BINVARIANTS: Enhancing Fuzzing of Closed-Source Binary Executables via Register-Level Likely Invariants

ZAO YANG, University of Utah, USA
STEFAN NAGY, University of Utah, USA

Closed-source software is ubiquitous in everyday computing, underscoring the need for robust security vetting of "binary-only" executable code. While code-coverage-guided fuzzing has long proven effective at unearthing software bugs, fuzzing in open-source contexts has since evolved *beyond* code coverage as its principal guiding metric. State-of-the-art fuzzing advancements demonstrate that *likely data invariants*—data-level properties which, if violated, expose unusual and often bug-preceding program states—significantly widen fuzzing's reach to defects ordinarily occluded by coverage-only testing. Unfortunately, current invariant-guided fuzzing universally depends on source-level abstractions, rendering it unportable to binary-only targets. Consequently, closed-source software fuzzing—and more importantly, binary-only bug discovery—remain stalled at now-obsolete coverage-only techniques, even as open-source software fuzzing advances well past them.

To bridge this longstanding gap, this paper introduces *register-level likely invariants*: the first technique to integrate likely data invariants *within* binary-only fuzzing. In contrast to contemporary source-level data invariant mining, our approach operates directly on CPU registers, capturing the low-level program states that themselves encode higher-level data relationships. From these low-level states, we automatically derive likely data invariants and expose their violations as fuzzer-observable signals via runtime instrumentation, steering fuzzing into states often unreachable by code coverage alone. In doing so, our approach surfaces qualitatively different states, complementing traditional coverage-guided fuzzing with distinct bug-finding capabilities.

We implement our approach as a prototype, BINVARIANTS, and evaluate its performance across 25 benchmark applications: 7 closed-source, as well as 18 open-source programs compiled as binary-only executables. Our results show that, compared to driving binary fuzzing solely via code coverage, register-level likely invariants helps fuzzing trigger over 27× more unique invariant violations beyond coverage-only fuzzing, thereby exercising a mean 52% more distinct code regions. Moreover, our approach uncovers 143 total bugs versus coverage-only fuzzing's 137—including 20 missed by code coverage—demonstrating how register-level likely invariants extends binary-only fuzzing's reach into execution states beyond what coverage alone is capable of.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

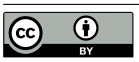
Additional Key Words and Phrases: Fuzzing, Binaries, Likely Invariants

ACM Reference Format:
Zao Yang and Stefan Nagy. 2026. BINVARIANTS: Enhancing Fuzzing of Closed-Source Binary Executables via Register-Level Likely Invariants. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE050 (July 2026), 24 pages. https://doi.org/10.1145/3797078

1 Introduction

Closed-source (or "binary-only") programs—software that *lacks* publicly-available source code—are vital to many critical sectors such as government [86], healthcare [27], telecommunications [71], and finance [106], making them among today's most significant targets for cyberattacks [43, 62, 73, 91]. Unlike open-source software, where third-party involvement often drives discovery and

Authors' Contact Information: Zao Yang, University of Utah, Salt Lake City, USA, zao.yang@utah.edu; Stefan Nagy, University of Utah, Salt Lake City, USA, snagy@cs.utah.edu.



ACM 2994-970X/2026/7-ARTFSE050
https://doi.org/10.1145/3797078

fixing of software defects [103], closed-source applications depend solely on *their own vendors* for security updates, consequently leaving many undiscovered flaws ripe for attackers. This asymmetry underscores the need for proactive vetting to unearth critical errors in closed-source software *before* attackers exploit them first. To this end, *coverage-guided fuzzing* [82] has emerged as among today's most effective automated testing techniques for vetting closed-source software.

Coverage-guided fuzzers (e.g., AFL++ [60], libFuzzer [80]) operate by repeatedly generating and executing millions of test cases on the target program, leveraging compile-time instrumentation (e.g., LLVM [76]) to identify those that increase code coverage—such as new control-flow edges, basic blocks, or execution counts—and thereby steer exploration toward previously unseen code regions. Over the last decade, advancements in binary instrumentation [34, 48, 72, 89] have broadly facilitated code coverage tracking in the absence of source code, driving coverage-guided fuzzing to become one of the most widely used techniques for vetting closed-source software today. Yet, code coverage on its own is blind to *data*-level program states, which expose how inputs propagate through the program and influence its underlying behavior [81, 83, 95, 100, 108]. To this end, recent advancements in open-source software fuzzing are supplementing coverage with *likely data invariants*: observed data-level properties which, if violated, constitute unusual—and often bug-preceding—program states, empowering fuzzers to find many more bugs than discoverable with code coverage alone [57, 70].

Unfortunately, despite its demonstrated advantages, invariant-guided fuzzing remains largely sidelined in binary-only settings due to three fundamental challenges. First, existing invariant-mining techniques [57, 70] rely on source-level variables and type information, which are wholly absent in binaries [84], making it difficult to recover meaningful abstractions of data-level program states. Second, adapting mechanisms for detecting invariant violations from source-level settings is nontrivial, as injecting new logic into closed-source binaries' existing instruction streams is inherently fragile and risks breaking execution [102]. Finally, minimizing runtime costs is especially critical for binary-only fuzzing [89, 90], where already high baseline overheads are easily compounded by additional instrumentation, reducing throughput and impeding timely bug discovery. Altogether, these challenges preclude the direct application of practical, high-performance invariant-guided fuzzing in closed-source settings—confining binary-only security vetting to now-obsolete coverage-only approaches that frequently *miss* more complex program vulnerabilities.

To overcome these challenges, this paper introduces **register-level likely invariants**: the first extension of likely data invariants for guiding binary-only fuzzing. Our approach exploits the role of CPU registers in capturing actively changing program state, providing a direct, low-level view of how input data is processed and propagated throughout execution. By monitoring register values at basic block boundaries, we infer likely invariants that encode value constraints and relationships across registers, thereby approximating expected program behaviors without relying on source-level abstractions. Building on this foundation, we incorporate likely invariants as a new source of feedback for binary-only fuzzing, dynamically tracking register states and surfacing invariant violations as observable signals alongside conventional code coverage. To maximize efficiency, we further restrict invariant inference and checking to only execution-relevant registers, reducing noise and avoiding redundant checks toward faster binary-only bug discovery. Collectively, our approach bridges invariant-guided fuzzing and binary-only settings, exposing data-level program behaviors that, until now, have remained beyond the reach of existing coverage-only binary fuzzing.

We implement register-level likely invariants as a prototype, BINVARIANTS, atop state-of-the-art binary fuzzer AFL++ [60] and instrumentation framework QEMU [34]; and evaluate it alongside coverage-only binary fuzzing with AFL++ across a suite of 25 benchmarks, spanning 18 open-source programs compiled as binary-only targets as well as 7 closed-source applications. Our results show that, across five 48-hour fuzzing campaigns per benchmark, register-level likely invariants drives testing toward qualitatively different program states, resulting in a mean of **27×** more

uniquely violated invariants and **52%** more uniquely covered code compared to coverage-only fuzzing. In turn, this increased state diversity enables BINVARIANTS to surface **20** bugs undiscovered by coverage-guided binary fuzzing, underscoring register-level likely invariants' effectiveness in exposing critical, bug-preceding data-level behaviors ordinarily missed by binary fuzzing.

In summary, our paper makes the following contributions:

- We introduce *register-level likely invariants*, capitalizing on the insight that CPU registers expose low-level program state and enable direct reasoning over data-level behaviors in binary programs.
- We extend register-level likely invariants to develop the first approach for incorporating invariant-violation feedback in binary fuzzing, overcoming the challenges of inferring, injecting, and efficiently checking invariants without the source-level semantics relied upon by prior approaches.
- We implement our approach as a prototype, BINVARIANTS, atop AFL++ and QEMU, and evaluate it against coverage-only binary fuzzing. Overall, we show BINVARIANTS steers fuzzing toward qualitatively different program states, achieving a mean 27× more uniquely violated invariants and 52% more uniquely covered code, and uncovering 20 bugs missed by coverage-guided binary fuzzing.
- We publicly release BINVARIANTS and our evaluation artifacts at the following URL to enable future research and advancement of binary-only fuzzing: https://github.com/FuturesLab/Binvariants.

2 Background and Motivation

This section provides an overview of closed-source software fuzzing, likely invariants and invariant-guided fuzzing, and the challenges of adapting invariant-guided fuzzing to closed-source software.

2.1 Security Risks of Closed-Source Software

Closed-source (or binary-only) software refers to software artifacts whose source code is not publicly accessible, often due to commercialization [10, 28], intellectual property protection [49, 55, 61], or legacy components for which source code has been lost or is otherwise unavailable [44]. In practice, closed-source software libraries [5], applications [28], and operating systems [32, 85] are prevalent across many of today's most security-critical sectors, including government [86], healthcare [27], telecommunications [71], and finance [106], making them among the most targeted classes of software by cyberattacks [43, 62, 73, 91]. As recent industry research shows that closed-source code makes up 65% of all software [69] and contains nearly 20% more vulnerabilities [79], scalable and proactive security vetting is more critical than ever for mitigating this high-value attack surface.

Unfortunately, the opaque nature of closed-source software significantly complicates security analysis. While today's open-source software ecosystem benefits from broad third-party participation in vulnerability discovery and responsible disclosure [31], traditional vetting approaches such as static analysis and manual security assessments are substantially hindered without source access, often allowing critical flaws in closed-source software to remain undiscovered until exploitation [92, 94]. To overcome these limitations, closed-source software developers and security researchers alike are increasingly adopting *coverage-guided fuzzing*: a high-volume automated testing strategy for scalable vulnerability discovery and security vetting of software components [82].

2.2 Coverage-Guided Fuzzing for Binaries

Fuzzing (or fuzz testing) is an automated testing technique that operates by repeatedly generating and executing millions of randomly mutated test cases on a target program, systematically exploring its behavior to uncover hidden, input-triggered vulnerabilities [82]. While early fuzzers [51, 87] embraced a wholly black-box approach—blindly feeding inputs to a target program and observing its outputs—modern fuzzers leverage *code coverage* guidance: deploying software instrumentation (e.g., via LLVM [76]) to measure each test case's coverage of the program's internals, and retaining and mutating only those that exercise *novel* coverage. Today, coverage-guided fuzzers such as

AFL++ [60] and libFuzzer [80] are widely used across virtually every software ecosystem, with large-scale fuzzing initiatives such as Google's OSS-Fuzz [103] and Linux's SyzKaller [64] having uncovered numerous critical vulnerabilities in thousands of widely used software components.

