Titan : Efficient Multi-target Directed Greybox Fuzzing

Heqing Huang†, Peisen Yao‡✉, Chiu Hung-Chun† Yiyuan Guo†, Charles Zhang†
†The Hong Kong University of Science and Technology, China
‡Zhejiang University, China
†{hhuangaz, hchiuab, yguoaz, charlesz}@cse.ust.hk, ‡pyaoaa@zju.edu.cn

***Abstract*—Modern directed fuzzing often faces scalability issues when analyzing multiple targets in a program simultaneously. We observe that the root cause is that directed fuzzers are unaware of the correlations among the targets, thereby could degenerate into a target-undirected method. As a result, directed fuzzing suffers severely from efficiency when reproducing multiple targets.**

This paper presents Titan, which enables fuzzers to distinguish correlations among various targets in the program and, thus, optimizes the input generation to reproduce multiple targets effectively. Leveraging these correlations, Titan differentiates seeds' potential of reaching each target for the scheduling and identifies bytes that can be changed simultaneously for the mutation. We compare our approach to eight state-of-the-art (directed) fuzzers. The evaluation demonstrates that Titan outperforms existing approaches by efficiently detecting multiple targets, achieving a 21.4x speedup and requiring 95.0% fewer number of executions. In addition, Titan detects ten incomplete fixes, which cannot be detected by other directed fuzzers, in the latest versions of the benchmark programs with two CVE IDs assigned.

***Index Terms*—Directed fuzzing, Multi-target, Path correlation**

1. Introduction

Directed fuzzing is a highly effective method for improving the diagnostics efficiency of program behaviors by focusing on testing specific program locations. It has many important applications, such as testing vulnerability patches with regression [1], [2], reproducing crashes [3], [4], validating hundreds of static analysis reports [5], [6], and generating proofs-of-concept for various vulnerabilities [7]. However, software systems are becoming increasingly large and evolving at a high speed. As a result, each run of fuzzing tests often needs to address a large number of targets simultaneously. For instance, a static analyzer could report more than 1000 potential targets for developers to verify in a single version of the project [8].

When facing such a massive number of targets in the program, one potential option is to use additional computation resources as a trade-off for better reproduction speed, e.g., deploying multiple concurrent fuzzer instances for each target using separate CPU core [9]. However, this could become impractical for developers. For instance, to verify the 1000 targets mentioned in the previous example, It is expensive (>24000 CPU hours, >1000 cores) for developers to run each target separately on an individual core within a conventional 24-hour time budget.

Unfortunately, the state-of-the-art directed fuzzers cannot scale well in front of such a large number of targets. Most directed fuzzers [2], [10]–[14] do not support reproducing multiple targets simultaneously. Their success depends on designing fine-grained feedback from analyzing the single target more precisely. While certain approaches [15]–[18] have endeavored to address multiple targets simultaneously using general feedback, e.g., average control-flow distance, their effectiveness significantly diminishes when tackling multiple targets in comparison to their performance on singular targets. Thus, the challenge of efficiently fuzzing multiple targets remains. For example, we compared the time for reproducing the same number of targets in the Magma benchmark using the state-of-the-art directed fuzzer, AFLGo [15] (Section 5.1). Compared to reproducing targets separately using multiple parallel fuzzer instances, the reproduction time significantly increases by 3.2x when attempting to reproduce all 141 targets in a single fuzzer instance. This stark contrast underscores the limitations of existing approaches for effectively conducting simultaneous fuzzing across multiple targets. However, it is crucial to note that the collective CPU time required for separately reproducing all targets surpasses the corresponding time for reproducing them collectively within a single fuzzer instance by a factor of 3.6x, which highlights the urgent need to embrace a multi-target directed fuzzing approach.

We observe that one of the root causes of this challenge is that existing approaches are unaware of the correlations between the targets and, as a result, could degenerate to undirected fuzzing as the number of targets grows. Eventually, the fuzzers cannot effectively select the seeds and generate the inputs for multiple targets efficiently, which we refer to as the *synergy ignorance* problem.

On one hand, ignoring the correlations could result in the ineffective selection of seeds to reach specific targets. Specifically, existing efforts [15], [19] design various one-size-fits-all scores as the probability of reaching all targets, e.g., average control-flow distance to the targets [15]. However, because these scores do not distinguish the hard-to-

```
1 int foo(){
2   signed a,b,c,d = parse_input();
3   int flag;
4
5   if(a>5){
6     if(b>1){
7       flag=0;
8     }
9     else {
10      flag=1;
11      if(...) {...} //target-irrelevant
12    }
13    ...
14    if(flag){
15      target1();
16    }
17    else if(c>2){ //contradict to flag==1
18      target2();
19    }
20  }
21
22  if(d>3){
23    target3();
24  }
25 }
```

Figure 1: Motivating example.

reach targets, they could innately introduce bias in seed selection, lowering the probability of exploring targets. For example, suppose a fuzzer attempts to detect the three targets in Figure 1 with two seeds, $A(a,b,c,d)$: $(6,3,1,1)$ and $B(a,b,c,d)$: $(5,0,3,1)$. Existing fuzzers should use seed A to detect the three targets since A has the shortest average control-flow distance to the three targets. However, seed A may not be the best seed for reproducing target 2, as mutating A could be challenging to satisfy the condition at branches 14, $flag = 1$, for target 1.

On the other hand, being unaware of the correlations among different targets can result in massive redundant executions due to ineffective mutation. For example, targets 1 and 3 in Figure 1 are independent because the conditions for reaching them, $a > 5 \wedge b \leqslant 1$ and $d > 3$, are related to different input bytes. As the fuzzer is unaware of the path correlation, it cannot simultaneously mutate the input bytes for a, b, and d to approach targets 1 and 3, resulting in additional mutations for reproducing the two targets.

To tackle the *synergy ignorance* problem, our key insight is that the fuzzer can leverage the correlations between multiple targets' feasible conditions, i.e., independence and contradiction, to synergize input generation towards those targets. More specifically, if we obtain the solution space of the variables related to the given targets, e.g., $d > 3$ for target 3, the fuzzer can distinguish the correlations between the targets to precisely guide the seed scheduling and mutation, thereby efficiently reaching multiple targets.

- First, the contradiction correlation indicates whether the conditions for triggering these targets are in conflict. Thus, we can assign distinct priorities for the seeds satisfying the conditions of different targets. For example, the targets 1 and 2 have conflicting conditions on variable b (i.e., $b > 1$ and $b \leqslant 1$). Thus, the fuzzer can use seed A (which satisfies $b > 1$) for target 2 and seed B (which satisfies $b \leqslant 1$) for target 1 to improve the effectiveness of seed scheduling.
- Second, the independence correlation indicates that multiple input bytes could affect different targets independently. Therefore, the fuzzer can attempt to mutate these input bytes simultaneously to approach multiple targets, which reduces the execution needed to reach the targets. For example, in Figure 1, since a and d are independent, the fuzzer can mutate their values simultaneously for targets 1 and 3.

Based on these observations, we present a synergy-aware directed greybox fuzzing approach for efficiently reaching multiple targets only with one fuzzer instance. We first discover the correlations of different targets based on the approximation of their path conditions. We then determine the correlations among targets by checking whether their approximations conflict. As a result, the fuzzer can have precise feedback for multiple targets by identifying and using the correlations instead of relying solely on the control-flow distances. Using the new fine-grained feedback, we introduce two optimized input generation techniques: 1) A synergy-aware scheduling strategy that finds the seed potentially reaching the most targets with a fine-grained ranking of the seeds. 2) A multi-target-oriented mutation that generates inputs for approaching multiple targets with fewer executions needed.

We implement our approach as Titan, and compare it with a state-of-the-art directed fuzzer, AFLGo [15], as well as widely-used fuzzers AFL [20], AFLFast [21], and FairFuzz [22], on both real-world programs and the state-of-the-art fuzzing benchmark, Magma [23]. On average, Titan achieves a 21.4x speed improvement while using 95% fewer number of execution and triggers 2.31x more bugs than other fuzzers. Additionally, Titan outperforms other fuzzers by detecting ten incomplete fixes on the newest version of the benchmark programs, two of which have been assigned CVE IDs. Moreover, our inferred correlations can be integrated with other directed fuzzers, such as Beacon [12], to speed up fuzzing multiple targets, accelerating vulnerability reproduction by an average of 1.5x. Titan is available at our GitHub repository.

To sum up, we make the following contributions:

- We design a cheap and precise static analysis for inferring the synergy to reach multiple testing targets, enabling us to mitigate the synergy ignorance issue by distinguishing the correlations among the targets.
- We propose a new feedback mechanism for multi-targets-directed fuzzing based on the path condition rather than the control-flow distance to schedule the seed with better precision.
- We design a multi-target-oriented mutation strategy to efficiently reach multiple targets in parallel.
- We provide empirical evidence that our approach is more efficient and effective than the state-of-the-art (directed) fuzzers and has the potential to improve the performance of other directed fuzzers.

2. Background and Motivation

This section surveys recent directed greybox fuzzers (Section 2.1) and summarizes the challenges we attempt to resolve in this paper (Section 2.2).

2.1. Directed Grey-Box Fuzzing

Existing directed greybox fuzzing aims to test a target of interest in a program efficiently. The intuitions proposed in recent efforts can be categorized into two aspects: *improving directness by finding the most potential seed closer to the target* and *avoiding energy waste by rejecting inputs that cannot reach the targets*.

