From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification

Andrew Bao[1], Wenjia Zhao[2], Yanhao Wang[3], Yueqiang Cheng[4], Stephen McCamant[1], and Pen-Chung Yew[1]
[1]*University of Minnesota, Twin Cities* [2]*Xi'an Jiaotong University* [3]*Independent Researcher* [4]*MediaTek*

Abstract

Effective verification of the true positives from false positives is crucial for improving the usability of static analysis tools and bolstering software security. Directed greybox fuzzing (DGF), based on dynamic execution, can confirm real vulnerabilities and provide proof-of-concept exploits, offering a promising solution. However, existing DGF tools are ineffective in verifying static analysis results because they are unaware of the semantic information about individual alarms and the correlations among multiple alarms.

In this paper, we fill this gap and present Lyso, the first multi-target, multi-step guided fuzzer that leverages semantic information (i.e., program flows) and correlations (i.e., shared root causes) derived from static analysis. By concurrently handling multiple alarms and prioritizing seeds that cover these root causes, Lyso efficiently explores multiple alarms. For each alarm, Lyso breaks down the goal of reaching an alarm into a sequence of manageable steps. By progressively following these steps, Lyso refines its search to reach the final step, significantly improving its ability to trigger challenging alarms.

We compared Lyso to eight state-of-the-art (directed) fuzzers. Our evaluation demonstrates that Lyso outperforms existing approaches, achieving an average 12.17x speedup while finding the highest absolute number of bugs. Additionally, we applied Lyso to verify static analysis results for real-world programs, and it successfully discovered eighteen new vulnerabilities.

1 Introduction

The rapid growth of software complexity has led to an increased demand for effective methods to identify and mitigate security bugs. Static analysis tools (e.g., Coverity [1]) are widely used for automated software bug detection but often produce a high number of false positives [2–4], leading to a significant burden of manually verifying each reported issue. Distinguishing true positives from false positives can be challenging and time-consuming. While previous research has explored alarm clustering [5, 6] and ranking [7, 8] to address this issue, these methods cannot definitively identify true positive alarms, as they do not involve actual program execution for verification.

Directed greybox fuzzing (DGF) has emerged as a promising method to address this limitation by guiding the execution to the locations of static analysis alarms, crashes, and patches [9]. While DGF has been proven effective in various scenarios such as patch testing [9, 10] and crash reproduction [11–13], its effectiveness in verifying static analysis results is limited. Unlike a Git commit in patch testing or a stack backtrace in crash reproduction, static analysis results often present a complex "landscape". Our analysis of several widely used static analysis tools [1, 14–16] revealed several key characteristics. (1) *Large Number of Alarms*: These tools generate many alarms of various types. (2) *Multi-Alarm Correlations*: Alarms produced by program reasoning tools are often correlated, with multiple true alarms sharing a common root cause [8, 17]. (3) *Promising Paths*: Static analysis tools not only flag potential bug locations but also highlight the promising paths leading to them.

Although applying DGF to verify static analysis results often poses greater challenges than in patch testing or crash reproduction, the above unique characteristics of static analysis results can, in turn, be leveraged to guide DGF toward more effective verification. Based on this insight, we propose Lyso, a novel multi-target, multi-step guided fuzzer with two key aspects of directness:

- **Inter-target directness**. Lyso addresses the challenge of handling multiple targets[1] concurrently by leveraging correlations among targets with the same root cause, It categorizes targets into two types: *independent* or *interdependent*. Independent targets have minimal overlap in root causes and require individual verification. In contrast, interdependent targets are correlated, sharing common root causes and allowing for grouped verification. This ensures that

[1]In this paper, the terms "alarm" and "target" are used interchangeably.

progress on one target accelerates progress on others, effectively killing multiple birds with one stone.

- **Intra-target directness**. Lyso enhances the fuzzing by providing more precise, incremental guidance toward a specific target. Unlike traditional DGF tools that explore all feasible paths to the target, Lyso prioritizes vulnerable paths identified through static analysis. By breaking down the target into multiple steps and following them sequentially, Lyso significantly reduces the Time-to-Exposure (TTE) for challenging targets, enabling more effective directness.

Lyso identifies critical steps for each target, categorizing them as *independent* or *interdependent*. To achieve *inter-target directness* when handling multiple targets, Lyso performs inter-target exploration, concurrently tracking step coverage across multiple targets and prioritizing seeds that cover interdependent ones (i.e., overlap metric). For *intra-target directness*, Lyso performs intra-target exploration, tracking executions relative to the sub-steps and prioritizing seeds that maximize step coverage (i.e., depth metric) toward the target. Additionally, Lyso employs step-by-step guidance, progressively refining the search process by directing executions closer to the next unexplored step (i.e., distance metric).

Leveraging these three key characteristics observed in static analysis results, Lyso optimizes seed selection through multi-target, multi-step metrics, including *overlap*, *depth*, and *distance*. To improve step coverage of rarely-hit targets, Lyso introduces seed density power scheduling, dynamically allocating more mutations to seeds associated with these targets.

We evaluated Lyso on the state-of-the-art fuzzing benchmark Magma [18]. We compared Lyso with three code coverage fuzzers [19–21], and five directed fuzzers [9, 22–25]. Our extensive experiments demonstrate that Lyso outperforms existing solutions, achieving an average 12.17x speedup in bug verification. Furthermore, we applied Lyso to automatically verify static analysis results for four well-tested programs, where it successfully discovered eighteen new vulnerabilities. Of these, sixteen were confirmed by the developers, and seven have been assigned CVE IDs.

In summary, we make the following contributions:

- We propose a novel multi-target, multi-step guided fuzzer, Lyso, to enhance existing methods by leveraging program flows and alarm correlations to verify static analysis results.

- We design a fast, precise, and space-efficient step tracking system to provide step-by-step guidance for handling multiple targets, each with multiple steps.

- We introduce multi-target, multi-step metrics that include overlap, depth, and distance to enhance seed selection. We also propose a seed density power scheduling to enhance the exploration of rarely-hit alarms.

- We open-source Lyso[2] and provide comprehensive experiments to demonstrate its superiority over existing techniques in verifying bugs.

2 Background

This section provides an overview of the background on alarms generated by static analysis and the various approaches to Directed Greybox Fuzzing (DGF).

2.1 Alarms from Static Analysis

In static analysis, the generation of alarms is closely tied to understanding the flow of a program, which is crucial for identifying potential issues. These flows are typically represented using graph-based structures such as the Control Flow Graph (CFG) [26], Data Flow Graph (DFG) [27], or more advanced variants like the Program Dependency Graph (PDG) [28]. These representations capture the structural or logical relationships between program statements and their execution paths. To formalize this concept, let $G = (N, E)$ represent an abstract graph model of the program, where N denotes the set of program elements (i.e., statements), and E represents the edges that capture transitions between these elements, either through control or data flow. Within this model, a *program flow* is formally defined as a directed path π in G, originating from a source node n_1 and terminating at a target node n_m, as shown in the following expression.

$$\pi = (n_1 \to \cdots \to n_m) \text{ where } (n_k, n_{k+1}) \in E \ \forall k \in [1, m-1]$$

- $n_1 \in N$ is the starting node, representing the entry point, source, or the beginning of the flow.

- $n_m \in N$ is the target node, representing the exit, sink, or endpoint of the flow.

- $n_2, \cdots, n_{m-1} \in N$ are intermediate nodes along the path.

Building on the definition of program flow, an *alarm* in static analysis is a program flow π that satisfies specific conditions indicative of a potential vulnerability or issue. Specifically:

- An alarm corresponds to a subpath $\pi_a \subseteq \pi$, where:

  $$\pi_a = (s_a \to \cdots \to n_i \to \cdots \to t_a)$$

  with s_a being the entry point of the alarm (e.g., the source of the data) and t_a being the exit point (e.g., the sink, where a potential issue occurs).

- Each node $n_i \in \pi_a$ represents a program statement, contributing to the propagation of control or data within the alarm's context.

[2]https://doi.org/10.5281/zenodo.14714504

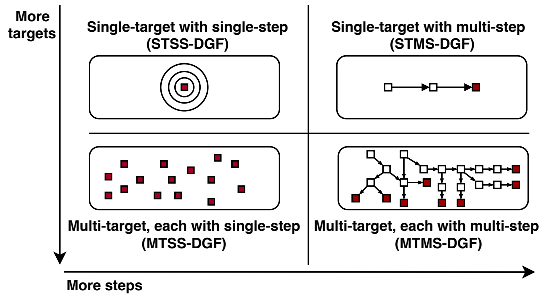

More targets
Single-target with single-step (STSS-DGF)
Single-target with multi-step (STMS-DGF)
Multi-target, each with single-step (MTSS-DGF)
Multi-target, each with multi-step (MTMS-DGF)
More steps

FIGURE 1: Varied scopes of DGF.

- The sequence of nodes $\{s_a, \cdots, n_i, \cdots, t_a\}$ is referred to as the ***steps*** of the alarm. These steps highlight the flow of execution or data relevant to the detected issue.

For instance, TIF009 (featured in the Magma [18] benchmark) in Figure 12, a null pointer dereference alarm detected by CodeQL [16], involves multiple steps. For simplicity and clarity, we have modified the original results excluding certain steps, and we present the following:

```
<L3, allocation (source)>
<L8, function call>
<L16, branch condition>
<L19, Setting the variable to null>
<L11, function call>
<L24, function call>
<L29, function call>
<L36, function call>
<L41, branch condition>
<L42, null pointer dereference (sink)>
```

2.2 Directed Greybox Fuzzing

We categorize existing research on Directed Greybox Fuzzing (DGF) into three domains based on whether their optimizations are designed for multi-target or multi-step scenarios. Figure 1 provides a visual representation of these categories.

Single-Target, Single-Step Directed Greybox Fuzzing (STSS-DGF). This category is primarily used for bug reproduction, where a specific buggy location has been identified, often through a git commit or manual inspection.

- AFLGO [9]: Introduces the concept of DGF by proposing a distance metric and a simulated-annealing-based power scheduling approach to guide execution toward a predefined target.
- WindRanger [29]: Enhances AFLGo's distance metric by considering deviated blocks.

Single-Target, Multi-Step Directed Greybox Fuzzing (STMS-DGF). This category is designed for crash reproduction, where reproducing a crash requires meeting specific preconditions (steps) in sequence. For instance, reproducing use-after-free (UAF) bugs typically involves triggering both the *free* and *use* operations in the correct order.

- UAFUZZ [13] and CAFL [12]: Refine the distance metric by recognizing that certain preconditions must be satisfied before a crash can occur.
- Hawkeye [11]: Improves the distance metric by addressing challenges related to indirect calls and incorporating call trace similarity for more accurate targeting.

Multi-Target, Single-Step Directed Greybox Fuzzing (MTSS-DGF). These techniques are designed to handle multiple targets and are commonly used to find bugs in sanitizer-instrumented code [23, 25], or verify static analysis results [24].

- Parmesan [25] and FishFuzz [23]: Propose a dynamic distance measurement that remains consistent across multiple targets.
- Titan [24]: Establishes a correlation synergy among multiple targets as feedback to handle multiple targets.

3 Motivation

This section introduces the limitations of current research and presents our proposed solutions. We then illustrate Lyso's functionality through concrete examples and provide an overview of its main workflow.

3.1 Prior Methods

Our study on widely used static analysis tools [1, 14–16] highlights three key characteristics of static analysis tools.

Large Number of Alarms. Static analysis tools tend to generate a high volume of alarms across various bug types. For example, when analyzing the *gpac* project using Coverity [1], the tool reported 896 alarms, flagging potential issues such as buffer overflows, integer overflows, and use-after-free (UAF) vulnerabilities. While these tools can identify true positives, they often generate many false positives. Existing DGF tools struggle to verify all alarms within a single fuzzing instance. For instance, AFLGo (STSS-DGF) [9] and CAFL (STMS-DGF) [12] use uniform distance metrics across all targets, diverting fuzzing efforts toward easily reachable, shallow targets while hindering progress toward deeper, more complex ones.

Interdependent Alarms. Alarms generated by static analysis tools are often not isolated, as multiple true alarms can share a common root cause. Our analysis of the Magma [18] benchmark reveals that 31% of true alarms are interdependent. This significant proportion underscores the need to account for their relationships to enhance verification effectiveness. However, existing MTSS-DGF tools [23, 25] do not account for these interdependencies. As a result, they fail to prioritize seeds that could potentially reach multiple interdependent alarms. While other MTSS-DGF tools like Titan [24] analyze

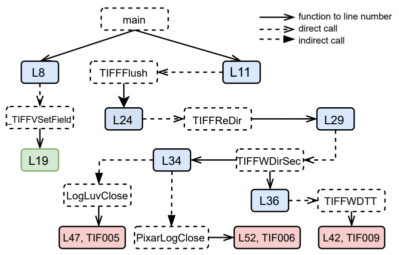


FIGURE 2: A motivating example based on the code presented in Figure 12. Nodes marked in blue represent the CFG node, the green node indicates the DFG node and the red node highlights the target.

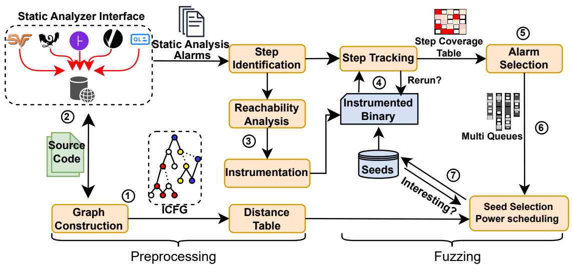


FIGURE 3: Workflow of Lyso.

multiple alarms that may share overlapping preconditions and prioritize seeds accordingly, Titan's static analysis struggles to scale with complex programs, and its taint analysis introduces significant overhead during fuzzing.

