TOPSEED: Learning Seed Selection Strategies for Symbolic Execution from Scratch

Jaehyeok Lee
Sungkyunkwan University
Republic of Korea
jaehyeok.lee@skku.edu

Sooyoung Cha*
Sungkyunkwan University
Republic of Korea
sooyoung.cha@skku.edu

***Abstract*—We present TOPSEED, a new approach that automatically selects optimal seeds to enhance symbolic execution. Recently, the performance of symbolic execution has significantly improved through various state-of-the-art techniques, including search strategies and state-pruning heuristics. However, these techniques have typically demonstrated their effectiveness without considering "seeding", which efficiently initializes program states for exploration. This paper aims to select valuable seeds from candidate inputs generated during interactions with any symbolic execution technique, without the need for a predefined seed corpus, thereby maximizing the technique's effectiveness. One major challenge is the vast number of candidates, making it difficult to identify promising seeds. To address this, we introduce a customized online learning algorithm that iteratively groups candidate inputs, ranks each group, and selects a seed from the top-ranked group based on data accumulated during symbolic execution. Experimental results on 17 open-source C programs show that TOPSEED significantly enhances four distinct cutting-edge techniques, implemented on top of two symbolic executors, in terms of branch coverage and bug-finding abilities.**

I. INTRODUCTION

Symbolic execution [1], [2], [3], [4] is a representative software testing method designed to automatically generate promising inputs that increase code coverage or trigger bugs in a target program. This method replaces program inputs with symbolic variables and explores distinct paths by executing the program both concretely and symbolically with these variables. More specifically, symbolic execution cycles through three core processes: choosing, executing, and forking a program state from a set of states maintained during its testing period. In particular, if the selected state encounters a branch condition containing the symbolic variables, the forking process checks whether both sides of the condition are satisfiable through the SMT solver [5], [6]. If both sides are satisfiable, it forks the state into two independent states. Through these iterative processes, when the state reaches the endpoint of the program, symbolic execution systematically generates an input by solving the conjunction of the accumulated symbolic branch conditions.

In practice, the forking process in symbolic execution faces two open challenges: state explosion and high constraint-solving costs. The former denotes the exponential increase in the number of states to be maintained during testing as the number of branch conditions in the program grows, complicating the decision of which state to explore first. The latter refers to the cost of checking the feasibilities of both sides of each branch using the SMT solver, which consumes the majority of the total testing budget. Recent advances in symbolic execution have significantly improved its performance by addressing these challenges through various state-of-the-art techniques. These include search strategies [4], [7], [8], [9], [10], [11], state-pruning strategies [12], [13], [14], [15], state-merging strategies [16], [17], and constraint-solving cost reduction strategies [18], [19], among others.

However, these recent techniques have typically demonstrated their effectiveness without considering "seeding". When given a seed, the seeding process initializes program states for exploration without the need for costly feasibility checks on each branch encountered during the regeneration of the seed. Seeding offers an advantageous starting point for state exploration and effectively reduces the cost of checks, thus serving as a complementary approach to various independent symbolic execution techniques. Unfortunately, this complementary factor has often been overlooked in state-of-the-art symbolic execution techniques. Existing techniques have predominantly shown their capabilities assuming no initial seeding [10], [11], [15] or have only demonstrated effectiveness with a predefined seed corpus [19], [20], [21].

In this paper, we present TOPSEED, a novel seeding approach designed to enhance diverse symbolic execution techniques from scratch, thereby maximizing their performance. Our method iteratively selects seeds from inputs generated during symbolic execution on the target program, without relying on a predefined seed corpus. A major challenge is the vast number of candidate seeds—often tens of thousands for real-world programs—which complicates the selection of promising seeds. To overcome this challenge, we have developed a customized online learning algorithm. This algorithm iteratively groups candidate inputs, ranks each group, and selects a promising seed from the top-ranked group. It also continuously updates two probabilistic distributions related to group ranking and seed selection, leveraging the data accumulated during symbolic execution.

Experimental results demonstrate that TOPSEED remarkably enhances three advanced symbolic execution techniques: search strategy [11], state-pruning strategy [15], and

*Corresponding Author.

constraint-solving strategy [18]. For our evaluation, we applied TOPSEED to each technique and evaluated its performance against these techniques operating without seeding across 17 open-source C programs, ranging from 8K to 251K LoC. In terms of branch coverage, the techniques integrated with TOPSEED covered 35.5% more branches on average than the techniques alone. Regarding bug-finding, the integrated techniques identified ten bugs in total, whereas the techniques without seeding detected seven of these bugs. We further demonstrate TOPSEED's efficacy by comparing it with a random seeding approach, and its generality by applying TOPSEED to concolic testing [1], [3], a significant variant of dynamic symbolic execution.

Contributions. We summarize our contributions below:

- We introduce a new synergistic approach that automatically selects promising seeds for enhancing arbitrary symbolic execution techniques further from scratch.
- We present a learning-based algorithm that continuously groups the inputs, prioritizes the groups, and chooses a seed from a high-priority group based on data accumulated while interacting with any symbolic execution technique.
- We demonstrate the effectiveness of TOPSEED by applying it to three symbolic execution techniques and comparing their performance with the original techniques across 17 open-source C programs. We make our tool, TOPSEED*, publicly available.

II. PRELIMINARIES

A. Symbolic Execution

The main aim of symbolic execution is to automatically generate inputs that trigger bugs or increase code coverage for a target program. To do so, symbolic execution iteratively selects, executes, and updates so-called "program states". We define a program state as a triple $(stmt, m, \Phi)$, where '$stmt$' represents the next statement to be executed, 'm' denotes a symbolic memory mapping from program variables to symbolic variables, and 'Φ' indicates a path condition, a sequence of symbolic branch conditions exercised so far in the state.

Algorithm 1 describes a generic symbolic execution algorithm. It takes a target program (pgm), a time budget ($time$), and a seed input ($seed$) as input. For now, let us set aside the last input, $seed$, and the associated function, FEASIBLE; their roles will be explained in Section II-B. At lines 2–3, the algorithm initializes S and I, two sets representing states and inputs, as a singleton set and an empty set, respectively. After the initialization, Algorithm 1 repeats the loop at lines 4–17 until the given time budget ($time$) expires.

At line 5, Algorithm 1 selects a state $(stmt, m, \Phi)$ to be explored further from the set S of states. If the next statement of the selected state, $stmt$, is an if/else statement with a branch condition ϕ, the algorithm checks whether the true and false sides of the branch are feasible via an SMT solver [5], [6] (lines 11–12). Specifically, if the path condition to which the branch condition is added, $(\Phi \wedge \phi)$, is satisfiable, the algorithm appends the new state $(stmt_t, m', \Phi \wedge \phi)$ to the set S; likewise, it adds another state $(stmt_f, m', \Phi \wedge \neg\phi)$ to S when the updated path condition, $(\Phi \wedge \neg\phi)$, is satisfiable. At lines 13–14, when the next statement of the selected state is a halt statement, the algorithm generates an input (v), a model of the path condition (Φ) accumulated in the state, based on the SMT solver. Then, it adds a pair of v and Φ in the set I. For other types of statements (e.g., load), the algorithm properly updates the state s to s' at line 16. For simplicity, the details of how each statement updates the state are omitted. This symbolic execution algorithm continues to repeat the process until the allocated time budget ($time$) is exhausted, at which point it returns the set I of generated inputs.

