BEACON : Directed Grey-Box Fuzzing with Provable Path Pruning

Heqing Huang†, Yiyuan Guo†, Qingkai Shi†✉, Peisen Yao†, Rongxin Wu‡, Charles Zhang†
†The Hong Kong University of Science and Technology, China
‡Xiamen University, China
†{hhuangaz, yguoaz, qshiaa, pyao, charlesz}@cse.ust.hk, ‡wurongxin@xmu.edu.cn

***Abstract*—Unlike coverage-based fuzzing that gives equal attention to every part of a code, directed fuzzing aims to direct a fuzzer to a specific target in the code, e.g., the code with potential vulnerabilities. Despite much progress, we observe that existing directed fuzzers are still not efficient as they often symbolically or concretely execute a lot of program paths that cannot reach the target code. They thus waste a lot of computational resources. This paper presents BEACON, which can effectively direct a grey-box fuzzer in the sea of paths in a provable manner. That is, assisted by a lightweight static analysis that computes abstracted preconditions for reaching the target, we can prune 82.94% of the executing paths at runtime with negligible analysis overhead (<5h) but with the guarantee that the pruned paths must be spurious with respect to the target. We have implemented our approach, BEACON, and compared it to five state-of-the-art (directed) fuzzers in the application scenario of vulnerability reproduction. The evaluation results demonstrate that BEACON is 11.50x faster on average than existing directed grey-box fuzzers and it can also improve the speed of the conventional coverage-guided fuzzers, AFL, AFL++, and Mopt, to reproduce specific bugs with 6.31x ,11.86x, and 10.92x speedup, respectively. More interestingly, when used to test the vulnerability patches, BEACON found 14 incomplete fixes of existing CVE-identified vulnerabilities and 8 new bugs while 10 of them are exploitable with new CVE ids assigned.**

***Index Terms*—Directed fuzzing, precondition inference, program transformation**

I. INTRODUCTION

Different from the conventional coverage-based fuzzing that pays equal attention to every part of the code, directed fuzzing aims to thoroughly test a specific part of the program [1]–[3]. It is widely adopted in many application scenarios such as testing vulnerability patches [4], [5], generating proof-of-concept of potential bugs [6], reproducing crashes [7], [8], and tracking information flow [9].

The key to achieving practicality in directed fuzzing is to reject the unreachable execution paths as early as possible. However, despite the great improvements made by existing works, namely the directed white-box fuzzing and the directed grey-box fuzzing, they still often execute a large number of paths that cannot reach the target code, which we refer to as the *infeasible-path-explosion* problem. Specifically, the directed white-box fuzzers [5], [10], [11] rely on symbolic execution to decide upon reachability by solving path constraints and aims to provide a theoretical guarantee for generating inputs that can reach the target. Therefore, their innate use of symbolic execution fundamentally limits their ability to scale. Moreover, the cost of path exploration is further exacerbated by both paths with unsatisfiable path conditions or the ones that cannot even reach the target points. For instance, directed white-box fuzzing is shown to be incapable of reproducing a vulnerability within 24 hours [1]. On the other hand, the directed grey-box fuzzers are in general not concerned with rejecting unreachable paths at all. They rely on prioritizing seeds according to their likelihood of reaching the target code using heuristics collected from the execution feedback [1]–[3]. They either employ lightweight meta-heuristics [1], [2], e.g., the distance towards the target, or machine learning techniques [3] to predict the reachability, with no guarantees of such prioritization leading to any rejection of infeasible paths. As a result, more than 95% of the inputs cannot reach the given program point in the 24-hour experiment mentioned in AFLGo [1], demonstrated in Figure 1.

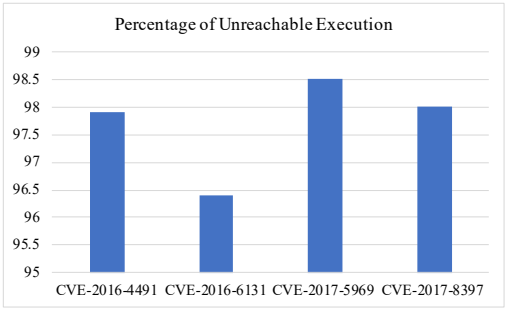

Figure 1: The proportion of executed unreachable paths in terms of different CVEs with AFLGo in 24-hour experiments.

This paper presents BEACON, a directed grey-box fuzzer that directly addresses this pruning of infeasible paths[1] with negligible overhead, thus dramatically increases the reproduction efficiency by 11.50 times on average comparing to the related work. Our key insight is that, through a cheap static analysis, we can calculate a sound approximation for the values of program variables that directly make the path-to-target infeasible. Armed with this approximation, our fuzzer can reject over 80% of the paths executed during fuzzing. More specifically, we not only directly prune a path when it hits an instruction that cannot reach the target on the control flow graph, but also the paths that are reachable to the target but have an unsatisfied path condition. For example, to reach the target code at Line 19 in Figure 2, the program states must satisfy the condition $w > 10$ in Line 18. Notice that the calculation in Lines 10 and 12, w must be initialized

[1]In this paper, we say a path is infeasible if it cannot reach the target code at runtime. We say a program state is infeasible if a path with the runtime state is infeasible.

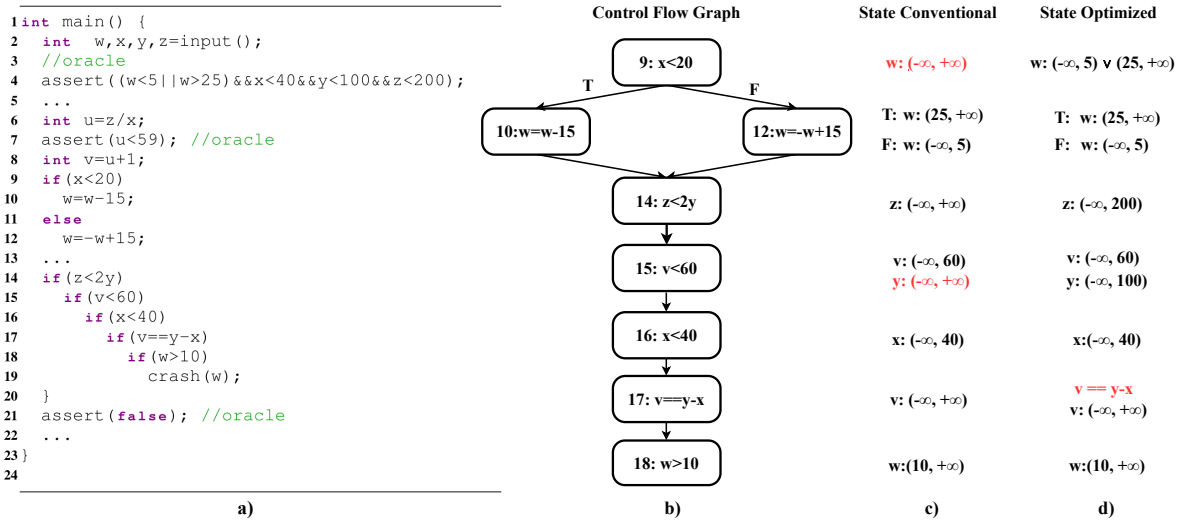

Figure 2: A Motivating Example. a) The code snippet for illustration. b) The control flow graph of the given program. c) Conventional backward interval analysis. d) Our backward interval analysis.

with $w < 5 \vee w > 25$. Otherwise, it is not necessary for us to explore any path after Line 2. Given the fact that such infeasible paths widely exist in practice, effectively pruning these paths allows a significant improvement in terms of testing efficiency.

To determine the infeasible program states efficiently, we employ a dedicated static analysis as a preprocessing procedure that analyzes the program and computes the necessary conditions, e.g., $w < 5 \vee w > 25$ in the last example, of program variables for reaching the target code. To be both precise and efficient, we arm the static analysis with two novel optimizations, termed respectively as relationship preservation and bounded disjunction. The former preserves the relationship among program variables and thus maintains precision. The latter bounds the number of disjunctions to avoid expensive logical reasoning and precision loss incurred by the exhaustive path merging. Such tailored static analysis is fast and simultaneously ensures precision and scalability.

To evaluate its effectiveness, we implement our approach, BEACON, as a direct grey-box fuzzer. We compare it to the state-of-the-art (directed) fuzzers, including AFL [12], Mopt [13], AFLGo, and Hawkeye [2], in terms of the capability of reproducing existing CVE-identified vulnerabilities and testing whether they are correctly fixed in the newest versions of the software. The results demonstrate that BEACON can early reject on an average of 82.94% paths during fuzz testing and, thus, exhibits 11.50x speedup compared to conventional directed fuzzers. Furthermore, BEACON can also greatly help non-directed fuzzer to reproduce a specific bug. Our experiments show that, BEACON can accelerate 6.31x, 11.86x, and 10.92x for fuzzers such as AFL, AFL++, and Mopt to reproduce the target bugs, respectively. More interestingly, when testing the vulnerability patches, BEACON found 14 incomplete fixes and 8 new bugs. All these incomplete fixes and new bugs have been confirmed by the software developers, and 10 of them are assigned with CVE ids. The prototype of BEACON is contributed in the Docker Hub: https://hub.docker.com/r/yguoaz/beacon. In summary, we make the following contributions:

- We design a fast and precise static analysis for computing necessary conditions for reaching the given testing target, enabling us to filter the infeasible program states.
- We implement a directed grey-box fuzzer that can prune a large number of infeasible paths with negligible runtime overhead.
- We provide empirical evidence that our approach is more efficient and effective than the state-of-the-art (directed) fuzzers and has the potential to improve the performance of non-directed fuzzers.

II. BACKGROUND

This section surveys recent directed grey-box fuzzers (Section II-A) and summarizes the challenges we try to resolve in this paper (Section II-B).

A. Directed Grey-Box Fuzzing

Directed grey-box fuzzing aims to thoroughly test a target part of a program with little runtime overhead. Recent work in this line mainly focuses on addressing two problems — one is how to specify which target to test, and the other is how to drive the fuzzer to reach the target code quickly.