Improving the directness. Recent mainstream of directed greybox fuzzers prioritizes testing inputs so that we can run inputs "closer" to the target in a higher priority. To this end, multiple distance metrics have been proposed:

- AFLGo [15]: It takes the first step to define directed fuzzing. It defines the distance of a testing input to the target basic block as the average distance between a block B and the target, where B ranges over the blocks reached by the executed path of the input.
- Hawkeye [10]: It optimizes the distance metric by considering call trace similarity, with the intuition that a vulnerability is triggered by a sequence of call instructions rather than a single program point.
- CAFL [11]: It further improves the metric's precision by observing that multiple predecessors should be reached before triggering a crash. Thus, it sets all the call sites from the crash dump of a vulnerability and requires the fuzzer to reach them in order.
- MC2 [18]: It transforms directed fuzzing as a Monte-Carlo counting model and uses the execution frequency of each branch to approximate the difficulties in reaching the targets and prioritize the seed with the lowest difficulties in reaching the targets.

While existing efforts have improved the precision of feedback for directed fuzzing towards a specific target, this feedback is uniquely designed for each target and cannot assist the fuzzer in efficiently reaching multiple targets.

Rejecting the infeasible inputs. Since there is no guarantee for such directness to find the most promising seed, another trend of existing efforts proposes to reject the inputs that are not likely to reach the targets.

- FuzzGuard [13]: It leverages an observation that reproducing a bug needs to reach the targets. Therefore, it trains a classifier as a predictor to filter the testing inputs so that inputs with a higher probability of reaching the targets can be executed on a higher priority.
- Beacon [12]: It further refines the rejection in a provable manner since reproducing a bug should satisfy the path conditions. It uses static analysis to infer the precondition for reaching the target and prunes inputs whenever executions violate the precondition.

Even though rejecting infeasible paths could be effective, it is orthogonal to our problem discussed in this paper. Still, we will demonstrate the possibility of integrating this approach with our proposed technique in Section 5.4.

2.2. Problem and Motivation

While existing efforts improve directed fuzzing to reach one target faster, the *synergy ignorance* problem could still hinder the performance from efficiently reproducing multiple vulnerabilities.

On the one hand, some targets could become hard to reach if seeds are selected based on the average control-flow distance to all targets [10], [15]. In Figure 1, suppose we have three seeds, $A(a,b,c,d)$: $(6,3,1,1)$, $B(a,b,c,d)$: $(5,0,1,1)$, and $C(a,b,c,d)$: $(3,3,3,1)$. Existing approaches find the closest seed to the three targets based on control-flow distance, e.g., the average harmonic distance between the blocks in the executed traces and the targets [15]. Thus, the fuzzer should choose the seed reaching the branch at Line 16, i.e., $A(a,b,c,d)$: $(6,3,1,1)$. However, mutating seed A could not efficiently satisfy the tight conditions at Line 6, b:$[0,1]$, for target 1. If we set a higher priority for seed C, the fuzzer could reach both targets 1 and 2 more easily by making $a > 5$. Essentially, the ineffective seed selection is because they are unaware of which targets the seeds could attempt to cover more efficiently, thereby wasting time approaching all of the targets blindly.

On the other hand, no existing directed fuzzers optimize the mutation towards multiple targets. One of the main reasons is that determining the exact relationships between the input bytes and targets is stunningly challenging. In the end, existing fuzzers have to be "myopic" in the mutation, only attempting to cover new branches faced so far [16]. However, the fuzzers may waste time covering unnecessary branches. For example, suppose seed A is chosen for target 2, and the branch at Line 11 is uncovered. Existing fuzzers are still unsure of which branch should be covered to reach the target faster. Therefore, covering the target-irrelevant branch at Line 11, which is reachable to the targets in the control flow, could introduce meaningless efforts and hinder the efficiency of directed fuzzing.

Our observation. The main problem preventing existing directed fuzzers from efficiently covering multiple targets is that they do not distinguish the correlations between different targets, primarily in the form of path condition overlapping, conflicting, and independence, as illustrated in Figure 2.

Overlapping and conflicting conditions for multiple targets guide the seed schedule. In Figure 1, the condition $a > 5$ is an overlapping condition for both targets 1 and 2 because reaching them both requires the condition to hold. Thus, the overlapping condition can direct the fuzzer to cover the true branch at Line 5, e.g., prioritizing the seeds that are more likely to cover the branch. On the other hand, reaching targets 1 and 2 have mutually exclusive demands on the condition of b, namely $b \leqslant 1$ and $b > 1$, which we refer to as a conflicting condition for the two targets. As a result, we can identify the seeds satisfying $b \leqslant 1$ as more

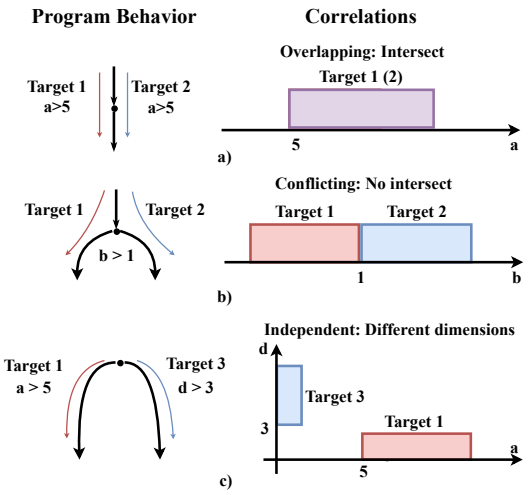

Figure 2: Potential correlations among the targets.

likely to cover target 1. The seeds satisfying $b > 1$ should focus more on covering target 2.

Independent conditions for multiple targets facilitate efficient mutation. In contrast to the "myopic" way of mutation, directed fuzzing can be more "far-sighted" if it is aware of the independence among the targets. In Figure 1, the condition $d > 3$ at Line 22 only influences the reachability of target 3 but does not affect whether targets 1 and 2 are reached. Similarly, condition $a > 5$ affects reaching targets 1 and 2 but not target 3. As a result, we can mutate the independent bytes simultaneously, which could help approach multiple targets with fewer executions. If we simultaneously mutate d with other bytes, such as a for targets 1 and 2, the fuzzer can attempt to approach all three targets in a single execution because the conditions for d and a do not influence each other.

From the above observation, we conclude that the relations of the conditions indicate the potential of detecting multiple targets. Directed fuzzing can efficiently reach multiple targets if we can precisely reason about those relations. First, the fuzzer can more accurately select seeds for multiple targets by using conflict correlations to differentiate the difficulties of reaching different targets, e.g., targets 1 and 2. Second, the fuzzer can generate inputs more effectively for multiple targets by using their independent correlations, e.g., simultaneously mutating the independent bytes b and d for targets 1 and 3.

However, to efficiently tackle multiple targets, there are two major challenges in utilizing the correlations between different targets to guide the fuzzer.

Challenge 1. How to schedule multiple targets for the seed effectively? To effectively select seeds, the fuzzer should be aware of the correlation among different targets. Although the correlations can reveal the relations of reaching the targets, we still need to justify the seed potentials for each target. Therefore, we need to design a new seed scheduling strategy for assessing the potential of a new seed with respect to its influenced targets.

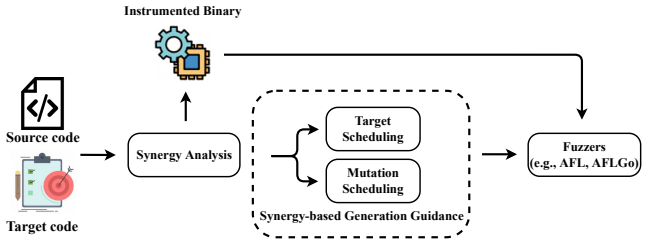

Figure 3: Workflow of Titan.

Challenge 2. How to generate the inputs for multiple targets efficiently? To generate inputs for multiple targets, the fuzzer should mutate more input bytes correlated with multiple targets to reduce the number of executions required. However, there may be a large number of relevant bytes for multiple targets. Therefore, we need to efficiently identify the input bytes for different targets and effectively prioritize the bytes for the mutation.

3. Titan in a Nutshell

Based on the observation in Section 2.2, we present Titan, a directed fuzzer to reach multiple targets efficiently. At a high level, our approach consists of two stages. First, we use a static analyzer to infer the correlations among multiple targets based on their path conditions. Second, we design a synergy-aware fuzzer that effectively generates inputs for multiple targets. Crucially, the correlations inferred by the static analysis provide fine-grained feedback for both seed scheduling and mutation.

3.1. Static Synergy Inference

Given a set of target program points, we design a two-phase static analysis: inferring the preconditions for reaching each target, and computing the correlations for distinguishing multiple targets.

Precondition inference. First, we infer the preconditions as the necessary conditions, e.g., $a > 5$ in the motivating example, of program variables for reaching the target code. To facilitate computing the condition, we leverage the interval domain [24] to over-approximate the feasible search space of the path condition for reaching the targets, similar to existing works [11], [12]. Formally, the static analysis infers the following information:

Definition 3.1. (Multi-target Reachability Summary) Given a set $T \subseteq N$ of target program points, the multi-target reachability summary is $\bigcup_{t \in T} S_t$ where $S_t(n) \subseteq Var \times Interval$, and $(v, itv) \in S_t(n)$ suggests that in order to reach the target program point t, the variable v must fall in the interval itv before executing the program point $n \in N$.

Example 3.1. Consider the example shown in Figure 1. Our static analyzer infers the multi-target reachability summary as follows:

$$S_{t_1}(l_5) = flag : [1,1], a : (5,+\infty), b : [0,1]$$
$$S_{t_2}(l_5) = flag : [0,0], a : (5,+\infty), b : (1,+\infty), c : (2,+\infty)$$
$$S_{t_3}(l_5) = d : (3,+\infty)$$

where l_i denotes the program location before Line i and S_{t_1}-S_{t_3} corresponds to targets 1-3, respectively.