Algorithm 1 Symbolic Execution with A Seed Input

Input: Program (pgm), budget ($time$), seed input ($seed$).
Output: Inputs (I)

```
1:  procedure SymbolicExecutor(pgm, time, seed)
2:      S ← {(stmt_0, m_0, true)}                    ▷ A set of states
3:      I ← ∅                                        ▷ A set of inputs
4:      repeat
5:          s ← Choose(S)                            ▷ s = (stmt, m, Φ)
6:          S ← S \ {s}
7:
8:          if stmt = if/else statement with a condition φ then
9:              if ((Φ ∧ φ) ∈ Feasible(seed)) then f_t ← 1 else f_t ← 0
10:             if ((Φ ∧ ¬φ) ∈ Feasible(seed)) then f_f ← 1 else f_f ← 0
11:             if (f_t ∨ Sat(Φ ∧ φ)) then  S ← S ∪ {(stmt_t, m', Φ ∧ φ)}
12:             if (f_f ∨ Sat(Φ ∧ ¬φ)) then  S ← S ∪ {(stmt_f, m', Φ ∧ ¬φ)}
13:         else if stmt = halt statement then
14:             I ← I ∪ {(v, Φ)}                     ▷ v = Model(Φ)
15:         else
16:             S ← S ∪ {s'}                         ▷ s' = (stmt', m', Φ)
17:     until time expires (or S = ∅)
18:     return I
```

B. Seeding For Symbolic Execution

A seeding process can skip the feasibility checking of the branches associated with a given seed input, thereby significantly reducing the high checking costs [19], [22], [23] introduced by the SMT solver. Specifically, Algorithm 1 bypasses this checking process through the FEASIBLE function at lines 9–10. We define the FEASIBLE function as one that accepts a seed input and returns all path conditions whose feasibility was confirmed (e.g., SAT(Φ)==true) during seed generation, as follows:

$$\text{FEASIBLE}(seed) = \{\Phi_1, \Phi_2, \cdots, \Phi_n\}$$

Using the FEASIBLE function, if the feasibility of the path condition $(\Phi \wedge \phi)$ has already been checked during the seed's generation, Algorithm 1 assigns true (1) to f_t at line 9. In this case, without invoking the SMT solver (e.g., the SAT function), the state $(stmt_t, m', \Phi \wedge \phi)$ is added to the set S (line 11). Conversely, If not checked, we set f_t to 0 at line 9, indicating the need to check feasibility through the SAT function at line 11. Similarly, if the feasibility of the condition $(\Phi \wedge \neg\phi)$ at line 10 was evaluated during seed generation, the call to

*TOPSEED: https://github.com/skkusal/TopSeed

Algorithm 2 Random Seeding Strategy for Symbolic Execution

Input: Program (pgm), budget ($time$).
Output: Total Inputs ($TotalI$)

```
1: ⟨TotalI, seed⟩ ← ⟨∅, ⟨⟩⟩
2: time_s ← η_time                                   ▷ η_time = 120
3: repeat
4:     I ← SymbolicExecutor(pgm, time_s, seed)
5:     TotalI ← TotalI ∪ I                           ▷ (v, Φ) ∈ I
6:     seed ← RandomSample(TotalI)    ▷ seed ∈ {v|(v, _) ∈ TotalI}
7: until time expires
8: return TotalI
```

TABLE I: The average number of candidate seed inputs on open-source C programs (time budget: 10h)

Benchmark	sqlite-3.33.0	gawk-5.1.0	grep-3.6	diff-3.7	patch-2.7.6
$\lvert TotalI \rvert$	14,398	25,124	14,931	14,384	23,721

SAT is skipped at line 12. This seeding approach effectively reduces the number of calls to the SMT solver by leveraging pre-evaluated feasibility results from seed generation.

Despite the clear benefits of seeding, we observed that existing techniques typically operate the symbolic executor (Algorithm 1) without seed inputs [11], [15], [18]. Alternatively, previous techniques have demonstrated their effectiveness only with a predefined seed corpus [19], [21], [24]. To enable advantageous seeding without prior knowledge, Algorithm 2 outlines a straightforward approach where the symbolic executor and a seed selection strategy interact from scratch. This algorithm requires only a target program (pgm) and a total budget ($time$) as inputs. At line 4, it initially runs the basic symbolic execution algorithm (Algorithm 1) on the program without any seed input (e.g., $seed$=$\langle\rangle$) during a brief period ($time_s$) and generates inputs (I), which serves as candidate seed inputs. Algorithm 2 then repeats the following process: 1) it adds the set I into the total set of inputs, $TotalI$ (line 5), 2) it randomly selects a seed input, $seed$, from $TotalI$ (line 6), and 3) it runs symbolic execution with $seed$ and generates further inputs (line 4). When the allocated total budget is exhausted, the algorithm returns $TotalI$, the total set of generated inputs.

However, when testing real-world programs, Algorithm 2 faces the challenge of deciding which of the generated inputs ($TotalI$) to use as seeds. Table I displays the average number of candidate seed inputs generated by the well-known symbolic executor, KLEE [4], across five open-source C programs over a period of 10 hours. For example, on the `gawk` program, the average number of candidates is 18,512, highlighting the difficulty of intelligently selecting the most promising seed. One alternative is to utilize all generated inputs as seeds for symbolic execution. However, this naive approach proved unsatisfactory. For instance, we observed that applying this method to the `grep` program resulted in branch coverage that was approximately 78.4% lower on average than the coverage achieved by performing symbolic execution without any seeds.

These observations motivated us to develop a novel technique that automatically selects promising seeds from candidate inputs generated while performing an arbitrary symbolic execution technique, even without prior knowledge. In this work, we define a "promising" seed as one that enables the symbolic executor to effectively explore branches not yet covered by initializing the executor's program states in a way that increases the likelihood of reaching unseen code segments.

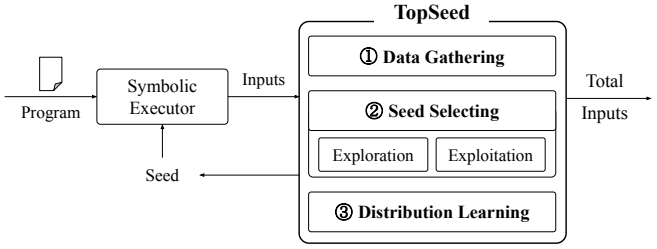

Fig. 1: Overview of TOPSEED

III. OUR APPROACH

In this section, we introduce a novel approach named TOPSEED, designed to automatically learn seed selection strategies based on data accumulated from interactions with a symbolic executor. Initially, we outline the overall process of TOPSEED. We then provide a detailed description of our data-driven seed selection strategies, encapsulated in Algorithm 3, which consist of three stages: Gather, Select, and Learn.

A. Overview

Figure 1 illustrates how TOPSEED automatically selects promising seeds while interacting with a symbolic executor. Initially, TOPSEED generates inputs by performing symbolic execution on a given program without any seeds. It then iteratively selects the most suitable seed from these inputs using the exploration and exploitation strategies and conducts further symbolic execution with the chosen seed. This iterative process enables TOPSEED to effectively utilize various seeds, even without prior knowledge. Hence, our approach divides the total time budget into smaller intervals, repeatedly performing symbolic execution within these periods.

TOPSEED identifies the optimal seed through three phases: Gather, Select, and Learn. In the Gather phase, it constructs useful data, such as branch coverage, from inputs generated during symbolic execution. This data is then utilized in the subsequent phases. The Select stage involves choosing the most promising seed based on two strategies: Explore and Exploit. The Explore strategy selects new seeds from the generated candidates, whereas the Exploit strategy reuses previously selected seeds. In the final Learn stage, TOPSEED refines these strategies by updating the corresponding probability distributions.

