One Fuzz Doesn’t Fit All: Optimizing Directed Fuzzing via Target-tailored Program State Restriction

Prashast Srivastava  
Purdue University  
United States

Stefan Nagy  
University of Utah  
United States

Matthew Hicks  
Virginia Tech  
United States

Antonio Bianchi  
Purdue University  
United States

Mathias Payer  
EPFL  
Switzerland

ABSTRACT
Fuzzing is the de-facto default technique to discover software flaws, randomly testing programs to discover crashing test cases. Yet, a particular scenario may only care about specific code regions (for, e.g., bug reproduction, patch or regression testing)—spurring the adoption of directed fuzzing. Given a set of pre-determined target locations, directed fuzzers drive exploration toward them through distance minimization strategies that (1) isolate the closest-reaching test cases and (2) mutate them stochastically. However, these strategies are applied onto every explored test case irrespective of whether they ever reach the targets—stalling progress on the paths where targets are unreachable. Accelerating directed fuzzing requires prioritizing target-reachable paths.

To overcome the bottleneck of wasteful exploration in directed fuzzing, we introduce tripwiring: a lightweight technique to pre-empt and terminate the fuzzing of paths that will never reach target locations. By constraining exploration to only the set of target-reachable program paths, tripwiring curtails directed fuzzers’ search noise—while unshackling them from the high-overhead instrumentation and bookkeeping of distance minimization—enabling directed fuzzers to obtain up to 99% higher test case throughput. We implement tripwiring-directed fuzzing as a prototype, SieveFuzz, and evaluate it alongside the state-of-the-art directed fuzzers AFLGo, BEACON and the leading undirected fuzzer AFL++. Overall, across nine benchmarks, SieveFuzz’s tripwiring enables it to trigger bugs on an average 47% more consistently and 117% faster than AFLGo, BEACON and AFL++.

CCS CONCEPTS
• Security and privacy → Software and application security.

KEYWORDS
directed fuzzing, tripwiring, hybrid analysis, state space restriction

ACM Reference Format:
Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. 2022. One Fuzz Doesn’t Fit All: Optimizing Directed Fuzzing via Target-tailored Program State Restriction. In Annual Computer Security Applications Conference (ACSAC ’22), December 5–9, 2022, Austin, TX, USA. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3564625.3564643

1 INTRODUCTION
Quality assurance is an important component of the software development life cycle, requiring significant resources for identifying, triaging, and fixing defects both pre- and post-deployment. In working toward offsetting this burden, the last two decades has seen software fuzz testing (fuzzing) become the most successful and ubiquitous approach for automated software defect discovery.

Most fuzzers target broad defect discovery (e.g., OSS-Fuzz [29], libFuzzer [28], and AFL++ [7])—embracing code coverage guidance to explore the software under test (SUT) by maximizing code coverage of generated test cases. But, despite the success of coverage-guided fuzzing [1, 8, 16, 18, 25], its from-scratch, all-or-nothing exploration style is unsuited to the many critical software QA tasks that target specific code locations (e.g., bug reproduction, regression testing, or patch testing). In such contexts, software testers instead turn to targeted fuzzing approaches known as directed fuzzing.

Directed fuzzers replace fuzzing’s conventional broad search with one targeting pre-determined locations (e.g., a suspected defect location), using distance minimization [4, 5, 23] to drive fuzzing closer and closer to them. To achieve directedness, distance minimization computes the distance of every generated test case relative to each target location, saving only those that shorten this distance as fuzzing continues. However, as distance measurement is performed at runtime for all test cases—including the overwhelming majority that are incapable of ever reaching the target locations—directed fuzzers incur significantly more overhead per execution due to the higher instrumentation cost associated with distance measurement. Furthermore, the current scheme of using distance minimization is specifically ill-suited for disjoint target locations—locations that can be reached without requiring a large part of the software functionality to be exercised. For such target locations, distance minimization’s costly, always-on analysis becomes overwhelmed by target-unreachable paths, thus slowing down directed fuzzers’ progress—beyond even their undirected counterparts.

To break free from distance minimization and quickly filter-out target-unreachable paths, we introduce tripwiring: a lightweight approach to accelerate directed fuzzing through a target-tailored restriction of program state. At the core of our efforts is our observation that a fuzzer’s search is stochastic and highly influenced by the program’s observable code coverage; and should a code region...
be made inaccessible, a fuzzer’s exploration will shift toward pursuing whatever program paths remain accessible. We demonstrate that, through a hybrid static and dynamic analysis technique, it is feasible to identify and refine the set of target-relevant code regions while tripwiring (i.e., preempting and terminating) target-irrelevant ones—enabling effective directed fuzzing of disjoint target locations that is unburdened by distance minimization. To evaluate tripwiring’s effectiveness, we implement a proof-of-concept directed fuzzer called SieveFuzz, and evaluate it alongside the state-of-the-art directed fuzzers AFLGo [4] and BEACON [13], as well as the state-of-the-art undirected fuzzer AFL++ [7]. We examine a real-world context in which directed fuzzing is deployed for targeted defect discovery—reproducing third-party-reported security vulnerabilities—and demonstrate that across a corpus of ten disjointly-located security vulnerabilities in nine varied benchmarks, tripwiring accelerates directed fuzzing by an average of 140%, 93%, and 118% faster than AFL++, AFLGo, and BEACON, respectively, while obtaining 37%, 42%, and 61% more consistent targeted defect discovery, respectively.

In summary, this paper makes the following contributions:

- We introduce tripwiring: a lightweight technique for target-tailored directed fuzzing that restricts fuzzing to only the program search space guaranteed relevant to reaching user-determined target locations.
- We expose the fundamental limitations that impede state-of-the-art directed fuzzers from achieving effective and efficient directedness for disjoint target locations. For such target locations, we show that tripwiring is a more optimal directed fuzzing methodology than distance minimization.
- We design SieveFuzz: an implementation of tripwiring for accelerated directed fuzzing. We evaluate it on a corpus of nine benchmarks with ten known disjointly-located security vulnerabilities; and show that, on average, SieveFuzz exposes these bugs in 117% less time and 47% more consistently than the leading undirected and directed fuzzing techniques.
- Source code of our framework along with the evaluation artifacts are made available at https://github.com/Hex Hive/SieveFuzz

2 BACKGROUND

Below we provide relevant details on software fuzzing, and the differentiation between guided and directed fuzzing policies.

