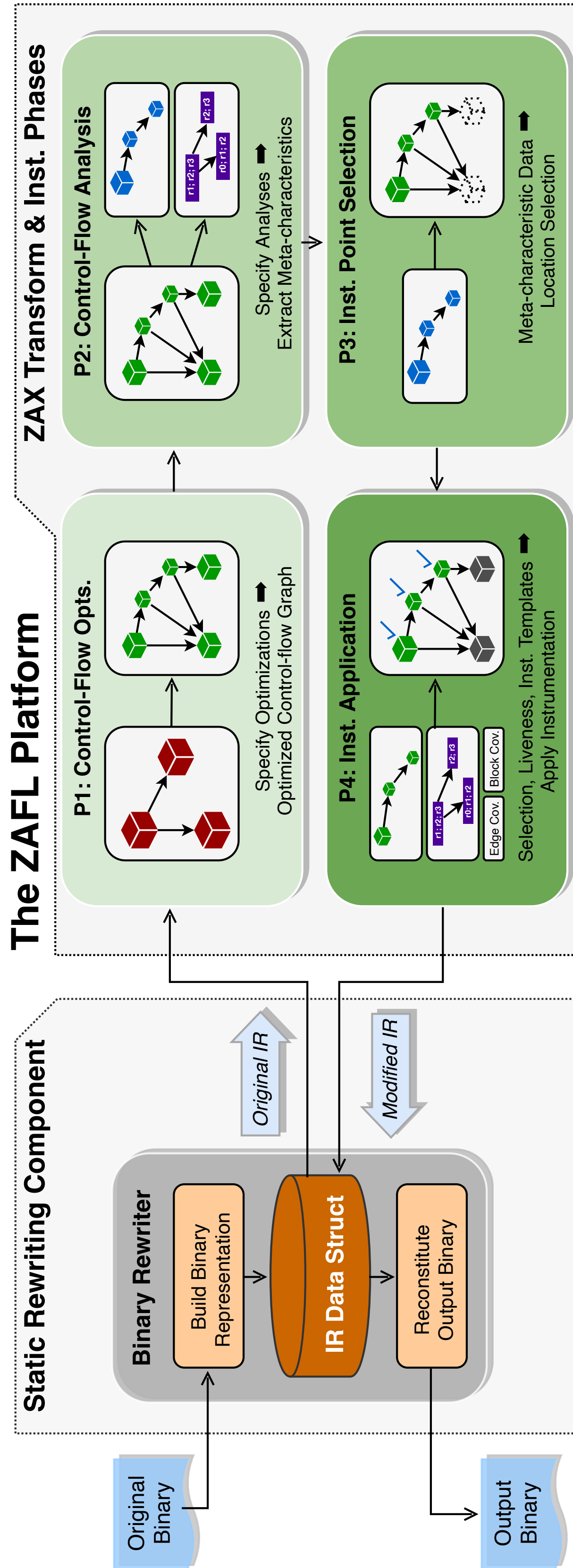


Figure 4.1: A high-level depiction of the ZAFL platform architecture and its four ZAX transformation and instrumentation phases.

intermediate representation (IR) of the target binary.

2. Zax. The resulting IR is then passed to Zax's four transformation and instrumentation phases:

   P1: Optimization,

   P2: Analysis,

   P3: Point Selection, and

   P4: Application.

3. **Binary Reconstitution**. After Zax applies program transformations and instrumentation at IR-level, Zafl transfers the modified IR back to the rewriting engine which generates the output binary for fuzzing.

Static Rewriting Engine

Zafl interacts with the binary rewriter of choice to first translate the target binary to an intermediate representation (IR) for subsequent processing in Zax; and secondly, to reconstitute an output binary from the Zax-modified IR.

I initially considered re-purposing LLVM IR-based rewriter McSema [31] due to its maturity and popularity in the static rewriting community, but ultimately ruled it out as both the literature [37] and my own preliminary evaluation reveal that it is a poor fit for fuzzing due to its high baseline overhead. Instead, for our prototype, we extend the GCC IR-inspired static rewriter Zipr [56, 64] as it meets the same criteria that McSema does (subsection 4.3.2), but has better baseline performance.

4.4.2 The Zax Transformation Architecture

Once target IR construction is finished, Zafl initiates Zax: our fuzzing instrumentation toolchain. Below I describe the intricacies of Zax's four core phases: **(1) Optimization**, **(2) Analysis**, **(3) Point Selection**, and **(4) Application**.

Optimization

Zax's first phase enables transformations that reduce the mutation effort required to fuzz-through deeper code regions (e.g., sub-instruction profiling). Given a pre-specified optimization criteria (e.g., "decompose multi-byte conditional constraints"), it scans the target binary's control-flow graph to identify sections of interest; and for every match, it applies the relevant IR-level transformations. As such transformations alter control-flow, we apply them before further analyses that depend on the finalized control-flow graph.

Analysis

With the optimized control-flow graph in hand, Zax's second phase computes meta-characteristics (e.g., predecessor-successor, data-flow, and dominance relationships). We model this after existing compiler mechanisms [3, 29, 86], and to facilitate integration of other desirable analyses appearing in the literature [2, 127]. The extent of possible analyses depends on the rewriter's IR; for example, low-level IR's modeled after GCC's RTL [45] permit intuitive analysis to infer register liveness; and other IRs may support equivalent analyses which could be used instead, but if not, such algorithms are well-known [86] and could be added to support Zax.

Point Selection

ZAX's third phase aims to identify *where* in the program to instrument. Given the binary's full control-flow graph and meta-characteristic data (e.g., liveness, dominator trees), this phase enumerates all candidate basic blocks and culls those deemed unnecessary for future instrumentation. ZAX's CFG-aware instrumentation pruning capabilities facilitate easy implementation of compiler-based techniques described in section 4.2.

Application

Finally, ZAX's applies the desired instrumentation configuration (e.g., block or edge coverage tracking). A challenge is identifying *how* to instrument each location; ensuring correct execution requires precise handling of registers around instrumentation code—necessitating careful consideration of liveness. As a block's instrumentation can theoretically be positioned *anywhere* within it, liveness analysis also facilitates "best-fit" location ranking by quantity of free registers; and since restoring condition code registers (e.g., x86's `eflags`) is often costlier than others, we further prioritize locations where these are free. Thus, ZAX's efficiency-maximizing instrumentation insertion is comparable to that of modern compilers [45, 76]. Though our current prototype (section 4.5) targets AFL-style fuzzers, support for others is possible through new instrumentation configurations.

4.5 Extending Compiler-quality Transforms to Binary-only Fuzzing

I review successful compiler-based fuzzing approaches and identify impactful fuzzing performance and feedback-enhancing program transformations. As these transformations provably

improve compiler-based fuzzers they thus are desirable for closed-source targets; however, they are largely neglected due to current binary instrumenters' limitations.

Performance Transformation	
Single Successor-based Pruning	[41]
Dominator-based Pruning	[65]
Instrumentation Downgrading	[43]
Feedback Transformation	
Sub-instruction Profiling	[1, 41, 70, 119]
Context-sensitive Coverage	[22, 41]

Table 4.3: A catalog of Zafl-implemented compiler-quality fuzzing-enhancing program transformations and their compiler-based origins.

To show the power of Zafl in applying and layering transformations ad-hoc, we extend three performance- and two feedback-enhancing compiler-based transformations to binary-only fuzzing, shown in Table 4.3. Below details our implementations of these five transformations using Zafl.

4.5.1 Performance-enhancing Transformations

We leverage Zafl's Zax architecture in deploying three fuzzing performance-enhancing program transformations: **single successor** and **dominator-based** instrumentation pruning, and edge **instrumentation downgrading**. I describe our implementation of each below.

Single Successor Instrumentation Pruning

Recent fuzzing works leverage flow graph reducibility techniques [57, 121] to cut down instrumentation overhead [65]. We borrow AFL-Dyninst's omitting of basic blocks which are not their function's entry, but are the single successor to their parent block [61]. Intuitively,

these are guaranteed to be covered as they are preceded by unconditional transfer and thus, their instrumentation is redundant. Our implementation applies a meta-characteristic predecessor-successor analysis in Zax's Analysis phase; and a location selector during Point Selection to omit basic blocks accordingly.

Dominator Tree Instrumentation Pruning

Tikir and Hollingsworth [127] expand on single predecessor/successor pruning by evaluating control-flow dominator relationships. A node A "dominates" B if and only if every possible path to B contains A [2]. Dominator-aware instrumentation audits the control-flow graph's corresponding dominator tree to consider nodes that are a dominator tree leaf, or precede another node in control-flow but do not dominate it.

In line with our other CFG-aware pruning, we implement a dominator tree meta-characteristic in Zax's Analysis phase; and a corresponding selector within Point Selection. My analysis reveals this omits 30–50% of blocks from instrumentation. I elect to apply Tikir and Hollingsworth's algorithm because it balances graph reduction and analysis effort. Other alternative, more aggressive algorithms exist [2, 65], which I believe are also implementable in Zafl.

Edge Instrumentation Downgrading

CollAFL [43] optimizes AFL-style edge coverage by downgrading select blocks to faster (i.e., fewer-instruction) block coverage. At a high level, blocks with a single predecessor can themselves represent that edge, eliminating the instruction cost of hashing the start and end points. We implement edge downgrading using a meta-characteristic analysis based on linear flows in Zax's Analysis phase; and construct both edge- and block-coverage instrumentation

templates utilized in the Application phase. My numbers show that roughly 35–45% of basic blocks benefit from this optimization.

4.5.2 Feedback-enhancing Transformations

Recent compiler-based fuzzing efforts attain improved code-penetration power by considering finer-grained execution information [22, 41]. Below I detail our Zafl implementations of two prominent examples: **sub-instruction profiling** and **context-sensitive coverage** tracking.

Sub-instruction Profiling

Sub-instruction profiling breaks down complex conditional constraints into nested single-byte comparisons—allowing the fuzzer to track *progress* toward matching the entire constraint, and significantly decreasing the overall mutation effort. Compiler-based implementations (e.g., laf-Intel [1] and CmpCov [70]) replace comparisons with nested micro-comparisons; however, as the goal is to augment control-flow with nested conditionals that permit increased feedback, I observe it is equally effective to insert these *before* the original. We implement a binary-only sub-instruction profiling for (up to) 64-bit unsigned integer comparisons: in Zax's Optimization phase, we scan the IR for comparison mnemonics (i.e., `cmp`), and then insert a one-byte nested comparison per constraint byte. We further incorporate handling for division operators to help reveal divide-by-zero bugs.