B. TOPSEED

We now explain how TOPSEED iteratively selects the most suitable seed based on accumulated data by following Algorithm 3 step by step. Similar to Algorithm 2, our approach (Algorithm 3) takes as input the target program (pgm) and the total time budget ($time$), and returns all inputs ($TotalI$)

Algorithm 3 Our Seeding Strategy for Symbolic Execution

Input: Program (pgm), budget ($time$).
Output: Total Inputs ($TotalI$)

```
1: ⟨D, seed, w, p⟩ ← ⟨∅, ⟨⟩, ⟨⟩, ⟨⟩⟩
2: time_s ← η_time                                   ▷ η_time =120
3: ⟨P_w, P_p⟩ ← ⟨U[−1, 1]^5, U(P)⟩
4: repeat
5:     for i = 1 to η_Learn do                       ▷ η_Learn = 20
6:         stgy ← sample from {Explore, Exploit} with prob = [η_r, 1−η_r]
7:
8:         /* Data Gathering */
9:         I ← SymbolicExecutor(pgm, time_s, seed)   ▷ inp ∈ I
10:        D' ← Gather(I)                            ▷ (inp, B, bug, seed, w, p) ∈ D'
11:        D ← D ∪ D'
12:        w, p ← Sample(P_w, P_p)
13:
14:        /* Seed Selecting */
15:        if stgy = Explore then
16:            {G_1, G_2, ··· , G_n} ← Group(D)      ▷ G_i ⊆ D
17:            G_top ← Rank(AllG, w)                 ▷ AllG = {G_1, G_2, ··· , G_n}
18:            seed ← Select(G_top, p)
19:
20:        if stgy = Exploit then
21:            S_good, S_bad ← Cluster(D)
22:            seed ← Select(S_good, p)
23:
24:    /* Distribution Learning */
25:    ⟨P_w, P_p⟩ ← Learn(D)
26: until time expires
27: return TotalI                                    ▷ TotalI = {inp|(inp, _, _, _, _, _) ∈ D}
```

generated within the budget. Algorithm 3 initializes four components, each as either an empty set or an empty vector: a set D of data, a seed $seed$, a weight vector w, and a policy p (line 1). The roles of w and p will be further discussed in Section III-D. It also sets the small time budget, η_{time}, for conducting multiple iterations of symbolic execution (Algorithm 1) on the program (line 2). This arrangement provides our approach with sufficient opportunities to select useful seeds. For our experiments, we set η_{time} to 120 seconds based on trial and error. At line 3, the algorithm sets two probability distributions, $\mathcal{P}_w$ and $\mathcal{P}_p$, to uniform distributions. In our work, these probability distributions are crucial for selecting promising seeds and are gradually refined based on the accumulated data. After the initialization, the algorithm enters the main loop, iteratively executing the three phases—Gather, Select, and Learn—until the total budget is exhausted (lines 4–26).

Upon entering the loop, Algorithm 3 samples one of the two seed selection strategies—Explore and Exploit—based on predefined sampling probabilities (line 6). In our experiments, we set the probabilities for Explore and Exploit to 75% and 25%, respectively, with a hyperparameter value of η_r at 0.75 (line 6). This distribution is based on our observation that exploring new seeds typically uncovers unseen branches more often than reusing seeds.

C. Data Gathering

The aim of the Gather stage is to obtain informative data, which will be used to refine and apply our seed selection strategies in subsequent stages. To achieve this, we run symbolic execution (Algorithm 1) with a specified seed, generating the set I of inputs at line 9. We then collect data d from each element inp of I, constructing the set D' as follows:

$$(inp, B, bug, seed, w, p) \in D'$$

Here, each element d in D' consists of a 6-tuple: an input inp, a set B of branches that inp covers, a boolean value bug indicating whether inp triggers a bug, the seed used to generate inp, and two additional values, w and p, used in seed selection. At line 11, the set D' is added to the total set D. Before proceeding to the next stage, the Sample function determines two values—a weight vector w and a policy p—from the distributions, $\mathcal{P}_w$ and $\mathcal{P}_p$, respectively (line 12).

D. Seed Selecting

In the Select stage, Algorithm 3 selects a seed using either the Explore or Exploit strategies. At a high-level, the Explore strategy aims to select a new input that has never been used as a seed from among all inputs generated through multiple symbolic execution runs (Algorithm 1). Conversely, the goal of the Exploit strategy is to reuse a seed that was previously selected by the Explore strategy.

Exploration Strategy. To efficiently select a new seed, the Explore strategy in Algorithm 3 initially groups all inputs accumulated during symbolic execution based on branch coverage (line 16). It then ranks these groups according to the weight vector w (line 17), which the Sample function determines at line 12. Finally, this strategy selects the most promising input from the top-ranked group as the seed, using the sampled policy p (line 18).

More specifically, we partition the set D into subsets, $\{G_1, G_2, \cdots, G_n\}$, where each subset contains inputs that cover the same set of branches (line 16). Formally, the i-th subset G_i is defined as:

$$G_i = \{(inp, B, bug, seed, w, p) \in D \mid (B_i = B)\}$$

where B_i denotes the set of covered branches of i-th element in D. Grouping inputs by the set of covered branches simplifies the seed selection process by reducing the challenge from selecting a single seed from all inputs to choosing one group from all groups. This approach is based on the intuition that inputs with identical branch coverage may possess similar potential as seeds.

We now explain how we score each group with the sampled weight vector w (line 17). First, we represent each group by a feature vector. In this work, we have crafted five features that describe the atomic yet useful properties of input groups. Each feature, π_i, takes a group G as input and returns a real value, calculated as follows:

$$\pi_i : \mathbb{G} \to \mathbb{R}$$

where $\mathbb{G}$ represents the set of all possible groups, which correspond to all possible subsets of the set D. A group G is

TABLE II: Five features for group selection

Feature	Description
#1 coverage	The number of branches covered by a group.
#2 frequency	Sum of the inverse frequencies of branches covered by a group.
#3 uniqueness	The number of branches exclusively covered by a group.
#4 bugs	The number of found bugs in a group.
#5 size	The size of a group.

thus represented by a 5-dimensional real-number vector as:

$$\pi(G) = \langle \pi_1(G), \pi_2(G), \pi_3(G), \pi_4(G), \pi_5(G) \rangle$$

We then calculate the score of the group using the weight vector w sampled at 12. In our work, the vector w corresponds to a 5-dimensional real vector. We compute the score of the group by calculating the dot product of the feature vector and w, expressed as:

$$\textsc{Score}(G, w) = \pi(G) \cdot w$$

Finally, we select G_{top}, the group having the highest score, from all candidate groups, $AllG$ (line 17):

$$G_{top} = \underset{G \in AllG}{\operatorname{argmax}} \textsc{Score}(G, w)$$

In this work, we developed five features to effectively characterize input groups generated through symbolic execution, focusing on their branch coverage and bug-finding capabilities, as detailed in Table II. These features aim to capture simplistic yet meaningful attributes of each group without incurring burdensome feature extraction costs.