Directed Fuzzing. For targeted exploration objectives such as patch testing, security researchers introduced the concept of directed fuzzing [4, 9, 37], which layers conventional guided fuzzing with additional mechanisms to “direct” fuzzing toward specific target locations. Most state-of-the-art directed fuzzers embrace distance minimization as their mechanism of directedness [4, 5, 23, 24]. In this technique, the SUT’s inter- and intra-procedural control-flow graphs are first instrumented to log distances of each basic block relative to the intended target site. Second, at runtime, the fuzzer computes each test case’s harmonic mean distance over its covered code. Lastly, mutation candidates are chosen from the pool of seeds with shortest distances to the target, ideally guiding fuzzing to converge on the shortest path.

3 PITFALLS OF DISTANCE MINIMIZATION

Distance-minimization-based directed fuzzers converge on target locations by focusing only on those test cases whose execution paths are closest to reaching them. Yet, this approach requires a directed fuzzer to (1) compute the path-to-target distances for every test case, including the overwhelming majority that will inevitably be discarded because they cannot reach target locations; and (2) perform a greedy search across all observed paths to pinpoint the small set of desired paths to continue exploring. The high costs of both of these steps creates a compounding bottleneck for directed fuzzing—incurring a much higher overhead per execution than undirected fuzzing—making it exceedingly difficult to recover when exploration plateaus on paths that will never reach target locations.

To quantify the performance cost of distance minimization, we replicate a common directed fuzzing usage scenario: identifying a target location (e.g., a suspected security vulnerability) [4, 5, 24] and using directed fuzzing to synthesize a proof-of-concept violating input. We perform a case study on a synthetic benchmark popular in the fuzzing literature [17, 31, 32, 42] that is known to contain a critical memory safety vulnerability (NULL pointer dereference), and detail our experimental results below.

Experiment Setup. For our defect discovery experiment, we select the DARPA Cyber Grand Challenge benchmark KPRCA-00038: a language interpreter containing a NULL pointer dereference in the function cgc_program_parse. As shown in Listing 1, to trigger this memory safety violation, a fuzzer must (1) satisfy the language semantics to first insert an empty statement; and (2) insert a non-empty statement that triggers the dereference. In this program, cgc_parse_statements represents a disjoint target because most of the program’s functionality (e.g., cgc_program_run and everything following it) does not precede it in execution.

To evaluate distance minimization, we select the state-of-the-art directed fuzzer AFLGo [4] and configure it to target the aforementioned vulnerable function; and further evaluate it alongside AFL [19], the state-of-the-art undirected (i.e., coverage-guided) fuzzer which AFLGo is implemented atop of. Following Klees et al. [15], we perform 10×24-hour fuzzing campaigns per each fuzzer.

Consequence 1: Poor Performance. After performing all fuzzing campaigns, we post-process observed crashes to ascertain which fuzzer trials successfully triggered cgc_parse_statements’s NULL pointer dereference. We compute and compare two metrics between
Listing 1 Simplified code snippet to show distance minimization’s wastefulness.

```c
int main(void) {
    io_t io;
    program_t p;
    cgc_io_init_fd(&io, STDIN);
    cgc_program_init(&p, &io);
    // Bug-triggering path through cgc_program_parse
    if (!cgc_parse_statements(prog, &tmp)) {
        // Irrelevant functionality below not relevant
        goto tail;
    }
    // Relevant towards triggering the bug
    if (cgc_parse_run(prog, &io)) {
        ...
    }
    else {
        ...
    }
    static int cgc_program_parse(program_t *prog) {
        ...
        stmt_t *tail = NULL;
        while () {
            stmt_t *tmp;
            // cgc_parse_statements may return NULL value in 'tmp'
            if (!cgc_parse_statements(prog, &tmp))
                goto tail;
            ...
            if (stmt == NULL) {
                tail = stmt = tmp;
            }
            else {
                tail->next = tmp;
            }
        }
    }
}
```

both fuzzers: (1) the relative time at which each fuzzer exposed the security vulnerability in the campaign; and (2) the unique number of trials which succeeded in exposing the vulnerability.

Overall, we observe that directed fuzzer AFLGo is outperformed by the undirected AFL, with AFL exposing the bug 92% faster. Furthermore, we observe that AFL successfully reaches and exposes the bug in 2 of 10 trials, while directed fuzzer AFLGo succeeds only once. Thus, distance minimization—despite its machinery designed to quickly converge on target locations—ultimately performs both slower and less reliably than undirected fuzzing in reproducing this disjointly-located security vulnerability.

Consequence 2: Unconstrained Exploration. To further evaluate the performance disparity between distance-minimization-directed and undirected fuzzing, we profile both fuzzers’ campaigns to measure the magnitude of effort spent on code irrelevant to reaching target locations. We observe that AFLGo has separate Exploration (i.e., undirected) and Exploitation (i.e., directed) modes, with Exploitation being where distance minimization is performed; and thus, we limit our profiling of AFLGo to its Exploitation mode. We cross-reference the set of code regions exercised by each fuzzer with the execution path of the vulnerability’s proof-of-concept (PoC) input, marking any non-PoC code regions as extraneous.

On average, our results show that both directed AFLGo and undirected AFL execute over 29% more program functions than contained in the vulnerability PoC trace. Thus, for disjoint target locations such as the vulnerable function cgc_program_parse, distance minimization is no more effective than undirected fuzzing at constraining the search down the set of target-relevant program paths. Coupled with its higher per-execution overhead, distance minimization pays a significant price for its greedy search across the program state space—leaving undirected fuzzing often more successful at targeted defect discovery.

Impetus: Distance minimization facilitates directedness via dynamic distance calculation and repeated fuzzing per test case. Yet, only a small minority of test cases converge on target locations. This higher common-case overhead leaves distance minimization costlier than undirected fuzzing—particularly when exploration stalls in regions that never reach target locations. Achieving faster and more consistent directedness necessitates an approach focusing on target-relevant code regions.

4 OVERCOMING THE BOTTLENECKS OF DIRECTEDNESS

Current directed fuzzers rely on distance minimization, performing directed search by prioritizing test cases reaching closer to target locations. However, as § 3 reveals, the sensitivity of distance minimization to search noise significantly impedes the effectiveness of directed fuzzing. Thus, as distance minimization’s problems are inherited by most directed fuzzers, the full performance potential of directed fuzzing remains unrealized.

Figure 1: A visualization of tripwiring-directed fuzzing.

To overcome the bottlenecks of directed fuzzing, we leverage the observation that a fuzzer’s search in the program state space is stochastic and highly influenced by the program’s reachable control-flow. An undirected fuzzer will aim to maximize exploration across all program paths; but, should only a subset of control-flow be reachable, it will aim to maximize its search across the subset. We thus envision an approach that achieves directed fuzzing by tailoring (i.e., restricting) the search space to only the subset of reachable paths that are guaranteed relevant to reaching the target location.

