Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing

Stefan Nagy, *Virginia Tech;* Anh Nguyen-Tuong, Jason D. Hiser, and Jack W. Davidson, *University of Virginia;* Matthew Hicks, *Virginia Tech*

https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing

Stefan Nagy
Virginia Tech
snagy2@vt.edu

Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson
University of Virginia
{nguyen, hiser, jwd}@virginia.edu

Matthew Hicks
Virginia Tech
mdhicks2@vt.edu

Abstract

Coverage-guided fuzzing is one of the most effective software security testing techniques. Fuzzing takes on one of two forms: compiler-based or binary-only, depending on the availability of source code. While the fuzzing community has improved compiler-based fuzzing with performance- and feedback-enhancing program transformations, binary-only fuzzing lags behind due to the semantic and performance limitations of instrumenting code at the binary level. Many fuzzing use cases are binary-only (i.e., closed source). Thus, applying *fuzzing-enhancing program transformations* to binary-only fuzzing—without sacrificing performance—remains a compelling challenge.

This paper examines the properties required to achieve compiler-quality binary-only fuzzing instrumentation. Based on our findings, we design ZAFL: a platform for applying fuzzing-enhancing program transformations to binary-only targets—maintaining compiler-level performance. We showcase ZAFL's capabilities in an implementation for the popular fuzzer AFL, including five compiler-style fuzzing-enhancing transformations, and evaluate it against the leading binary-only fuzzing instrumenters AFL-QEMU and AFL-Dyninst. Across LAVA-M and real-world targets, ZAFL improves crash-finding by **26–96%** and **37–131%**; and throughput by **48–78%** and **159–203%** compared to AFL-Dyninst and AFL-QEMU, respectively—while maintaining compiler-level of overhead of **27%**. We also show that ZAFL supports real-world open- and closed-source software of varying size (**10K–100MB**), complexity (**100–1M** basic blocks), platform (Linux and Windows), and format (e.g., stripped and PIC).

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 [93], libFuzzer [70], and honggFuzz [75].

Most modern fuzzers require access to the target's source code, embracing *compiler instrumentation*'s low overhead for high fuzzing throughput [70, 75, 93] and increased crash finding. State-of-the-art fuzzers further use compilers to apply *fuzzing-enhancing program transformation* that improves target speed [32, 47], makes code easier-to-penetrate [1], or tracks interesting behavior [18]. 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 [64], PIN [56], and QEMU [8]). 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 [9, 41, 46, 86, 87], it remains an open question whether compiler-quality instrumentation capabilities *and* performance are within reach for binary-only fuzzing.

To address this challenge we 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. We demonstrate how ZAFL facilitates powerful fuzzing enhancements with a suite of five transformations, ported from compiler-based fuzzing contexts. We 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. We 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, we 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 paper contributes the following:

- We 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 our 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.
- We 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.
- We show that ZAFL supports real-world binaries of varying characteristics, size, complexity, and platform—even those binaries not supported by other instrumenters.
- We will open-source ZAFL and all benchmark corpora at `https://git.zephyr-software.com/opensrc/zafl`.

2 Background on Fuzzing

Coverage-guided grey-box fuzzing remains one of the most successful software security auditing techniques. Fuzzers of this type iteratively mutate test cases to increase code coverage, using lightweight instrumentation to collect this coverage at runtime. This section details the fundamental components of coverage-guided grey-box fuzzing.

2.1 An Overview of Fuzzing

Fuzzing is designed to root-out software vulnerabilities automatically. Given a target program and a set of seed test cases, a standard fuzzing cycle consists of (Figure 1):

0. **Instrumentation**: modify target program as desired (e.g., to track code coverage).
1. **Test Case Generation**: select a seed and mutate it to generate a batch of candidate test cases.
2. **Execution Monitoring and Feedback Collection**: run each candidate test case and monitor the target program's execution, collecting feedback via instrumentation.
3. **Feedback Decision-making**: keep only test cases with execution behavior matching some pre-specified constraint(s) (e.g., cover new code).
4. Return to step 1.

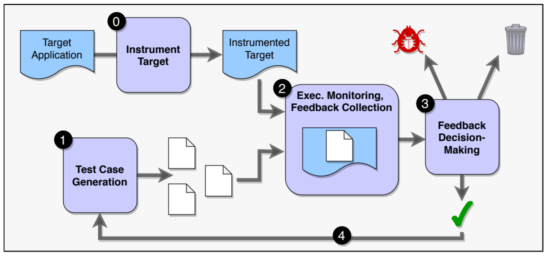


Figure 1: A high-level overview of the basic fuzzing workflow.