1) **coverage** (π_1): Calculates the number of branches covered by inputs in a group G as:

$$\pi_1(G) = |B|, \textbf{where } (\_, B, \_, \_, \_, \_) \in G$$

2) **frequency** (π_2): Assigns high values to groups covering branches that are infrequently covered. We calculate branch frequency using the Freq function as:

$$\mathsf{Freq}(b, D) = |\{inp \mid (b \in B) \wedge (inp, B, \_, \_, \_, \_) \in D\}|$$

This function takes as input a branch b and the accumulated data D and returns the number of inputs covering b in D. Formally, $\pi_2(G)$ is defined as:

$$\pi_2(G) = \sum_{b \in B} \frac{1}{\mathsf{Freq}(b, D)}, \textbf{where } (\_, B, \_, \_, \_, \_) \in G$$

3) **uniqueness** (π_3): Measures the uniqueness of branches covered exclusively by group G compared to those covered by groups already selected for seed selection. For instance, if SB denotes the set of branches covered by previously selected groups, π_3 is computed as:

$$\pi_3(G) = |B \setminus SB|, \textbf{where } (\_, B, \_, \_, \_, \_) \in G$$

4) **bugs** (π_4): Counts the total number of bugs found in the group G as:

$$\pi_4(G) = |\{\!\{bug \mid (\_, \_, bug, \_, \_, \_) \in G\}\!\}|$$

where the $\{\!\{\}\!\}$ indicates a multiset that allows duplicated elements, enabling the counting of found bugs.

5) **size** (π_5): Returns the group size, i.e., $|G|$.

These features are designed to provide a comprehensive assessment of each input group's potential efficacy when used as a seed for the symbolic executor.

TABLE III: Four policies for seed selection

Policy	Description
#1 unique	Selecting the seed having the unique path-condition.
#2 long	Selecting the seed having the longest path-condition.
#3 short	Selecting the seed having the shortest path-condition.
#4 random	Selecting the seed randomly.

Finally, the Select function in Algorithm 3 selects the most promising input from the top-ranked group G_{top} as a seed (line 18). To faciliate this selection, we have devised four straightforward policies, as defined in Table III. The underlying rationale for these policies is that all inputs in the group G_{top} cover the same set of branches and are thus considered to have comparable potential as seeds. Our goal is to identify the most effective seed within G_{top} by analyzing simple properties of each input inp (e.g., $(inp,\_,\_,\_,\_,\_) \in G_{top}$). We particularly assess the characteristics of the path-condition Φ used to generate each input inp, defined as the pair (v, Φ). Our evaluation primarily considers atomic factors like the length of Φ (e.g., $|\Phi|$). Using the sampled policy p, one of the four policies outlined in Table III, we choose the seed most likely to optimize the performance of the symbolic executor.

Exploitation Strategy. The Exploit strategy in Algorithm 3 is predicated on the observation that certain previously used seeds remain reusable and can enhance the performance of symbolic execution. Initially, this strategy categorizes seeds into good and bad clusters based on their actual effectiveness (line 21). It then selects a reusable seed from the good cluster, guided by the policy p (line 22).

To organize the seeds into two clusters, we evaluate each seed's effectiveness. We begin by calculating B_{seed}, which represents the set of branches covered by inputs generated during a symbolic execution run with each seed, defined as:

$$B_{seed} = \{b \in B \mid (inp, B, \_, seed, \_, \_) \in D\}$$

Given k previously used seeds, we compute B_{seed_i} for each seed (where $1 \leq i \leq k$) and aggregate these into set TB:

$$TB = \{B_{seed_1}, B_{seed_2}, \cdots, B_{seed_k}\}$$

We then compute the frequency of each branch b in B_{seed} across all seeds using the Freq' function:

$$\mathsf{Freq}'(b, TB) = |\{B_{seed} \mid (b \in B_{seed}) \wedge (B_{seed} \in TB)\}| \quad (1)$$

This frequency indicates how many seeds cover branch b. Each seed's effectiveness is scored based on the sum of inverse

frequencies of branches in B_{seed}:

$$\textsc{Eval}(seed, TB) = \sum_{b \in B_{seed}} \frac{1}{\mathsf{Freq}'(b, TB)} \quad (2)$$

This scoring quantifies a seed's ability to cover branches and the rarity with which these branches are typically reached.

Next, we divide the k seeds into good and bad clusters. Assuming the set SD contains pairs of each seed and its score:

$$SD = \{(seed_1, score_1), (seed_2, score_2), \cdots, (seed_k, score_k)\}$$

We apply the k-means clustering algorithm [25] (where k to 2) to categorize SD into good and bad clusters, S_{good} and S_{bad}, based on their scores.

Finally, the Exploit strategy selects a seed to reuse from S_{good} (line 22) by using the Select function, identical to that used in the Explore strategy. This selection is made based on a policy sampled from the four policies in Table III.

E. Distribution Learning

In the Learn phase, TOPSEED (Algorithm 3) updates two probability distributions, $\mathcal{P}_w$ and $\mathcal{P}_p$, utilizing the accumulated data D (line 25). This update is crucial as our seed selection strategies depend on the weight vector w and the policy p derived from $\mathcal{P}_w$ and $\mathcal{P}_p$, respectively. The Learn function evaluates the performance of seeds selected by these two parameters, refining $\mathcal{P}_w$ and $\mathcal{P}_p$ to enhance their effectiveness. In short, the intuition behind these updates is to refine the probability distributions so that relatively better weight vectors and policies can be sampled more frequently based on the currently accumulated data D.

To first assess the effectiveness of the weight vectors, we compute the set B_w of branches covered using each weight vector w from data set D:

$$B_w = \{b \in B \mid (\_, B, \_, \_, w, \_) \in D\}$$

Assuming WB contains B_w for each of the n used weight vectors (e.g., W=$\{w_1, w_2, \cdots, w_n\}$):

$$WB = \{B_{w_1}, B_{w_2}, \cdots, B_{w_n}\}$$

We evaluate each vector w using the equations (1) and (2):

$$\textsc{Eval}(w, WB) = \sum_{b \in B_w} \frac{1}{\mathsf{Freq}'(b, WB)} \quad (3)$$

This evaluation score reflects a weight vector's effectiveness based on the number and uniqueness of the branches it covers. The set of weight vectors, W, is then classified into good and bad groups, $TopW$ and $BotW$, using the k-means clustering algorithm, where k is 2.

Finally, we update the probability distribution $\mathcal{P}_w^i$ for the weight value of the i-th feature (π_i) in Table II, based on the sets $TopW$ and $BotW$. If $TopW^i$ represents the set of good weight values for π_i, we update $\mathcal{P}_w^i$ as a truncated normal distribution centered around the mean value $\mu(TopW^i)$ with a standard deviation value $\sigma(TopW^i)$, and constrained it to the interval [-1, 1]. This interval is the same as the range initialized in line 3 of Algorithm 3. Exceptionally, if the distributions derived from $TopW^i$ and $BotW^i$ are similar, values are sampled from a uniform distribution between -1 and 1, due to the instability of the distribution from $TopW^i$.

Updating the probability distribution $\mathcal{P}_p$ for sampling four policies in Table III follows a similar principle to updating the sampling probabilities $\mathcal{P}_w$ for weight vectors. The goal is to increase the sampling probability of policies that have effectively increased branch coverage. We begin by calculating the set B_p of branches covered using each p of the four policies from dataset D, defined as $B_p = \{b \in B \mid (\_, B, \_, \_, \_, p) \in D\}$. From these calculations, we derive the set PB containing 4 elements: $\{B_{p_1}, B_{p_2}, B_{p_3}, B_{p_4}\}$. Each policy p is then evaluated using the EVAL function (the equation (3)), where the effectiveness of p is calculated as $\textsc{Eval}(p, PB)$. Finally, we update the probability $\mathcal{P}_p^i$ for sampling i-th policy as:

$$\mathcal{P}_p^i = \frac{\textsc{Eval}(p_i, PB)}{\sum_{1 \le i \le 4} \textsc{Eval}(p_i, PB)}$$

This update aims to increase the sampling probability of policies that have effectively improved branch coverage.