Promising Paths. For individual alarms, static analysis tools not only flag the bug locations (i.e., sinks) but also highlight the most promising paths leading to them. Both STSS-DGF and MTSS-DGF tools focus exclusively on reaching the sink while disregarding other steps. This approach introduces two key limitations. First, these tools attempt to explore all possible paths leading to the sink, even when static analysis highlights specific promising paths. By considering the steps along the promising paths, tools can avoid wasting effort on unpromising ones. Second, By prioritizing the shortest path to the sink [11], existing tools may miss longer yet more exploitable paths. Since static analysis does not always identify the shortest paths, incorporating these steps can improve the effectiveness of DGF by enabling deeper exploration of vulnerable execution paths

3.2 Motivating Example

Based on the three key characteristics of static analysis results we discussed in §3.1, it necessitates a new design for DGF to effectively verify these alarms. Previous methods have focused on enhancing DGF either for multi-target or multi-step scenarios, which limits their overall effectiveness in verifying static analysis results. To address this, we introduce Lyso, a multi-target, multi-step directed greybox fuzzing (MTMS-DGF) tool, which advances the state of the art in verifying static analysis results. To demonstrate Lyso's enhanced directness, we examine TIF005, TIF006, and TIF009, vulnerabilities marked in the Magma benchmark [18] (see Figure 2).

Lyso extracts critical steps for each alarm, categorizing them as Control Flow Graph (CFG) and Data Flow Graph (DFG) steps. To improve *inter-target directness*, Lyso identifies TIF005, TIF006, and TIF009 as interdependent targets, sharing a common function at the sink (see (§4.1). During inter-target exploration, Lyso prioritizes and mutates seeds that traverse these interdependent targets. As a result, Lyso directs significantly more execution effort toward TIF006 and TIF009, with an average of 132.40x than other DGF tools, which include Titan [24], FishFuzz [23], AFLGo [9], and SelectFuzz [22]. Notably, Lyso is the only tool that reaches and triggers TIF005.

To enhance *intra-target diretness*, as exemplified by TIF009, Lyso guides the execution through its critical steps. During intra-target exploration, after `L8` is covered, Lyso prioritizes seeds that covered `L8` and are closer to the next unexplored steps `L19`. This process repeats until all steps are covered. The step-by-step guidance accelerates Lyso towards reaching TIF009, with speedups of 14.40x, 11.16x, 19.20x, and 1.83x compared to the aforementioned DGF tools. Upon reaching the final step (sink) at `L42`, DFG step `L19` facilitates efficient alarm exploitation (Note: while passing through `L19` is not a prerequisite to reaching `L42`, it is a necessary step to trigger `L42`). As a result, Lyso outperforms the other DGF tools by 227.43x, 83.62x, 97.50x, and 16.57x in execution, leading to the successful triggering of TIF009.

3.3 Overview

Lyso's workflow, depicted in Figure 3, is divided into two main phases: preprocessing and fuzzing.

Preprocessing Phase. ❶ Program under test (PUT) is analyzed to construct an Inter-procedural Control Flow Graph (ICFG), which is used to compute a distance table supporting the distance metric during fuzzing. ❷ The PUT is passed through the static analyzer interface. In this work, Lyso uses CodeQL [16] to detect potential alarms. The static analysis results are then used to extract critical steps. ❸ To enable lightweight step tracking, reachability analysis is conducted to identify the basic blocks that can reach the critical steps. These identified blocks are then instrumented to facilitate step tracking during exeuction.

Fuzzing Phase. ❹ Lyso monitors the instrumented binary, tracking newly covered steps. Additionally, ❺ alarm selection prioritizes less-explored alarms with fewer associated seeds. ❻ Seed selection is refined based on multi-target multi-step metrics. For each selected seed, power scheduling adaptively allocates mutations based on the number of associated seeds to maximize effectiveness.

4 Preprocessing

In this section, we first describe how Lyso identifies the critical steps associated with each alarm and examines the interdependencies among multiple alarms based on critical steps (§4.1). Next, we outline the instrumentation strategies (§4.2) employed by Lyso to enable lightweight step tracking, facilitating step coverage feedback during fuzzing (§5.3.1). Finally, we detail the precomputation of a distance table for all basic block pairs (§4.3), enabling efficient dynamic calculation of distances between seeds and alarms during fuzzing (§5.1).

4.1 Critical Step Identification

Not all steps in an alarm contribute equally to guiding fuzzing toward the sink. To address this, we define critical steps as those that play a pivotal role in either data propagation or control flow. These critical steps are categorized into two types: *data steps* (i.e., *DFG steps*) and *control steps* (i.e., *CFG steps*).

DFG Step Identification. DFG steps capture the critical stages in the lifecycle of data associated with an alarm and are key to understanding how vulnerabilities propagate. Each alarm typically involves three steps: the source, the vulnerable operation, and the sink. The source is where the variable is initially defined, marking the beginning of its lifecycle. The vulnerable operation represents an intermediate point where the variable's state is either modified or accessed in a way that introduces the potential for unsafe behavior. Finally, the sink is where the variable is ultimately used, triggering the alarm.

The nature of the vulnerable operation can vary depending on the type of alarm. To systematically identify these operations, we leverage the rules defined in CodeQL [16], which provide a structured framework for detecting vulnerable patterns. By classifying the vulnerable operations for each alarm type (as detailed in Table 6 and summarized in Table 1), we ensure that DFG step identification aligns with the unique characteristics of different alarms. For example, in the case of TIF009, a null pointer dereference alarm described in §2.1, the steps at `L3`, `L19`, and `L42` correspond to the source, the vulnerable operation, and the sink, respectively. Additionally, for alarm types not listed in Table 1, Lyso applies generalized rules, identifying only source and sink locations.

CFG Step Identification. A step is classified as a CFG step if it corresponds to a function call site that directly influences the execution flow leading to the sink. Function call sites are prioritized over branch conditions (e.g., steps at `L16` and `L41` in §2.1 are excluded for CFG steps). This prioritization is based on the observation that function calls have a greater influence on overall control flow, whereas branch conditions may limit exploration to paths within individual functions. To maintain effectiveness, we empirically limit the number of function call sites to a maximum of five per alarm, selecting the five closest to the sink for analysis (the rationale for this threshold is discussed in §7.4). For example, steps at `L36`, `L29`, `L24`, `L11`, and `L8` are selected for TIF009. In cases with fewer than five function call sites, all available are selected.

TABLE 1: Critical steps identified based on the alarm type. Type delineates the various categories of alarms. '...' indicates the omitted call sites.

Type	Critical Steps Pattern
IO	$\cdots \rightarrow$ Alloc $\rightarrow \cdots \rightarrow$ Integer Assignment (arithmetic) $\rightarrow \cdots \rightarrow$ Memory Access
OOB	$\cdots \rightarrow$ Alloc $\rightarrow \cdots \rightarrow$ Variable Assignment (index) $\rightarrow \cdots \rightarrow$Memory Access
UAF	$\cdots \rightarrow$ Alloc $\rightarrow \cdots \rightarrow$ Free $\rightarrow \cdots \rightarrow$ Use
NPD	$\cdots \rightarrow$ Alloc $\rightarrow \cdots \rightarrow$ NULL Pointer Assignment $\rightarrow \cdots \rightarrow$ Dereference
UBI	$\cdots \rightarrow$ Alloc $\rightarrow \cdots \rightarrow$ Conditional Initialization $\rightarrow \cdots \rightarrow$ Use
DBZ	$\cdots \rightarrow$ Alloc $\rightarrow \cdots \rightarrow$ Zero Assignment $\rightarrow \cdots \rightarrow$ Divide

Alarm Interdependence Identification. To calculate the overlap metric between different alarms during fuzzing, it is crucial to first define how to measure the correlation between them. Fine-grained comparisons (e.g., comparing every step within the paths of two alarms) may fail to capture correlations due to minor variations. Instead, we focus on the DFG steps—source, vulnerable operation, and sink—which represent critical data propagation points along the triggering paths for various types of vulnerabilities. These steps efficiently capture the essential characteristics of program flow while leveraging the coarse-grained runtime context they provide. To further simplify the comparison, we associate the DFG steps with their respective functions since they naturally encapsulate richer runtime contexts for these steps. Two alarms are then considered interdependent if they share at least one function associated with any of their DFG steps. Based on this criterion, the overlap metric is quantified by counting the number of shared functions across the DFG steps of two alarms.

4.2 Step Instrumentation

To achieve lightweight step coverage tracking and feedback, Lyso selectively instruments only those basic blocks associated with critical steps (i.e., basic blocks that reach the critical steps), rather than instrumenting every block in the program. To determine which basic blocks are reachable for the critical steps, Lyso performs a control flow reachability analysis.

Algorithm 1 first constructs an interprocedural control flow graph (ICFG) for the given program P. Then, starting from the final critical step s_k of an alarm, Lyso performs a backward search using `reverse_dfs`. This process continues until the initial critical step s_1 is reached, ensuring that all relevant basic blocks are identified. Here, each critical step is uniquely identified with its corresponding basic block. Once all reachable basic blocks are identified, Lyso assigns a unique integer identifier to each block. These identifiers are statically instrumented into the program binary during the compilation. At runtime, this instrumentation enables Lyso to dynamically track the execution of these basic blocks and generate step coverage feedback, as described in §5.3.1.

Algorithm 1 Static Reachability Inference

```
1:  Input: Program $P$; alarms $(alarm_1, \dots, alarm_m)$
2:  Output: Reachable basic blocks $res$
3:  function REACHABLE_BB_INFERENCE($P$)
4:      $G \leftarrow build\_icfg(P)$                          ▷ build ICFG
5:      $res \leftarrow \{\}$
6:      for $i = 1$ to $m$ do                                  ▷ m: total alarms
7:          $[s_1, \dots, s_k] \leftarrow critical\_steps\_identify(P, alarm_i)$
8:          $S \leftarrow reverse\_dfs(G, [s_1, \dots, s_k])$
9:          $res \leftarrow res \cup S$
10:     end for
11:     return $res$
12: end function
```

4.3 Distance Table Precomputation

To enable efficient dynamic calculation of distances between seeds and alarms during fuzzing, a distance table for all basic block pairs is precomputed before the fuzzing phase. The distance measurement involves two main stages. First, similar to step instrumentation, an ICFG is constructed. This graph incorporates indirect calls through type-based analysis. Second, a precomputed distance table is generated, recording the shortest distances between all pairs of basic blocks in the ICFG. Additional details about these two stages are provided in §A.2. To further improve efficiency, the distance table is optimized with a hash map, allowing constant-time ($O(1)$) lookups and minimizing query overhead. During fuzzing, the optimized table is used to calculate the distance metric (§5.1).

5 Fuzzing

This section provides a comprehensive overview of the fuzzing methodology. We begin by defining the multi-target and multi-step metrics (§5.1). Next, we introduce their role in seed selection during the fuzzing stage (§5.2.1). We then describe the power scheduling mechanism (§5.2.2). Finally, we elaborate on how Lyso implements these metrics through its step tracking system (§5.3).

5.1 Multi-target Multi-step Metrics

We formally define the key metrics used in Lyso: *overlap*, *depth*, and *distance*, which are essential for seed selection to achieve effective guidance for multiple alarms, each with multiple steps.

Definition 1. *(Overlap) The overlap of a seed refers to the number of interdependent alarms covered for at least one step by the execution of the seed during fuzzing.*

Definition 2. *(Depth) The depth of a seed is the maximum number of consecutive steps covered for any alarm by the execution of the seed during fuzzing.*

Before defining the *distance* metric, we introduce the following key concepts: (1) **Last Matched Point**: When comparing a seed's execution trace with the sequence of steps associated with an alarm, the last matched point is the basic block in the execution trace that corresponds to the most recently matched step in the sequence. (2) **Suffix Trace**: The execution trace is divided into two segments based on the last matched point: the prefix trace and the suffix trace. The prefix trace includes the portion of the trace before the last matched point, while the suffix trace starts after the last matched point to the end. (3) **Next Unexplored Step**: The first step in the alarm step sequence that has not yet been matched with any part of the execution trace.

Definition 3. *(Distance) The distance of a seed denoted as $D(ST(\xi(s), k), bb_t)$, is defined as the shortest distance D from the suffix trace $ST(\xi(s), k)$ to the next unexplored step bb_t (basic block). $\xi(s)$ represents the sequence of basic blocks traversed during the execution of seed s, and k is the last matched point.*

The distance $D(ST(\xi(s), k), bb_t)$ is calculated as follows:

$$D(ST(\xi(s), k), bb_t) = \min_{bb_s \in ST(\xi(s), k)} d(bb_s, bb_t) \qquad (1)$$

Here, $d(bb_s, bb_t)$ are inter-procedural basic block distance defined in Equation 7, which is stored in the precomputed distance table (§4.3).

5.2 Fuzzing Loop

Lyso's fuzzing logic is built upon AFL's [19] fuzzing loop, incorporating seed selection (§5.2.1) and power scheduling (§5.2.2) strategies to enable multi-target and multi-step guidance. Algorithm 2 presents Lyso's fuzzing loop, which operates in two distinct phases: inter-target exploration and intra-target exploration.

- Inter-target exploration: In this phase, Lyso focuses on exploring multiple alarms. Seeds are selected based on their ability to uncover new steps or maximize coverage of interdependent targets (i.e., overlap).
- Intra-target exploration: In this phase, Lyso narrows its focus to a specific alarm. Seeds are prioritized based on greater step coverage (i.e., depth) for the alarm or closer to the next unexplored step (i.e., distance).

The transitions between these phases are dynamically managed based on two key mechanisms: (1) *Timeout-based switching*: The `timeout` variable defines a fixed time window for each phase. Both inter-target and intra-target exploration phases are executed within the same window, ensuring balanced execution time. The current window size is set to 10 minutes. (2) *Favored seed-based switching*: The `is_favored` function determines phase switching based on code coverage. When no seed is favored in terms of code coverage, Lyso shifts from inter-target to intra-target exploration.

Algorithm 2 Lyso's Fuzzing Loop Algorithm

```
1:  while True do
2:      while $\neg$ $timeout \lor is\_favored(queue)$ do        ▷ Inter-target exploration
3:          $seed \leftarrow cull\_queue(queue, code\_top, step\_top)$
4:          $energy \leftarrow power\_schedule(seed)$
5:          for $i = 0$ to $energy$ do
6:              $child \leftarrow mutate(seed)$
7:              $run(P, child)$
8:              if $save\_if\_interesting(child)$ then
9:                  $queue \leftarrow queue \cup \{child\}$
10:             end if
11:         end for
12:     end while
13:     $alarm \leftarrow choose\_alarm(heat\_table)$
14:     while $\neg timeout$ do        ▷ Intra-target exploration
15:         $p\_queue \leftarrow rank\_queue(queue, alarm)$
16:         $seed \leftarrow next\_seed(p\_queue)$
17:         $energy \leftarrow density\_power\_schedule(seed)$
18:         for $i = 0$ to $energy$ do
19:             $child \leftarrow mutate(seed)$
20:             $run(P, child)$
21:             if $save\_if\_interesting(child)$ then
22:                 $queue \leftarrow queue \cup \{child\}$
23:             end if
24:         end for
25:     end while
26: end while
```

5.2.1 Seed Selection

Lyso's seed selection strategy is tailored to each exploration phase. In the inter-target exploration phase, the seed selection process is handled by the `cull_queue` function, which evaluates two types of feedback: (1) *Code Coverage Feedback* (`code_top`): identifies seeds that increase code coverage. (2) *Step Coverage Feedback* (`step_top`): identifies seeds that either discover new steps or cover multiple interdependent alarms. Seeds contributing to either feedback are prioritized to maximize overall code coverage and step coverage across multiple alarms.