Though fuzzers vary by generation (i.e., mutation- [70, 75, 93] or grammar-based [35, 50, 60]), execution monitoring (i.e., white- [17, 22, 36], black- [60, 63, 83], or grey-box [70, 75, 93]), and feedback decision-making strategies (i.e., directed [13, 33, 41, 89] or coverage-guided [14, 70, 75, 93]), we elide their differentiation as they are outside the focus of this paper.

2.2 Coverage-guided Grey-box Fuzzing

By far the most popular fuzzing technique is *coverage-guided grey-box fuzzing* (e.g., AFL [93], honggFuzz [75], and libFuzzer [70]). As the name implies, coverage-guided grey-box fuzzers focus exclusively on test cases that increase code coverage, with the aim of testing as much of a target program's functionality as possible to find its deeply-rooted bugs. Its "grey-box" quality refers to a middle-ground between the deep and shallow program analyses used by white- and black-box fuzzers, respectively: lightweight instrumentation is used track test cases' coverage of the target, which is then post-processed to verify if new code has been covered.

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. More recent state-of-the-art efforts leverage compilers' ability to apply complex program transformations. Researchers have shown that such transformations improve fuzzing effectiveness by enhancing performance [32, 47] or introspection [1, 18, 31, 51]. Most real-world fuzzing is undertaken in the absence of target source (i.e., binary-only). This restricts fuzzing to existing binary instrumenters which are unsupportive of compiler-quality transformation, facing prohibitively-high overhead—often as high as 1000% for coverage tracing alone [62].

3 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 [79, 93]. Though advanced fuzzers introduce more heavyweight analyses [7, 18, 74, 92],

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

the core of these approaches remains the standard coverage-guided fuzzing loop (Figure 1)—amounting to over 90% of their execution time [62]; recent feedback enhancements (e.g., context sensitivity) only increase the proportion of time spent tracing execution. Thus, our focus is *performant fuzzing-enhancing transformations* in the absence of source code.

State-of-the-art fuzzers leverage compiler instrumentation to add transformations that improve fuzzing *performance* and *feedback* (e.g., AFL++ [31], Angora [18], CollAFL [32], honggFuzz [75], INSTRIM [47], libFuzzer [70]). 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. We broadly examine popular fuzzers and identify four categories of fuzzing-enhancing transformation that target the core coverage-guided loop (Table 1): (1) *instrumentation pruning*, (2) *instrumentation downgrading*, (3) *sub-instruction profiling*, and (4) *extra-coverage behavior tracking*. Below we detail each transformation.

3.1 Instrumentation Pruning

Graph reducibility techniques [42, 77] are used in fuzzing to elide instrumenting some target basic blocks, thus lowering overall runtime overhead. AFL's [93] 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 [31, 47] 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 [47] only instruments blocks targeted by backward edges and tracks loops either by entry or pre-entry blocks (the latter forgoing loop iteration tracking).

3.2 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 [93]. 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 [32]'s compiler instrumentation optimizes single-predecessor blocks by downgrading them to fewer-instruction block coverage (i.e., $cov(A \rightarrow B) \equiv cov(B)$).

3.3 Sub-instruction Profiling

Fuzzers struggle to penetrate code guarded by complex predicates like "magic bytes" [68], 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 [51], honggFuzz [75], laf-Intel [1]). Such splitting of roadblocks into smaller, simpler problems facilitates greater fuzzing code coverage.

3.4 Extra-coverage Behavior Tracking

An area of current research in fuzzing is the inclusion of execution behavior beyond traditional code coverage. Although we 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 [18, 31].

4 Binary-only Fuzzing: the Bad & the Ugly

Program transformation has become ubiquitous in compiler-based fuzzers (e.g., AFL++ [31], CollAFL [32], 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. We follow this exploration with an identification of the key instrumenter design attributes necessary to support *compiler-quality* fuzzing-enhancing program transformation and speed.

4.1 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,

Name	Fuzzing Appearances	Fuzzing Overhead	Supports Xform	Instrumentation			Supported Programs			
				type	invoked	liveness	PIC & PDC	C & C++	stripped	PE32+
LLVM	[1,6,13,18,19,31,32, 47,70,75,93]	18–32%	✓	static	inline	✓	N/A	✓	N/A	N/A
Intel PT	[7,11,20,37,75]	19–48%	✗	hardware	replay	✗	✓	✓	✓	✓
DynamoRIO	[37,43,73]	>1,000%	✓	dynamic	inline	✓	✓	✓	✓	✓
PIN	[45,49,63,68,92]	>10,000%	✓	dynamic	inline	✓	✓	✓	✓	✓
QEMU	[23,31,91,93]	>600%	✓	dynamic	inline	✓	✓	✓	✓	✓
Dyninst	[44,55,62,76]	>500%	✓	static	tramp.	✗	✓	✓	✗	✗
RetroWrite	[26]	20–64%	✗	static	tramp.	✓	✗	✗	✗	✗

Table 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.