After updating the two probability distributions, $\mathcal{P}_w$ and $\mathcal{P}_p$, (line 25), TOPSEED (Algorithm 3) samples the weight vector (w) and the policy (p) from these updated distributions (line 12). In this manner, without prior knowledge, TOPSEED intelligently selects promising seeds based on $\mathcal{P}_w$ and $\mathcal{P}_p$, which are continuously updated based on the accumulated data D during symbolic execution.

TABLE IV: 17 benchmark programs

Benchmark	LoC	# of Branches	Benchmark	LoC	# of Branches
sqlite-3.33.0	251K	29,260	combine-0.4.0	8K	2,359
gawk-5.1.0	70K	18,320	pr-8.31	13K	1,977
grep-3.6	35K	8,270	ln-8.31	13K	1,564
diff-3.7	33K	7,683	dd-8.31	11K	1,529
patch-2.7.6	27K	6,221	factor-8.31	13K	1,502
expr-8.31	21K	4,545	od-8.31	9K	1,135
csplit-8.31	19K	4,489	tr-8.31	8K	1,063
ginstall-8.31	22K	3,501	[-8.31	8K	1,032
trueprint-5.4	9K	2,518			

IV. EXPERIMENTS

In this section, we evaluate the effectiveness of TOPSEED based on the following research questions:

1) **Effectiveness:** How effectively can TOPSEED enhance state-of-the-art techniques for symbolic execution in terms of branch coverage and bug-finding capability?
2) **Efficacy of Each Stage:** How effectively does each stage in TOPSEED contribute to improving the performance?
3) **Generality:** Is TOPSEED applicable to a concolic testing technique, a variant of dynamic symbolic execution?
4) **Impact of Hyperparameters:** How do hyperparameter values affect the performance of TOPSEED?

We implemented TOPSEED on top of KLEE-2.1 [4], an actively maintained symbolic executor that incorporates diverse techniques [11], [15], [17], [18], [26]. All experiments were

TABLE V: Average branch coverage achieved by four baselines under three different settings across 17 benchmarks.

Benchmark	KLEE [4] +			LEARCH [11] +			HOMI [15] +			KLEE-ARRAY [18] +		
	BASE	RANDSEED	TOPSEED	BASE	RANDSEED	TOPSEED	BASE	RANDSEED	TOPSEED	BASE	RANDSEED	TOPSEED
sqlite-3.33.0	4,281	4,053	**6,448**	5,112	5,251	**5,944**	5,483	5,143	**6,123**	4,905	5,017	**6,417**
gawk-5.1.0	3,279	3,352	**4,054**	2,933	3,086	**3,226**	1,952	3,778	**4,608**	3,129	3,474	**4,379**
grep-3.6	2,154	2,567	**3,104**	1,168	2,075	**3,294**	1,936	2,601	**3,002**	1,740	2,546	**3,333**
diff-3.7	985	922	**1,653**	529	968	**1,713**	1,017	1,044	**1,941**	788	940	**1,795**
patch-2.7.6	800	825	**1,114**	769	849	**1,247**	1,027	1,112	**1,193**	741	790	**1,091**
expr-8.31	1,282	1,174	**1,306**	815	935	**1,252**	1,255	1,299	**1,346**	1,283	1,232	**1,311**
csplit-8.31	750	741	**781**	661	704	**774**	711	711	**731**	742	753	**779**
ginstall-8.31	788	840	**901**	778	678	**855**	978	1,012	**1,104**	440	651	**717**
trueprint-5.4	611	562	**814**	518	517	**537**	728	932	**1,227**	633	551	**974**
combine-0.4.0	485	506	**696**	444	719	**790**	889	914	**939**	444	526	**731**
pr-8.31	749	817	**1,049**	812	776	**918**	1,170	1,165	**1,200**	723	782	**1,086**
ln-8.31	482	439	**653**	539	413	**545**	807	768	**821**	609	457	**682**
dd-8.31	494	505	**606**	375	138	**516**	553	548	**582**	451	495	**619**
factor-8.31	264	264	**264**	261	263	**264**	264	264	**264**	264	264	**264**
od-8.31	626	667	**727**	661	562	**723**	682	689	**698**	702	669	**726**
tr-8.31	457	427	**477**	297	395	**478**	450	474	**482**	464	425	**484**
[-8.31	249	263	**266**	236	264	**266**	249	263	**266**	251	258	**258**
Total	18,736	18,924	**24,913**	16,908	18,593	**23,342**	20,151	22,717	**26,527**	18,309	19,830	**25,646**

conducted on a single Linux machine equipped with two Intel Xeon Gold 6230R processors.

A. Experimental Settings

Baselines. We compared the performance of four baselines integrated with TOPSEED against their performance without TOPSEED integration. The four baselines are: (1) base symbolic executor (KLEE [4]), (2) search strategy (LEARCH [11]), (3) state-pruning strategy (HOMI [15]), and (4) constraint-solving strategy (KLEE-ARRAY [18]). Notably, the last three techniques are considered state-of-the-art, each enhancing symbolic execution with distinct methodologies. LEARCH [11] is a novel learning-based search strategy that prioritizes states for exploration. HOMI [15] is a cutting-edge state-pruning strategy that automatically eliminates redundant states while keeping promising ones. KLEE-ARRAY [18] is an effective constraint-solving strategy designed to accelerate the process of computationally expensive constraints involving arrays.

Specifically, we evaluated the performance of TOPSEED by running each baseline with three different settings: the baseline without seeding (BASE), the baseline with a random seed selection strategy (RANDSEED), corresponding to Algorithm 2, and the baseline integrated with our approach (TOPSEED), corresponding to Algorithm 3.

Benchmarks. We used 17 open-source C programs in Table IV, which displays the number of lines and branches for each program. To construct our benchmark suite, we initially collected all available programs used in the experiments by the four baselines [4], [11], [15], [18]. From these, we selected 17 program benchmarks for evaluation based on two filtering criteria among them: (1) we filtered out smaller programs with fewer than 1,000 total branches, as the baselines achieved high code coverage for these programs even without seeding (e.g., `tty-8.31`). (2) we also omitted programs (e.g., `objcopy-2.36`) where all baselines managed to cover less than 2.0% of total branches over 10 hours, indicating that the baselines failed to test the core functionality of these programs and only test error handing functions. Note that for our 17 programs, at least one of the four baselines was able to cover more than 22.5% of total branches on average within 10 hours.

Other Settings. For all experiments, we allocated a total testing budget of 10 hours per benchmark program. Each experiment was conducted five times, and the average results were reported. Additionally, to ensure a fair comparison, we ran each baseline without seeding (BASE) in two ways: (1) by running it once for the total testing budget (10 hours) without termination, or (2) by running it multiple times within a shorter budget (120 seconds), similar to TOPSEED (Algorithm 3). The better result between the two approaches was reported as BASE.

B. Effectiveness of TOPSEED

Branch Coverage. Table V shows that integrating TOPSEED into each of the four baselines significantly increased branch coverage across all benchmarks, compared to the baselines without TOPSEED. Overall, TOPSEED covered 33.0% more branches than KLEE+BASE, the base symbolic executor without seeding, across all benchmarks. When compared with three state-of-the-art techniques, TOPSEED substantially enhanced their average branch coverage: integrating TOPSEED with LEARCH, HOMI, and KLEE-ARRAY, resulted in 38.1%, 31.6%, 40.1% increases in branch coverage, respectively, over each baseline without seeding. We obtained these results using gcov [27], a popular tool for calculating coverage.

