Everything is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference

Heqing Huang*, Anshunkang Zhou†, Mathias Payer‡, Charles Zhang†
*City University of Hong Kong
†The Hong Kong University of Science and Technology
‡ EPFL

***Abstract*—Directed fuzzing demonstrates the potential to reproduce bug reports, verify patches, and debug vulnerabilities. State-of-the-art directed fuzzers prioritize inputs that are more likely to trigger the target vulnerability or filter irrelevant inputs unrelated to the targets. Despite these efforts, existing approaches struggle to reproduce specific vulnerabilities as most generated inputs are irrelevant. For instance, in the Magma benchmark, more than 94% of generated inputs miss the target vulnerability. We call this challenge the *indirect input generation* problem.**

We propose to increase the yield of inputs that reach the target location by restraining input generation. Our key insight is to infer likely invariants from both reachable and unreachable executed inputs to constrain the search space of the subsequent input generation and produce more reachable inputs. Moreover, we propose two selection strategies to minimize the fraction of unnecessary inputs for efficient invariant inference and deprioritize imprecise invariants for effective input generation. Halo, our prototype implementation, outperforms state-of-the-art directed fuzzers with a 15.3x speedup in reproducing target vulnerabilities by generating 6.2x more reachable inputs. During our evaluation, we also detected ten previously unknown bugs involving seven incomplete fixes in the latest versions of well-fuzzed targets.

1. Introduction

Fuzzing is arguably the most effective way of finding vulnerabilities in today's software systems. However, the complexity of modern software architectures and the ever-growing threat of cyber-attacks pose significant challenges for developers to verify, analyze, and fix vulnerabilities. In response to these challenges, directed fuzzing has emerged as a powerful tool to target specific areas in software (compared to general-purpose fuzzing). Building on conventional fuzz testing, directed fuzzing allows developers to diagnose specific program behaviors and uncover vulnerabilities that other testing methods might overlook. It has many vital applications, such as testing patches with regression [1], [2], reproducing crashes [3], [4], validating static analysis reports [5], [6], and generating proof-of-concept inputs for various vulnerabilities [7].

The key to efficiently reproducing specific vulnerabilities is to generate inputs that get close to the target areas as early as possible. However, such directed input generation requires solving path conditions, which suffer from path explosion and expensive constraint-solving problems, to be scalable. For example, KATCH [8], the directed version of the most widely-used symbolic execution engine, KLEE [9], cannot expose Heartbleed even in 24 hours, while directed fuzzers AFLGo takes less than 20 minutes [10]. Therefore, unlike conventional approaches that rely on heavyweight symbolic execution to generate inputs [2], [8], [11], directed fuzzers mitigate the scalability issue by designing various fitness functions to have a higher priority in examining the target vulnerabilities.

However, despite tremendous efforts made, the majority of inputs generated by existing directed fuzzers still can not reach the target areas, which we refer to as *indirect input generation* problem. Specifically, the majority of existing efforts [10], [12], [13], [14], [15], [16] merely generate random inputs based on seed templates. Their intuition is to prioritize a few seeds that are "closer" to the target and, therefore, more likely to reach the target area. Since most input generation methods rely on randomly mutating the seed, existing fuzzers generate an enormous number of unreachable inputs that entirely miss the targets. For instance, over 95% of the generated inputs from directed greybox fuzzing fail to reach the targets [17]. This low yield puts a burden on the fuzzer as resources are unnecessarily spent on unreachable paths. Alternatively, some state-of-the-art approaches [17], [18], [19] propose to reject unreachable inputs. Their intuition is to terminate the execution once it reaches program points that cannot reach the targets. Even though resources can be saved by terminating the execution of irrelevant inputs in advance, these fuzzers still spend substantial effort generating unreachable inputs.

This paper presents an effective directed greybox fuzzing technique to address the *indirect input generation* problem by efficiently generating more inputs aimed toward the targets. Our key insight is that historically executed inputs, either reachable or unreachable to the target, can be used to approximate conditions for reaching the targets, thus minimizing the search space of subsequent input generation. With more inputs reaching the target, our fuzzer has a higher chance of triggering the target vulnerabilities. Specifically,

executed inputs can be regarded as a sample distribution from the search space described by path conditions. If the exact distribution can be inferred, the fuzzer can only generate inputs satisfying the conditions to reach the target. For example, if the fuzzer can infer the relation of $flag = 2a \wedge flag < 20$ when reaching Line 11 in Figure 1 based on observed values of a in various executed inputs, it could enforce subsequent inputs to be constrained by $a < 10$ and only explore relevant branches toward the target at Line 14. Moreover, even with a less precise relation, e.g., $a < 30$, fuzzers still filter unreachable inputs from $a \geq 30$.

To accurately deduce the conditions required to reach the desired targets, we dynamically utilize the executed inputs, both reachable or unreachable toward the target, to infer likely invariants approximating the condition. Invariants derived from reachable inputs approximate the path condition to the target. On complementary, invariants inferred from unreachable inputs indicate the condition subsequent inputs should not satisfy. To efficiently generate inputs that can satisfy the target condition, we have devised two strategies that optimize the deduction of the likely invariant: distance-based input selection and similarity-based invariant selection. The former strategy aims to reduce the number of inputs needed for inferring the precise invariants efficiently, while the latter eliminates imprecise invariants to enhance input generation effectiveness.

We implement our design of this input-constraining directed greybox fuzzer in Halo. Our evaluation uses the Magma [20] benchmark to compare Halo against four state-of-the-art directed fuzzers: AFLGo [10], Beacon [17], WindRanger [15], and SelectFuzz [16]; and four state-of-the-art non-directed fuzzers including AFL [21], AFL++ [22], ParmeSan [23], and SymCC [24]. The results demonstrate that Halo can generate 6.2x more reachable inputs during the fuzzing process. Therefore, Halo significantly outperforms existing efforts in reproducing specific vulnerabilities with a 15.3x speedup. Halo also triggers 18 more targets in Magma that, so far, other evaluated fuzzers cannot detect. Moreover, Halo also proves great practicality in real-world scenarios by finding ten previously undiscovered bugs involving seven incomplete fixes in the newest version of the projects, five of which cannot be detected by existing fuzzers.

To sum up, we make the following key contributions:

1) We propose to utilize the executed inputs to dynamically infer likely invariants for constraining input search space, thus speeding up the vulnerability reproduction process in directed fuzzing.
2) We design two novel selection strategies to enhance the input generation for efficiently reducing the proportion of the irrelevant input generated.
3) We provide empirical evidence that our approach designed is more efficient and effective than the state-of-the-art (directed) fuzzers.

2. Motivation

To better demonstrate the problem and our motivation, we first summarize the intuition of existing work and justify their weaknesses in Section 2.1. Subsequently, we outline our motivation to address these issues in Section 2.2.

```
1 int foo(){
2   char* buf=input();
3   unsigned a,b,c,d=input();
4   int flag = 0;
5
6   for(int i=0;i<a;i++){
7     flag+=2;
8     b+=1;
9   }
10
11  if(flag<20){ //a<10
12    if(b>10){ //a+b>10
13      if(c<20){ //c<20
14          buf[b+c];//overflow when a+b+c>=sizeof(buf)
15      }
16    }
17  }
18  if(d>3){
19    ... //target-irrelevant
20  }
21 }
```

Figure 1: Motivating example.

2.1. Existing Directed Fuzzers

The main intuition of directed fuzzing is only to examine the program behaviors defined by the targets. To achieve this purpose, there are two main trends in state-of-the-art directed fuzzing. One is improving the directness by selecting the most promising seeds close to the target. The other is culling the infeasible execution for the inputs that cannot reach the targets.

Sophisticated Seed Scheduling The majority of existing efforts [10], [12], [13], [14] pursue selecting the seeds with the highest probability of detecting the target behavior. They propose different levels of granularity for the fitness function to measure the likelihood of reaching the targets. Specifically, the key intuition is to select the seed closest to the targets, which requires precise distance measurement. AFLGo [10] introduced the idea of prioritizing seeds with minimal distance to the target in the control-flow graph. Hawkeye [13] optimizes this distance metric by considering the average "call-trace-distance" obtained from the crash report. CAFL [12] further refines the call trace with the necessary instructions to trigger the crash. WindRanger [15] proposes to refine the fitness by only collecting the distance from the control-flow deviation blocks. Then, SelectFuzz [16] calculates distance only from the variables and blocks through the data dependence. Meanwhile, another intuition is to measure the difficulties of reaching the targets. MC2 [14] transforms directed fuzzing into a Monte-Carlo counting model, which uses the execution frequency of each branch to approximate the difficulties in reaching the targets and, thus, prioritizes seeds with the lowest difficulties.

Although various priority-based approaches can help directed fuzzing reach the target faster, the majority of the inputs generated still rely on random mutation, with only a few prioritized seed inputs. Consequently, randomly

generated inputs may cause directed fuzzing to waste effort executing them irrelevant to the target.

Culling Infeasible Executions Some fuzzer-created inputs may never reach the target area. To reduce the cost of executions, fuzzers may stop execution as soon as it becomes clear that the target can no longer be reached [17], [18], [19]. Their intuition is to identify the execution that cannot reach the targets and terminate them as early as possible. To this end, FuzzGuard [18] utilizes a machine-learning model to predict whether the inputs can reach the targets. Beacon [17] infers the necessary preconditions to reach the targets and terminate the execution once the precondition is violated. SieveFuzz [19] proposes to refine the unreachable paths that can be pruned adaptively according to the dynamic feedback to overcome the indirect call issues. While rejecting a vast number of infeasible inputs can improve the efficiency of directed fuzzing, it is still a compensatory measure since there is no indication to generate inputs directly targeted toward the target.

2.2. Example and Key Challenges

Even though there are tremendous efforts committed to directed fuzzing, the *indirect input generation* problem could still hinder their effectiveness of reproducing the targets.

On the one hand, some targets could be challenging to reach even if most seeds are prioritized by distance [10], [12], [13], [14]. In Figure 1, suppose we have three seeds, $A(a,b,c,d)$: $(15,5,10,10)$, $B(a,b,c,d)$: $(5,5,10,10)$, and $C(a,b,c,d)$: $(15,5,10,0)$. Even though existing fuzzers can prioritize seed B over seeds A and C since B can reach Line 12, which is closer than seeds A and C reaching Line 11, randomly mutating the seeds may not quickly satisfy the tight path conditions for a, b, and c from Lines 11-13, not to mention satisfying the overflow condition, $a + b + c > sizeof(buf)$. Although seed B is prioritized, the fuzzer could still generate an enormous number of inputs violating the path condition, leading to exploring irrelevant programs, e.g., block at Line 18.

On the other hand, culling infeasible execution leading to the targets does not help generate the input toward the targets [17], [18]. For example, Beacon [17] could terminate the execution based on reachability and path conditions. Therefore, the fuzzer can stop executing inputs once the execution reaches Line 18. Nevertheless, the fuzzer still lacks the knowledge of how to generate input satisfying the condition at Lines 11-14.