While most coverage-guided fuzzing targets open-source software—adding coverage-tracking instrumentation at compilation time [60, 80]—academic [48, 50, 89, 110] and industrial [25, 26, 34] innovations have broadly extended fuzzing to closed-source software, primarily through advancements in binary instrumentation: mechanisms for inserting fuzzing's requisite coverage-tracking mechanisms into pre-compiled, closed-source binaries. Among available binary instrumentation techniques, dynamic translation (e.g., QEMU [34], Unicorn [26], Frida [25]) remains the most popular and well-supported by current fuzzers [60, 66] due to its flexibility and extensive architectural support, inserting coverage-tracking instrumentation at runtime on each executed basic block. Yet, while coverage-guided fuzzing remains today's most popular approach for testing closed-source, binary-only code [67, 68, 105], open-source software fuzzing has progressed beyond coverage-only exploration to additionally target *data*-level program behaviors—via *likely data invariants* [57, 70].

2.3 Guiding Fuzzing via Likely Data Invariants

At a high level, data *invariants* are properties of program states that are upheld during execution, capturing relationships among source variables (e.g., `idx < size`) as well as constraints on their values (e.g., `ptr != NULL`). While *true* invariants describe properties that hold across *all* possible executions, they are generally difficult to determine precisely—especially for complex software with incomplete or unclear specifications [52]. To address this, prior work introduces *likely* invariants [42, 53], which approximate true invariants by analyzing program executions over a set of concrete inputs, inferring properties that consistently hold across all observed runs. Today, invariant-mining tools such as Daikon [52, 54] and DIG [93] automate this process by instrumenting programs to produce runtime traces over user-provided inputs, from which likely invariants are then inferred.

```
1  // mlen: from input; not bounds-checked against buffer w
2  while ((mlen -= 32) >= 0) {
3      sum += w[0]; . . .  sum += w[15];      // OOB read
4      w += 16;
5  }
```

Invariant: $32 \leq$ `mlen` ≤ 64

input	`mlen`	overread	new cov.?	inv. met?
i_1	32	0	✔	✔
i_2	44	0	✗	✔
i_3	64	0	✗	✔
i_4	72	0	✗	✗
i_5	96	**32**	✗	✗

Fig. 1. Example invariant-driven bug in TCPDump [21] file `in_cksum` [20]: `mlen` is initialized from an attacker-influenced packet length, and drives an unrolled loop that decrements `mlen` by 32 per iteration, while accessing buffer `w` without bounds checking. Monitoring `mlen` across seed inputs i_1–i_3 yields the likely invariant $32 \leq$ `mlen` ≤ 64. Inputs i_2–i_4 traverse the same loop edges as i_1 and are therefore redundant under coverage-guided fuzzing; however, i_4 violates the invariant. This distinction is invisible to coverage-only fuzzing, while *invariant*-guided fuzzing retains i_4 and mutates it into i_5, whose larger `mlen` triggers the out-of-bounds read.

In practice, invariant *violations* often reveal unusual data-level program states that precede complex, state-dependent vulnerabilities. For example, Figure 1 highlights a vulnerability in TCPDump [21] missed by coverage-only fuzzing in our evaluation (Table 5), where progressively larger values beyond the inferred range `[32,64]` eventually trigger an out-of-bounds read. Accordingly, recent fuzzing advancements [57, 70] are now leveraging likely invariants as a source of runtime feedback, guiding the fuzzer to *violate* them and thereby surface bugs beyond what coverage-only fuzzing exposes. In the prevailing state-of-the-art approach introduced by Fioraldi et al. [57], invariants are first mined from pre-generated inputs (e.g., via Daikon [52]), and subsequently encoded into the program as fuzzer-reachable conditions; violations are exposed as runtime signals analogous to new code coverage (e.g., § 2.2), causing the fuzzer to retain and re-mutate the corresponding inputs to further explore program states—and uncover bugs—that deviate from expected behavior.

2.4 Motivation: Challenges of Adapting Invariant-Guided Fuzzing to Binaries

Although likely invariants extend fuzzing's reach beyond program states accessible through code coverage alone, current implementations [57, 70] remain tightly coupled to open-source software—precluding the use of invariant-guided fuzzing in binary-only contexts to date. In the following, we outline the fundamental challenges that hinder the extension of invariant-based feedback to binary-only fuzzing: invariant inference, invariant injection, and upholding speed.

2.4.1 Challenge 1: Invariant Inference. As § 2.3 discusses, traditional invariant inference tooling [52, 93] depend on higher-level source constructs, with no existing techniques supporting invariant extraction from binary-only software. While program decompilation [41] and lifting [30] attempt to recover higher-level representations (e.g., C, LLVM IR) from binaries, current tools seldom achieve soundness due to the inherent challenges of reconstructing precise control flow, data flow, and type information from low-level code, compounded by compiler optimizations and indirect control transfers [78, 96, 113]. Consequently, these tools frequently produce code that cannot recompile—let alone support reinstrumentation for the runtime tracing required for invariant collection [52, 93]—preventing invariant inference in binary-only contexts whatsoever.

2.4.2 Challenge 2: Invariant Injection. Even if likely invariants could be reliably inferred from binary executables, signaling their violations during fuzzing requires injecting corresponding checks back into the program—a non-trivial task in the absence of source code. Decompilation and binary lifting do not provide a viable path, as they often fail to produce recompilable output code [78, 104, 111, 113], preventing porting of the compiler-inserted invariant checks [76] used in existing source-level invariant-guided fuzzing approaches [57]. While static binary rewriters [48, 50, 89, 110] enable direct modification of executables, their effectiveness depends heavily on debug and relocation metadata, which are often stripped away by developers of closed-source software [102]. Moreover, unlike fuzzing's code coverage instrumentation, which is largely uniform in applying the same instrumentation across all program basic blocks, invariant checks are inherently *specific* and need tailoring per each program location [107], making scalable injection across diverse binaries particularly challenging. As a result, neither compiler-inserted instrumentation nor static binary rewriting provides a reliable solution for injecting invariant checks for binary-only fuzzing.

2.4.3 Challenge 3: Upholding Speed. Speed is critical in fuzzing [112], as faster program execution enables greater test case throughput for more rapid vulnerability discovery. However, program instrumentation inherently introduces runtime overhead relative to the original program [40], with inserted invariant checks [57] further amplifying this cost. Yet, unlike existing approaches' [57] compiler-based instrumentation [76], which seamlessly integrates invariant checks within the original program's code via optimizations like inlining, binary instrumentation is far more sensitive to overhead: code must be carefully inserted so as not to disrupt existing program functionality [102], often requiring the relocation of existing code blocks [89], additional control-flow redirection [48], and scaffolding for saving and restoring in-use registers [37]. As a result, invariant instrumentation must be applied judiciously, as even limited redundancy—such as effectively *inviolable* checks—incur substantial runtime overhead, degrading fuzzing throughput and delaying bug discovery.

Impetus: Extending Likely Data Invariants to Binary-only Fuzzing: Ensuring the robustness of closed-source software remains more critical than ever. While coverage-guided fuzzing has proven effective on binaries, the use of *likely data invariants* is enabling exploration of richer, data-level program states—uncovering defects outside the reach of coverage-only fuzzing. However, existing invariant-guided fuzzing remains reliant on source code, **necessitating new techniques for inferring and leveraging invariants directly in binary-only settings**.

3 Register-level Likely Invariants

To tackle the challenges of applying invariant-guided fuzzing to binary-only programs, we introduce *register-level likely invariants* (Figure 2): a methodology for inferring likely data invariants directly from binaries—at the level of CPU registers—and integrating them into the fuzzing loop, analogous to how source-level likely invariants guide open-source software fuzzing [57]. In the following, we detail how register-level likely invariants are generated, along with our optimizations for improving efficiency by constraining invariant generation and eliminating fuzzing-irrelevant checks.

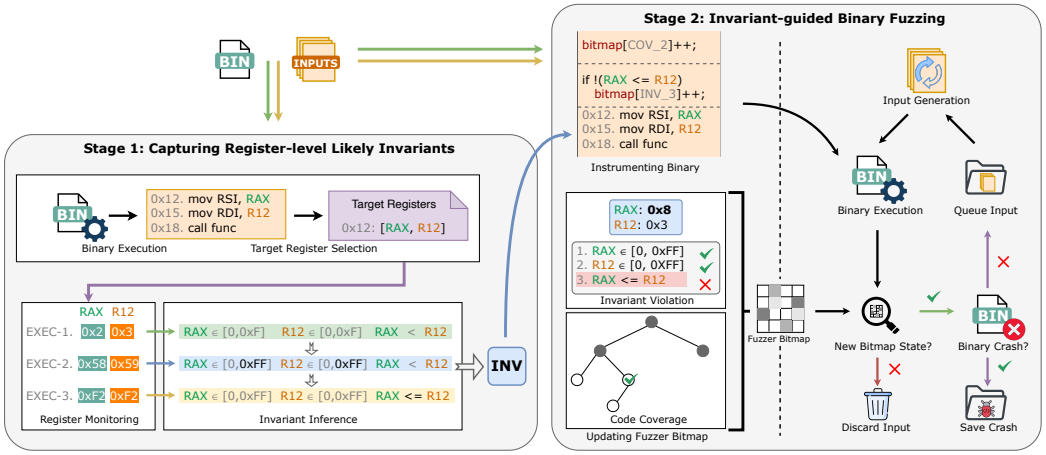


Fig. 2. Workflow of register-level likely invariants for binary-only fuzzing, consisting of two stages. In the first stage, relevant registers are selected, their values are monitored, and likely invariants are inferred over a set of concrete inputs. In the second, the target binary is instrumented with invariant checks; violations observed during fuzzing update the fuzzer's bitmap, and inputs triggering such updates are retained.

3.1 Inferring Likely Invariants from Binaries

While existing invariant-guided fuzzing [57, 70] relies on source-level invariant mining tools [52, 93], as discussed in § 2.4.1, attempting to recover source-like representations from binaries often introduces inaccuracies or even incorrectness. However, we observe that *CPU registers* provide a direct view of how input data propagates during program execution, capturing internal state that would otherwise be expressed through source-level variables. Accordingly, monitoring register values across executions enables the inference of likely invariants that characterize expected data-level behavior—without requiring source-level abstractions. To realize this, we focus on two classes of likely invariants: (1) relationship invariants between pairs of registers, as well as (2) value-constraint invariants over single registers. We describe our approaches for capturing each below.

Table 1. Semantics of updating register-level relationship invariants after observing new runtime register states. Each cell indicates whether the invariant is violated, unchanged, or weakened to a less-constrained form.