Particularly, TOPSEED dramatically increased branch coverage achieved by four baselines on large benchmarks, which have a greater potential for improvement than smaller ones. For the five largest programs, integrating TOPSEED with LEARCH, HOMI, and KLEE-ARRAY, increased branch coverage by 46.7%, 47.8%, 50.5%, respectively, compared to each baseline (BASE). Conversely, on the five smallest programs, the improvements were modest, with TOPSEED integration

TABLE VI: Comparison of bug-finding performance: baselines with and without TOPSEED

	Benchmark	BASE	RANDSEED	TOPSEED
KLEE	patch-2.7.6	0	**1**	**1**
	combine-0.4.0	1	1	1
LEARCH	patch-2.7.6	2	2	2
	combine-0.4.0	1	1	1
HOMI	gawk-5.1.0	1	1	1
	patch-2.7.6	0	0	**1**
	combine-0.4.0	2	2	**3**
Total		7	8	**10**

resulting in coverage increases of 22.8%, 4.3% and 10.3% for LEARCH, HOMI, and KLEE-ARRAY, respectively, highlighting smaller gains compared to larger programs.

Running each baseline with a random seed selection strategy (RANDSEED) typically resulted in higher branch coverage compared to running it without seeds (BASE). However, RANDSEED sometimes underperformed relative to BASE. For instance, on the largest program, `sqlite`, KLEE+RANDSEED and HOMI+RANDSEED covered 5.3% and 6.2% fewer branches on average than KLEE+BASE and HOMI+BASE, respectively. Surprisingly, for the `dd` program, LEARCH+RANDSEED covered 63.2% fewer branches than LEARCH+BASE. This inconsistency of RANDSEED underscores the necessity of our approach (Algorithm 3) to achieve stable and high coverage across various benchmark programs.

These results in Table V were statistically significant: on 16 of the benchmark programs, the p value between BASE and TOPSEED across the four baselines was less than 0.05 according to the Wilcoxon signed-rank test. An exception occurs in the `factor` program, where the gap in branch coverage between TOPSEED and BASE across four baselines was minimal. The standard deviations of branch coverage across four baselines on our benchmarks were as follows: BASE (38.6), RANDSEED (73.1), and TOPSEED (52.0). As expected, RANDSEED showed the highest variability. While the standard deviation of TOPSEED was slightly higher than BASE, this difference was negligible given the significant performance improvements achieved by TOPSEED.

Bug-Finding Capability. Enabling seeding for each baseline via TOPSEED improved its bug-finding capabilities. Table VI demonstrates that TOPSEED successfully identified ten bugs in total across three baselines and benchmarks: `patch`, `combine`, and `gawk`. To gather these bug-finding results, we first generated inputs by running each baseline in three modes—BASE, RANDSEED, and TOPSEED—on all benchmark programs, respectively. We then selected inputs flagged as bugs by each baseline. Finally, we verified and extracted reproducible bug-triggering inputs by executing the original binary of each program with these selected inputs. We reported all the bugs discovered by TOPSEED, and among them, the bug in `gawk` was confirmed by the original developers.

Table VI also highlights that integrating TOPSEED with the state-pruning strategy (HOMI) led to the discovery of five unique bugs across the three benchmarks, whereas the original state-pruning strategy (HOMI+BASE) found only three bugs among them. Note that two bugs identified by TOPSEED were not only missed by HOMI+BASE but also by the random seed selection strategy (HOMI+RANDSEED). This indicates that randomly choosing seeds for symbolic execution is insufficient for improving bug-finding capabilities.

TABLE VII: Efficacy evaluation of each stage in TOPSEED

Benchmark	TOPSEED - (Group, Learn)	TOPSEED - (Group)	TOPSEED - (Learn)	TOPSEED
sqlite-3.33.0	5,673	5,703	6,100	**6,448**
gawk-5.1.0	3,819	3,850	3,890	**4,054**
grep-3.6	2,793	2,821	2,922	**3,104**
diff-3.7	1,161	1,314	1,609	**1,653**
patch-2.7.6	924	1,028	955	**1,114**
Total	14,370	14,716	15,476	**16,373**

C. Efficacy of Each Stage

We evaluated the effectiveness of the Select and Learn stages in TOPSEED (Algorithm 3). Specifically, for the Select phase, we assessed TOPSEED's performance without the grouping and clustering algorithms used in the Explore and Exploit strategies. In this variant, dubbed 'TOPSEED without grouping', each input generated during symbolic execution is treated as a separate group. This variant negates the need to choose a seed from top-ranked groups since each group consists of only one input. In the Learn phase, we examined how TOPSEED performs without updating the probabilistic distributions $\mathcal{P}_w$ and $\mathcal{P}_p$, which are used to sample the values w and p. In this variant, 'TOPSEED without learning', both $\mathcal{P}_w$ and $\mathcal{P}_p$ are set to uniform distributions. On the five largest programs, we calculated the average branch coverage by using three variants: 'TOPSEED without both grouping and learning', 'TOPSEED without grouping', and 'TOPSEED without learning'.

Results in Table VII confirm the critical roles of both Select and Learn stages for enhancing branch coverage. TOPSEED with full functionality covered 13.9% more branches on average across the five benchmarks than its variant without grouping and learning. Coverage improvements from each stage varied across benchmarks; for instance, the `diff` program benefited significantly from grouping, while learning algorithms notably boosted performance for the `patch` program. Although both of TOPSEED's primary algorithms—seed grouping and distribution learning—are essential, the grouping algorithm has a more pronounced impact. As shown in Table VII, the variant without grouping achieved 11.3% lower branch coverage, whereas the variant without learning achieved about 5.8% lower branch coverage, emphasizing the greater importance of grouping.

Important Features. We also analyzed the importance of features by observing how their weights change over 10 hours. Each weight value associated with the five features in Table II is utilized for ranking input groups in the Explore

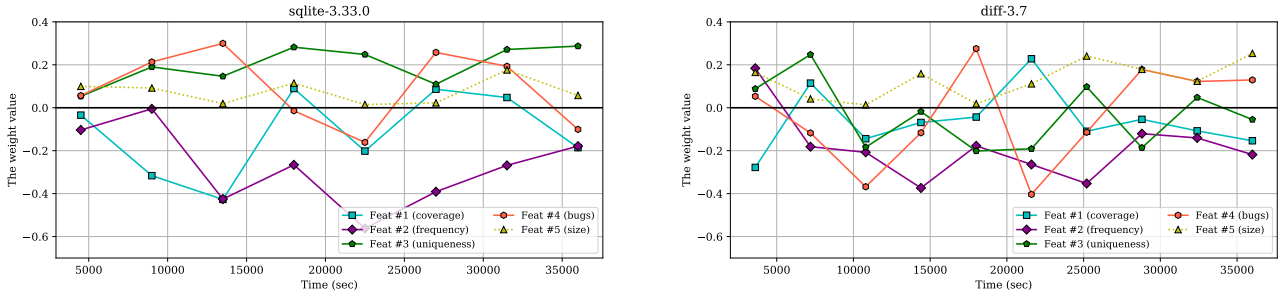

Fig. 2: Changes in average weight value corresponding to each of the five features over 10 hours

TABLE VIII: The number of covered branches and found bugs by concolic testing techniques with and without TOPSEED

Benchmark	CREST[28]	CREST[28] + TOPSEED	CHAMELEON[32]	CHAMELEON[32] + TOPSEED
gawk-3.0.3	3,198(1)	**3,230(1)**	3,380(2)	**3,544(2)**
grep-2.2	1,978(1)	**2,017(1)**	2,213(1)	**2,258(1)**
sed-1.17	1,125(1)	**1,524(2)**	1,504(1)	**1,573(2)**
Total	6,282(3)	**6,771(4)**	7,097(4)	**7,375(5)**

strategy. If a i-th feature's weight value is negative, it suggests that the strategy is less likely to select groups containing this feature; conversely, a positive value indicates a higher likelihood of selection. Thus, both negative and positive weight values are crucial for TOPSEED as they directly influence the selection or non-selection of input groups. Figure 2 illustrates the changes in each feature's weight value over 10 hours for two benchmark programs—`sqlite` and `diff`—which show the highest coverage improvements in our benchmark suite.