In the Intra-target exploration phase, Lyso focuses on a specific alarm. The process involves two key stages: (1) *Alarm Selection*: The `choose_alarm` function selects an alarm based on the `heat_table`, which tracks the number of seeds associated with each alarm (i.e., execution of the seed cover at least one step for the alarm). Alarms with fewer associated seeds are prioritized to balance exploration and avoid neglecting less-explored alarms. (2) *Seed Prioritization*: For the selected alarm, seeds are ranked using the `rank_queue` function. Seeds are sorted into a priority queue (`p_queue`) based on the following criteria: (i) Depth: Seeds that provide deeper coverage for the selected alarm are ranked higher. (ii) Distance: For seeds with the same *depth*, those with a shorter *distance* to the next unexplored step are prioritized (§5.3.2).

In both phases, Lyso leverages the `save_if_interesting` function (described in §5.3.1) to identify and retain interesting seeds. Seeds are evaluated based on code, or step coverage with their *overlap* and *depth* values serving as key metrics.

5.2.2 Power Scheduling

Before introducing our power scheduling strategies, we define the concept of *Density*.

Definition 4. *(Density) For a given alarm $\mathcal{A}$, the density of a seed* s *is defined as the ratio of the total number of seeds whose execution paths cover at least one step associated with $\mathcal{A}$ (denoted as $\mathcal{M}_{\mathcal{A}}$) to the total number of seeds whose execution paths cover at least one step associated with any alarm (denoted as $\mathcal{M}_{\mathcal{N}}$).*

$$\rho_s = \frac{\mathcal{M}_{\mathcal{A}}}{\mathcal{M}_{\mathcal{N}}} \tag{2}$$

In the inter-target exploration phase, the `power_schedule` function utilizes AFL's default metrics to determine the number of mutations allocated to each seed. During intra-target exploitation, Lyso introduces a seed density-based power scheduling strategy to allocate mutation efforts. This is implemented through the `density_power_schedule` function, defined as:

$$\hat{p}(s) = \rho_s \cdot \mathcal{N} \tag{3}$$

where $\mathcal{N}$ represents the total number of alarms.

The use of seed density ensures a balanced distribution of mutation efforts across alarms. Alarms with a higher density (ρ_s), the mutation budget should be distributed more evenly for its associated seed. This prevents excessive focus on individual seeds, promoting broader exploration. Conversely, alarms with lower density have fewer associated seeds. These seeds should receive more mutations to enable deeper exploitation, increasing the chances of uncovering bugs in these less-explored alarms.

5.3 Step Tracking

Building on the preprocessing phase in §4, Lyso monitors the execution against each critical step of alarms. It gathers step coverage feedback, measuring the *overlap* and *depth* value of each seed (§5.3.1). Lyso then guides the execution toward the next unexplored step, leveraging the *distance* metric (Equation 1).

5.3.1 Step Coverage Feedback

Lyso determines whether an input achieves new step coverage by comparing its execution trace with the sequence of steps

Algorithm 3 Interesting Seeds Determination

```
1:  Input: Seed $s$
2:  Output: True or False
3:  function SAVE_IF_INTERESTING($s$)
4:      $hnb \leftarrow has\_new\_bits()$                      ▷ code coverage feedback
5:      $hnsc \leftarrow has\_new\_step\_cov(cur\_step, vir\_step)$   ▷ step coverage feedback
6:      if $hnb$ || $hnsc$ then
7:          $s.overlap \leftarrow calculate\_overlap(cur\_step)$
8:          $s.depth \leftarrow calculate\_depth(cur\_step)$
9:      end if
10:     return $hnb$ || $hnsc$
11: end function
```

associated with each alarm in the Step Table. It identifies previously uncovered steps by maintaining the Step Table and updating the Step Coverage Table. When the execution trace matches the steps in the Step Table, Lyso updates the Step Coverage Table to reflect the newly covered steps.

- **Step Table** is a two-dimensional array where each row corresponds to a specific alarm, and each column within a row represents the index of a step associated with that alarm. Each element in the table is an integer value corresponding to the basic block assigned to the step during instrumentation (as detailed in §4.2). This design ensures a consistent and precise mapping between the steps in the instrumented binary program and their representation in the alarm.

- **Step Coverage Table** is structurally identical to the Step Table, but each element is a binary value indicating whether the corresponding step in the Step Table has been covered.

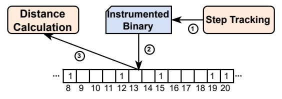
Lyso extends AFL's `save_if_interesting` function, which evaluates code coverage feedback, by incorporating step coverage feedback to identify inputs that reveal new step coverage. As shown in Algorithm 3, the Step Coverage Table `cur_step` and `vir_step` track step coverage for the current execution and accumulated executions from previous seeds, respectively. The `has_new_step_cov` function detects inputs introducing new step coverage by comparing these tables. For each seed, Lyso records *overlap* and *depth* values, which are later used for seed selection (§5.2.1). Specifically, the `calculate_overlap` function evaluates the *overlap* among multiple alarms for the current step coverage `cur_step`. Similarly, the `calculate_depth` function computes the *depth* for each alarm.

5.3.2 Step-by-Step Guidance

To further steer Lyso toward the next unexplored step, we introduce a step-by-step guidance approach, implemented in the `rank_queue` function during the intra-target exploration phase. This approach comprises two key stages: (1) Prefix-Trace Matching: Identify the last matched point in the execution trace to pinpoint the suffix trace and the next unexplored step. (2) Suffix-Trace Guiding: Collect suffix trace and use the *distance* metric defined in Equation 1.

Trace: 9, 14, 11, 20, 12, 20, 15, 20, 19, 8
Step Table: 9, 20, 39, 3

(A) Prefix-Trace Matching



(B) Execution Replay

FIGURE 4: Figure 4a shows the prefix-trace matching. Lyso identifies the first occurrence of basic block 20 as the last matched point. Through the execution replay as shown in Figure 4b, Lyso collects only the suffix trace consisting of a set of blocks {8, 12, 15, 19, 20} encoded in the bitmap.

Prefix-Trace Matching. Lyso employs a runtime mechanism called *in-execution tracking* to efficiently identify the last matched point in an execution trace. This mechanism dynamically compares each executing basic block with the sequence of steps associated with the alarm in the step table. When a basic block matches the current unexplored step, Lyso marks the step as covered and shifts the comparison to the next step for subsequent basic blocks. If no match is found, the comparison continues with the same step until a match occurs. For example, consider a step table containing one alarm with four steps, as illustrated in Figure 4a. Lyso begins with basic block `9`, matches it to step `9`, and proceeds to step `20`. It then encounters non-matching blocks `14` and `11`, which are skipped. Upon reaching block `20`, it matches step `20` and advances to step `39`. This process continues until the trace ends, identifying the first occurrence of block `20` in the trace as the last matched point. The portion of the trace after this point is designated as the suffix trace, with step `39` marked as the next unexplored step.

Suffix-Trace Guiding. Lyso leverages the suffix trace to evaluate the seed's distance to the next unexplored step. To collect the suffix trace while minimizing memory overhead, Lyso uses a bitmap—a fixed-size array—to record the current execution trace. The index of the array corresponds to the value (i.e., an integer) of instrumented basic blocks, with its value indicating whether the block has been executed. While the bitmap efficiently captures the trace coverage by indicating whether each basic block was executed, it does not preserve the order of the execution trace. As a result, without the sequential execution trace, in-execution tracking in a single run cannot collect the suffix trace from the entire trace in the bitmap using only the last matched point.

To overcome this limitation, Lyso performs an execution replay. During the replay, the prefix trace is explicitly excluded from the bitmap to ensure that only the suffix trace is recorded. Using the example from Figure 4b: First, the step tracking system signals the instrumented binary to delay recording the trace into the bitmap until the last matched point is reached (e.g., block `20`). Second, During execution replay, the binary records the suffix trace starting after block `20` and stores it in the bitmap. Third, Lyso then calculates the shortest distance from the recorded suffix trace to the next unexplored step.

6 Implementation

We implemented Lyso on top of AFL v2.57b, adding approximately 5,000 lines of code. This includes 1,644 lines of C/C++ for constructing an ICFG that facilitates indirect-call and static reachability analysis based on LLVM 11.0.0 [30]. Additionally, we added 414 lines of C/C++ for code coverage instrumentation and 159 lines of C code for step instrumentation and trace tracking. Furthermore, approximately 1,500 lines of C were added to afl-fuzz.c to support seed selection and power scheduling. Our development also includes 694 lines of Python, 137 lines of C++ for distance calculation, and 530 lines of Python to convert static analysis reports into sequences of steps.

7 Evaluation

In this section, we evaluated Lyso to answer the following research questions:
RQ1: How does Lyso verify bugs across multiple targets compared to other fuzzers? (§7.2)
RQ2: How effective is the multi-target, multi-step guidance in both multi-target and single-target scenarios? (§7.3)
RQ3: How do configurations (steps identification) and components (seed selection and power scheduling) affect Lyso's performance? (§7.4)
RQ4: What is Lyso's runtime overhead? (§A.4)
RQ5: Can Lyso detect new vulnerabilities in real-world programs? (§7.5)

7.1 Evaluation Setup

We followed the revised best practices for fuzzing evaluation [31, 32] and presented our evaluation setup.

Benchmarks and Fuzzers. We used the state-of-the-art Magma [18] benchmark and its provided initial seeds. This benchmark provides real-world projects containing multiple bugs, allowing us to assess a fuzzer's ability to verify diverse bug types. This setup closely mirrors the scenario in static analysis reports verification, where multiple true alarms need to be verified by fuzzer. We also compared Lyso against eight state-of-the-art fuzzers, as listed in Table 7, and detailed configuration information can be found in §A.3.

Evaluation metrics. We evaluated fuzzer performance using several key metrics. First, Time-to-Exposure (TTE) measures the time from the start of fuzzing to the first detection of a given bug. Second, Time-to-Reach (TTR) captures the time from the start of fuzzing to when executions first touch the bug location (i.e. the sink) without necessarily triggering it. Third, Success Rate (SR) evaluates consistency, defined as the ratio of successful bug reaches or triggers to the total number of fuzzing attempts. Finally, we recorded the total number of unique bugs reached or triggered by the fuzzer at least once across all runs.

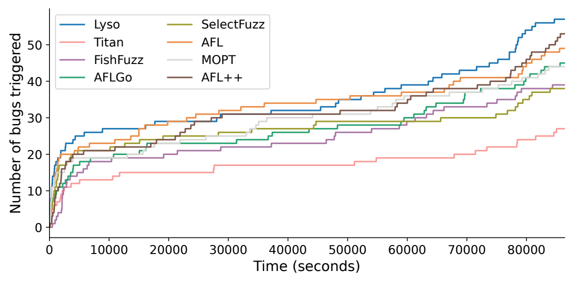


FIGURE 5: Number of unique bugs triggered over time in Table 2.

Bug selection. We set buggy locations marked by the `MAGMA_LOG` macro as targets for other DGF tools. For Lyso, which supports multiple steps, we employed CodeQL [16] dataflow analysis to identify the source-sink trace related to the marco. To ensure a fair comparison, we excluded bugs that were triggered within 100 seconds or not triggered within 24 hours by all fuzzers, as such cases may not accurately reflect a fuzzer's ability to verify bugs. Thus, these bugs are not listed in Table 2 and Table 5.

Environment. All experiments were run on the two computers, each equipped with Intel(R) Core(TM) i7-12700K with 20 cores and 96 GB of RAM, running under Ubuntu 22.04.3 LTS.

7.2 Enhancing Bug Verification across Multiple Targets

We compared Lyso against state-of-the-art fuzzers, as summarized in Table 7. Each fuzzer was executed 10 times for 24 hours (i.e., 86,400 seconds) per run, and the average TTE and SR were calculated. The complete evaluation results are summarized in Table 2, with its corresponding p-values from the Mann-Whitney U test listed in Table 10.

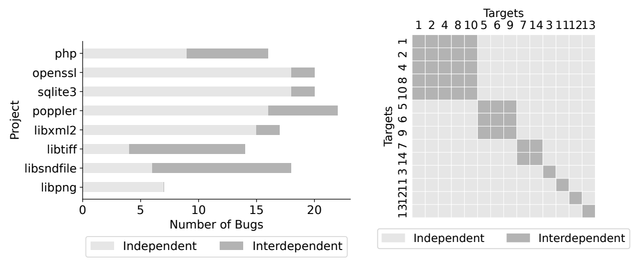
Time-to-Exposure. Based on the p-values presented in Table 10, our analysis indicates that Lyso significantly reduces Time-to-Exppsure (TTE) across the majority of target programs. Overall, Lyso outperforms state-of-the-art non-directed fuzzers (AFL, MOPT, AFL++) by an average of 8.89x. Compared to the STSS-DGF tool AFLGo, Lyso achieves a 13.89x speedup. Additionally, Lyso demonstrates an average improvement of 21.09x over MTSS-DGF tools Titan and FishFuzz. Both Lyso and SelectFuzz leverage CFG and DFG for directed fuzzing. In our evaluation, Lyso outperforms SelectFuzz, achieving a speedup of 2.43x. Figure 5 visually illustrates the TTE for unique bugs triggered at least once across 10 runs. Compared to other fuzzers, Lyso consistently triggers more bugs over time.

TABLE 2: We evaluated TTE and SR for targets in Magma. *Time* represents the average TTE in ten runs. *SR* represents success rate. *T.O* denotes the fuzzer fails to trigger the target within 24 hours. *Ratio* measures the improvement ratio achieved by Lyso compared to other fuzzers. ∅ indicates deployment was not feasible. The two best TTE results for a case are highlighted in light green and dark green.