Motivation The key reason for the existing deficiency in directed fuzzing is that *input generation* is oblivious to *path conditions to the target*. Our observation is that, even though existing approaches optimize seed selection with additional feedback or cull unnecessary execution, we must specialize input generation for directed fuzzing. So far, existing work has missed this optimization opportunity. Specifically, existing directed fuzzers only retain little feedback from executing the seeds for prioritization, accounting for a minor part of the generated inputs. Most of the inputs are discarded after execution. For example, Table 1 shows that the state-of-the-art directed fuzzer, AFLGo, retains feedback from fewer than 0.01% of the executed inputs on the Magma benchmark [20]. Consequently, AFLGo cannot effectively reproduce most of the targets (19 out of 138 targets from the benchmark), according to our evaluation in Section 4.1. Therefore, our idea is to utilize the executed inputs for approximating the target condition, which can constrain the search space for subsequent input generation.

Table 1: The numbers of inputs and seeds by fuzzing the Magma benchmark with a 24-hour budget using AFLGo.

Project	$Num_{Execution}$	Num_{Seed}	$\frac{Num_{Seed}}{Num_{Execution}}$
libpng	3.24E+07	676	2.09E-05
libsndfile	1.00E+08	2638	2.63E-05
libtiff	1.12E+08	3125	2.79E-05
libxml2	1.81E+07	5457	3.01E-04
lua	8.01E+06	2068	2.58E-04
openssl	5.60E+06	3279	5.86E-04
poppler	4.57E+06	10514	2.30E-03
sqlite3	8.45E+07	4679	5.54E-05
php	2.77E+08	2475	8.94E-06
Avg.	7.14E+07	3879	3.98E-04

We use Figure 2 to illustrate our basic intuition. Inputs can be regarded as samples from the distribution described by various program paths, whose path conditions are boundaries that differentiate the paths. Benefiting from the efficient input generation speed in fuzzing, we infer these boundaries based on the significant number of samples. Thus, fuzzers can generate new inputs constrained by the boundary, increasing the probability of triggering crashes. For instance, by observing the inputs generated by mutating the values of a in seeds A, B, and C from the previous example, we can approximate a condition to reach the target is $a < 15$ since it is the borderline value of negating the condition at Line 11. Although this approximation is not as precise as the original path condition, $a < 10$, the fuzzer does not need to generate numerous infeasible inputs ($a \geq 15$) that cannot reach the target, which improves the effectiveness of reproducing the target vulnerability. Moreover, we can refine this approximation if a counterexample is found, i.e., $a = 10$, to further enhance practicalness.

However, there are two main challenges for efficiently approximating conditions and generating inputs:

Challenge 1. How to infer conditions from executed inputs efficiently? To achieve high efficiency, the fuzzer must infer the condition accurately with as few inputs as possible. While fuzzers are capable of generating a large number of inputs, recording execution feedback for each input can introduce significant overhead. Therefore, an effective input selection strategy is needed to choose only the necessary inputs for condition inference, minimizing the burden on the fuzzer's resources.

Challenge 2. How to generate inputs constrained by conditions efficiently? To reproduce the target effectively, the fuzzer should generate the constrained inputs efficiently

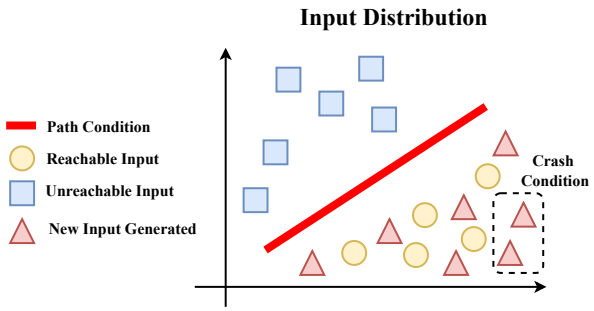


Figure 2: Motivation of Halo. The path condition approximated by the executed inputs can constrain the search space of the subsequent input generation to reproduce the target vulnerability efficiently.

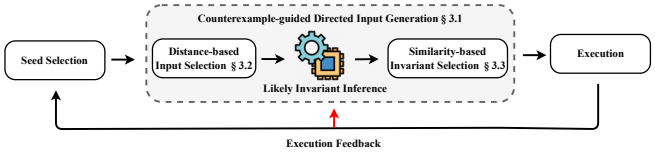


Figure 3: Workflow of Halo

with the condition inferred rather than continuously invoking the expensive constraint solver. However, the conditions could be various while not all of them are equally useful; e.g., a condition, $b > 1$, may not significantly constrain the search space for b compared to the condition, $a < 10$, for a in Figure 1. Making the condition unnecessarily complex could not only slow down the input generation but also cause the generated inputs to miss the targets. Therefore, it is essential to efficiently identify the usefulness of inferred conditions to improve the quality of input generation.

3. Counterexample-guided Directed Fuzzing

Following the motivation in Section 2.2, we present Halo, a directed fuzzer to effectively generate inputs leading to the target guided by constraining the input generation using the approximated conditions. As illustrated in Figure 3, our core functionality is to infer a series of likely invariants to effectively approximate the path condition based on the executed inputs generated by the fuzzers (Section 3.1). To efficiently infer invariants (**Challenge 1**), we reduce the number of inputs the inference engine needs by selecting the representative inputs that can still precisely infer invariants (Section 3.2). To efficiently generate the inputs satisfying the likely invariant (**Challenge 2**), we deprioritize the invariant that cannot precisely describe the path condition based on the adaptive execution feedback from fuzzing to increase the probability of generating feasible inputs (Section 3.3).

3.1. Intuition: Approximating the Condition with Dynamic Likely-Invariant Inference

From a high-level view, our approach optimizes the input generation phase for directed fuzzing with additional execution feedback, as illustrated in Algorithm 1. When the fuzzer starts to generate inputs with the selected seed (Line 2), Halo first approximates conditions leading to the targets based on a set of inputs generated from the chosen seed (Lines 3-8). Intuitively, the path condition can be regarded as invariants that should be satisfied by multiple input bytes. Therefore, our basic intuition is to find an invariant that all the given inputs can satisfy as the approximation of the target condition. An invariant is a property that holds at a certain point or points in a program, which is represented by the conditions among variables, e.g., being a constant, $a = 10$, or linear relation, $a + b > +10$. To simplify the computation, we only calculate invariants for input bytes, which remains sufficient for input generation.

Algorithm 1 Counterexample-guided Directed Fuzzing

Require: Seeds S, Target t
1: **repeat**
2: $s, I_{use} \leftarrow ChooseNext(\text{S})$
3: **if** $I_{use} = \emptyset$ or I_{use} *is not effective* **then**
4: $S_{reach}, S_{unreach} \leftarrow$ InputSelect(s, t) (§3.2)
5: $I_{reach} \leftarrow$ InvInfer(S_{reach})
6: $I_{unreach} \leftarrow$ InvInfer($S_{unreach}$)
7: $I_{use} \leftarrow$ InvSelect($I_{reach}, \neg I_{unreach}$) (§3.3)
8: **end if**
9: $Input_{new} \leftarrow Sample(I_{use})$
10: Tracing($Input_{new}$)
11: Havoc&Splice(s)
12: **until** $Timeout$ is reached

Drawing on this intuition, we propose a lightweight method inspired by the dynamic likely invariant inference [25]. The invariants serve as the approximation of the path condition to the target represented by input bytes, which can constrain the subsequent input generation.

Preliminary of Invariant Inference. Formally, given a series of inputs, S, the inference engine infers a set of invariants, I, that $\forall x \in S$, $x \models i$, where $i \in I$. We use I to approximate path conditions for reaching the target that the input bytes $b \in x$ should satisfy. Existing invariant inference approaches [25], [26], [27] apply a divide-and-conquer-style search to obtain precise likely invariants and guarantee the termination of the algorithm. Specifically, the invariant inference starts from the over-approximation in the form of a range, $[Min, Max]$, for a set of template invariants, $c_1t_1 + c_2t_2 + ... + c_nt_n \geq 0$, representing conditions with n coefficient, c, and relation, t, which consist of polynomials over program variables to approximate path conditions among them. Based on the given input, the inference engine first checks whether the input can satisfy current invariants during the execution. If so, nothing changes. Otherwise, the algorithm minimizes the range based on the counter inputs. Meanwhile, template invariants without valid ranges are discarded. Therefore, the precision of the invariant inference relies on the number of given inputs and template invariants. With more inputs provided, the algorithm gradually refines the invariants for better accuracy.

Example 3.1. Assumed the inference engine starts with $(-\infty, +\infty)$ for three template invariants, a, b, and $a+b$, to approximate the condition at Line 12 in Figure 1. Suppose a given input, $a = 5, b = 0$. The inference engine finds it cannot satisfy the condition with the execution feedback, and thus shrinks the range of the invariants as: $a \in (5, +\infty)$, $b \in (0, +\infty)$, and $a + b \in (5, +\infty)$ to increase precision.

The goal of this work is not to improve the invariant inference algorithm but instead to explicitly apply invariant inference to improve directed fuzzing. Specifically, we apply the state-of-the-art invariant inference engine [26] to infer invariants based on the set of given inputs (Lines 5-6 in Algorithm 1), which indicates the potential relations among input bytes. The initial template invariants we employed include constant checking, lower bound examination involving a single variable, and linear inequation for co-dependency among multiple variables, which is widely used as the default setting in existing invariant inference techniques [25], [26], [28], [29]. It is worth noting that selecting the ideal template invariant remains an unsolved optimization opportunity in dynamic invariant inference, which is not the main problem we tackle in this paper. We present more details about existing efforts in Section 5.

Input Collection. Taking advantage of the large number of inputs generated in fuzzing, Halo infers such invariants to improve further input generation. Specifically, Halo records the input bytes and their values influencing reaching the targets (Line 4 in Algorithm 1). We use the execution feedback indicating whether the input reaches the targets to cluster them as S_{reach} and $S_{unreach}$. Section 3.2 presents details on collecting the values for the relevant input bytes.

Example 3.2. Considering the example in Figure 1 with five inputs not triggering the crashes, $A(a, b, c, d)$: $(15, 5, 20, 10)$, $B(a, b, c, d)$: $(6, 8, 10, 10)$, $C(a, b, c, d)$: $(12, 5, 30, 0)$, $D(a, b, c, d)$: $(6, 10, 12, 0)$, and $E(a, b, c, d)$: $(6, 100, 12, 0)$. Halo only records the inputs bytes influencing the variables a, b, and c and its values as:

$$S_{reach} = B : (6, 8, 10), D : (6, 10, 12), E : (6, 100, 12)$$
$$S_{unreach} = A : (15, 5, 20), C : (12, 5, 30)$$

Invariant Inference. With the clustered inputs, we infer the invariants, I_{reach} and $I_{unreach}$, from each cluster of inputs (Lines 5-6 in Algorithm 1), respectively. I_{reach} indicates the approximated conditions reaching the target. Complementary, $I_{unreach}$ measures the conditions that inputs may not reach the target. Therefore, the newly generated input should satisfy I_{reach} and the negation of $I_{unreach}$.

