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, jdh8d, 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 FIBRE: a platform for applying fuzzing-enhancing program transformation to binary-only targets—maintaining compiler-level performance. We showcase FIBRE’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-Dyninst and AFL-QEMU. Across LAVA-M and real-world targets, FIBRE 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 FIBRE 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 [106], libFuzzer [78], and honggFuzz [83].

Most modern fuzzers require access to the target’s source code, embracing compiler instrumentation’s low overhead for high fuzzing throughput [78, 83, 106] and increased crash finding. State-of-the-art fuzzers further use compilers to apply fuzzing-enhancing program transformation that improves target speed [33, 50], makes code easier-to-penetrate [1], or tracks interesting behavior [19]. 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 [70], PIN [60], and QEMU [9]). But while binary instrumentation succeeds in many non-fuzzing domains (e.g., program analysis, emulation, and profiling), available options for binary-only fuzzing are simply unable to uphold both the speed and transformation of their compiler counterparts—limiting fuzzing effectiveness. Despite advances in general-purpose binary instrumentation [10, 42, 49, 69, 96, 97], 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 FIBRE: an instrumentation platform bringing compiler-quality capabilities and speed to x86-64 binary-only fuzzing. We demonstrate how FIBRE facilitates powerful fuzzing enhancements with a suite of five transformations, ported from compiler-based fuzzing contexts. We show how FIBRE’s capabilities improve binary-only fuzzing bug-finding: among evaluations on the LAVA-M corpus and eight real-world binaries, FIBRE 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 FIBRE 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 FIBRE 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 criterion for success, and highlighting where popular binary-only instrumenters fit with respect to our criteria.
- We apply this criterion in designing FIBRE: a platform for state-of-the-art compiler-quality instrumentation—and speed—in binary-only fuzzing. FIBRE’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 FIBRE, and evaluating runtime overhead.
- We demonstrate how FIBRE improves fuzzing effectiveness; on average FIBRE’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 FIBRE supports real-world binaries of varying characteristics, size, complexity, and platform—even those binaries not supported by other instrumenters.
- We open-source FIBRE and all benchmark corpora [2].

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.

Though fuzzers vary by generation (i.e., mutation- [78, 83, 106] or grammar-based [36, 53, 64]), execution monitoring (i.e., white- [18, 23, 37], black- [64, 68, 93], or grey-box [78, 83, 106]), and feedback decision-making strategies (i.e., directed [14, 34, 42, 99] or coverage-guided [15, 33, 78, 80, 83, 106]), 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 [106], honggfuzz [83], and libFuzzer [78]). 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 [33, 50] or introspection [1, 19, 47, 54]. 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 100% for coverage tracing alone [66].

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 [89, 106]. Though advanced fuzzers introduce more heavyweight analyses [8, 19, 82, 104], the core of these approaches remains the standard coverage-
guided fuzzing loop (Figure 1)—amounting to over 90% of their execution time [66]; 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++ [32], Angora [19], CollAFL [33], honggFuzz [83], INSTRIM [50], libFuzzer [78]). 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 [43, 85] are used in fuzzing to elide instrumenting some target basic blocks, thus lowering overall runtime overhead. AFL's [106] 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 [47, 50] 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 [50] 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 [106]. 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 instruc-

### 3.3 Sub-instruction Profiling

Fuzzers struggle to penetrate code guarded by complex predicates like “magic bytes” [75], nested checksums [8], 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 [54], honggFuzz [83], laf-Intel [11]). 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 callee-level context sensitivity [19, 47].