Program & Bug ID		**Lyso**		**Titan**		**FishFuzz**		**AFLGo**		**SelectFuzz**		**AFL**		**MOPT**		**AFL++**	
		Time	*SR*	*Ratio*	*SR*	*Ratio*	*SR*	*Ratio*	*SR*	*Ratio*	*SR*	*Ratio*	*SR*	*Ratio*	*SR*	*Ratio*	*SR*
libpng_fuzzer	PNG001	*T.O.*	0/10	∼1.00x	0/10	∼1.00x	0/10	∼0.81x	3/10	∼1.00x	0/10	∼0.92x	1/10	∼0.92x	1/10	∼1.00x	0/10
	PNG006	>78257	1/10	∼1.10x	0/10	∼1.10x	0/10	∼1.10x	0/10	∼1.10x	0/10	∼1.10x	0/10	∼1.10x	0/10	<0.0007x	10/10
	PNG007	2644	10/10	4.44x	10/10	2.39x	10/10	1.91x	10/10	0.19x	10/10	1.83x	10/10	>13.09x	9/10	>10.94x	9/10
sndfile_fuzzer	SND001	36	10/10	9.03x	10/10	55.91x	10/10	3.39x	10/10	2.28x	10/10	2.94x	10/10	4.58x	10/10	11.19x	10/10
	SND006	309	10/10	11.71x	10/10	7.01x	10/10	0.85x	10/10	2.30x	10/10	0.91x	10/10	0.65x	10/10	11.52x	10/10
	SND007	38	10/10	9.79x	10/10	61.53x	10/10	3.82x	10/10	2.84x	10/10	3.45x	10/10	5.42x	10/10	21.30x	10/10
	SND017	26	10/10	17.62x	10/10	35.92x	10/10	32.54x	10/10	29.23x	10/10	13.35x	10/10	1.00x	10/10	14.42x	10/10
	SND020	46	10/10	53.76x	10/10	25.28x	10/10	24.98x	10/10	25.74x	10/10	19.41x	10/10	7.22x	10/10	22.61x	10/10
	SND024	39	10/10	9.46x	10/10	59.44x	10/10	3.64x	10/10	2.73x	10/10	3.33x	10/10	5.10x	10/10	19.46x	10/10
tiff_fuzzer	TIF002	28035	10/10	>2.80x	1/10	>1.66x	9/10	>1.72x	7/10	1.21x	10/10	>0.71x	9/10	>1.30x	9/10	>1.74x	6/10
	TIF008	>47993	6/10	∼1.63x	1/10	∼1.32x	6/10	∼0.91x	7/10	∼1.80x	0/10	∼0.59x	8/10	∼1.40x	3/10	∼1.77x	2/10
	TIF012	164	10/10	11.54x	10/10	8.70x	10/10	1.18x	10/10	0.81x	10/10	0.66x	10/10	2.19x	10/10	5.49x	10/10
	TIF014	429	10/10	2.90x	10/10	4.90x	10/10	6.23x	10/10	11.06x	10/10	0.62x	10/10	0.47x	10/10	4.72x	10/10
tiffcp	TIF002	>52237	9/10	∼1.65x	0/10	∼1.65x	0/10	∼1.20x	6/10	∼0.85x	7/10	∼1.27x	4/10	∼1.65x	0/10	∼1.37x	2/10
	TIF005	>80902	2/10	∼1.07x	0/10	∼1.07x	0/10	∼1.07x	0/10	∼1.07x	0/10	∼1.07x	0/10	∼1.07x	0/10	<0.02x	10/10
	TIF006	37145	10/10	>1.38x	6/10	>1.45x	8/10	>0.84x	9/10	0.51x	10/10	0.66x	10/10	0.90x	9/10	0.07x	10/10
	TIF008	>84667	1/10	∼1.02x	0/10	∼1.02x	0/10	∼1.02x	0/10	∼1.02x	0/10	∼0.81x	3/10	∼1.02x	0/10	∼0.87x	2/10
	TIF009	1900	10/10	>14.48x	7/10	11.28x	10/10	>19.30x	9/10	1.90x	10/10	7.89x	10/10	>13.82x	9/10	>14.28x	9/10
	TIF012	512	10/10	4.17x	10/10	5.56x	10/10	1.07x	10/10	1.25x	10/10	1.79x	10/10	1.65x	10/10	1.75x	10/10
	TIF014	3928	10/10	0.45x	10/10	0.74x	10/10	0.38x	10/10	0.73x	10/10	0.31x	10/10	0.26x	10/10	0.49x	10/10
lua	LUA004	5816	10/10	>11.67x	3/10	1.79x	10/10	2.78x	10/10	1.75x	10/10	1.15x	10/10	2.24x	10/10	3.91x	10/10
libxml2_fuzzer	XML001	>56243	5/10	∼1.54x	0/10	∼1.39x	1/10	∼1.48x	2/10	∼1.41x	1/10	∼1.47x	1/10	∼1.03x	6/10	<0.10x	10/10
	XML002	>75190	2/10	∼1.15x	0/10	∼0.80x	5/10	∼1.15x	0/10	∼1.15x	0/10	∼1.06x	2/10	∼1.15x	0/10	∼1.15x	0/10
	XML003	17662	10/10	>3.10x	4/10	1.08x	10/10	0.61x	10/10	0.71x	10/10	0.91x	10/10	>2.50x	8/10	0.22x	10/10
	XML009	1682	10/10	3.00x	10/10	2.74x	10/10	2.11x	10/10	0.95x	10/10	0.19x	10/10	0.73x	10/10	0.62x	10/10
	XML012	>78619	2/10	∼1.10x	0/10	∼1.10x	0/10	∼0.92x	4/10	∼1.03x	2/10	∼0.75x	7/10	∼0.73x	4/10	∼1.03x	2/10
xmllint	XML001	>48474	8/10	∼1.78x	0/10	∼1.36x	3/10	∼1.22x	7/10	∼0.92x	8/10	<0.47x	10/10	<0.72x	10/10	∼1.25x	7/10
	XML002	>78131	1/10	∼1.11x	0/10	∼0.96x	2/10	∼1.11x	0/10	∼1.11x	0/10	∼1.11x	0/10	∼1.05x	1/10	∼1.06x	1/10
	XML009	8879	10/10	1.19x	10/10	0.74x	10/10	0.45x	10/10	0.38x	10/10	0.05x	10/10	0.09x	10/10	0.19x	10/10
pdf_fuzzer	PDF002	>68736	2/10	∼1.26x	0/10	∼1.26x	0/10	∼1.26x	0/10	∼1.26x	0/10	∼1.26x	0/10	∼1.26x	0/10	∼1.26x	0/10
	PDF004	>77117	2/10	∼1.12x	0/10	∼1.12x	0/10	∼1.12x	0/10	∼1.12x	0/10	∼1.08x	1/10	∼1.12x	0/10	∼1.12x	0/10
	PDF010	758	10/10	1.02x	10/10	0.64x	10/10	0.32x	10/10	1.91x	10/10	2.38x	10/10	3.32x	10/10	3.61x	10/10
	PDF011	>81491	2/10	∼0.90x	2/10	∼1.06x	0/10	∼1.06x	0/10	∼0.94x	2/10	∼1.06x	0/10	∼0.95x	3/10	∼0.95x	1/10
	PDF018	334	10/10	>258.68x	0/10	>258.68x	0/10	>233.66x	1/10	4.07x	10/10	4.66x	10/10	8.16x	10/10	72.78x	10/10
	PDF019	>77123	2/10	∼1.12x	0/10	∼1.12x	0/10	∼1.07x	1/10	∼1.12x	0/10	∼1.03x	2/10	∼1.12x	0/10	∼1.05x	2/10
	PDF021	*T.O.*	0/10	∼1.00x	0/10	∼1.00x	0/10	∼1.00x	0/10	∼1.00x	0/10	∼1.00x	0/10	∼1.00x	0/10	∼0.67x	5/10
pdftoppm	PDF002	>65791	5/10	∼1.29x	1/10	∼1.31x	0/10	∼1.31x	0/10	∼1.31x	0/10	∼1.31x	0/10	∼1.14x	2/10	∼1.31x	0/10
	PDF006	*T.O.*	0/10	∼1.00x	0/10	∼1.00x	0/10	∼1.00x	0/10	∼1.00x	0/10	∼1.00x	0/10	∼1.00x	0/10	∼0.74x	3/10
	PDF010	1076	10/10	0.09x	10/10	1.97x	10/10	2.04x	10/10	0.84x	10/10	0.19x	10/10	0.12x	10/10	1.82x	10/10
	PDF011	>71624	4/10	∼1.19x	1/10	∼1.21x	0/10	∼1.21x	0/10	∼1.21x	0/10	∼1.10x	1/10	∼1.01x	3/10	∼1.17x	1/10
	PDF018	481	10/10	>179.63x	0/10	>179.63x	0/10	>144.83x	2/10	2.38x	10/10	2.86x	10/10	5.37x	10/10	49.28x	10/10
	PDF019	>63523	6/10	∼1.36x	0/10	∼1.36x	0/10	∼1.36x	0/10	∼1.36x	0/10	∼1.36x	0/10	∼1.32x	1/10	∼1.34x	1/10
pdfimages	PDF002	28663	10/10	>2.84x	3/10	>3.01x	0/10	>3.01x	0/10	>3.01x	0/10	>3.01x	0/10	>3.01x	0/10	>3.01x	0/10
	PDF003	16102	10/10	1.71x	10/10	2.31x	10/10	0.94x	10/10	1.76x	10/10	0.68x	10/10	1.00x	10/10	1.31x	10/10
	PDF008	>58979	6/10	∼1.46x	0/10	∼1.46x	0/10	∼1.46x	0/10	∼1.46x	0/10	∼1.46x	0/10	∼1.46x	0/10	∼1.45x	1/10
	PDF011	>73750	3/10	∼0.97x	3/10	∼1.17x	0/10	∼1.17x	0/10	∼1.06x	1/10	∼1.17x	0/10	∼0.75x	6/10	∼1.06x	2/10
	PDF018	335	10/10	>257.91x	0/10	>257.91x	0/10	>182.38x	7/10	1.33x	10/10	12.47x	10/10	1.08x	10/10	50.70x	10/10
	PDF019	>64310	5/10	∼1.34x	0/10	∼1.34x	0/10	∼1.24x	2/10	∼1.34x	0/10	∼1.34x	0/10	0.84x	6/10	∼1.34x	0/10
	PDF021	>79791	1/10	∼1.08x	0/10	∼1.08x	0/10	∼1.01x	2/10	∼1.05x	1/10	∼1.07x	1/10	∼1.08x	0/10	∼0.75x	6/10
sqlite3_fuzz	SQL002	1383	10/10	∅	∅	1.53x	10/10	2.24x	10/10	0.21x	10/10	0.58x	10/10	2.85x	10/10	1.45x	10/10
	SQL003	>78493	1/10	∅	∅	∼1.01x	1/10	∼1.01x	1/10	∼1.10x	0/10	∼1.03x	1/10	∼1.10x	0/10	∼1.10x	0/10
	SQL012	>79271	2/10	∅	∅	∼0.77x	7/10	∼0.80x	5/10	∼0.94x	3/10	∼0.56x	7/10	∼1.09x	0/10	∼0.97x	4/10
	SQL013	*T.O.*	0/10	∅	∅	∼0.74x	4/10	∼0.75x	4/10	∼0.90x	1/10	∼0.58x	6/10	∼1.00x	0/10	∼0.93x	3/10
	SQL014	4309	10/10	∅	∅	6.66x	10/10	3.79x	10/10	3.50x	10/10	3.16x	10/10	>7.66x	8/10	4.16x	10/10
	SQL015	*T.O.*	0/10	∅	∅	∼0.85x	3/10	∼0.99x	1/10	∼0.93x	1/10	∼0.78x	4/10	∼1.00x	0/10	∼1.00x	0/10
	SQL018	2714	10/10	∅	∅	1.28x	10/10	2.46x	10/10	0.55x	10/10	0.49x	10/10	6.48x	10/10	2.02x	10/10
	SQL020	>54354	5/10	∅	∅	∼0.88x	7/10	∼1.09x	5/10	∼1.21x	3/10	∼0.59x	8/10	∼1.59x	0/10	∼1.16x	5/10
asn1	SSL001	>45461	6/10	∅	∅	∼1.86x	1/10	∼1.90x	0/10	∅	∅	>1.49x	6/10	>1.83x	1/10	<0.24x	10/10
x509	SSL009	817	10/10	∅	∅	>58.46x	9/10	>45.72x	9/10	∅	∅	18.36x	10/10	>26.77x	9/10	>96.97x	1/10
server	SSL020	863	10/10	∅	∅	>90.54x	4/10	>80.43x	3/10	∅	∅	41.74x	10/10	18.16x	10/10	>89.57x	3/10
exif	PHP004	152	10/10	∅	∅	37.22x	10/10	1.30x	10/10	∅	∅	2.51x	10/10	2.57x	10/10	>384.85x	6/10
	PHP009	235	10/10	∅	∅	>249.63x	5/10	16.66x	10/10	∅	∅	4.05x	10/10	>253.06x	5/10	>93.18x	9/10
			0.66	**∼18.44x**	**0.39**	**∼23.74x**	**0.49**	**∼13.89x**	**0.53**	**∼2.43x**	**0.53**	**∼3.04x**	**0.62**	**∼7.04x**	**0.55**	**∼16.59x**	**0.60**

Total bugs and success rate. Lyso outperforms both non-directed fuzzers (AFL, MOPT, AFL++) and directed fuzzers (Titan, FishFuzz, AFLGo, SelectFuzz) in terms of total bugs found and success rate (SR). On average, Lyso triggers 7.33 more bugs and achieves an 11.9% improvement in SR compared to non-directed fuzzers. When compared to directed fuzzers, Lyso triggers an average of 18.75 more bugs and shows a 36.1% improvement in SR. Additionally, we found that several bugs, such as PDF002 and PDF019 are particularly challenging for other fuzzers to trigger. These targets are located far from the main function, requiring traversal of over ten intermediate functions. Many alternative unpromising paths caused other fuzzers to deviate from the target. This suggests that providing promising paths can offer valuable guidance and help prevent deviation from the target.