Correlation inference. We observe that the inferred variable ranges for different targets can have many correlations. For instance, consider the multi-target reachability summary in the above example. Reaching targets 1 and 2 both require the same range for the variable a, but they have an exclusive demand on the range of the variable b. Formally, we capture the observation below:

Definition 3.2. (Correlation among Multiple Targets) Suppose the static analysis result is $\bigcup_{t\in T} S_t$. Given a variable v and a program point l, we define the independence, conflict, and overlap relations based on whether the summaries intersect as follows:

$$\begin{aligned} independent(v,l,t_1,t_2) &\stackrel{\text{def}}{=} (v \in S_{t_1}(l) \wedge v \notin S_{t_2}(l)) \\ &\quad \vee (v \in S_{t_2}(l) \wedge v \notin S_{t_1}(l)) \\ conflict(v,l,t_1,t_2) &\stackrel{\text{def}}{=} S_{t_1}(l)(v) \cap S_{t_2}(l)(v) = \varnothing \\ overlap(v,l,t_1,t_2) &\stackrel{\text{def}}{=} S_{t_1}(l)(v) \cap S_{t_2}(l)(v) \neq \varnothing \end{aligned}$$

Then, we can infer the correlations among multiple targets from the multi-target reachability summary.

Example 3.2. Consider Figure 1. Targets 1 and 2 conflict at variables b and $flag$ since their abstractions have no intersections, and overlap at variable a as their abstractions are the same. We also mark the condition independent of others, i.e., variable c is uniquely constrained for target 2. Overall, we formally define the correlations as follows:

$$overlap(a,l_5,t_1,t_2), conflict(b,l_5,t_1,t_2)$$
$$independent(c,l_5,t_1,t_2), independent(d,l_5,t_1,t_3)$$
$$independent(d,l_5,t_2,t_3)$$

Usage of the synergy through instrumentation. We can instrument the program with the computed correlations to provide fine-grained feedback for fuzzing. Specifically, when reaching a program point n, we check whether the variable v defined before n may affect the reachability of the targets, e.g., t_1, t_2, based on the correlations in Definition 3.2. For example, in Figure 1, suppose the execution reaches Line 5. We can use the correlation, $conflict(b,l_5,t_1,t_2) = true$, to know that the variable b defined at Line 2 influences targets 1 and 2. Thus, the fuzzer is aware that the current seed is easier to reach one of the targets (1 or 2) because the seed already satisfies the condition, which further helps determine the seed's potential and how to mutate the seed.

3.2. Synergy-guided Directed Fuzzing

With the inferred correlations, the fuzzer can have fine-grained feedback as guidance for both seed scheduling and mutation. Specifically, we aim to fuzz multiple targets simultaneously by tackling the two challenges mentioned above respectively:

Synergy-aware seed scheduling. To address Challenge 1, a directed fuzzer should know the correlations among the targets to select the seeds precisely. In Titan, we utilize the correlations to (1) identify which targets should be focused on by a given seed and (2) measure the difficulties of reaching them.

First, we identify the targets a seed should consider based on the intuition illustrated in Figure 2. Specifically, if the variables reached belong to the overlap or independent correlations (as shown in a) and c) in Figure 2), we consider the seed could reach these targets because the intersected search space indicates that they could be covered simultaneously. We use the conflict correlation to filter the targets for the current seed: If a target's precondition is not satisfied by the current seed, we can temporarily exclude that target from consideration.

Example 3.3. Consider two seeds $A(a,b,c,d)$: $(6,3,1,1)$ and $B(a,b,c,d)$: $(6,0,1,1)$. We propose that the fuzzer should use them for targets 1 and 2, respectively, according to the conflicts in the correlation, $conflict(b,l_5,t_1,t_2)$ in Figure 1. We recognize that A is more likely to trigger target 1 whereas B is more likely to trigger target 2, based on the values of b observed in the executions of the seeds, which satisfy distinct conditions in the conflict correlation.

Second, we need to prioritize the focused targets for the seeds because reaching each target could vary in difficulty. Specifically, we regard reaching a target as satisfying its conditions on multiple variables in the correlations. The seed input whose value satisfies the conditions for most variables should be prioritized. Therefore, we analyze the execution feedback to check the number of preconditions met during the executions. For example, in the previous example, we prioritize B over A since B satisfies the preconditions of two variables, a and b for target 2, while A only satisfies the precondition of one variable, a, for target 1. Therefore, with a fine-grained feedback measurement, we can select the most promising seed to cover multiple targets.

Multi-target-oriented mutation. To solve Challenge 2, we mutate the seed for approaching multiple targets at each run to reduce redundant executions. Recall that some bytes could be mutated simultaneously to reduce the number of executions needed for reaching multiple targets, i.e., a and d in Figure 1. To achieve this, we leverage the independent correlation to mutate multiple bytes related to independent variables. Specifically, we first dynamically infer the input bytes that can influence the values of the variables in the independent correlations using inference-based taint analysis [22], [25], [26]. Then, we mutate the bytes that influence the independent variables respectively, enabling the fuzzer to approach multiple targets with fewer mutations.

Algorithm 1 Static Reachability Inference

Input: An inter-procedure control flow graph, G
Input: A set of targets, T
Output: Multi-target reachability summary $\bigcup_{t \in T} S_t$

1: **procedure** ABSTRACTIONINFERENCE(G, T)
2: $res \leftarrow \varnothing$
3: **for all** $t \in T$ **do**
4: $S_t \leftarrow \varnothing$
5: **while** $v' \xrightarrow{stmt} v \in ReverseDFS(G, t)$ **do**
6: $S_t[v'] = S_t[v'] \bigsqcup transfer(S_t[v], stmt)$
7: **end while**
8: $res \leftarrow res \cup S_t$
9: **end for**
10: **return** res
11: **end procedure**

Example 3.4. For the seed, $A(a, b, c, d)$: $(6, 3, 1, 1)$, we can infer that the input bytes for variables used in the branch at Lines 5 and 22 are controlled by different bytes from a and d. Thus, we inform the fuzzer to mutate them simultaneously, which increases the probability of reaching targets 2 and 3 with less execution needed than mutating them separately.

Furthermore, as the number of targets considered increases, input bytes could influence these targets differently. Hence, to expedite the process of reaching multiple targets, we can further prioritize these input bytes for mutation.

4. Methodology

In this section, we first present our static analysis algorithm for inferring the multi-target reachability summary (Definition 3.1) in Section 4.1. We then detail how to leverage the correlation inferred from the summary to improve seed scheduling in Section 4.2 and generate inputs for multiple targets in Section 4.3.

4.1. Static Synergy Inference

Algorithm 1 presents our static analysis for inferring the preconditions to reach multiple targets. At a high level, the analysis traverses the inter-procedural control flow graph G backward, starting from the target location t (denoted by $ReverseDFS(G, t)$), during which it maintains and updates the preconditions for different program locations to reach the target t. More precisely, for each program point v', the algorithm computes an abstraction of the precondition for reaching t by iteratively applying the semantic transfer function $transfer$ of each program statement $stmt$ (Lines 5-7). We refer the reader to existing works [12] for the design and implementation of the transfer functions.

Based on the computed multi-target reachability summary, we can obtain the correlations among multiple targets for a seed according to Definition 4.1.

Definition 4.1. (Correlations of a seed) Given a seed, S, suppose S reaches a set of program locations $(v_1, l_1), ..., (v_n, l_n)$, where variable, v_i, is defined at location, l_i. The set of correlation, $C(S)$, of the seed S is:

$$C(S) \stackrel{\text{def}}{=} \{r(v_i, l_i, t_j, t_k) | \\ i \in (1, n), r \in \{independent, conflict, overlap\}\}$$

where (t_j, t_k) is a pair of targets.

Algorithm 2 Influential Targets Inference

Input: A seed S,
Input: A set of targets T
Output: Targets should be considered by the seed, T_{res}

1: **procedure** TARGETCLUSTERING(S, T)
2: $T_{res}, T_{conf}, T_{ind} \leftarrow \varnothing$
3: $C(S) \leftarrow getCorrelation(s)$
4: **for all** $correlation \in C(S)$ **do**
5: **if** $correlation == overlap(v, l, t_i, t_j)$ **then**
6: $T_{res} \leftarrow T_{res} \cup \{t_i, t_j\}$
7: **end if**
8: **if** $correlation == conflict(v, l, t_i, t_j)$ **then**
9: **if** concrete value of v satisfies $S_{t_i}(l)(v)$ **then**
10: $T_{res} \leftarrow T_{res} \cup \{t_i\}$
11: $T_{conf} \leftarrow T_{conf} \cup \{t_j\}$
12: **end if**
13: **end if**
14: **if** $correlation == independent(v, l, t_i, t_j)$ **then**
15: $T_{ind} \leftarrow T_{ind} \cup \{(t_i, t_j)\}$
16: $T_{res} \leftarrow T_{res} \cup \{t_i, t_j\}$
17: **end if**
18: **end for**
19: $T_{res} \leftarrow T_{res} \backslash T_{conf}$
20: $T_{res} \leftarrow T_{res} \cup \{t' | \, t \in T_{res}, (t, t') \in T_{ind}\}$
21: **return** T_{res}
22: **end procedure**

4.2. Synergy-aware Seed Scheduling

To find the seed that could reach the most targets, we first determine which targets should be considered by each seed based on the inferred correlations. Then, we measure the potential for each seed to reach these targets. Since seeds discover a different part of the program, their focused targets could vary based on the reached parts of the program. Therefore, we need to adaptively determine the related targets for a seed S according to its reached correlations $C(S)$.

Example 4.1. If a seed does not reach Line 6 in Figure 1, no conflict relation for targets 1 and 2 is established. In this case, the fuzzer can use those seeds, e.g., seeds reaching Line 5 with $a < 5$, to approach the two targets. However, if the seeds reach Line 6, e.g., $A(a, b, c, d)$: $(6, 3, 1, 1)$ and $B(a, b, c, d)$: $(6, 0, 1, 1)$, the fuzzer should be aware of the conflict correlation and distinguish the seeds for different targets, i.e., seed B not satisfying the condition $b > 2$ at Line 6 needs more mutations to reach target 1 than seed A.