Example 3.3. Using the inputs from the previous example, assume we may obtain the invariants as follows:

$$I_{reach} = a < 7, b > 7, c \leq 10$$
$$I_{unreach} = a > 12, b < 6, c \geq 20$$

I_{reach} could indicate the condition reachable inputs should satisfy. Moreover, combining with the negation of invariants generated from the unreachable inputs, $I_{unreach}$, we can have more precise conditions that the inputs should satisfy through refining invariants for c:

$$a < 7, a \leq 12, b > 7, b \geq 7, c \leq 10, c \leq 20$$

Overall, compared with random mutation, we can avoid generating unreachable inputs from $a > 12$ and $c > 20$.

Remark 1. Instead of leveraging conventional symbolic execution to precisely reason path conditions, inferred invariants are a lightweight approximation, though less precise than actual path conditions. Still, they are sufficient for effectively filtering unreachable input generation leveraging invariants inferred by both reachable and unreachable inputs. Utilizing only execution feedback and inputs also has better scalability when tackling complex program behaviors such as loops. Additionally, unlike existing dynamic invariant inference approaches, which suffer from the scalability issue of the significant runtime overhead introduced by recording intermediate values during execution [26], we only infer the relations among input bytes to address the *indirect input generation* problem, thereby reducing overhead.

Despite the effectiveness of filtering unreachable inputs with the inferred invariant, it can be prohibitively expensive to use invariant inference in fuzzing when dealing with large input spaces with 1) *explosive inputs* and 2) *complex relations*. Therefore, we aim to minimize the dimension explosion caused by these two primary factors in invariant inference to improve its scalability for fuzzing.

3.2. Efficient Invariant Inference with Distance-based Input Selection

While increasing the number of inputs can improve the accuracy of the inferred likely invariant, inferring the invariant with a significant number of inputs is time-consuming. Moreover, the deficiency issue aggravates when generating many random inputs, as most of them cannot help the fuzzer reach the target.

To efficiently infer the precise invariant, our goal is to use a minimized set of representative inputs that accurately portray the target condition. Specifically, we observe that inputs contribute differently to the precision of the invariant inference, thus motivating us to minimize the number of inputs needed for invariant inference.

Example 3.4. Considering the inputs in the previous Example 3.2, we notice that the input E does not change the inference results since it is subsumed by the input D. Therefore, we do not need to include this input for invariant inference.

However, quantifying the effectiveness of inputs for invariant inference is challenging. To tackle this issue, we regard it as a data clustering problem for the input distribution, where only inputs close to the boundary of the condition can aid in inferring the invariant. Inputs that are far from the boundary cannot effectively describe the target condition; for example, inputs outside the red lines shown in Figure 4 are less likely to produce a precise invariant than

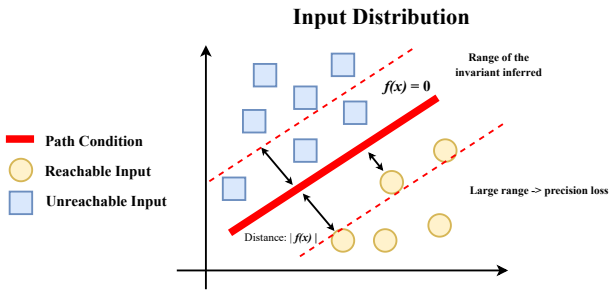


Figure 4: Intuition for distance-based input selection: Inputs near the boundary approximate the precise invariant.

those inside the lines. Therefore, we can select inputs based on their distance from the boundary of the condition.

How to Select Inputs. Based on this intuition, we design a distance-based input selection strategy to choose the inputs that could infer a more precise invariant. The distance metric describes the closeness of each input to the boundary of the condition. As shown in Figure 2, the path condition can be regarded as a function $f(x)$, where x is the vector of related input bytes. The inputs making $f(x) = 0$ consist of the boundary of the condition. The distance to the boundary can be represented as $|f(x)|$. Therefore, the inputs with small $|f(x)|$ should be chosen for the invariant inference.

Specifically, before the first time mutating each seed, we follow the procedure of *inference-based taint analysis* [30], [31], [32], [33], [34] to obtain the input bytes that influence reaching the targets by sequentially mutating each byte to see whether it can change the values of the target points. We leave potential customization and fine-tuning of taint analysis as future work. During the analysis, we also record the values of these related bytes and $|f(x)|$. To facilitate the computation, we use the sum of values at all branch conditions not in the loop that occurred in the execution. Thus, Halo can preserve a set of inputs closer to the targets with a small $|f(x)|$.

Example 3.5. Suppose we want to choose three inputs from inputs used in Example 3.2, $B(a,b,c,d)$: $(6,8,10,10)$, $D(a,b,c,d)$: $(6,10,12,0)$, and $E(a,b,c,d)$: $(6,100,12,0)$ for the program in Figure 1. We first calculate the $|f(x)|$ as the distance for each input based on the conditions at Lines 11-13:

$$|f(B)| = 20 - 12 + 14 - 10 + 20 - 10 = 22$$
$$|f(D)| = 20 - 12 + 16 - 10 + 20 - 12 = 22$$
$$|f(E)| = 20 - 12 + 106 - 10 + 20 - 12 = 112$$

Since the distance indicates that input E is far from the boundary of the condition, we only select B and D as the representative inputs for the invariant inference.

Nevertheless, we cannot obtain the distance for inputs that do not reach the targets since $f(x)$ does not entirely execute. Therefore, Halo separates the inputs for invariant inference based on whether they are reachable to the targets. Before the target is reached, Halo leverages only unreachable inputs to infer likely invariants for all input bytes, which indicates the condition that reachable inputs should not satisfy. By negating invariants inferred from unreachable inputs, the fuzzer can still constrain the search space for input generation.

Example 3.6. Suppose we only have the invariants, $I_{unreach}$:$\{a > 12, b < 6, c > 20, d < 10\}$, inferred from the unreachable inputs in Example 3.2. Even though we may not filter the reachable inputs when $a \in \{10, 11, 12\}$ utilizing the negated invariant, $a \leq 12$ from $a > 12$ in $I_{unreach}$ as $a < 7$ in I_{reach} does, the fuzzer could still effectively filter inputs from $a > 12$ and $c > 20$.

How Many Input Selected. We determine the number of reachable and unreachable inputs used in the invariant inference based on the statistic theory [35], which measure the minimum samples needed, n, to approximate the whole distribution as: $n = \frac{4Z^2}{\varepsilon^2} = 385$ where Z is a constant Z-score determine by the margin of error ε, which is 0.05 for the 0.95 confidence level.

Remark 2. Unlike existing directed fuzzing approaches that merely store intermediate states using seeds, Halo preserves more execution feedback from executed inputs to constrain the search space for subsequent input generation. On top of this intuition, the input selection aims to strike a balance between the additional time costs for data collection and the precision of the inferred invariants. Moreover, utilizing the invariants inferred from both reachable and unreachable inputs not only enhances the scalability of the method, but also helps improve precision by squeezing the approximate search space from both directions.

3.3. Effective Input Generation with Similarity-based Invariant Selection

Although likely invariants can help constrain the search space for subsequent input generation, the inference engine could provide multiple potential invariants with varying precision, which may not filter the unreachable input equally. Invariants may be over-constrained based on the given inputs, e.g., invariants inferred exclusively from the unreachable inputs at an early stage. Consequently, using all the invariants for input generation could introduce a significant overhead without providing sufficient precision.

Example 3.7. Consider the invariant inferred in Example 3.3. Compared with the invariant, $c < 10$, $c \leq 20$ could describe the path condition, $c < 20$ at Line 13 more precisely. Meanwhile, invariant $a < 7$ is over-constrained since the reachable values of $a \in \{7, 8, 9\}$ are filtered.

To efficiently generate inputs constrained by the invariant, our basic intuition is to adaptively select and refine the invariant based on new input generation, as shown in Figure 5. Ideally, inputs generated by the most precise invariant should always satisfy the path condition. The precision loss is represented by the counterexamples found in the approximated distribution. A more precise invariant should generate fewer counterexamples. Therefore, we utilize the execution

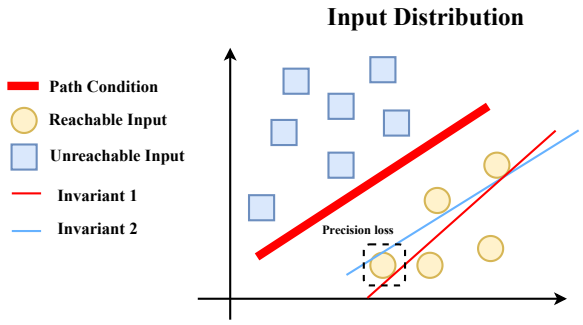


Figure 5: Intuition for similarity-based invariants selection: Search space described by precise invariants should generate more reachable inputs.

feedback of generated inputs to estimate which invariant is more similar to the original path condition. The similarity can be viewed as the probability of generating more reachable inputs using inferred invariants. Halo prioritizes inferred invariants based on the ratio of the counterexample found in the subsequent input generation. Therefore, the similar search space described by the selected invariants can improve the effectiveness of generating reachable inputs.

How to Select Invariants. Specifically, we design a similarity-based invariant selection approach to adaptively choose the most effective invariants for reachable input generation. The similarity of each invariant is served as the probability of the invariant selected by the fuzzer. An invariant with higher similarity should have a higher probability of being chosen by the fuzzers. Specifically, we use the proportion of the reachable inputs generated using the invariant, Φ, as its similarity:

$$Similarity(\Phi) = \frac{n}{N}$$

where n and N are the reachable and total input generated from the invariant, Φ, respectively.

Example 3.8. Suppose we use the invariants obtained from the previous Example 3.3 to generate new inputs. Halo could rapidly find the condition, $b \geq 7$, could be over-constrained since the value of b in the subsequent inputs generated should also depend on the value of a, according to the condition at Line 12 in Figure 1. Therefore, Halo decreases the priority of this invariant based on the new counterexamples to avoid generating counterexample input not reaching the target.

Input Generation with Adaptive Invariants. Once we select the invariant, we sample inputs that satisfy the conditions. Intuitively, since our inferred invariants are linear, they can form a polytope that describes the search space of reachable inputs. Therefore, we use the state-of-the-art sampling approach, Vaidya walk [36], which efficiently samples from a polytope to generate inputs within linear complexity based on the number of selected invariants. Furthermore, the similarities of each invariant are continuously updated based on subsequent input generation during the fuzzing process. If all inferred invariants could not effectively generate reachable inputs with low priorities, Halo collects the discovered counterexamples and infers a new set of invariants with better precision to maintain the effectiveness in input generation.