(2) dynamic binary translation, or (3) static binary rewriting. Below we 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 [48]). Fuzzing implementations are burdened by the need for costly trace post-processing, which reportedly incurs overheads as high as **50%** over compilers [7,20]; but despite some optimistic performance improvements [37], 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 [43], PIN [56], and QEMU [8]). Translators generally support many architectures and binary characteristics; and offer deep introspection that simplifies analysis and transformation [31,93]. However, existing dynamic translators attain the worst-known fuzzing performance: recent work shows AFL-QEMU's average overhead is well over **600%** [62], and AFL-DynamoRIO [43] and AFL-PIN [45] 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 [44]). 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%** [62] 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 [93]; and moreover, it does not overcome the generalizability challenges of prior attempts at reassembleable-assembly (e.g., Uroboros [87], Ramblr [86]), and is hence limited to position-independent Linux C programs. Neither scale well to stripped binaries.

As summarized in Table 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 our knowledge the only two such implementations exist atop of AFL-Dyninst (instruction pruning [44]) and AFL-PIN (context sensitivity [92])—both suffering from the central flaw that any of their potential benefits are outweighed by the steep overheads of their respective binary instrumenters (over **500%** and **10,000%**, respectively [45,62]).

Impetus: Current binary instrumenters are fundamentally ill-equipped to support compiler-quality fuzzing instrumentation. We 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.

4.2 Fundamental Design Considerations

Our 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 we 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 [8]. 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 [43,45,93]. While translation does facilitate transformations like sub-instruction profiling [31], 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. We 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++ [26,86,87]. others apply to only position-independent (i.e., relocatable) code and neglect the bulk of software that remains position-dependent [26]; many presume access to debugging symbols (i.e., non-stripped) but this seldom holds true when fuzzing proprietary software [44]; and most are only Linux-compatible, leaving some of the world's most popular commodity software (Windows 64-bit PE32+) unsupported [26,44,86,87]. 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{\text{C}}$1) and thus incur orders-of-magnitude performance penalties—before applying any feedback-enhancing transformation. Dyninst and RetroWrite embrace static rewriting but both rely on costlier trampoline-based invocation ($\overline{\text{C}}$2) and fail to support commodity binary formats and characteristics ($\overline{\text{C}}$4); and moreover, Dyninst's liveness-aware instrumentation failed on our evaluation benchmarks ($\overline{\text{C}}$3). Thus, compiler-quality instrumentation in a binary-only context demands a new approach that satisfies all four criteria.

5 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 (§ 3), ZAFL streamlines instrumentation through four extensible phases, facilitating intuitive implementation and layering of state-of-the-art fuzzing-enhancing program transformations. Below we detail ZAFL's internal architecture and guiding design principles.