Specifying the Targets. In many applications like testing a patch, we can manually specify the target code, i.e., where the patch is made. Meanwhile, recent work also attempts to automatically specify the testing target: Semfuzz [14] leverages natural language processing to analyze bug reports and retrieves the potential buggy points as its targets. ParmeSan [15] labels all the potential buggy points indicated by various sanitizers [16].

Reaching the Targets. Another recent mainstream directed grey-box fuzzers prioritize testing inputs so that we can run them "closer" to the target with higher priority. To this end, multiple distance metrics have been proposed: AFLGo [1] is one of the first to define the concept of directed fuzzing. It defines the distance of a testing input towards a target basic block as the average of the distances between a block B and the target, where B ranges over the blocks that an execution against the input goes through. Hawkeye [2] optimizes the distance metric with the intuition that a vulnerability is triggered by a sequence of operations rather than a single program point. Therefore, Hawkeye also takes the call trace similarity into account. FuzzGuard [3] leverages an observation that reproducing a bug needs to satisfy its path condition. Therefore, it trains a classifier as a predictor to prioritize the testing inputs so those with a higher probability of satisfying the path condition can be executed at a higher priority. Savior [17] integrates fuzzing with symbolic execution. During dynamic testing, it drives symbolic execution to solve the path constraint with higher priority if this path visits more branches that can reach more targets with potentially buggy code.

B. Problem and Challenges

As discussed before, existing directed fuzzers all suffer from the infeasible-path-explosion problem as they execute lots of infeasible paths that cannot reach a given target program point. To solve this problem, our basic idea is to employ a lightweight static analyzer to compute sound intermediate program states (in the form of first-order logic conditions) as the precondition for executions to reach the target. Any execution violating these intermediate conditions should be immediately terminated.

In our approach, the intermediate program states are computed as an approximation of the weakest precondition (also known as the necessary precondition), which has been widely studied in static analysis [18]–[21]. Formally, given a target program point l and another program location p at the control flow graph, *the weakest precondition* $wp(p, l)$ categorizes the least restricted precondition at p that can guarantee the reachability of l [22]. $wp(p, l)$ is usually represented as a first order logic formula over the program variables defined before p, and any path reaching p that does not satisfy $wp(p, l)$ could be safely pruned during fuzzing. In our demonstration, we use l_i to represent the program location after Line i of the source code. For example, in Figure 2, $wp(l_{13}, l_{18}) \equiv z < 2y \wedge v < 60 \wedge x < 40 \wedge v = y - x \wedge w > 10$, meaning that any execution reaching Line 14 not satisfying this condition (e.g, an execution with $x = 500$) could be safely pruned.

As with all static analyses, statically inferring the necessary precondition is challenging as it is difficult to be precise and efficient at the same time. Existing studies often make two tradeoffs to compromise precision for speed. First, fast static analyses often do not precisely reason about the path condition: they either ignore path conditions sheerly by focusing on checking a particular property [18], [19] or perform limited reasoning on simple path conditions, e.g., discarding the relationship among variables in branch conditions [23] or neglecting condition satisfiability [24] to achieve high speed. To illustrate, let us consider the code in Figure 2 and its control flow graph. We perform a backward analysis from Line 18 to infer the precondition $wp(l_i, l_{18})$ using explicit pattern matching. When reaching Lines 18, 16, and 15, we obtain the path conditions, $w > 10$, $x < 40$ and $v < 60$, which determine the preconditions for variables w, x and v before Line 14 as $w \in (10, +\infty), x \in (-\infty, 40), v \in (-\infty, 60)$. Such deduction is easy because of the simplicity of the constraints in those lines. However, the path conditions can get rather complex, such as $v = y - x$ at Line 17 and $z < 2y$ at Line 14, which normally involves an expensive constraint solver [25]–[27].

Second, there are also many techniques that attempt to precisely reason about path conditions [21], [28]–[30]. However, to avoid path explosion (or the explosive number of disjunctions), they either consider a limited number of paths and produce unsound results [21] or merge the conditions computed from different branches at merge points on the control flow graph, which may notably lose precision [28]–[30]. On the one hand, since unsound results may let us incorrectly prune a path that can reach the target, we cannot use these approaches in our scenario. On the other hand, merging conditions from different branches may lead to precision loss [31]. For example, in Figure 2, the precise precondition on variable w before Lines 10 and 12 is $w \in (25, +\infty)$ and $w \in (-\infty, 5)$, respectively. To be efficient, the analysis then merges the conditions from the two branches, which results in an imprecise condition $w \in (-\infty, +\infty)$ before Line 9, where the false positives in the interval, $w \in [5, 25]$, are introduced. Thus any execution with w residing in this range cannot be pruned by the preconditions.

To make the static inference of a necessary precondition practical for directed fuzzing, our analysis attempts to address the following two problems in the aforementioned approaches:

1) How to efficiently reason about the path conditions in precondition inference?
2) How to design a sound analysis while avoiding severe precision loss brought by merging paths?

III. Beacon in a Nutshell

As illustrated in Figure 3, Beacon takes the program source code and the fuzzing targets as the inputs. To simplify the static analysis and prune apparent infeasible paths, we firstly perform a reachability analysis on the inter-procedural control flow graph and slice away paths that apparently cannot reach any target. For example, we place an `assert(false)` at Line 21 in Figure 2 to prune paths reaching this Line because the target at Line 19 is not reachable from Line 21 in the control flow graph. To prune more infeasible paths (such as those with unsatisfied path conditions), we employ a dedicated static analyzer, as illustrated below.

Backward Interval Analysis. After slicing the program with statically computed control flow information, we start our backward analysis to infer the weakest precondition such as $wp(l_{13}, l_{18}) \equiv z < 2y \wedge v < 60 \wedge x < 40 \wedge v = y - x \wedge w > 10$

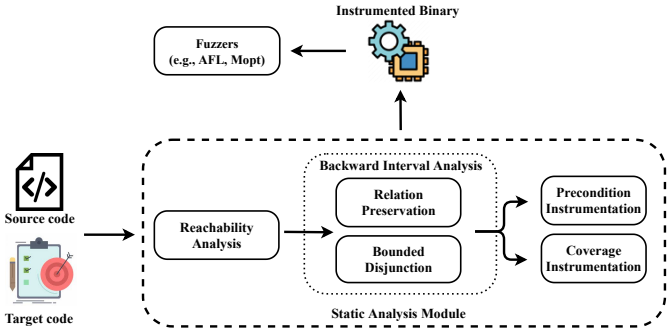

Figure 3: Workflow of BEACON.

in Figure 2. However, such an exact weakest precondition is not computable in general. Thus, we often compute a sound abstraction (or over-approximation), $\hat{wp}(l_{13}, l_{18})$ as the necessary precondition, with respect to a predefined abstract domain. Commonly used numerical abstract domains include the interval domain [32], octagon domain [33], polyhedral domain [34], etc, and can aid in reasoning possible values of variables occurring in the program. For instance, when the interval domain is used, we have $\hat{wp}(l_{13}, l_{18}) \equiv w \in (10, +\infty) \wedge x \in (-\infty, 40) \wedge v \in (-\infty, 60) \wedge y \in (-\infty, 100) \wedge z \in (-\infty, 200)$.

In this work, we choose the interval domain because it is almost the cheapest domain that can be computed via a featherweight static analysis [35]. Such an efficient static analysis is practical as it does not introduce much overhead for fuzz testing. Meanwhile, notice that preconditions expressed using an interval domain can be efficiently checked at runtime, requiring only comparisons between variables and constants (i.e., bounds of the intervals). Therefore, it does not interfere with the original execution with much runtime overhead.

Despite these merits of the interval domain for efficiently reasoning about the preconditions, it is known that the interval abstraction is coarse and can produce imprecise results because it does not respect inter-variable relations [36]. We also have shown in the previous section that the merging of backward paths can also lead to imprecise results. In this work, we argue these are the two problems significantly exacerbate the perceived imprecision of which we need to address.

(1) Relationship Preservation. Preserving the relationship among variables leads to more precise preconditions, thus pruning more paths during the directed fuzzing. Formally, this problem can be described as: given a path condition ϕ, how to compute the interval of a variable v. For example, in Figure 2, we perform a backward analysis starting from the location after Line 18 with the interval domain to obtain the necessary precondition $\hat{wp}(l_{13}, l_{18})$. Similar to the conventional approach, we obtain the interval of $w \in (10, +\infty)$ from Line 18. Moreover, even though we cannot deduce the precondition of v, y, x when analyzing the path condition $v == y - x$ at Line 17, we still track the fact that $v == y - x$ holds. Therefore, after the intervals for x and v are inferred at Lines 16 and 15 respectively, we can utlize the tracked relation $v == y - x$ to infer the interval for y at Line 15, as the sum of the intervals of v and x: $y \in (-\infty, 60) + (-\infty, 40) = (-\infty, 100)$. This updated interval for y also enables us to further infer the interval of z at Line 14.

(2) Bounded Disjunction. As discussed in the last section, the conventional approaches merge static analysis results from different paths, e.g., $w \in (-\infty, 5)$ and $w \in (25, +\infty)$ from Line 9 to Line 12 in Figure 2, which results in an imprecise result, e.g., $w \in (-\infty, +\infty)$. However, separately keeping the results of each path by disjuncting the results, e.g., $w \in (-\infty, 5) \vee w \in (25, +\infty)$, can lead to an explosive condition size and make the analysis slow. To control the loss of precision and, meanwhile, be fast, we maintain a bounded disjunction for the analysis results and only merge results when the number of paths exceeds a threshold, τ. As an example, the conventional approach has $\tau = 1$ as we always merge the results.

The key question here is that, when the number of disjunctive conditions exceeds the threshold τ, which ones should be merged so that the merged precondition is still precise and can prune more infeasible paths. For instance, given the threshold $\tau = 2$ and the disjunction $w \in (-\infty, 5) \vee w \in (25, 40) \vee w \in (55, +\infty)$, merging $w \in (25, 40) \vee w \in (55, +\infty)$ is better than $w \in (-\infty, 5) \vee w \in (25, 40)$ because the former adds fewer false positives, i.e., $[40, 55]$, than the latter, i.e., $[5, 25]$. While the example looks simple, this question becomes very challenging for a complex disjunctive condition, which we detail later.