Remark 3. We address input generation in directed fuzzing from the perspective of the search space, which allows us to efficiently sample reachable inputs without solving precise path conditions. Meanwhile, this intuition motivates us to refine the imprecise search space by utilizing the executed inputs as a regression. Since invariants are approximations of actual path conditions, the potential loss of precision could imprecisely enlarge or shrink the approximated search space and, thus, hinder their effectiveness in generating reachable inputs. Moreover, instead of potential runtime overhead caused by utilizing dynamic program analysis in existing invariant inference [26], [28], [37], [38], we take advantage of the tremendous inputs available in fuzzing, which approximate the distribution of the search space, to serve as a lightweight oracle for selecting the precise invariants. Therefore, we can reduce the number of imprecise or redundant invariants selected to generate reachable inputs more efficiently during the fuzzing process.

4. Evaluation

We implemented Halo, a greybox fuzzer with an adaptive condition inference engine to generate the inputs leading to the target effectively. By default, we use AFL++ [22] as the fuzzing engine. The condition inference engine is based on existing work, DIG [26].

Based on the implementation, we design a series of evaluations to demonstrate the effectiveness of Halo by answering the following research questions:

- **RQ1**: Can Halo reproduce the target vulnerabilities faster compared with state-of-the-art (directed) fuzzers?
- **RQ2**: Can Halo effectively tackle the *indirect input generation* problem?
- **RQ3**: Can Halo practically detect incomplete fix and new bugs?
- **RQ4**: Can selection strategies designed in Halo enable invariant inference assist directed fuzzing effectively?

To answer RQ1, we evaluate the reproduction time of the target vulnerabilities compared with state-of-the-art (directed) fuzzers in Section 4.1, which is the primary goal of directed fuzzing. To answer RQ2, we evaluate the proportion of reachable inputs generated during reproducing the target vulnerabilities and the additional time cost brought by invariant inference in Section 4.2. To answer RQ3, we argue the practicalness of our designed methods by demonstrating the incomplete fix and new bugs found by the Halo in Section 4.3. To answer RQ4, we provide ablation studies on each component of our designed system, named distance-based input selection and similarity-based invariant selection, to better understand the benefits of the two strategies in Section 4.4.

Table 2: Compared fuzzers.

Fuzzer	Category	Description
AFLGo	Directed	Sophisticated seeds prioritization
Beacon	Directed	Rejecting infeasible execution
WindRanger	Directed	Seeds prioritization based on deviation blocks
SelectFuzz	Directed	Seeds prioritization based on data-dependent code
AFL	Coverage	Evolutionary mutation strategies
AFL++	Coverage	AFL with multiple optimizations from the community
Parmesan	Coverage	Guide the fuzzer using the sanitizer labels
SymCC	Coverage	Integrated concolic execution with the fuzzer

Table 3: Magma benchmark

Project	Version	Size (CLOC)	Input format	Num. CVEs
libpng	1.6.38	95K	PNG	7
libsndfile	1.2.0	83K	Various	18
libtiff	4.1.0	95K	TIFF	14
libxml2	2.9.10	320K	XML	17
openssl	3.0.0	630K	Binary	20
poppler	0.88.0	260K	PDF	22
sqlite3	3.32.0	387K	SQL	20
lua	5.4.0	31K	LUA	4
php	8.0.0-dev	1.6M	Various	16

Baseline. We compare Halo with the fuzzers mentioned in Table 2. AFLGo [10], Beacon [17], Windranger [15], and SelectFuzz [16] are four state-of-the-art directed greybox fuzzers. We set the last instruction where the crashes occur as the targets for these two directed fuzzers. As their prototype implementations are not publicly available, we, unfortunately, cannot compare Halo to the directed fuzzers FuzzGuard, Hawkeye, or CAFL. While we contacted the authors, we did not receive feedback yet. Similarly, the MC2 [14] prototype is restricted to the core algorithm without an executable prototype, which will be released in the future, as mentioned by the author through our email. We also do not compare against the results in the MC2 paper as we were unable to reproduce the baselines of the existing fuzzers in our environment. Should the prototypes become available, we will compare them against these other fuzzers.

Moreover, we also compared Halo with other coverage-based approaches that optimize the input generation as our baseline for a more reliable comparison. We pick AFL [21] and AFL++ [22], the fundamental greybox fuzzer and its updated versions, as the baseline. Parmesan [23] is a bug-driven fuzzer that uses the program points labeled by the sanitizers [39] as potential target points. Unlike existing directed fuzzing, Parmesan cannot ensure whether the labeled targets can lead to crashes. Thus, it provides a general exploring strategy similar to coverage-guided fuzzing. SymCC [24] is a hybrid fuzzing approach integrated with symbolic execution to generate the inputs for covering those complex path conditions. Their technical details are mentioned in Section 6 for other related work discussions.

Benchmark. To answer our research questions, we use the state-of-the-art fuzzing benchmark Magma [20]. Magma consists of various CVEs chosen across nine benchmark programs frequently evaluated in the state-of-the-art fuzzing with diverse functionalities shown in Table 3. For each vulnerability, we follow the instructions according to the benchmark to set the last instruction where the crash is triggered as the target. Meanwhile, ParmeSan does not work with `SQLite3` and `PHP`. WindRanger does not work with `Libxml2`, `lua`, `SQLite3`, or `PHP` While we have reached out to the developers, they deferred to future improvements that could not be integrated into this evaluation. Therefore, we exclude those targets from the benchmark to compare against ParmeSan and WindRanger. For other projects, we follow the instructions of the benchmark to configure each fuzzer with the given seeds[1] and set the crash point of each CVE as the targets for directed fuzzers.

To demonstrate the practicality, we also evaluate the effectiveness of Halo for efficiently reproducing real vulnerabilities and detecting incomplete fixes. Specifically, we reproduce the CVEs in the newest version of the program used in Magma.

Configuration. The initial seed corpus determines the effectiveness of fuzzing [40]. To achieve the best performance of related work, we use the seeds provided in the Magma benchmark. In general, we conducted every experiment 10 times. For each time, the experiment was run with a time budget of 24 hours indicated by the benchmark [20]. We employ the Mann-Whitney U Test [41] to demonstrate the statistical significance of the contribution made by our framework. Some extra configuration details are mentioned in the related experiments, respectively.

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.

4.1. Vulnerabilities Reproduction Ability

The primary goal of directed fuzzing is to efficiently reach and trigger a given location, e.g., to reproduce specific vulnerabilities. Therefore, we compare the reproduction speed of Halo with the state-of-the-art fuzzers using the Magma benchmark. We utilize the average time cost of reaching and reproducing the target by the fuzzer with 24-hour time budgets for each run. For Halo, the results are the accumulated time for both fuzzing and invariant inference.

Table 4 and Table 5 show our results across all targets found by the evaluated fuzzers. The significant difference ($>$1000x on average) between the reach and the trigger time in both state-of-the-art (un)directed fuzzers reveals the necessity of solving the indirect input generation problem. Fortunately, Halo outperforms the existing efforts with an average 3.4x and 15.3x speedup in reaching and reproducing the target vulnerabilities, respectively. Overall, Halo requires 2.0x less time to reproduce the targets after they have been reached compared with existing fuzzers, demonstrating the effectiveness of mitigating the indirect input generation problem. Moreover, Halo discovers all the targets that other fuzzers achieve and detects an additional 18 targets on average.

Compared with directed fuzzers, AFLGo, Beacon, WindRanger, and SelectFuzz, Halo demonstrates a significant

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

Table 4: Reproduction time for each target in the Magma benchmark compared with directed fuzzers. R and T indicate the reachable and the reproduction time (second) averaged over ten runs, respectively. $T.O.$ 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 Halo. $\emptyset$ indicates that the fuzzer cannot deploy in the project.