Determine the targets influenced by seeds. As mentioned in the previous section, our basic intuition is that a

Table 1: Example of how Titan determines the targets influenced by seed through checking the reached correlation during the execution. We use a seed, (a, b, c, d): $(6, 3, 1, 1)$ on the program in Figure 1.

Line	Condition	Abstractions	Values	Influence Targets
5	$a > 5$	a_1, a_2: $[6, +\infty)$	$a = 6$	$T_{res} = \{1, 2\}$
6	$b > 1$	b_1: $(-\infty, 1]$, b_2: $[2, +\infty)$	$b = 1$	$T_{conf} = \{2\}$
14	$flag \neq 0$	$flag_1$: $[1, +\infty]$, $flag_2$: $[0, 0]$	$flag = 0$	$T_{conf} = \{2\}$
22	$d > 3$	d_3: $[4, +\infty)$	$d = 1$	$T_{ind} = \{(t_1, t_3), (t_2, t_3)\}$

seed should focus on targets with more satisfied conditions but not solve the condition for all targets simultaneously. Therefore, not all targets should be considered by a seed. Since the conflict correlation indicates the contradiction of the two targets, we regard one of them as irrelevant to the current seed during the scheduling.

Specifically, Algorithm 2 determines each seed's affected targets, T_{res}, according to its correlation. We first obtain the correlation of the seed, $C(s)$, to decide the targets this seed should focus on (Line 3). Based on correlations in $C(s)$, we distinguish the targets according to whether they are overlapping, conflicting, or independent. If the targets are overlapped or independent of each other (Lines 5-7, Lines 14-17), they can be fuzzed synergistically, i.e., they should be considered by the current seed (Lines 6 and 16). On the other hand, if two targets are conflicting at a program location (Lines 8-13), the seed should only consider the target whose condition is satisfied by the intermediate values and discard the other (Lines 9-12, assuming t_i is the satisfied target). Additionally, we regard any target that is independent of targets in T_{res} as a potential target of the seed, but remove any conflicting targets (Lines 19-20).

Example 4.2. We use the seed $A(a, b, c, d)$: $(6, 3, 1, 1)$ in Figure 1 to demonstrate the process in Table 1. We record the correlations discovered during executing the seed A to determine which target seed A should focus. Specifically, the overlap correlation at Line 5 adds targets 1 and 2; the conflict correlation at Lines 6 and 14 filters target 1; the independent correlation at Line 22 adds target 3. Therefore, we can use seed A to reach targets 2 and 3.

Determine the potential to reach the influenced targets. Besides identifying the affected targets for each seed, we also need to determine the potential to reach these targets and prioritize them for fuzzing. To this end, we propose to leverage the satisfied preconditions used in the correlations instead of using control-flow-based distance to have better precision for the measurement. Intuitively, the more preconditions in the summary that a target has to satisfy, the more challenging it will be to reach the target.

Example 4.3. The seed A satisfying $a \leqslant 5 \wedge b > 2 \wedge c > 2$ could reach targets 1 and 2 faster than the seed B with $a > 5 \wedge b \leqslant 2 \wedge c <= 2$ because the fuzzer only needs to mutate a. In contrast, the methods using control-flow distance would favor seed B, because covering the true branch of Line 5 makes the seed closer to the targets.

Thus, we measure the potential of a target t to be reached by a seed (denoted $potential(t)$) using the proportion of the precondition satisfied in the execution:

$$potential(t) = \begin{cases} 0 & \text{if } t \text{ has been triggered} \\ \frac{n(t)}{N(t)} & otherwise \end{cases} \quad (1)$$

where $n(t)$ is the number of satisfied preconditions for t observed when executing the seed input, and $N(t)$ is the total number of preconditions inferred statically for t. The higher the potential value of a seed for a given target, the better the seed is for reaching the target.

With the potential for each target, we accumulate them for a seed s to measure the likelihood of reaching them defined as:

$$potential(s) = \sum_{t \in T_{res}} p(t) \quad (2)$$

Example 4.4. The potential of seed A in the previous example to reach target 2 is $\frac{3}{4} = 0.75$ since there are four preconditions for target 2 (a, b, c and $flag$) and three of them are satisfied by A. The overall potential for seed A is $potential(t_2) + potential(t_3) = 0.75 + 1 = 1.75$.

Remark 1. To effectively select seeds for multiple targets, we differentiate the seeds for different targets using the correlations rather than designing a one-size-fits-all score for all targets as in existing works [10], [11], [15]. Furthermore, we utilize the correlations to measure the potential of reaching the targets regarding the solution space. Admittedly, quantifying the exact solution space is stunningly challenging for real-world programs [27]–[29]. We use the approximation of the solution space from existing work [12], which, albeit may suffer from imprecision, is more fine-grained than the control-flow-based distance.

4.3. Multi-target-oriented Mutation

To efficiently generate inputs for multiple targets, we propose mutating multiple bytes simultaneously for different targets. Specifically, we first infer the bytes influencing different targets to determine which can be mutated simultaneously. We then prioritize mutating those bytes that could reach more targets.

Inferring the influenced bytes. Intuitively, the variables in the independent correlations could be changed simultaneously to approach different targets. Therefore, if these variables' values depend on different bytes, the fuzzer can mutate them simultaneously to generate inputs for multiple targets. Based on this intuition, we formally define the bytes that can be simultaneously mutated as follows:

Definition 4.2. (Independent Byte) Given a seed S and a target program point t, Let B_t be the set of bytes that affects the reachability of t, i.e., the set of bytes controlling the values of variables in $C(S)$. We define the bytes, B_{ind} that are independent based on whether they are uniquely served for a target as follows:

$$B_{ind}(t_i, t_j) \stackrel{\text{def}}{=} (B_{t_i} \cup B_{t_j}) - (B_{t_i} \cap B_{t_j})$$

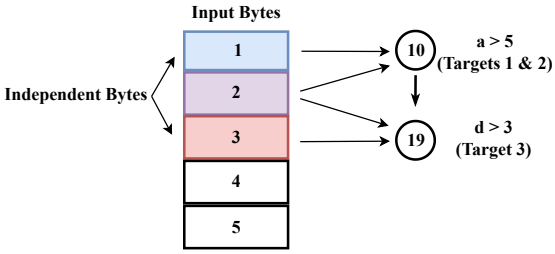

Figure 4: The intuition of mutating the independent bytes to improve the efficiency of input generation. If bytes 1 and 3 are mutated simultaneously, fuzzers could satisfy both conditions for three targets with fewer executions.

Independent bytes have the potential to be mutated simultaneously for multiple targets. For example, in Figure 4, suppose that that bytes 1 and 2 control the conditions at Line 5, and bytes 2 and 3 control those at Line 22 in Figure 1. Based on Definition 4.2, the independent bytes are 1 and 3. Simultaneously mutating these bytes increases the possibility of satisfying both conditions for multiple targets, thereby reducing the number of executions to reach them.

However, precisely identifying the independent bytes, B_{ind}, for each seed can be computationally expensive. Thus, we adapt the *inference-based taint analysis* [22], [25], [30] to estimate the independent bytes. The intuition is that mutating the independent bytes should not change the values of variables in the correlations for different targets. Unlike existing works that examine every input byte for one target at a time, we employ a lazy approach that infers the bytes only for the variables in the correlations. Specifically, we gradually remove the bytes that cannot be independent for different targets. This reduces substantial runtime overhead when the number of targets increases in conventional methods.

We start by assuming that all bytes are related to the targets and record the variable values in the correlations when preserving the seeds. During the inference-based taint analysis, we track the bytes that simultaneously modify the variable values of the targets in T_{ind} identified by Algorithm 2 as non-independent bytes.

Example 4.5. Suppose we mutate bytes 2 of the seed shown in Figure 4. In that case, the new input could simultaneously modify the values of two variables, a and d, for multiple targets. Therefore, byte 2 cannot be an independent byte and should not be modified simultaneously with other bytes during fuzzing.

Mutating bytes for multiple targets. While multiple targets exist in the program, each byte could influence a different number of targets. Therefore, instead of mutating each byte with equal quality, we propose prioritizing the bytes that can influence the most non-triggered targets, which could make the input generation more effective towards multiple targets.

Our basic intuition is to prioritize the bytes according to how many targets it can influence. This is achieved through an adaptive approach, where we first infer the bytes for each target and then mutate the input bytes for its focused targets indicated by the T_{res} defined in Section 4.2. For each byte, b, of seed, S, we measure the number of its influenced targets that have not been triggered as $inf(b)$. We choose the bytes with probability, p, defined as follows:

$$p(b) = \frac{inf(b)}{\sum_{b_i \in S} inf(b_i)} \tag{3}$$

We then mutate the independent bytes of the chosen one simultaneously by querying the targets in T_{ind}.

Example 4.6. The input byte 1 in Figure 4 can be chosen with a probability of $\frac{2}{2+3+3} = 0.25$, while byte 2 influencing more targets could be chosen with a higher probability of $\frac{3}{2+3+3} = 0.375$. Since targets 1 and 3 are independent according to T_{ind}, Titan could mutate byte 3 for target 3 if byte 1–on which target 1 depends–is chosen.

Remark 2. Our approach has a flavor of dynamic taint analysis as we infer the relevant bytes of the targets. Different from the existing usage [22], [25], [26], we lazily infer the bytes to reduce the overhead because we do not need to go through every byte for every target. Meanwhile, our approach helps to mitigate the under-taint issue [31] in the conventional methods, as we will not remove the bytes that do not change the intermediate values toward the targets. Note that we do not claim to optimize the concrete mutation operators as previous efforts [22], [25], [26], [32], but instead guide the mutation by finding the bytes that can be mutated simultaneously with higher potential. Overall, apart from the original bytes chosen by the fuzzer, we pick extra bytes and conduct mutations to increase the probability of approaching multiple targets in one run.