Selective Instrumentation. It is expensive and unnecessary to instrument all statements in the code under test. In BEACON, we instrument two kinds of statements: variable-defining statements and branch statements. For a statement that defines a variable x, we insert a new statement, `assert(c(x))`, after this one where $c(x)$ represents the inferred precondition in disjunctive form over the variable x. For example, we insert the statement, `assert(w<5||w>25)`, at Line 4 in Figure 2 so that every execution reaching this point with the condition $5 \leqslant w \leqslant 25$ is aborted immediately after the variable w is defined. For a branch statement, we insert a statement, `assert(false)`, after if the branch condition does not allow an execution to reach the target. For example, we insert the statement, `assert(false)`, at Line 21 because, as discussed above, this block cannot reach the target point through control flow.

IV. METHODOLOGY

BEACON prunes the infeasible paths from two aspects, control flow reachability and path condition satisfiability. Our first step is to prune basic blocks that cannot reach the target code by applying a graph reachability analysis on the interprocedural control flow graph (ICFG) of the program. This step is straightforward except that we employ an efficient and sound pointer analysis to resolve function pointers [37]. In this section, we focus on pruning basic blocks that can reach the target code by using a dedicated static analysis. To be clear, we first define a simple language on which we demonstrate our algorithm and the background on precondition inference in IV-A. Then, we illustrate the details of how BEACON

Program P	::=	*fun+*
Function fun	::=	*fun* : $(v_1, ..., v_n) \rightarrow r$ $\{s\}$
Statement s	::=	$s_1; s_2 \mid l : s_1 \star s_2 : l' \mid l : i : l'$
Instr i	::=	$v := e \mid v_1 := *v_2 \mid *v = e$
	|	goto l | assume b
	|	$v :=$ call fun$(a_1, ..., a_n)$
Expression e	::=	$v \mid c \mid e_1$ op e_2, op $\in \{+, -, \times, \div\}$
Boolean b	::=	$true \mid false$
	|	e_1 cmp e_2, cmp $\in \{<, >, =\}$
	|	$b_1 \wedge b_2 \mid \neg b$
Location		contains all program location l

Figure 4: A simple programing language.

infers the precondition of reaching the target in IV-B, and optimizations to maintain analysis precision in IV-C. Finally, we present how we leverage the preconditions inferred for fuzzing through the instrumentation process IV-D.

A. Preliminary

Language. Figure 4 shows a simple language we use to demonstrate our methodology. In the language, a program consists of one or more functions, each taking a vector of formal parameters $(v_1, ..., v_n)$ and returns a value r. The statement inside a function is a sequence $s_1 : s_2$, a non-deterministic branch $s_1 \star s_2$, or an atomic instruction i. We label the program location before and after a statement as l and l' respectively. Most of the instructions are standard. The *assume* instruction states that a Boolean condition must hold. Notice that common language constructs like *if (b) then* s_1 *else* s_2 can be reduced to $(assume\ b; s_1) \star (assume\ \neg b; s_2)$. Loops can be transformed into our simple language in a similar manner.

Precondition Inference. Traditionally, the weakest precondition for reaching the target is computed by performing the analysis backwards: we start with an initial postcondition $true$ at the target and repeatedly transform it according to the semantics of the current instruction. Algorithm 1 shows the standard predicate transformers for various instructions [22].

For example, an assignment $v := e$ transforms the precondition, ϕ, into $\phi[e/v]$ by replacing variable v, in ϕ with expression, e. The load and the store statements can be similarly handled [18] by introducing a new symbol $Sym(*v)$ to represent the dereference of pointer v, and taking care of possible aliases of v in $alias(v)$. The interprocedural analysis is facilitated by renaming the variables in ϕ according to the contexts of the callee or the caller, details omitted here due to the limit of space. As an example, in Figure 2, $wp(l_{13}, l_{18}) \equiv (z < 2y \wedge v < 60 \wedge x < 40 \wedge v = y - x \wedge w > 10)$ by applying the transformer of $assume$ (Line 9-10, Algorithm 1) successively. The assignment in Line 12 can then transform $w > 10$ in $wp(l_{13}, l_{18})$ into $(w > 10)[-w + 15/w] \equiv w < 5$.

In practice, we start from the target location and propagate the condition backwards into multiple paths, which need to be combined and approximated to make the analysis sound and tractable. While Algorithm 1 gives a precise characterization of a precondition, it outputs complex constraints that are expensive to reason. Therefore, trade-offs are needed between the precision of the preconditions and the overhead in reasoning.

Algorithm 1 Predicate transformers for precondition inference

Input: $t = l : i : l'$: target instruction, ϕ: postcondition at l'
Output: a set of preconditions at l

```
1:  procedure COMPUTEPRECOND(t, φ)
2:      match i
3:          case v := e
4:              return {φ[e/v]}
5:          case v1 := *v2
6:              return {φ[Sym(*v2)/v1]}
7:          case *v := e
8:              return {φ[e/Sym(*u)] | u ∈ alias(v)}
9:          case assume b
10:             return {φ ∧ b}
11:         case goto l0
12:             return {φ}
13: end procedure
```

B. Backward Interval Analysis

We aim for an analysis that is sound, ensuring that all pruned paths can not reach the target, and precise, pruning away a large proportion of infeasible paths. Our designed analysis is shown in Algorithm 2. Given a target location l'_0, it computes for location l a set of preconditions $\hat{wp}(l, l'_0)$ that is necessary for reaching the target (formally, $wp(l, l'_0) \Rightarrow \bigvee \hat{wp}(l, l'_0)$). We abbreviate $\hat{wp}(l, l'_0)$ as $\hat{wp}(l)$ when the target location is clear from the context.

The algorithm is parameterized by two functions, α and γ, for reasoning the path conditions (mentioned later). To be sound, we need to over-approximate the effects of all backward paths starting from the target point. We achieve this by using a worklist containing all active (instruction, postcondition) pairs.

More specifically, the target t and initial postcondition $true$ are added into the worklist (Line 3). During the analysis, whenever an item $(\langle l, i, l' \rangle, \phi)$ is popped from the worklist (Line 5), the postcondition, ϕ, is transformed according to instruction, i, (Line 6), as discussed in Algorithm 1. After this step, every newly computed precondition, ϕ', is further propagated backwards to update the value of $\hat{wp}(l)$, potentially adding new items to the worklist if $\hat{wp}$ is updated (Lines 7-24, discussed later). We soundly consider all backward paths with the predicate transformers iteratively applied and the worklist tracking all active executions.

As an example, in Figure 2, starting from the target location l_{18} and initial condition $true$, Algorithm 2 performs the analysis backwards up to l_{14}. we are then faced with a conditional branch at Line 9 and the path splits into two paths p_1 and p_2. When reaching l_8, the accumulated condition for p_1 and p_2 is pc_1 and pc_2, respectively:

$$pc_1 : \bigwedge \{z < 2y, v < 60, x < 40, v = y - x, w < 5, x \geqslant 20\}$$
$$pc_2 : \bigwedge \{z < 2y, v < 60, x < 40, v = y - x, w > 25, x < 20\}$$

Algorithm 2 Backward interval analysis

Input: target $t = \langle l_0 : i : l'_0 \rangle, \alpha : condition \to \Lambda, \gamma : \Lambda \to condition$
Output: $\hat{wp} : Location \to \{condition\}$

```
1: procedure PRECONDINFER(t)
2:     ŵp(l'_0) ← {true}
3:     Worklist wl ← {(⟨l_0, i, l'_0⟩, true)}
4:     while wl not empty do
5:         (⟨l, i, l'⟩, φ) ← pop(wl)
6:         preconds ← computePreCond(⟨l, i, l'⟩, φ)
7:         for φ' ∈ preconds do
8:             updated ← false
9:             if size of ŵp(l) is less than threshold then
10:                ŵp(l) ← ŵp(l) ∪ {φ'}
11:                updated ← true
12:            else
13:                S' ← joinPaths(ŵp(l), φ')
14:                if S' ≠ ŵp(l) then
15:                    updated ← true
16:                    ŵp(l) ← S'
17:                end if
18:            end if
19:            if updated then
20:                for all e = ⟨l'' : inst : l⟩ do
21:                    wl ← wl ∪ {(e, ŵp(l).lastAddedElem)}
22:                end for
23:            end if
24:        end for
25:    end while
26: end procedure
27: procedure JOINPATHS(conds, φ')
28:    c ← pickOne(conds)
29:    newabs ← α(φ') ⊔ α(c)
30:    c' ← γ(newabs)
31:    return conds\{c} ∪ {c'}
32: end procedure
```

As dicussed in II-B, reasoning complex conditions such as pc_1 and pc_2 are hard, which usually requires a prohibitively expensive SMT solver. Moreover, since the two paths p_1 and p_2 confluence at l_8, their effects need to be combined in $\hat{wp}(l_8)$ to maintain soundness with certain precision loss. To be efficient, we propose to use an interval abstraction to support both lightweight reasoning of path conditions and a sound over-approximation of backward paths.

To this end, we utilize an abstraction function α (discussed later) that abstracts the path condition into the lifted interval domain $\Lambda : V \to Interval$, where $Interval \stackrel{\text{def}}{=} \{[a, b] \mid a \leqslant b, a \in Integer \cup \{-\infty\}, b \in Integer \cup \{+\infty\}\} \cup \{\bot\}$. We use $\top$ to denote the interval with $a = -\infty, b = +\infty$, $\bot$ to denote the empty interval, and the notation (a, b) to represent an open interval. Its corresponding concretization function γ maps the abstracted value back to the logical constraints in a straightforward way, and is defined by $\gamma(\Lambda) = \bigwedge_{v \in dom(\Lambda)} cons(v)$ where

$$cons(v) = \begin{cases} true, & \text{if } \Lambda(v) = \top \\ a \leqslant v \leqslant b, & \text{else if } \Lambda(v) = [a, b] \\ false, & \text{otherwise} \end{cases}$$

With the help of interval abstraction, we soundly combine different backward paths at l. $\hat{wp}(l)$ is a summary that records the already propagated conditions at l, in the form of interval abstractions. When a newly computed condition ϕ' reaches l, we first check if $\hat{wp}(l)$ is empty (Line 9 of Algorithm 2, for now assume $threshold = 1$). If this is the case, ϕ' is recorded in $\hat{wp}(l)$. Otherwise we combine ϕ' with $\hat{wp}(l)$ in *joinPaths* (Line 13) by joining[2] $\alpha(\phi')$ and $\alpha(c)$ into a new abstract element $newabs$, where c is the only condition in $\hat{wp}(l)$ (Line 29). We then use the concretization function γ to map $newabs$ back to the constraints (Line 30), and set it as the new value of $\hat{wp}(l)$. Finally, if $\hat{wp}(l)$ has an update, the newly added condition further propagates backwards by entering the worklist (Lines 19-23).