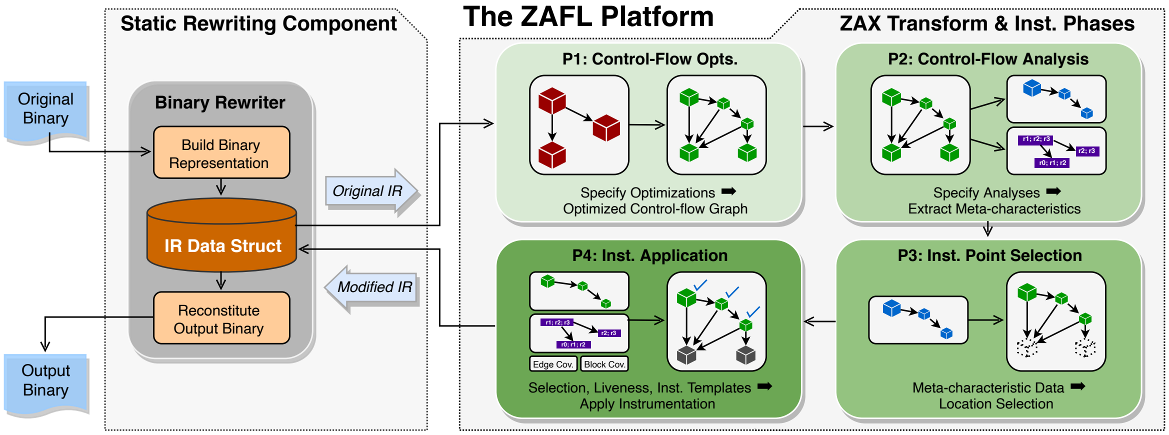

The ZAFL Platform
Static Rewriting Component
Original Binary
Binary Rewriter
Build Binary Representation
IR Data Struct
Reconstitute Output Binary
Output Binary
Original IR
Modified IR
ZAX Transform & Inst. Phases
P1: Control-Flow Opts.
Specify Optimizations ➡ Optimized Control-flow Graph
P2: Control-Flow Analysis
r1; r2; r3
r2; r3
r0; r1; r2
Specify Analyses ➡ Extract Meta-characteristics
P3: Inst. Point Selection
Meta-characteristic Data ➡ Location Selection
P4: Inst. Application
r1; r2; r3
r2; r3
r0; r1; r2
Edge Cov.
Block Cov.
Selection, Liveness, Inst. Templates ➡ Apply Instrumentation

Figure 2: A high-level depiction of the ZAFL platform architecture and its four ZAX transformation and instrumentation phases.

5.1 Design Overview

As shown in Figure 2, 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 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.

5.1.1 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.

We initially considered re-purposing LLVM IR-based rewriter McSema [25] due to its maturity and popularity in the static rewriting community, but ultimately ruled it out as both the literature [29] and our 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 [41, 46] as it meets the same criteria that McSema does (§ 4.2), but has better baseline performance.

5.2 The ZAX Transformation Architecture

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

5.2.1 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.

5.2.2 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, 24, 61], and to facilitate integration of other desirable analyses appearing in the literature [2, 81]. The extent of possible analyses depends on the rewriter's IR; for example, low-level IR's modeled after GCC's RTL [34] 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 [61] and could be added to support ZAX.

5.2.3 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 § 3.

Performance Transformation	
Single Successor-based Pruning	[31]
Dominator-based Pruning	[47]
Instrumentation Downgrading	[32]
Feedback Transformation	
Sub-instruction Profiling	[1, 31, 51, 75]
Context-sensitive Coverage	[18, 31]

Table 3: A catalog of ZAFL-implemented compiler-quality fuzzing-enhancing program transformations and their compiler-based origins.

5.2.4 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 [34, 53]. Though our current prototype (§ 6) targets AFL-style fuzzers, support for others is possible through new instrumentation configurations.

6 Extending Compiler-quality Transforms to Binary-only Fuzzing

We 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.

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 3. Below details our implementations of these five transformations using ZAFL.

6.1 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**. We describe our implementation of each below.

6.1.1 Single Successor Instrumentation Pruning

Recent fuzzing works leverage flow graph reducibility techniques [42, 77] to cut down instrumentation overhead [47]. 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 [44]. 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.

6.1.2 Dominator Tree Instrumentation Pruning

Tikir and Hollingsworth [81] 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. Our analysis reveals this omits 30–50% of blocks from instrumentation. We elect to apply Tikir and Hollingsworth's algorithm because it balances graph reduction and analysis effort. Other alternative, more aggressive algorithms exist [2, 47], which we believe are also implementable in ZAFL.

6.1.3 Edge Instrumentation Downgrading

CollAFL [32] 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. Our numbers show that roughly 35–45% of basic blocks benefit from this optimization.

6.2 Feedback-enhancing Transformations

Recent compiler-based fuzzing efforts attain improved code-penetration power by considering finer-grained execution information [18, 31]. Below we detail our ZAFL implementations of two prominent examples: **sub-instruction profiling** and **context-sensitive coverage** tracking.

6.2.1 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 [51]) replace comparisons with nested micro-comparisons; however, as the goal is to augment control-flow with nested conditionals that permit increased feedback, we 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.

6.2.2 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 [18] or function-level [31]. 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.

7 Evaluation

Our 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?

We first perform an evaluation of ZAFL against the leading binary-only fuzzing instrumenters AFL-Dyninst and AFL-QEMU on the LAVA-M benchmark corpus [28]. Second, to see if LAVA-M results hold for real-world programs, we expand our evaluation to eight popular programs well-known to the fuzzing literature, selecting older versions *known to contain bugs* to ensure self-evident comparison. Third, we evaluate these instrumenters' fuzzing overhead across each. Fourth, we evaluate ZAFL alongside AFL-Dyninst and AFL-QEMU in fuzzing five varied *closed-source* binaries. Fifth, we test ZAFL's support for 42 open- and closed-source programs of varying size, complexity, and platform. Finally, we use industry-standard reverse-engineering tools as ground-truth to assess ZAFL's precision.