We call this approach tripwiring (Figure 1): at a high level, we repurpose conventional control-flow and path detection to identify (and refine ad hoc) the set of paths to the target location; and modify the coverage-guided fuzzing workflow to only explore these regions, preempting and terminating when a fuzzing execution “trips” this region’s boundary—thereby achieving directedness through constraining stochastic search toward the target.

5 PREEMPTIVE TERMINATION

Existing directed fuzzers rely on distance minimization to steer exploration toward target locations. However, this mechanism is kept always-on for all test cases irrespective of their relevance to reaching the target locations—making these fuzzers highly sensitive to search noise: code regions (e.g., functions and basic blocks) guaranteed to never precede target locations in execution flow. The inability to recognize and suppress search noise leaves minimization-directed fuzzers crippled by the instrumentation and bookkeeping costs that they waste on these paths—and thus, too slow to be effective at bug discovery.
We posit that directed fuzzing wastefulness is avoidable by preemptively terminating exploration of regions proven to never precede target locations. This presents two key performance advantages: (i) SUT execution—over 90% of fuzzers’ runtime [20]—will not be wasted on repeatedly measuring the code coverage and target distances of target-irrelevant paths, and (ii) as we filter-out these paths before they are ever explored, fuzzers will not waste any resources on processing these test cases in succession.

5.1 Tripwiring

In this section, we present our methodology for identifying regions guaranteed to be search noise (i.e., will never precede target locations). Furthermore, we detail how we resolve analysis obstacles caused by indirect control-flows.

**Methodology.** To eliminate directed fuzzing search noise, SieveFuzz requires knowing which code regions are (1) on target-reachable paths and (2) not on them. To this aim, we statically analyze the SUT’s inter-procedural control flow graph (ICFG) and call graph (CG), and flag all regions on identifiable paths from the program entry to the target sites. Algorithm 1 details our approach.

**Algorithm 1** Tripwiring algorithm for pruning target-unreachable code regions.

```
Require: Target code location \( T \), ICFG \( I \), and call graph \( C \).
Ensure: Set of tripwired code regions \( R \).
1: \( N = I.\text{getEntryNode}(T) \)
2: \( W = [-N] \)
3: \( \text{Allow} = \emptyset \)
4: while \( W \neq \emptyset \) do
5: \( N^\prime = W.\text{pop}() \)
6: \( E = I.\text{getInEdges}(N^\prime) \)
7: for \( E' \in E \) do
8: if \( \text{notSeen}(E') \) then
9: \( N^\prime\prime = E'.\text{getSource}() \)
10: if \( C.\text{isReachable}(N^\prime\prime, T) \) then
11: \( \text{Allow}.\text{add}(N^\prime\prime.\text{getRegionID}()) \)
12: \( W.\text{add}(N^\prime\prime) \)
13: end if
14: \( \text{addSeen}(E') \)
15: end if
16: end for
17: end while
18: \( U = C.\text{get}(\text{AllRegions}()) \)
19: return \( S = U - \text{Allow} \)
```

We deploy our lightweight analysis atop the SUT’s ICFG and incorporate calling-context sensitivity using the CG for higher precision. First, we initialize a work-list (Line 2) of the target location’s entry node (Line 1) as well as an empty allow-list (Line 3). Then, for each work-list member, we perform the following: (i) Pick all incoming edges for the node from the ICFG, (ii) For each edge, identify its source and corresponding node from the ICFG, and (iii) Using the CG, check if the target is reachable from the source; and if so, add the source to the allow-list and the corresponding node to the work-list. All regions outside the above constructed allow-list are marked unnecessary and tripwired for termination (Line 19).

As our ICFG is insensitive to calling contexts, our analysis may initially over-approximate the set of target-relevant code regions. To mitigate this, we incorporate the results of a function reachability analysis performed on the CG (exemplified in Appendix A).

**Indirect Transfers.** Because we rely on static analysis to generate our ICFG and CG, another challenge in supporting tripwiring is handling indirect transfers: control-flow to dynamically-determined destinations. Solving indirect transfers statically for real-world codebases using techniques such as points-to and/or value-set analysis results in significant over-approximation of candidates targets for these transfers. This in turn brings a high risk of under-tripwiring—i.e., over-approximating the code that should be explored and, hence, an inability to uphold fuzzing directedness.

To avoid the risk of under-tripwiring, we dynamically update our CG with every newly-covered indirect branch. With each new piece of information, we re-perform our analysis to refine our view of the reachable area and adjust our tripwiring accordingly. Though this re-analysis interposes some overhead on fuzzing, the exponentially-decreasing rate of new coverage [20] ensures that re-analysis is a rare event in practice—and thus adds no discernible slowdown.

As we resolve indirect calls dynamically, target locations may be absent from our initial reachability analysis. However, we observe that it is sufficient to merely seed our analysis with traces from a few fuzzer-generated program test cases. Should a more diverse set of seed traces be needed, we expect to incur only a slightly higher upfront cost (e.g., an initial cycle of undirected fuzzing).

6 IMPLEMENTATION: SIEVEFUZZ

In this section we introduce SieveFuzz: our implementation of tripwiring for accelerated directed grey-box fuzzing. In its current prototype, it operates over the source code of the fuzz target. Below we discuss SieveFuzz’s core architecture.

6.1 Architectural Overview

We implement SieveFuzz atop the industry-standard grey-box fuzzer AFL++ [7]. To facilitate on-demand reachability analysis (§ 5.1), we integrate a client-server communication between our fuzzer and analysis components—forwarding any indirect edges captured to our static analysis, which then updates the dynamic control-flow graph before updating reachability and tripwiring analyses. For our static analysis we utilize the LLVM-based SVF framework [33]. We inject the instrumentation to perform preemptive termination at function-level granularity using an LLVM pass.

6.2 High-level Fuzzing Workflow

SieveFuzz follows the state machine model presented in Figure 2, comprising of the following three steps:

![Figure 2: SieveFuzz’s high-level state machine. Here, reachable denotes that our analysis identifies some path(s) from the program entry point to the target location.](image-url)
**Initial Analysis (INIT):** Initially, our fuzzer queries to determine whether the target is reachable from our initial ICFG and CG analyses. Should the target be unreachable, we conclude some statically-identifiable indirect call edge(s) are missing and attempt to recover them by briefly running the exploration state (EXP). However, as discussed in § 5.1, we often avoid exploration by repurposing commonly-provided developer test suites or test cases from prior fuzzing campaigns as seed traces to recover these edges. When the target is reachable, we then move on to our Fuzzing (FUZZ) stage.