Bug ID	Halo		AFLGo				Beacon				WindRanger				SelectFuzz			
	R	T	R	T	$Ratio$	p	R	T	$Ratio$	p	R	T	$Ratio$	p	R	T	$Ratio$	p
PNG003	10	15	10	15	1.0x	-	10	20	1.3x	0.02	10	15	1.0x	-	10	15	1.0x	-
PNG006	10	72	15	*T.O.*	*N.A*	-	15	98	1.4x	0.02	15	*T.O.*	*N.A*	-	15	*T.O.*	*N.A*	-
PNG007	15	26350	15	68058	2.6x	<0.01	15	57100	2.2x	<0.01	15	74446	2.8x	0.01	15	44576	1.7x	<0.01
SND001	345	815	55	*T.O.*	*N.A*	-	15	536	0.7x	-	15	61594	91.3x	<0.01	36	7509	9.2x	<0.01
SND005	15	36	10	10357	287.7x	<0.01	10	42	1.0x	0.04	15	3556	98.7x	<0.01	10	*T.O.*	*N.A*	-
SND006	15	1711	55	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	15	72742	42.5x	<0.01	36	*T.O.*	*N.A*	-
SND007	183	1842	55	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	15	73001	39.6x	<0.01	36	*T.O.*	*N.A*	-
SND017	876	1053	257	8336	7.9x	<0.01	1240	1352	1.3x	0.01	10	125	0.1x	-	93	3040	2.9x	0.01
SND020	1757	1955	302	78007	39.9x	<0.01	2475	3491	1.8x	<0.01	61	4176	2.1x	0.02	95	*T.O.*	*N.A*	-
SND024	15	1703	55	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	15	72743	42.7x	<0.01	36	*T.O.*	*N.A*	-
TIF002	24050	48565	1104	*T.O.*	*N.A*	-	59791	*T.O.*	*N.A*	-	1497	*T.O.*	*N.A*	-	62886	84897	1.7x	<0.01
TIF005	1249	1899	*T.O.*	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-
TIF006	4695	5868	79175	82787	14.1x	<0.01	38939	52149	8.9x	<0.01	42325	47876	8.2x	<0.01	*T.O.*	*T.O.*	*N.A*	-
TIF007	18	159	77	18252	114.8x	<0.01	47	250	1.6x	0.01	169	6839	43.0x	<0.01	38	301	1.9x	0.02
TIF009	29399	32051	62564	74482	2.3x	<0.01	41249	53560	1.7x	<0.01	52361	59910	1.9x	<0.01	52389	75960	2.4x	<0.01
TIF012	10	2862	10	33250	11.6x	<0.01	5	3686	1.3x	<0.01	10	17396	6.1x	0.02	10	1568	0.5x	-
TIF014	125	2074	840	57240	27.6x	<0.01	47	4709	2.3x	<0.01	169	68943	33.2x	<0.01	10	74607	36.0x	<0.01
XML001	21	2613	15	*T.O.*	*N.A*	-	15	*T.O.*	*N.A*	-			$\emptyset$		15	*T.O.*	*N.A*	-
XML003	15	22501	15	*T.O.*	*N.A*	-	15	75004	3.3x	<0.01			$\emptyset$		15	*T.O.*	*N.A*	-
XML009	10	4573	15	*T.O.*	*N.A*	-	15	6077	1.3x	<0.01			$\emptyset$		15	*T.O.*	*N.A*	-
XML017	15	21	10	22	1.0x	0.04	10	46	2.2x	0.01			$\emptyset$		15	*T.O.*	*N.A*	-
SSL001	15	24237	*T.O.*	*T.O.*	*N.A*	-	20	33937	1.4x	<0.01	831	*T.O.*	*N.A*	-	20	*T.O.*	*N.A*	-
SSL003	15	153	*T.O.*	*T.O.*	*N.A*	-	15	527	3.4x	<0.01	15	211	1.4x	0.03	45	*T.O.*	*N.A*	-
SSL020	15	63109	*T.O.*	*T.O.*	*N.A*	-	15	83335	1.3x	<0.01	15	*T.O.*	*N.A*	-	15	21626	0.3x	-
PDF003	15	49733	31	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	15	52761	1.1x	0.02	20	*T.O.*	*N.A*	-
PDF010	20	5806	17	17536	3.0x	<0.01	20	7836	1.3x	0.01	20	*T.O.*	*N.A*	-	15	2157	0.4x	-
PDF014	20	84172	33	*T.O.*	*N.A*	-	28	*T.O.*	*N.A*	-	145	*T.O.*	*N.A*	-	30	*T.O.*	*N.A*	-
PDF016	10	37	10	258	7.0x	0.01	10	112	3.0x	0.03	10	3319	89.7x	0.01	10	167	4.5x	0.01
PDF018	27340	39561	*T.O.*	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-
PDF021	15	64442	20	*T.O.*	*N.A*	-	20	83251	1.3x	<0.01	20	*T.O.*	*N.A*	-	15	*T.O.*	*N.A*	-
SQL002	694	5446	36732	62739	11.5x	<0.01	403	10209	1.9x	<0.01			$\emptyset$		3667	51191	9.4x	<0.01
SQL014	17986	48253	68916	80432	1.67x	<0.01	19079	75868	1.6x	<0.01			$\emptyset$		*T.O.*	*T.O.*	*N.A*	-
SQL018	5067	31595	56886	76433	2.4x	<0.01	4003	45515	1.4x	<0.01			$\emptyset$		59283	72405	2.3x	<0.01
LUA004	12741	16193	*T.O.*	*T.O.*	*N.A*	-	198	*T.O.*	*N.A*	-			$\emptyset$		*T.O.*	*T.O.*	*N.A*	-
PHP004	10	121	15	1772	14.6x	<0.01	15	3980	32.9x	<0.01			$\emptyset$		15	187	1.5x	0.02
PHP009	15	4862	15	11469	2.4x	<0.01	15	5718	1.2x	<0.01			$\emptyset$		15	3349	0.7x	-
PHP011	15	621	15	2087	3.4x	<0.01	15	869	1.4x	0.02			$\emptyset$		10	3004	4.8x	0.01
Avg.			**>3.9x**		**>28.9x**		**>1.3x**		**>3.1x**		**>4.1x**		**>28.8x**		**>1.6x**		**>4.8x**	

improvement by reproducing the targets 28.9x, 3.1x, 28.8x, and 3.8x faster on average, with 18, 10, 19, and 20 more targets found, respectively, indicating the effectiveness of our design in tackling the *indirect input generation* problem using invariant inference. Meanwhile, Halo can reach the targets faster compared with directed fuzzers with a 1.3x 9.0x speedup, demonstrating the effectiveness of reaching the target based on invariants inferred exclusively using unreachable inputs.

Furthermore, compared to non-directed fuzzing, Halo achieves detecting 17 more targets with 14.3x faster reproduction speed averaged across ten runs. Compared to the SymCC hybrid fuzzer, we notice that even though SymCC rapidly triggers some targets based on its concolic execution, e.g., SND017 and SSL020, its inherent scalability issues limit its effectiveness in reproducing bugs in large programs, such as PHP and SQLite. This deficiency underscores the importance of leveraging a lightweight method, i.e., the likely invariant inference used in Halo, to address the path condition for generating inputs.

4.2. Effective Input Generation and Its Time Cost

We conducted further analysis to investigate the reason for the significant improvement achieved by Halo by examining whether we successfully addressed the *indirect input generation* problem. To this end, we evaluate the percentage of the reachable input generation during the previous evaluation and compare our results with other directed fuzzers. Since Beacon does not optimize the input generation and follows the same setting as AFLGo, according to their paper, its results should be the same as AFLGo.

Figure 6 illustrates the ratio of reachable input generation collected from various fuzzers. Our results show that Halo generates over 43% reachable inputs that reach the targets during the fuzzing process before triggering them, while AFLGo, WindRanger, and SelectFuzz generated only 5.3%, 5.8%, and 14.6% reachable inputs, respectively. This performance is 6.2x better than the state-of-the-art directed fuzzers. The significant improvement suggests that Halo has effectively mitigated the issue of *indirect input generation* commonly encountered in directed fuzzing.

These findings also underscore the importance of not only reaching the target but also triggering it effectively.

Table 5: Reproduction time for each target in the Magma benchmark compared with non-directed fuzzers. R and T indicate the reachable and the reproduction time (second) averaged over ten runs, respectively. $T.O.$ 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 Halo. $\emptyset$ indicates that the fuzzer cannot deploy in the project.

Bug ID	Halo		Parmesan				AFL				AFL++				SymCC			
	R	*T*	*R*	*T*	*Ratio*	*p*	*R*	*T*	*Ratio*	*p*	*R*	*T*	*Ratio*	*p*	*R*	*Time*	*Ratio*	*p*
PNG003	10	15	10	44	2.9x	<0.01	10	15	1.0x	-	15	21	1.4x	0.02	10	96	6.4x	<0.01
PNG006	10	72	15	87	1.2x	0.01	15	*T.O.*	*N.A*	-	10	162	2.3x	<0.01	15	1538	21.4x	<0.01
PNG007	15	26350	15	*T.O.*	*N.A*	-	15	69243	2.6x	<0.01	15	46021	1.7x	0.01	15	*T.O.*	*N.A*	-
SND001	345	815	*T.O.*	*T.O.*	*N.A*	-	88	84960	104.2x	<0.01	21	1017	1.2x	0.02	32	5235	6.4x	<0.01
SND005	15	36	10	*T.O.*	*N.A*	-	10	1633	44.1x	<0.01	66	5105	138.0x	<0.01	10	1818	49.1x	<0.01
SND006	15	1711	*T.O.*	*T.O.*	*N.A*	-	280	*T.O.*	*N.A*	-	17	15856	9.3x	<0.01	32	*T.O.*	*N.A*	-
SND007	183	1842	*T.O.*	*T.O.*	*N.A*	-	280	*T.O.*	*N.A*	-	17	4772	2.6x	<0.01	32	*T.O.*	*N.A*	-
SND017	876	1053	*T.O.*	*T.O.*	*N.A*	-	312	3148	3.0x	<0.01	1079	1107	1.1x	0.03	130	140	0.1x	-
SND020	1757	1955	*T.O.*	*T.O.*	*N.A*	-	898	*T.O.*	*N.A*	-	2322	2585	1.3x	0.01	911	*T.O.*	*N.A*	-
SND024	15	1703	*T.O.*	*T.O.*	*N.A*	-	88	*T.O.*	*N.A*	-	15	4392	2.6x	<0.01	32	*T.O.*	*N.A*	-
TIF002	24050	48565	31286	*T.O.*	*N.A*	-	694	*T.O.*	*N.A*	-	72365	80966	1.7x	<0.01	12503	*T.O.*	*N.A*	-
TIF005	1249	1899	35682	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	19950	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-
TIF006	4695	5868	43630	79631	13.6x	<0.01	78638	78942	13.5x	<0.01	21715	21715	3.7x	<0.01	70092	*T.O.*	*N.A*	-
TIF007	18	159	502	2003	12.6x	<0.01	61	4176	26.3x	<0.01	13	210	1.3x	0.02	328	12756	80.2x	<0.01
TIF009	29399	32051	*T.O.*	*T.O.*	*N.A*	-	59699	75359	2.4x	<0.01	65557	65557	2.0x	<0.01	29561	33773	1.1x	<0.01
TIF012	10	2862	5	52250	18.3x	<0.01	10	6842	2.4x	<0.01	15	4282	1.5x	<0.01	5	37607	13.1x	<0.01
TIF014	125	2074	9049	*T.O.*	*N.A*	-	61	47372	22.8x	<0.01	13	12648	6.1x	<0.01	993	53888	26.0x	<0.01
XML001	21	2613	55	*T.O.*	*N.A*	-	15	*T.O.*	*N.A*	-	22	6206	2.4x	<0.01	15	*T.O.*	*N.A*	-
XML003	15	22501	15	65379	2.9x	<0.01	15	*T.O.*	*N.A*	-	15	63381	2.8x	<0.01	15	*T.O.*	*N.A*	-
XML009	10	4573	10	70335	15.4x	<0.01	15	*T.O.*	*N.A*	-	10	4897	1.1x	0.02	15	*T.O.*	*N.A*	-
XML017	15	21	15	53	2.5x	<0.01	10	22	1.0x	0.04	15	41	2.0x	<0.01	10	21	1.0x	0.04
SSL001	15	24237	16	*T.O.*	*N.A*	-	15	*T.O.*	*N.A*	-	20	26207	1.1x	0.01	20	*T.O.*	*N.A*	-
SSL003	15	153	10	401	2.6x	<0.01	12	393	2.6x	<0.01	15	470	3.1x	<0.01	15	437	2.9x	<0.01
SSL020	15	63109	*T.O.*	*T.O.*	*N.A*	-	15	*T.O.*	*N.A*	-	15	*T.O.*	*N.A*	-	15	55520	0.9x	-
PDF003	15	49733	*T.O.*	*T.O.*	*N.A*	-	20	*T.O.*	*N.A*	-	15	82152	1.7x	<0.01	15	*T.O.*	*N.A*	-
PDF010	20	5806	*T.O.*	*T.O.*	*N.A*	-	15	13259	2.3x	<0.01	35	7512	1.3x	<0.01	15	14752	2.5x	<0.01
PDF014	20	84172	*T.O.*	*T.O.*	*N.A*	-	27	*T.O.*	*N.A*	-	20	*T.O.*	*N.A*	-	25	*T.O.*	*N.A*	-
PDF016	10	37	15	312	8.4x	<0.01	10	274	7.4x	<0.01	10	85	2.3x	<0.01	10	75	2.0x	<0.01
PDF018	27340	39561	*T.O.*	*T.O.*	*N.A*	-	*T.O.*	*T.O.*	*N.A*	-	25335	51783	1.3x	<0.01	*T.O.*	*T.O.*	*N.A*	-
PDF021	15	64442	*T.O.*	*T.O.*	*N.A*	-	15	*T.O.*	*N.A*	-	15	73116	1.1x	0.01	15	*T.O.*	*N.A*	-
SQL002	694	5446			$\emptyset$		4469	42315	7.8x	<0.01	210	15601	2.9x	<0.01	12465	*T.O.*	*N.A*	-
SQL014	17986	48253			$\emptyset$		74680	*T.O.*	*N.A*	-	5517	63845	1.3x	<0.01	*T.O.*	*T.O.*	*N.A*	-
SQL018	5067	31595			$\emptyset$		54367	74737	2.4x	<0.01	1773	41364	1.3x	<0.01	*T.O.*	*T.O.*	*N.A*	-
LUA004	12741	16193	*T.O.*	*T.O.*	*N.A*	-	75164	*T.O.*	*N.A*	-	30570	30580	1.9x	<0.01	*T.O.*	*T.O.*	*N.A*	-
PHP004	10	121			$\emptyset$		15	136	1.1x	<0.01	15	57470	475.0x	<0.01	15	*T.O.*	*N.A*	-
PHP009	15	4862			$\emptyset$		15	39842	8.2x	<0.01	15	30599	6.3x	<0.01	15	*T.O.*	*N.A*	-
PHP011	15	621			$\emptyset$		15	860	1.4x	<0.01	15	2949	4.7x	<0.01	10	*T.O.*	*N.A*	-
Avg.			**>9.0x**		**>8.0x**		**>2.8x**		**>13.5x**		**>1.7x**		**>20.3x**		**>2.7x**		**>15.2x**	