Context-sensitive Coverage

Context sensitivity considers calling contexts to enable finer-grained coverage. For hash-indexing fuzzers like AFL, this merely requires that the hash index calculation additionally

incorporates a *context value*. Several LLVM-based efforts compute values at callsite-level [22] or function-level [41]. Though context values can assigned statically or obtained dynamically (e.g., from a stack trace), an easy solution is to create a global context variable which is updated on-the-fly: we create function-level context sensitivity by instrumenting each function with a random value, which at function entry/exit is `XOR`'d to a global context value that is used during edge hashing. We implement function-level context sensitivity in ZAX's Application phase. Callsite-level context sensitivity is also possible by adjusting where values are inserted.

4.6 Evaluation

My evaluation answers three high-level questions:

Q1: Does ZAFL enable compiler-style program transformations while maintaining performance?

Q2: Do performant fuzzing-enhancing program transformations increase binary-only fuzzing's effectiveness?

Q3: Does ZAFL support *real-world, complex* targets?

I first perform an evaluation of ZAFL against the leading binary-only fuzzing instrumenters AFL-Dyninst and AFL-QEMU on the LAVA-M benchmark corpus [34]. Second, to see if LAVA-M results hold for real-world programs, I expand my evaluation to eight popular programs well-known to the fuzzing literature, selecting older versions *known to contain bugs* to ensure self-evident comparison. Third, I evaluate these instrumenters' fuzzing overhead across each. Fourth, I evaluate ZAFL alongside AFL-Dyninst and AFL-QEMU in fuzzing

five varied *closed-source* binaries. Fifth, I test ZAFL's support for 42 open- and closed-source programs of varying size, complexity, and platform. Finally, I use industry-standard reverse-engineering tools as ground-truth to assess ZAFL's precision.

4.6.1 Evaluation-wide Instrumenter Setup

I evaluate ZAFL against the fastest-available binary-only fuzzing instrumenters; I thus omit AFL-PIN [62, 97, 126] and AFL-DynamoRIO [60, 114, 128] variants as their reported overheads are much higher than AFL-Dyninst's and AFL-QEMU's; and Intel PT [66] as it does not support instrumentation (Table 4.2). I configure AFL-Dyninst and AFL-QEMU with recent updates which purportedly increase their fuzzing performance by 2–3x and 3–4x, respectively. I detail these below in addition to my setup of ZAFL.

AFL-Dyninst: A recent AFL-Dyninst update [61] adds two optimizations which increase performance by 2–3x: (1) CFG-aware "single successor" instrumentation pruning; and (2) two optimally-set Dyninst BPatch API settings (`setTrampRecursive` and `setSaveFPR`).[1] I discovered three other performance-impacting BPatch settings (`setLivenessAnalysis`, `setMergeTramp`, and `setInstrStackFrames`). For fairness I apply the fastest-possible AFL-Dyninst configurations to all benchmarks; but for `setLivenessAnalysis` I am restricted to its non-optimal setting on all as they otherwise crash; and likewise for `setSaveFPR` on `sfconvert` and `tcpdump`.

AFL-QEMU: QEMU attempts to optimize its expensive block-level translation with caching, enabling translation-free *chaining* of directly-linked fetched-block sequences. Until recently, AFL-QEMU invoked its instrumentation via trampoline *after* translation—rendering block

[1] This AFL-Dyninst update [61] also adds a third optimization that replaces Dyninst-inserted instructions with a custom, optimized set. However, in addition to having only a negligible performance benefit according to its author, its current implementation is experimental and crashes each of my benchmarks. For these reasons I omit it in my experiments.

chaining incompatible as skipping translation leaves some blocks uninstrumented, potentially missing coverage. A newly-released AFL-QEMU update [11] claims a 3–4x performance improvement through enabling support for chaining by instead applying instrumentation *within* translated blocks. To ensure best-available AFL-QEMU performance I apply this update in all experiments.

Zafl: To explore the effects of compiler-quality fuzzing-enhancing transformation on binary-only fuzzing I instrument benchmarks with all transformations shown in Table 4.3.

4.6.2 LAVA-M Benchmarking

For my initial crash-finding evaluation I select the LAVA-M corpus as it provides ground-truth on its programs' bugs. Below I detail my evaluation setup and results.

Benchmarks

I compile each benchmark with Clang/LLVM before instrumenting with AFL-Dyninst and Zafl; for AFL-QEMU I simply run compiled binaries in AFL using "QEMU mode". As fuzzer effectiveness on LAVA-M is sensitive to starting seeds and/or dictionary usage, I fuzz each instrumented binary per four configurations: empty and default seeds both with and without dictionaries. I build dictionaries as instructed by one of LAVA-M's authors [33].

Experimental Setup and Infrastructure

I adopt the standard set by other LAVA-M evaluations [7, 110, 143] and fuzz each instrumented binary for five hours with the coverage-guided fuzzer AFL [146]; each for five trials per the four seed/dictionary configurations. All instrumenters are configured as detailed

in subsection 4.6.1. To maintain performance neutrality, I distribute trials across eight VM's spanning two Ubuntu 16.04 x86-64 systems with 6-core 3.50GHz Intel Core i7-7800x CPU's and 64GB RAM. Each VM runs in VirtualBox with 6GB RAM and one core allocated.

Data Processing and Crash Triage

I log both the number of AFL-saved crashes and test cases processed (i.e., *total* − *hang* − *calibration* − *trim* executions); and in post-processing match each crash to a specific number of test cases seen—allowing me to pinpoint *when* each crash occurred in its trial. I then triage all crashes and create ⟨`crash_id, testcases_done, triage_data`⟩ triples; and apply set operations to obtain the unique crashes over test cases done (i.e., ⟨`triaged_crashes, testcases_done`⟩). For LAVA-M I triage solely by its benchmarks' self-reported bug ID's.

I compute the average unique crashes, total processed and queued test cases for all instrumenter-benchmark trial groupings. To show Zafl's effectiveness, I report its mean relative increase for all three metrics per-trial group, and geometric mean relative increases among all benchmarks. Following Klees et al.'s [75] recommendation, to determine if Zafl's gains are statistically significant, I compute a Mann-Whitney U-test with a 0.05 significance level, and report the geometric mean p-values across all benchmarks.

Results

I do not include Zafl's context sensitivity in my LAVA-M trials as I observe it slightly inhibits effectiveness (∼2%), likely due to LAVA-M's focus on a specific type of synthetic bug (i.e., "magic bytes"). This also enhances the distinction on the impact of Zafl's sub-instruction profiling transformation based on number of queued (i.e., coverage-increasing) test cases. Table 4.4 shows Zafl's mean relative increase in triaged crashes, total and queued

test cases over AFL-Dyninst and AFL-QEMU per configuration.

Binary	**Seed, Dictionary**	**ZAFL vs. AFL-Dyninst**			**ZAFL vs. AFL-QEMU**		
		rel. crash	rel. total	rel. queue	rel. crash	rel. total	rel. queue
`base64`	default, none	1.00	13.71	1.70	1.00	13.71	1.58
	default, dict.	1.00	13.70	1.70	1.00	13.71	1.58
	empty, none	✗	1.34	3.16	✗	1.67	2.88
	empty, dict.	2.46	1.33	2.80	1.05	2.57	2.61
`md5sum`	default, none	✗	0.88	45.22	✗	2.22	4.39
	default, dict.	5.52	0.94	32.17	1.00	1.88	2.15
	empty, none	✗	1.01	45.54	✗	2.15	4.39
	empty, dict.	4.00	0.96	77.77	0.87	1.91	2.22
`uniq`	default, none	1.00	1.62	1.37	1.00	1.98	1.21
	default, dict.	5.75	1.04	2.39	7.67	1.23	1.64
	empty, none	✗	1.97	4.37	✗	3.92	3.71
	empty, dict.	2.23	1.55	2.60	1.04	2.15	2.43
`who`	default, none	1.00	1.32	27.07	1.00	2.44	21.86
	default, dict.	3.78	1.18	40.24	3.68	1.7	36.36
	empty, none	1.00	4.13	12.62	1.00	4.20	9.50
	empty, dict.	1.24	1.15	11.22	2.54	10.00	15.74
Mean Rel. Increase		**+96%**	**+78%**	**+751%**	**+42%**	**+203%**	**+296%**
Mean MWU Score		**0.023**	**0.022**	**0.005**	**0.039**	**0.007**	**0.005**

Table 4.4: ZAFL's LAVA-M mean bugs and total/queued test cases relative to AFL-Dyninst and AFL-QEMU. I report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance). ✗ = ZAFL finds crashes while competitor finds zero.

ZAFL versus AFL-Dyninst: Across all 16 configurations ZAFL executes 78% more test cases than AFL-Dyninst and either matches or beats it with 96% more crashes on average, additionally finding crashes in four cases where AFL-Dyninst finds none. As I observe Mann-Whitney U p-values (0.005–0.023) below the 0.05 threshold I conclude this difference in effectiveness is statistically significant. Though ZAFL averages slightly fewer (4–12%) test cases on `md5sum` this is not to its disadvantage: ZAFL queues 3100–7600% more test cases and finds well over 300% more crashes, thus revealing the value of its control-flow-optimizing program transformations.

Zafl versus AFL-QEMU: Zafl matches or surpasses AFL-QEMU among 15 benchmark configurations, averaging 42% more crashes and 203% more test cases seen. As with AFL-Dyninst, Zafl successfully finds crashes in four cases for which AFL-QEMU finds none. Additionally, the Mann-Whitney U p-values (0.005–0.039) reveal a statistically significant difference between AFL-QEMU and Zafl. Zafl finds 13% fewer crashes relative to AFL-QEMU on `md5sum` with empty seeds and dictionary, but as Zafl's queue is 91% larger, I believe this specific seed/dictionary configuration and Zafl's transformations result in a "burst" of hot paths, which the fuzzer struggles to prioritize. Such occurrences are rare given Zafl's superiority in other trials, and likely correctable through orthogonal advancements in fuzzing path prioritization [14, 26, 77, 148].

To my surprise, AFL-QEMU finds more crashes than AFL-Dyninst despite executing the least test cases. This indicates that Dyninst's instrumentation, while faster, is less sound than QEMU's in important ways. Achieving compiler-quality instrumentation requires upholding both performance *and* soundness, which neither QEMU nor Dyninst achieve in concert, but Zafl does (see subsection 4.6.5).