Original Invariant	**Newly Observed Condition**		
	R1 = R2	**R1 < R2**	**R1 > R2**
R1 = R2	✔ *unchanged*	⇒ R1 ≤ R2	⇒ R1 ≥ R2
R1 < R2	⇒ R1 ≤ R2	✔ *unchanged*	✗ **violated**
R1 ≤ R2	✔ *unchanged*	✔ *unchanged*	✗ **violated**
R1 > R2	⇒ R1 ≥ R2	⇒ R1 ≠ R2	✔ *unchanged*
R1 ≥ R2	✔ *unchanged*	✗ **violated**	✔ *unchanged*
R1 ≠ R2	✗ **violated**	✔ *unchanged*	✔ *unchanged*

3.1.1 Capturing Relationship Invariants. Given a set of concrete inputs, we characterize pairwise relations between CPU registers by considering basic relational comparisons such as *less than* (<), *greater than* (>), and *equal to* (==). As program executions over these inputs are observed, these invariants are updated when discrepancies arise, as shown in Table 1. For example, if an invariant initially asserts `R1 > R2` but a subsequent execution observes `R1 == R2`, it is relaxed to `R1 >= R2`; if instead `R1 < R2` is observed, it is weakened to `R1 != R2`. In cases where newly captured relationships contradict a prior invariant entirely—such as observing `R1 < R2` under an existing `R1 >= R2` constraint—the invariant is subsequently invalidated and discarded.

3.1.2 Capturing Value-Constraint Invariants. For each register, we additionally collect value-constraint invariants represented by approximate lower and upper bounds. Yet, unlike source-level settings [57, 70], which track a small set of semantically meaningful variables, binary execution exposes a large number of transient, untyped register values at every instruction, including short-lived temporaries, pointer-derived offsets, and mixed-width artifacts (e.g., zero- or sign-extended values). Consequently, we observe that tracking each register's *exact* minima and maxima often yields unstable invariants that overfit to individual executions.

To obtain more robust constraints, we instead generalize observed values to coarse, power-of-two-aligned boundaries (e.g., `0`, `0xFF`, `0xFFFF`), capturing value magnitude while abstracting away incidental variation. These boundaries also implicitly encode both unsigned and signed interpretations: values near zero correspond to small positive integers, whereas values near $2^{64}-1$ correspond to small-magnitude negative values under two's complement representation. During invariant learning, if a newly observed value falls outside the current range, the invariant is widened to the next boundary that encompasses all observed values; for example, an initial invariant for `R1`, `0x10 <= R1 <= 0xFF`, expands to `0x10 <= R1 <= 0xFFFF` upon observing `R1=0x123`.

While our power-of-two-aligned boundaries represent one choice within a broader design space, our implementation remains flexible in how value ranges are abstracted. In particular, our coarse-grained discretization is conceptually similar to the *hit count bucketing* used by modern fuzzers [90], where exact control-flow edge execution counts are discretized into eight fixed ranges—avoiding overfitting while still capturing meaningful shifts in execution behavior (e.g., exercising a loop for 1 iteration versus 100 consecutive iterations). Likewise, our value-constraint invariants capture semantically meaningful magnitude changes (e.g., crossing size or overflow thresholds) without being tied to exact values, yielding invariants whose violations correspond to substantive—and often bug-preceding—deviations in program behavior.

3.1.3 Eliminating Noise. As noted in § 2.4.3, high fuzzing throughput is key to timely bug discovery. Accordingly, to maximize speed, we restrict invariants to a targeted set of registers per basic block—specifically, those whose values are *read* within that block, either explicitly (e.g., via instruction operands) or implicitly (e.g., `RAX` in `mul` and `div` instructions). Our intuition is that only registers whose values are consumed by subsequent instructions directly influence data-dependent program behavior—and expose the unexpected program states that precede bugs (e.g., Figure 1)—whereas others are more likely to reflect transient or otherwise irrelevant state. Altogether, this selective approach reduces the risk of inferring inviolable invariants and lowers invariant-checking overhead during fuzzing, thereby improving throughput and accelerating binary-level bug discovery.

3.2 Injecting Likely Invariants into Binaries

As § 2.4.2 details, instrumenting binaries to check invariants at runtime via decompilation or static binary rewriting is often unreliable or infeasible in practice. To overcome this, we instead encode register-level likely invariant checks directly into the program during execution via *dynamic binary translation*: once inferred, invariants are translated into corresponding violation checks, and inlined

into the currently executed basic block as fuzzer-reachable conditional instructions. In total, our approach preserves the original program's logic, whilst enabling real-time detection of unexpected data-level states—exposing invariant violations as feedback to guide subsequent fuzzing input generation and program state exploration. While we currently leverage the QEMU dynamic binary translation engine, our underlying approach readily generalizes to other similar tools (§ 6.5).

4 Implementation: BINVARIANTS

We realize our approach as a prototype, BINVARIANTS: a binary-only fuzzing framework that leverages register-level likely invariants to guide input mutation and program exploration via invariant-violation feedback (Figure 2). Built on QEMU [34] and the AFL++ [60] fuzzer, BINVARIANTS follows a standard fuzzing workflow with the following key modifications, which we detail below.

4.1 Target Register Selection

QEMU translates the target binary by decoding machine instructions into its Tiny Code Generator intermediate representation (TCG IRs) [35] and subsequently emitting host-architecture machine code. During decoding, instruction boundaries (e.g., start addresses and lengths) are available, which we leverage to select target registers for invariant inference. Specifically, we invoke QEMU's integrated Capstone disassembler APIs [3, 18] for each instruction. Given an instruction's start address and length, its disassembled form is retrieved and parsed to identify source operands. From these operands, registers whose values are read in the instruction are selected, as defined in § 3.1.3. As QEMU also maintains basic block boundaries, we further track registers on a per-block basis.

4.2 Register Monitoring & Invariant Inference

When a program is executed under QEMU, its registers are continuously updated throughout execution via QEMU's built-in `CPUArchState` structure [46], which we directly hook-into to access register values at runtime. Following our strategies discussed in § 3.1.3 and § 4.1, we inspect each instruction during translation to identify target registers read within each basic block; at execution, we collect their values, applying our invariant capture rules (§ 3.1) to infer their relevant likely invariants, which are subsequently injected into the target program for use during fuzzing (§ 4.3).

```
1  ld_i64 tmp0, CPUArchState, RAX                          // load RAX into tmp0
2  ld_i64 tmp1, CPUArchState, RBX                          // load RBX into tmp1
3  brcond_i64 tmp0, tmp1, gtu, $NO_VIOLATION               // skip if RAX > RBX
4  uint32_t inv_id = ((RAX & 0x0F) << 4) | (RBX & 0x0F);   // hash register values
5  inv_id ^= TB_addr & MAP_SIZE;                           // mix-in block address
6  fuzzer_bitmap[inv_id]++;                                // mark violation in bitmap
7  $NO_VIOLATION: ORIGINAL_TB                              // continue executing block
```

Fig. 3. Abridged inlined QEMU [34] TCG IR used by BINVARIANTS for checking invariant `RAX > RBX`. If the invariant holds after comparing the current `RAX` and `RBX`, control jumps directly to the original translated block. Otherwise, the violation is surfaced to the fuzzer as an observable bitmap write, and execution resumes.

4.3 Invariant Check Storing

At runtime, QEMU stores all translated program basic blocks in its Translation Block (TB) cache [36, 38], enabling the avoidance of repeated instruction translation to the host machine's architecture. QEMU additionally employs TB chaining, allowing one block to transfer control directly to the next without returning to QEMU's more costly execution dispatching. Leveraging these optimizations, we emit invariant checks as TCG IR [35], inlined into basic blocks during QEMU's on-the-fly translation. These checks are stored in QEMU's TB cache alongside the block's original translated code, eliminating need for re-translating invariant checks across executions of the same basic block.

Furthermore, the inserted TCG IRs (shown in Figure 3) execute directly within each block without function calls, avoiding costly control-flow redirection overhead and preserving chaining efficiency.

4.4 Violation Feedback

To maintain compatibility with AFL++'s existing runtime feedback mechanisms, we interpose on AFL++'s shared-memory code coverage bitmap to record invariant violations and their frequencies. Each invariant is assigned a unique index in the bitmap, computed at runtime from its associated registers and basic block address, with the block's inlined TCG IR incrementing its corresponding bitmap-slot byte upon violation (Figure 3). After each test case execution, the fuzzer compares the updated bitmap against its prior state, retaining inputs that trigger either (1) new code coverage, (2) a previously unseen invariant violation, or (3) an increased hit-count bucket for an already observed invariant—saving them to the input *queue* for subsequent future mutation.

Altogether, BINVARIANTS augments AFL++'s feedback loop to recognize violations of register-level likely invariants, enabling it to uncover—and further explore—**binary-level program states beyond those reachable through code coverage guidance alone**.

5 Evaluation

Our evaluation of the effects of register-level likely invariants on binary-only fuzzing is guided by the following three research questions:

Q1: Do register-level likely invariants enhance fuzzing's ability to explore diverse program states?
Q2: Can register-level likely invariants surface vulnerabilities beyond coverage-only fuzzing?
Q3: Are register-level likely invariants comparable to source-level likely data invariants?

Table 2. The 25 benchmark binaries used throughout our evaluation of BINVARIANTS, alongside their library/program versions, total invariants learned by BINVARIANTS, and whether they are open or closed source. ✱ = open-source libraries for which we constructed and used a dedicated fuzzing harness as the target binary.

Program	Tested Version	Learned Invariants	Source Access
cert-basic [13]	1.0.3	9,749	Open
exiv2 [6]	0.26	70,822	Open
gpmfdemo [7]	04c4b97	4,673	Open
hdf5 [9] *	0394b03	61,821	Open
imginfo [11]	1.701.0	21,590	Open
jasper [11]	1.701.0	11,051	Open
lame [12]	3.99.5	53,355	Open
mp4split [2]	1.5.1.0	14,968	Open
opj_decompress [15]	2.1.1	36,170	Open
pdfimages [24]	4.00	31,210	Open
pdftops [24]	4.00	60,800	Open
pdftotext [24]	4.05	37,947	Open
readelf [8]	2.30	20,726	Open
sfconvert [1]	0.2.7	7,565	Open
stormlib [19] *	6052223	14,649	Open
tcpdump [21]	4.5.1	28,993	Open
tiff2bw [22]	3.9.7	12,434	Open
xls2csv [4]	0.95	2,564	Open
cpdf [16]	2.8.1	8,006	Closed
cuobjdump [5]	12.9.0	10,699	Closed
idapro [10]	7.1	7,825	Closed
lzturbo [14]	1.2	10,246	Closed
nconvert [23]	7.221	18,047	Closed
nvdisasm [5]	12.9.0	39,787	Closed
pngout [17]	2020-1-15	7,680	Closed

5.1 Experiment Setup

To evaluate register-level likely invariants, we deploy our prototype BINVARIANTS (§ 4) in fuzzing campaigns across 25 real-world programs (Table 2), including 18 open-source programs compiled as statically linked executables as well as 7 proprietary closed-source executables. Following the procedure of Fioraldi et al.'s source-level invariant-guided fuzzing [57], we first run a 24-hour

fuzzing campaign for each benchmark to generate a primary input corpus, which is then used for both invariant inference and as the initial seeds for all fuzzing experiments.