### 4 Binary-only Fuzzing: the Bad & the Ugly

Program transformation has become ubiquitous in compiler-based fuzzers (e.g., AFL++ [48], CollAFL [33], 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, (2) dynamic binary translation, or (3) static binary rewriting.
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fuzzing Appearances</th>
<th>Fuzzing Overhead</th>
<th>Supports Xform</th>
<th>Instrumentation type</th>
<th>Instrumentation invoked</th>
<th>liveness</th>
<th>PIC &amp; PDC</th>
<th>Supported Programs</th>
<th>Supported Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM</td>
<td>[1,7,19,33,48,50,57,</td>
<td>18–32%</td>
<td>✓</td>
<td>static</td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>72,78,80,83,106]</td>
<td></td>
<td></td>
<td>inline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel PT</td>
<td>[8,12,21,38,83,106]</td>
<td>19–48%</td>
<td>✓</td>
<td>dynamic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DynamoRIO</td>
<td>[38,44,81]</td>
<td>&gt;1,000%</td>
<td>✓</td>
<td>hardware</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIN</td>
<td>[46,52,68,75,104]</td>
<td>&gt;10,000%</td>
<td>✓</td>
<td>dynamic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QEMU</td>
<td>[24,48,102,106]</td>
<td>&gt;600%</td>
<td>✓</td>
<td>dynamic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dyninst</td>
<td>[45,59,67,84]</td>
<td>&gt;500%</td>
<td>✓</td>
<td>static</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RetroWrite</td>
<td>[27]</td>
<td>20–64%</td>
<td>×</td>
<td>static</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 [51]). Fuzzing implementations are burdened by the need for costly trace post-processing, which reportedly incurs overheads as high as 50% over compilers [8,21,107]; but despite some optimistic performance improvements [38], 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 [44], PIN [60], and QEMU [9]). Translators generally support many architectures and binary characteristics; and offer deep introspection that simplifies analysis and transformation [48,106]. However, existing dynamic translators attain the worst-known fuzzing performance: recent work shows AFL-QEMU’s average overhead is well over 600% [66], and AFL-DynamoRIO [44] and AFL-PIN [46] 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 [45]). Unfortunately, static rewriting options for binary-only fuzzing are limited. AFL-Dyninst is the most popular, but sees prohibitive-high fuzzing overheads of over 500% [66] 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 [106]; and moreover, it does not overcome the generalizability challenges of prior attempts at reassembleable-assembly (e.g., Uroboros [97], Ramblr [96]), 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 [45]) and AFL-PIN (context sensitivity [104])—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 [46,66]).

**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 transpolining**, (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 [9]. Unfortunately, this requires heavy-lifting to interpret target instructions to the host architecture; and incurs significant runtime overhead, as evidenced by the poor performance of AFL-DynamoRIO/PIN/QEMU [44,46,106]. While translation does facilitate transformations like sub-instruction profiling [48], 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++ [27,96,97], others apply to only position-independent (i.e., relocatable) code and neglect the bulk of software that remains position-dependent [27]; many presume access to debugging symbols (i.e., non-stripped) but this seldom holds true when fuzzing proprietary software [45]; and most are only Linux-compatible, leaving some of the world’s most popular commodity software (Windows 64-bit PE32+) unsupported [27,45,96,97]. 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 (C1) and thus incur orders-of-magnitude performance penalties—before applying any feedback-enhancing transformation. Dyninst and RetroWrite embrace static rewriting but both rely on costlier trampoline-based invocation (C2) and fail to support commodity binary formats and characteristics (C4); and moreover, Dyninst’s liveness-aware instrumentation failed on our evaluation benchmarks (C3). Thus, compiler-quality instrumentation in a binary-only context demands a new approach that satisfies all four criteria.

## 5 The FIBRE 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 **FIBRE**: a compiler-quality instrumenter for x86-64 binary fuzzing. FIBRE 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), FIBRE streamlines instrumentation through four extensible phases, facilitating intuitive implementation and layering of state-of-the-art fuzzing-enhancing program transformations. Below we detail FIBRE’s internal architecture and guiding design principles.