Zafl versus AFL-LLVM: To gain a sense of whether Zafl's transformation is comparable to existing compiler-based implementations, I ran Zafl alongside the the analogous configuration of AFL's LLVM instrumentation with its INSTRIM [65] and laf-Intel [1] transformations applied. Results show that the two instrumentation approaches result in **statistically indistinguishable** (MWU p-value 0.10) bug finding performance.

4.6.3 Fuzzing Real-world Software

Though my LAVA-M results show compiler-quality fuzzing-enhancing program transformations are beneficial to binary-only fuzzing, it is an open question as to whether this carries

over to real-world programs. I therefore expand my crash-finding evaluation to eight diverse, real-world benchmarks and extend all trials to 24 hours as per the standard set by Klees et al. [75]. I further show that ZAFL achieves compiler-quality performance in a coverage-tracing overhead comparison of all three instrumenters.

Benchmarks

To capture the diversity of real-world software I select eight binaries of varying type, size, and libraries which previously appear in the fuzzing literature: `bsdtar`, `cert-basic`, `clean_text`, `jasper`, `readelf`, `sfconvert`, `tcpdump`, and `unrtf`. I intentionally select older versions known to contain *AFL-findable* bugs to facilitate a self-evident bug-finding comparison. Statistics for each (e.g., package, size, number of basic blocks) are listed in Table 4.8.

Experimental Setup and Infrastructure

In both crash-finding and overhead experiments I configure instrumenters and binaries as described in subsection 4.6.1 and section 4.6.2, and utilize either AFL- or developer-provided seed inputs in fuzzing evaluations. For crash-finding, I fuzz all instrumented binaries with AFL on a cluster for 8×24-hour trials each and to evaluate overhead, I perform 5×24-hour trials on my LAVA-M experiment infrastructure (section 4.6.2).

Real-world Crash-finding

I apply all ZAFL-implemented transformations (Table 4.3) to all eight binaries, but omit context sensitivity for `clean_text` as it otherwise consumes 100% of its coverage map. Triage is performed as in section 4.6.2 but is based on stack hashing as seen in the literature [75, 82,

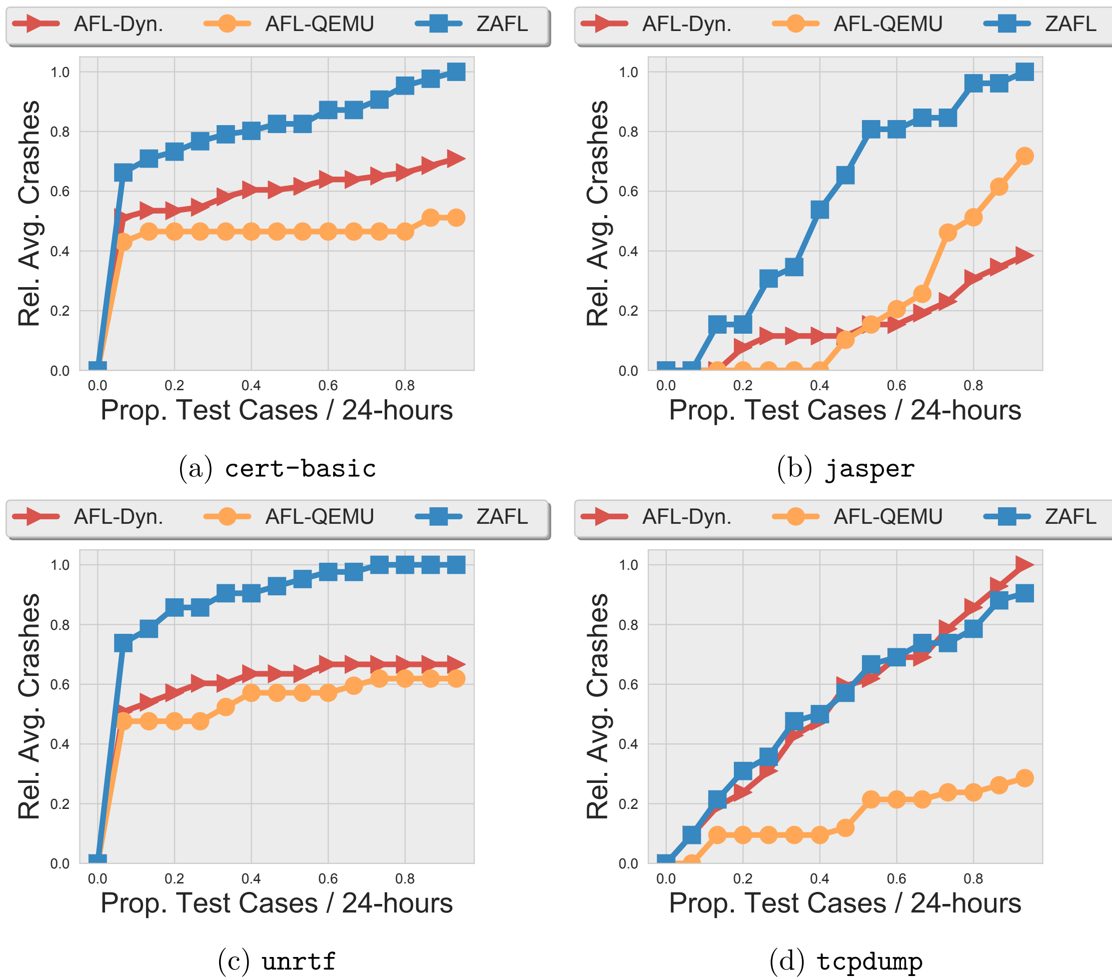


Figure 4.2: Real-world software fuzzing unique triaged crashes averaged over 8×24-hour trials.

Binary	ZAFL vs. AFL-Dyninst			ZAFL vs. AFL-QEMU		
	rel. crash	rel. total	rel. queue	rel. crash	rel. total	rel. queue
`bsdtar`	0.80	1.25	1.06	8.00	2.59	2.79
`cert-basic`	1.41	1.79	4.99	1.95	3.05	4.51
`clean_text`	1.25	1.48	1.68	6.25	3.23	1.93
`jasper`	2.60	2.70	2.30	1.39	2.05	1.67
`readelf`	1.00	1.03	3.52	1.00	3.44	5.60
`sfconvert`	1.30	0.96	4.04	1.18	0.90	3.30
`tcpdump`	0.90	1.44	2.68	3.17	4.95	4.99
`unrtf`	1.50	1.78	36.1	1.62	2.51	35.5
Mean Rel. Increase	**+26%**	**+48%**	**+260%**	**+131%**	**+159%**	**+337%**
Mean MWU Score	**0.018**	**0.001**	**0.001**	**0.002**	**0.001**	**0.001**

Table 4.5: ZAFL's real-world software mean triaged crashes and total/queued test cases rel. to AFL-Dyninst and AFL-QEMU. I report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance).

99].[2] Table 4.5 shows ZAFL-instrumented fuzzing crash-finding as well as total and queued test cases relative to AFL-Dyninst and AFL-QEMU. I further report the geometric mean Mann-Whitney U significance test p-values across all metrics.

ZAFL versus AFL-Dyninst: My results show ZAFL averages 26% more real-world crashes and 48% more test cases than AFL-Dyninst in 24 hours. Though ZAFL finds 10–20% fewer on `bsdtar` and `tcpdump`, the raw differences amount to only 1–2 crashes, suggesting that it and AFL-Dyninst converge on these two benchmarks (as shown in Figure 4.2d). Likewise for `readelf` my triage reveals two unique crashes across all trials, both found by all three instrumenters. For all others ZAFL holds a lead (as shown in Figure 4.2), averaging 61% more crashes. Given the Mann-Whitney U p-values (0.001–0.018) below the 0.05 significance level, I conclude that ZAFL's compiler-quality transformations bear a statistically significant advantage over AFL-Dyninst.

[2] In stack hashing I consider both function names and lines; and condense recursive calls as they would otherwise over-approximate bug counts.

Zafl versus AFL-QEMU: While Zafl surpasses AFL-QEMU's LAVA-M crash-finding by 42%, Zafl's real-world crash-finding is an even higher 131%. Apart from the two `readelf` bugs found by all three instrumenters, Zafl's fuzzing-enhancing program transformations and 159% higher execution rate allow it to hone-in on more crash-triggering paths on average. As with AFL-Dyninst, comparing to AFL-QEMU produces Mann-Whitney U p-values (0.001–0.002) which prove Zafl's increased effectiveness is statistically significant. Furthermore the disparity between AFL-QEMU's LAVA-M and real-world crash-finding suggests that increasingly-complex binaries heighten the need for more powerful binary rewriters.

Real-world Coverage-tracing Overhead

For my coverage-tracing overhead evaluation I follow established practice [89]: I collect 5×24-hour test case dumps per benchmark; instrument a forkserver-only "baseline" (i.e., no coverage-tracing) version of each benchmark; log every instrumented binary's coverage-tracing time for each test case per dump; apply 30% trimmed-mean de-noising on the execution times per instrumenter-benchmark pair; and scale the resulting overheads relative to baseline.

I compare Zafl to AFL-Dyninst, AFL-QEMU, and to the compiler- and assembler-based instrumentation available in AFL [146]. I assess all aspects of Zafl's performance: (1) its baseline forkserver-only rewritten binary overhead (Zafl-fsrvr); and instrumentation overheads (2) with no transformations (Zafl-none), (3) only performance-enhancing transformations (Zafl-perf), and (4) all (Table 4.3) transformations (Zafl-all). I additionally compute geometric mean Mann-Whitney U p-values of both Zafl-none's and Zafl-all's execution times compared to those of compiler and assembler instrumentation, AFL-Dyninst, and AFL-QEMU among all benchmarks.