**Exploration (EXP):** If the target is unreachable (i.e., no path(s) exist to it from the SUT’s entry), we turn to undirected, non-tripwired fuzzing to diversify the set of candidate seed traces. At each step, we monitor for new indirect edges and update our reachability analysis accordingly; should a new path(s) be seen intersecting the target location, we exit and move on to our Fuzzing (FUZZ) stage.

**Tripwired Fuzzing (FUZZ):** As soon as the targets are reachable (i.e., there are some path(s) to the target), tripwired-directed fuzzing begins: preempting and terminating execution of regions not within our target-reachable coverage set. As in the Exploration phase, we report any newly-covered indirect edges to our static analysis server. Following reachability analysis updates, we amend our tripwiring instrumentation (e.g., adding or removing tripwires). As this process continues and our tripwiring evolves, we steer fuzzing closer to reaching the target location.

### 6.3 Maintaining Fast On-demand Analysis

To refine our tripwiring, we engage reachability analysis on-demand when new indirect edges are found during fuzzing. While we can perform this analysis between fully stopping and restarting fuzzing, the cost of reinitiating fuzzing from a terminated state incurs a prohibitively-high startup overhead that cripples fuzzing throughput. We instead adopt a client-server communication protocol: upon analyzing a new indirect edge, we resume the client fuzzer from a paused (but not terminated) state after the static analysis server reports its completion. Our current implementation adopts a single-core sequential design. Regardless, the negligible rate of coverage-increasing test cases (less than 1 in 10,000 on average [20]) means that this analysis is only invoked sparingly—amortizing this infrequent-case cost over the course of fuzzing.

### 6.4 Maintaining Fast SUT Execution

SieveFuzz maintains high-throughput directed fuzzing through its lightweight instrumentation passes to accommodate tripwiring’s (1) preemptive termination and (2) indirect edge monitoring.

**Preemptive Termination.** As the SUT is being instrumented for fuzzing, we assign a unique numeric ID to each code region in the SUT (in our current prototype: functions). Then, we hook the start of each region to call into a runtime library with its ID; we link this library to the SUT, and utilize it to enforce (and dynamically update) our tripwiring preemptive termination policy.

In our prototype implementation, we maintain an activation bitmap with each bit corresponding to the unique ID assigned to each code region (i.e., function) in the SUT. If a bit is unset, then the function corresponding to that bit is tripwired and prevented from being executed. If a bit is set, the corresponding function is permitted uninterrupted execution. SieveFuzz dynamically maintains this bitmap in sync with the set of target-relevant regions identified by the static analysis module (§ 6.3). Thus, all regions marked for tripwiring will have their corresponding activation map bit unset.

**Indirect Call Tracking.** We instrument all indirect branch sites to extract these edges’ destinations during runtime. We utilize this technique in our current function-level prototype to track indirect (caller, callee) pairs: we assign each function a unique 32-bit ID; and for every indirect call edge, we compute a 64-bit edge ID by splicing-together the ID’s of its caller and callee. As tracking such calls (1) requires only constant-time operations and (2) attains a linear complexity (O(e) where e = the total number of unique indirect edges), our analysis cost adds insignificant overhead.

### 6.5 Maintaining Exploration Diversity

Tripwiring achieves directedness by driving conventional fuzzing’s coverage-maximizing search strategy toward target locations: constraining the region of accessible control-flow to only the code relevant to reaching the target. However, in case no new coverage is found, most fuzzers will begin shuffling seeds for mutation at random. Yet, such strategies are incompatible with certain bugs’ complex triggering semantics that require successive execution of the target itself (e.g., stack exhaustions). Thus, an effective directed fuzz tool must not only reach a target bug—but also trigger it.

To overcome this issue in SieveFuzz, we develop an on-demand execution diversity heuristic to prioritize the mutation of test cases with greater coverage of target-relevant code regions. It focuses SieveFuzz’s available fuzzing on program paths that intersect more bug-relevant program subroutines. By steering a plateaued fuzzing expedition in this way, we increase the likelihood of triggering new runtime states to reach and trigger complex bugs.

We insert instrumentation in the fuzz target to keep track of trace length for each test case. Here, trace length refers to the number of target-relevant code regions triggered by a test case. In SieveFuzz, the trace length corresponds to the number of functions executed by a test case and to calculate the trace length, SieveFuzz inserts a single integer increment operation at function-level granularity.

The observed trace lengths can drastically vary depending on the fuzz target complexity and the fuzzer capabilities to explore the underlying program state space. Consequently, using the trace length as a metric as-is to decide on test case prioritization can lead to SieveFuzz wasting its computation cycles fuzzing test cases with a large trace length. To address this issue, SieveFuzz keeps track of the average trace length observed over the course of a fuzzing campaign. The computation cycles allocated to a test case are decided on the basis of the degree to which an test case is proportionally larger or smaller than the average trace length observed until that point.

### 7 EVALUATION

Our evaluation of the effectiveness of tripwiring-directed fuzzing is guided by three fundamental research questions: (i) **RQ1:** Is tripwiring effective and fast at restricting fuzzing-reachable search space?, (ii) **RQ2:** Do the benefits of tripwiring improve directed fuzzing effectiveness and speed?, and (iii) **RQ3:** Are there properties that make a target location well suited to tripwiring?
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bug Type</th>
<th>Functionality</th>
<th>Type</th>
<th>Benchmark</th>
<th>Bug Type</th>
<th>Functionality</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROMU-00039</td>
<td>Stack BoF</td>
<td>Network protocol</td>
<td>S</td>
<td>gif2tga</td>
<td>NPD</td>
<td>GIF format converter</td>
<td>R</td>
</tr>
<tr>
<td>KPRCA-00038</td>
<td>NPD</td>
<td>Language Interpreter</td>
<td>S</td>
<td>jasper</td>
<td>Heap BoF</td>
<td>Image processing tool</td>
<td>R</td>
</tr>
<tr>
<td>KPRCA-00051</td>
<td>Global BoF</td>
<td>Bookkeeping</td>
<td>S</td>
<td>listswf</td>
<td>Heap BoF</td>
<td>Flash format processor</td>
<td>R</td>
</tr>
<tr>
<td>mJS</td>
<td>FPE</td>
<td>Language interpreter</td>
<td>R</td>
<td>Tidy</td>
<td>Heap UAF</td>
<td>Markup language parser</td>
<td>R</td>
</tr>
<tr>
<td>tiffcp-1</td>
<td>OOB Read</td>
<td>TIFF format manipulator</td>
<td>R</td>
<td>tiffcp-2</td>
<td>Resource Exhaustion</td>
<td>TIFF format manipulator</td>
<td>R</td>
</tr>
</tbody>
</table>


We compare our tripwiring prototype, SieveFuzz, against the state-of-the-art distance-minimization-directed fuzzer AFLGo [4]. To examine how SieveFuzz performs versus undirected fuzzing, we further evaluate the state-of-the-art undirected fuzzer AFL++ [7]. Lastly, we evaluate the newly-released (at the time of writing) directed fuzzer BEACON [13], which employs an alternative directedness approach that aims to synthesize and satisfy target-specific path preconditions (i.e., “precondition-directed”). Below details our evaluation benchmarks and procedures.