### 5.1 Design Overview

As shown in Figure 2, FIBRE consists of two primary components (1) a static rewriting engine and (2) F1x: our four
IR-modifying phases for integrating compiler-quality instrumentation and fuzzing enhancements. Given a target binary, FIBRE operates as follows:

1. **IR Extraction.** From our (or any compatible) binary rewriter, FIBRE requests an intermediate representation (IR) of the target binary.
2. **Fix.** The resulting IR is then passed to Fix’s four transformation and instrumentation phases:
   - **P1:** Optimization,
   - **P2:** Analysis,
   - **P3:** Point Selection, and
   - **P4:** Application.
3. **Binary Reconstitution.** After Fix applies program transformations and instrumentation at IR-level, FIBRE transfers the modified IR back to the rewriting engine which generates the output binary for fuzzing.

### 5.1.1 Static Rewriting Engine

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

We initially considered re-purposing the LLVM IR-based rewriter McSema [26] due to its maturity and popularity in the static rewriting community, but ultimately ruled it out as both the literature [30] and our 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 binary rewriter Zipr [42,49] as it meets the same criteria that McSema does (§ 4.2), but has better baseline performance. Our hope is that the McSema maintainers will improve their rewriter based on the properties uncovered in this paper.

### 5.2 The Fix Transformation Architecture

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

#### 5.2.1 Optimization

Fix’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, Fix’s second phase computes meta-characteristics (e.g., predecessor-successor, data-flow, and dominance relationships). We model this after existing compiler mechanisms [4,25,65], and to facilitate integration of other desirable analyses appearing in the literature [3,91]. The extent of possible analyses depends on the rewriter’s IR; for example, low-level IR’s modeled after GCC’s RTL [35] 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 [65] and could be added to support Fix.

#### 5.2.3 Point Selection

Fix’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. Fix’s CFG-aware instrumentation pruning capabilities facilitate easy implementation of compiler-based techniques described in § 3.

#### 5.2.4 Application

Finally, Fix’s applies the desired instrumentation configuration (e.g., block or edge coverage tracking). A challenge is identifying how to instrument each location; ensuring correct
Table 3: A catalog of FIBRE-implemented compiler-quality fuzzing-enhancing program transformations and their compiler-based origins.

### Performance Transformation

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Successor-based Pruning</td>
<td>[47]</td>
</tr>
<tr>
<td>Dominator-based Pruning</td>
<td>[50]</td>
</tr>
<tr>
<td>Instrumentation Downgrading</td>
<td>[33]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feedback Transformation</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub-instruction Profiling</td>
<td>[1,48,54,83]</td>
</tr>
<tr>
<td>Context-sensitive Coverage</td>
<td>[19,47]</td>
</tr>
</tbody>
</table>

Recent fuzzing works leverage flow graph reducibility techniques [43,85] to cut down instrumentation overhead [50]. 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 [45]. 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 FIX’s Analysis phase; and a location selector during Point Selection to omit basic blocks accordingly.

#### 6.1 Performance-enhancing Transformations

We leverage FIBRE’s FIX 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 [43,85] to cut down instrumentation overhead [50]. 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 [45]. 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 FIX’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 [91] 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 [3]. 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 FIX’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 [3,50], which we believe are also implementable in FIBRE.

##### 6.1.3 Edge Instrumentation Downgrading

CollAFL [33] 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 FIX’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 [19,47]. Below we detail our FIBRE 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 [54]) 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 Fix’s Optimization phase, we scan the IR for comparison mnemonics (i.e., \texttt{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 [19] or function-level [47]. 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 Fix’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 FIBRE enable compiler-style program transformations while maintaining performance?

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

Q3: Does FIBRE support real-world, complex targets?

We first perform an evaluation of FIBRE against the leading binary-only fuzzing instrumenters AFL-Dyninst and AFL-QEMU on the LAVA-M benchmark corpus [29]. 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 FIBRE alongside AFL-Dyninst and AFL-QEMU in fuzzing five varied closed-source binaries. Fifth, we test FIBRE’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 FIBRE’s precision.

7.1 Evaluation-wide Instrumenter Setup

We evaluate FIBRE against the fastest-available binary-only fuzzing instrumenters; we thus omit AFL-PIN [46, 71, 90] and AFL-DynamoRIO [44, 81, 92] variants as their reported overheads are much higher than AFL-Dyninst’s and AFL-QEMU’s; and Intel PT [51] 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 FIBRE.