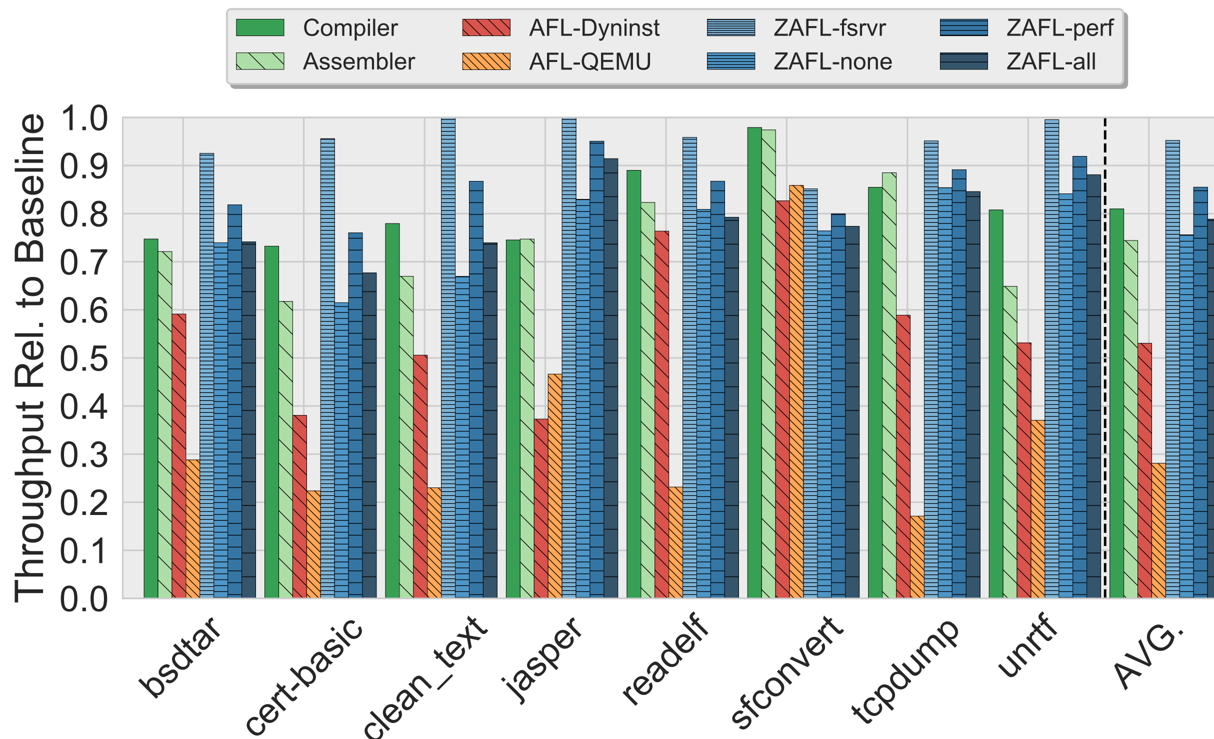


Figure 4.3: Compiler, assembler, AFL-Dyninst, AFL-QEMU, and ZAFL fuzzing instrumentation performance relative to baseline (higher is better).

Figure 4.3 displays the instrumenters' relative overheads. On average, ZAFL-FSRVR, ZAFL-NONE, ZAFL-PERF, and ZAFL-ALL obtain overheads of 5%, 32%, 17%, and 27%, while compiler and assembler instrumentation average 24% and 34%, and AFL-Dyninst and AFL-QEMU average 88% and 256%, respectively. Thus, even ZAFL with *all* fuzzing-enhancing transformations approaches compiler performance.

ZAFL versus AFL-Dyninst: I observe ZAFL performs slightly worse on `sfconvert` as it has the fewest basic blocks by far I believe our rewriting overhead is more pronounced on such tiny binaries. Other results suggest that this case is pathological. Even ZAFL's most heavyweight configuration (ZAFL-ALL) incurs 61% less average overhead than AFL-Dyninst, even though this comparison includes ZAFL's performance-enhancing transformations. If omitted, this still leaves ZAFL ahead of AFL-Dyninst—which, too, benefits from performance-enhancing

single successor-based pruning. Comparing the execution times of ZAFL-NONE and ZAFL-ALL to AFL-Dyninst's yields mean Mann-Whitney U p-values of 0.020–0.023. As these are below 0.05, suggesting that ZAFL, both with- and without-transformations, achieves statistically better performance over AFL-Dyninst.

ZAFL versus AFL-QEMU: Though AFL-QEMU's block caching reduces its overhead from previous reports [89], ZAFL outperforms it with nearly 229% less overhead. Interestingly, AFL-QEMU beats AFL-Dyninst on `jasper`, consistent with the relative throughput gains in Table 4.5. Thus, while it appears some binary characteristics are better-suited for dynamic vs. static rewriting, existing instrumenters do not match ZAFL's performance across all benchmarks. My Mann-Whitney U tests reveal that both ZAFL-NONE and ZAFL-ALL obtain p-values of 0.012, suggesting that ZAFL achieves statistically better performance over AFL-QEMU.

Comparing ZAFL to Compiler Instrumentation: On average, compared to a forkserver-only binary, ZAFL incurs a baseline overhead of 5% just for adding rewriting support to the binary; tracing all code coverage increases overhead to 32%; optimizing coverage tracing using graph analysis reduces overhead to 20%; and applying *all* fuzzing-enhancing program transformations brings overhead back up to 27%. These overheads are similar to the 24% overhead of AFL's compiler-based instrumentation, and slightly better than AFL's assembler-based trampolining overhead of 34%. Comparing ZAFL-NONE and ZAFL-ALL to compiler instrumentation yields mean Mann-Whitney U p-values ranging 0.12–0.18 which, being larger than 0.05, suggests that ZAFL **is indistinguishable from compiler-level performance**.

4.6.4 Fuzzing Closed-source Binaries

To evaluate whether ZAFL's improvements extend to *true* binary-only use cases, I expand my evaluation with five diverse, closed-source binary benchmarks. My results show that ZAFL's compiler-quality instrumentation and speed help reveal more unique crashes than AFL-Dyninst and AFL-QEMU across all benchmarks. I further conduct several case studies showing that ZAFL achieves far shorter time-to-bug-discovery compared to AFL-Dyninst and AFL-QEMU.

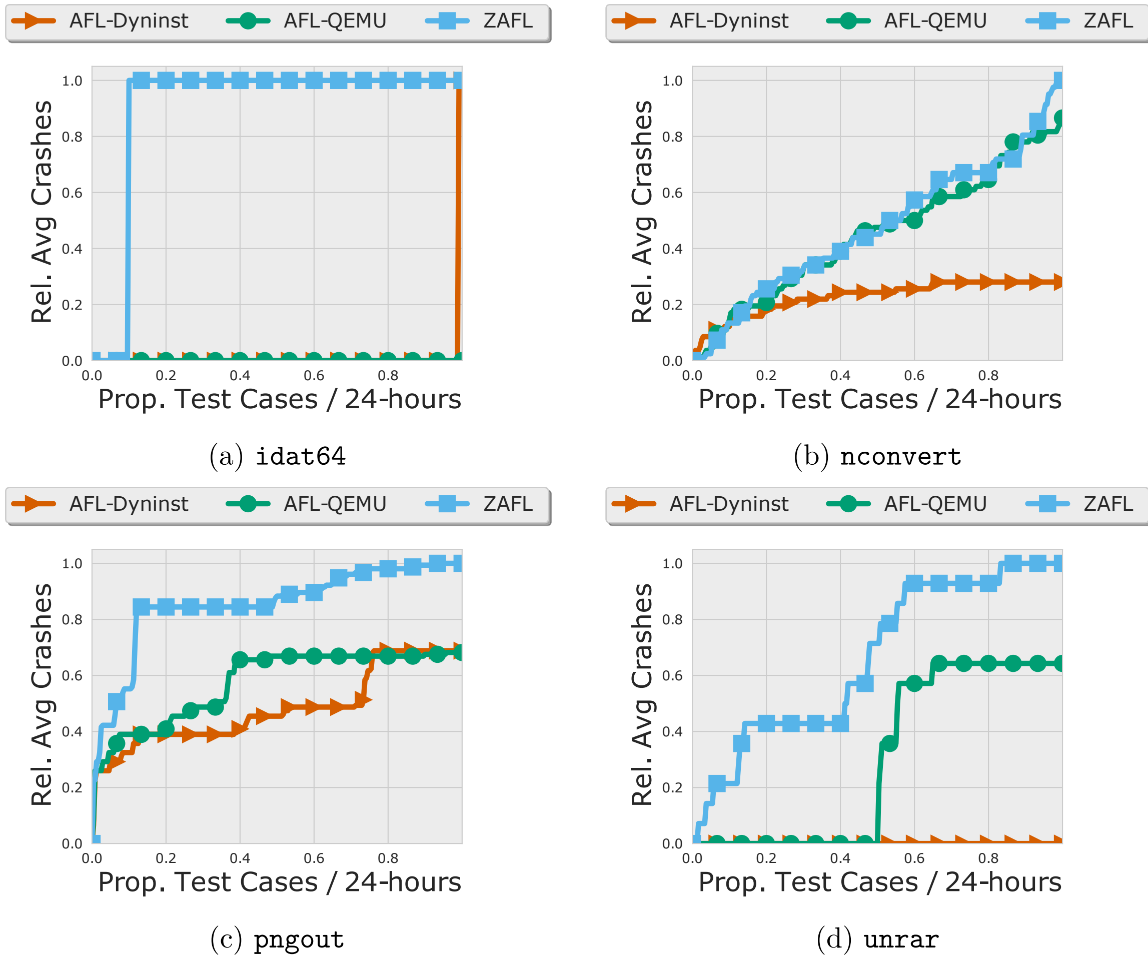
Benchmarks

I drill-down the set of all closed-source binaries I tested with ZAFL (Table 4.9) into five *AFL-compatible* (i.e., command-line interfacing) benchmarks: `idat64` from IDA Pro, `nconvert` from XNView's NConvert, `nvdisasm` from NVIDIA's CUDA Utilities, `pngout` from Ken Silverman's PNGOUT, and `unrar` from RarLab's RAR. Table 4.9 lists the key features of each benchmark.

Closed-source Crash-finding

I repeat the evaluation from section 4.6.4, running five 24-hour experiments per configuration. My results (mean unique triaged crashes, total and queued test cases, and MWU p-scores) among all benchmarks are shown in Table 4.6; and plots of unique triaged crashes over time are shown in Figure 4.4.

ZAFL versus AFL-Dyninst: Despite AFL-Dyninst being faster on `idat64`, `nconvert`, `nvdisasm`, and `unrar`, ZAFL averages a statistically-significant (mean MWU p-value of 0.036) 55% higher crash-finding. I believe AFL-Dyninst's speed, small queues, and lack of crashes in `unrar` are due to it missing significant parts of these binaries, as my own testing with its



(a) `idat64` (b) `nconvert`

(c) `pngout` (d) `unrar`

Figure 4.4: Closed-source binary fuzzing unique triaged crashes averaged over 5×24-hour trials.