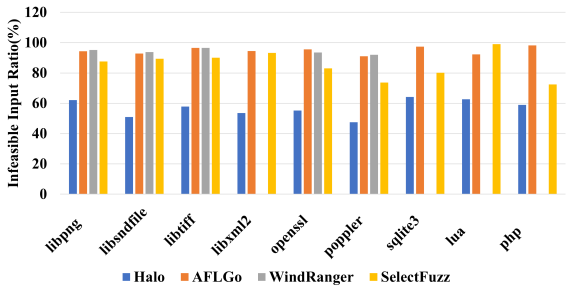


Figure 6: The average proportion of the unreachable input generated in Halo compared with other directed fuzzers when fuzzing the projects in the Magma benchmark. The y-axis is the ratio of the unreachable input generated during the evaluation.

Despite many inputs being capable of reaching the targets, triggering them is not always straightforward, as evidenced by the results in Table 1. Thus, the significant improvement in reachable input generation provided by Halo enables it to focus on triggering the targets effectively and ultimately reproducing target vulnerabilities faster than existing directed fuzzers, as demonstrated in Section 4.1.

Table 6: The average proportion of the time costs for likely invariant inference, input generation, and fuzzing engine in the 24-hour evaluation of Halo in the Magma benchmark.

Project	Inference	Sample	Fuzzing
libpng	8348 (9.7%)	398 (0.5%)	77654 (89.9%)
libsndfile	9799 (11.3%)	348 (0.4%)	76253 (88.3%)
libtiff	13329 (15.4%)	225 (0.3%)	72846 (84.3%)
libxml2	14868 (17.2%)	182 (0.2%)	71350 (82.6%)
openssl	12837 (14.9%)	659 (0.8%)	72904 (84.4%)
poppler	9963 (11.5%)	732 (0.8%)	75705 (87.6%)
sqlite3	15853 (18.3%)	537 (0.6%)	70010 (81.0%)
lua	14235 (16.5%)	549 (0.6%)	71616 (82.9%)
php	16234 (18.8%)	691 (0.8%)	69475 (80.4%)
Avg.	12829 (14.8%)	480 (0.6%)	73090 (84.6%)

Furthermore, in Table 6, we investigate the trade-off between the improvement in *reachable input generation* and the additional time cost of *invariant inference and input generation sampling* in Halo. Invariant inference and input sampling account for 14.8% and 0.6% of the total fuzzing time, respectively, which is no more than 5 hours across all evaluation projects. This time cost demonstrates the efficiency of the adapted invariant inference used in Halo.

Table 7: The number of bugs detected in the newest version of the projects evaluated in the Magma benchmark. N_{bug} represents the number of bugs detected by Halo. **Category** represents whether the bug is a new one or an incomplete fix of a previous bug or CVE.

Project	Version	N_{bug}	N_{AFLGo}	N_{Beacon}	$N_{WindRanger}$	$N_{SelectFuzz}$	Category
libpng	1.6.39	1	1	1	1	1	New
libsndfile	1.2.0	2	0	1	0	1	New
		1	0	0	0	0	Incomplete fix
libtiff	4.5.0	2	0	0	1	0	Incomplete fix
poppler	23.4.0	3	0	1	0	1	Incomplete fix
lua	5.4.4	1	0	0	0	0	Incomplete fix

These results highlight the potential of self-optimized fuzzing with an evolutionary input generation. We also observe that the time costs of invariant inference and sampling do not increase significantly along with the size of evaluated projects since we only infer the invariants for input bytes, demonstrating the scalability of the adapted invariant inference approaches in Halo.

4.3. Incomplete Fix Detection Ability

To demonstrate the practicalness of Halo, We also evaluate the effectiveness of whether the optimized directed fuzzer, Halo, can find incomplete fixes in the newest version of real-world projects to help developers thoroughly fix the bugs. We collect the newest version of the programs in the Magma benchmark to compare the detection ability. For each program, we go through the issue list in its repository or Bugzilla to obtain the target points of the related vulnerabilities for directed fuzzers.

Table 7 shows the detected bugs across the compared fuzzers. Notably, all evaluated projects are widely used in the community and are continuously fuzzed through OSS-fuzz [7], a large-scale cloud-fuzzing environment. Despite the scrutiny, Halo has shown significant practicality by detecting ten previously unknown bugs involving seven incomplete fixes. This finding highlights the effectiveness of directed fuzzing with optimized input generation for identifying previously undiscovered bugs.

To ensure the transparency and reproducibility of our study, we reported all detected issues to the respective developers, and all issues were subsequently confirmed [42].

4.4. Ablation Study

Performance Variation of Reproducing Targets. After demonstrating the effectiveness of Halo in outperforming existing efforts, we proceed to investigate the reason for its efficient generation of reachable inputs by examining the specific design of our invariant inference engine, which are the distance-based input selection and similarity-based invariant selection. Specifically, to assess the importance of these two features, we create two variants of Halo, Halo-Similarity, and Halo-Distance, which disable the distance-based input selection and similarity-based invariant selection, respectively. Furthermore, for Halo-Similarity, we notice that it is impractical to record all inputs since the invariant inference engine cannot process them before exhausting the server's memory. We restrict the analysis to the first 2,000 fuzzer-generated inputs to facilitate the evaluation.

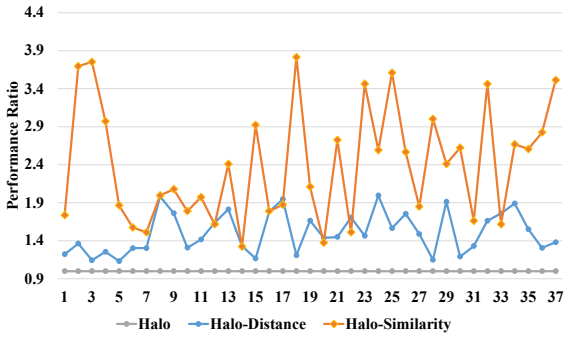


Figure 7: Reproduction time comparison of Halo, Halo-Similarity, and Halo-Distance. We utilize the performance of Halo as the baseline. The x-axis is the bug listed in Table 4. The y-axis is the ratio of the time costs in Halo-Similarity and Halo-Distance compared with Halo.

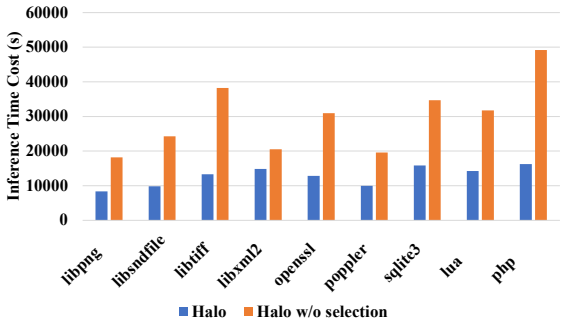
We first evaluate the high-level performance of the two variants by measuring the difference in reproduction time in the Magma benchmark, using the same settings as in the previous evaluation. Our results, presented in Figure 7, demonstrate that Halo outperforms both Halo-Similarity and Halo-Distance, yielding an average speedup of 2.40x and 1.51x in reproducing the targets, respectively. These results indicate that the combination of distance-based input and similarity-based invariant selection is necessary to achieve optimal performance in Halo.

Efficiency Enhancement. Subsequently, we analyzed the underlying causes of the performance gap. To underscore the significance of the efficiency enhancements introduced by the selection strategies we developed, we assessed the time required for invariant inference and input generation when these optimizations are turned off.

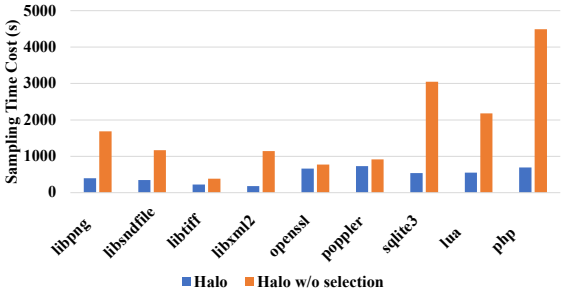
Overall, the average proportion of the time costs used for invariant inference and sampling without optimization is 36.3%, compared to 15.4% in Halo, as shown in Figure 8. Surprisingly, we observe that leveraging the original invariant inference approach suffers from severe scalability issues in most evaluated projects, accounting for up to half of the 24-hour evaluation time budget. Fortunately, armed with our designed selection strategies, Halo significantly improves the speed of invariant inference and sampling with 56.6% and 73.6% time reduction, respectively. Thus, the combination of strategies in Halo enables the efficient generation of reachable input to reproduce the targets.

While the efficiency improvement could benefit fuzzing, our designed strategies could also influence the precision of likely invariants inferred. Therefore, we then study the effectiveness influence caused by the strategies.

4.4.1. Influence of Distance-based Input Selection on Input Generation. To examine the influence of input selection on the effectiveness of Halo, we evaluated the ratio of unreachable input generation using invariants inferred with



(a)



(b)

Figure 8: Time costs of invariant inference and sampling in Halo, Halo-Similarity, and Halo-Distance. Figure 8a and Figure 8b are the comparison of the time costs for the invariant inference and sampling, respectively.