AFL-Dyninst: A recent AFL-Dyninst update [45] 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). 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.


FIBRE: 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 FIBRE; 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 [28].

1This AFL-Dyninst update [45] 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.
We do not include FIBRE’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 FIBRE’s sub-instruction profiling transformation based on number of queued (i.e., coverage-increasing) test cases. Table 4 shows FIBRE’s mean relative increase in triaged crashes, total and queued test cases over AFL-Dyninst and AFL-QEMU per configuration.

**FIBRE versus AFL-Dyninst:** Across all 16 configurations FIBRE 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 FIBRE averages slightly fewer (4–12%) test cases on md5sum this is not to its disadvantage: FIBRE queues 3100–7600% more test cases and finds well over 300% more crashes, thus revealing the value of its control-flow-optimizing program transformations.

**FIBRE versus AFL-QEMU:** FIBRE matches or surpasses AFL-QEMU among 15 benchmark configurations, averaging 42% more crashes and 203% more test cases seen. As with AFL-Dyninst, FIBRE 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 FIBRE. FIBRE finds 13% fewer crashes relative to AFL-QEMU on md5sum with empty seeds and dictionary, but as FIBRE’s queue is 91% larger, we believe this specific seed/dictionary configuration and FIBRE’s transformations result in a “burst” of hot paths, which the fuzzer struggles to prioritize. Such occurrences are rare given FIBRE’s superiority in other trials, and likely correctable through orthogonal advancements in fuzzing path prioritization [15, 22, 58, 108].

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 FIBRE does (see § 7.5).

**FIBRE versus AFL-LLVM:** To gain a sense of whether FIBRE’s transformation is comparable to existing compiler-based implementations, we ran FIBRE alongside the the analogous configuration of AFL’s LLVM instrumentation with its INSTRIM [50] 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 car-
We apply all FIBRE-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 [55,61,73]. Table 5 shows FIBRE-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.

**FIBRE versus AFL-Dyninst:** Our results show FIBRE averages 26% more real-world crashes and 48% more test cases than AFL-Dyninst in 24 hours. Though FIBRE 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 FIBRE 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 FIBRE’s compiler-quality transformations bear a statistically significant advantage over AFL-Dyninst.

**FIBRE versus AFL-QEMU:** While FIBRE surpasses AFL-QEMU’s LAVA-M crash-finding by 42%, FIBRE’s real-world crash-finding is an even higher 131%. Apart from the two `readelf` bugs found by all three instrumenters, FIBRE’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 FIBRE’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.

### 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).

### Real-world Crash-finding

We apply all FIBRE-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 [55,61,73]. Table 5 shows FIBRE-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.

**FIBRE versus AFL-Dyninst:** Our results show FIBRE averages 26% more real-world crashes and 48% more test cases than AFL-Dyninst in 24 hours. Though FIBRE 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 FIBRE 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 FIBRE’s compiler-quality transformations bear a statistically significant advantage over AFL-Dyninst.

**FIBRE versus AFL-QEMU:** While FIBRE surpasses AFL-QEMU’s LAVA-M crash-finding by 42%, FIBRE’s real-world crash-finding is an even higher 131%. Apart from the two `readelf` bugs found by all three instrumenters, FIBRE’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 FIBRE’s increased effectiveness is statistically significant. Furthermore the disparity between AFL-QEMU’s LAVA-M and real-world crash-finding suggests that increasingly-complex binaries heighten the need for more powerful binary rewriters.

#### Real-world Coverage-tracing Overhead

For our coverage-tracing overhead evaluation we follow established practice [66]: we collect 5×24-hour test case dumps per benchmark; instrument a forkserver-only [105] “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

---

2. In stack hashing we consider both function names and lines; and condense recursive calls as they would otherwise over-approximate bug counts.
scale the resulting overheads relative to baseline.