Notice that the use of the interval abstraction enables us to efficiently reason about path conditions. Additionally, by always combining the newly discovered abstract elements with the join operation, all backward paths are soundly over-approximated without enumeration, alleviating the path explosion problem. In the previous example, if $\hat{wp}(l_8) = \{pc_1\}$, and a new condition pc_2 propagates to l_8, the abstraction of pc_1 and pc_2 are as follows:

Abstraction	v	w	x	y	z
$\alpha(pc_1)$	$(-\infty, 60)$	$(-\infty, 5)$	$[20, 40)$	$(-\infty, 100)$	$(-\infty, 200)$
$\alpha(pc_2)$	$(-\infty, 60)$	$(25, +\infty)$	$(-\infty, 20)$	$(-\infty, 80)$	$(-\infty, 160)$
$\alpha(pc_1) \sqcup \alpha(pc_2)$	$(-\infty, 60)$	$\top$	$(-\infty, 40)$	$(-\infty, 100)$	$(-\infty, 200)$

In *joinPaths*, we combine the two paths by joining the interval abstractions of pc_1 and pc_2, and replace pc_1 in $\hat{wp}(l_8)$ with $c' = \gamma(\alpha(pc_1) \sqcup \alpha(pc_2)) \equiv v < 60 \wedge x < 40 \wedge y < 100 \wedge z < 200$. Since the new condition is different from pc_1 meaning that $\hat{wp}(l_8)$ has been updated, c' is propagated backwards further, summarizing the effects of both paths.

The interval abstraction used in the above analysis can be imprecise in practice. We propose two optimization methods that improve its precision without harming its speed too much:

1) We design an interval abstraction α that tracks certain inter-variable relations explicitly.
2) We design a bounded disjunction strategy that determines when and how to perform the join operations.

C. Optimizations for Maintaining Precision

Relationship Preservation. The interval abstraction α is used in Algorithm 2 to deduce ranges for variables occurring in the path conditions. To make the inferred ranges both sound and precise, we design the inference rules shown in Figure 5 to perform the interval abstraction.

[2]join is replaced by widening after a finite number of steps to ensure termination, as is standard in abstract interpretation [32].

$$\frac{e \in E \cup B, \Lambda(e) \in \mathit{Interval} \quad (\Lambda, e) \Downarrow \mathit{val}}{\mathit{update}(\Lambda, e, \mathit{val})} \tag{1}$$

$$\frac{(\Lambda, b_1 \wedge b_2) \Downarrow \mathit{true}}{(\Lambda, b_1) \Downarrow \mathit{true}, (\Lambda, b_2) \Downarrow \mathit{true}} \tag{2}$$

$$\frac{(\Lambda, \neg b_1) \Downarrow b, b \in \{\mathit{true}, \mathit{false}\}}{(\Lambda, b_1) \Downarrow \neg b} \tag{3}$$

$$\frac{}{(\Lambda, c) \Downarrow [c, c]} \tag{4}$$

$$\frac{\Lambda(e_1) = itv_1, \Lambda(e_2) = itv_2}{(\Lambda, e_1\ \mathit{binOp}\ e_2) \Downarrow (itv_1\ \widehat{\mathit{binOp}}\ itv_2)} \tag{5}$$

$$\frac{\Lambda(e\ \mathit{cmp}\ c) = \mathit{true}}{(\Lambda, e) \Downarrow \begin{cases} (-\infty, c), & \mathit{cmp} = \text{“}<\text{”} \\ [c, c], & \mathit{cmp} = \text{“}=\text{”} \\ (c, +\infty) & \mathit{cmp} = \text{“}>\text{”} \end{cases}} \tag{6}$$

$$\frac{\Lambda(e_1\ \mathit{cmp}\ e_2) = \mathit{true}, \Lambda(e_2) = [a, b]}{(\Lambda, e_1) \Downarrow \begin{cases} (-\infty, b), & \mathit{cmp} = \text{“}<\text{”} \\ [a, b], & \mathit{cmp} = \text{“}=\text{”} \\ (a, +\infty) & \mathit{cmp} = \text{“}>\text{”} \end{cases}} \tag{7}$$

$$\frac{\Lambda(e_1\ \mathit{op}\ e_2) = [a, b], \Lambda(e_2) = [c, d]}{(\Lambda, e_1) \Downarrow [a, b]\ \mathit{rev}(\widehat{\mathit{op}})\ [c, d]} \tag{8}$$

$$\mathit{update}(\Lambda, e, v) = \begin{cases} \Lambda[e \mapsto v], & e \notin \mathit{dom}(\Lambda) \\ \bot, & \Lambda(e) \sqcap v = \bot \\ \Lambda[e \mapsto \Lambda(e) \sqcap v], & \text{otherwise} \end{cases} \tag{9}$$

$$\frac{\mathit{update}(\Lambda, e, \mathit{val}) = \bot}{\mathit{UNSAT}} \tag{10}$$

Figure 5: Inference rules for interval abstraction. Especially, we design the rules (7) and (8) for interval analysis transformer to maintain relations among variables during the precondition inference.

Expression	Newly Discovered Intervals	
	BEACON	Conventional
① $v < 60$	Rule (6), $v \in (-\infty, 60)$	$v \in (-\infty, 60)$
② $x < 20$	Rule (6), $x \in (-\infty, 20)$	$x \in (-\infty, 20)$
③ $w > 25$	Rule (6), $w \in (25, +\infty)$	$w \in (25, +\infty)$
④ $v = y - x$	Rule (7), $y - x \in (-\infty, 60)$	$v \in (-\infty, +\infty)$
⑤ $y - x$	Rule (8), $y \in (-\infty, 80)$	N/A
⑥ $2y$	Rule (5), $2y \in (-\infty, 160)$	N/A
⑦ $z < 2y$	Rule (7), $z \in (-\infty, 160)$	$z \in (-\infty, +\infty)$

Figure 6: Example of interval abstraction inference for $pc_2 = \bigwedge\{z < 2y, v < 60, x < 20, v = y - x, w > 25\}$ using our inference rules and by conventional interval analysis. N/A means no inference rule is applicable.

In the conventional interval analyses [38], [39], each statement occurring in the program transforms the interval abstract state. In similar spirits, we design a top-down analysis that performs a recursive descent traversal over the path conditions and propagates the known interval values along the way to infer new interval values in a sound manner by respecting laws of interval arithmetic [40]. Unlike conventional interval analysis, our analysis tracks intervals for not only variables but also expressions occurring in the path conditions. Afterward, we propagate these value ranges of expressions to their parents and child expressions to make the analysis more precise.

In Figure 5, we use Λ to represent a map from expressions to their interval value ranges. For convenience, the boolean constants $true, false$ are represented as interval $[1, 1], [0, 0]$ respectively. Given constraint e, the inference starts with $(\Lambda, e) \Downarrow true$. $(\Lambda, e) \Downarrow val$ means that a new interval value val has been inferred for e, and triggers $update(\Lambda, e, val)$ (Rule (1)). If we derive a contradiction from the old interval $\Lambda(e)$ and the new interval *val*, the entire condition is indeed unsatisfiable, and the computation ends. Otherwise, $\Lambda(e)$ is updated by intersecting *val* (Rules (9)-(10)). Rules (2)-(3) recurses down and carries the interval values to the sub-logical formulas. Rules (4)-(6) corresponds to the abstract state transformer used in conventional interval analysis: a constant c gets range $[c, c]$ (Rule (4)); In Rule (5), the arithmetic operation, op, and the comparison operation, cmp, (denoted uniformly by $binOp$) are replaced by their interval counterparts: $\widehat{op} \in \{\widehat{+}, \widehat{-}, \widehat{\times}, \widehat{\div}\}$ and $\widehat{cmp} \in \{\widehat{<}, \widehat{>}, \widehat{=}\}$, respectively; Rule (6) updates the range of an expression based on the conditional test against a constant (The case when $\Lambda(e_1\ cmp\ c) = \mathit{false}$ is similar and omitted).

Rules (7)-(8) encode heuristics to improve the precision of the interval abstraction. In these rules, we try to refine the range of a given expression based on the updated ranges of other expressions. Rule (7) extends Rule (6) to consider the comparison between any two expressions, utilizing the range of one expression and the comparison result to refine the range of the other expression. In Rule (8), when the parent expression is an arithmetic expression $e_1\ op\ e_2$ and its range is updated to $[a, b]$, we can refine the range of its operand by reversing the binary operator in the interval domain: $rev(\widehat{+}) = \widehat{-}$, $rev(\widehat{\times}) = \widehat{\div}$, etc. Since the expressions in Rules (7)-(8) may preserve certain inter-variable relations not captured by the conventional interval domain, these rules may bring precision improvement through refinement.

Figure 6 shows how we apply these inference rules on a simplified version of condition pc_2. Starting with $(\Lambda, pc_2) \Downarrow true$, we deduce from the conjunction in pc_2 and rule (2) that all conjunct subexpressions $z < 2y, v < 60, x < 20, v = y - x, w > 25$ are $true$. Subexpreesions that compares a variable with a constant enable us to infer ranges for the involved variables (Steps ① − ③). Step ④ propagates the known interval for v to expression $y - x$. Step ⑤ uses the updated range of $y - x$ to refine the interval of its subexpression y to $\Lambda(y-x)\widehat{+}\Lambda(x) = (-\infty, 60)\widehat{+}(-\infty, 20) = (-\infty, 80)$, and

the refined interval for y in turn propagates to $2y, z$ in ⑥−⑦.