5. Evaluation

We implemented Titan, a greybox fuzzer with a correlation analyzer and an instrumentation component, based on LLVM [33]. As shown in Figure 3, we first compile the input source code to LLVM bitcode, on which the synergy analysis and the instrumentation for runtime feedback are performed. After instrumentation, the LLVM bitcode is compiled into an executable binary, which can be integrated with various fuzzing engines, such as AFLGo [15] and Beacon [12]. By default, we use AFL 2.57 [20] as the fuzzing engine.

In this section, we evaluate the effectiveness of Titan by investigating the following research questions:

- **RQ1**: How efficiently can Titan reproduce the vulnerabilities compared with other fuzzer? (Section 5.1)
- **RQ2**: How effectively do the correlations inferred by Titan help reproduce the vulnerabilities? (Section 5.3)
- **RQ3**: How effectively can Titan help other directed fuzzing for multiple targets? (Section 5.4)
- **RQ4**: What is the runtime overhead brought by Titan? (Section 5.5)

Baselines. We compare Titan with the fuzzers listed in Table 2. AFLGo [15] and Beacon [12] are two state-of-the-

Table 2: Compared fuzzers.

Fuzzer	Category	Description
AFLGo [15]	Directed	Sophisticated seeds prioritization
Beacon [12]	Directed	Rejecting infeasible execution
Parmesan [16]	Coverage	Guide fuzzer using the sanitizer labels
AFL [20]	Coverage	Evolutionary mutation strategies
AFL++ [34]	Coverage	AFL with multiple optimizations from the community
FairFuzz [22]	Coverage	Mutation with crucial bytes fixed
AFLFast [21]	Coverage	Power scheduling with Markov chain
Entropic [35]	Coverage	Power scheduling with information entropy

art directed gray-box fuzzers. Since AFLGo supports multiple targets, we use AFLGo-Multi to indicate the version that fuzzes all the targets simultaneously and AFLGo-Single to indicate the version that reaches each target individually. Beacon is a state-of-the-art directed fuzzing by early terminating the infeasible execution. We choose AFL [20] and AFL++ [34], the fundamental greybox fuzzer and its updated versions, as the baseline. We further choose three greybox fuzzers that optimize the seed scheduling with proper energy assigned [21], [35] and mutation strategies [16], [22] for more reliable comparison. Their technical details are mentioned in Section 6. For each coverage-guided fuzzer, we follow the configuration provided by Magma.[1] We also planned to compare with FuzzGuard, Hawkeye, MC2, and CAFL. However, none of these tools is publicly available. Thus, we exclude them from our experiment.

Benchmarks. We have chosen two benchmarks to evaluate Titan.

First, to answer the research questions, we use the state-of-the-art benchmark, Magma [23], which consists of various CVEs chosen from nine programs frequently evaluated in the state-of-the-art fuzzing, with diverse functionalities as shown in Table 3. We do not use the benchmark `lua` since it is newly added in the Magma, and no state-of-the-art directed fuzzers can correctly deploy on this program. For each vulnerability, we follow the instructions of the benchmark to set the last program location where the crash is triggered as the target. Meanwhile, Parmesan also cannot work on `sqlite3` and `PhP`. We have required assistance from the developers of Magma and Parmesan, who told us the improvements could be made in the future. However, since no further action has been conducted, we have temporarily excluded these programs from our comparison.

Second, to show the effectiveness of Titan for effectively reproducing real vulnerabilities and detecting incomplete fixes, we also reproduce the CVEs in the newest version of the program used in both Magma and the state-of-the-art directed fuzzing [10]–[13], [15].

Configurations. The initial seed corpus heavily affects the effectiveness of fuzzing [36]. To reduce bias, we use the seeds provided in the Magma benchmark. For each experiment, we conducted ten trials. Each trial has a 24-hour time budget, as indicated by the benchmark [23]. We use the Mann-Whitney U Test [37] to demonstrate the statistical significance of our framework's contribution. Additional configuration details are provided in the related experiments.

1. https://github.com/HexHive/magma/tree/v1.2/fuzzers

Table 3: Benchmark programs in Magma and the static analysis time (s) used for preprocessing.

Project	libpng	libsndfile	libtiff	libxml2	openssl	poppler	sqlite3	php
Size/Target	95K/7	83K/25	95K/14	320K/17	630K/20	260K/22	387K/20	1.6M/16
Titan	56	140	268	2279	31256	19015	3819	55803
AFLGo	103	542	2230	5608	21492	13487	12588	43052

All experiments are conducted on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 64GB of memory, running Ubuntu 20.04 LTS.

5.1. Efficiency in Reproducing Vulnerabilities

The main goal of directed fuzzing is to reproduce the vulnerabilities efficiently. Thus, we first compare Titan with the state-of-the-art fuzzers mentioned in Table 3. We run each fuzzer ten times and measure the average time to reproduce the marked bugs in the Magma benchmark.

The results are shown in Tables 4 and 5, which list all the targets found by the evaluated fuzzers.

Comparing the evaluated directed fuzzers, we make the following observations:

- For most targets, Titan shows a significant improvement with p-values less than 0.05. On average, Titan outperforms the state-of-the-art directed fuzzers with 13.7x improvement while detecting ten more targets than the compared fuzzers.
- AFLGo-Multi could take 3.2x more time than AFLGo-Single to reproduce each target, indicating the bias in AFLGo caused by the *synergy ignorance* issue. Nevertheless, AFLGo-Single needs 3.6x more accumulative CPU time to reproduce these targets than AFLGo-Multi, demonstrating the necessity of multi-target directed fuzzing.
- While some single-target fuzzers, such as AFLGo-Single and Beacon, outperform Titan for specific targets, the average reproduction speed of Titan is faster than AFLGo-Single and Beacon, with 9.1x and 3.2x speedups, respectively.
- Titan exhibits better accumulative performance with 11.5x improvement on average compared with AFLGo-Multi, AFLGo-Single, and Beacon, highlighting the effectiveness of Titan as a multi-target directed fuzzing.

Compared with the non-directed fuzzers, Titan detects 12 more bugs with 25x speed improvement on average, showing the effectiveness of Titan as a multi-target directed fuzzer. We study the reason that AFLFast and Entropic achieve better results in two cases, SND017 and PHP011. We find that Titan introduces a minor time cost when using the inference-based taint analysis to find the bytes that can be mutated simultaneously. Still, Titan detects 12 and 18 more bugs with 14.9x and 3.3x speedup compared with AFLFast and Entropic, respectively. We also observe that AFL++, integrating AFL with multiple state-of-the-art techniques, outperforms AFL by finding 11 more targets, though its performance may not be stable in some targets, e.g., SND006.

Table 4: Reproduction time for each target in Magma. The $Time$ indicates the reproduction time (second) averaged over ten runs. $T.O.$ indicates the fuzzer cannot reproduce the targets within the given time budget, 24 hours. $Ratio$ and p indicates the improvement ratio and p-value compared with Titan. $\varnothing$ indicates that the fuzzer cannot deploy in the project.