We compare Fibre to AFL-Dyninst, AFL-QEMU, and to the compiler- and assembler-based instrumentation available in AFL [106]. We assess all aspects of Fibre’s performance: (1) its baseline forkserver-only rewritten binary overhead (Fibre-FSsrvr); and instrumentation overheads (2) with no transformations (Fibre-NONE), (3) only performance-enhancing transformations (Fibre-PERF), and (4) all (Table 3) transformations (Fibre-ALL). We additionally compute geometric mean Mann-Whitney U p-values of both Fibre-NONE’s and Fibre-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, Fibre-FSsrvr, Fibre-NONE, Fibre-PERF, and Fibre-ALL obtain overheads of 5%, 32%, 17%, and 27%, while compiler and assembler instrumentation average 25% and 34%, and AFL-Dyninst and AFL-QEMU average 88% and 256%, respectively. Thus, even Fibre with all fuzzing-enhancing transformations approaches compiler performance.

**Fibre versus AFL-Dyninst:** We observe Fibre 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 Fibre’s most heavyweight configuration (Fibre-ALL) incurs 61% less average overhead than AFL-Dyninst, even though this comparison includes Fibre’s performance-enhancing transformations. If omitted, this still leaves Fibre ahead of AFL-Dyninst—which, too, benefits from performance-enhancing single successor-based pruning. Comparing the execution times of Fibre-NONE and Fibre-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 Fibre, both with- and without-transformations, achieves statistically better performance over AFL-Dyninst.

**Fibre versus AFL-QEMU:** Though AFL-QEMU’s block caching reduces its overhead from previous reports [66], Fibre 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 Fibre’s performance across all benchmarks. Our Mann-Whitney U tests reveal that both Fibre-NONE and Fibre-ALL obtain p-values of 0.012, suggesting that Fibre achieves statistically better performance over AFL-QEMU.

**Comparing Fibre to Compiler Instrumentation:** On average, compared to a forkserver-only binary, Fibre 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 25% overhead of AFL’s compiler-based instrumentation, and slightly better than AFL’s assembler-based trampolining overhead of 34%. Comparing Fibre-NONE and Fibre-ALL to compiler instrumentation yields mean Mann-Whitney U p-values ranging 0.12–0.18 which, being larger than 0.05, suggests that Fibre is indistinguishable from compiler-level performance.

### 7.4 Fuzzing Closed-source Binaries

To evaluate whether Fibre’s improvements extend to true binary-only use cases, we expand our evaluation with five diverse, closed-source binary benchmarks. Our results show that Fibre’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 Fibre 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 Fibre (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.

**Fibre versus AFL-Dyninst:** Despite AFL-Dyninst being faster on idat64, nconvert, nvdisasm, and unrar, Fibre 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
AFL-QEMU vs. FIBRE:

FIBRE versus AFL-QEMU: FIBRE’s speed and transformations enable it to average 38% more triaged crashes and 52% more test cases than AFL-QEMU. While FIBRE 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 FIBRE’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 FIBRE and AFL-QEMU is not overwhelming, FIBRE successfully uncovers a heap overread crash in idat64—while AFL-QEMU finds nothing.

7.5.3 Bug-finding Case Study

Following additional manual triage with binary-level memory error checkers (e.g., QASan [31] and Dr. Memory [17]), we compare the time-to-discovery (TTD) for five closed-source binary bugs found by FIBRE, 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 overflow 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, FIBRE finds these bugs 660% faster than AFL-Dyninst, and 113% faster than AFL-QEMU. Thus, FIBRE’s balance of compiler-quality transformation and performance lends a valuable asset to bug-finding in closed-source code.

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 FIBRE’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 FIBRE 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 FIBRE’s AFL-like configuration; we do the same for Windows binaries using FIBRE’s cross-instrumentation support. We test instrumented binaries either with our automated regression test suite (used throughout FIBRE’s development); or by manually running the application (for Windows) or testing the instrumentation output with afl-showmap [106] (for Linux).