Steps ④, ⑤, ⑦ of our approach in Figure 6 clearly shows that our additional inference rules (7)-(8) utilize the relations among variables encoded in the form of expression to propagate the interval values, and hence achieve higher precison. If we solely apply the abstract state transformers of the interval domain to conjuncts of pc_2, as in conventional interval analysis, the result is not as precise as what our rules produce. The rightmost column of Figure 6 shows that the conventional interval analysis can obtain the same intervals for v, x, w in ① − ③. However, it can not further refine the range for y and z: At Step ④ when facing the expression, $v = y - x$, the conventional interval analysis tries to deduce the interval of v by subtracting interval of x from the interval of y. Since y is unbounded at ④, no useful intervals can be inferred. Moreover, since the conventional method does not track the range of the expression $v = y - x$, it has lost all the relations among variables occurring in the expression, leading to imprecise results for y and z.

Bounded Disjunctions. As discussed in Section III, by keeping the propagated conditions from different paths separately, and selectively joining them, we stand a better chance of gaining precision. Specifically, we design a bounded disjunctions strategy to maintain precision during the backward propagation. With a given bound $threshold$, we preserve the propagated conditions to a program location l in a set $\hat{wp}(l)$, detailed in Algorithm 2. When the number of paths reaching l is less than the $threshold$, their conditions are kept separately in $\hat{wp}(l)$ and propagate backwards individually (Lines 9-11). Therefore, we can take the precision benefits from the disjunctive form whenever the size is less than $threshold$. With $threshold > 1$, Algorithm 2 outputs a set of abstracted conditions in $\hat{wp}(l)$ when it finishes. We obtain the final precondition through a disjunction over $\hat{wp}(l)$. For example, with the two paths p_1 and p_2 reaching line, l_8, in Figure 2, the precondition at l_8 on w is $w \in (-\infty, 5) \vee (25, +\infty)$, which comes from an explicit disjunction of interval abstractions $\alpha(pc_1)$ and $\alpha(pc_2)$. Nonetheless, the number of paths could grow extremely large. Thus, we still need to join parts of the interval abstraction from different paths for efficiency.

This is carried out by the *joinPath* function in Algorithm 2: When a new condition, ϕ', propagates to line, l, but the size of $\hat{wp}(l)$ has already reached the predefined $threshold$ (Line 13), we pick one of the stored conditions in $\hat{wp}(l)$, denoted by c (Line 28), and join the interval abstractions of ϕ' and c (Line 29). The joint result is concretized, and replace c in $\hat{wp}(l)$, keeping the number of tracked conditions in $\hat{wp}(l)$ lower than the $threshold$.

Different choices of conditions to join may lead to different damages of precision. For example, in Figure 7, there are different levels of precision losses of joining the interval abstractions of w. We define the problem of minimizing precision loss in join as finding the smallest intersection area among different polygon search spaces described by each abstraction. Conventionally, the intersections of two abstract domains can

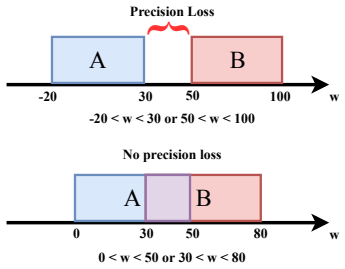

Figure 7: Precision loss of joining different disjunctive path conditions for backward precondition inference.

be measured by the Fréchet distance [41]. Specifically for our interval domain, we only need to calculate the precision loss by accumulating the distance from the interval of each variable. Thus, the intersection can be measured by the distance of two intervals, which is defined as:

$$L = \sum_{v \in \phi_1 \cap \phi_2} \min(\max(0, u_1 - l_2), \max(0, u_2 - l_1))$$

where v is a shared variable of the path conditions ϕ_1 and ϕ_2, whose interval abstractions are v_{ϕ_1}=$[l_1, u_1]$ and v_{ϕ_2}=$[l_2, u_2]$.

As an example, In Algorithm 2, suppose the program location l already has two paths reaching it and $\hat{wp}(l) = \{B, C\}$ records their propagated conditions, and a new condition with interval abstraction A now reaches l:

$$\text{A: } x \in (-\infty, 20) \wedge y \in (20, 50)$$
$$\text{B: } x \in (30, 50) \wedge y \in (60, 70)$$
$$\text{C: } x \in (90, 140) \wedge y \in (100, +\infty) \wedge z \in (200, +\infty)$$

With $threshold = 2$, A needs to be joined with either B or C to restrict the numebr of paths. we can measure the precision loss induced by joining as distance L: $L(A, B)$=10+10=20, and $L(A, C)$=70+50=120. We choose to join A, B since it suffers from the minimum precision loss, and the joined abstraction used for further propagation is:

$$x \in (-\infty, 50) \wedge y \in (20, 70)$$

D. Precondition Instrumentation

We instrument the program using the inferred necessary precondition to prune infeasible paths at runtime. Notice that the elaborative analysis result may put a heavy burden on instrumentation and increase the runtime overhead of fuzzing: it contains a map from variable to its disjunctive interval values at various program locations. Therefore, we need to perform the instrumentation selectively to reduce the overhead.

We observe that it is unnecessary to instrument and check the values of all variables. Also, it is unnecessary to check for one variable at various program locations. For instance, in Figure 2, instrumentation is added for u (Line 7) but not v, since the value of u determines the value of v. Also, we place the instrumentation for variables w, x, y, z right after their definitions in Line 4, instead of checking them at every program location that uses these variables since their values do not change after definition. Based on these observations, we apply the lightweight instrumentation as following:

1) We first transform the program into SSA form [42], and only consider variable definitions as the candidate program locations for instrumentation. This is correct because the SSA form guarantees that a variable is not written after being defined.
2) When the value of a variable v_1 depends only on another variable v_2, v_1 should not be instrumented. Such information can be computed by the reaching definition data flow analysis [43]. As an example, in Figure 2, v depends only on u instead of x or z.

V. Evaluation

We implemented Beacon, a grey-box fuzzer with a precondition analysis and an instrumentation component, based on LLVM [44]. That is, as shown in Figure 3, we first compile the input source code to LLVM bitcode, on which the precondition analysis, the instrumentation for checking preconditions, and other coverage-related instrumentation are performed. After instrumentation, the LLVM bitcode is compiled to an executable binary, which can be integrated with various fuzzing engines, such as AFL [12], AFLGo [1], amongst many others. By default, we choose to use AFLGo as the fuzzing engine.

With the implementation, we conducted a series of experiments to evaluate the effectiveness of Beacon. First, we compared Beacon with four state-of-the-art (directed) fuzzers in the application scenario of vulnerability reproduction (Section V-A). This experiment aims to show that Beacon is far more efficient than existing directed fuzzers, and the performance of existing non-directed fuzzers can be notably improved when armed with the path pruning methods of Beacon. Second, since Beacon prunes paths based on both path slicing (slices away infeasible paths based on the reachability on the control flow graph) and precondition checking (prunes infeasible paths according to the precondition analysis), and thus, to better understand the two strategies, we also evaluated how much they contribute to the time reduction in fuzzing (Section V-B). Third, we argued that our precondition analysis is both precise and fast due to two techniques, namely relationship preservation and bounded disjunction. Therefore, we evaluated their impacts on fuzzing by removing them from the static analysis, respectively (Section V-C). Fourth, we also evaluated the runtime overhead introduced by our instrumentation, which aims to show the effectiveness of our instrumentation strategy.

Baselines. We compared Beacon with the fuzzers mentioned in Table I. AFLGo [1] and Hawkeye [2] are two recent directed grey-box fuzzers that prioritize inputs so that inputs closer to the target code can be executed in a high priority. Their technical details are mentioned in Section II-A. We also planed to compare with Fuzzguard and Savior. However, Fuzzguard is not open source, and we cannot reproduce the experiments mentioned in their paper in our environments. Savior mainly depends on prioritizing the symbolic execution engine for multiple targets (provided by the address sanitizer) in the programs. Since it is different from grey-box fuzzing and the applicable scenarios are different, we suppose it is orthogonal to our approach.

Table I: Compared fuzzers.

Fuzzer	Category	Description
AFLGO [1]	Directed	Sophisticated seeds prioritization
Hawkeye [2]	Directed	Optimized fitness function + mutation strategies
AFL [12]	Greybox	Evolutionary mutation strategies
Mopt [13]	Greybox	Mutation operator prioritiztion
AFL++ [45]	Greybox	Optimization of overall fuzzing framework

Table II: Real-world benchmark programs and vulnerabilities.

Project	Program	Version	Input format	Num. CVEs
Binutils	cxxfilt	2.26	TXT	2
	objdump	2.28	ELF	7
	objcopy	2.28	ELF	4
Libjpeg	cjpeg	2.04	JPG	1
	cjpeg	1.98	JPG	1
Ming	swftophp	0.4.7	SWF	7
	swftophp	0.4.8	SWF	10
Libxml2	xmllint	20902	XML	4
Lrzip	lrzip	0.631	ZIP	2
Libpng	pngimage	1.6.35	PNG	1
Libpoppler	pdftoppm	0.74	PDF	3
	pdftops	0.74	PDF	1
	pdfdetach	0.71	PDF	3
Libav	avaconv	12.3	AVI/AAC	5

To show the capability of Beacon to cooperate with other fuzzers, we also choose AFL, AFL++, and Mopt, three coverage-guided grey-box fuzzers, to evaluate how our idea of path pruning can improve their performance. AFL is one of the most widely-used fuzzers nowadays, and many existing works are built based on AFL, such as AFLGo and Hawkeye. Mopt and AFL++ are also built upon AFL. The former improves input generation by prioritizing the mutation strategies. The latter integrates with multiple engineer optimizations to improve the overall performance.

Benchmarks. We chose 51 vulnerabilities in 14 real-world programs that have been frequently evaluated in the existing fuzzing frameworks [1], [2]. The chosen programs, which are shown in Table II, also have diverse functionalities as well as different program sizes. Moreover, the vulnerabilities chosen are either causing multiple issues (cause several CVEs) or too complicated to be fixed completely even after several patches.