7.1 Evaluation-wide Instrumenter Setup

We evaluate ZAFL against the fastest-available binary-only fuzzing instrumenters; we thus omit AFL-PIN [45, 65, 80] and AFL-DynamoRIO [43, 73, 82] variants as their reported overheads are much higher than AFL-Dyninst's and AFL-QEMU's; and Intel PT [48] as it does not support instrumentation (Table 2). We configure AFL-Dyninst and AFL-QEMU with recent updates which purportedly increase their fuzzing performance by 2–3x and 3–4x, respectively. We detail these below in addition to our setup of ZAFL.

AFL-Dyninst: A recent AFL-Dyninst update [44] 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] We discovered three other performance-impacting BPatch settings (`setLivenessAnalysis`, `setMergeTramp`, and `setInstrStackFrames`). For fairness we apply the fastest-possible AFL-Dyninst configurations to all benchmarks; but for `setLivenessAnalysis` we are 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 chaining incompatible as skipping translation leaves some blocks uninstrumented, potentially missing coverage. A newly-released AFL-QEMU update [10] 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 we apply this update in all experiments.

ZAFL: To explore the effects of compiler-quality fuzzing-enhancing transformation on binary-only fuzzing we instrument benchmarks with all transformations shown in Table 3.

7.2 LAVA-M Benchmarking

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

7.2.1 Benchmarks

We compile each benchmark with Clang/LLVM before instrumenting with AFL-Dyninst and ZAFL; for AFL-QEMU we simply run compiled binaries in AFL using "QEMU mode". As fuzzer effectiveness on LAVA-M is sensitive to starting seeds and/or dictionary usage, we fuzz each instrumented binary per four configurations: empty and default seeds both with and without dictionaries. We build dictionaries as instructed by one of LAVA-M's authors [27].

[1] This AFL-Dyninst update [44] 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 our benchmarks. For these reasons we omit it in our experiments.

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

7.2.2 Experimental Setup and Infrastructure

We adopt the standard set by other LAVA-M evaluations [7, 72, 92] and fuzz each instrumented binary for five hours with the coverage-guided fuzzer AFL [93]; each for five trials per the four seed/dictionary configurations. All instrumenters are configured as detailed in § 7.1. To maintain performance neutrality, we 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.

7.2.3 Data Processing and Crash Triage

We 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 us to pinpoint *when* each crash occurred in its trial. We 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 we triage solely by its benchmarks' self-reported bug ID's.

We compute the average unique crashes, total processed and queued test cases for all instrumenter-benchmark trial groupings. To show ZAFL's effectiveness, we 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 [52] recommendation, to determine if ZAFL's gains are statistically significant, we compute a Mann-Whitney U-test with a 0.05 significance level, and report the geometric mean p-values across all benchmarks.

7.2.4 Results

We do not include ZAFL's context sensitivity in our LAVA-M trials as we 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 shows ZAFL's mean relative increase in triaged crashes, total and queued test cases over AFL-Dyninst and AFL-QEMU per configuration.

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 we observe Mann-Whitney U p-values (0.005–0.023) below the 0.05 threshold we 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.

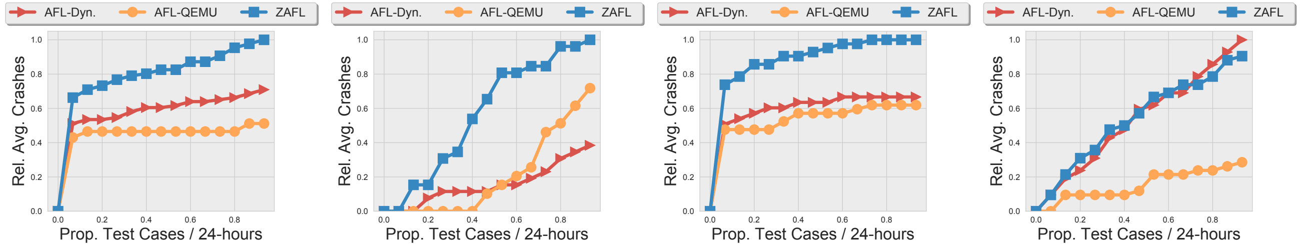
ZAFL versus AFL-QEMU: ZAFL matches or surpasses AFL-QEMU among 15 benchmark configurations, averaging 42% more crashes and 203% more test cases seen. As with AFL-Dyninst, ZAFL successfully finds crashes in four cases for which AFL-QEMU finds none. Additionally, the Mann-Whitney U p-values (0.005–0.039) 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, we 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, 21, 54, 94].

To our 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 § 7.5).