**Benchmarks.** To replicate the conditions under which directed fuzzing is deployed in real-world targeted detected discovery, we distill a set of three ground-truth memory bugs sourced from the DARPA Cyber Grand Challenge (CGC) [3] corpus due to its popularity in the fuzzing literature [25, 32, 40]. We further expand this set with five benchmarks from real-world, open-source software bug reports and two ground-truth bugs in real-world programs from the Magma fuzzing benchmark suite [11]. As Table 1 shows, our benchmark selection covers a diverse range of defect semantics (e.g., overflows and dangling pointers) and functionality. Furthermore, as we will show later in § 7.1, this selection of benchmarks contain ground truth bugs in target locations that are disjoint from the rest of the program to a varying degree.

**Experiment Procedure and Infrastructure.** To answer RQ1, we compute SieveFuzz’s search space reduction as the percentage of target-irrelevant code regions tripwired (i.e., functions irrelevant to reaching bug locations). For answering RQ2, we record each fuzzer’s time-to-exposure for all ten bugs. To answer RQ3, we investigate if there is a correlation between the disjointness of a target location and the performance of SieveFuzz and AFLGo.

We follow the evaluation standard in the literature [10, 14, 15, 39, 42] and select a 24-hour trial duration for each experiment at 10 trials to attain sufficient statistical certainty. To determine the magnitudes of statistical differences, we perform the Vargha and Delaney A12 test [35] in comparing bug exposure times. For each campaign, we run each fuzzer on a core single in single-threaded mode. We configure both AFLGo and SieveFuzz by targeting them on the source code locations corresponding to each benchmark’s bug (i.e., the crashing instruction as reported by triage tools like AddressSanitizer [30]). All fuzzing trials are seeded with a one-character starting test case except for bugs from the Magma benchmark for which we use the author-provided seeds. We conduct all evaluations on an Intel Cascade Lake instance on the Google Cloud Platform with 40GB RAM running Debian 9.

### 7.1 RQ1: Tripwiring’s Search Space Restriction

To understand tripwiring’s effectiveness and efficiency in supporting directed fuzzing, we perform experiments to (1) measure the percentage of code regions tripwired-out (restricted from fuzzing); and (2) compute the costs of pre-fuzzing (tripwiring initialization) and on-demand analysis (handling new indirect edges). We discuss our procedures and results below.

**Results: Magnitude of Space Restriction.** To perform effective directed fuzzing, tripwiring must remove target-irrelevant functionality. To capture the extent to which tripwiring achieves this goal, we modify SieveFuzz to report the total number of code regions (at function-level) culled when the target function becomes reachable, and report our results in Table 2.

Across all benchmarks, tripwiring eliminates 29% of code regions on average as target-irrelevant functionality—preventing directed fuzzing from wasting computation on the many paths that do not reach these bugs. For two bugs in jasper and listswf, tripwiring omits a smaller percentage of code regions (8–12%); in manually examining these, we observe that both bugs intersect the majority of code paths, forcing tripwiring to perform a conservative reduction.

**Results: Initialization Cost.** Current directed fuzzers [4, 5] incur significant initialization overheads [23] due to the excessive instrumentation-time effort needed to compute and embed target distances for all code regions. As it is crucial for developers to spin-up directed fuzzing as timely and effortlessly as possible, we measure the initialization cost of tripwiring-directed fuzzing by profiling SieveFuzz’s analyses times and report our results in Table 2.

On average, we see that it takes SieveFuzz on an average just 59 ms to complete the tripwiring process across our nine benchmarks. More importantly, throughout our evaluation, we observe a linear relationship between the tripwiring analysis time and the benchmark size showcasing evidence of the scalability of our approach. In addition, we observe that AFLGo and BEACON incur mean initialization costs 188x and 36.3x higher than SieveFuzz’s cumulative runtime analyses (Re-run Cost in Table 2) time respectively. Recently, AFLGo added an alternative feature aimed towards reducing this overhead. With this feature, AFLGo’s initialization overhead drops down to 2.2x more than SieveFuzz’s cumulative runtime analyses time. Therefore, beyond attaining a low fuzzing-startup cost, we conclude that tripwiring’s negligible analysis time is well-suited to deployment during fuzzing—making tripwiring supportive of high-throughput directed fuzzing.
Table 2: Percentage of code regions (at function level) removed by tripwiring during fuzzing, and the analysis time spent in tripwiring’s pre-fuzzing initialization.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reduction</th>
<th>Analysis Cost (ms)</th>
<th>New Indir Edges</th>
<th>Re-runs</th>
<th>Re-run Cost (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROMU-00039</td>
<td>54%</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>KPRCA-00038</td>
<td>54%</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>KPRCA-00051</td>
<td>34%</td>
<td>23</td>
<td>4</td>
<td>3</td>
<td>0.07</td>
</tr>
<tr>
<td>gif2tga</td>
<td>38%</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>jasper</td>
<td>8%</td>
<td>60</td>
<td>71</td>
<td>29</td>
<td>1.74</td>
</tr>
<tr>
<td>listswf</td>
<td>12%</td>
<td>10</td>
<td>73</td>
<td>31</td>
<td>0.31</td>
</tr>
<tr>
<td>mjs</td>
<td>39%</td>
<td>26</td>
<td>2</td>
<td>2</td>
<td>0.05</td>
</tr>
<tr>
<td>Tidy</td>
<td>20%</td>
<td>91</td>
<td>87</td>
<td>44</td>
<td>4.00</td>
</tr>
<tr>
<td>tiffcp-1</td>
<td>18%</td>
<td>194</td>
<td>87</td>
<td>29</td>
<td>5.62</td>
</tr>
<tr>
<td>tiffcp-2</td>
<td>18%</td>
<td>175</td>
<td>87</td>
<td>29</td>
<td>5.07</td>
</tr>
<tr>
<td>Mean</td>
<td>29%</td>
<td>59 ms</td>
<td>41</td>
<td>16.7</td>
<td>1.69s</td>
</tr>
</tbody>
</table>

Results: On-demand Analysis Cost. As discussed in § 5.1, we update the dynamic ICFG and CG with every newly-discovered indirect edge to ensure that tripwiring’s reachability analysis does not miss edges that precede target locations. However, should tripwiring analysis be re-run frequently (i.e., when the rate of new indirect edges is high), then directed fuzzing performance will quickly deteriorate due to the accumulated overhead. To measure the impact of tripwiring’s ad hoc analysis on directed fuzzing, we profile SieveFuzz’s 10x24-hour fuzzing campaigns to record (1) the total indirect edges discovered and (2) the mean instances that tripwiring is re-run. Our results are shown in Table 2.