Table 8: The average ratio of the unreachable input generated using different input selection approaches in Halo. We use the performance of Halo-Simiarity as the baseline.

Project	Halo-Similarity	Number of Input Selected in Halo			
		385	500	1000	2000
libpng	82.5%	62.1%	61.3%	61.0%	60.8%
libsndfile	71.2%	51.0%	50.4%	50.1%	49.9%
libtiff	76.4%	57.8%	56.9%	56.6%	56.4%
libxml2	73.0%	53.6%	53.0%	52.6%	52.3%
openssl	81.8%	55.2%	54.6%	54.2%	54.0%
poppler	79.5%	47.5%	47.2%	47.1%	46.9%
sqlite3	83.6%	64.2%	63.1%	62.5%	61.7%
lua	75.3%	62.6%	61.7%	61.1%	60.8%
php	83.7%	58.9%	58.3%	57.9%	57.6%
Avg.	**78.6%**	**57.0%**	**56.3%**	**55.9%**	**55.6%**

different numbers of selected inputs. We used the variant, Halo-Similarity, which does not select inputs, as a baseline for comparison. All the variant fuzzers generate inputs from the inferred invariants using the same sampling approaches.

As shown in Table 8, the average ratio of reachable input generated in Halo is 57.0%. Compared to the invariant inferred without selecting inputs, Halo can generate 21.6% more reachable inputs on average, demonstrating the effectiveness of our distance-based input selection to enhance invariant inference.

Moreover, the average ratios of reachable input generated are 56.3%, 55.9%, and 55.6% using 500, 1000, and 2000 inputs, respectively. Overall, the ratio of reachable input generated does not improve significantly along with more inputs selected by the inference engine. This phenomenon indicates that it is unnecessary to utilize all inputs for approximating the path condition, as the explosive number of inputs can exacerbate the deficiency issues in invariant inference and ultimately hinder the performance of reproducing the targets, as shown in Figure 7 and Figure 8.

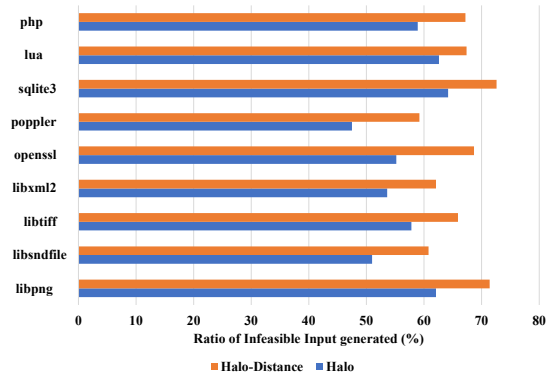


Figure 9: The average ratio of unreachable inputs generated influenced by the invariants selection strategy.

4.4.2. Influence of Similarity-based Invariant Selection on Input Generation. To examine the influence brought by the invariant selection, we evaluate the reachable input generation ratio in Halo and Halo-Distance, which employ all invariants provided by the inference engine to generate new inputs.

Figure 9 shows the results. Overall, despite the fluctuation of the performance of invariant inference without selection in different evaluated projects (ranging from 59.2% to 72.6%), Halo outperforms Halo-Distance in every project by generating 9.2% more reachable inputs. These results underscore the effectiveness of selecting likely invariants to maintain the precision of approximating target conditions. Furthermore, the fluctuation in the performance highlights the necessity of selecting invariants, as dynamic invariant inference cannot always provide precise invariants based on the given inputs. Fortunately, similarity-based invariant selection in Halo not only selects invariants that are more likely to generate reachable inputs, but also refines the invariant if all invariants are not sufficiently precise to generate reachable inputs effectively.

In summary, both distance-based input selection and similarity-based invariant selection strategies contribute to efficiently generating more reachable inputs in Halo. Additionally, we conclude the reasons that Halo-Distance could outperform Halo-Similarity, as shown in Figure 7. First, invariant inference accounts for a larger proportion of the time costs compared to sampling, as shown in Table 6, which makes the similar efficiency optimization more significant for directed fuzzing. Second, optimizing the input selection can help generate more reachable inputs than the invariant selection, according to Table 8 and Figure 9.

```
1 int foo() {
2   int tag1,tag2,others=input();
3   int* buff=input();
4   int size; // real buffer size
5   int type_size; // standard format size
6
7   // initial correctness checks
8   if(tag1==5...){
9     // complex buff parsing
10    size=buff_parsing(buff);
11
12    if(tag2==6){
13      for(int i=0;i<type_size;i++){
14        // crash if real size < type size
15        buff[i] ....
16      }
17    }
18  }
19 }
```

Figure 10: Case study for one incomplete fix in Libtiff

4.5. Case Study

To demonstrate the impact of Halo and its general applicability, we study an incomplete fix discovered in `libtiff`. This bug relates to parsing the format and types with the values of tags. We demonstrate the simplified version of the code and discuss its impact in Figure 10;

This bug is caused by trusting user-supplied size data in format standards. Specifically, the program parses the input file with preliminary type and correctness checking (Lines 8-12). However, the actual size of the input buffer can be different from the predefined values according to the type header, which can be manipulated by an adversary. In that case, the program accesses invalid memory since it assumes the given input is well-formatted (Line 15).

However, such cases are challenging for existing directed fuzzers to detect. Scheduling-based approaches cannot help fuzzers generate diverse inputs to satisfy the condition to reach the target, not to mention satisfying the implicit triggering condition to trigger the bug. For culling-based approaches, as this parsing function is in the early stage of the program, it cannot cull sufficiently many paths. The scalability issues hinder the effectiveness of symbolic execution since it cannot efficiently handle the loop (Line 13) and avoid state-explosion at the parsing function (Line 10). Fortunately, Halo individually infers invariants for conditions at Lines 8 and 12 to help the fuzzer generate inputs that always satisfy $tag1 == 5 \wedge tag2 == 6$ to increase the chance of triggering this bug. Moreover, detecting such issues also helps the developer to increase the robustness of their format design.

5. Discussion

After demonstrating the effectiveness of Halo through the evaluation, we would like to discuss the potential of techniques used in Halo beyond directed fuzzing and its limitations that can be optimized in the future.

5.1. Potentials

Assisting Dynamic Analysis with Fuzzing. Dynamic program analysis, which analyzes the target during execution, has been widely adopted in various application scenarios, e.g., taint analysis [43], [44], program synthesis [45], [46], and specification generation [47], [48]. Compared with conventional static analysis, dynamic analysis can provide precise results with fewer false positives based on the execution feedback. However, it is challenging for dynamic analysis to thoroughly analyze the programs without providing a sufficient number of inputs [49]. Halo leverages state-of-the-art fuzzing techniques to generate sufficient inputs for dynamic analysis to approximate program semantics. Moreover, our input selection strategy can effectively restrain the number of inputs to achieve better scalability. Apart from path conditions approximated by the likely invariant in Halo to handle memory safety issues, complex semantics such as state machine model and partial order constraint can also be considered to support detecting vulnerabilities involving logical correctness.

Semantic Extraction from Inputs. While one of the main trends in fuzzing is to extract target program semantics at different granularities, such as using symbolic execution to solve the path condition, the invariants inferred by the executed inputs in Halo indicate another potential direction for fuzzing: Utilizing the observed unreachable executions to prevent fuzzers from continuously exploring irrelevant program behaviors. In Halo, we also infer the invariant based on the unreachable inputs to constrain the search space of the subsequent input generation for efficiently approaching the target vulnerabilities. Since fuzzers can generate a tremendous number of inputs to examine the program, it is possible to extract more semantics from the executed inputs apart from the coverage to guide fuzzers in exploring the programs.

5.2. Limitations

Scalability of the Invariant Inference. While dynamic likely-invariant inference outperforms symbolic execution, its effectiveness can still suffer from scalability issues, for example, due to the excessive size of given inputs or unlimited candidate invariants.

To mitigate these issues, Halo provides an empirical optimization to select the proper samples for inference and precise invariant to generate inputs. Nevertheless, we have not optimized the quality of candidate invariants for improving the precision of approximating the target condition, Therefore, the performance of Halo may fluctuate in other projects that have not yet been evaluated.

Nevertheless, we consider this problem to be orthogonal to improving directed fuzzing, which remains an open question for improving invariant inference (see Section 6.2 for a discussion of the existing literature). Halo may also be further improved through optimizations from this related but orthogonal research.

Solving Complex Path Conditions. The strength of Halo comes from constraining the search space using invariants inferred from both reachable and unreachable inputs. Even though we have demonstrated the effectiveness of invariants inferred exclusively from the unreachable inputs in Table 4, it may be less effective against using inputs from both categories, e.g., the improvement ratio of the reaching time is smaller than the triggering time. To mitigate this issue, it is possible to collaborate Halo with other input generation techniques, such as hybrid fuzzing with concolic execution, to reach the targets faster and collect inputs for both categories. Meanwhile, the implementation of Halo based on AFL++ also makes integrating with AFL-based fuzzing frameworks feasible.

Support to Rich Semantics. Apart from complex path conditions, vulnerabilities could involve various rich semantics. For example, logic errors in network servers can require programs to execute functions satisfying specific protocols. However, since our main focus is the path condition in this paper, Halo may not reproduce such kinds of target vulnerability efficiently. Still, we would like to summarize such issues as a different question to the one solved by Halo, as discussed in Section 5.1, which may be addressed in future work.

6. Related Work

Apart from the existing literature in directed greybox fuzzing (Section 2.1), we survey other related techniques that may help improve directed fuzzing to generate input effectively (Section 6.1. Moreover, we also study the potential optimization in invariant inference to make it more practical for fuzzing (Section 6.2).

6.1. Sophisticated Input Generation

Mutating Relevant Bytes. One major trend of optimizing mutation is to mutate the related input offsets to satisfy the uncovered branch conditions. Other than random mutation, Fairfuzz [50] identifies the input offsets where the values are not necessary to change, Thus, minimizing the input search space improves the efficiency of mutation. Angora [51] 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 [32] proposes to use the intermediate values as the feedback to modify the values in the inputs. PATA [30] improves the precision of the taint analysis by distinguishing different paths and contexts. Parmesan [23] utilizes the labels obtained from the santizers [39] to minimize the number of branches needed to be analyzed by the taint analysis.

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 cover that can help reach the target faster. Some branches may not even satisfy the path conditions of the targets. Therefore, it is challenging to explicitly adapt these approaches for directed fuzzing, which is also why Halo leverages invariant inference only for the input bytes rather than analyzing the whole program.