Configurations. The initial seed corpus determines the effectiveness of fuzzing [46]. To achieve the best performance of related work, we used the seeds provided by AFLGo in their Github repository[3] with the intuition that the related works should perform better in their own proposed setting. By experience, we set the $threshold = 5$ for bounded disjunction in Beacon. We conducted every experiment 10 times and, for each time, the experiment is run with a time budget of 120 hours. Besides, we employed the Mann-Whitney U Test [47] to demonstrate the statistical significance of the contribution made by each part of our framework.

All experiments were conducted on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 64GB of memory running Ubuntu 16.04 LTS.

[3]https://github.com/aflgo/aflgo/tree/master/scripts/fuzz

Table III: Comparing to AFLGo with 10 repeated experiments of vulnerability reproduction. T_{sa} and T_f are the time cost of static analysis and fuzzing, respectively. N is the number of executions. F is the ratio of the executions that are early stopped by BEACON.

No.	Program	CVE	AFLGo	Beacon				
			T_{AFLGo}	T_{sa}	T_f	T_{all}	N	F
1	ming$_{4.7}$	2016-9827	1.25h	43s	0.31h	0.32h	0.28M	80.7%
2		2016-9829	T.O.	18s	5.54h	5.55h	5.25M	82.6%
3		2016-9831	2.52h	16s	0.62h	0.62h	0.47M	84.3%
4		2017-7578	2.43h	20s	0.29h	0.30h	0.26M	80.8%
5		2017-9988	37.99h	20s	1.45h	1.46h	1.26M	72.3%
6		2017-11728	T.O.	27s	11.14h	11.15h	23.70M	84.6%
7		2017-11729	4.34h	27s	1.02h	1.03h	2.02M	82.0%
8	ming$_{4.8}$	2018-7868	T.O.	21s	1.75h	1.76h	3.91M	85.8%
9		2018-8807	10.71h	16s	1.89h	1.89h	4.42M	83.9%
10		2018-8962	35.39h	20s	1.92h	1.93h	5.96M	88.8%
11		2018-11095	60.29h	20s	3.13h	3.14h	6.07M	84.9%
12		2018-11225	34.23h	17s	2.84h	2.84h	4.33M	91.7%
13		2018-11226	37.59h	18s	3.98h	3.99h	5.25M	89.0%
14		2018-20427	T.O.	21s	3.14h	3.15h	7.81M	86.2%
15		2019-9114	T.O.	22s	3.53h	3.54h	7.08M	84.0%
16		2019-12982	T.O.	20s	2.47h	2.48h	4.12M	82.3%
17		2020-6628	T.O.	24s	3.91h	3.92h	8.76M	84.3%
18	lrzip	2017-8846	5.05h	61s	1.78h	1.80h	1.32M	86.4%
19		2018-11496	3.01h	68s	1.17h	1.19h	0.89M	92.4%
20	cxxfilt	2016-4491	7.74h	2,229s	1.38h	2.00h	7.69M	95.9%
21		2016-6131	5.88h	2,258s	0.84h	1.47h	3.76M	94.9%
22	xmllint	2017-5969	2.07h	5,381s	0.17h	1.66h	0.33M	95.1%
23		2017-9047	T.O.	5,238s	16.55h	18.01h	16.46M	83.6%
24		2017-9048	T.O.	7,049s	18.00h	19.96h	18.47M	85.1%
25		2017-9049	T.O.	5,672s	31.56h	33.14h	40.15M	95.2%
26	objdump	2017-8392	T.O.	2,654s	8.42h	9.16h	1.97M	79.3%
27		2017-8396	T.O.	2,909s	39.03h	39.84h	95.85M	91.2%
28		2017-8397	T.O.	3,067s	83.46h	84.31h	261.2M	96.1%
29		2017-8398	T.O.	2,825s	40.51h	41.29h	166.3M	96.0%
30		2017-14940	T.O.	3,420s	61.38h	62.33h	41.13M	86.1%
31		2017-16828	T.O.	3,326s	22.24h	10.59h	4.32M	94.3%
32		2018-17360	T.O.	2,950s	45.69h	46.51h	121.55M	92.6%
33	objcopy	2017-7303	T.O.	2,033s	20.09h	20.65h	31.75M	85.7%
34		2017-8393	T.O.	2,484s	19.78h	20.47h	20.21M	90.5%
35		2017-8394	T.O.	2,671s	4.46h	5.20h	4.74M	92.3%
36		2017-8395	T.O.	2,608s	3.83h	4.55h	4.31M	96.9%
37	cjpeg	2018-14498	49.78h	93s	11.46h	11.46h	20.12M	91.2%
38		2020-13790	7.34h	106s	3.98h	4.01h	19.78M	90.4%
39	pngimage	2018-13785	T.O.	85s	3.22h	3.23h	65.81M	91.3%
40	pdftoppm	2019-10872	T.O.	4,904s	102.90h	104.26h	2.98M	76.5%
41		2019-10873	T.O.	5,899s	90.25h	91.89h	3.76M	69.4%
42		2019-14494	T.O.	4,153s	95.35h	96.50h	3.13M	67.2%
43	pdftops	2019-10871	T.O.	6,593s	62.54h	64.37h	12.94M	75.2%
44	pdfdetach	2018-19058	T.O.	2,950s	73.98h	74.80h	14.60M	86.3%
45		2018-19059	T.O.	2,686s	82.46h	83.21h	14.21M	81.1%
46		2018-19060	T.O.	2,995s	92.65h	93.48h	13.76M	70.1%
47	avconv	2018-11102	T.O.	14,335s	89.57h	93.55h	6.00M	53.4%
48		2018-11224	T.O.	14,893s	98.20h	102.34h	7.65M	69.8%
49		2018-18829	T.O.	14,623s	47.97h	52.03h	3.44M	56.7%
50		2019-14441	T.O.	16,600s	52.86h	57.47h	6.89M	51.3%
51		2019-14443	T.O.	14,239s	95.81h	99.77h	13.49M	52.6%
Avg.						**11.50x**		**82.9%**

T.O.: time outs (>120 hours)

Table IV: Comparing to AFLGo and Hawkeye of average results (s) from 10 repeated experiments. Since Hawkeye is not open source, we use the data reported in its paper. The p-value is the comparison between AFLGo and BEACON.

CVE	AFLGo	Hawkeye	Beacon	p-value
2016-4487/8	412 (x2.7)	177 (x1.1)	151	0.00906
2016-4489	567 (x3.1)	206 (x1.1)	180	0.00804
2016-4490	306 (x3.7)	103 (x1.2)	82	0.01596
2016-4491	27,880 (x5.6)	18,733 (x3.6)	4985	0.00018
2016-4492	540 (x1.7)	477 (x1.5)	325	0.00804
2016-6131	21,180 (x7.3)	17,314 (x5.7)	3013	0.00018

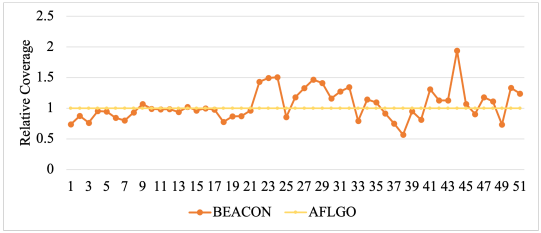

Figure 8: Coverage comparison between AFLGo and BEACON. The x-axis is the CVEs listed in Table III. The y-axis is the relative coverage compared with AFLGo.

A. Compared to the State of the Art

To compare BEACON to the (directed) fuzzers in Table I, we ran them to reproduce the CVE-identified vulnerabilities listed in Table III.

1) Compared to AFLGo: Table III lists the comparison results, where we show the time cost of AFLGo and BEACON for reproducing the vulnerabilities, as well as the time cost of the static analysis employed by BEACON. We observe that AFLGo fails to reproduce 34 out of the 51 vulnerabilities in 120 hours, while BEACON can succeed in reproducing all of them, 23 within 5 hours, 33 in 24 hours (including the time of static analysis). Overall, BEACON achieved 1.2x to 68.5x speedup, 11.50x speedup on average, compared to AFLGo. Table III also lists the number of program executions for reproducing a vulnerability, as well as the ratio of the executions that can be stopped early by BEACON. We can observe that in most cases, more than 80% of the executions can be stopped early during fuzz testing, which allows BEACON to save much time.

We observe that the time spent for the precondition analysis in BEACON is at most 5 hours, and, in many cases, it can complete in only a few minutes. As discussed above, even with the static analysis time, BEACON is still much faster than AFLGo. In practice, we can further speed up the static analysis by leveraging other techniques, e.g., incremental analysis, proposed by the static analysis community. However, as this is out of the scope of this paper, we leave it as our future work.

In addition to the experiments of vulnerability reproduction, we also ran BEACON and AFLGo to test the patches of the CVE-identified vulnerabilities. Surprisingly, BEACON detected 3, 9, and 2 incomplete patches in Binutils, Ming, and Lrzip, respectively, and 8 additional bugs, whereas AFLGo only detected 6 incomplete patches. We have reported the detected issues to the developers, and all of them have been confirmed. All links to the bug reports are available through this link.

Moreover, we evaluated the coverage achieved when the vulnerabilities are reproduced. The results are shown in Figure 8. Interestingly, we find the coverage achieved by BEACON and AFLGo fluctuates. On the one hand, BEACON needs less coverage (91.2% on average) for those vulnerabilities reproducible for AFLGo. On the other hand, BEACON could achieve higher coverage in some scenarios, especially when AFLGo cannot reproduce the vulnerabilities. We find the reason is that AFLGo spends too much time executing infeasible paths, and thus the execution filtration rates are high ($>$ 80%) as shown in Table III.