Across all directed fuzzing trials, we observe a maximum of 87 new indirect edges—confirming that re-performing tripwiring re-analysis is, at worst, an infrequent-case event relative to the total test cases generated. However, we see that reanalysis is often invoked a fewer number of times than the total indirect edges discovered (e.g., jasper, listswf, tiffcp-1, tiffcp-2, and Tidy). In examining this, we find that individual test cases generally cover multiple indirect edges; and as tripwiring operates on the full coverage trace, its overall footprint on directed fuzzing overhead is minimal. Thus, in 24 hours of directed fuzzing, the cost of re-running tripwiring is at most less than 6 seconds of fuzzer runtime.

7.2 RQ2: Targeted Defect Discovery

To answer RQ2 and determine whether tripwiring translates to improved directed fuzzing effectiveness, we evaluate SieveFuzz, alongside minimization-directed AFLGo, precondition-directed BEACON, and undirected AFL++ in discovering ten reported bugs (Table 1)—a common real-world application of targeted testing—comparing their bug-triggering (1) consistency and (2) speed (Table 3).

Results: Tripwiring vs. Minimization-directed Fuzzing. In 10 trials per each of our ten ground truth bugs, tripwiring-directed SieveFuzz attains a 42% higher average bug exposure effectiveness over minimization-directed AFLGo (7.1 versus 5.00, respectively). Compared to AFLGo’s 6.73-hour mean exposure time, tripwiring accelerates directed fuzzing to find these bugs in just 3.49 hours—close to less than half the time of AFLGo—with a statistically-large mean improvement in bug exposure times (\(A_{12} = 0.79 > 0.71\)). Note that SieveFuzz is the only tool which finds tiffcp-2.

This performance can be attributed to the use of tripwiring which allows SieveFuzz to synthesize the complex preconditions to trigger the bug. While AFLGo is slightly more consistent on CROMU-00039, we see that SieveFuzz is able to find it 6.56x faster (3.81h vs 0.58h). On jasper, and mJS, we also see AFLGo perform slightly better; however, the difference is not statistically large (\(A_{12} < 0.71\)), meaning that SieveFuzz is on-par with AFLGo. Overall, tripwiring accelerates directed fuzzing for faster and more consistent defect discovery.

Results: Tripwiring vs. Precondition-directed Fuzzing. BEACON does not use LLVM’s sanitizer instrumentation. For a fair comparison between SieveFuzz and BEACON, we evaluate a variant of SieveFuzz that matches BEACON’s instrumentation style without sanitizer instrumentation. We exclude benchmark mJS in this experiment as BEACON’s instrumentation pass crashes during its compilation; as well as benchmarks KPRCA-00051 and tidy as their respective bugs are undetectable without sanitizer instrumentation.

As shown in Table 4, SieveFuzz achieves 2.19x faster bug discovery over BEACON (2.82 hours versus BEACON’s 6.17 hours). This performance improvement is statistically large (\(A_{12} = 0.71\)), indicating a substantial speedup of SieveFuzz over BEACON. Furthermore, SieveFuzz attains 1.60x more consistent bug discovery than BEACON (8.7 successful campaigns versus BEACON’s 5.4).

For three benchmarks (KPRCA-00038, tiffcp-1, and tiffcp-2), BEACON fails to uncover their corresponding bugs in any trials. To investigate why BEACON fails in these cases—and why SieveFuzz succeeds—we manually examined BEACON-instrumented binaries alongside their SieveFuzz-instrumented counterparts. Compared to SieveFuzz, BEACON’s path analysis over-prunes—eliminating reachable, bug-relevant program states in all three benchmarks—making it impossible for BEACON to synthesize the complex program states needed to reach and trigger these bugs.


Table 3: Bug exposure effectiveness; and mean exposure times and effect sizes for SieveFuzz versus minimization-directed AFLGo and undirected AFL++ across 10×24-hour fuzzing trials per our ten ground-truth bugs. Bold effect sizes reflect statistically-large (i.e., Vargha and Delaney $A_{12} > 0.71$) improvements in bug exposure times; while [n/a] denotes that the statistical test cannot be performed due to an insufficient number of exposing trials by SieveFuzz’s competitor.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bug Exposure Effectiveness (#trials)</th>
<th>Mean Exposure Time (hrs)</th>
<th>Relative Exposure Time Effect Size ($A_{12}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(higher is better)</td>
<td>(lower is better)</td>
<td>(higher is better)</td>
</tr>
<tr>
<td></td>
<td>AFL++ AFLGo SieveFuzz</td>
<td>AFLGo SieveFuzz</td>
<td>SieveFuzz / AFL++ SieveFuzz / AFLGo</td>
</tr>
<tr>
<td>CROMU-00039</td>
<td>9 8 5</td>
<td>1.25 3.81 0.58</td>
<td>0.68 0.72</td>
</tr>
<tr>
<td>KPRCA-00038</td>
<td>10 1 10</td>
<td>2.43 1.71 2.45</td>
<td>0.53 1.00</td>
</tr>
<tr>
<td>KPRCA-00051</td>
<td>7 9 10</td>
<td>9.90 7.86 0.19</td>
<td>1.00 1.00</td>
</tr>
<tr>
<td>gif2tga</td>
<td>2 0 4</td>
<td>9.86 n/a 6.83</td>
<td>n/a n/a</td>
</tr>
<tr>
<td>jasper</td>
<td>4 8 8</td>
<td>16.85 6.10 8.77</td>
<td>0.89 0.37</td>
</tr>
<tr>
<td>listswf</td>
<td>10 9 10</td>
<td>3.49 5.27 0.97</td>
<td>0.74 0.88</td>
</tr>
<tr>
<td>mJS</td>
<td>2 8 5</td>
<td>8.16 10.02 7.20</td>
<td>0.5 0.69</td>
</tr>
<tr>
<td>Tidy</td>
<td>4 5 7</td>
<td>19.10 14.28 6.20</td>
<td>1.00 0.67</td>
</tr>
<tr>
<td>tiffcp-1</td>
<td>4 2 10</td>
<td>4.20 4.80 1.36</td>
<td>0.75 1.00</td>
</tr>
<tr>
<td>tiffcp-2</td>
<td>0 0 2</td>
<td>n/a n/a 0.32</td>
<td>n/a n/a</td>
</tr>
<tr>
<td>Mean:</td>
<td>5.2 5 7.1</td>
<td>8.36 6.73 3.49</td>
<td>0.73 0.79</td>
</tr>
</tbody>
</table>