Promising paths. We assessed whether the promising paths (i.e., alarm steps) identified by CodeQL were followed during runtime. Our analysis revealed that 48 out of 62 cases were fully explored during testing. Among these, 44 cases (corresponding to 48 bugs) successfully triggered crashes via the identified paths. For these triggered cases, we manually compared the crash backtraces with the corresponding promising paths, finding that all fully aligned with the expected execution paths. These results highlight that promising



(A) (B)

FIGURE 6: Figure 6a shows the number of interdependent and independent targets in Magma projects. Figure 6b shows interdependent groups in the tiffcp program.

paths derived from static analysis significantly enhance Lyso's ability to effectively trigger bugs.

7.3 Effectiveness of Multi-target, Multi-step Guidance

To evaluate the effectiveness of multi-target, multi-step guidance, we examined how it enhances Lyso's ability to reach both multiple and single targets. TTR was used as a supplementary metric in this evaluation. To ensure a more rigorous analysis, we refined our target set by excluding those that could be reached easily by all fuzzers within 30 minutes.

How does the multi-target, multi-step guidance accelerate Lyso in reaching multiple targets? In our evaluation, Lyso outperforms the non-directed fuzzers (AFL, MOPT, AFL++), achieving an average improvement of 4.74x. Compared to the STSS-DGF tool AFLGo, Lyso demonstrates an average speedup of 18.88x. When tested against MTSS-DGF tools like Titan and FishFuzz, Lyso shows an average improvement of 32.46x. Moreover, Lyso demonstrates its superiority by reaching all 30 targets, outperforming its competitors: AFL++ (27), AFL (26), FishFuzz (24), MOPT (23), AFLGo (23), SelectFuzz (22), and Titan (10). Further details are provided in Table 5.

We analyzed each project to evaluate the impact of interdependent targets on improving *inter-target directness*. As shown in Figure 6a, Lyso identifies 31% of targets as interdependent. The large proportion of interdependent targets underscores the importance of accounting for these relationships to improve *inter-target directness*. A focused case study on the *tiffcp* program reveals that 10 out of 14 targets are interdependent, forming three distinct groups, as shown in Figure 6b. We conducted a detailed analysis of the reaching executions within the group containing TIF005, TIF006, and TIF009, which also served as the motivating example in Figure 2. The results, presented in Figure 7, demonstrate that Lyso directs a significantly higher number of executions—by two orders of magnitude—toward the group. Notably, Lyso is the only fuzzer that reached TIF005.

How does the multi-step guidance accelerate Lyso to reach single targets? To investigate the impact of the multi-

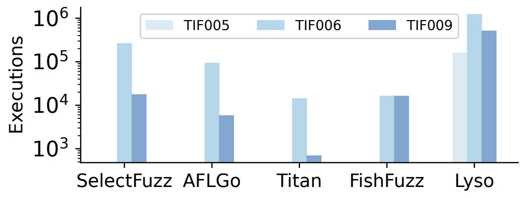


FIGURE 7: Comparison of reaching executions. The effect of interdependent targets allows Lyso to achieve more reaching executions.

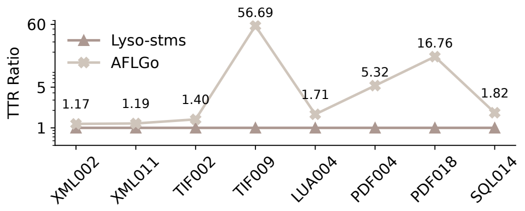


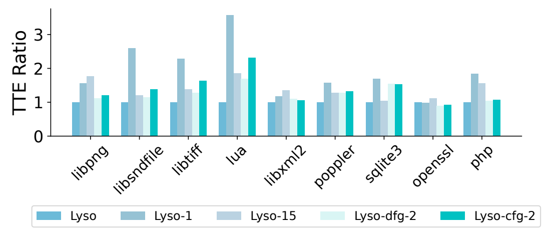
FIGURE 8: Comparison of Lyso-*stms* and AFLGo in reaching single targets. The y-axis shows the TTR ratio, highlighting the improvement achieved by Lyso-*stms* over AFLGo.

step guidance on enhancing *intra-target directness* in single-target scenarios, we applied Lyso to single targets with multiple steps, denoted as Lyso-*stms*, and compared it against AFLGo, which is specifically optimized for single-target directed fuzzing. In this setup, Lyso-*stms* focuses solely on the multi-step guidance, as the multi-target guidance is inherently deactivated due to only a single target. We selected one target from each program listed in Table 5 and conducted 10 runs per target. The selection criteria were: (1) targets should be reached by both Lyso and AFLGo within 24 hours, and (2) if a program had only one available target, that target was automatically chosen. The results are presented in Figure 8.

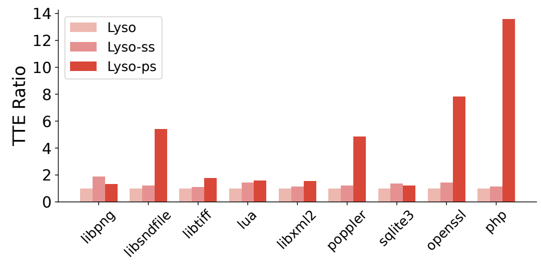
Overall, Lyso-*stms* achieves an average speedup of 10.76x compared to AFLGo. To better understand the underlying reasons for Lyso-*stms*'s significant speedup on certain targets, we conducted an analysis of execution progression over time. Specifically, we performed a case study on PDF018, as detailed in Table 3. Lyso-*stms* demonstrates a significant increase in reaching executions, growing from hundreds of thousands at 0.5 hours to over 30 million after 24 hours. In contrast, AFLGo exhibits much smaller gains, with executions increasing slowly from around 500 at 0.5 hours to 2,000 after 24 hours. Notably, after 24 hours, Lyso-*stms* achieves 15,424x executions than AFLGo. This analysis demonstrates that Lyso-*stms*'s seed selection effectively identifies the most promising seeds (i.e., depth and distance) capable of reaching the target, and its power scheduling allocates more mutations to these selected seeds.

TABLE 3: Total number of reaching executions over time for PDF018.

Bug ID	**Fuzzer**	**0.5h**	**4h**	**8h**	**16h**	**24h**
PDF018	Lyso-*stms*	2.26×10^5	2.92×10^6	6.83×10^6	1.81×10^7	3.08×10^7
	AFLGo	4.73×10^2	5.94×10^2	1.08×10^3	1.43×10^3	2.00×10^3



(A) Configuration comparison



(B) Component comparison

FIGURE 9: Figure 9a shows the impact of step count and types, while Figure 9b shows the effects of seed and power scheduling. The ratio in Y-axis indicates Lyso's performance improvement over other configurations or components.

7.4 Ablation Analysis

In this section, we conducted an ablation analysis using the same setup in §7.2 to assess how different configurations and components in Lyso contribute to overall performance.

How do step count and types affect Lyso in verifying bugs? Lyso's default configuration uses a maximum of five CFG steps and three default DFG steps. To evaluate the impact of step count, we compared the default configuration of Lyso against two variants: Lyso-1 (only the sink) and Lyso-15 (all steps)[3]. The results, presented in Figure 9a, show that Lyso outperforms both Lyso-1 and Lyso-15, achieving speedups of 1.92x and 1.40x, respectively. This suggests that using no intermediate steps offers limited guidance and provides insufficient direction for the search to reach the sink, while incorporating too many steps may increase the search effort unnecessarily.

However, in OpenSSL, we observe that Lyso-1 slightly outperforms both Lyso and Lyso-15. This can be attributed to two key factors: (1) all bugs in OpenSSL are reachable within a short timeframe (i.e., 100 seconds), meaning some executions had already reached the sink, thereby reducing the effectiveness of Lyso's multi-target, multi-step guidance, and (2) the average number of initial seeds in Magma for OpenSSL is 1,889, providing a broad spectrum of test cases that could potentially expose bugs. In this scenario, Lyso-1's seed scheduling effectively prioritizes promising seeds that have reached the bug but not yet triggered it.

[3]We set the maximum threshold to 15 based on the observation that the most alarms have no more than 15 steps.

TABLE 4: Newly identified vulnerabilities. "Reported" refers to alarms generated by CodeQL, while "Triggered" denotes true positives found by Lyso. "Status" indicates the current vulnerability status: CVE assigned, confirmed ("C"), or pending ("P").

Program	Reported	Triggered	Bug Type	Status
Libsndfile	75	5	Heap Buffer Overflow	CVE-2024-42882
			Memory Leak	CVE-2024-42883
			Use Before Initialization	CVE-2024-42884
			Use Before Initialization	CVE-2024-42884
			Use Before Initialization	CVE-2024-42884
GPAC	856	8	Stack Buffer Overflow	issue-2935 (C)
			Null Pointer Dereference	issue-2934 (C)
			Null Pointer Dereference	issue-2933 (C)
			Memory Leak	issue-2931 (C)
			Null Pointer Dereference	issue-2929 (C)
			Null Pointer Dereference	issue-2926 (C)
			Null Pointer Dereference	issue-2925 (C)
			Type Confusion	issue-2924 (C)
Poppler	809	3	Stack Overflow	issue-1508 (C)
			Stack Overflow	issue-1509 (P)
			Out-of-Bounds Write	issue-1511 (C)
Xpdf	298	2	Divide By Zero	CVE-2024-7867
			Stack Overflow	CVE-2024-7866

Additionally, to evaluate the impact of different step types, we compared three Lyso variants: Lyso-1 (only the sink), Lyso-dfg-2 (the sink and the vulnerable operation), and Lyso-cfg-2 (the sink and a function call site, with the middle in the sequence being selected). Overall, Lyso-dfg-2 demonstrates improvements of 10.86% and 35.93% compared to Lyso-cfg-2 and Lyso-1, respectively. These results suggest that incorporating additional steps, whether a DFG or CFG step, enhances the effectiveness of bug verification compared to relying solely on the sink. Furthermore, a vulnerable operation of a DFG step is more effective than a function call site of a CFG step in verifying bugs.

How do the seed scheduling and the power scheduling affect Lyso in verifying bugs? We assessed the influence of seed scheduling and power scheduling on Lyso's performance by comparing it with two modified versions: Lyso-*ss* (power scheduling disabled) and Lyso-*ps* (seed scheduling disabled). Overall, Lyso outperforms Lyso-*ss* by a factor of 1.32x and Lyso-*ps* by 4.34x. These results highlight the substantial benefits of Lyso's integrated scheduling strategies over the use of either strategy alone. Moreover, the results indicate that seed scheduling plays a more critical role than power scheduling in enhancing bug verification.

7.5 Detecting New Vulnerabilities

We investigated the feasibility of using Lyso to automatically verify static analysis results generated by CodeQL [16]. A summary of the results is presented in Table 4. In total, static analysis identified 2,038 alarms related to security bugs, and Lyso successfully discovered 18 new vulnerabilities. The distribution of alarm types across the tested programs is illustrated in Figure 10.

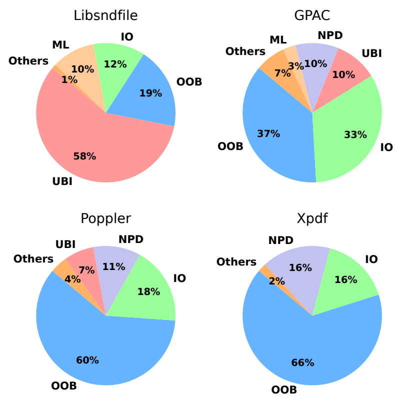


FIGURE 10: Proportions of different types of alarms. OOB: Out of Bound, IO: Integer Overflow, ML: Memory Leak, UBI: Use Before Initialization, NPD: Null Pointer Dereference.

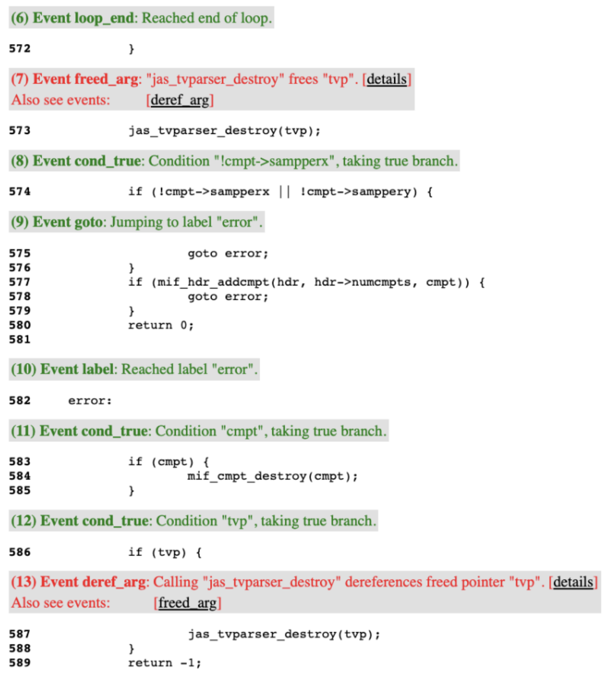
Each program was compiled with AddressSanitizer (Asan) [33] and MemorySanitizer (Msan) [34]. Given the large number of alarms, Lyso was executed for 72 hours per program. Overall, 35 out of 75 alarms in Libsndfile, 235 out of 856 in GPAC, 150 out of 809 in Poppler, and 68 out of 298 in Xpdf were fully followed after 72 hours. We then compared the crashes reported by Asan and Msan to the promising paths identified by static analysis. Our findings indicate that 15 out of the 18 bugs were triggered along the paths identified by static analysis, which Lyso successfully verified. For instance, CodeQL identified a memory leak bug in Libsndfile, generating a path from the source function, `psf_allocate` at `sndfile.c:362`, to the sink function, `psf_open_file` at `sndfile.c:436`. A comparison of the backtrace confirmed alignment with the path generated by CodeQL, validating Lyso's effectiveness.

Additionally, Lyso triggered three previously unreported vulnerabilities: CVE-2024-7866, issue-1508, and issue-1509. Although these vulnerabilities were not directly reported by CodeQL, we observed that some reported alarms were located within the same functions as these vulnerabilities. We hypothesize that these alarms, which shared a common sink function with the vulnerabilities, enabled Lyso's overlap metric to prioritize seeds targeting these functions. This prioritization likely increased the likelihood of successfully triggering the vulnerabilities.