The results in Figure 2 demonstrate that the significance of features varies across the target program. For `sqlite`, the weight of the second feature, 'frequency', remained negative throughout the 10 hours, indicating a disinclination to select groups with this feature. Conversely, the weight of the third feature, 'uniqueness', was consistently positive, effectively enhancing branch coverage. However, this trend did not hold for the `diff` program; the weight for 'uniqueness' fluctuated over time. These observations suggest that our learning approach adaptively refines the probabilistic distributions corresponding to each feature to maximize coverage.

D. Generality of TOPSEED

We examined if our seeding approach is applicable to concolic testing [1], [3], a major variant of symbolic execution. We integrated TOPSEED with CREST [28], a popular concolic testing tool that incorporates diverse techniques [29], [30], [31], [32]. Additionally, we evaluated TOPSEED's potential to enhance the performance of CHAMELEON [32], a state-of-the-art search strategy implemented on CREST. For each baseline, we reported the number of branches covered and bugs identified across the three large benchmarks used in [32]. In this evaluation, we use default values of hyperparameters as mentioned in Section IV-E.

TABLE IX: Evaluation of performance with different hyperparmeter settings

Parameters	η_{time}			η_{Learn}			η_r		
Benchmark	60	120	240	1	20	40	0.5	0.75	1.0
sqlite-3.33.0	5,748	**6,448**	5,898	6,159	**6,448**	5,970	5,832	**6,448**	6,200
gawk-5.1.0	3,888	**4,054**	3,905	3,916	**4,054**	3,986	3,897	**4,054**	3,896
grep-3.6	2,757	**3,104**	2,806	2,946	**3,104**	2,974	3,014	3,104	**3,108**
diff-3.7	1,469	**1,653**	1,611	1,607	1,653	**1,674**	1,583	**1,653**	1,602
patch-2.7.6	918	**1,114**	978	1,011	**1,114**	928	991	**1,114**	954
Total	14,780	**16,373**	15,198	15,639	**16,373**	15,532	15,317	**16,373**	15,760

Table VIII demonstrates that TOPSEED successfully improved the effectiveness of concolic testing in terms of code coverage and bug-finding abilities. On average, TOPSEED increased branch coverage by 7.8% compared to CREST and 3.9% compared to CHAMELEON. Interestingly, with the seeding provided by TOPSEED, the basic concolic testing tool CREST occasionally outperformed the advanced, unseeded CHAMELEON on the `sed` program. In terms of bug-finding, CHAMELEON+TOPSEED identified five unique bugs: two in `gawk`, one in `grep`, and two in `sed`; in contrast, CHAMELEON alone found four of these. These results indicate that TOPSEED is versatile and can be generalized to other symbolic execution tools, significantly enhancing both branch coverage and bug-finding abilities.

E. Impact of Hyperparameters

We assessed the impact of varying three hyperparameters—η_{time}, η_{Learn}, and η_r—on the effectiveness of TOPSEED. Our approach (Algorithm 3) sets these default values to 120, 20, and 0.75, respectively. The first parameter, η_{time}, dictates the small time budget allocated for each run of symbolic execution. The second parameter, η_{Learn}, defines the frequency at which the two probabilistic distributions are updated. The third parameter, η_r, controls the sampling ratio between two seed selection strategies: Explore and Exploit. On the five largest benchmarks, we calculated the branch coverage achieved by KLEE+TOPSEED using various values around the default settings we established for each hyperparameter.

Table IX demonstrates that the default hyperparameter values generally achieved the most stable and high branch coverage on the five benchmarks. Specifically, the η_{time} value of 120 seconds consistently provided the highest branch coverage across all the benchmarks compared to shorter (60 seconds) or

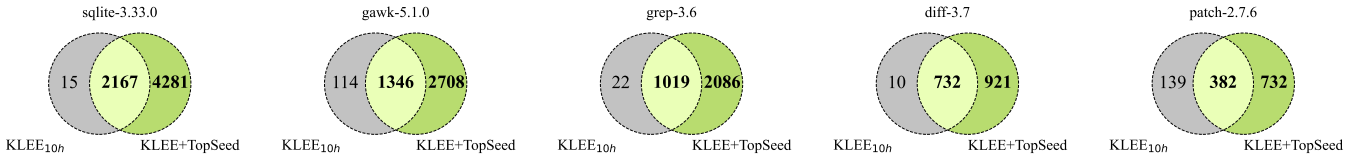

Fig. 3: Venn-diagrams depicting the sets of branches covered by KLEE_{10h} and KLEE+TopSeed

```
1  static tree *parse_ex(token *token) {
2      bin_tree_t *tree;
3      switch(token->type) {
4          case CHARACTER:
5              tree = create_token_tree(token);
6          case OP_OPEN_SUBEXP:
7          case OP_OPEN_BRACKET:
8          /* Total 19 case statements and 12 functions */
9      }
10 }
```

Fig. 4: A simplified code snippet from `diff-3.7`

longer periods (240 seconds). Notably, a η_{time} of 60 seconds led to a 10.9% drop in branch coverage compared to the default, indicating that an excessively short time budget for symbolic execution (Algorithm 1) can degrade performance.

Furthermore, the η_r value of 0.75 resulted in the most stable branch coverage compared to lower (0.5) or higher (1.0) settings. When η_r was set to 1.0, indicating exclusive use of the Explore strategy for seed selection, branch coverage was approximately 3.7% lower than with the default η_r of 0.75. This underscores the importance of the Exploit strategy, which reuses previously selected seeds, demonstrating that a combination of Explore and Exploit strategies is crucial for optimal performance.

F. Discussions

Comparison with Running KLEE Uninterrupted for 10 Hours. TopSeed divides the total time budget into smaller intervals, performing multiple symbolic executions within each to leverage various seeds from scratch. We assessed how effectively TopSeed covers branches compared to performing symbolic execution (KLEE) uninterrupted for 10 hours. Specifically, on the five largest programs listed in Table IV, we analyzed the set of branches covered by both KLEE+TopSeed and continuous KLEE (KLEE_{10h}).

Figure 3 displays Venn diagrams comparing branches uniquely covered by KLEE+TopSeed and KLEE_{10h}. Results indicate that TopSeed covers, on average, 95% of the branches reached during continuous 10-hour symbolic executions (KLEE_{10h}) across the benchmarks. Surprisingly, KLEE_{10h} failed to cover 65.5% of the total branches exercised by TopSeed. Notably, for the largest program, `sqlite`, KLEE+TopSeed missed only 15 branches covered that KLEE_{10h} covered, yet it exclusively covered 4,281 branches that KLEE_{10h} failed to reach. These findings suggest that the branches uniquely reachable by prolonged symbolic execution without TopSeed are minimal, demonstrating that TopSeed's iterative runs can maximize branch coverage.

TABLE X: The number of SMT calls skipped for regenerating the seeds selected by TopSeed

Benchmark	# of Skipped Calls
sqlite-3.33.0	10,758
gawk-5.1.0	14,183
grep-3.6	12,588
diff-3.7	18,188
patch-2.7.6	9,150
Average	12,973

Characteristics of Good Seeds. To explain how TopSeed achieves superior branch coverage, we conducted a case study on the `diff` program, which showed the highest improvement in terms of branch coverage. TopSeed covered, on average, 114.0% more branches than each of the four baselines without seeding (Base). We collected all the selected seeds by TopSeed, ranked them based on their branch coverage, and analyzed the characteristics of the top 10% ranked seeds.