We verify FIBRE achieves success on 33 open-source Linux and Windows binaries, shown in Table 8. To confirm FIBRE’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 FIBRE 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).
As discussed in § 4.2, register liveness analysis enables optimized instrumentation insertion for closer-to-compiler-level speed. While liveness false positives introduce overhead from the additional instructions needed to save/restore registers, liveness false negatives may leave live registers erroneously overwritten—potentially breaking program functionality. If FIBRE’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 FIBRE’s speed with/without liveness-aware instrumentation. As Figure 6 shows, liveness-unaware FIBRE faces 31% more overhead across all benchmarks. While 13–16% slower than AFL-Dyninst on bsdtar and sfconvert, FIBRE’s unoptimized instrumentation still averages 25% and 193% less overhead than AFL-Dyninst and AFL-QEMU, respectively. Thus, even in the worst case FIBRE generally outperforms other binary-only fuzzing instrumenters.

7.5.2 Liveness-aware Optimization

<table>
<thead>
<tr>
<th>Application</th>
<th>OS</th>
<th>Binary</th>
<th>Size</th>
<th>Blocks</th>
<th>P*C</th>
<th>Sym</th>
<th>Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>L</td>
<td>httpd</td>
<td>1.0M</td>
<td>55,559</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AudioFile</td>
<td></td>
<td>sfcvnt</td>
<td>568K</td>
<td>5,814</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BIND</td>
<td></td>
<td>named</td>
<td>9.4M</td>
<td>120,665</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Binutils</td>
<td></td>
<td>readelf</td>
<td>1.4M</td>
<td>21,085</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CallBoost</td>
<td></td>
<td>callbo</td>
<td>153M</td>
<td>1,308,249</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cJSON</td>
<td></td>
<td>cjson</td>
<td>43K</td>
<td>1,490</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Clang</td>
<td></td>
<td>clang</td>
<td>36.4M</td>
<td>1,750,126</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DNSSMasq</td>
<td></td>
<td>dnsmasq</td>
<td>373K</td>
<td>20,302</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Gumbo</td>
<td></td>
<td>clean_text</td>
<td>571K</td>
<td>5,008</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JanPee</td>
<td></td>
<td>jasper</td>
<td>1.1M</td>
<td>14,795</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libarchive</td>
<td></td>
<td>bsdtar</td>
<td>2.1M</td>
<td>29,868</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libjpeg</td>
<td></td>
<td>djjpeg</td>
<td>667K</td>
<td>5,066</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libksba</td>
<td></td>
<td>corn-bas</td>
<td>435K</td>
<td>5,247</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lighttpd</td>
<td></td>
<td>lighttpd</td>
<td>1.1M</td>
<td>12,558</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mosh</td>
<td></td>
<td>mosh-cli</td>
<td>4.2M</td>
<td>14,311</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NGINX</td>
<td></td>
<td>nginx</td>
<td>4.8M</td>
<td>29,507</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OpenSSH</td>
<td></td>
<td>sshd</td>
<td>2.3M</td>
<td>33,115</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OpenVPN</td>
<td></td>
<td>openvpn</td>
<td>2.9M</td>
<td>34,521</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Poppler</td>
<td></td>
<td>pdftot</td>
<td>1.5M</td>
<td>2,814</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Redis</td>
<td></td>
<td>redis-ser</td>
<td>5.7M</td>
<td>74,515</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Samba</td>
<td></td>
<td>smbclient</td>
<td>226K</td>
<td>6,279</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SIPwitch</td>
<td></td>
<td>sipcontrol</td>
<td>226K</td>
<td>772</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Squid</td>
<td></td>
<td>squid</td>
<td>32.7M</td>
<td>212,746</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tcpdump</td>
<td></td>
<td>tcpdump</td>
<td>2.3M</td>
<td>24,451</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>thirtdp</td>
<td></td>
<td>thirtdp</td>
<td>119K</td>
<td>3,428</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>UnRTF</td>
<td></td>
<td>unrtf</td>
<td>170K</td>
<td>1,657</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>7-Zip</td>
<td></td>
<td>7z</td>
<td>447K</td>
<td>23,353</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AcelPad</td>
<td>W</td>
<td>AcelPad</td>
<td>540K</td>
<td>31,140</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cygwin64</td>
<td>W</td>
<td>bash</td>
<td>740K</td>
<td>38,397</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cygwin64</td>
<td>W</td>
<td>ls</td>
<td>120K</td>
<td>5,661</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fre:ac</td>
<td>W</td>
<td>freacmd</td>
<td>97K</td>
<td>521</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fremedia</td>
<td>W</td>
<td>fremedia</td>
<td>176K</td>
<td>3,106</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fmedia</td>
<td>W</td>
<td>fmedia-pa</td>
<td>173K</td>
<td>1,363</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 8: Open-source binaries tested successfully with FIBRE. \( L/W = \) Linux/Windows; \( \text{Opt} \) = whether register liveness-aware optimization succeeds.