8 Discussion

8.1 The Generality of Lyso

Lyso's critical steps pattern, outlined in Table 1, is designed for broad applicability. Extending Lyso to other bug categories requires only the definition of new rules to identify vulnerable operations. For example, in detecting missing security check bugs [35], the vulnerable operation corresponds to the step where a security check is bypassed. Additionally, Lyso



(6) Event loop_end: Reached end of loop.

```
572             }
```

(7) Event freed_arg: "jas_tvparser_destroy" frees "tvp". [details]
Also see events: [deref_arg]

```
573             jas_tvparser_destroy(tvp);
```

(8) Event cond_true: Condition "!cmpt->sampperx", taking true branch.

```
574             if (!cmpt->sampperx || !cmpt->samppery) {
```

(9) Event goto: Jumping to label "error".

```
575                     goto error;
576             }
577             if (mif_hdr_addcmpt(hdr, hdr->numcmpts, cmpt)) {
578                     goto error;
579             }
580             return 0;
581
```

(10) Event label: Reached label "error".

```
582     error:
```

(11) Event cond_true: Condition "cmpt", taking true branch.

```
583             if (cmpt) {
584                     mif_cmpt_destroy(cmpt);
585             }
```

(12) Event cond_true: Condition "tvp", taking true branch.

```
586             if (tvp) {
```

(13) Event deref_arg: Calling "jas_tvparser_destroy" dereferences freed pointer "tvp". [details]
Also see events: [freed_arg]

```
587                     jas_tvparser_destroy(tvp);
588             }
589             return -1;
```

FIGURE 11: A Use-After-Free alarm detected in Jasper by Coverity.

can integrate with static analysis tools like Coverity [1] and Infer [15]. To demonstrate this, we conducted case studies where Lyso verified alarms generated by these tools.

Coverity. Lyso was applied to verify a Use-After-Free (UAF) alarm in Jasper flagged by Coverity, as illustrated in Figure 11. Coverity identifies the sink inside the `jas_tvparser_destroy` function at `mif_code.c:587`, where a previously freed pointer is accessed. Due to space constraints, details of the variable `tvp`'s allocation and preceding steps (prior to line 572) are omitted. Our analysis identifies three function call sites: the allocation of `tvp`, the call sites at line 573, and the call site at line 587. These were used as CFG steps. The vulnerable operation is identified as the *free* operation within the `jas_tvparser_destroy` function. After 24 hours of fuzzing, we analyzed the root cause of the crash by examining the backtrace. Lyso triggered the crash approximately four hours after fuzzing. The crash occurred due to a *free* operation at `mif_code.c:587` being executed after the same pointer had already been freed at `mif_code.c:573`, confirming a double-free bug (a specific type of UAF bug). This backtrace matched the path identified by Coverity.

Infer. Lyso was also applied to verify a null pointer dereference alarm (Figure 13) in Jasper flagged by Infer. Infer identified the sink at `jas_image:777`, where the `image` object could potentially be null and is dereferenced at this location. Our analysis collects four function call sites: call to `jas_image_copy` function, and call sites at lines 213, 190, and 214. The vulnerable operation is identified at line 191 where `jas_image_create0` return NULL. No crashes related to the sink were observed after 24 hours of testing. We manually confirmed this as a true positive alarm, but triggering the bug would require the fuzzer to craft an extremely large image object, causing the system to run out of memory.

8.2 Threats to Validity

To prevent false positives from interfering with true positive verification, we excluded false positives from our main evaluation. A potential question is whether false positive alarms should also be included. Lyso, guided by promising paths from static analysis, focuses on alarms likely to lead to real vulnerabilities. False positive alarms, however, cause Lyso to waste resources attempting to verify alarms that cannot result in a definitive outcome (e.g., a crash). This limitation also affects other DGF tools. Furthermore, as shown in real-world testing scenarios (§7.5), Lyso successfully verifies true positive alarms even when false positives are present.

CodeQL [16] is used to derive promising paths in our main evaluation. A potential question is why other static analysis tools were not used. The reason is that CodeQL's ability to define custom source-sink queries provides greater accuracy and flexibility, especially when source and sink locations are available. This is especially useful for the Magma benchmark, where sink information and vulnerable variables are available. More importantly, our work does not aim to generate promising paths but to demonstrate that, given such paths, Lyso significantly improves the verification of true positive alarms.

8.3 Limitations

In Lyso, five CFG steps and three DFG steps were empirically chosen as the optimal balance between providing effective guidance and minimizing overhead. As demonstrated in §7.4, using fewer steps offers limited guidance, which may fail to sufficiently direct the search toward the sink. Conversely, incorporating more steps increases search overhead, potentially leading to diminishing returns. However, determining the optimal number of steps remains an open research question, as the ideal balance may vary depending on the program complexity, the nature of the alarms, and the underlying static analysis tool.

9 Related Work

In addition to prior work on DGF (§2.2), we review related work that enhances DGF by employing pruning techniques that are orthogonal to Lyso's methodology. Beacon [36] uses static analysis to identify unreachable code regions and inserts assertion checks to terminate executions that fail to meet the preconditions for reaching the targets. SieveFuzz [37] proposes a dynamic technique to terminate the unreachable paths at runtime. SelectFuzz [22] and DAFL [38] leverage static analysis to selectively instrument only data- and control-dependent code regions relevant to target sites.

Apart from DGF, several other dynamic methods have been proposed to address and verify true positive alarms from static analysis reports. Maria Christakis et al. [39] utilized statements corresponding to the annotations generated by the static analyzer to guide dynamic symbolic execution (DSE) in verifying potential vulnerable paths and alarms. To mitigate the substantial time and computational resources required by DSE, FuzzSlice [40] focuses fuzzing efforts solely on the function containing the warning, aiming to reduce possible false positives. However, by replacing entire-program fuzzing with function-level fuzzing, FuzzSlice faces the issue of false positives due to the lack of runtime context for each vulnerable function.

Various other methods have been developed to prune false alarms and identify true bugs. Wang et al. [41] conducted a systematic evaluation of features proposed in the literature and identified "Golden Features" as the most critical for detecting actionable static warnings. Yang et al. [42] also propose the use of machine learning and deep learning techniques, such as SVM, LLM, which have proven effective in detecting true bugs. However, these methods, which rely on learning features to identify these false positives, may miss or misclassify cases involving intricate control flow and data flow dependencies. In contrast, DGF uses dynamic execution to guide the analysis, allowing it to delve deeper into complex paths and more effectively uncover true bugs.

10 Conclusion

In conclusion, Lyso effectively addresses the critical challenge of verifying true positives from static analysis tools. Its multi-target, multi-step guided fuzzing approach, which utilizes promising paths and alarm correlation, significantly improves existing techniques in static analysis results verification. Our evaluation against the state-of-the-art (directed) fuzzers demonstrates Lyso's superior performance, achieving an average speedup of 12.17x in verifying multiple alarms. Additionally, Lyso has proven its practical usefulness by discovering eighteen new vulnerabilities in real-world applications with the aid of a static analysis tool. This research underscores Lyso's potential to strengthen software security and improve the usability of static analysis.

11 Acknowledgment

We would like to thank the anonymous reviewers for their feedback and suggestions. Wenjia Zhao was supported by the National Natural Science Foundation of China (62302388) and the China Postdoctoral Science Foundation (2023M742793). All opinions expressed in this paper are solely those of the authors.

References

[1] "Coverity." [Online]. Available: https://scan.coverity.com/

[2] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[3] M. Alfadel, D. E. Costa, E. Shihab, and B. Adams, "On the discoverability of npm vulnerabilities in node. js projects," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–27, 2023.

[4] M. Nadeem, B. J. Williams, and E. B. Allen, "High false positive detection of security vulnerabilities: a case study," in *Proceedings of the 50th Annual Southeast Regional Conference*, 2012, pp. 359–360.

[5] H. Kim, M. Raghothaman, and K. Heo, "Learning probabilistic models for static analysis alarms," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1282–1293.

[6] K. Julisch and M. Dacier, "Mining intrusion detection alarms for actionable knowledge," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002, pp. 366–375.

[7] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 83–93, 2004.

[8] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik, "User-guided program reasoning using bayesian inference," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 722–735.

[9] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[10] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections." *IEEE Softw.*, vol. 38, no. 3, pp. 79–86, 2021.

[11] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2095–2108.

[12] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[13] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 47–62.

[14] "Clang static analyzer," https://clang-analyzer.llvm.org/.

[15] "Infer." [Online]. Available: https://fbinfer.com/

[16] "Codeql." [Online]. Available: https://codeql.github.com/

[17] K. Heo, M. Raghothaman, X. Si, and M. Naik, "Continuously reasoning about programs using differential bayesian inference," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 561–575.

[18] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[19] "American fuzzy loop." [Online]. Available: https://lcamtuf.coredump.cx/afl/

[20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[21] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949–1966.

[22] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1050–1064.

[23] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "FISHFUZZ: Catch deeper bugs by throwing larger nets," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1343–1360.

[24] H. Huang, P. Yao, H.-C. Chiu, Y. Guo, and C. Zhang, "Titan: Efficient multi-target directed greybox fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 59–59.

[25] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2289–2306.

[26] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[27] A. L. Davis and R. M. Keller, "Data flow program graphs," *Computer*, vol. 15, no. 02, pp. 26–41, 1982.

[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[29] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: a directed greybox fuzzer driven by deviation basic blocks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2440–2451.

[30] "llvm." [Online]. Available: https://llvm.org/

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[32] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 137–137.

[33] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[34] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 46–55.

[35] K. Lu, A. Pakki, and Q. Wu, "Detecting Missing-Check bugs via semantic-and Context-Aware criticalness and constraints inferences," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1769–1786.

[36] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 36–50.

[37] P. Srivastava, S. Nagy, M. Hicks, A. Bianchi, and M. Payer, "One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 388–399.

[38] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "DAFL: Directed grey-box fuzzing guided by data dependency," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4931–4948.

[39] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 144–155.

[40] A. Murali, N. Mathews, M. Alfadel, M. Nagappan, and M. Xu, "Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[41] J. Wang, S. Wang, and Q. Wang, "Is there a" golden" feature set for static warning identification? an experimental evaluation," in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–10.

[42] X. Yang, J. Chen, R. Yedida, Z. Yu, and T. Menzies, "Learning to recognize actionable static code warnings (is intrinsically easy)," *Empirical Software Engineering*, vol. 26, pp. 1–24, 2021.

[43] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[44] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level." in *NDSS*, 2021.

[45] L. O. Andersen, "Program analysis and specialization for the c programming language," 1994.

[46] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

A Appendix

A.1 Motivating Example

To demonstrate how Lyso improves *inter-target* and *intra-target directness* in real-world programs, we present a motivating example, as shown in Figure 12. The example is a modified version of the *tiffcp* program from the Magma benchmark. For clarity and conciseness, we have omitted code that is not directly relevant to the bugs.

```
typedef struct{
    ...
    uint16_t td_transferfunction[3];
    ...
} TIFFDirectory

int main(){
    _TIFFVSetField();
    ...
    if (/* Unsuccessful handle image*/)
       TIFFFlush();
}

int _TIFFVSetField(){
  ...
  case TIFFTAG_TRANSFERFUNCTION:
    v = (td_samplesperpixel - td_extrasamples) > 1 ? 3 :
         1; // A DFG step to TIF009
    for (i = 0; i < v; i++)
        _TIFFsetShortArray(&td->td_transferfunction[i],
             va_arg(ap, uint16_t*), 1U <<
             td_bitspersample);
}

int TIFFFlush() {
    ...
    TIFFReDir(); // A CFG step to TIF005, 006, 009
}

int TIFFReDir(){
    ...
    TIFFWDirSec(); // A CFG step to TIF005, 006, 009
}

int TIFFWDirSec(){
    ...
    (*tif->tif_close)(); // A CFG step to TIF005, 006
    ...
    TIFFWDTT(); // A CFG step to TIF009
}

int TIFFWDTT() {
    ...
    if (td_samplesperpixel - td_extrasamples == 3)
       assert(&td->td_transferfunction[2] == NULL); //
            TIF009 sink
}

void LogLuvClose() {
    ...
    assert(sp->encoder_state == 0); // TIF005 sink
}

void PixarLogClose(){
    ...
    assert(sp->state & PLSTATE_INIT == 0); // TIF006 sink
}
```

FIGURE 12: A motivating example for Lyso's improvements in inter-target and intra-target directness.

A.2 Distance Measurement

Graph construction. To measure the distance from the execution trace to a given target basic block, Lyso first builds the Call Graph (CG) and CFG, which are then merged into an ICFG. However, indirect calls present challenges in building an accurate ICFG [43], [44], leading to incomplete CG edges and imprecise distance calculation. To address this, Lyso employs a type-based analysis [43], which is more scalable than points-to analysis [45] used in other DGF tools [11, 22, 38]. This scalability makes Lyso suitable for large applications, such as OpenSSL and PHP.

Distance Table Precomputation. To calculate the shortest distance for an execution to a target basic block during fuzzing, Lyso pre-computes a table that records the shortest distance between all pairs of basic blocks within the program. To further optimize performance, this table is implemented using a hash map, allowing constant time lookups and minimizing query overhead. We use bb and bb' to represent any two basic blocks for which the distance needs to be calculated. We assume that bb belongs to function f and bb' belongs to function f', namely $\forall bb,\ bb',\ bb \in f,\ bb' \in f'$. To accurately calculate the distance between two targets, an inter-procedural analysis approach is adopted when constructing the control flow graph. $d(bb,\ bb')$ represents the inter-procedural basic block distance calculated between two basic blocks. The distance calculation process is divided into four distinct phases:

- Basic Block Distance in Function:

$$bbd(bb_i, bb_j) = \begin{cases} 0 & bb_i = bb_j, \\ \min\limits_{bb_s \in \mathrm{succ}(bb_i)} bbd(bb_s, bb_j) + 1 & bb_i \to bb_j, \\ \infty & \text{otherwise.} \end{cases} \tag{4}$$

- Edge Weight for Function Distance:

$$\mathrm{weight}(f, f') = \min_{bb_{f'} \in \mathrm{callsite}(f, f')} bbd(entry(f),\ bb_{f'}) \tag{5}$$

- Function Distance:

$$fd(f, f') = \begin{cases} 0 & f = f', \\ \min\limits_{f_s \in \mathrm{succ}(f)} fd(f_s, f') + \mathrm{weight}(f, f_s) & f \to f', \\ \infty & \text{otherwise.} \end{cases} \tag{6}$$

- Inter-procedural Basic Block Distance:

$$d(bb, bb') = \begin{cases} bbd(bb, bb') & f = f', \\ fd(f, f') + bbd(bb, callsite(f, f')) \\ \qquad + bbd(entry(f'), bb') & f \to f', \\ \infty & \text{otherwise.} \end{cases} \tag{7}$$

Here, in Equation 4 and Equation 6, the `succ` function returns all intermediate successors. In addition, in Equation 5,

TABLE 5: We evaluated TTR and SR for targets in Magma. *Time* represents the average TTR in ten runs. *SR* represents success rate. *T.O* denotes the fuzzer fails to reach the target within 24 hours. *Ratio* measures the improvement ratio achieved by Lyso compared to other fuzzers. ∅ indicates deployment was not feasible.