In line with the fuzzing evaluation standards established by Klees et al. [74], we perform five independent 48-hour fuzzing campaigns per benchmark, and assess statistical significance via the Mann-Whitney U test at the $p = 0.05$ significance level. We compute per-benchmark means as the arithmetic mean across all five trials, and report summary statistics across all benchmarks as geometric means. All fuzzing campaigns are performed using AFL++ [60] v4.21c via its QEMU [47] binary fuzzing mode (commit `a6f0632`), with experiments and post-processing all conducted on x86-64 Ubuntu 22.04 machines equipped with Intel i9-12900K CPUs and 64GB RAM.

5.2 Q1: Binary-Level State Exploration

To evaluate how register-level likely invariants influence fuzzing's state-level exploration of binary-only targets, we measure two key aspects: BINVARIANTS' ability to violate its discovered likely invariants, as well as its resulting binary-level code coverage and performance trade-offs.

Table 3. BINVARIANTS' total and uniquely-violated register-level invariants, as well as total and uniquely-exercised binary basic block coverage relative to coverage-only binary fuzzing's. Statistically significant differences (MWU test $p < 0.05$) are shown **bolded**. "n/a" denotes cases where coverage-only fuzzing attains zero unique code coverage, with coverage comparisons and statistical significance testing therefore omitted.

Program	Relative Violated Register-level Invs.				Relative Basic Block Coverage			
	Total	MWU_p	Uniq.	MWU_p	Total	MWU_p	Uniq.	MWU_p
cert-basic	56.82×	**0.008**	438.67×	**0.011**	1.00×	1.000	1.10×	0.834
cpdf	2.25×	**0.008**	7.74×	**0.008**	0.99×	0.841	1.81×	0.548
cuobjdump	8.11×	**0.008**	288.88×	**0.012**	1.11×	**0.008**	10.82×	**0.008**
exiv2	5.80×	**0.008**	13.29×	**0.008**	1.05×	**0.008**	1.81×	**0.008**
gpmfdemo	7.42×	**0.008**	181.50×	**0.012**	1.00×	**0.010**	n/a	n/a
hdf5	2.58×	**0.008**	10.26×	**0.008**	1.00×	0.421	3.18×	0.548
idapro	2.79×	**0.008**	16.70×	**0.008**	1.00×	1.000	n/a	n/a
imginfo	6.65×	**0.008**	47.70×	**0.008**	0.98×	**0.012**	0.15×	**0.011**
jasper	3.38×	**0.012**	25.94×	**0.008**	0.98×	0.075	0.30×	0.073
lame	2.78×	**0.008**	8.94×	**0.008**	1.00×	**0.036**	0.21×	**0.036**
lzturbo	3.99×	**0.012**	117.59×	**0.012**	1.01×	**0.083**	14.36×	**0.044**
mp4split	3.46×	**0.008**	12.20×	**0.008**	1.00×	0.917	1.97×	1.000
nconvert	3.08×	**0.008**	68.86×	**0.012**	1.05×	**0.032**	2.87×	**0.032**
nvdisasm	2.56×	**0.008**	12.33×	**0.008**	1.02×	0.095	4.91×	0.056
opj_decompress	4.58×	**0.008**	44.10×	**0.008**	1.01×	0.056	6.05×	0.056
pdfimages	2.48×	**0.008**	14.65×	**0.012**	0.99×	0.548	1.65×	0.421
pdftops	1.36×	**0.016**	1.93×	**0.032**	0.98×	0.095	0.50×	**0.008**
pdftotext	1.57×	**0.008**	6.95×	**0.008**	0.99×	0.548	0.91×	0.222
pngout	3.22×	**0.008**	59.99×	**0.008**	0.99×	0.530	2.88×	0.530
readelf	9.98×	**0.008**	70.42×	**0.008**	1.04×	0.222	3.10×	0.222
sfconvert	1.13×	**0.032**	4.60×	**0.036**	0.99×	0.249	0.41×	0.346
stormlib	2.55×	**0.008**	15.80×	**0.008**	0.98×	0.548	3.18×	0.421
tcpdump	8.33×	**0.008**	113.56×	**0.008**	1.05×	0.056	3.52×	**0.008**
tiff2bw	2.62×	**0.008**	8.05×	**0.008**	0.96×	**0.047**	0.35×	0.056
xls2csv	19.16×	**0.008**	125.55×	**0.012**	1.00×	0.118	0.50×	**0.025**
Mean:	**4.10×**		**28.11×**		**1.01×**		**1.52×**	

5.2.1 Violated Likely Invariants. To assess whether register-level likely invariants increase the incidence of invariant violations compared to coverage-only binary fuzzing, we measure the number of distinct invariants violated per benchmark by BINVARIANTS and by standard coverage-only fuzzing via AFL++ [60] in its default QEMU mode. We collect the test cases generated by each fuzzer from their 48-hour campaigns, and replay them under the same invariant checks used by BINVARIANTS, recording all violated invariants per trial. We compute BINVARIANTS' mean total and unique invariant violations (i.e., violations observed only under BINVARIANTS and not by coverage-only fuzzing) relative to coverage-only fuzzing, with Table 3 reporting our results.

As shown in Table 3, BINVARIANTS surfaces substantially more invariant violations, achieving **3.10**× more total and **27.11**× more unique violations than coverage-only binary fuzzing. Notably, BINVARIANTS finds more total and unique invariant violations across *all* 25 benchmarks—and in six instances, over 100× more unique invariant violations—with all improvements yielding statistically significant differences. Altogether, these results reveal that register-level likely invariants successfully drive binary-only fuzzing toward a broader range of data-level program states, uncovering substantially more behaviors beyond those explored by coverage-only fuzzing.

Table 4. BINVARIANTS' total throughput (i.e., test cases executed), and total and favored queued inputs relative to coverage-only binary fuzzing's. Statistically significant differences (MWU test $p < 0.05$) are shown **bolded**.

Program	Relative Throughput		Relative Queued Inputs			
	Value	MWU_p	Total	MWU_p	Favored	MWU_p
cuobjdump	0.75×	**0.008**	1.57×	**0.008**	1.68×	**0.012**
nvdisasm	0.56×	**0.008**	1.09×	0.151	1.72×	**0.012**
nconvert	0.93×	0.548	1.46×	**0.008**	1.57×	**0.008**
pngout	1.06×	0.690	1.48×	**0.008**	2.63×	**0.008**
idapro	1.10×	0.222	1.12×	**0.012**	2.77×	**0.011**
lzturbo	0.22×	**0.008**	1.25×	**0.008**	6.16×	**0.012**
cpdf	1.12×	0.310	1.01×	0.548	1.05×	0.142
jasper	0.28×	**0.008**	1.32×	**0.008**	1.34×	**0.012**
tcpdump	0.95×	**0.032**	1.37×	**0.008**	1.42×	**0.012**
cert-basic	0.45×	**0.008**	1.50×	**0.008**	1.43×	**0.008**
sfconvert	1.08×	0.548	0.97×	1.000	1.40×	**0.012**
hdf5	0.72×	**0.008**	1.42×	**0.012**	1.73×	**0.012**
gpmfdemo	0.79×	**0.008**	1.49×	**0.008**	1.65×	**0.008**
pdftotext	0.63×	**0.008**	1.06×	**0.008**	1.19×	**0.008**
opj_decompress	0.47×	**0.008**	1.67×	**0.008**	2.09×	**0.012**
xls2csv	0.96×	1.000	1.50×	**0.008**	1.88×	**0.012**
stormlib	0.60×	**0.016**	1.29×	**0.008**	2.01×	**0.008**
readelf	0.90×	**0.008**	1.04×	**0.008**	1.04×	**0.032**
exiv2	0.43×	**0.008**	1.18×	**0.008**	1.31×	**0.012**
imginfo	0.35×	**0.008**	1.33×	**0.008**	1.65×	**0.008**
pdftops	0.49×	**0.008**	1.02×	0.222	1.35×	**0.008**
pdfimages	0.79×	**0.008**	1.21×	**0.008**	1.39×	**0.008**
lame	0.76×	0.095	1.08×	**0.008**	2.13×	**0.012**
tiff2bw	0.32×	**0.008**	1.49×	**0.008**	1.76×	**0.008**
mp4split	0.61×	**0.008**	1.26×	**0.008**	1.52×	**0.008**
Mean:	**0.63×**		**1.27×**		**1.69×**	

5.2.2 Binary-Level Code Coverage. As prior work [74, 75] shows that higher code coverage is strongly correlated with increased bug discovery, we examine how the additional data-level states

reached by register-level likely invariants (§ 5.2.1) translate to binary-level code coverage. Accordingly, we measure the total and unique code coverage of BINVARIANTS versus coverage-only fuzzing, leveraging the AFL-QEMU-Cov [56] coverage collection tool to obtain basic-block-level code coverage per trial over all 25 binary benchmarks. Table 3 reports BINVARIANTS' mean total and unique coverage per benchmark relative to coverage-only binary fuzzing.

As Table 3 shows, BINVARIANTS achieves binary code coverage comparable to coverage-only fuzzing, with a mean **1%** relative improvement in total basic blocks covered. This aligns with prior findings from source-level invariant-guided fuzzing, where InvsCov [57] similarly reports total coverage on-par with coverage-only fuzzing. However, we observe that BINVARIANTS attains **52%** higher mean *unique* coverage, confirming that violating register-level likely invariants successfully exposes alternative code paths unexplored by coverage-only binary fuzzing.