ZAFL versus AFL-LLVM: To gain a sense of whether ZAFL's transformation is comparable to existing compiler-based implementations, we ran ZAFL alongside the the analogous configuration of AFL's LLVM instrumentation with its INSTRIM [47] 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.

7.3 Fuzzing Real-world Software

Though our 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



(a) `cert-basic` (b) `jasper` (c) `unrtf` (d) `tcpdump`

Figure 3: 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 5: ZAFL's real-world software mean triaged crashes and total/queued test cases rel. to AFL-Dyninst and AFL-QEMU. We report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance).

over to real-world programs. We therefore expand our 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. [52]. We further show that ZAFL achieves compiler-quality performance in a coverage-tracing overhead comparison of all three instrumenters.

7.3.1 Benchmarks

To capture the diversity of real-world software we 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`. We 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 8.

7.3.2 Experimental Setup and Infrastructure

In both crash-finding and overhead experiments we configure instrumenters and binaries as described in § 7.1 and § 7.2.1, and utilize either AFL- or developer-provided seed inputs in fuzzing evaluations. For crash-finding, we fuzz all instrumented binaries with AFL on a cluster for 8×24-hour trials each and to evaluate overhead, we perform 5×24-hour trials on our LAVA-M experiment infrastructure (§ 7.2.2).

7.3.3 Real-world Crash-finding

We apply all ZAFL-implemented transformations (Table 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 § 7.2.3 but is based on stack hashing as seen in the literature [52, 57, 66].[2] Table 5 shows ZAFL-instrumented fuzzing crash-finding as well as total and queued test cases relative to AFL-Dyninst and AFL-QEMU. We further report the geometric mean Mann-Whitney U significance test p-values across all metrics.

ZAFL versus AFL-Dyninst: Our 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 3d). Likewise for `readelf` our 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 3), averaging 61% more crashes. Given the Mann-Whitney U p-values (0.001–0.018) below the 0.05 significance level, we conclude that ZAFL's compiler-quality transformations bear a statistically significant advantage over AFL-Dyninst.

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.

7.3.4 Real-world Coverage-tracing Overhead

For our coverage-tracing overhead evaluation we follow established practice [62]: we 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

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

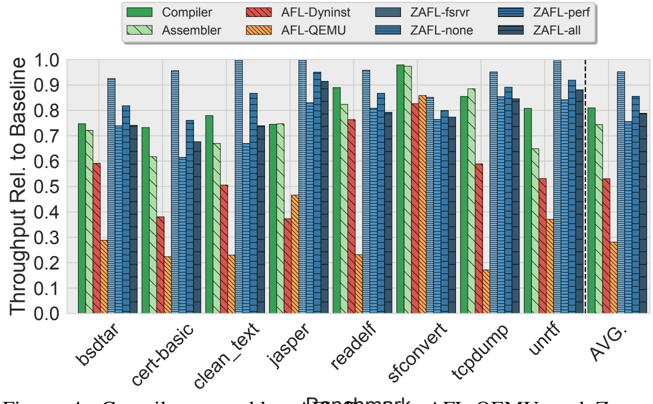


Figure 4: Compiler, assembler, AFL-Dyninst, AFL-QEMU, and ZAFL fuzzing instrumentation performance relative to baseline (higher is better).

resulting overheads relative to baseline.

We compare ZAFL to AFL-Dyninst, AFL-QEMU, and to the compiler- and assembler-based instrumentation available in AFL [93]. We 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 3) transformations (ZAFL-ALL). We 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 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: We observe ZAFL performs slightly worse on `sfconvert` as it has the fewest basic blocks by far we 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 [62], ZAFL outperforms it with nearly 229% less overhead. Interestingly, AFL-QEMU beats AFL-Dyninst on `jasper`, consistent with the relative throughput gains in Table 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. Our 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**.

7.4 Fuzzing Closed-source Binaries

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

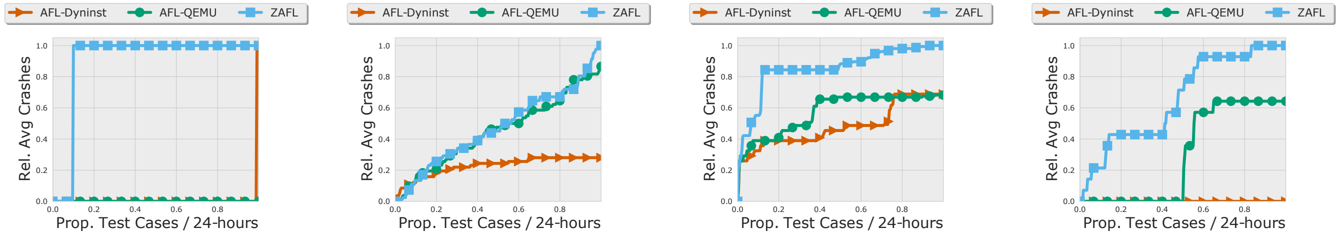
7.4.1 Benchmarks

We drill-down the set of all closed-source binaries we tested with ZAFL (Table 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 9 lists the key features of each benchmark.