Program & Bug ID		Lyso		Titan			FishFuzz			AFLGo			SelectFuzz			AFL			MOPT			AFL++		
		Time	*SR*	*Time*	*Ratio*	*SR*	*Time*	*Ratio*	*SR*	*Time*	*Ratio*	*SR*	*Time*	*Ratio*	*SR*	*Time*	*Ratio*	*SR*	*Time*	*Ratio*	*SR*	*Time*	*Ratio*	*SR*
tiff_fuzzer	TIF001	21648	10/10	>78525	>3.63x	1/10	>40952	>1.89x	9/10	>49181	>2.27x	6/10	>47033	>2.17x	8/10	>24686	>1.14x	9/10	>52450	>2.42x	6/10	>77408	>3.57x	2/10
	TIF002	23310	10/10	>64286	>2.76x	4/10	33787	1.45x	10/10	28220	1.21x	10/10	24214	1.04x	10/10	>18451	>0.79x	9/10	27766	1.19x	10/10	10786	0.46x	10/10
	TIF008	>47959	6/10	>78104	~ 1.63x	1/10	>62476	~1.30x	6/10	>43044	~0.90x	7/10	*T.O.*	~1.80x	0/10	>28273	~0.59x	8/10	>64803	~1.35x	3/10	>83161	~1.73x	2/10
	TIF010	330	10/10	>21555	>65.32x	8/10	1780	5.39x	10/10	536	1.62x	10/10	374	1.13x	10/10	230	0.70x	10/10	450	1.36x	10/10	1991	6.03x	10/10
tiffcp	TIF001	26353	10/10	*T.O.*	>3.28x	0/10	>77358	>2.94x	2/10	>78936	>3.00x	5/10	>64484	>2.45x	5/10	>60700	>2.30x	4/10	>65602	>2.49x	4/10	>73927	>2.81x	3/10
	TIF002	40242	10/10	>54354	>1.35x	5/10	>85568	>2.13x	1/10	>53575	>1.33x	7/10	>43268	>1.08x	7/10	>63196	>1.57x	5/10	>73442	>1.83x	3/10	>36871	>0.92x	8/10
	TIF005	>80902	2/10	*T.O.*	~1.07x	0/10	*T.O.*	~ 1.07x	0/10	*T.O.*	~1.07x	0/10	*T.O.*	~1.07x	0/10	*T.O.*	~1.07x	0/10	*T.O.*	~1.07x	0/10	1321	<0.016x	10/10
	TIF006	37151	10/10	>51135	>1.38x	6/10	>54004	>1.45x	8/10	>31366	>0.84x	9/10	19039	0.51x	10/10	24526	0.66x	10/10	>32554	>0.88x	9/10	2622	0.07x	10/10
	TIF008	>84667	1/10	*T.O.*	~1.02x	0/10	>81514	~0.96x	1/10	*T.O.*	~1.02x	0/10	*T.O.*	~1.02x	0/10	>58705	~0.69x	3/10	>74964	~0.89x	2/10	>72169	~0.85x	3/10
	TIF009	1910	10/10	>27513	>14.40x	7/10	21324	11.16x	10/10	>36669	>19.20x	9/10	3486	1.83x	10/10	14966	7.84x	10/10	>24536	>12.85x	9/10	>27123	>14.20x	9/10
	TIF010	673	10/10	3646	5.42x	10/10	7998	11.88x	10/10	7302	10.85x	10/10	>9702	>14.42x	9/10	1031	1.53x	10/10	1155	1.72x	10/10	1987	2.95x	10/10
lua	LUA002	7761	10/10	>76227	>9.82x	2/10	18079	2.33x	10/10	>38092	>4.91x	9/10	20531	2.65x	10/10	>19306	>2.49x	9/10	21175	2.73x	10/10	24841	3.20x	10/10
	LUA004	5794	10/10	>67860	>11.71x	3/10	10297	1.78x	10/10	14644	2.53x	10/10	10117	1.75x	10/10	6689	1.15x	10/10	13052	2.25x	10/10	22753	3.93x	10/10
libxml2_fuzzer	XML002	>75190	2/10	*T.O.*	~1.15x	0/10	>59914	~0.80x	5/10	*T.O.*	~1.15x	0/10	*T.O.*	~1.15x	0/10	>79964	~1.06x	2/10	*T.O.*	~1.15x	0/10	*T.O.*	~1.15x	0/10
xmllint	XML002	>78131	1/10	*T.O.*	~1.11x	0/10	>74967	~0.96x	2/10	*T.O.*	~1.11x	0/10	*T.O.*	~1.11x	0/10	*T.O.*	~1.11x	0/10	>81925	~1.05x	1/10	>83195	~1.06x	1/10
	XML011	>73714	2/10	*T.O.*	~1.17x	0/10	>78303	~1.06x	1/10	*T.O.*	~1.17x	0/10	74512	~1.01x	3/10	>80134	~1.09x	1/10	>72765	~0.99x	3/10	*T.O.*	~1.17x	0/10
pdf_fuzzer	PDF004	29732	10/10	*T.O.*	>2.91x	0/10	>78070	>2.63x	1/10	*T.O.*	>2.91x	0/10	>77973	>2.62x	2/10	>76610	>2.58x	2/10	>83461	>2.81x	1/10	>67600	>2.27x	3/10
	PDF018	332	10/10	*T.O.*	>260.24x	0/10	*T.O.*	>260.24x	0/10	>76927	>231.70x	2/10	1373	4.14x	10/10	1556	4.69x	10/10	2725	8.21x	10/10	25728	77.49x	10/10
pdftoppm	PDF004	>69562	4/10	*T.O.*	~1.24x	0/10	*T.O.*	~1.24x	0/10	*T.O.*	~1.24x	0/10	*T.O.*	~1.24x	0/10	*T.O.*	~1.24x	0/10	>75098	~1.08x	4/10	>70230	~1.01x	3/10
	PDF018	480	10/10	*T.O.*	>180.00x	0/10	*T.O.*	>180.00x	0/10	>69664	>145.13x	2/10	1145	2.39x	10/10	1377	2.87x	10/10	2579	5.37x	10/10	23699	49.37x	10/10
pdfimages	PDF004	335	10/10	*T.O.*	>257.91x	0/10	*T.O.*	>257.91x	0/10	>40565	>121.09x	9/10	446	1.33x	10/10	4176	12.47x	10/10	372	1.11x	10/10	16984	50.70x	10/10
sqlite3_fuzz	SQL003	>73470	2/10	∅	∅	∅	>77541	~1.06x	2/10	>62330	~0.85x	5/10	*T.O.*	~1.18x	0/10	>47363	~0.64x	7/10	*T.O.*	~1.18x	0/10	>63589	~0.87x	7/10
	SQL006	>46395	6/10	∅	∅	∅	17769	<0.38x	10/10	>41224	~0.89x	8/10	>63607	~1.37x	4/10	>19089	~0.41x	9/10	>79231	~1.71x	2/10	>58147	~1.25x	6/10
	SQL009	728	10/10	∅	∅	∅	1556	2.14x	10/10	783	~1.08x	10/10	339	0.47x	10/10	387	0.53x	10/10	1990	2.73x	10/10	1563	2.15x	10/10
	SQL011	>73188	2/10	∅	∅	∅	*T.O.*	~1.18x	0/10	>72694	~0.99x	2/10	*T.O.*	~1.18x	0/10	*T.O.*	~1.18x	0/10	*T.O.*	~1.18x	0/10	*T.O.*	~1.18x	0/10
	SQL012	36481	10/10	∅	∅	∅	>40891	>1.12x	8/10	25672	~0.70x	10/10	25866	0.71x	10/10	8947	0.25x	10/10	*T.O.*	>2.37x	0/10	26532	0.73x	10/10
	SQL013	>73586	2/10	∅	∅	∅	>48183	~0.65x	7/10	>54165	~0.74x	6/10	>68125	~0.93x	3/10	>35258	~0.48x	8/10	*T.O.*	~1.17x	0/10	>75530	~1.03x	4/10
	SQL014	1678	10/10	∅	∅	∅	1994	1.19x	10/10	3017	1.80x	10/10	1029	0.61x	10/10	400	0.24x	10/10	15146	9.03x	10/10	3236	1.93x	10/10
	SQL018	239	10/10	∅	∅	∅	1187	4.97x	10/10	513	2.15x	10/10	239	1.00x	10/10	266	1.11x	10/10	13015	54.46x	10/10	1460	6.11x	10/10
	SQL020	>54351	6/10	∅	∅	∅	>40687	~0.75x	8/10	>59314	~1.09x	5/10	>67273	~1.24x	3/10	>30756	~0.57x	8/10	*T.O.*	~1.59x	0/10	>63171	~1.16x	5/10
			0.72		**~39.45x**	**0.22**		**~25.47x**	**0.54**		**~18.88x**	**0.57**		**~1.89x**	**0.58**		**~1.83x**	**0.68**		**~4.34x**	**0.52**		**~8.05x**	**0.65**

TABLE 6: Empirical analysis of vulnerable operations derived from CodeQL rules. The "CodeQL Rules" column lists references corresponding to specific CodeQL queries.

Bug Type	**CWE ID**	**CodeQL Rules**	**The Vulnerable Operation**
Integer Overflow (IO)	CWE-190, CWE-192	1, 2	integer arithmetic
Out of Bound (OOB)	CWE-119, CWE-120	1	index assignment
Use after free (UAF)	CWE-416	1	free operation
NULL Pointer Dereference (NPD)	CWE-476	1	pointer assigned to null
Use Before Initialization (UBI)	CWE-457	1	conditional initialization
Divide By Zero (DBZ)	CWE-369	1	value assigned to zero

the `callsite` function returns all call sites in the caller function f that leads to the callee function f'. The `entry` function return the entry basic block of f. In Equation 7, the inter-procedural basic block level distance considers two scenarios. If both basic blocks bb and bb' are within the same function, namely $f = f'$, the distance is simply the intra-procedural basic block distance between these two blocks. However, if bb and bb' are in different functions, and f' is reachable from f, the total distance is calculated as the sum of three components: $fd(f, f')$, which is the inter-procedural function distance between the two functions; $bbd(bb, callsite(f, f'))$, which represents the intra-procedural basic block level distance from bb to the call site basic block within f that leads to f'; and $bbd(entry(f'), bb')$, which is the intra-procedural basic block level distance from the entry block of f' to bb'.

A.3 Configurations

Our fuzzer selection criteria are as follows: For STSS-DGF, we included AFLGo [9]. For MTSS-DGF, we selected Parmesan, FishFuzz, and Titan [23–25]. We excluded Parmesan from Table 2 because it does not support non-deterministic mode. However, its results are provided separately in §A.5. To evaluate the effectiveness of Lyso's multi-target, multi-step guidance, we compared it with SelectFuzz [22], which utilizes selective instrumentation, an orthogonal but considered effective approach. Although we would like to include STMS-DGF tools like CAFL [12] and UAFUZZ [13] in our comparisons, CAFL was unavailable, and UAFUZZ, which is specifically designed for UAF bugs in binary analysis, was incompatible with the Magma [18] benchmark. Additionally, we included AFL++ [20] and MOPT [21], as they are the top two coverage-guided fuzzers in the Magma benchmark. AFL [19] was used as the baseline for all fuzzers.

To ensure a fair comparison, we ran all fuzzers in non-deterministic mode, as AFL++, MOPT, and Titan use this mode by default. For the other fuzzers, we enabled non-deterministic mode using the *-d* flag. Additionally, for AFL++, we activated the *cmplog* feature, which is critical for its bug-finding performance. The detailed configuration of each fuzzer used in our evaluation is summarized in Table 7.

In our evaluation, we used the state-of-the-art Magma benchmark [18]. Since Magma does not natively support AFLGo [9], FishFuzz [23], or SelectFuzz [22], we extended the Magma scripts to integrate these tools. Additionally, we made modifications to the source code to ensure compatibility with Magma. Specifically, we made the following changes:

- SelectFuzz, based on LLVM 4.0.0, required modifications to the LLVM source code to ensure compatibility (sigaltstack error) with Ubuntu 18.04.

- FishFuzz, originally designed for sanitizer-instrumented targets, was adapted for use with Magma by modifying the `has_sanitizer_instrumentation` function in the `afl-llvm-pass.so.cc` file, following guidance from the author of FishFuzz.

For certain programs where Titan and SelectFuzz were incompatible, we encountered the following issues:

- SelectFuzz's `libDFUZZPASS.so` caused a segmentation fault when run on PHP and OpenSSL.

- Titan encountered several issues during testing. First, we fixed a linking error when running Titan on SQLite3 and submitted a patch to the authors, but the instrumentation still failed to provide coverage feedback. Second, Titan could not run on OpenSSL because its static analysis failed to generate the `bug_conf_cluster` file. Third, when running on PHP, its static analysis used over 88.7GB of memory, exceeding the 96GB RAM limit of our machine. We reported these issues to the Titan team, but no resolution has been provided to date.