5.2.3 Performance Trade-Offs. To understand the performance trade-offs of register-level likely invariants, we measure BINVARIANTS' execution throughput (i.e., total test cases processed during fuzzing) relative to coverage-only fuzzing, along with the total number of fuzzer-queued inputs (i.e., inputs retained as interesting) and the subset of fuzzer-favored inputs (i.e., those actively explored), as shown in Table 4. Overall, despite a mean **37%** reduction in throughput, BINVARIANTS generates a mean **27%** more queued inputs and **69%** more favored inputs. Altogether, these results underscore that register-level likely invariants enable more diverse exploration across program states and code regions, despite facing a reduced overall throughput versus coverage-only fuzzing.

Q1: Register-level likely invariants uncover data-level program states distinct from code coverage alone, expanding binary fuzzing's reach into code regions ordinarily untested by coverage-only exploration.

5.3 Q2: Binary-Level Bug Discovery

To evaluate the impacts of register-level likely invariants on binary fuzzing bug discovery, we measure BINVARIANTS' total and uniquely found bugs, BINVARIANTS' relative speedup in triggering bugs also found by coverage-only fuzzing, and the role of invariant violations in surfacing bugs.

5.3.1 Overall Bug Discovery. To assess the impact of register-level likely invariants on bug discovery, we compare the sets of deduplicated bugs found by both BINVARIANTS and coverage-only fuzzing. We cluster all fuzzer-generated crash-triggering test cases via CASR [101], which analyzes stack traces obtained from AddressSanitizer (ASAN) [65]. For closed-source benchmarks, we employ QASan [59], a QEMU-based ASAN implementation that enables memory error detection on binary-only programs; while for open-source benchmarks we use standard compiler-instrumented ASAN [76]. We then manually deduplicate CASR's resulting crash clusters by cross-referencing source and decompiled code, yielding the final set of unique bugs reported in Table 5.

Table 5 reports the total counts of deduplicated bugs across five trials, along with the number of bugs uniquely found by each approach. Overall, BINVARIANTS uncovers **143** bugs in total, compared to coverage-only fuzzing's 137, including **20** bugs that coverage-only fuzzing fails to find. As our benchmarks include many older versions of programs (Table 2), we replay all triggering test cases on all latest available versions and apply the same deduplication procedure (§ 5.3). In doing so, we find that BINVARIANTS discovers **68 new bugs**, compared to 62 with coverage-only fuzzing, with all reported to their respective developers and **31** since confirmed and/or fixed. Altogether, these findings are consistent with the outcomes of our invariant violation (§ 5.2.1) and code coverage (§ 5.2.2) evaluations, showing that BINVARIANTS' exploration of program states beyond those reached by code coverage alone helps uncover additional distinct binary program bugs.

Table 5. BINVARIANTS' total, new, and uniquely-found program bugs versus coverage-only binary fuzzing's, alongside the subset of mutually-found bugs BINVARIANTS reaches with a lower mean time-to-discovery (TTD).

Program	CODECOV-found Bugs			BINVARIANTS-found Bugs			
	Total	New	Uniq.	Total	New	Uniq.	↓TTD
cert-basic	5	1	0	5	1	0	0
cpdf	1	1	0	1	1	0	0
cuobjdump	2	2	0	2	2	0	2
exiv2	10	0	0	10	0	0	7
gpmfdemo	2	2	0	3	3	1	2
hdf5	4	3	1	4	3	1	0
idapro	1	0	0	1	0	0	1
imginfo	11	0	0	11	0	0	4
jasper	9	0	0	10	0	1	5
lame	2	0	0	2	0	0	1
lzturbo	0	0	0	0	0	0	0
mp4split	3	2	1	2	2	0	1
nconvert	15	15	4	15	15	4	6
nvdisasm	1	1	0	1	1	0	0
opj_decompress	5	0	2	3	0	0	0
pdfimages	9	2	2	10	2	3	2
pdftops	8	2	1	7	3	0	5
pdftotext	1	1	0	1	1	0	0
pngout	16	16	2	19	19	5	4
readelf	0	0	0	0	0	0	0
sfconvert	9	5	1	8	5	0	6
stormlib	2	2	0	2	2	0	0
tcpdump	9	0	0	13	0	4	2
tiff2bw	5	0	0	5	0	0	3
xls2csv	7	7	0	8	8	1	6
Total:	**137**	**62**	**14**	**143**	**68**	**20**	**57**

5.3.2 Bug Discovery Time. To examine how register-level likely invariants guide fuzzer exploration over binary programs, we compute the mean time-to-discovery (TTD) of bugs found by both BINVARIANTS and coverage-only fuzzing, averaged across all trials in which each bug is observed, and count the number of cases where BINVARIANTS triggers the same crash more quickly. As shown by Table 5, BINVARIANTS exposes a total of **57** bugs faster than coverage-only fuzzing across multiple benchmarks (e.g., `NConvert`, `Jasper`, `PNGOUT`, and `TCPDump`)—suggesting that register-level likely invariants not only surfaces bugs unreached by coverage-only fuzzing, but in many cases also better steers fuzzer exploration toward failure-preceding program states that enable overlapping bugs to be discovered more quickly.

5.3.3 Influence of Invariant Violations. To further understand the relationship of register-level likely invariants and bug discovery, we analyze the correspondence between the invariant violations and bugs uniquely found by BINVARIANTS. For each bug-triggering input and its direct predecessor input (i.e., the test case from which it was mutated), we evaluate whether it (1) triggers any invariant violation, (2) violates invariants not violated by coverage-only fuzzing, and (3) covers basic blocks not exercised by coverage-only fuzzing—attributing a bug to register-level likely invariants if at least one of these conditions is satisfied. Table 6 summarizes all per-bug findings.

Table 6. Correspondence between register-level likely invariants and BINVARIANTS' uniquely-found bugs (Table 5). For both bug-triggering inputs and their direct predecessor inputs, "*w/ Viols.*" denotes the total that violate likely invariants, "*Uniq. Inv.*" denotes those that uniquely violate invariants unviolated by coverage-only fuzzing, and "*Uniq. Cov.*" denotes those that uniquely cover basic blocks unexercised by coverage-only fuzzing.

Program	Uniq. Bugs	Bug Triggering Inputs			Bug Predecessor Inputs		
		w/ Viols.	Uniq. Inv.	Uniq. Cov.	w/ Viols.	Uniq. Inv.	Uniq. Cov.
gpmfdemo	1	0	0	0	0	0	0
hdf5	1	1	1	1	1	1	1
jasper	1	1	1	0	1	1	0
nconvert	4	4	1	2	4	2	3
pdfimages	3	3	3	3	3	3	3
pngout	5	5	5	0	5	1	0
tcpdump	4	4	4	2	4	4	2
xls2csv	1	0	0	0	0	0	0
Total:	20	18/20	15/20	8/20	18/20	12/20	9/20

As shown in Table 6, **18** out of 20 bugs uniquely discovered by BINVARIANTS are associated with invariant-violating program states, for both bug-triggering inputs and their predecessors. Among these, **15** involve violations of invariants not observed under coverage-only fuzzing, while **8** additionally exercise basic blocks not reached by coverage-only fuzzing. Notably, invariant violations are already present in **18** predecessor inputs, with **12** exhibiting invariants uniquely violated by BINVARIANTS, indicating that invariant-guided exploration successfully steers fuzzing toward bug-preceding states before the final crashing input is produced (e.g., Figure 1).

In the following, we detail several bugs uniquely found by BINVARIANTS, and how violations of register-level likely invariants facilitate their discovery during fuzzing.

- **TCPDump: unbounded loop counter.** In this bug discovered by BINVARIANTS in TCPDump [21] (Figure 4), register `R13` influences the computation of `nshorts`, which acts as a decrementing loop bound for subsequent memory accesses. Accordingly, BINVARIANTS infers a value-range invariant on `R13` of `0x10 <= R13 <= 0xFFFF`. During fuzzing, inputs that violate this invariant—despite not altering control flow and thus remaining indistinguishable to coverage-only fuzzing—are ultimately retained and further mutated by BINVARIANTS. Consequently, BINVARIANTS' exploration is driven into atypical data states, eventually producing an input where `R13` takes an even substantially larger value (`0x2008FB`). As a result, `nshorts` becomes excessively large, causing the loop to iterate far beyond its intended bounds and leading to an invalid memory read.

```
1 nshorts = R13 / VAL;
2 while (--nshorts >= 0) *ptr++;
```

Invariant: `0x10 ≤ R13 ≤ 0xFFFF`
Crash value: `R13 = 0x2008FB`

Fig. 4. Unbounded loop in TCPDump [21]. Exceeding the expected range `[0x10,0xFFFF]` of the loop counter `R13` leads the fuzzer to inputs with an oversized loop bound, causing an out-of-bounds read.

```
1 dest = base + (int32)RSI;
2 var = MEM[dest];
```

Invariant: `0 ≤ RSI ≤ 0xFF`
Crash value: `RSI = 0xF292BC7B`

Fig. 5. Signed underflow in PNGOUT [17]. When offset `RSI` moves beyond normal range of `[0,0xFF]`, the fuzzer finds inputs where `(int32)RSI` is negative, positioning `dest` in memory far before base.

- **PNGOUT: signed offset underflow.** As Figure 5 shows, in this bug found in closed-source binary PNGOUT [17], the lower 32 bits of register `RSI` are used to compute the destination address of a memory write (`dest = base + (int32)RSI`). BINVARIANTS infers value-range invariant `0 <= RSI <= 0xFF`, reflecting `RSI`'s typical bounds seen during execution. However, fuzzer inputs

violating this invariant end up further mutated by BINVARIANTS, eventually producing a test case where RSI takes a much larger value (0xF292BC7B). Due to the signed cast, this results in a large negative offset, causing dest to point 215 MB *before* base, leading to a heap-overflow crash.

- **PNGOUT: unchecked memory write.** Figure 6 highlights another vulnerability in PNGOUT where RSI determines the size argument to memset via size = (int32)RSI * VALUE. BINVARIANTS infers a value-range invariant 0 <= RSI <= 0xFFFF, which it later violates during fuzzing, eventually converging on inputs where RSI balloons to 0x40000032. Yet, this resulting size far exceeds the destination buffer's capacity, causing an unchecked write and subsequent crash.

```
1  size = (int32)RSI * VAL;
2  memset(buf, 0, size);
```

Invariant: $0 \leq RSI \leq 0xFFFF$
Crash value: RSI = 0x40000032

Fig. 6. Unchecked memory write in PNGOUT [17]. As RSI moves beyond its typical range [0,0xFFFF], the fuzzer converges on inputs where size exceeds buf's bounds, resulting in an out-of-bounds write.

```
1  while (++iter <= RSI) MEM[0x914970 +iter*4] = 0;
2  *MEM[0x916528] = val;
```

Invariant: $0 \leq RSI \leq 0xFF$
Crash value: RSI = 0xFF05

Fig. 7. Pointer overwrite in NConvert [23]. After exceeding RSI's typical upper limit of 0x100, the fuzzer discovers inputs that extend the loop, zeroing a pointer and causing a crash upon dereference.