Table 4: Bug exposure effectiveness; and mean exposure times and effect sizes for SieveFuzz versus precondition-directed BEACON across 10×24-hour fuzzing trials per our eight ground-truth bugs. In this experiment, we run SieveFuzz with the same fuzz target configuration as BEACON. Bold effect sizes reflect statistically-large (i.e., Vargha and Delaney $A_{12} > 0.71$) improvements in bug exposure times; while [n/a] denotes that the statistical test cannot be performed due to an insufficient number of exposing trials by BEACON.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bug Exposure Effectiveness (#trials)</th>
<th>Mean Exposure Time (hrs)</th>
<th>Relative Exposure Time Effect Size ($A_{12}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(higher is better)</td>
<td>(lower is better)</td>
<td>(higher is better)</td>
</tr>
<tr>
<td></td>
<td>BEACON SieveFuzz</td>
<td>BEACON SieveFuzz</td>
<td>SieveFuzz / BEACON</td>
</tr>
<tr>
<td>CROMU-00039</td>
<td>10 10</td>
<td>0.67 0.43</td>
<td>0.68 n/a</td>
</tr>
<tr>
<td>KPRCA-00038</td>
<td>0 10</td>
<td>n/a 3.9</td>
<td>n/a n/a</td>
</tr>
<tr>
<td>gif2tga</td>
<td>10 10</td>
<td>2.15 0.17</td>
<td>0.59 n/a</td>
</tr>
<tr>
<td>jasper</td>
<td>10 6</td>
<td>8.51 7.8</td>
<td>0.58 1.00</td>
</tr>
<tr>
<td>listswf</td>
<td>8 10</td>
<td>13.36 0.51</td>
<td>1.00 n/a</td>
</tr>
<tr>
<td>tiffcp-1</td>
<td>0 9</td>
<td>n/a 0.30</td>
<td>n/a n/a</td>
</tr>
<tr>
<td>tiffcp-2</td>
<td>0 6</td>
<td>n/a 6.65</td>
<td>n/a n/a</td>
</tr>
<tr>
<td>Mean:</td>
<td>5.4 8.7</td>
<td>6.17 2.82</td>
<td>0.71 n/a</td>
</tr>
</tbody>
</table>

For KPRCA-0038, BEACON’s reachability analysis incorrectly marks a bug-relevant conditional branch as unreachable. This bug exists in the else branch in one of the program’s conditional statements; however, triggering the bug requires that the adjacent if branch is hit first. Because BEACON’s basic-block-level analysis deems the if branch irrelevant to the bug, it only permits the else branch to be taken—leaving BEACON unable to ever reach the bug-triggering state hidden in the if branch. SieveFuzz’s function-level analysis does not restrict either branch, enabling SieveFuzz to reach the correct sequence of branches needed to trigger the bug.

For tiffcp-1 and tiffcp-2, BEACON incorrectly prunes an indirectly-called function along the path to each bug. We observe that this function is passed as comparator function to a C standard library function, which then calls them. Because BEACON’s path analysis is only performed statically—unlike SieveFuzz’s which updates itself with new information as it is uncovered during fuzzing—BEACON will miss complex indirect control flows like this. We confirm that SieveFuzz successfully observes and incorporates the corresponding indirect edge in its dynamic control-flow graph.

On three of our remaining four benchmarks, we observe that SieveFuzz outperforms BEACON’s bug discovery. After profiling BEACON’s performance, we observe that the significant runtime overhead of its precondition-directed fuzzing is BEACON’s main bottleneck—resulting in an overall low throughput. Our results show that SieveFuzz averages a 23.5x higher fuzzing test case throughput than BEACON (Table 5). The only exception to this performance trend in bug discovery is jasper where BEACON finds it more consistently than SieveFuzz (10 vs 6 campaigns). From Table 2,
we infer that the bug lies in the least disjoint location among our target set with only 8% of the code regions being removed during tripwiring. Therefore, BEACON’s finer-grained analysis is a better fit for uncovering this bug. Though BEACON attains higher throughput on tiffcp-1 and tiffcp-2, its over-pruning of their respective state spaces prohibits BEACON from exposing either bug (Table 4). For this target, SieveFuzz’s lower throughput is due to it covering more of the bug-relevant paths that incur a higher runtime overhead from intersecting subroutines that set up bug-critical program state (e.g., key data structures). In general, SieveFuzz’s higher overall speed—and effectiveness—indicates that tripwiring is a less invasive directedness strategy than BEACON’s path precondition-directed approach, and thus is better suited for fuzzing disjoint target locations.

Results: Tripwiring vs. Undirected Fuzzing. As Table 3 shows, SieveFuzz’s advantages also hold over undirected fuzzing: with 140% faster bug exposure time than AFL++ (3.49 hours versus AFL++’s 8.36 hours) and a statistically-large mean effect size ($A_{12} = 0.73 > 0.71$). In CROMU-00039, AFL++ outperforms SieveFuzz; yet our statistical analysis shows that these differences are in fact insignificant, as comparison results in statistically-small effect sizes ($A_{12} = (0.68 < 0.71)$). Interestingly, on three benchmarks (KPRCA-00038, listswf, tiffcp-1), we see that undirected fuzzer AFL++ attains both a consistency and overall mean exposure time better than minimization-directed AFLGo—revealing that distance minimization often translates to worse-than-undirected fuzzing effectiveness in targeted testing. Thus, tripwiring enables SieveFuzz to surpass both minimization-directed AFLGo and undirected AFL++ while expanding directed fuzzing’s reach to use cases where current directed fuzzers fall short.

RQ2: By filtering out all target-irrelevant exploration, tripwiring achieves effective, high-speed directed fuzzing.