TABLE 7: Arguments and versions of each fuzzer in the evaluation

Fuzzer	Arguments	Version
Lyso	-d	-
AFL [19]	-d	v2.57b
AFLGo [9]	-d	fa125da
SelectFuzz [22]	-d	6da35e0
FishFuzz [23]	-d	862df0f
Titan [24]	-d	a625968
MOPT [21]	-L	a9a5dc5
AFL++ [20]	-d -c cmplog	458eb08
Parmesan [25]	-	fac5801

A.4 Overhead

To assess the potential overhead introduced by Lyso, we compared its performance with other state-of-the-art DGF tools by analyzing the number of executions, as shown in Table 8. Lyso demonstrates a performance advantage, executing 2.52x more test cases than the MTSS-DGF tool Titan. This is because the dynamic taint analysis used in Titan can significantly slow down its performance.

Additionally, Lyso outperforms another MTSS-DGF tool, FishFuzz, which employs function-level distance guidance. Lyso achieves 19% more executions while providing finer-grained basic block-level guidance. This performance gain is primarily due to Lyso's step instrumentation approach (§4.2). Unlike full instrumentation, which includes all basic blocks, Lyso selectively instruments only the reachable basic blocks for critical steps. Our experiment demonstrates that step instrumentation achieves an average speed improvement of 2.86x compared to full instrumentation.

However, we observed that SelectFuzz and the STSS-DGF tool AFLGo achieved slightly better results compared to Lyso, with 1.37x and 1.12x more executions, respectively. This is because they do not utilize the trace tracking mechanism implemented in Lyso. While this reduces computational overhead, it results in a trade-off by increasing the Time-to-Exposure (TTE).

TABLE 8: Execution counts for different fuzzers across programs (in scientific notation). ∅ indicates deployment was not feasible.

Program	Lyso	Titan	FishFuzz	AFLGo	SelectFuzz
libpng_fuzzer	1.67×10^9	2.31×10^8	1.36×10^9	2.67×10^9	2.29×10^9
sndfile_fuzzer	1.60×10^8	1.27×10^8	9.88×10^7	1.16×10^8	9.88×10^7
tiff_fuzzer	5.92×10^8	1.00×10^8	5.00×10^8	2.55×10^8	6.72×10^8
tiffcp	1.63×10^8	1.28×10^8	1.33×10^8	1.34×10^8	1.65×10^8
lua	9.49×10^7	9.28×10^7	8.81×10^7	8.02×10^7	1.17×10^8
libxml2_fuzzer	1.69×10^8	5.69×10^7	4.29×10^8	3.05×10^8	3.68×10^8
xmllint	6.93×10^7	6.42×10^7	9.65×10^7	8.60×10^7	1.23×10^8
pdf_fuzzer	1.28×10^7	5.20×10^6	5.20×10^6	6.24×10^6	1.30×10^7
pdftoppm	1.38×10^7	1.32×10^7	6.18×10^6	6.82×10^6	1.42×10^7
pdfimages	2.14×10^7	9.25×10^6	9.25×10^6	1.17×10^7	2.25×10^7
sqlite3_fuzz	1.24×10^8	∅	1.67×10^8	1.96×10^8	5.08×10^8
asn1	2.17×10^7	∅	1.06×10^7	9.31×10^6	∅
x509	6.17×10^8	∅	3.65×10^8	5.05×10^8	∅
server	1.97×10^7	∅	8.56×10^6	6.10×10^6	∅
exif	6.38×10^8	∅	3.96×10^8	5.05×10^8	∅
Average	2.92×10^8	0.83×10^8	2.45×10^8	3.26×10^8	4.00×10^8

A.5 Lyso vs. Parmesan

We compared Lyso with Parmesan and evaluated their TTE in Magma. We excluded SQLite3, Poppler, OpenSSL, and PHP from the comparison because Parmesan cannot be compiled with those programs. For most targets, Lyso outperforms Parmesan, as detailed in Table 9. However, we also noticed that for PNG006 and XML003, Parmesan achieved an improvement of two orders of magnitude. This is because Parmesan is based on Angora [46], which employs effective dynamic taint analysis to mutate the corresponding inputs related to path constraints that lead to the bug, rather than relying on random mutation as in Lyso. A potential future direction for Lyso is to incorporate dynamic taint analysis to enhance its performance for cases where path constraints are hard to solve by random mutations.

TABLE 9: TTE comparisons between Lyso and Parmesan.

Program & Bug ID		**Lyso**	**Parmesan**
		Time	*Time*
libpng_fuzzer	PNG001	*T.O.*	*T.O.*
	PNG006	>78257	52
	PNG007	2644	*T.O.*
sndfile_fuzzer	SND001	36	*T.O.*
	SND006	309	*T.O.*
	SND007	38	*T.O.*
	SND017	26	*T.O.*
	SND020	46	*T.O.*
	SND024	39	*T.O.*
tiff_fuzzer	TIF002	28035	*T.O.*
	TIF008	>47993	*T.O.*
	TIF012	164	> 64148
	TIF014	429	*T.O.*
tiffcp	TIF002	>52237	*T.O.*
	TIF005	>80902	>26875
	TIF006	37145	>18256
	TIF008	>84667	*T.O.*
	TIF009	1900	*T.O.*
	TIF012	512	> 60470
	TIF014	3928	>52391
lua	LUA004	5816	*T.O.*
libxml2_fuzzer	XML001	>56243	>69775
	XML002	>75190	*T.O.*
	XML003	17662	60
	XML009	1682	> 50869
	XML012	>78619	*T.O.*
xmllint	XML001	>48474	*T.O.*
	XML002	>78131	*T.O.*
	XML009	8879	>80064

TABLE 10: P-values of the Mann-Whitney U test for the experiments presented in Table 2.

Program & Bug ID		**Titan**	**FishFuzz**	**AFLGo**	**SelectFuzz**	**AFL**	**MOPT**	**AFL++**
libpng_fuzzer	PNG001	1.00	1.00	0.97	1.00	0.86	0.86	1.00
	PNG006	0.18	0.18	0.18	0.18	0.18	0.18	0.99
	PNG007	<0.05	0.39	0.21	0.98	0.56	<0.05	<0.05
sndfile_fuzzer	SND001	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	SND006	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	SND007	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	SND017	<0.05	<0.05	<0.05	<0.05	<0.05	0.78	<0.05
	SND020	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	SND024	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
tiff_fuzzer	TIF002	<0.05	<0.05	0.01	0.14	0.92	0.08	0.14
	TIF008	0.01	0.21	0.42	<0.05	0.87	0.08	0.01
	TIF012	<0.05	<0.05	0.96	0.78	0.95	0.01	<0.05
	TIF014	<0.05	<0.05	0.010	<0.05	0.81	0.73	<0.05
tiffcp	TIF002	<0.05	<0.05	0.26	0.75	0.13	<0.05	0.018
	TIF005	0.08	0.08	0.08	0.08	0.08	0.08	0.99
	TIF006	0.21	0.12	0.71	0.96	0.89	0.71	0.99
	TIF008	0.18	0.18	0.18	0.18	0.90	0.18	0.79
	TIF009	<0.05	<0.05	<0.05	0.06	<0.05	<0.05	<0.05
	TIF012	<0.05	<0.05	0.07	0.06	0.06	<0.05	0.13
	TIF014	0.95	0.74	0.96	0.66	0.97	0.98	0.95
lua	LUA004	<0.05	0.66	<0.05	<0.05	0.60	0.21	<0.05
libxml2_fuzzer	XML001	<0.05	<0.05	<0.05	<0.05	<0.05	0.48	0.99
	XML002	0.08	0.92	0.08	0.08	0.48	0.08	0.08
	XML003	<0.05	0.29	0.52	0.80	0.31	<0.05	0.99
	XML009	<0.05	<0.05	0.31	0.58	0.99	0.99	0.96
	XML012	0.08	0.08	0.84	0.48	0.99	0.91	0.48
xmllint	XML001	<0.05	0.07	0.23	0.60	0.97	0.74	0.15
	XML002	0.18	0.71	0.18	0.18	0.71	0.50	0.50
	XML009	0.39	0.45	0.93	0.81	0.99	0.99	0.99
pdf_fuzzer	PDF002	0.08	0.08	0.08	0.08	0.08	0.08	0.08
	PDF004	0.08	0.08	0.08	0.08	0.29	0.08	0.08
	PDF010	0.59	0.36	0.36	0.14	0.07	<0.05	<0.05
	PDF011	0.34	0.08	0.34	0.62	0.08	0.71	0.34
	PDF018	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	PDF019	0.08	0.08	0.08	0.08	0.29	0.08	0.48
	PDF021	1.00	1.00	1.00	1.00	1.00	1.00	0.99
pdftoppm	PDF002	<0.05	<0.05	<0.05	<0.05	<0.05	0.13	<0.05
	PDF006	1.00	1.00	1.00	1.00	1.00	1.00	0.97
	PDF010	0.99	0.27	0.24	0.59	0.59	0.88	<0.05
	PDF011	0.06	<0.05	0.30	<0.05	0.08	0.38	0.07
	PDF018	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	PDF019	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
pdfimages	PDF002	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	PDF003	0.07	<0.05	0.52	<0.05	0.77	0.63	0.40
	PDF008	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	PDF011	0.54	<0.05	<0.05	<0.05	0.19	0.94	0.33
	PDF018	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	PDF019	<0.05	<0.05	0.08	<0.05	<0.05	0.86	<0.05
	PDF021	0.18	0.18	0.71	0.50	0.50	0.18	0.98
sqlite3_fuzz	SQL002	∅	<0.05	0.34	0.99	0.93	<0.05	0.06
	SQL003	∅	0.50	0.50	0.18	0.50	0.18	0.18
	SQL012	∅	0.97	0.93	0.67	0.99	0.08	0.81
	SQL013	∅	0.98	0.98	0.86	0.99	1.00	0.97
	SQL014	∅	<0.05	<0.05	0.29	<0.05	<0.05	<0.05
	SQL015	∅	0.97	0.86	0.86	0.97	1.00	1.00
	SQL018	∅	0.37	0.052	0.93	0.94	<0.05	0.24
	SQL020	∅	0.73	0.29	0.25	0.94	<0.05	0.34
asn1	SSL001	∅	<0.05	<0.05	∅	0.11	<0.05	0.99
x509	SSL009	∅	<0.05	<0.05	∅	<0.05	<0.05	<0.05
server	SSL020	∅	<0.05	<0.05	∅	<0.05	<0.05	<0.05
exif	PHP004	∅	<0.05	0.32	∅	<0.05	0.96	<0.05
	PHP009	∅	<0.05	<0.05	∅	<0.05	<0.05	<0.05

```
{"bug_type":"NULLPTR_DEREFERENCE","qualifier":"`newimage` could be null (from the call to `jas_image_create0()` on line 213) and is dereferenced in call to
`jas_image_growcmpts()`.","severity":"ERROR","line":214,"column":6,"procedure":"jas_image_copy","procedure_start_line":208,"file":"jas_image.c","bug_trace":[
{"level":1,"filename":"src/libjasper/base/jas_image.c","line_number":213,"column_number":13,"description":"in call to `jas_image_create0`"},
{"level":2,"filename":"src/libjasper/base/jas_image.c","line_number":190,"column_number":16,"description":"in call to `jas_malloc`"},
{"level":3,"filename":"src/libjasper/base/jas_malloc.c","line_number":106,"column_number":9,"description":"in call to `malloc (null case)` (modelled)"},
{"level":3,"filename":"src/libjasper/base/jas_malloc.c","line_number":106,"column_number":9,"description":"is assigned to the null pointer"},
{"level":3,"filename":"src/libjasper/base/jas_malloc.c","line_number":106,"column_number":2,"description":"returned"},
{"level":2,"filename":"src/libjasper/base/jas_image.c","line_number":190,"column_number":16,"description":"return from call to `jas_malloc`"},
{"level":2,"filename":"src/libjasper/base/jas_image.c","line_number":190,"column_number":8,"description":"assigned"},
{"level":2,"filename":"src/libjasper/base/jas_image.c","line_number":190,"column_number":8,"description":"taking \"then\" branch"},
{"level":2,"filename":"src/libjasper/base/jas_image.c","line_number":191,"column_number":3,"description":"is assigned to the null pointer"},
{"level":2,"filename":"src/libjasper/base/jas_image.c","line_number":191,"column_number":3,"description":"returned"},
{"level":1,"filename":"src/libjasper/base/jas_image.c","line_number":213,"column_number":13,"description":"return from call to `jas_image_create0`"},
{"level":1,"filename":"src/libjasper/base/jas_image.c","line_number":213,"column_number":2,"description":"assigned"},
{"level":1,"filename":"src/libjasper/base/jas_image.c","line_number":214,"column_number":6,"description":"when calling `jas_image_growcmpts` here"},
{"level":2,"filename":"src/libjasper/base/jas_image.c","line_number":772,"column_number":1,"description":"parameter `image` of jas_image_growcmpts"},
{"level":2,"filename":"src/libjasper/base/jas_image.c","line_number":777,"column_number":15,"description":"invalid access occurs here"}],
"key":"jas_image.c|jas_image_copy|NULLPTR_DEREFERENCE","hash":"cbc14cb2e8086fc44c9954e59f800ae8","bug_type_hum":"Null Dereference","extras":{}}
```

FIGURE 13: A null pointer dereference alarm detected in Jasper by Infer

B Ethics Considerations and Compliance with Open Science Policy

B.1 Ethics Considerations

The primary ethical consideration in our work is ensuring that the vulnerabilities we identify and the associated proof-of-concept exploits are responsibly disclosed. We have followed standard practices for responsible disclosure by notifying the affected parties of any vulnerabilities discovered during our experiments before any public release of the findings. This ensures that the software vendors have the opportunity to address the vulnerabilities before they are exposed to the public. Additionally, our work does not involve any experiments on human subjects, use of personal data, or other activities that could raise significant ethical concerns.

B.2 Open Science Policy Compliance

In compliance with USENIX Security's new open science policy, we commit to making our research results publicly available. The source code for our tool, Lyso, as well as the datasets generated during our experiments, will be released under an open-source license. This will allow other researchers and practitioners to replicate our findings, build upon our work, and further the advancement of software security. We believe that making these resources available will contribute to the transparency and reproducibility of research in the field of cybersecurity.

However, we recognize that there may be limitations regarding the release of certain details, especially concerning vulnerabilities that have not yet been fully mitigated by the affected vendors. In such cases, we will provide redacted versions of our datasets or delay the release of certain information until it is safe to do so. Any deviations from complete transparency will be clearly justified in our documentation, in alignment with the open science policy.

Overall, our commitment to open science aims to balance the need for transparency and reproducibility with the ethical responsibility to protect software systems and users from potential harm.