- **NConvert: pointer overwrite.** Figure 7 presents a bug in the closed-source program NConvert [23], where RSI controls the loop bound for zeroing memory starting at 0x914970. BINVARIANTS identifies a value-range invariant 0 <= RSI <= 0xFF, which is violated during fuzzing, leading to inputs where RSI gradually grows to 0xFF05. This causes the loop to zero-out the large memory region between addresses 0x914970 and 0x954580. Although this overwrite does not immediately crash the program, it corrupts a pointer stored at address 0x916528—which is later dereferenced after being overwritten, triggering a program NULL pointer dereference crash.

Collectively, these results demonstrate that violating register-level likely invariants, and thereby exploring atypical data-level program states, plays a key role in exposing many bugs missed by coverage-only binary fuzzing. For the remaining two bugs in Table 6 that are not associated with invariant violations, we attribute their discovery to the inherent randomness of fuzzing.

Q2: By exploring program states different from those found by coverage-only fuzzing, register-level likely invariants reach numerous bugs in less time, in addition to finding many bugs missed by coverage alone.

5.4 Q3: Comparison with Source-Level Likely Invariants

To compare register-level likely invariants with conventional *source*-level likely invariants [57, 70], we evaluate BINVARIANTS against the state-of-the-art source-level invariant-guided fuzzing approach InvsCov [57]. Using the same initial input corpus (§ 5.1), we configure InvsCov to learn source-level invariants and instrument each program with both invariant checks and coverage tracking, before fuzzing for five 48-hour trials using AFL++. We compare BINVARIANTS to InvsCov on our 10 open-source benchmarks supported by InvsCov; for the remaining 8 programs, InvsCov either fails to infer invariants within 24 hours or produces instrumentation that breaks the programs. To account for BINVARIANTS' higher overhead—largely due to its QEMU-based [34] binary instrumentation—we normalize results by execution count per trial, collecting InvsCov's generated test cases and crashes up to the same number of executions as BINVARIANTS.

Table 7. BINVARIANTS' total and uniquely-exercised program basic block coverage relative to source-level invariant-guided fuzzing approach InvsCov [57], alongside the total and uniquely-found program bugs discovered by both approaches. Statistically significant differences in coverage (MWU test $p < 0.05$) are **bolded**.

Program	Rel. Basic Block Coverage				INVSCOV-found Bugs		BINVARIANTS-found Bugs	
	Total	MWU_p	Uniq.	MWU_p	Total	Uniq.	Total	Uniq.
cert-basic	1.00×	0.091	0.39×	**0.024**	5	0	5	0
gpmfdemo	1.00×	**0.018**	0.34×	**0.023**	2	0	3	1
hdf5	1.00×	**0.016**	9.95×	**0.032**	4	1	4	1
imginfo	1.00×	0.530	3.76×	0.600	12	1	11	0
jasper	1.00×	1.000	1.05×	0.834	8	0	10	2
mp4split	1.01×	0.095	8.08×	0.095	2	0	2	0
pdftotext	1.00×	0.690	1.17×	0.548	2	1	1	0
sfconvert	1.05×	0.056	20.8×	**0.011**	9	1	8	0
stormlib	0.99×	0.690	12.17×	0.675	4	3	2	1
tcpdump	1.06×	**0.008**	2.26×	**0.008**	13	2	13	2
Mean / Total:	**1.01×**		**2.78×**		**61**	**9**	**59**	**7**

5.4.1 Bug Discovery. Following our same procedures for coverage and bug measurement as described in § 5.2.2 and § 5.3.1, we report BINVARIANTS' mean relative code coverage versus InvsCov's across five trials per benchmark (Table 7), along with total and uniquely found deduplicated bugs. Overall, BINVARIANTS achieves coverage comparable to source-level InvsCov, with a mean **1%** increase in total covered basic blocks. However, we observe that BINVARIANTS exercises more unique basic blocks in 8 out of 10 benchmarks, with a mean **1.78**× increase overall. Moreover, while InvsCov finds two more bugs in total, BINVARIANTS uncovers **7** distinct bugs that InvsCov does not trigger. Together, these results suggest that register-level likely invariants expose qualitatively different program states than source-level invariants.

Table 8. Distributions of the *source*-level likely invariants (generated via Daikon [52]) uniquely violated by InvsCov and BINVARIANTS. *Type-1*: constraints on a single variable (e.g., `v0>=0`, `v0∈[0,4]`); *Type-2*: relationships between two variables (e.g., `v0<=v1`); *Type-3*: linear equalities spanning three variables (e.g., `v0-v1-v2==0`).

Program	INVSCOV-violated Source-level Invs.			BINVARIANTS-violated Source-level Invs.		
	Type-1	Type-2	Type-3	Type-1	Type-2	Type-3
cert-basic	74.35%	25.65%	0.00%	86.98%	13.02%	0.00%
gpmfdemo	44.90%	55.10%	0.00%	85.79%	14.21%	0.00%
hdf5	47.89%	50.82%	1.29%	56.65%	43.13%	0.22%
imginfo	53.70%	46.20%	0.10%	74.53%	25.47%	0.00%
jasper	60.52%	39.48%	0.00%	67.57%	32.43%	0.00%
mp4split	31.60%	68.40%	0.00%	58.75%	41.25%	0.00%
pdftotext	17.30%	82.63%	0.08%	32.22%	67.29%	0.49%
sfconvert	22.05%	77.95%	0.00%	69.29%	30.29%	0.43%
stormlib	15.87%	83.87%	0.26%	51.77%	47.88%	0.34%
tcpdump	67.46%	32.54%	0.00%	72.60%	27.40%	0.00%
Mean:	**38.25%**	**52.45%**	**0.22%**	**63.40%**	**30.63%**	**0.36%**

5.4.2 Analysis of Violated Invariants. To further assess the correspondence between register-level likely invariants and source-level invariants, we rerun the test cases generated by BINVARIANTS and InvsCov under the same Daikon-generated [52, 54] source-level invariants used by InvsCov, and

record each fuzzer's uniquely violated invariants. We categorize violations into three types—**Type 1** (value constraints on a single variable), **Type 2** (relationships between two variables), and **Type 3** (linear relations across three variables)—reporting their mean distributions per approach in Table 8.

As shown in Table 8, InvsCov-driven fuzzing predominantly violates **Type-2** invariants, accounting for 52.45% of its violations, whereas BINVARIANTS-driven fuzzing shifts toward **Type-1** violations (63.40%). We observe that this shift is consistent across most benchmarks, with BINVARIANTS producing a higher proportion of Type-1 violations in 8 out of 10 cases, while violations of Type-3 invariants remain negligible for both approaches. Overall, these differences reflect how each approach models program behavior: InvsCov operates over semantically meaningful source-level variables and inferred types, enabling it to better capture relationships among multiple variables, whereas BINVARIANTS' register-level invariants focus on value ranges and low-level data manipulations—biasing exploration toward values outside typical ranges and boundary conditions.

Importantly, we see that BINVARIANTS' 1–67% relative increase in queued test cases (Table 4) remains on-par with InvsCov's self-reported mean queue size increase of 62% [57], suggesting that register-level likely invariants do not oversaturate exploration compared to source-level likely invariants. Altogether, these results demonstrate that register- and source-level invariants capture distinct classes of program properties, driving fuzzers toward different—yet equally valuable—patterns of invariant violations, code coverage, and bug discovery.

Q3: Absent source-level semantics, BINVARIANTS naturally reveals program states distinct from InvsCov's, whilst extending the benefits of invariant-guided fuzzing to the many use cases where source is *unavailable*.

6 Discussion

Below, we weigh several limitations as well as potential future enhancements of register-level likely invariants and our prototype implementation, BINVARIANTS.

6.1 Improving Runtime Overhead

As Table 3 shows, BINVARIANTS achieves a mean 37% lower execution throughput (i.e., 0.63× relative throughput) compared to coverage-only binary fuzzing. We attribute much of this overhead to the already high baseline cost of QEMU-based instrumentation [34], which—even with optimizations such as translated block caching [38] and chaining [36]—remains over an order of magnitude slower than the compile-time instrumentation used in source-available fuzzing [89, 90], and is only further compounded by the runtime checks added by register-level likely invariants (e.g., Figure 3). However, we posit that some performance trade-offs are expected when extending coverage-only fuzzing with data-level feedback signals, and ultimately do not hinder BINVARIANTS' effectiveness in reaching both distinct code regions (Table 3) and bugs (Table 5) missed by coverage-only fuzzing.

Looking ahead, we believe that static binary rewriting (§ 7.2) offers a promising direction for improving runtime overhead. As source-level invariant-guided fuzzing [57] reportedly incurs only a 7% throughput reduction relative to compile-time instrumentation—and current static binary rewriters attain efficiency *on-par* with compile-time instrumentation [50, 89, 110]—we anticipate adopting static binary rewriting will provide a more performant foundation for register-level likely invariants, narrowing the gap with both coverage-only binary fuzzing as well as source-level invariant-guided fuzzing. We leave the requisite engineering and reevaluation to future work.

6.2 Learning Invariants During Fuzzing

Our current implementation of BINVARIANTS mirrors the source-level invariant-guided fuzzing of InvsCov [57], learning register-level likely invariants from a pre-generated corpus of inputs

prior to the main fuzzing campaign. While this approach indeed drives fuzzing to explore unique program states missed by coverage-only fuzzing, both BINVARIANTS' and InvsCov's invariant generation remains limited to the data-level state diversity captured in the initial corpus. Accordingly, generating and updating invariants on-the-fly *during* fuzzing to reflect newly observed behaviors would expand exploration into additional unseen states—and likely uncover many more bugs.

However, in examining the feasibility of extending our current approach to support on-the-fly invariant learning during fuzzing, we observe significant technical roadblocks: in particular, dynamically capturing invariants will incur more frequent (1) invariant violations and retroactive updates, (2) increased evictions from QEMU's translated basic block cache, (3) re-translation of basic blocks to incorporate updated invariants, and (4) reconstruction of translated block chaining. While we expect these obstacles are likely addressable, we anticipate that maximizing performance will require substantial QEMU-side refactoring, and we therefore leave this exploration as future work.