Bug ID	Titan	AFLGo-Multi			AFLGo-Single			Beacon			Parmesan			AFLFast			FairFuzz			AFL			AFL++			Entropic		
	Time	*Time*	*Ratio*	*p*	*Time*	*Ratio*	*p*	*Time*	*Ratio*	*p*	*Time*	*Ratio*	*p*	*Time*	*Ratio*	*p*	*Time*	*Ratio*	*p*	*Time*	*Ratio*	*p*	*Time*	*Ratio*	*p*	*Time*	*Ratio*	*p*
PNG003	15	15	1.0x	0.03	15	1.0x	0.04	20	1.3x	<0.01	44	2.9x	-	15	1.0x	0.03	15	1.0x	0.03	15	1.0x	0.03	21	1.4x	0.01	18	1.2x	0.02
PNG007	28591	68058	2.4x	<0.01	63515	2.2x	0.01	57100	2.0x	<0.01	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	45304	1.6x	0.01	69243	2.4x	0.01	46021	1.6x	<0.01	80605	2.8x	<0.01
SND001	524	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	536	1.0x	0.04	*T.O.*	*N.A*	-	79799	152.3x	<0.01	*T.O.*	*N.A*	-	84960	162.1x	<0.01	1017	1.9x	0.01	*T.O.*	*N.A*	-
SND005	39	10357	265.6x	<0.01	999	25.6x	<0.01	42	1.0x	0.04	*T.O.*	*N.A*	-	2618	67.1x	<0.01	29281	748.7x	<0.01	1633	41.9x	<0.01	5105	130.9x	<0.01	*T.O.*	*N.A*	-
SND006	611	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	80777	132.2x	<0.01	*T.O.*	*N.A*	-	15856	26.0x	<0.01	*T.O.*	*N.A*	-
SND007	609	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	80146	131.6x	<0.01	*T.O.*	*N.A*	-	4772	7.8x	<0.01	*T.O.*	*N.A*	-
SND017	825	8336	10.1x	<0.01	3769	4.6x	<0.01	1352	1.6x	<0.01	*T.O.*	*N.A*	-	248	0.3x	-	86219	104.5x	<0.01	3148	3.8x	<0.01	1107	1.3x	0.01	*T.O.*	*N.A*	-
SND020	5321	78007	14.7x	<0.01	*T.O.*	*N.A*	-	3491	0.7x	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	2585	0.5x	-	*T.O.*	*N.A*	-
SND024	607	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	80777	133.1x	<0.01	*T.O.*	*N.A*	-	4392	7.2x	<0.01	*T.O.*	*N.A*	-
TIF002	79027	*T.O.*	*N.A*	-	81827	1.0x	0.01	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	81614	1.0x	0.03	*T.O.*	*N.A*	-	80966	1.0x	0.03	*T.O.*	*N.A*	-
TIF006	78717	82787	1.1x	<0.01	79175	1.0x	0.04	52149	0.7x	-	79631	1.0x	0.03	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	78942	1.0x	0.04	21715	0.3x	-	*T.O.*	*N.A*	-
TIF007	187	18252	97.6x	<0.01	5562	29.7x	<0.01	250	1.3x	0.01	2003	10.7x	<0.01	770	4.1x	<0.01	644	3.4x	<0.01	4176	22.3x	<0.01	210	1.1x	0.03	3396	18.2x	<0.01
TIF009	69305	74482	1.1x	0.02	61443	0.9x	-	53560	0.8x	-	*T.O.*	*N.A*	-	72149	1.0x	0.03	76534	1.1x	0.02	75359	1.1x	0.04	65557	1.0x	-	*T.O.*	*N.A*	-
TIF012	2569	33250	12.9x	<0.01	27796	10.8x	<0.01	3686	1.4x	<0.01	52250	20.3x	<0.01	3585	1.4x	<0.01	3312	1.3x	<0.01	6842	2.7x	<0.01	4282	1.7x	0.01	*T.O.*	*N.A*	-
TIF014	2696	57240	21.2x	<0.01	38140	14.2x	<0.01	4709	1.8x	<0.01	*T.O.*	*N.A*	-	13994	5.2x	<0.01	76364	28.3x	<0.01	47372	17.6x	<0.01	12648	4.7x	<0.01	*T.O.*	*N.A*	-
XML003	60000	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	75004	1.3x	<0.01	65379	1.1x	0.01	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	63381	1.0x	0.03	82657	1.4x	0.01
XML009	3427	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	6077	1.8x	<0.01	70335	20.5x	<0.01	50644	14.8x	<0.01	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	4897	1.4x	<0.01	19803	5.8x	<0.01
XML017	21	22	1.0x	0.03	21	1.0x	0.04	46	2.2x	<0.01	53	2.4x	<0.01	23	1.0x	0.03	23	1.0x	<0.01	22	1.0x	0.04	41	2.0x	0.02	21	1.0x	0.04
SSL001	42260	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	33937	0.8x	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	26207	0.6x	-	*T.O.*	*N.A*	-
SSL003	382	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	527	1.4x	0.01	401	1.0x	0.03	398	1.0x	0.04	389	1.0x	0.04	393	1.0x	0.04	470	1.2x	0.03	385	1.0x	0.04
SSL020	78986	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	83335	1.1x	<0.01	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	79877	1.0x	<0.01	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-
PDF003	73612	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	82152	1.1x	0.02	*T.O.*	*N.A*	-
PDF010	7501	17536	2.3x	<0.01	23002	3.1x	<0.01	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	40507	5.4x	<0.01	23592	3.1x	<0.01	13259	1.8x	0.01	7512	1.0x	0.03	7813	1.0x	0.04
PDF018	35396	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	51783	1.5x	<0.01	79742	2.3x	0.01
SQL002	11477	62739	5.5x	<0.01	37761	3.3x	<0.01	10209	0.9x	-		$\varnothing$		20146	1.8x	<0.01	65013	5.7x	<0.01	42315	3.7x	<0.01	15601	1.4x	<0.01	30770	2.7x	-
SQL014	70950	80432	1.1x	0.02	81419	1.2x	0.04	75868	1.1x	0.01		$\varnothing$		77984	1.1x	0.03	*T.O.*	*N.A*	-	*T.O.*	*N.A*	-	63845	0.9x	-	*T.O.*	*N.A*	-
SQL018	39017	76433	2.0x	<0.01	56886	1.5x	<0.01	45515	1.2x	<0.01		$\varnothing$		75473	1.9x	<0.01	78600	2.0x	<0.01	74737	1.9x	<0.01	41364	1.1x	0.02	66385	1.7x	0.03
PHP004	103	1772.5	17.2x	0.01	451	4.4x	<0.01	3980	38.6x	<0.01		$\varnothing$		107	1.0x	0.02	436	4.2x	<0.01	136	1.3x	<0.01	57470	558.0x	<0.01	*T.O.*	*N.A*	-
PHP009	636	11469	18.0x	0.01	34390	54.1x	<0.01	5718	9.0x	<0.01		$\varnothing$		2572	4.0x	<0.01	*T.O.*	*N.A*	-	39842	62.6x	<0.01	30599	48.1x	<0.01	*T.O.*	*N.A*	-
PHP011	49	2087	42.6x	<0.01	230	4.7x	<0.01	41	0.8x	-		$\varnothing$		206	4.2x	<0.01	12019	245.3x	<0.01	110	2.2x	<0.01	2949	60.4x	<0.01	23	0.5x	-
Avg.		**>28.7x**			**>9.1x**			**>3.2x**			**>7.4x**			**>14.9x**			**>77.6x**			**>18.4x**			**>30.3x**			**>3.3x**		

Table 5: Accumulative CPU time for detecting all targets mentioned in Table 4 using directed fuzzers. Max indicates the maximal time for finding the last targets using multi-target directed fuzzers. Sum indicates the accumulative time for finding all targets using single-target directed fuzzers. $Ratio$ is the improvement ratio compared with Titan.

Project	Titan	AFLGo-Multi		AFLGo-Single		Beacon	
	Max	*Max*	*Ratio*	*Sum*	*Ratio*	*Sum*	*Ratio*
libpng	28591	68058	2.3x	63530	2.2x	57120	2.0x
libsndfile	5321	86400	16.2x	436768	82.1x	267380	50.2x
libtiff	79027	86400	1.1x	293243	3.7x	200754	2.5x
libxml2	60000	86400	1.4x	172820	2.9x	81127	1.4x
openssl	78986	86400	1.1x	259200	3.3x	117799	1.5x
poppler	73612	86400	1.2x	195802	2.7x	259200	3.5x
sqlite3	70950	80432	1.1x	176066	2.5x	131592	1.9x
php	636	11469	18.0x	35071	55.1x	9739	15.3x
Avg.			**5.3x**		**19.3x**		**9.8x**

In terms of the pre-analysis time, Titan requires no more than half an hour to analyze each target in the evaluation benchmark shown in Table 3. Compared with AFLGo, while Titan may require longer pre-analysis for larger programs, e.g., `openssl` and `php`, it is acceptable because the analysis time could be spent offline, and Titan saves more time during the fuzzing period. Furthermore, we can optimize the pre-analysis time via incremental or compositional static analysis techniques. Since it is orthogonal to the problem solved by this work, we will address it in future work.

5.2. Incomplete Fix Detection Ability

Moreover, we also evaluate the effectiveness of Titan in detecting incomplete fixes of the previously reported bugs in the newest version of the programs. To conduct this evaluation, we evaluate all the projects mentioned in existing directed fuzzers [10]–[13], [15] to examine whether Titan can identify the incomplete fixes in the newest version of the evaluated programs.

We show the results in Table 6, which include all the previously unseen bugs detected uniquely by Titan. Specifically, Titan detects ten incomplete fixes and one new bug that existing efforts cannot detect, two of which have been assigned CVE IDs. In contrast, AFLGo-Multi cannot find any bugs in the newest version of the projects compared with two found by AFLGo-Single, highlighting the existence of the *synergy ignorance* problem. By addressing this issue, Titan detects 12 more bugs than AFLGo-Multi. Moreover, the coarse-grained distance feedback used in AFLGo-Single prevents it from detecting the vulnerabilities that Titan does with the precondition feedback. To illustrate the effectiveness of Titan, we further provide a case study in Section 5.6. The links to all bug reports can be found here.

Table 6: Projects evaluated in existing directed fuzzing with the bugs detected. N_{bug} represents the number of bugs detected. Status represents whether the bug is a new one or an incomplete fix of a previous bug or CVE.

Project	Evaluated Fuzzer	N_{bug}	$N_{AFLGo_{Multi}}$	$N_{AFLGo_{Single}}$	Category
libpng	Magma, [10], [11], [13], [15]	1	0	0	New
libtiff	Magma, [11], [13], [15]	1	0	0	Incomplete fix
poppler	Magma, [12], [13]	3	0	1	Incomplete fix
binutils	[10]–[13], [15]	1	0	0	Incomplete fix
tcpreplay	[13]	1	0	0	Incomplete fix
bento4	[11], [13]	5	0	1	Incomplete fix

5.3. Effectiveness of the Correlation Inference

Having evaluated the key ability of Titan in reproducing multiple targets, we further study the strategies of Titan with an ablation study. We first conduct a general comparison using two variants of Titan by disabling the synergy-aware scheduling and multi-target-oriented mutation, which we denote as Titan-m and Titan-s, respectively. We used the same evaluation setting in the previous experiment. According to the results in Figure 5, Titan outperforms Titan-m and Titan-s to reproduce the targets with 1.31x and 1.28x times improvement. We then conduct further analyses of our designed method.

How does the synergy-aware scheduling help directed fuzzing for multiple targets? We first study how

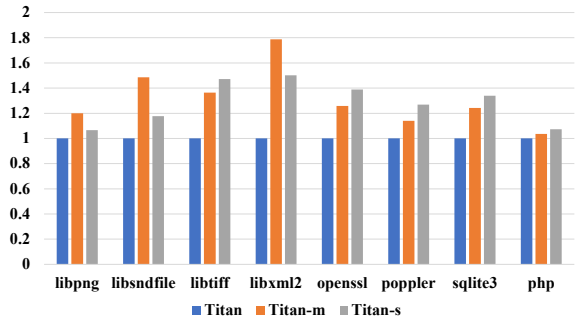

Figure 5: Comparison of Titan-m and Titan-s with the Titan.

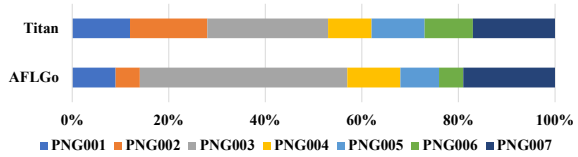

Figure 6: Distribution of the seed chosen for the targets in `Libpng` compared with AFLGo.

synergy-aware scheduling helps directed fuzzing find the most promising seed. We examine how many correlations exist in the evaluated projects and their proportions in the program. The presence of more correlations means that the feedback provided to the fuzzer can be more precise, helping it distinguish between different seeds for multiple targets.