	ZAFL vs. AFL-Dyninst			ZAFL vs. AFL-QEMU		
Binary	rel. crash	rel. total	rel. queue	rel. crash	rel. total	rel. queue
`idat64`	1.000	0.789	2.332	✗	1.657	1.192
`nconvert`	3.538	0.708	48.140	1.095	1.910	1.303
`nvdisasm`	1.111	0.757	1.484	1.111	0.578	1.252
`pngout`	1.476	5.842	1.380	1.476	3.419	1.023
`unrar`	✗	0.838	6.112	2.000	1.284	1.249
Mean Rel. Increase	**+55%**	**+16%**	**+326%**	**+38%**	**+52%**	**+20%**
Mean MWU Score	**0.036**	**0.041**	**0.009**	**0.082**	**0.021**	**0.045**

Table 4.6: ZAFL's closed-source binary mean triaged crashes and total/queued test cases relative to AFL-Dyninst and AFL-QEMU. I report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance). ✗ = ZAFL finds crashes while competitor finds zero.

graph-pruning off shows it leaves over 50% of basic blocks uninstrumented for all but `pngout`. I conclude that ZAFL's support for complex, stripped binaries brings a decisive advantage over existing tools like AFL-Dyninst.

ZAFL versus AFL-QEMU: ZAFL's speed and transformations enable it to average 38% more triaged crashes and 52% more test cases than AFL-QEMU. While ZAFL offers a statistically significant improvement in throughput for four benchmarks (mean MWU p-value of 0.021), I posit that its slower speed on `nvdisasm` is due to AFL prioritizing slower paths: AFL's logs show ZAFL's initial speed is over 2×AFL-QEMU's (2500 execs/s vs. 1200), but it fluctuates around 5 execs/s for much of the campaign afterwards. Though the crash-finding gap between ZAFL and AFL-QEMU is not overwhelming, ZAFL successfully uncovers a heap overread crash in `idat64`—while AFL-QEMU finds nothing.

Bug-finding Case Study

Following additional manual triage with binary-level memory error checkers (e.g., QASan [40] and Dr. Memory [19]), I compare the time-to-discovery (TTD) for five closed-source binary bugs found by Zafl, AFL-Dyninst, or AFL-QEMU: a heap overflow in `nconvert`, a stack overflow in `unrar`, a heap use-after-free and heap overflow in `pngout`, and a heap overread in `idat64`'s `libida64.so`.

Error Type	**Location**	**AFL-Dyninst**	**AFL-QEMU**	**Zafl**
heap overflow	`nconvert`	✗	18.3 hrs	12.7 hrs
stack overflow	`unrar`	✗	12.3 hrs	9.04 hrs
heap overflow	`pngout`	12.6 hrs	6.26 hrs	1.93 hrs
use-after-free	`pngout`	9.35 hrs	4.67 hrs	1.44 hrs
heap overread	`libida64.so`	23.7 hrs	✗	2.30 hrs
Zafl Mean Rel. Decrease		**-660%**	**-113%**	

Table 4.7: Mean time-to-discovery of closed-source binary bugs found for AFL-Dyninst, AFL-QEMU, and Zafl over 5×24-hour fuzzing trials. ✗ = bug is not reached in any trials for that instrumenter configuration.

Table 4.7 reports the geometric mean TTD among all five bugs for all three instrumenters. I observe that, on average, Zafl finds these bugs 660% faster than AFL-Dyninst, and 113% faster than AFL-QEMU. Thus, Zafl's balance of compiler-quality transformation and performance lends a valuable asset to bug-finding in closed-source code.

4.6.5 Scalability and Precision

I recognize the fuzzing community's overwhelming desire for new tools that support many types of software—with a growing emphasis on more complex, real-world targets. But for a static rewriter to meet the needs of the fuzzing community, it must also achieve high

precision with respect to compiler-generated code. This section examines Zafl's scalability to binaries beyond my evaluation benchmarks, as well as key considerations related to its static rewriting precision.

Scalability

I instrument and test Zafl on a multitude of popular real-world binaries of varying size, complexity, source availability, and platform. I focus on Linux and Windows as these platforms' binary formats are common high-value targets for fuzzing. All binaries are instrumented with Zafl's AFL-like configuration; I do the same for Windows binaries using Zafl's cross-instrumentation support. I test instrumented binaries either with our automated regression test suite (used throughout Zafl's development); or by manually running the application (for Windows) or testing the instrumentation output with `afl-showmap` [146] (for Linux).

I verify Zafl achieves success on 33 open-source Linux and Windows binaries, shown in Table 4.8. To confirm Zafl's applicability to *true* binary-only use cases, I expand my testing with 23 closed-source binaries from 19 proprietary and commercial applications, listed in Table 4.9. In summary, my findings show that Zafl can instrument Linux and Windows binaries of varying size (e.g., 100K–100M bytes), complexity (100–1M basic blocks), and characteristics (open- and closed-source, PIC and PDC, and stripped binaries).

Liveness-aware Optimization

As discussed in subsection 4.3.2, register liveness analysis enables optimized instrumentation insertion for closer-to-compiler-level speed. While liveness *false positives* introduce overhead from the additional instructions needed to save/restore registers, liveness *false negatives* may leave live registers erroneously overwritten—potentially breaking program functionality. If

Zafl's liveness analysis (section 4.4.2) cannot guarantee correctness, it conservatively halts this optimization to avoid false negatives, and instead safely inserts code at basic block starts.

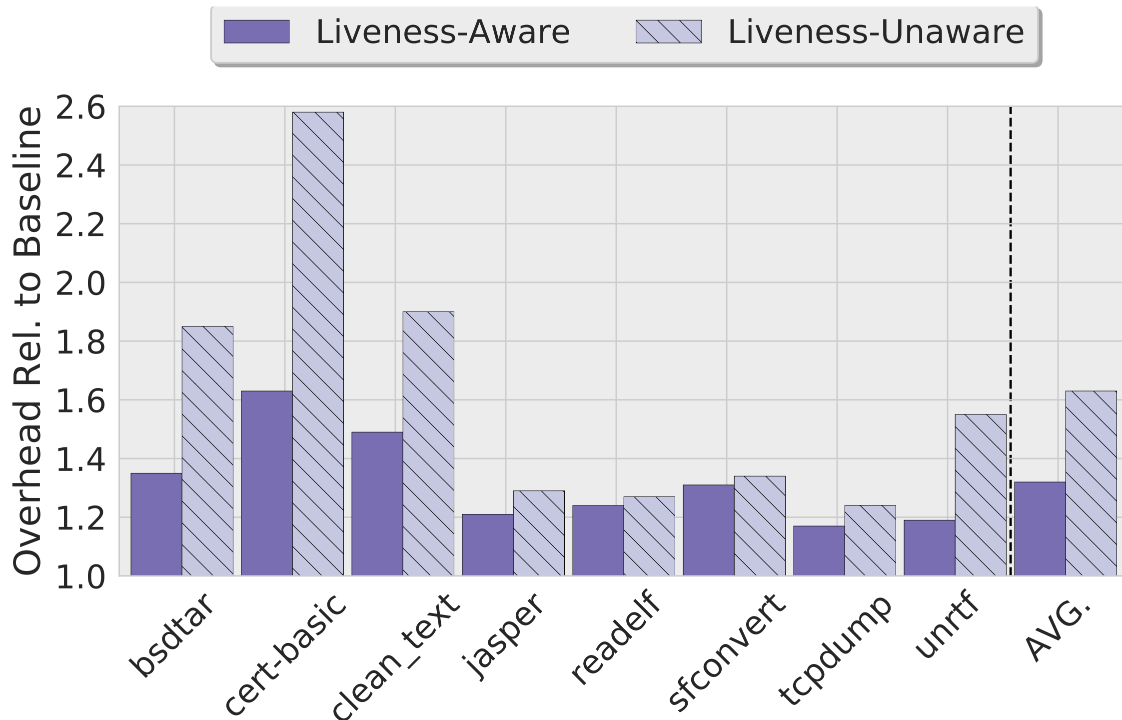


Figure 4.5: A comparison of Zafl's runtime overhead with and without register liveness-aware instrumentation optimization (lower is better).

To assess the impact of skipping register liveness-aware optimization, I replicate my overhead evaluation (section 4.6.3) to compare Zafl's speed with/without liveness-aware instrumentation. As Figure 4.5 shows, liveness-*un*aware Zafl faces 31% more overhead across all eight benchmarks. While 13–16% slower than AFL-Dyninst on `bsdtar` and `sfconvert`, Zafl's unoptimized instrumentation still averages 25% and 193% less overhead than AFL-Dyninst and AFL-QEMU, respectively. Thus, even in the worst case Zafl generally outperforms other binary-only fuzzing instrumenters.

As Table 4.8 and Table 4.9 show, I successfully apply liveness-aware instrumentation for

all 44 Linux benchmarks. I posit that with further engineering, the same robustness is achievable for Windows binaries.

Instruction Recovery

Recovery of the original binary's full instructions is paramount to static rewriting. It is especially important for binary-only fuzzing, as *false positive* instructions misguide coverage-guidance; while *false negatives* introduce coverage blind-spots or break functionality. Further, precise instruction recovery heads fuzzing-enhancing transformation, as it is necessary to know where/how to modify code (e.g., targeting `cmp`'s for sub-instruction profiling (section 4.5.2)).

I evaluate ZAFL's instruction recovery using ground-truth disassemblies of binary `.TEXT` sections generated by `objdump`, which is shown to achieve ~100% accuracy [5] (specifically, I use the version shipped in LLVM-10 [76]). To see how ZAFL fairs with respect to the state-of-the-art in binary analysis, I also evaluate disassemblies of the commercial tools IDA Pro 7.1 and Binary Ninja 1.1.1259. As all three only recover instructions they deem "reachable", I compute false negative recovery rates from the mean number of unique unrecovered instructions that are *actually* reached among five 24-hour fuzzing campaigns per benchmark.

Table 4.10 lists the total instructions; and total and reached unrecovered instructions per my five closed-source benchmarks.[3] As I observe zero false positives for any tool on any benchmark, I focus only on false negatives. Though all three achieve near-perfect accuracy, ZAFL is the only to maintain a 0% false negative rate among all benchmarks, as IDA and Binary Ninja erroneously unrecover an average of 0–0.68% of instructions. While static rewriting is fraught with challenges—many of which require further engineering work to overcome

[3] I omit results for my eight open-source benchmarks as all three tools achieve a 0% false negative instruction recovery rate on each.

(subsection 4.7.3)—these results suggest that Zafl's *common-case* instruction recovery is sound.