6.3 Capturing Multi-location Invariants

As our current implementation of BINVARIANTS tracks register-level likely invariants within *individual* basic blocks, inserted violation checks consider only the register state local to each block. In practice, violations of invariants spanning *multiple* code locations (e.g., constraints on a register across control-flow transitions) may surface additional interesting program states overlooked by coverage-only fuzzing. Because our current approach sits atop QEMU's block-by-block instrumentation workflow (§ 4.3)—analogous to how AFL++ interposes binary-level coverage instrumentation at each executed basic block [45]—supporting invariants across multiple blocks introduces considerable overhead in learning, inserting, and monitoring these more complex invariants, particularly given the combinatorial explosion from tracking and refining many more register-to-register relationships. Accordingly, we leave exploring these trade-offs to future work.

6.4 Risk of Invariant and Coverage Bitmap Collisions

As Figure 3 shows, our approach records violations of register-level likely invariants in the fuzzer's *bitmap* (e.g., AFL's `trace_bits`), which also stores coverage information (e.g., control-flow edges and hit counts). Although the total number of learned invariants exceeds 61,000 for larger targets such as HDF5 (Table 2), we observe that only a small fraction are violated—averaging around 3%—and thus mapped into the shared invariant/coverage bitmap. This low invariant-specific bitmap utilization indicates that collisions between invariant- and coverage-specific bitmap entries, which potentially lead to false-negative invariant violations, remain unlikely. While enlarging the fuzzer bitmap beyond its default 64KB size remains a straightforward solution, prior work shows that this incurs significant performance loss, particularly when the bitmap no longer fits in the system L2 cache [29, 63]. More broadly, recent advances in fuzzer instrumentation aim to reduce coverage-level bitmap collisions (e.g., CollAFL [63], BigMap [29], and AFL++'s non-colliding coverage [60]), suggesting opportunities to mitigate collisions by partitioning the bitmap into separate coverage and invariant regions. We plan to explore these enhancements in future updates to BINVARIANTS.

6.5 Supporting Other Architectures and Executable Formats

Our invariant-guided fuzzing approach is implemented atop QEMU [34], leveraging its support for dynamic instrumentation of x86-64 Linux binaries. Extending register-level likely invariants to other architectures (e.g., ARM, PowerPC) primarily depends on support for those architectures in the underlying dynamic translator. Yet, as QEMU already supports a wide range of architectures, we posit that the remaining effort to port BINVARIANTS to these architectures requires straightforward changes to our handling of their specific register semantics (e.g., calling conventions).

Supporting additional executable formats, such as macOS Mach-O and Windows PE, presents a different challenge: while QEMU provides broad architectural support, its support for OS-specific executable formats is more limited. In such cases, alternative instrumentation frameworks such as Frida [25], which offer robust support for Mach-O and PE binaries, provide a practical path forward. As these frameworks expose register-level introspection capabilities similar to QEMU, we expect that extending BINVARIANTS to these non-Linux platforms remains feasible with minimal changes.

7 Related Work

Below, we discuss recent related enhancements in fuzzing as well as binary-level instrumentation.

7.1 Runtime Feedback for Guiding Fuzzing

Modern fuzzers [82] leverage runtime introspection to pinpoint and prioritize interesting test cases, gradually steering testing toward novel program states—and bugs—seldom reachable through now-obsolete black-box (i.e., feedback-agnostic) fuzzing [51, 87]. In practice, *code coverage* (e.g., basic blocks, control-flow edges) continues to remain today's most common program feedback metric for fuzzer guidance, facilitated via compile-time [60, 80] as well as binary-level [34, 89] instrumentation. Related approaches extend fuzzers' code coverage mechanisms to better prioritize inputs exercising rarely-reached code regions [77], track finer-grained control-flow information [98], and reduce overhead through optimized instrumentation [63, 88, 90, 115] and data structures [29]. By capturing both code coverage *and* register-level likely invariants, BINVARIANTS provides orthogonal guidance to conventional coverage-centric fuzzing approaches—and stands to be further strengthened by current and future enhancements and optimizations in coverage-based fuzzing techniques.

Recently, a number of approaches are enhancing fuzzing with capabilities for differentiating unique *data*-level program states. TaintScope [109] and Angora [40] utilize taint tracking and constraint solving to model input-to-data dependencies, while RedQueen [33] and Weizz [58] recover concrete mappings between input bytes and conditional comparison operands toward enabling more targeted branch solving. InvsCov [57] expands on these approaches by tracking violations of *likely data invariants*, exposing unusual and often bug-preceding (Table 5) data-level states that are frequently missed by code coverage alone. Building on these advancements, BINVARIANTS extends first-of-its-kind likely invariant feedback to binary-only fuzzing, facilitating data-level exploration in fuzzing contexts ordinarily restricted by coverage-only guidance.

7.2 Advancements in Binary-Only Fuzzing

With fuzzing's continued reliance on runtime information such as code coverage, many efforts focus on adapting these techniques to binary-level contexts through improved program instrumentation capabilities. *Dynamic binary translators* (e.g., QEMU [34], DynamoRIO [39], and Intel PIN [72]) perform instrumentation on-the-fly during fuzzing, enabling fine-grained runtime analysis (e.g., register-level introspection) but incurring significant overhead due to their emulation-based execution. Among these, QEMU [34] remains among the most widely adopted, serving as the de facto binary instrumentation backend for commodity fuzzers such as AFL++ [45]. Other dynamic instrumentation frameworks, including Unicorn [26] and Frida [25], offer similar capabilities, extending binary-only fuzzing to additional software domains such as mobile applications. While BINVARIANTS currently leverages QEMU as its underlying instrumentation framework, the principles of register-level likely invariants are generalizable to most modern dynamic binary translators.

To overcome the performance limitations of dynamic translation, recent approaches instead turn to *static binary rewriting* (e.g., ZAFL [89], E9Patch [50], and Egalito [110]), instrumenting programs prior to fuzzing in a manner analogous to compile-time instrumentation. While existing static rewriters achieve orders-of-magnitude lower overhead than dynamic translation, they face many

practical limitations, including binary-breaking modifications and compatibility challenges across architectures and executable formats [102]. With continued advancement, we anticipate these frameworks will be well-positioned to support register-level likely invariants, enabling a more performant re-implementation of BINVARIANTS toward accelerating binary-only bug discovery.

Several recent works showcase alternative *hybrid* approaches for binary-only fuzzing. QSYM [114] and SymFit [99] combine coverage-guided binary fuzzing with selective concolic execution, enabling targeted branch solving for improved program coverage. Similarly, SymQEMU [97] accelerates this approach through optimized QEMU-level instrumentation. While our current implementation of register-level likely invariants mirrors the design of source-level invariant-guided fuzzing (i.e., InvsCov [57]) in *complementing* conventional fuzzing workflows, it also offers strong potential for synergistic integration with hybrid concolic approaches—further improving bug discovery through more targeted, selective violating of distinct likely invariants.

8 Conclusion

As fuzzing has advanced beyond code coverage guidance, *likely data invariants* have emerged as an effective feedback signal for driving testing into new program states. However, existing techniques depend on the availability of program source code, preventing the adoption of likely-invariant feedback in binary-only fuzzing contexts. To overcome this challenge, we present *register-level likely invariants*: the first approach for extracting likely invariants directly from binaries, enabling capture of low-level, fuzzing-critical program states without relying on source code. When integrated in binary-only fuzzing, our approach effectively drives testing into states unexplored by code coverage alone: exploring **52% more** distinct code coverage as well as uncovering **20 bugs** missed by coverage-only fuzzing. Altogether, our work underscores the value of alternative feedback mechanisms for driving binary-only fuzzing—motivating future adaptation of traditionally source-only fuzzing enhancements to today's ever-critical closed-source, binary-only use cases.

9 Data Availability

We release the source code of BINVARIANTS and all evaluation artifacts at the following URL: https://github.com/FuturesLab/Binvariants.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Award No. 2419798, and by the Defense Advanced Research Projects Agency (DARPA) under Award No. FA8750-24-2-0002, Subaward No. GR105409-SUB00001384. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or DARPA.

References

[1] 2025. Audio File Library. https://audiofile.68k.org/
[2] 2025. Bento4: Full-featured MP4 format, MPEG DASH, HLS, CMAF SDK and Tools. https://github.com/axiomatic-systems/Bento4
[3] 2025. Capstone Disassembly/Disassembler Framework. https://github.com/capstone-engine/capstone
[4] 2025. CATDOC. https://github.com/vbwagner/catdoc
[5] 2025. CUDA Binary Utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/
[6] 2025. Exiv2: Image Metadata Library and Tools. https://github.com/Exiv2/exiv2
[7] 2025. GMPF Parser. https://github.com/gopro/gpmf-parser
[8] 2025. GNU BinUtils. https://www.gnu.org/software/binutils/
[9] 2025. HDF5 Library Repository. https://github.com/HDFGroup/hdf5
[10] 2025. IDA Pro: Powerful Disassembler, Decompiler & Debugger. https://hex-rays.com/ida-pro
[11] 2025. JasPer Image Coding Toolkit. https://github.com/jasper-software/jasper