The results are shown in Table 7. Overall, Titan detects an average of 41739 correlations in each program, with 4360 overlaps, 1739 conflicts, and 35640 independent relations. The large numbers demonstrate the necessity of considering the relations among multiple targets. Additionally, the number of relations constitutes only 6.3% of the instructions that need to be instrumented on average. Compared to the 13.7% instrumentation ratio in distance-based directed fuzzers, our correlations can provide fine-grained feedback for reaching multiple targets with less instrumentation. Moreover, we study the distribution of the seeds chosen for different targets in Libpng with AFLGo in Figure 6. Titan explores the targets more evenly, while AFLGo only focuses on exploring part of the targets, e.g., PNG007, Therefore, Titan has a higher chance of detecting more bugs in the programs, which indicates the effectiveness of synergy-aware scheduling.

How does multi-target-oriented mutation minimize the executions for multiple targets? Next, we study the improvement brought by multi-target-oriented mutation. Intuitively, Titan-m should reproduce multiple targets faster with few mutations since we mutate the bytes for different targets. Therefore, we evaluate the executions needed to reproduce the Magma targets.

Figure 7 shows the results. On average, Titan uses 95.0% fewer number of executions than AFLGo to reproduce the targets. Interestingly, the range of the executions needed for AFLGo could fluctuate wider than Titan in the same project, indicating the potential of using correlation to mutate for different targets synergistically.

Table 7: The number of variables found in the overlap, conflict, and independent correlations in each project from Magma. $Size\ N_I/N_B$ indicates the size of each project, where N_I and N_B indicate the number of instructions and blocks in the LLVM IR.

Project	$Size\ N_I/N_B$	Overlap	Conflict	Independent
libpng	49352 / 6882	132	178	3275
libsndfile	152409 / 16726	738	540	5771
libtiff	117531 / 15696	623	402	10133
libxml2	391529 / 67961	803	496	4030
openssl	795285 / 101436	2095	1763	18201
poppler	360622 / 42552	893	1041	12640
sqlite3	380979 / 59291	8052	1953	34602
php	2500805 / 354455	21542	7538	196473

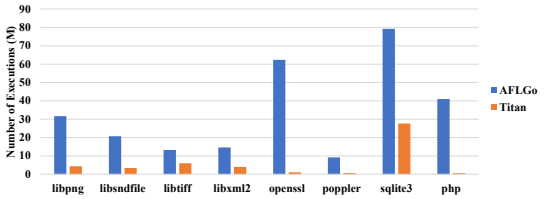

Figure 7: The average number of executions needed for reproducing the targets in Magma compared with AFLGo.

How does the precision of static analysis influence the reproduction speed? Finally, we evaluate the influence of the static analysis engine, whose accuracy influences the quality of the inferred correlation. We vary two crucial settings for static analysis: the granularity of the pointer analysis and the number of loop unrolling. By default, Titan uses a flow-insensitive pointer analysis and unrolls the loop twice. We evaluate the results using a flow-sensitive pointer and unroll the loop 5, 15, and 30 times.

The results are shown in Figure 8. When combined with a more precise pointer analysis, Titan-precise uses 0.82% time in reproducing the targets at the expense of 3.4x pre-analysis time. Similarly, along with unrolling the loop, the reproduction results could slightly improve (1.03x

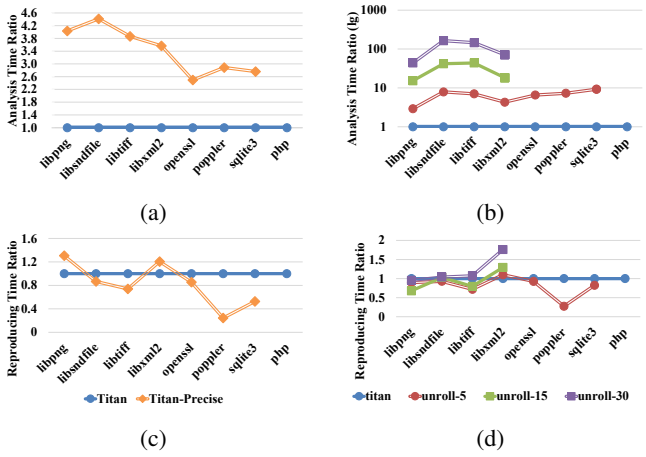

Figure 8: Influence of static analysis in reproduction and pre-analysis time. Figures 8a and 8c demonstrate the results for pointer analysis, while Figures 8b and 8d for loop unrolling.

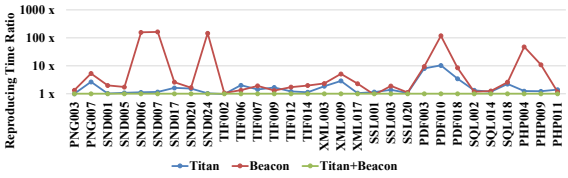

Figure 9: Comparison of the integration. The y-axis is the ratio of the reproducing time compared with Titan + Beacon.

on average), using the much more (47.2x) pre-analysis time. Moreover, Titan cannot finish analyzing the project within a 24-hour time budget over half of the project when unrolling the loop more than 15 times.

In conclusion, even though the question remains, our results demonstrate that Titan provides a practical implementation design for striking an optimal balance between precision and efficiency, allowing us to achieve sufficient accuracy in detecting vulnerabilities while minimizing the associated computational costs. We also plan to address this ongoing challenge in future work.

5.4. Compatibility of Assisting Single-Target Fuzzers Reach Multiple Targets

To show the compatibility of Titan, we integrate it with Beacon, a state-of-the-art fuzzer that prunes infeasible executions for directed fuzzing. The precondition inferred by Titan can also help to prune infeasible executions. Therefore, we use Titan to determine whether the results from Beacon can be merged for multiple targets. Specifically, since Titan also infers the preconditions for each target, we analyze the results exported by Beacon and check whether they can be merged for multiple targets. Then, we use the merged results to generate a pruned binary for fuzzing to evaluate the efficiency of reproducing multiple targets.

The results for Beacon and Titan + Beacon are shown in Figure 9. Titan can modify Beacon to support multiple targets and improve its performance by 1.5x and 6.0x compared with using Titan or Beacon alone, respectively. This improvement indicates that Titan effectively addresses the *synergy ignorance* problem and can help other fuzzers support multi-target directed fuzzing.

5.5. Runtime Overhead Comparison

Because Titan provides fine-grained feedback for directed fuzzing to reach multiple targets efficiently, the extra instrumentation may harm the performance. Therefore, we compare the runtime overhead introduced by Titan and AFLGo to the original AFL. We use the seed in the Magma benchmark and force the fuzzer to generate the same number of inputs using the deterministic mode, ensuring that all inputs are the same.

The results are presented in Table 8. Titan achieves 1.8% less overhead than AFLGo on average. The maximum overhead is 30.4% for `openssl`. Although we do not need to instrument every block for recording the distance according to the discussion in Section 5.3, the runtime overhead can vary based on the paths executed by the provided inputs, which may result in Titan having a higher overhead than AFLGo in few projects, such as `libtiff` and `poppler`. Nevertheless, our evaluation demonstrates that the runtime overhead of Titan is still manageable, considering the optimized reproduction time. Therefore, although Titan may have a higher overhead in some cases, it remains a viable and effective approach for detecting vulnerabilities in real-world software systems.

Table 8: Runtime overhead comparison of Titan and AFLGo-Multi running the same amount of input. N represents the number of executions conducted by the fuzzers.

Project	N	Titan	AFLGo	AFL
libpng	66.7K	124 (2.5%)	129 (6.6%)	121
libsndfile	43428	130 (2.3%)	137 (7.8%)	127
libtiff	1.6M	4595 (17.5%)	4008 (2.5%)	3909
libxml2	34.3K	80 (5.4%)	84 (10.5%)	76
openssl	52.5K	277 (30.4%)	345 (62.0%)	213
poppler	1.0M	27990 (28.8%)	25626 (18.0%)	21726
sqlite3	698.5K	1029 (6.7%)	999 (3.6%)	964
php	1.7M	1517 (4.6%)	1469(1.4%)	1449
Avg.		**12.3%**	**14.1%**	

5.6. Case Study

To understand the benefit brought by Titan, we study the incomplete fix of CVE-2020-16599 detected in Binutils, which both state-of-the-art academic and industrial fuzzers have frequently evaluated. We simultaneously set multiple targets for various CVEs in Binutils for Titan and AFLGo to detect incomplete fixes. We also reproduce the targets detected by Titan using the single-target version of AFLGo. Unfortunately, despite the vulnerability being patched for two years, this incomplete fix has not been detected by previous fuzzers.

We simplify the program in Figure 10. CVE-2020-16599 involves memory access violations using an ELF binary with a malformed header. We set multiple targets of different vulnerabilities for Titan, e.g., targets at 12 and 15. CVE-2020-16599 requires satisfying the header check at Line 6 to make $flag = 1$ and reach Line 15. When setting multiple targets for fuzzers, AFLGo-Multi explores the branch at Line 12 more often since other targets could be easy to reach, e.g., the target at Line 12, introducing a bias to the average distance, which makes reaching the target at Line 15 more difficult.

Even if setting Line 15 as the only target, AFLGo-Single does not consider covering the branch at Line 7 since it does not influence the control-flow distance. Fortunately, Titan can identify one of the root causes for CVE-2020-16599 is to make `flag` set to 1, and thus prioritize the seed satisfying this condition for the target at Line 15 in both single-target and multi-target scenarios. Therefore, we successfully detect this incomplete fix.

```
int main(){
  int header,content = parse_input();
  int flag = 0;

  // incomplete fix of CVE-2020-16599

  if(header_check(header)) {
    flag = 1;
  } else {...}

  if(parse_header()) {
    if(parse_content()) {
      ...//other reachable targets
    }
    else if(flag&&...) {
      crash(); // CVE-2020-16599 occurs;
    }
  }
}
```

Figure 10: Incomplete fix of CVE-2020-16599.