7.4.2 Closed-source Crash-finding

We repeat the evaluation from § 7.3.3, running five 24-hour experiments per configuration. Our results (mean unique triaged crashes, total and queued test cases, and MWU p-scores) among all benchmarks are shown in Table 6; and plots of unique triaged crashes over time are shown in Figure 5.

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. We believe AFL-Dyninst's speed, small queues, and lack of crashes in `unrar` are due to it missing significant parts of these binaries, as our own



(a) idat64 (b) nconvert (c) pngout (d) unrar

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

Binary	ZAFL vs. AFL-Dyninst rel. crash	 rel. total	 rel. queue	ZAFL vs. AFL-QEMU 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 6: ZAFL's closed-source binary mean triaged crashes and total/queued test cases relative to AFL-Dyninst and AFL-QEMU. We report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance). ✗ = ZAFL finds crashes while competitor finds zero.

testing with its graph-pruning off shows it leaves over 50% of basic blocks uninstrumented for all but pngout. We 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), we 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.

7.4.3 Bug-finding Case Study

Following additional manual triage with binary-level memory error checkers (e.g., QASan [30] and Dr. Memory [16]), we 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.

Table 7 reports the geometric mean TTD among all five bugs for all three instrumenters. We 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.

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 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.

7.5 Scalability and Precision

We 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 our evaluation benchmarks, as well as key considerations related to its static rewriting precision.

7.5.1 Scalability

We instrument and test ZAFL on a multitude of popular real-world binaries of varying size, complexity, source availability, and platform. We 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; we do the same for Windows binaries using ZAFL's cross-instrumentation support. We 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 [93] (for Linux).

We verify ZAFL achieves success on 33 open-source Linux and Windows binaries, shown in Table 8. To confirm ZAFL's applicability to *true* binary-only use cases, we expand our testing with 23 closed-source binaries from 19 proprietary and commercial applications, listed in Table 9. In summary, our 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).

7.5.2 Liveness-aware Optimization

As discussed in § 4.2, register liveness analysis enables optimized instrumentation insertion for closer-to-compiler-level

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 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 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.

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 (§ 5.2.4) cannot guarantee correctness, it conservatively halts this optimization to avoid false negatives, and instead safely inserts code at basic block starts.

To assess the impact of skipping register liveness-aware optimization, we replicate our overhead evaluation (§ 7.3.4) to compare ZAFL's speed with/without liveness-aware instrumentation. As Figure 6 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.

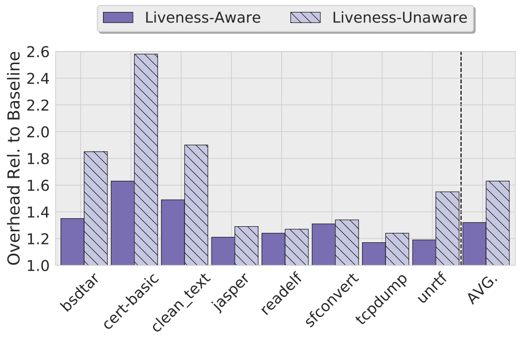


Figure 6: A comparison of ZAFL's runtime overhead with and without register liveness-aware instrumentation optimization (lower is better).

As Table 8 and Table 9 show, we successfully apply liveness-aware instrumentation for all 44 Linux benchmarks. We posit that with further engineering, the same robustness is achievable for Windows binaries.

7.5.3 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 (§ 6.2.1)).

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 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.

We 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, we use the version shipped in LLVM-10 [53]). To see how ZAFL fairs with respect to the state-of-the-art in binary analysis, we 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", we 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 10 lists the total instructions; and total and reached unrecovered instructions per our five closed-source benchmarks.[3] As we observe zero false positives for any tool on any benchmark, we 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 (§ 8.3)—these results suggest that ZAFL's *common-case* instruction recovery is sound.

7.5.4 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, we run all test cases generated over five 24-hour trials for our eight open-source benchmarks on both a ZAFL- and a ground-truth LLVM-instrumented binary, and log when each report new coverage.

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 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.

As Table 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, we 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, we conclude that ZAFL succeeds in preserving the control-flow of compiler-generated code.