[12] 2025. LAME MP3 Encoder. https://lame.sourceforge.io/index.php
[13] 2025. LibKSBA: A Library to Access X.509 Certificates and CMS Data. https://github.com/gpg/libksba
[14] 2025. LZTurbo. https://sites.google.com/site/powturbo/
[15] 2025. OpenJPEG. https://github.com/uclouvain/openjpeg
[16] 2025. PDF Command Line Tools. https://www.coherentpdf.com/
[17] 2025. PNGOUT Linux Port. https://www.jonof.id.au/kenutils.html
[18] 2025. QEMUAFL Capstone Disassembler. https://github.com/AFLplusplus/qemuafl/blob/master/disas/capstone.c
[19] 2025. StormLib. https://github.com/ladislav-zezula/StormLib
[20] 2025. TCPDump Commit #4ac7226. https://github.com/the-tcpdump-group/tcpdump/commit/4ac72261ee255b36a0f0117dfef10801be6659dd
[21] 2025. The TCPdump Network Dissector. https://github.com/the-tcpdump-group/tcpdump
[22] 2025. TIFF Decoding Library. https://github.com/libsdl-org/libtiff
[23] 2025. XnView NConvert. https://www.xnview.com/en/nconvert/
[24] 2025. XpdfReader. https://www.xpdfreader.com/
[25] 2026. Frida: Dynamic Instrumentation Toolkit. https://github.com/frida/frida
[26] 2026. Unicorn Engine. https://github.com/unicorn-engine/unicorn
[27] Rawan Abulibdeh, Matthew G Crowson, Molly J Douglas, Mena Ramos, Noelle N Saillant, and Leo Anthony Celi. 2026. A Problem of Epic Proportion. *PLOS Digital Health* 5, 3 (2026).
[28] Adobe Inc. 2024. Adobe Acrobat Reader. https://www.adobe.com/acrobat/pdf-reader.html
[29] Alif Ahmed, Jason D Hiser, Anh Nguyen-Tuong, Jack W Davidson, and Kevin Skadron. 2021. Bigmap: Future-proofing Fuzzers with Efficient Large Maps. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[30] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A Compiler-Level Intermediate Representation Based Binary Analysis and Rewriting System. In *ACM European Conference on Computer Systems (EuroSys)*.
[31] Ross Anderson. 2002. Security in Open versus Closed Systems: The Dance of Boltzmann, Coase and Moore. (2002). https://www.cl.cam.ac.uk/ftp/users/rja14/toulouse.pdf
[32] Apple Corporation. 2025. Apple macOS. https://www.apple.com/macos
[33] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *ISOC Network and Distributed System Security Symposium (NDSS)*.
[34] Fabrice Bellard. 2025. QEMU: A Generic and Open Source Machine Emulator and Virtualizer. https://www.qemu.org/
[35] Fabrice Bellard. 2025. QEMU Documentation: TCG Intermediate Representation. https://www.qemu.org/docs/master/devel/tcg-ops.html
[36] Fabrice Bellard. 2025. QEMU Documentation: Translator Internals. https://www.qemu.org/docs/master/devel/tcg.html
[37] Andrew R Bernat and Barton P Miller. 2011. Anywhere, Any-Time Binary Instrumentation. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE)*.
[38] Andrea Biondo. 2018. Improving AFL's QEMU Mode Performance. https://abiondo.me/2018/09/21/improving-afl-qemu-mode/
[39] Derek Bruening. 2025. DynamoRIO: Dynamic Instrumentation Tool Platform. https://dynamorio.org/
[40] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (Oakland)*.
[41] Cristina Cifuentes. 1994. Reverse Compilation Techniques. https://rgaucher.info/pub/decompilation_thesis.pdf
[42] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.
[43] Cybersecurity and Infrastructure Security Agency. 2021. Kaseya Ransomware Attack: Guidance for Affected MSPs and Their Customers. https://www.cisa.gov/news-events/news/kaseya-ransomware-attack-guidance-affected-msps-and-their-customers
[44] Defense Advanced Research Projects Agency (DARPA). 2020. Assured Micropatching (AMP). https://www.darpa.mil/research/programs/assured-micropatching
[45] AFL++ Developers. 2025. AFL++ QEMU Mode. https://github.com/AFLplusplus/AFLplusplus/tree/stable/qemu_mode
[46] AFL++ Developers. 2025. AFL++ QEMU Mode: qemuafl/target/i386/cpu.h. https://github.com/AFLplusplus/qemuafl/blob/master/target/i386/cpu.h
[47] AFL++ Developers. 2025. AFLplusplus/qemuafl. https://github.com/AFLplusplus/qemuafl
[48] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (Oakland)*.
[49] dotPDN LLC and Rick Brewster. 2020. Paint.NET. https://www.getpaint.net/

[50] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary Rewriting Without Control Flow Recovery. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[51] Michael Eddington. 2025. Peach Fuzzing Platform. https://peachtech.gitlab.io/peach-fuzzer-community/
[52] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001).
[53] Michael D Ernst, Adam Czeisler, William G Griswold, and David Notkin. 2000. Quickly Detecting Relevant Program Invariants. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
[54] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of computer programming* 69, 1-3 (2007).
[55] FBReader: Favourite Book Reader. 2020. FBReader. https://fbreader.org/
[56] Andrea Fioraldi. 2024. Andreafioraldi/Afl-Qemu-Cov. https://github.com/andreafioraldi/afl-qemu-cov
[57] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *USENIX Security Symposium (SEC)*.
[58] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[59] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2021. Fuzzing Binaries for Memory Safety Errors with QASan. https://github.com/andreafioraldi/qasan
[60] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*.
[61] Benjamin Fleischer. 2020. FUSE for macOS. https://osxfuse.github.io/
[62] Fortinet. 2025. SolarWinds Supply Chain Attack. https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack
[63] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*.
[64] Google. 2015. SyzKaller: Unsupervised Coverage-Guided Kernel Fuzzer. https://github.com/google/syzkaller
[65] Google. 2025. AddressSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer
[66] Google. 2025. Honggfuzz. https://github.com/google/honggfuzz
[67] Google Project Zero. 2016. WinAFL: A Coverage-Guided Fuzzer for Windows Applications. https://github.com/googleprojectzero/winafl
[68] Google Project Zero. 2018. JackaLope: Coverage-Guided Binary Fuzzing Framework. https://github.com/googleprojectzero/Jackalope
[69] Israel Herraiz, Daniel Izquierdo-Cortazar, and Francisco Rivas-Hernández. 2009. FLOSSMetrics: Free/Libre/Open Source Software Metrics. In *European Conference on Software Maintenance and Reengineering*.
[70] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. 2024. Everything is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference. In *IEEE Symposium on Security and Privacy (Oakland)*.
[71] IDT Corporation. 2021. IDT 2021 Annual Report. https://www.annualreports.com/HostedData/AnnualReportArchive/i/NYSE_IDT_2021.pdf
[72] Intel Corporation. 2025. Pin: A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html
[73] Microsoft 365 Security Intelligence, Microsoft Threat. 2021. HAFNIUM Targeting Exchange Servers with 0-Day Exploits. https://www.microsoft.com/en-us/security/blog/2021/03/02/hafnium-targeting-exchange-servers/
[74] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[75] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.
[76] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*.
[77] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *IEEE/ACM International Conference Automated Software Engineering (ASE)*.
[78] Zhibo Liu and Shuai Wang. 2020. How Far We Have Come: Testing Decompilation Correctness of C Decompilers. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[79] Mel Llaguno. 2017. *2017 Coverity Scan Report.* Technical Report. Synopsys Inc. https://www.synopsys.com/blogs/software-security/2017-coverity-scan-report-open-source-security/
[80] LLVM. 2025. libFuzzer: A Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html

[81] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-Box Fuzzing Towards Combinatorial Difference. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.
[82] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021).
[83] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with Data Dependency Information. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.
[84] Xiaozhu Meng and Barton P Miller. 2016. Binary Code Is Not Easy. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[85] Microsoft Corporation. 2025. Microsoft Windows. https://www.microsoft.com/windows
[86] Microsoft Corporation. 2025. Windows 365 Government. https://www.microsoft.com/en-us/windows-365/government
[87] Mozilla Security. 2025. Dharma: A Generation-based, Context-free Grammar Fuzzer. https://github.com/MozillaSecurity/dharma
[88] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead Through Coverage-Guided Tracing. In *IEEE Symposium on Security and Privacy (Oakland)*.
[89] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Breaking Through Binaries: Compiler-Quality Instrumentation for Better Binary-Only Fuzzing. In *USENIX Security Symposium (SEC)*.
[90] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[91] National Audit Office. 2017. Investigation: WannaCry Cyber Attack and the NHS: NAO Report. https://www.nao.org.uk/reports/investigation-wannacry-cyber-attack-and-the-nhs/
[92] National Vulnerability Database. 2023. NVD: CVE-2023-41061. https://nvd.nist.gov/vuln/detail/cve-2023-41061
[93] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Transactions on Software Engineering and Methodology* 23, 4 (2014).
[94] Oracle Corporation. 2021. Oracle Security Alert Advisory: CVE-2021-44228. https://www.oracle.com/security-alerts/alert-cve-2021-44228.html
[95] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019).
[96] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *IEEE Symposium on Security and Privacy (Oakland)*.
[97] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-Based Symbolic Execution for Binaries. In *ISOC Network and Distributed System Security Symposium (NDSS)*.
[98] Giacomo Priamo, Daniele Cono D'Elia, Mathias Payer, and Leonardo Querzoni. 2026. Towards Path-Aware Coverage-Guided Fuzzing. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
[99] Zhenxiao Qi, Jie Hu, Zhaoqi Xiao, and Heng Yin. 2024. SymFit: Making the Common (Concrete) Case Fast for {Binary-Code} Concolic Execution. In *USENIX Security Symposium (SEC)*.
[100] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. 2023. NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023).
[101] Georgy Savidov and Andrey Fedotov. 2021. Casr-Cluster: Crash Clustering for Linux Applications. In *Ivannikov ISP RAS Open Conference (ISPRAS)*.
[102] Eric Schulte, Michael D Brown, and Vlad Folts. 2022. A Broad Comparative Evaluation of x86-64 Binary Rewriters. In *Workshop on Cyber Security Experimentation and Test (CSET)*.
[103] Kostya Serebryany. 2017. OSS-Fuzz: Google's Continuous Fuzzing Service for Open Source Software. In *USENIX Security Symposium (SEC)*.
[104] Avast Software. 2019. RetDec Issue #563. https://github.com/avast/retdec/issues/563
[105] Axel Souchet. 2020. What The Fuzz: A Snapshot-Based Fuzzer for Windows. https://github.com/0vercl0k/wtf
[106] U.S. Securities and Exchange Commission. 2025. UWM Commission File. https://www.sec.gov/Archives/edgar/data/1783398/000178339826000013/uwmc-20251231.htm
[107] UW Program Languages and Software Engineering Group. 2025. The Daikon Invariant Detector User Manual. https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Invariant-list
[108] Pengfei Wang, Xu Zhou, Tai Yue, Peihong Lin, Yingying Liu, and Kai Lu. 2024. The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing. *Software Testing, Verification and Reliability* 34, 2 (2024).
[109] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy (Oakland)*.

[110] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic Binary Recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[111] Wei-Cheng Wu, Yutian Yan, Hallgrimur David Egilsson, David Park, Steven Chan, Christophe Hauser, and Weihang Wang. 2024. Is This the Same Code? A Comprehensive Study of Decompilation Techniques for WebAssembly Binaries. In *International Conference on Security and Privacy in Communication Systems (SecureComm)*.
[112] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[113] Zao Yang and Stefan Nagy. 2025. BIN2WRONG: A Unified Fuzzing Framework for Uncovering Semantic Errors in Binary-to-C Decompilers. In *USENIX Annual Technical Conference (ATC)*.
[114] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium (SEC)*.
[115] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Received 2025-09-11; accepted 2025-12-22