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)).

<table>
<thead>
<tr>
<th>Binary</th>
<th>Total Insns</th>
<th>IDA Pro</th>
<th>IDA Pro</th>
<th>IDA Pro</th>
<th>IDA Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>作了</td>
<td>补救</td>
<td>补救</td>
<td>补救</td>
</tr>
</tbody>
</table>

Table 10: Instruction recovery statistics for IDA Pro, Binary Ninja, and FIBRE, with ground-truth disassembly from LLVM-10’s objdump. \( \text{Reached} \) = mean unrecovered instructions reached by fuzzing (hence, erroneously-unrecovered); \( \text{FalseNeg} \) = erroneously-unrecovered instructions over total.

We evaluate FIBRE’s instruction recovery using ground-truth disassemblies of binary .TEXT sections generated by objdump, which is shown to achieve \( \sim 100\% \) accuracy [6] (specifically, we use the version shipped in LLVM-10 [56]). To see how FIBRE 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 trials for our eight open-source benchmarks.

Though we face some challenges in precise code/data disambiguation (DRM) protections—both of which, while uncommon, appear in many proprietary and commercial applications. While neither FIBRE nor its rewriter currently support obfuscated or DRM-protected binaries, a growing body of research is working toward overcoming these obstacles [13, 101]. Thus, we believe that with new advances in binary deobfuscation and DRM-stripping, FIBRE will achieve broader success across other Windows binary formats and versions.

Another grey area for static binary rewriters is deprecated Windows binary formats and versions. While FIBRE 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, FIBRE will achieve broader success across other Windows binary formats and versions.


table 10 lists the total instructions; and total and reached unrecovered instructions per our five closed-source benchmarks. 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, FIBRE 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 FIBRE’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 FIBRE’s control-flow recovery, we run all test cases generated over five 24-hour trials for our eight open-source benchmarks on both a FIBRE- and a ground-truth LLVM-instrumented binary, and log when each report new coverage.

<p>|</p>
<table>
<thead>
<tr>
<th>Binary</th>
<th>Coverage TPR</th>
<th>Coverage TNR</th>
<th>Coverage Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsdar</td>
<td>97.28%</td>
<td>&gt;99.99%</td>
<td>&gt;99.99%</td>
</tr>
<tr>
<td>cert-basic</td>
<td>96.67%</td>
<td>&gt;99.99%</td>
<td>&gt;99.99%</td>
</tr>
<tr>
<td>clean_text</td>
<td>96.39%</td>
<td>&gt;99.99%</td>
<td>&gt;99.99%</td>
</tr>
<tr>
<td>jasper</td>
<td>98.82%</td>
<td>&gt;99.99%</td>
<td>&gt;99.99%</td>
</tr>
<tr>
<td>readelf</td>
<td>99.98%</td>
<td>&gt;99.99%</td>
<td>&gt;99.99%</td>
</tr>
<tr>
<td>sfconvert</td>
<td>98.71%</td>
<td>&gt;99.99%</td>
<td>&gt;99.99%</td>
</tr>
<tr>
<td>tcpdump</td>
<td>98.51%</td>
<td>&gt;99.99%</td>
<td>&gt;99.99%</td>
</tr>
<tr>
<td>unrfl</td>
<td>94.17%</td>
<td>&gt;99.99%</td>
<td>&gt;99.99%</td>
</tr>
</tbody>
</table>

Mean 97.30% 100.00% 100.00%

Table 11: FIBRE’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, FIBRE’s coverage identification is near-identical to LLVM’s: achieving 97.3% sensitivity, ~100% specificity, and ~100% accuracy. While FIBRE 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 FIBRE eventually finds replacement test cases, thus, FIBRE reaches every bug reached by LLVM. Thus, we conclude that FIBRE succeeds in preserving the control-flow of compiler-generated code.