5.7. Discussion

Synergy with rich semantics. In Titan, we utilize the correlations among targets using the fundamental path conditions, enabling us to fuzz multiple targets more efficiently in one instance than other directed fuzzers. However, some bugs could involve complex semantics apart from path conditions, such as excessive memory usage or critical content leakage in various application scenarios, e.g., network service and operating system. The synergy analysis in Titanhas not supported these rich semantics, which could be extended in future work by integrating additional static analysis for various program behaviors, e.g., lock analysis for concurrency issues.

Cooperating static analysis with fuzzing. Static analysis has shown a great potential for optimizing fuzzing in many aspects, such as seed scheduling [10], [11], mutation [16], [17], [38], and tailored instrumentation [39]. In Titan, we leverage static analysis to infer the synergy relations among multiple targets to improve seed scheduling and mutation. However, the precision-efficiency paradox of static analysis remains an open question that influences the effectiveness of directed fuzzing, as shown in Figure 8. Statically determining the granularity of the static analysis for every project evaluated by Titan could become a threat to validity. For example, an ideal field-, path-, and context-sensitive static analysis could help Titan identify the correlations more precisely involving complex semantics with data structure, nested loop, and indirect call, even though such a precise analysis suffers from scalability issues, which can hinder the effectiveness of assisting other applications such as fuzzing. Our work does not focus on balancing this paradox but adapts existing designs for the directed fuzzing of multiple targets. In future work, we plan to design an automatic selection strategy to find the most proper static analysis for different targets.

6. Related Work

Apart from the existing literature in directed grey-box fuzzing (Section 2.1), we survey other related work that can be adapted for directed fuzzing for further improvements.

Sophistcated seed scheduling. The idea of scheduling is first proposed by AFLFast [21], which describes the probability of executing different paths in the programs using the Markov chain. However, since the probability cannot be obtained precisely in practice, AFLFast uses execution frequency to estimate the probability of finding new paths as the potential for each seed. Based on this intuition, one trend of existing works is to estimate the probability more precisely and reasonably. For example, Entropic [35] uses the information theory and the entropy to approximate the probability. EcoFuzz [40] design a multi-armed bandit model to measure the potential gains of the seed. Meanwhile, some efforts propose to use the uncovered part of the program as the potential gains for the seeds. K-Scheduler [41] proposes to use graph centrality to approximate the potential gains of the seed. BeliefFuzz [42] models the seed scheduling as a Monte Carlo planning process to simultaneously consider the cost and the benefits. Moreover, Parmesan [16] proposes to set the program points labelled by sanitizers as the targets and redesign the fitness for fuzzing to achieve better outcomes. Still, these techniques are designed to achieve higher coverage, which cannot be directly adapted for directed fuzzing.

Optimized mutation strategies. One major trend of optimizing mutation is to mutate the related input offsets to satisfy the uncovered branch conditions. Other than random mutation, Fairfuzz [22] identifies the input offsets where the values are not necessary to change, Thus, minimizing the input search space improves the efficiency of mutation. Angora [32] adapts byte-level taint tracking to discover the related input bytes of the target condition and then applies a gradient-descent-based search strategy. Redqueen [25] proposes to use the intermediate values as the feedback to modify the values in the inputs. The second direction is to integrate fuzzers with concolic/symbolic execution, a.k.a. hybrid fuzzing, for tackling complex and tight path constraints. For example, QSYM [43] solves part of the path constraint for a basis seed and leverages mutation for validated inputs satisfying the actual condition. Intriguer [44] further replaces symbolic emulation with dynamic taint analysis, which decreases the overhead of modelling a large amount of mov-like instructions. Pangolin [38] proposes to preserve the constraint as an abstraction and reuse it to guide further input generation.

Nevertheless, these methods all require a specific branch to find the related bytes. However, directed fuzzing cannot choose the branches as coverage-guided fuzzing does since it is challenging to determine which branch to be covered to reach the target faster. Some branches may not even satisfy the path conditions of the targets. Thus, directed fuzzing cannot trivially adapt existing mutation approaches to generate the inputs toward the targets effectively. Fortunately, Titan can specify branches using the variables in the inferred correlations and make the mutation target-directed. Therefore, it might solve an orthogonal problem of optimizing mutation for directed fuzzing.

7. Conclusion

We have presented Titan, which provides fine-grained feedback by distinguishing the path correlations for directed fuzzers to reach multiple targets efficiently. Titan improves the effectiveness of seed scheduling while optimizing the mutation with less redundancy. The empirical results prove that the improvement brought by Titan is significant, which Titan outperforms existing directed fuzzers 21.4x in reaching multiple targets using 95.0% fewer executions. Moreover, Titan also detects ten previously unseen incomplete fixes with two CVE IDs assigned.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their valuable feedback. The authors are supported, in part, by Hong Kong Research Grant Council under Grant No. RGC16206517, Hong Kong Innovation and Technology Commission under Grant No. ITS/440/18FP, National Natural Science Foundation of China under Grant No. 62302434, donations from Huawei, and Qizhen Scholar Foundation of Zhejiang University. Peisen Yao is the corresponding author.

References

[1] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," November 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/

[2] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[3] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: Let existing test cases help," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 910–913. [Online]. Available: https://doi.org/10.1145/2786805.2803206

[4] M. Soltani, A. Panichella, and A. Van Deursen, "Search-based crash reproduction and its impact on debugging," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[5] 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*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 144–155. [Online]. Available: https://doi.org/10.1145/2884781.2884843

[6] F. Brown, D. Stefan, and D. Engler, "Sys: A Static/Symbolic tool for finding good bugs in good (browser) code," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 199–216. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/brown

[7] "Oss-fuzz report," https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html, 2018, accessed: 2018-11-06.

[8] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik, "User-guided program reasoning using bayesian inference," *SIGPLAN Not.*, vol. 53, no. 4, p. 722–735, jun 2018. [Online]. Available: https://doi.org/10.1145/3296979.3192417

[9] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 809–814. [Online]. Available: https://doi.org/10.1145/3236024.3275525

[10] 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*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2095–2108. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243849

[11] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3559–3576.

[12] 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.

[13] "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zong

[14] Z. Du, Y. Li, Y. Liu, B. Mao, L. Chen, J. Guo, Z. He, D. Mu, C. Pang, R. Yu *et al.*, "Windranger: A directed greybox fuzzer driven by deviation basic blocks," in *2022 IEEE/ACM 44st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022.

[15] 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*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2329–2344. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134020

[16] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2289–2306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[17] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: towards bug-driven hybrid testing," *CoRR*, vol. abs/1906.07327, 2019. [Online]. Available: http://arxiv.org/abs/1906.07327

[18] A. Shah, D. She, S. Sadhu, K. Singal, P. Coffman, and S. Jana, "Mc2: Rigorous and efficient directed greybox fuzzing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2595–2609. [Online]. Available: https://doi.org/10.1145/3548606.3560648

[19] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2169–2182. [Online]. Available: https://doi.org/10.1145/3460120.3484596

[20] "Afl: american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2013, accessed: 2013.

[21] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1032–1043. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978428

[22] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 475–485. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238176

[23] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[24] P. Yao, Q. Shi, H. Huang, and C. Zhang, "Program analysis via efficient symbolic abstraction," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485495

[25] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[26] "GREYONE: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, aug 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[27] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 166–176.

[28] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 49–60. [Online]. Available: https://doi.org/10.1145/2884781.2884794

[29] G. Smith, "On the foundations of quantitative information flow," in *International Conference on Foundations of Software Science and Computational Structures*. Springer, 2009, pp. 288–302.

[30] W. You, X. Wang, S. Ma, J. Huang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," 05 2019, pp. 769–786.

[31] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1–17.

[32] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, May 2018, pp. 711–725. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2018.00046

[33] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[34] 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)*. USENIX Association, Aug. 2020.

[35] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 678–689. [Online]. Available: https://doi.org/10.1145/3368089.3409748

[36] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 861–875. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[37] P. E. McKnight and J. Najab, *Mann-Whitney U Test*. American Cancer Society, 2010, pp. 1–1. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470479216.corpsy0524

[38] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 1144–1158. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00063

[39] M. Wang, J. Liang, C. Zhou, Y. Jiang, R. Wang, C. Sun, and J. Sun, "{RIFF}: Reduced instruction footprint for {Coverage-Guided} fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 147–159.

[40] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2307–2324. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[41] D. She, A. Shah, and S. S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 2194–2211, 2022.

[42] H. Huang, H.-C. Chiu, Q. Shi, P. Yao, and C. Zhang, "Balance seed scheduling via monte carlo planning," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–15, 2023.

[43] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[44] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 515–530. [Online]. Available: https://doi.org/10.1145/3319535.3354249

Appendix A.
Meta-Review

A.1. Summary

This paper presents Titan, a tool to provide multi-target directed greybox fuzzing. The core idea of Titan is to explore correlations among different targets as guidance for directed fuzzing; based on the correlations, Titan differentiates the potential of seeds for different targets during seed scheduling and identifies the bytes that can be changed simultaneously during mutations. The evaluation shows that Titan outperforms the existing directed greybox fuzzing tools when it is applied to handle multiple targets.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) This paper pinpoints an unnoticed problem that multiple-target directed fuzzing suffers great performance loss when the number of targets explodes.
2) This paper designs a new directed greybox fuzzing tool Titan, which performs much better than the existing tools in multiple-target scenarios in the evaluation.
3) The authors claim to release and keep maintaining their prototype, which will facilitate research in this field.

A.4. Noteworthy Concerns

1) Titan was evaluated on the fuzzing benchmark Magma, yet the interdependence between the targets can become more intricate in real-world scenarios. It remains unclear whether Titan can handle such intricate cases without encountering unexpected issues, including some potential compilation errors, which have been observed in similar tools.