Control-flow Recovery

Preserving the original binary's control-flow is critical to fuzzing's coverage-guidance. Excessive *false positives* add noise that misguide fuzzing or overwhelm its seed scheduling processes; while *false negatives* may cause fuzzing to overlook entire code regions or bug-triggering paths. To examine Zafl's control-flow recovery, I run all test cases generated over five 24-hour trials for my eight open-source benchmarks on both a Zafl- and a ground-truth LLVM-instrumented binary, and log when each report new coverage.

As Table 4.11 shows, Zafl's coverage identification is near-identical to LLVM's: achieving 97.3% sensitivity, ∼100% specificity, and ∼100% accuracy. While Zafl encounters some false positives, they are so infrequent (1–20 test cases out of 1–20 million) that the total noise is negligible. In investigating false negatives, I see that in only 7/40 fuzzing campaigns do missed test cases precede bug-triggering paths; however, further triage reveals that Zafl eventually finds replacement test cases, thus, Zafl reaches every bug reached by LLVM. Thus, I conclude that Zafl succeeds in preserving the control-flow of compiler-generated code.

4.7 Limitations

Below I briefly discuss limitations unique to Zafl, and others fundamental to static binary rewriting.

4.7.1 Improving Baseline Performance

My performance evaluation section 4.6.3 shows Zafl's baseline (i.e., non-tracing) overhead is around 5%. I believe that our rewriter's code layout algorithm is likely the biggest contributing factor to performance and have since tested experimental optimizations that bring baseline overhead down to ∼1%. But as Zafl's full fuzzing performance is already near modern compiler's, I leave further optimization and the requisite re-evaluation to future work.

4.7.2 Supporting New Architectures, Formats, and Platforms

Our current Zafl prototype is limited to x86-64 C/C++ binaries. As our current static rewriting engine handles both 32- and 64-bit x86 and ARM binaries (as well as prototype 32-bit MIPS support), I believe supporting these in Zafl is achievable with future engineering work.

Extending to other compiled languages similarly depends on the rewriter's capabilities. We have some experimental success for Go/Rust binaries, but more Zafl-side engineering is needed to achieve soundness. I leave instrumenting non-C/C++ languages for future work.

While Zafl is engineered with Linux targets in mind, my evaluation shows it also supports many Windows applications; few other static binary rewriters support Windows binaries. Though we face some challenges in precise code/data disambiguation and at this time are restricted to Windows 7 64-bit PE32+ formats, I expect that with future rewriter-level enhancements, Zafl will achieve broader success across other Windows binary formats and versions.

4.7.3 Static Rewriting's Limitations

Though static rewriting's speed makes it an attractive choice over dynamic translation for many binary-only use cases and matches what compilers do, static rewriting normally fails on software crafted to thwart reverse engineering. Two such examples are code obfuscation and digital rights management (DRM) protections—both of which, while uncommon, appear in many proprietary and commercial applications. While neither Zafl nor its rewriter currently support obfuscated or DRM-protected binaries, a growing body of research is working toward overcoming these obstacles [12, 140]. Thus, I believe that with new advances in binary deobfuscation and DRM-stripping, Zafl will be able to bring performant binary-only fuzzing to high-value closed-source targets like Dropbox, Skype, and Spotify.

Another grey area for static binary rewriters is deprecated language constructs. For example, C++'s dynamic exception specification—obsolete as of C++11—is unsupported in Zafl and simply ignored. I recognize there are trade-offs between static binary rewriting generalizability and precision, and leave addressing such gaps as future work.

Most modern static binary rewriters perform their core analyses—disassembly, code/data disambiguation, and indirect branch target identification—via third-party tools like Capstone [100] and IDA [54], consequently inheriting their limitations. For example, if the utilized disassembler is not up-to-date with the latest x86 ISA extension, binaries containing such code cannot be fully interpreted. I posit that trickle-down dependency limitations are an inherent problem to modern static binary rewriting; and while perfection is never guaranteed [84, 106], most common roadblocks are mitigated with further heuristics or engineering.

4.8 Related Work

Below I discuss related works in orthogonal areas static rewriting, fuzzing test case generation, hybrid fuzzing, and emergent fuzzing transformations.

4.8.1 Static Binary Rewriting

Static rewriters generally differ by their underlying methodologies. Uroboros [135], Ramblr [134], and RetroWrite [32] reconstruct binary assembly code "reassembleable" by compilers. Others translate directly to compiler-level intermediate representations (IR); Hasabnis et. al [55] target GCC [45] while McSema [31], SecondWrite [4], and dagger [16] focus on LLVM IR. GTIRB [52] and Zipr [64] implement their own custom IR's. I believe static rewriters with robust, low-level IR's are best-suited to supporting ZAFL.

4.8.2 Improving Fuzzing Test Case Generation

Research continues to improve test case generation from many perspectives. Input data-inference (e.g., Angora [22], VUzzer [104], TIFF [67]) augments mutation with type/shape characteristics. Other works bridge the gap between naive- and grammar-based fuzzing with models inferred statically (e.g., Shastry et. al [109], Skyfire [132]) or dynamically (e.g., pFuzzer [83], NAUTILUS [8], Superion [133], AFLSmart [99]). Such approaches mainly augment fuzzing at the mutator-level, and thus complement ZAFL's compiler-quality instrumentation in binary-only contexts.

Another area of improvement is path prioritization. AFLFast [14] allocates mutation to test cases exercising deep paths. FairFuzz [77] focuses on data segments triggering rare basic blocks. VUzzer [104] assigns deeper blocks high scores to prioritize test cases reaching them;

and QTEP [136] similarly targets code near program faults. Zafl's feedback-enhancing transformations result in greater path discovery, thus increasing the importance of smart path prioritization.

4.8.3 Hybrid Fuzzing

Many recent fuzzers are hybrid: using coverage-guided fuzzing for most test cases but sparingly invoking more heavyweight analyses. Angora [22] uses taint tracking to infer mutation information, but runs all mutatees in the standard fuzzing loop; REDQUEEN [7] operates similarly but forgoes taint tracking for program state monitoring. Driller's [117] concolic execution starts when fuzzing coverage stalls; QSYM's [143] instead runs in parallel, as do DigFuzz's [148] and SAVIOR's [25], which improve by prioritizing rare and bug-honing paths, respectively. While this work's focus is applying performant, compiler-quality transformations to the standard coverage-guided fuzzing loop, I imagine leveraging Zafl to also enhance the more heavyweight techniques central to hybrid fuzzing.

4.8.4 Emergent Fuzzing Transformations

LLVM [76] offers several robust "sanitizers" useful for software debugging. In fuzzing, sanitizers are typically reserved for post-fuzzing crash triage due to their performance bloat; but recently, several works achieve success with sanitizers *intra*-fuzzing: AFLGo [15] compiles binaries with AddressSanitizer for more effective crash-finding; Angora [22] builds its taint tracking atop DataFlowSanitizer [122]; and SAVIOR [25] uses UndefinedBehaviorSanitizer to steer concolic execution toward bug-exercising paths. I thus foresee increasing desire for sanitizers in binary-only fuzzing, however, their heavyweight nature makes porting them a challenge. RetroWrite [32] reveals the possibility that lightweight versions of sanitizers can

be incorporated in the main fuzzing loop while maintaining performance. I expect that such transformations can be realized with Zafl.

4.9 Conclusion

Zafl leverages state-of-the-art binary rewriting to extend compiler-quality instrumentation's capabilities to binary-only fuzzing—with compiler-level performance. I show its improved effectiveness among synthetic and real-world benchmarks: compared to the leading binary instrumenters, Zafl enables fuzzers to average 26–131% more unique crashes, 48–203% more test cases, achieve 60–229% less overhead, and find crashes in instances where competing instrumenters find none. I further show that Zafl scales well to real-world open- and closed-source software of varying size and complexity, and has Windows binary support.

My results highlight the requirements and need for compiler-quality instrumentation in binary-only fuzzing. Through careful matching of compiler instrumentation properties in a static binary rewriter, state-of-the-art compiler-based approaches can be ported to binary-only fuzzing—without degrading performance. Thus, I envision a future where fuzzing is no longer burdened by a disparity between compiler-based and binary instrumentation. a

Application	OS	Binary	Size	Blocks	Opt
Apache	L	httpd	1.0M	25,547	✔
AudioFile	L	sfconvert	568K	5,814	✔
BIND	L	named	9.4M	120,665	✔
Binutils	L	readelf	1.4M	21,085	✔
CatBoost	L	catboost	153M	1,308,249	✔
cJSON	L	cjson	43K	1,409	✔
Clang	L	clang	36.4M	1,756,126	✔
DNSMasq	L	dnsmasq	375K	20,302	✔
Gumbo	L	clean_text	571K	5,008	✔
JasPer	L	jasper	1.1M	14,795	✔
libarchive	L	bsdtar	2.1M	29,868	✔
libjpeg	L	djpeg	667K	5,066	✔
libksba	L	cert-basic	435K	5,247	✔
lighttpd	L	lighttpd	1.1M	12,558	✔
Mosh	L	mosh-client	4.2M	14,311	✔
NGINX	L	nginx	4.8M	29,507	✔
OpenSSH	L	sshd	2.3M	33,115	✔
OpenVPN	L	vpn	2.9M	34,521	✔
Poppler	L	pdftohtml	1.5M	2,814	✔
Redis	L	redis-server	5.7M	74,515	✔
Samba	L	smbclient	226K	6,279	✔
SIPWitch	L	sipcontrol	226K	772	✔
Squid	L	squid	32.7M	212,746	✔
tcpdump	L	tcpdump	2.3M	24,451	✔
thttpd	L	thttpd	119K	3,428	✔
UnRTF	L	unrtf	170K	1,657	✔
7-Zip	W	7z	447K	23,353	✗
AkelPad	W	AkelPad	540K	31,140	✗
cygwin64	W	bash	740K	38,397	✗
cygwin64	W	ls	128K	5,661	✗
fre:ac	W	freaccmd	97K	521	✗
fmedia	W	fmedia	178K	3,016	✗
fmedia	W	fmedia-gui	173K	1,363	✗

Table 4.8: Open-source binaries tested successfully with ZAFL. *L* **/ *W* = Linux/Windows; *Opt* = whether register liveness-aware optimization succeeds.**