2) Compared to Hawkeye: We also tried to compare with Hawkeye, the other recent directed grey-box fuzzer, that was

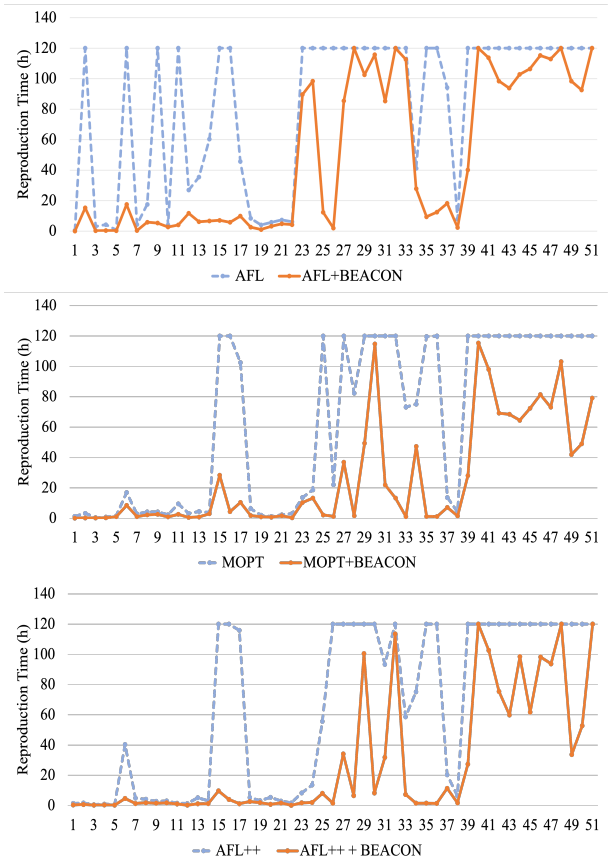

Figure 9: Reproduction time comparison among AFL, AFL+BEACON, Mopt, Mopt+BEACON AFL++, and AFL++ +BEACON. The x-axis is the vulnerabilities in Table III. The y-axis is the average reproduction time in 10 repeated experiments. We used 120h as the timeout bound.

reported to be more effective than AFLGo. However, Hawkeye is not open source. Thus, we tried to reproduce Hawkeye's experiments using AFLGo and BEACON. We then compared the results with those reported in Hawkeye's paper. The results are shown in Table IV. BEACON outperformed AFLGo and Hawkeye for reproducing all the vulnerabilities, with all p-values less than 0.05. For CVE-2016-4491 and CVE-2016-6131, in particular, BEACON can achieve a 3.6x and 5.7x speedup compared to Hawkeye.

3) Compared to AFL, AFL++, and Mopt: Since the path-pruning idea is orthogonal to existing fuzzing techniques, the idea of BEACON can be leveraged to speed up almost all fuzzers for the thorough testing of a specific target in a given program. To illustrate the generality of our idea, we integrate BEACON with AFL, AFL++, and Mopt, three non-directed fuzzers, to help them prune paths for reproducing vulnerabilities in Table III. The results in Figure 9 show the improvement brought about by BEACON for non-directed fuzzing. On average, compared to the original tools, AFL+BEACON AFL+++BEACON, and Mopt+BEACON can achieve 6.31x, 11.86x and 10.92x speedup, respectively.

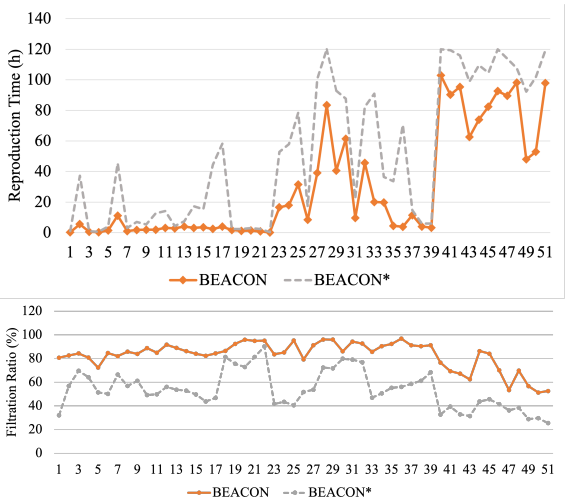

Figure 10: Comparison of BEACON and BEACON*. The x-axes are the CVE-identified vulnerabilities listed in Table III. The y-axes are the reproduction time and the ratio of paths that are stopped early, respectively. We used 120h as the maximum timeout budget.

B. Impacts of Path Slicing & Precondition Checking

Recall that, in addition to the precondition-based path pruning, BEACON also leverages conventional reachability analysis on the control flow graph to slice away paths that simply cannot reach the target code. To evaluate how path slicing and precondition checking contribute to the time reduction in BEACON, we also set up a naive variant of BEACON, BEACON*, which disables the precondition analysis. We then reran the experiments discussed before using BEACON and BEACON*. The experimental results are shown in Figure 10, where we can observe that BEACON is much faster (1.1x to 18.4x) than BEACON* for reproducing the vulnerabilities, as it prunes 29.1% more paths than BEACON* on average. In some cases (e.g., CVE-2017-8397), BEACON* cannot even reproduce the vulnerability in 120 hours. This result demonstrates the significance and necessity of the precondition analysis, which allows us to achieve notable performance improvement.

C. Impacts of Relation Preservation & Bounded Disjunction

To effectively prune paths during fuzzing, we proposed a dedicated precondition analysis that is armed with two key strategies, i.e., relationship preservation and bounded disjunction, to ensure both scalability and precision. To evaluate how relationship preservation and bounded disjunction contribute to the time reduction in BEACON, we also set up another variant of BEACON, BEACON-*rp* and BEACON-*bd*, which disable the bounded disjunction and the relationship preservation, respectively. We then reran the experiments discussed before using BEACON-*rp* and BEACON-*bd*.

The experimental results are shown in Figure 11, where we can observe that BEACON is much faster (1.05x to 4.9x, 1.05x to 5.34x) than BEACON-*rp* and BEACON-*bd* for reproducing the vulnerabilities, respectively. This result demonstrates the

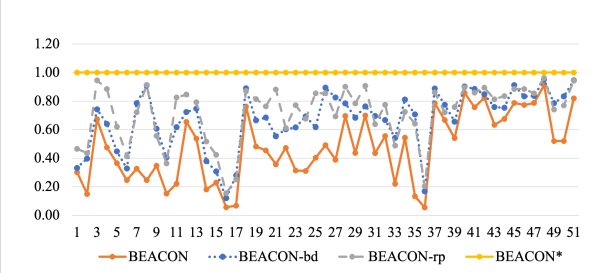

Figure 11: Reproduction time comparison of BEACON BEACON-*rp*, and BEACON-*bd*. We use the results of BEACON with only reachability pruning as the baseline. The x-axis is the CVE-identified vulnerabilities listed in Table III. The y-axis is the reproduction time compared with BEACON*.

Table V: Different configurations of the bound used for precondition inference and its time costs. We sample one CVE from each project 10 times and record the average filteration ratio with these preconditions till reproducing the CVEs. T_{inf} denotes the time costs, F represents the filtration ratios. *oom* denotes the analysis used up the 30G memory budgets.

Project	5		20		50	
	T_{inf}	F	T_{inf}	F	T_{inf}	F
Ming	27s	81.3%	101s	82.5%	280s	83.0%
Binutils	2560s	90.4%	20.11h	90.8%	*oom*	n/a
Libxml	6418s	88.5%	13.17h	89.1%	*oom*	n/a
Lrzip	67s	87.1%	464s	89.2%	1806s	90.0%
Libjpeg	103s	90.7%	423s	91.3%	1096s	92.6%
Libpng	85s	92.9%	441s	93.7%	3836s	93.8%
Libpoppler	4290s	70.5%	4.91h	71.8%	*oom*	n/a
Libav	14895s	61.1%	14.94h	64.3%	*oom*	n/a

significance and necessity of both strategies, as both of them contribute to the precision of the precondition analysis, and their combination allows us to achieve greater precision and prune a lot more paths during the fuzz testing.

We also studied the influence brought about by the bound threshold used in bounded disjunction. The results are shown in Table V. The execution filtration ratio improves slightly from 0.9% to 3.2% when the bound threshold increase from 5 to 50. However, the time costs burst dramatically and even used up the server memory. Therefore, BEACON uses 5 as the threshold to get a sweet spot of efficiency and effectiveness. Meanwhile, the distance calculated for picking the merging states preserves the program semantics to prevent too much precision loss.

D. Instrumentation Overhead

BEACON prunes the infeasible paths through instrumentation, which may cause additional runtime overhead. To evaluate the runtime overhead, we run the same inputs against two versions of each benchmark program. One is the vanilla version without any instrumentation, and the other is instrumented by BEACON, where we add precondition checks but do not let the program exit early when a precondition check fails. We do not exit the program early because this evaluation needs to ensure that we run the same paths on the two versions. The results are shown in Table VI, where we show the number of executions of each program (N_{exec}), the original time cost (T_{orig}), and the time cost after our instrumentation (T_{Beacon}). We observe that BEACON introduces up to 9.8% runtime overhead and 5.7% on average. We believe that such low overhead is acceptable in practice, and the previous evaluation has shown BEACON is much faster than the existing fuzzers.

Table VI: Runtime overhead comparison with pure instrumentation from AFL and BEACON without filteration using the same inputs generated from the deterministic stage of AFL.

Project	N_{exec}	T_{orig}	T_{Beacon}	Overhead
Ming	126K	4.52m	4.80m	6.2%
Binutils	2.38M	2.48h	2.51h	1.2%
Libxml	25K	0.63h	0.65h	3.2%
Lrzip	1.07M	1.43h	1.57h	9.8%
Libjpeg	2.97M	0.59h	0.61h	3.4%
Libpng	1.41M	7.55h	8.01h	6.1%
Libpoppler	1.21M	23.73h	25.27h	6.5%
Libav	7.62M	23.87h	26.13h	9.5%

```
1 int init(char* input) {
2     int type,length1,length2=extract(input);
3
4     // assertion inserted by Beacon
5     assert(type==5&&length1+length2<42);
6
7     // an overly lengthy function
8     data=processing(input);
9
10    // crash on some condition
11    if(type==5&&length1+length2<42)
12        crash();
13 }
```

Figure 12: An example of a case study.