Leveraging the Constraint Solver. 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 [52] solves part of the path constraint for a basis seed and leverages mutation for validated inputs satisfying the actual condition. Intriguer [53] further replaces symbolic emulation with dynamic taint analysis, which decreases the overhead of modeling a large amount of mov-like instructions. Pangolin [54] proposes to preserve the constraint as an abstraction and reuse it to guide further input generation. To mitigate the scalability issues of the symbolic execution engine for better practicalness, DigFuzz [55] prioritizes solving the condition for branches that could be difficult to be covered by the fuzzers. It estimates the difficulties by the inverse proportion of the path execution frequency. Savior [56] utilizes the number of labels marked by the sanitizers to prioritize the paths that SymCC [24] improves the scalability of concolic execution by instrumenting the whole functionality of symbolic execution into the compiled binary and, thus, reducing the overhead of dynamic interference from the conventional concolic engine.

However, these approaches still suffer from the inherent scalability issues from the symbolic execution, as we demonstrated in the evaluation, which is challenging to handle complex program behaviors, such as loop or nested data structures. Instead, Halo approximates the conditions solely based on values of input bytes and their execution feedback, which mitigates the analysis overhead for the whole program and achieves better efficiency of the vulnerability reproduction.

6.2. Efficient Invariant Inference

Invariant inference has emerged as a promising technique in program verification [57], [58], [59], [60], software testing [38], [61], [62], [63], and property checking [25], [26], [28], [29], [37], [64], [65], [66]. It is effective for ensuring the correctness of software systems, as invariants can serve as formal specifications of the expected behavior of a program.

The primary objective of invariant inference is to efficiently infer the precise invariants that hold for all program behaviors. Conventional verification-based approaches aim to detect the invariant precisely. The majority of the existing efforts [29], [57], [58], [59], [60], [63] uses abstract interpretation and symbolic execution to statically approximate the fix points of the program behaviors as the properties. Despite their theoretical accuracy, their efficiency is restricted by the innate scalability issues in symbolic analysis. Therefore, their approaches are typically designed for specific application scenarios with limited assumptions.

Dynamic invariant inference approaches do not pursue the accurate assumption for the whole program and in-

fer the invariant based on the given inputs. Starting with Daikon [25], [64], the fundamental dynamic invariant inference engine, existing literature monitors the execution traces to validate a set of candidate invariants and concretize the coefficient and terms. DIDUCE [66] relaxes the hypothesis of the invariant as violations may lead to anomaly program behaviors and prioritizes the violations with confidence for fault localization.

As the quality of the invariant inferred highly relies on the given inputs and template invariants, existing efforts utilize more program semantics extracted from the execution to refine the accuracy of invariants. For example, DySy [37] leverages concolic execution to reasoning the potential template invariants. iDiscovery [28] utilizes the intermediate symbolic states to conduct an incremental invariant inference to refine the accuracy of invariants. MRI [38] proposes to model the inference process as a searching problem and uses mining-based approaches to obtain precise invariants. Dig [26] relies on the execution feedback from the counterexample to prune the infeasible range of the invariants and gradually refine the precision.

The invariant inference engine used in Halo leverages the insights of the above-demonstrated techniques to achieve better efficiency. So far, none of the existing efforts focuses on optimizing the quality of given inputs [28], which is one of the crucial factors influencing the effectiveness of invariant inference. Instead, Halo leverages the fuzzer as an input generator to improve the precision of the invariant and, eventually, enhance the input generation for fuzzing itself.

7. Conclusion

We have presented Halo, a self-optimized directed fuzzing guided by the dynamic likely invariant inferred from the generated inputs, to significantly improve the effectiveness of generating reachable inputs triggering the target vulnerabilities. The likely invariants help Halo constrain the search space of subsequent input generation. The empirical results demonstrate the dramatic improvement brought by our design methods: Halo outperforms existing directed fuzzers with a 15.3x speedup in reproducing the target vulnerabilities by generating 6.2 times more reachable inputs. Moreover, Halo also detects ten previously undiscovered bugs involving seven incomplete fixes of the previous bugs and CVEs from the newest patched version of the evaluated projects in the Magma benchmark.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their valuable feedback. The authors are supported, in part, by the Hong Kong Research Grant Council under Grant No. RGC16206517, Hong Kong Innovation and Technology Commission under Grant No. ITS/440/18FP, SNSF PCEGP2_186974, ERC H2020 grant 850868, and donations from Huawei.

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] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 235–245. [Online]. Available: https://doi.org/10.1145/2491411.2491438

[9] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[10] 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

[11] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 49–64. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534772

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

[13] 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

[14] 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

[15] 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.

[16] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 2693–2707. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00060

[17] 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.

[18] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2255–2269. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zong

[19] 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*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 388–399. [Online]. Available: https://doi.org/10.1145/3564625.3564643

[20] 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

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

[22] 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.

[23] 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

[24] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[25] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007, special issue on Experimental Software and Toolkits. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016764230700161X

[26] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Dig: A dynamic invariant generator for polynomial and array invariants," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, sep 2014. [Online]. Available: https://doi.org/10.1145/2556782

[27] ——, "Using dynamic analysis to discover polynomial and array invariants," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 683–693.

[28] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 362–372. [Online]. Available: https://doi.org/10.1145/2610384.2610389

[29] J. W. Nimmer and M. D. Ernst, "Invariant inference for static checking: An empirical evaluation," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 11–20, 2002.

[30] 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.

[31] "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

[32] 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/

[33] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer, "Igor: Crash deduplication through root-cause clustering," 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. 3318–3336. [Online]. Available: https://doi.org/10.1145/3460120.3485364

[34] Z. Jiang, S. Gan, A. Herrera, F. Toffalini, L. Romerio, C. Tang, M. Egele, C. Zhang, and M. Payer, "Evocatio: Conjuring bug capabilities from a single poc," 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. 1599–1613. [Online]. Available: https://doi.org/10.1145/3548606.3560575

[35] A. S. Singh and M. B. Masuku, "Sampling techniques & determination of sample size in applied statistics research: An overview," *International Journal of economics, commerce and management*, vol. 2, no. 11, pp. 1–22, 2014.

[36] Y. Chen, R. Dwivedi, M. J. Wainwright, and B. Yu, "Vaidya walk: A sampling algorithm based on the volumetric barrier," in *2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2017, pp. 1220–1227.

[37] C. Csallner, N. Tillmann, and Y. Smaragdakis, "Dysy: Dynamic symbolic execution for invariant inference," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 281–290.

[38] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 701–712. [Online]. Available: https://doi.org/10.1145/2642937.2642994

[39] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *USENIX ATC 2012*, 2012. [Online]. Available: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[40] 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

[41] 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

[42] M. P. Heqing Huang, Anshunkang Zhou and C. Zhang, "Halo bug reports," https://shorturl.at/qDQSZ.

[43] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[44] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," 1 2005. [Online]. Available: https://kilthub.cmu.edu/articles/Dynamic_Taint_Analysis_for_Automatic_Detection_Analysis_and_Signature_Generation_of_Exploits_on_Commodity_Software/6468716

[45] T. Dillig, I. Dillig, and S. Chaudhuri, "Optimal guard synthesis for memory safety," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 491–507.

[46] B. Li, I. Dillig, T. Dillig, K. McMillan, and M. Sagiv, "Synthesis of circular compositional program proofs via abduction," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 370–384.

[47] H. Zhu, T. Dillig, and I. Dillig, "Automated inference of library specifications for source-sink property verification," in *Asian Symposium on Programming Languages and Systems*. Springer, 2013, pp. 290–306.

[48] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *ACM SIGPLAN Notices*, vol. 44, no. 1. ACM, 2009, pp. 289–300.

[49] T. Ball, "The concept of dynamic analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, 1999.

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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

[55] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[56] 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

[57] Y. M. Y. Feldman, N. Immerman, M. Sagiv, and S. Shoham, "Complexity and information in invariant inference," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, dec 2019. [Online]. Available: https://doi.org/10.1145/3371073

[58] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: A verifier for gpu kernels," *SIGPLAN Not.*, vol. 47, no. 10, p. 113–132, oct 2012. [Online]. Available: https://doi.org/10.1145/2398857.2384625

[59] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proceedings of the 33rd annual ACM/IEEE symposium on logic in computer science*, 2018, pp. 13–25.

[60] B.-Y. E. Chang and K. R. M. Leino, "Abstract interpretation with alien expressions and heap structures," in *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005. Proceedings 6*. Springer, 2005, pp. 147–163.

[61] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[62] D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[63] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 144–154.

[64] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 213–224. [Online]. Available: https://doi.org/10.1145/302405.302467

[65] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *USENIX Security Symposium*, vol. 58, 2010.

[66] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th international conference on Software engineering*, 2002, pp. 291–301.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

Directed fuzzing is a technique designed to more efficiently generate fuzzing inputs that lead to specific target areas. The existing directed fuzzing solutions create fitness functions that prioritize inputs close to the target areas, or cull infeasible executions as early as possible. However, these solutions still generate many inputs that fail to reach the targets.

This paper presents a new approach to directed fuzzing. Its core idea is to approximate path conditions for inputs that can reach the target of analysis by inferring from the inputs that have or have not reached the target. The authors implemented this idea in a prototype called Halo. Compared to other recent directed fuzzers against the Magma fuzzing benchmark, Halo finds more reachable inputs and finds new, previously undisclosed vulnerabilities in a number of newer versions of the programs used by Magma.

A.2. Scientific Contributions

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

A.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. The paper targets the problem of directed fuzzing, which has been studied in several papers already. The idea of inferring likely invariants on past inputs and then using the invariants to guide fuzzing towards targets is interesting. Evaluation also shows that it can make inputs generation more efficient in producing feasible inputs that can reach targets.
2) The paper addresses a long-known issue. It is a long-known issue that existing directed fuzzing tools still generate many inputs that do not reach the targets of analysis. The proposed approach is able to substantially generate more inputs that can reach the targets.
3) The paper creates a new tool to enable future science. The authors implemented a new directed fuzzing tool called Halo, which has been empirically demonstrated to be more effective than the existing tools.

A.4. Noteworthy Concerns

1) Some reviewers felt certain technical details were not discussed clearly and thoroughly.
2) Some reviewers also thought the evaluation could be further strengthened by evaluating how the accuracy of such identification affects the invariants inference and thus fuzzing, discussing the impact of the vulnerabilities found, and perhaps demonstrating larger applicability by expanding the corpus of programs under test.

Appendix B.
Response to the Meta-Review

The major concern relates to details of invariant inference. Our paper does not claim any contribution to optimizing existing invariant inference techniques but merely reuses and repurposes existing techniques. Meanwhile, it is an open question for likely invariant inference to consider complex conditions beyond linear form with type information and increase precision. Incorporating and adjusting invariant inference for this use case remains a question for future work. Instead, our main contribution relates to leveraging randomly mutated inputs to infer both over- and under-approximating constraints. The above clarification also addresses, in part, the second concern related to the influence of invariant inference with different precision. In future work, we will explore further optimization opportunities based on this direction.