Application	**OS**	**Binary**	**Size**	**Blocks**	**P*C**	**Sym**	**Opt**
B1FreeArchiver	L	b1	4.1M	150,138	D	✔	✔
B1FreeArchiver	L	b1manager	19.3M	290,628	D	✔	✔
BinaryNinja	L	binaryninja	34.4M	998,630	D	✔	✔
BurnInTest	L	bit_cmd_line	2.6M	73,229	D	✗	✔
BurnInTest	L	bit_gui	3.4M	107,897	D	✗	✔
Coherent PDF	L	smpdf	3.9M	61,204	D	✔	✔
IDA Free	L	ida64	4.5M	173,551	I	✗	✔
IDA Pro	L	idat64	1.8M	82,869	I	✗	✔
LzTurbo	L	lzturbo	314K	13,361	D	✗	✔
NConvert	L	nconvert	2.6M	111,652	D	✗	✔
NVIDIA CUDA	L	nvdisasm	19M	46,190	D	✗	✔
Object2VR	L	object2vr	8.1M	239,089	D	✔	✔
PNGOUT	L	pngout	89K	4,017	D	✗	✔
RARLab	L	rar	566K	25,287	D	✗	✔
RARLab	L	unrar	311K	13,384	D	✗	✔
RealVNC	L	VNC-Viewer	7.9M	338,581	D	✗	✔
VivaDesigner	L	VivaDesigner	28.9M	1,097,993	D	✗	✔
VueScan	L	vuescan	15.4M	396,555	D	✗	✔
Everything	W	Everything	2.2M	115,980	D	✔	✗
Imagine	W	Imagine64	15K	99	D	✗	✗
NirSoft	W	AppNetworkCounter	122K	4,091	D	✗	✗
OcenAudio	W	ocenaudio	6.1M	178,339	D	✗	✗
USBDView	W	USBDeview	185K	7,367	D	✗	✗

Table 4.9: Closed-source binaries tested successfully with ZAFL. *L/W* = Linux/Windows; *D/I* = position-dependent/independent; *Sym* = binary is non-stripped; *Opt* = whether register liveness-aware optimization succeeds.

Binary	Total Insns	IDA Pro			Binary Ninja			Zafl		
		Unrecov	Reached	FalseNeg	Unrecov	Reached	FalseNeg	Unrecov	Reached	FalseNeg
`idat64`	268K	1681	0	0	5342	2	0	958	0	0
`nconvert`	458K	105K	3117	0.68%	3569	0	0	33.0K	0	0
`nvdisasm`	162K	180	0	0	3814	21.4	0.01%	0	0	0
`pngout`	16.8K	645	0	0	752	112.5	0.67%	1724	0	0
`unrar`	37.8K	1523	0	0	1941	138.2	0.37%	40	0	0

Table 4.10: Instruction recovery statistics for IDA Pro, Binary Ninja, and Zafl, with ground-truth disassembly from LLVM-10's objdump. *Reached* = mean unrecovered instructions reached by fuzzing (hence, erroneously-unrecovered); *FalseNeg* = erroneously-unrecovered instructions over total.

Binary	Coverage TPR	Coverage TNR	Coverage Accuracy
`bsdtar`	97.28%	>99.99%	>99.99%
`cert-basic`	96.67%	>99.99%	>99.99%
`clean_text`	96.39%	>99.99%	>99.99%
`jasper`	98.82%	>99.99%	>99.99%
`readelf`	99.98%	>99.99%	>99.99%
`sfconvert`	98.71%	>99.99%	>99.99%
`tcpdump`	96.51%	>99.99%	>99.99%
`unrtf`	94.17%	>99.99%	>99.99%
Mean	**97.30%**	**100.00%**	**100.00%**

Table 4.11: Zafl's fuzzing code coverage true positive and true negative rates, and accuracy with respect to the LLVM compiler over 5×24-hour trials.

Chapter 5

Conclusion and Future Work

Fuzzing's aggressive, high-volume testing approach has shown immense success over previous-generation software security vetting. However, feedback and instrumentation asymmetries have long made effective fuzzing unattainable for *closed-source* software such as commercial, proprietary, and legacy code. This thesis offers a paradigm shift in software fuzzing: making it finally possible to achieve practical, precise, and performant closed-source software fuzzing that outperforms even open-source fuzzing techniques, while operating with just the limited information contained in pre-compiled binary executables. Through the techniques and tools I have introduced—**UnTracer**, **HeXcite**, and **Zafl**—software developers, maintainers, and practitioners are better equipped than ever before to uncover critical vulnerabilities in the commercial, proprietary, and legacy software that dominates our computing world. Moreover, the high-level principles of this work—(1) optimizing common-case insights, (2) balancing precision and performance, and (3) identifying and matching desirable properties—offers new motivation for future advancements to fuzzing as a whole: binary- *or* source-level.

5.1 Future Work: Improving *Scalability* of Closed-source Software Fuzzing Program Transformation

While the contributions of my thesis tackle Asymmetries 1–3 outlined in section 1.1 (low overhead code coverage, fine-grained code coverage, and low-overhead program transformation), Asymmetry 4—*scalable* program transformation—has and continues to remain a non-trivial problem for the field of software security as a whole. Though Zafl shows that compiler-style fuzzing enhancements *are* attainable for closed-source binaries on Linux and Windows, there are significant challenges in (1) program analysis and (2) automated testing that prevent the deployment of these advancements across the *broader* closed-source ecosystem. In the following, I will detail the fundamental next breakthroughs required to extend practical, precise, and performant automated security testing across the broader closed-source ecosystem, as well as my research vision for addressing each.

5.1.1 Tackling Asymmetries in Program Analysis

Security auditing hinges on precise program analysis, as virtually all vetting strategies require deep semantic understanding of the target code. Unfortunately, *binary-only* code is a far more hostile environment for program analysis, which impedes efforts to vet security-critical proprietary, commercial, and legacy codebases. **To lay the scientific and technical foundations for expanding vetting across the closed-source software ecosystem, my short-term research will first target the asymmetries hindering the *analysis* of opaque software and systems.**

Shining light on opaque tools. A challenge in practical binary analysis is understanding where and why heuristics and algorithms (e.g., function detection) *fail*. While prior work has scrutinized and improved open-source binary analysis tools, today's state-of-the-art, industry-standard toolchains center on commercial platforms like IDA Pro and Binary Ninja—whose internals are *not* public. The opaque mechanics of these tools places a significant burden on

users: for practitioners, dealing with analysis errors means either waiting for developer-issued fixes or taking on the challenge of hand-writing custom plugins; and for academics, this means tedious ground-truth benchmarking. Yet no two studies adopt the same corpora, resulting in contradictory outcomes and the need to reassess for every new tool update, architecture, and ISA extension. Therefore, the logical evolution to strengthening the state-of-the-art in binary analysis is *automated testing*. To this end, I will develop program synthesis-driven tools to tease out these frameworks' heuristics in a source/API-agnostic way and apply targeted fuzzing to uncover their failure points. Building from this, I will explore automated testing's efficacy in other security-oriented binary analysis tasks like semantic lifting and decompilation.

Retroactive repair. Another obstacle in binary analysis is recovering metadata (e.g., code and data symbols), which is commonly *stripped* by developers to impede third-party analyses. The difficulty of precise metadata recovery forces many academically-sourced tools to place unrealistic assumptions on users (e.g., that the binary possesses *full* symbols), restricting their applicability to only a small subset of software. Even in the best-case scenario where an executable's symbols are present, recent work shows that compilers and linkers frequently generate *incorrect* metadata, derailing analysis even further. To address this, I will extend the frontiers of program and compiler analysis with data-driven approaches to identify and retroactively mend malformed and miscompiled binary metadata. My aims in this direction are to not only improve the precision of state-of-the-art analysis tools, but to also afford even constrained (e.g., metadata-requiring) tools a broader compatibility, and bring a wider selection of tools and techniques to the forefront of practical binary analysis.

Next-generation tooling. Current binary analysis tools broadly fall under two archetypes: (1) transparent internals with immutable analyses (e.g., angr, Dyninst), or (2) opaque internals with mutable analyses (e.g., IDA Pro, Binary Ninja). *Transparency* means that a

tool's underlying heuristics and algorithms are interpretable, while *mutability* dictates that its binary representation is modifiable (e.g., toggling-on a heuristic) without need for *de novo* reanalysis. In my collaboration with Trail of Bits, I leveraged the power of logical programming (e.g., Datalog, Prolog) to support binary analysis that meets *both* properties: facilitating flexible yet succinct analysis. I plan to ramp up this work by engaging with HCI and industry experts to understand what gaps exist between binary analysis theory and practice, and make use of these insights to design more *human-centric* platforms for driving future research on vetting closed-source software and systems.

5.1.2 Tackling Asymmetries in Automated Testing

Safeguarding future computing demands making automated testing as efficient and effective in closed-source contexts as in open-source ones. Capitalizing on advancements in program analysis, **I will apply my expertise in binary fuzzing to tackle the asymmetries that restrict *automated testing* across the closed-source computing ecosystem.**

Fully-automated harnessing. Harnessing refers to the task of distilling a larger codebase or library into smaller driver programs on which to perform fuzzing. Current harnessing techniques are only *partially* automated and require human expertise (e.g., knowledge of data-flow dependencies) to refine candidate programs into *sound* harnesses. For open-source targets, this domain knowledge is obtainable through manual code review; yet for closed-source targets, the learning curve is much higher due to binary code's semantic opaqueness. Thus, fuzzing must embrace *fully-automated* harnessing in the complex and opaque domains where human intervention is becoming more and more unscalable. In this direction, I will develop approaches that combine data-driven program analysis with incremental,

stochastic program synthesis to generate, grow, and refine candidate harnesses "on the fly". Additionally, as even industry-scale fuzzing is shown to miss security bugs due to harness *blindspots* (where harnesses erroneously miss some security-critical paths), I will introduce new approaches to quantify, evaluate, and maximize *harness diversity* to help enable more thorough security auditing.

Cross-platform instrumentation. The lack of performant binary instrumentation outside of Linux has long impeded automated security auditing on Windows and macOS. As these platforms represent the largest ecosystems for commodity and closed-source software today, I will leverage my binary analysis and fuzzing expertise to devise new techniques for *platform-agnostic* compiler-quality fuzzing instrumentation. Advances in this space will empower existing fuzzing efforts (e.g., WinAFL, Jackalope) with the ability to port Linux-only enhancements (e.g., QSYM, AFL++) to *non-Linux* domains, getting one step closer to bridging the Linux, Windows, and macOS fuzzing communities. The primary technical challenges are soundly handling non-Linux binary formats which, in addition to introducing *platform-specific* layout conventions, also add *compiler-specific* patterns (e.g., MSVSC's aggressive data-in-code inlining), requiring fundamentally new heuristics and algorithms. With these challenges addressed, I will seek to adapt platform-agnostic instrumentation to other security-oriented tasks such as control-flow integrity, de-obfuscation, sanitization, taint tracking, and program hardening.