7.3 RQ3: Target Location Feasibility for Tripwiring

To help practitioners pinpoint locations well-suited to tripwiring-directed fuzzing, we believe that the percentage of search space removed by tripwiring represents the most promising metric. Figure 3 shows the amount of tripwiring-removed search space per target location (showing its disjointness) and the mean time taken to uncover the ground truth bug at this location by both AFLGo and SieveFuzz. We do not include BEACON in this analysis since we do not have enough timing data corresponding to bug discovery for this framework (only 4 out of the 10 ground truth bugs were successfully triggered by BEACON). We exclude the synthetic benchmarks (CROMU-00039, KPRCA-00038, and KPRCA-00051) to ensure no unintended noise is added to this experiment. Then, we use Spearman’s rank-order correlation coefficient [27] to identify if there exists a correlation in the performance difference of distance-minimization (AFLGo) against tripwiring (SieveFuzz) during bug discovery and the degree to which a target location is disjoint.

The Spearman’s rank-order shows a strong positive correlation (0.30) in the performance difference observed between distance-minimization and tripwiring and the amount of state space removed by tripwiring. I.e., the more disjoint a target site is—shown by an increasing percentage of code regions removed—the larger is the performance difference seen between a distance-minimization-based fuzzer and a tripwiring-directed fuzzer. Correspondingly, the more disjoint is a target location, the faster tripwiring becomes at uncovering the bug. We thus conclude that (1) quantifying the percentage of code that cannot reach target locations is a reliable metric for identifying disjoint target locations; and (2) for such locations, tripwiring (SieveFuzz) is a better choice for directed fuzzing than distance minimization (AFLGo).

RQ3: Tripwiring is an optimal directedness strategy for fuzzing target locations which exhibit disjointness.

8 DISCUSSION AND FUTURE WORK

Below we discuss several opportunities for enhancing and extending tripwiring in support of more powerful directed fuzzing.

Refinements in Path Analysis. In SieveFuzz’s approach to perform tripwiring, the dynamic resolution of indirect transfers is a source of incompleteness while identifying target-reachable paths. Specifically, if the target location is already reachable in the CG of the fuzz target, SieveFuzz will not identify alternative target-reachable paths via unresolved indirect calls that may exist in tripwired code regions. Consequently, there is a corner-case where these missed alternative paths are bug-triggering. While we did not observe this corner-case as a part of our evaluation, we do acknowledge that it may occur in other testing scenarios.

The root cause of the above mentioned scenario is the reliance of SieveFuzz on dynamically resolving indirect calls and its opportunistic movement towards performing tripwired fuzzing as soon as the target location becomes reachable. Therefore, to mitigate it, we envision several possible improvements, such as alternating between Exploration and Tripwired Fuzzing (§ 6.2) or a new phase specifically targeting resolving indirect calls. In addition, incorporating additional data sources will improve the resolution of tripwiring’s target-reachability analyses. NEUZZ [31] and Fuzz-Guard [43] show that machine learning can model the likelihood of exercising program paths; and as directed fuzzing is commonly deployed on well-fuzzed targets, we expect that it is practical to
Wüstholz et al. [38] gave mutation priority to inputs that were statistically better, leading to a higher throughput of test cases (as shown in Table 5). In comparison, tripwiring is much more lightweight than precondition-directed fuzzing, though it may be slightly more heavyweight than conventional distance minimization-based fuzzing.

**Path Prioritization.** While tripwiring aims to steer exploration down set of target-reaching paths, deciding which of these paths to prioritize is a universal challenge for all directed fuzzers. Wüstholz et al. [38] gave mutation priority to inputs that were statistically deemed to exercise paths not containing the target location. The intuition being that mutants generated from such inputs will exercise target-reachable paths. In future work, we will explore incorporating mutation priority enhancements into tripwiring to prioritize promising target-specific paths in an effort to reach and trigger bugs in the target location faster and more effectively.

### 9 RELATED WORK

This section discusses related literature on directed fuzzing, as well as orthogonal efforts to improve fuzzing performance.

**Directed Fuzzing.** Recent works extend fuzzing’s success at general-purpose software testing to more targeted testing scenarios (e.g., patch testing, bug reproduction). Most fuzzers of this type approach this as a distance minimization problem. AFLGo [4] performs simulated annealing optimization across call and control-flow graphs to find the shortest-length paths to the user-specified target locations. Hawkeye [5] expands AFLGo’s technique with algorithmic and analysis refinements, and additional coverage heuristics to avoid biasing unfruitful paths. ParmeSan [24] obtains its interesting target locations from sanitizer metadata (e.g., AddressSanitizer [30]). UAFuzz [23] and UAFL [36] mine target locations based on memory allocation patterns to maximize the chances of triggering heap corruptions. BEACON [13] performs directed fuzzing by identifying necessary preconditions for a given target location and then instrumenting the fuzz target to terminate paths that do not satisfy these preconditions. This approach is significantly more heavyweight than conventional distance minimization-based fuzzing in consistency, and efficiency.

### 10 CONCLUSION

Existing distance minimization-based directed fuzzers are universally bottlenecked by their employed search strategies. Tripwiring speeds-up directed fuzzing by culling irrelevant code—preempting and exiting unwanted paths to guide fuzzing only toward targeted locations. SieveFuzz demonstrates how tripwiring effectively supports directedness for security-critical targeted testing tasks like bug reproduction while interposing near-zero runtime overhead; and significantly outperforms conventional distance-minimization-based directed fuzzing in consistency, and efficiency.

### ACKNOWLEDGEMENTS

We thank the anonymous reviewers for precise and detailed feedback. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 850868), DARPA under HR001119S0089-AMP-FP-034 and N6600120C4031, AFRL under FA8655-20-1-7048, and SNSF under PCEGP2_186974. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

### REFERENCES


Listing 2 A code example to highlight the imprecision of context-insensitive ICFG analysis. In this example, determining that edge qed→bar is unreachable requires the additional consideration of the call graph.

```c
void foo() {
    bar();
    target();
}

void qed() {
    bar();
}

void target() {
    printf(argv[1]); // vulnerable
}
```

A EFFECT OF CONTEXT-INSENSITIVITY

Here, we showcase a concrete example how performing reachability analysis solely over the context-insensitive ICFG may over-approximate the target-reaching code regions. Consider the code snippet shown in Listing 2: a context-insensitive ICFG for this example contains the call edge from qed to bar, while the CG shows qed does not reach (i.e., is not an ancestor of) target. Therefore, if the tripwiring algorithm (Algorithm 1) does not consider the CG in performing the reachability check from qed to bar, it will incorrectly include qed in the allow-list.