E. Case Study

To provide a better understanding of why BEACON can achieve good performance as discussed before, we provide an example in Figure 12, which is simplified from a real bug detected by us[4]. In the code snippet, a crash at Line 12 may happen on the branch condition at Line 11. Before the crash, it exists an overly lengthy procedure at Line 8. Thus, if we cannot determine if an input can reach the crash, we have to waste a lot of time on the overly lengthy procedure. BEACON can compute the precondition on which the crash may happen and inserts the precondition before the overly lengthy procedure at Line 5. In this manner, we can stop early before Line 8 if the precondition is violated, whereas conventional directed fuzzers cannot prune any path.

F. Discussion

Assisting other fuzzers: Conventionally, input generation [15], [17] and seed prioritization [1], [2] are the two main dimensions for improving the performance of directed fuzzing. Nonetheless, fuzzers still have the possibility of failing to reproduce the vulnerabilities without a provable guarantee. Therefore, BEACON proposes another direction for directed fuzzing, which prunes the infeasible paths away to minimize the penalty brought about by the randomness.

[4]We cannot provide the original code since it is still reproducible in the newest version of the program and may cause a malicious attack.

Threats to validity: The main concern is the randomness in input generation. Even though we have conducted the experiments multiple times for fairness, different input sequences might influence the outcomes in these projects. Still, the results meet the expectation that BEACON achieves faster crash reproduction than AFLGo by filtering those infeasible paths. Meanwhile, the integration with other fuzzers shows the capability of BEACON in improving existing fuzzing.

Another issue is that we have not proved the capability of BEACON to assist fuzzers relying on symbolic executions. However, since the scalability issue is the major concern of symbolic execution, how to efficiently handle the large-scale programs chosen in the experiment could become another challenge. Therefore, we attempt to tackle this in future work.

VI. RELATED WORK

In addition to the related work discussed in Section II, this section surveys other related work.

A. Directed White-box Fuzzing

The idea of directed fuzzing begins with white-box fuzzing [5], [11], [48], which mainly depends on symbolic or concolic execution, such as Klee [49], to generate an exploitable input for bug reproduction. However, the path explosion problem and the notoriously expensive constraint solving make them hard to scale for real-world programs. Therefore, existing works attempt to leverage the prior knowledge of the vulnerabilities to make symbolic execution focus on the relevant program states.

One direction is to prioritize program paths for the symbolic execution to explore. For example, Hercules [50] uses an unsound function summary to prioritize the reachable paths. Others either rely on bug reports [51], critical system calls [10], or changes in patches [52] to identify the potential bug trace. However, these works usually require manual expertise to ensure the quality of this prior knowledge, which may lead to a varied performance on different programs. The other direction is to accelerate the symbolic execution itself for approaching the targets faster. For example, existing works preserve the execution states either symbolically [53] or concretely [54] with snapshot mechanisms to avoid redundant path explorations. Chopper [55] adapts online static analysis to provide state merging strategies while minimizing the number of analyzed paths on the fly. DiSE [56] identifies the relations among the branching conditions and solves them incrementally.

Even though lots of effort has been devoted to symbolic execution, scalability is still a major concern in research nowadays. This is also why we choose directed grey-box fuzzing, which often exhibits promising scalability in practice.

B. Coverage-guided Fuzzing

Optimizations for conventional coverage-guided fuzzers also have the potential to improve directed fuzzing. First, we can optimize input generation with dynamic taint analysis. The basic idea is to mutate the related input offsets to satisfy the uncovered branch conditions. Other than random mutation, Angora [57] adapts byte-level taint tracking to discover the related input bytes of the target condition, and then applies a gradient-descent-based search strategy. To make the gradient-descent-based search more reasonable, Neuzz [58] proposes to use the neural network to smooth the search progress. There are also some techniques involving a lightweight program analysis and transformation to improve the effectiveness of the mutation. Fairfuzz [59] identifies the input offsets where it is not necessary to change the values, thus, minimizing the input search space improves the efficiency of the mutation. Mopt [13] proposes a novel mutation operator scheduling strategy to adjust mutation strategies for different programs.

The second direction is to integrate fuzzers with concolic/symbolic execution, a.k.a., hybrid fuzzing, for tackling complex and tight path constraints. Hybrid fuzzing combines the advantages of efficient mutation and precise constraint solving to evaluate the programs, which could be the future direction for white-box fuzzing. With the development of fuzz testing, the majority of the path exploring the demand offloads to the fuzzers to avoid the path explosion problem in symbolic/concolic execution. In short, the state-of-the-art hybrid fuzzing selectively solves the path constraints to improve the performance. For example, Driller [60] proposes to solve those uncovered paths for fuzzing rather than exploring all paths with concolic execution. However, how to effectively integrate concolic execution with fuzzing is always under consideration. QSYM [61] solves part of the path constraint for a basis seed and leverages the mutation for validated inputs satisfying the actual condition. Intriguer [62] further replaces symbolic emulation with dynamic taint analysis, which decreases the overhead of modeling a large amount of mov-like instructions. Pangolin [63] proposes to preserve the constraint as an abstraction and reuse it to guide further input generation. Overall, these methods are orthogonal to BEACON and can be integrated with BEACON for better performance.

VII. CONCLUSION

We have presented BEACON, which directs the grey-box fuzzer in the sea of paths to avoid unnecessary program execution and, thus, saves a lot of time cost. Compared to existing directed grey-box fuzzers, BEACON can prune infeasible paths provably and more effectively, via the assistance of a dedicated cheap, sound, and precise static analysis. We have provided empirical evidence that BEACON is more effective than the state-of-the-art (directed) fuzzers.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and opinions for improving this work. Rongxin Wu is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001) and NSFC61902329. Other authors are supported by the RGC16206517 and ITS/440/18FP grants from the Hong Kong Research Grant Council, and the donations from Microsoft Donation and Huawei. Qingkai Shi is the corresponding author.

References

[1] 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

[2] 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

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

[4] 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/

[5] 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

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

[7] 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

[8] 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.

[9] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2619091

[10] 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

[11] 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

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

[13] "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[14] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2139–2154.

[15] 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

[16] 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

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

[18] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 439–452. [Online]. Available: https://doi.org/10.1145/2535838.2535888

[19] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang, "Pse: Explaining program failures via postmortem static analysis," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '04/FSE-12. New York, NY, USA: Association for Computing Machinery, 2004, p. 63–72. [Online]. Available: https://doi.org/10.1145/1029894.1029907

[20] S. Blackshear, B.-Y. E. Chang, and M. Sridharan, "Thresher: Precise refutations for heap reachability," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 275–286. [Online]. Available: https://doi.org/10.1145/2491956.2462186

[21] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: A powerful approach to weakest preconditions," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 363–374. [Online]. Available: https://doi.org/10.1145/1542476.1542517

[22] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, p. 453–457, Aug. 1975. [Online]. Available: https://doi.org/10.1145/360933.360975

[23] C. Urban and A. Miné, "Proving guarantee and recurrence temporal properties by abstract interpretation," in *Verification, Model Checking, and Abstract Interpretation*, D. D'Souza, A. Lal, and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 190–208.

[24] K. Ferles, V. Wüstholz, M. Christakis, and I. Dillig, "Failure-directed program trimming," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 174–185. [Online]. Available: https://doi.org/10.1145/3106237.3106249

[25] G. Singh, M. Püschel, and M. Vechev, "Fast polyhedra abstract domain," *SIGPLAN Not.*, vol. 52, no. 1, p. 46–59, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3093333.3009885

[26] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," *SIGPLAN Not.*, vol. 37, no. 1, p. 58–70, Jan. 2002. [Online]. Available: https://doi.org/10.1145/565816.503279

[27] A. V. Thakur, "Symbolic abstraction: Algorithms and applications," Ph.D. dissertation, The University of Wisconsin-Madison, 2014.

[28] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 196–207. [Online]. Available: https://doi.org/10.1145/781131.781153

[29] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *IN NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM*, 2000, pp. 3–17.

[30] N. Dor, M. Rodeh, and M. Sagiv, "Cssv: Towards a realistic tool for statically detecting all buffer overflows in c," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 155–167. [Online]. Available: https://doi.org/10.1145/781131.781149

[31] S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta, "Static analysis in disjunctive numerical domains," in *Static Analysis*, K. Yi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 3–17.

[32] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

[33] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[34] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1978, pp. 84–96.

[35] A. Miné, "Tutorial on static inference of numeric invariants by abstract interpretation," *Found. Trends Program. Lang.*, vol. 4, no. 3–4, p. 120–372, Dec. 2017. [Online]. Available: https://doi.org/10.1561/2500000034

[36] S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Program analysis using symbolic ranges," in *Static Analysis*, H. R. Nielson and G. Filé, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 366–383.

[37] F. M. Q. Pereira and D. Berlin, "Wave propagation and deep propagation for pointer analysis," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. USA: IEEE Computer Society, 2009, p. 126–135. [Online]. Available: https://doi.org/10.1109/CGO.2009.9

[38] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 238–252. [Online]. Available: https://doi.org/10.1145/512950.512973

[39] R. E. Rodrigues, V. H. Sperle Campos, and F. M. Quintão Pereira, "A fast and low-overhead technique to secure programs against integer overflows," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–11.

[40] R. E. Moore, *Interval analysis*. Prentice-Hall Englewood Cliffs, 1966, vol. 4.

[41] T. Eiter and H. Mannila, "Computing discrete frechet distance," 05 1994.

[42] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, Oct. 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[43] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

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

[45] 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.

[46] 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

[47] 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

[48] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 474–484.

[49] 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.

[50] V. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, "Hercules: Reproducing crashes in real-world application binaries," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 891–901.

[51] 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

[52] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Partition-based regression verification," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 302–311.

[53] S. Bugrara and D. Engler, "Redundant state detection for dynamic symbolic execution," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 199–211. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bugrara

[54] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2007.41

[55] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 350–360. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180251

[56] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, Oct. 2014. [Online]. Available: https://doi.org/10.1145/2629536

[57] 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

[58] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program learning," *CoRR*, vol. abs/1807.05620, 2018. [Online]. Available: http://arxiv.org/abs/1807.05620

[59] 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

[60] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[61] 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

[62] 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

[63] 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