8 Limitations

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

8.1 Improving Baseline Performance

Our performance evaluation § 7.3.4 shows FIBRE’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 FIBRE’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 FIBRE 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 FIBRE 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 FIBRE-side engineering is needed to achieve soundness. We leave instrumentalizing non-C/C++ languages for future work.

While FIBRE 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, FIBRE 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 FIBRE nor its rewriter currently support obfuscated or DRM-protected binaries, a growing body of research is working toward overcoming these obstacles [13, 101]. Thus, we believe that with new advances in binary deobfuscation and DRM-stripping, FIBRE 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 FIBRE and simply ignored. We recognize there are trade-offs between static binary rewriting generalizability and precision, and leave addressing such gaps as future work.

Most modern static binary rewriters perform their core analyses—disassembly, code/data disambiguation, and indirect branch target identification—via third-party tools like Capstone [74] and IDA [40], consequently inheriting their limitations. For example, if the utilized disassembler is not up-to-date with the latest x86 ISA extension [100], 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 [63, 76], 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 [97], Ramblr [96], and RetroWrite [27] reconstruct binary assembly code “reassemblable” by compilers. Others translate directly to compiler-level intermediate representations (IR); Hasabnis et al [41] target GCC [35] while McSema [26], SecondWrite [5, 69], and dagger [16] focus on LLVM IR. GTIRB [39] and Zipr [49] implement their own custom IR’s. We believe static rewriters with robust, low-level IR’s are best-suited to supporting FIBRE.

9.2 Improving Fuzzing Test Case Generation

Research continues to improve test case generation from many perspectives. Input data-inference (e.g., Angora [19], VUzzer [75], TIFF [52], ProFuzzer [103]) 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 [79], Skyfire [94]) or dynamically (e.g., pFuzzer [62], NAUTILUS [7], Superior [95], AFLSmart [73]). Such approaches mainly augment fuzzing at the mutator-level, and thus complement FIBRE’s compiler-quality instrumentation in binary-only contexts. Another area of improvement is path prioritization. AFLFast [15] allocates mutation to test cases exercising deep paths. FairFuzz [58] focuses on data segments triggering rare basic blocks. VUzzer [75] assigns deeper blocks high scores to prioritize test cases reaching them; and QTEP [98] similarly targets code near program faults. FIBRE’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 [19] uses taint tracking to infer mutation information, but runs all mutates in the standard fuzzing loop; REDQUEEN [8] operates similarly but forgoes taint tracking for program state monitoring. Driller’s [82] concolic execution starts when fuzzing coverage stalls; QSYM’s [104] instead runs in parallel, as do DigFuzz’s [108] and SAVIOR’s [20], which improve by prioritizing rare and bug-honing paths, respectively. EnFuzz [22] fuses multiple coverage-guided and hybrid techniques. While this paper’s focus is applying performant, compiler-quality transformations to the standard coverage-guided fuzzing loop, we imagine leveraging FIBRE to also enhance the more heavyweight techniques central to hybrid fuzzing.

9.4 Emergent Fuzzing Transformations

LLVM [56] offers several robust “sanitizers” useful for software debugging (e.g., [77, 86, 88]). 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 [14] compiles binaries with AddressSanitizer for more effective crash-finding; Angora [19] builds its taint tracking atop DataFlowSanitizer [87]; and SAVIOR [20] 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 [27] 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 FIBRE.

10 Conclusion

FIBRE 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, FIBRE 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 FIBRE 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.

References