8 Limitations

Below we briefly discuss limitations unique to ZAFL, and others fundamental to static binary rewriting.

8.1 Improving Baseline Performance

Our performance evaluation § 7.3.4 shows ZAFL's baseline (i.e., non-tracing) overhead is around 5%. We 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, we leave further optimization and the requisite re-evaluation to future work.

8.2 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), we 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. We leave instrumenting non-C/C++ languages for future work.

While ZAFL is engineered with Linux targets in mind, our 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, we expect that with future rewriter-level enhancements, ZAFL will achieve broader success across other Windows binary formats and versions.

8.3 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, 90]. Thus, we 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. We recognize there are trade-offs between static binary rewriting generalizability and precision,

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

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 [67] and IDA [39], 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. We posit that trickle-down dependency limitations are an inherent problem to modern static binary rewriting; and while perfection is never guaranteed [59, 69], most common roadblocks are mitigated with further heuristics or engineering.

9 Related Work

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

9.1 Static Binary Rewriting

Static rewriters generally differ by their underlying methodologies. Uroboros [87], Ramblr [86], and RetroWrite [26] reconstruct binary assembly code "reassembleable" by compilers. Others translate directly to compiler-level intermediate representations (IR); Hasabnis et. al [40] target GCC [34] while McSema [25], SecondWrite [4], and dagger [15] focus on LLVM IR. GTIRB [38] and Zipr [46] implement their own custom IR's. We believe static rewriters with robust, low-level IR's are best-suited to supporting ZAFL.

9.2 Improving Fuzzing Test Case Generation

Research continues to improve test case generation from many perspectives. Input data-inference (e.g., Angora [18], VUzzer [68], TIFF [49]) 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 [71], Skyfire [84]) or dynamically (e.g., pFuzzer [58], NAUTILUS [6], Superion [85], AFLSmart [66]). 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 [54] focuses on data segments triggering rare basic blocks. VUzzer [68] assigns deeper blocks high scores to prioritize test cases reaching them; and QTEP [88] similarly targets code near program faults. ZAFL's feedback-enhancing transformations result in greater path discovery, thus increasing the importance of smart path prioritization.

9.3 Hybrid Fuzzing

Many recent fuzzers are hybrid: using coverage-guided fuzzing for most test cases but sparingly invoking more heavyweight analyses. Angora [18] 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 [74] concolic execution starts when fuzzing coverage stalls; QSYM's [92] instead runs in parallel, as do DigFuzz's [94] and SAVIOR's [19], which improve by prioritizing rare and bug-honing paths, respectively. While this paper's focus is applying performant, compiler-quality transformations to the standard coverage-guided fuzzing loop, we imagine leveraging ZAFL to also enhance the more heavyweight techniques central to hybrid fuzzing.

9.4 Emergent Fuzzing Transformations

LLVM [53] 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 [13] compiles binaries with AddressSanitizer for more effective crash-finding; Angora [18] builds its taint tracking atop DataFlowSanitizer [78]; and SAVIOR [19] uses UndefinedBehaviorSanitizer to steer concolic execution toward bug-exercising paths. We thus foresee increasing desire for sanitizers in binary-only fuzzing, however, their heavyweight nature makes porting them a challenge. RetroWrite [26] reveals the possibility that lightweight versions of sanitizers can be incorporated in the main fuzzing loop while maintaining performance. We expect that such transformations can be realized with ZAFL.

10 Conclusion

ZAFL leverages state-of-the-art binary rewriting to extend compiler-quality instrumentation's capabilities to binary-only fuzzing—with compiler-level performance. We 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. We further show that ZAFL scales well to real-world open- and closed-source software of varying size and complexity, and has Windows binary support.

Our 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, we envision a future where fuzzing is no longer burdened by a disparity between compiler-based and binary instrumentation.

Acknowledgment

We would like to thank our reviewers for helping us improve the paper. This material is based upon work supported by the Defense Advanced Research Projects Agency under Contract No. W911NF-18-C-0019, and the National Science Foundation under Grant No. 1650540.

References

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

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

[3] 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] 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.

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

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

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

[11] 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.

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

[13] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] 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.

[29] 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.

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

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

[32] 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.

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

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

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

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

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

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

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

[40] 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.

[41] 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.

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

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

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

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

[46] 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.

[47] 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.

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

[49] 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.

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

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

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

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

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

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

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

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

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

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

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

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

[68] 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.

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

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

[71] 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.

[72] 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.

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

[74] 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.

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

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

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

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

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

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

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

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

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

[84] 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.

[85] 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.

[86] 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.

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

[88] 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.

[89] 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.

[90] 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.

[91] 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.

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

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

[94] 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.