Bibliography

[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] F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137, 1976.

[4] Kapil Anand, Matthew Smithson, Aparna Kotha, Rajeev Barua, and Khaled Elwazeer. Decompilation to Compiler High IR in a binary rewriter. Technical report, University of Maryland, 2010.

[5] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*, USENIX, 2019.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark Stillwell, and others. SCONE: Secure Linux Containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2016.

[7] 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.

[8] Cornelius Aschermann, Patrick Jauernig, Tommaso Frassetto, Ahmad-Reza Sadeghi, Thorsten Holz, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed System Security Symposium*, NDSS, 2019.

[9] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, ATC, 2005.

[10] Andrew R. Bernat and Barton P. Miller. Anywhere, Any-time Binary Instrumentation. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE, 2011.

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

[12] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *USENIX Security Symposium*, USENIX, 2017.

[13] 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.

[14] 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.

[15] 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.

[16] Ahmed Bougacha. Dagger, 2018. URL `https://github.com/repzret/dagger`.

[17] 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/`.

[18] Amy Brown and Greg Wilson. The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks. 1, 2012.

[19] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization*, CGO, 2011.

[20] Cristian Cadar, Daniel Dunbar, Dawson 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*, OSDI, 2008.

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

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

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

[24] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary. In *ACM ASIA Conference on Computer and Communications Security*, ASIACCS, 2019. arXiv: 1905.10499.

[25] 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.

[26] 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.

[27] Vitaly Chipounov, Volodymyr Kuznetsov, and George 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*, ASPLOS, 2011.

[28] 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.

[29] Keith D Cooper and Timothy J Harvey. Compiler-Controlled Memory. In *ACM SIGOPS Operating Systems Review*, OSR, 1998.

[30] DARPA. DARPA Cyber Grand Challenge, 2018. URL `https://github.com/cybergrandchallenge`.

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

[32] 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.

[33] Brendan Dolan-Gavitt. Of Bugs and Baselines, 2018. URL `http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html`.

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

[35] dotPDN LLC and Rick Brewster. Paint.NET, 2020. URL `https://www.getpaint.net/`.

[36] Michael Eddington. Peach fuzzing platform, 2018. URL `https://www.peach.tech/products/peach-fuzzer/`.

[37] Alexis Engelke and Josef Weidendorfer. Using LLVM for Optimized Lightweight Binary Re-Writing at Runtime. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW, May 2017.

[38] 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`.

[39] FBReader.ORG Limited. FBReader, 2020. URL `https://fbreader.org/`.

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

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

[42] Benjamin Fleischer. FUSE for macOS, 2020. URL `https://osxfuse.github.io/`.

[43] 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.

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

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

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

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

[48] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[49] Google. ClusterFuzz, 2018. URL `https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md`.

[50] Google. fuzzer-test-suite: Set of tests for fuzzing engines, 2018. URL `https://github.com/google/fuzzer-test-suite`.

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

[52] GrammaTech. GTIRB, 2019. URL `https://github.com/GrammaTech/gtirb`.

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

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

[55] Niranjan Hasabnis and R. Sekar. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

[56] 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.

[57] Matthew S Hecht and Jeffrey D Ullman. Flow Graph Reducibility. *SIAM Journal on Computing*, 1(2):188–202, 1972.

[58] Israel Herraiz, Daniel Izquierdo-Cortazar, and Francisco Rivas-Hernández. FLOSSMetrics: Free/Libre/Open Source Software Metrics. In *European Conference on Software Maintenance and Reengineering*, 2009.

[59] Jesse Hertz and Tim Newsham. ProjectTriforce: AFL/QEMU fuzzing with full-system emulation., 2017. URL `https://github.com/nccgroup/TriforceAFL`.

[60] Marc Heuse. AFL-DynamoRIO, 2018. URL `https://github.com/vanhauser-thc/afl-dynamorio`.

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

[62] Marc Heuse. AFL-PIN, 2018. URL `https://github.com/vanhauser-thc/afl-pin`.

[63] Masami Hiramatsu and Satoshi Oshima. Djprobe–Kernel probing with the smallest overhead. In *Linux Symposium*, Linux Symposium, 2007.

[64] 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.

[65] 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.

[66] Intel. Intel Processor Trace Tools, 2017. URL `https://software.intel.com/en-us/node/721535`.

[67] 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.

[68] James Johnson. gramfuzz, 2018. URL `https://github.com/d0c-s4vage/gramfuzz`.

[69] 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.

[70] Mateusz Jurczyk. CmpCov, 2019. URL `https://github.com/googleprojectzero/CompareCoverage`.

[71] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES, 2002.

[72] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. Kernel probes (kprobes). *Documentation provided with the Linux kernel sources (v2. 6.29)*, 2016.

[73] 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.

[74] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, VMCAI, 2009.

[75] 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.

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

[77] 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.

[78] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically Generating Pathological Inputs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2018.

[79] 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.

[80] Mel Llaguno. 2017 Coverity Scan Report. Technical report, Synopsys Inc, 2017. URL https://www.synopsys.com/blogs/software-security/2017-coverity-scan-report-open-source-security/.

[81] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2005.

[82] 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.

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

[84] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2016.

[85] Mozilla Security. Dharma: A generation-based, context-free grammar fuzzer., 2018. URL https://github.com/MozillaSecurity/dharma.

[86] Robert Muth. Register Liveness Analysis of Executable Code. 1998.

[87] Stefan Nagy and Matthew Hicks. afl-fid: A suite of AFL modifications for fixed input dataset experiments, 2019. URL https://github.com/FoRTE-Research/afl-fid.

[88] Stefan Nagy and Matthew Hicks. FoRTE-FuzzBench: FoRTE-Research's fuzzing benchmarks, 2019. URL https://github.com/FoRTE-Research/FoRTE-FuzzBench.

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

[90] Stefan Nagy and Matthew Hicks. UnTracer-AFL: An AFL implementation with UnTracer (our coverage-guided tracer), 2019. URL `https://github.com/FoRTE-Research/UnTracer-AFL`.

[91] 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.

[92] Nikolaos Naziridis and Zisis Sialveras. Choronzon - An evolutionary knowledge-based fuzzer, 2016. URL `https://github.com/CENSUS/choronzon`.

[93] Aleksandar Nikolic. Guided Fuzzing And Binary Blobs, 2016. URL `https://www.youtube.com/watch?v=zQb-QT7tiFQ`.

[94] Serkan Özkan. CVE Details: The ultimate security vulnerability datasource. Technical report, 2018. URL `https://www.cvedetails.com/vulnerabilities-by-types.php`.

[95] 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.

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

[97] Chen Peng. AFL_pin_mode, 2017. URL `https://github.com/spinpx/afl_pin_mode`.

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

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

[100] Nguyen Anh Quynh. Capstone: The Ultimate Disassembler, 2019. URL `http://www.capstone-engine.org/`.

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

[102] Michael Rash. afl-cve: A collection of vulnerabilities discovered by the AFL fuzzer (afl-fuzz), 2017. URL `https://github.com/mrash/afl-cve`.

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

[104] 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.

[105] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, USENIX, 2017.

[106] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*, WCRE, 2002.

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

[108] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, USENIX, 2017.

[109] Bhargava Shastry, Federico Maggi, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing. In *USENIX Workshop on Offensive Technologies*, WOOT, 2017.

[110] 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.

[111] Shellphish. ShellPhuzz, 2018. URL `https://github.com/shellphish/fuzzer`.

[112] Yan Shoshitaishvili. CGC Binaries: Compiled CGC binaries for experimentation porpoises., 2017. URL `https://github.com/zardus/cgc-bins`.

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

[114] Maksim Shudrak. drAFL, 2019. URL `https://github.com/mxmssh/drAFL`.

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

[116] Richard Stallman, Roland Pesch, Stan Shebs, and others. Debugging with GDB. *Free Software Foundation*, 675, 1988.

[117] 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.

[118] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

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

[120] talos-vulndev. AFL-Dyninst, 2018. URL `https://github.com/talos-vulndev/afl-dyninst`.

[121] R Tarjan. Testing Flow Graph Reducibility. In *ACM Symposium on Theory of Computing*, STOC, 1973.

[122] The Clang Team. DataFlowSanitizer, 2019. URL `https://clang.llvm.org/docs/DataFlowSanitizer.html`.

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

[124] The Council of Economic Advisers. The Cost of Malicious Cyber Activity to the U.S. Economy. Technical report, The White House, 2018. URL `https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf`.

[125] Henrik Theiling. Extracting safe and precise control flow from binaries. In *IEEE International Conference on Real-Time Systems and Applications*, RCTSA, 2000.

[126] Parker Thompson. AFLPIN, 2015. URL `https://github.com/mothran/aflpin`.

[127] Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient Instrumentation for Code Coverage Testing. *ACM SIGSOFT Software Engineering Notes*, 27:86–96, 2002.

[128] Anatoly Trosinenko. AFL-Dr, 2017. URL `https://github.com/atrosinenko/afl-dr`.

[129] András Vargha and Harold D Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

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

[131] Martin Vuagnoux. Autodafe, an Act of Software Torture, 2006. URL `http://autodafe.sourceforge.net/`.

[132] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. In *IEEE Symposium on Security and Privacy*, Oakland, 2017.

[133] 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.

[134] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Network and Distributed System Security Symposium*, NDSS, 2017.

[135] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *USENIX Security Symposium*, USENIX, 2015.

[136] Song Wang, Jaechang Nam, and Lin Tan. QTEP: Quality-aware Test Case Prioritization. In *ACM Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, 2017.

[137] 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*, Oakland, 2010.

[138] 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.

[139] 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.

[140] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, Oakland, 2015.

[141] 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.

[142] 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.

[143] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Prac-

tical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, USENIX, 2018.

[144] Michal Zalewski. Fuzzing random programs without execve(), 2014. URL `http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html`.

[145] Michal Zalewski. afl-users > Re: ”FidgetyAFL” implemented in 2.31b, 2016. URL `goo.gl/zmcvZf`.

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

[147] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. PTfuzz: Guided Fuzzing with Processor Trace Feedback. *IEEE Access*, 6:37302–37313, 2018.

[148] 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.