Our key observation is that the top-ranked seeds typically target switch-case statements with numerous cases, allowing them to trigger multiple unique functions within those cases. Figure 4 shows a simplified code snippet from the 'parse_ex' function in `diff-3.7`. This function contains a switch-case statement with 19 cases and 12 distinct functions under those cases. TopSeed executed 7 out of the 19 total cases on average, compared to just one case executed by the baseline without seeding. This case study demonstrates that effective seeds tend to enhance the efficiency of symbolic execution by initializing program states in a manner that increases the likelihood of reaching unseen code.

The Number of Skipped SMT Calls. We investigated how many SMT solver calls TopSeed can skip through its seeding process. Specifically, we calculated the number of SMT calls skipped while initializing program states using the seeds selected by TopSeed across the five largest benchmark programs over a 10-hour period.

Table X presents the number of skipped solver calls due to the seeds selected by TopSeed over the 10-hour period. On average, TopSeed's selected seeds saved an average of approximately 12,973 SMT calls across the five largest benchmarks. Notably, for the `diff` program, TopSeed achieved the highest reduction in SMT solver calls, which contributed to the most significant branch coverage improvement. As shown in Table V, TopSeed achieved 114.0% higher branch coverage compared to all baselines.

G. Threats to Validity

(1) TOPSEED was evaluated using KLEE and CREST, which are prominent symbolic executors. Its performance may vary when applied to other symbolic executors. (2) The three hyperparameters in TOPSEED, used for running the symbolic execution engine KLEE, were determined through trial and error. Although Section IV-E demonstrates stable performance across our benchmark suite and another symbolic execution engine, CREST, these settings may not be optimal for other programs or symbolic engines. (3) We used 17 carefully selected benchmarks. While this benchmark suite includes all major programs used in the four baselines, it may not sufficiently evaluate the full scope of TOPSEED's performance.

V. RELATED WORK

Boosting Symbolic Execution. Our work shares a common goal with various existing techniques aimed at enhancing symbolic execution [8], [11], [15], [18], [19], [33], [34], [35], [36]. To our knowledge, it is the first to achieve this goal by adaptively selecting optimal seeds without the need for a predefined seed corpus. Search strategies [8], [11], [33], [37] prioritize which states to explore, focusing on unique criteria such as less-traveled paths [8] and minimal distance to uncovered codes [37]. In contrast, state-pruning strategies [15], [34], [35] aim to eliminate redundant states, guided by specific pruning criteria, including previously explored basic blocks [12] and executed path suffixes [14]. Constraint-solving techniques [18], [19], [36], [38] have developed methods for effectively solving expensive and complex constraints, such as floating-point calculation [36], to increase the overall effectiveness of symbolic execution. TOPSEED is orthogonal to these existing methods, enabling it to be integrated to further enhance their performance.

Symbolic Execution with Machine Learning. Our approach belongs to the techniques that utilize machine learning to enhance symbolic execution [11], [15], [33], [39], [40], [41]. However, the specific aims of applying machine learning differ among these techniques. For example, Learch [11] and ParadySE [33] employ offline learning algorithms to automatically develop effective search strategies for symbolic execution. Leo [40] aims to learn how to transform the target program under test for boosting the efficacy of symbolic execution. Homi [15] uses learned discrete and continuous probability distributions to retain only valuable program states while minimizing the total state count. Distinguished from these, TOPSEED is the first to implement an online learning algorithm that automatically identifies useful seeds from scratch.

Symbolic Execution with Seeding. Although several techniques [20], [22], [24], [42] utilize seed inputs in symbolic execution to achieve specific goals—such as reaching patch codes [20], generating highly-structured inputs [24], and prioritizing inputs [42]—they rely on predefined seeds. In contrast, TOPSEED does not require a predefined seed corpus. Instead, it automatically selects effective seeds from inputs generated during symbolic execution without prior knowledge.

VI. CONCLUSION

State-of-the-art symbolic execution techniques undoubtedly enhance performance through various unique methods. However, we note that these performance gains are often achieved without seeding. This paper aims to automatically select promising seeds from inputs generated during interactions with these advanced techniques, without any prior knowledge, thereby further boosting their effectiveness. Our approach, TOPSEED, involves iteratively grouping generated inputs, ranking each group, and selecting the most promising seed from the top-ranked group based on accumulated data. Experimental results demonstrated that integrating TOPSEED with four advanced techniques substantially improves performance in terms of bug detection and code coverage.

ACKNOWLEDGEMENT

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2021R1C1C2006410, No. RS-2023-00208094). This work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00438686, Development of software reliability improvement technology through identification of abnormal open sources and automatic application of DevSecOps). This work was partly supported by the Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (RS-2019-II190421, AI Graduate School Support Program(Sungkyunkwan University)).

REFERENCES

[1] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '05, 2005, pp. 263–272.

[2] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1–10:38, 2008.

[3] 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, 2005, pp. 213–223.

[4] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '08, 2008, pp. 209–224.

[5] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '08, 2008, pp. 337–340.

[6] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *International Conference on Computer Aided Verification*, ser. CAV '07, 2007, pp. 519–531.

[7] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, 2009, pp. 359–368.

[8] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," *ACM SigPlan Notices*, vol. 48, no. 10, pp. 19–32, 2013.

[9] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 291–302.

[10] S. Cha, S. Hong, J. Bak, J. Kim, J. Lee, and H. Oh, "Enhancing dynamic symbolic execution by automatically learning search heuristics," *IEEE Transactions on Software Engineering*, pp. 3640–3663, 2022.

[11] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2526–2540.

[12] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '08, 2008, pp. 351–366.

[13] S. Bugrara and D. Engler, "Redundant state detection for dynamic symbolic execution," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13, 2013, pp. 199–212.

[14] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Eliminating path redundancy via postconditioned symbolic execution," *IEEE Transactions on Software Engineering*, pp. 25–43, 2018.

[15] S. Cha and H. Oh, "Making symbolic execution promising by learning aggressive state-pruning strategy," in *The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20, 2020.

[16] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, p. 193–204.

[17] D. Trabish, N. Rinetzky, S. Shoham, and V. Sharma, "State merging with quantifiers in symbolic execution," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1140–1152.

[18] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 68–78.

[19] T. Kapus, F. Busse, and C. Cadar, "Pending constraints in symbolic execution for better exploration and seeding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 115–126.

[20] 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*, 2013, pp. 235–245.

[21] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: testing for divergences between software versions," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1181–1192.

[22] J. H. Siddiqui and S. Khurshid, "Staged symbolic execution," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 2012, pp. 1339–1346.

[23] H. M. Le, "Kluzzer: Whitebox fuzzing on top of llvm," in *Automated Technology for Verification and Analysis: 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings 17*. Springer, 2019, pp. 246–252.

[24] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[25] K. Krishna and M. Narasimha Murty, "Genetic k-means algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, pp. 433–439, 1999.

[26] J. Yoon and S. Cha, "Featmaker: Automated feature engineering for search strategy of symbolic execution," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, 2024.

[27] G. A. tool for measuring coverage, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2021.

[28] C. A. concolic test generation tool for C, https://github.com/jburnim/crest, 2008.

[29] J. Jaffar, V. Murali, and J. A. Navas, "Boosting concolic testing via interpolation," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '13, 2013, pp. 48–58.

[30] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 413–424.

[31] Y. Kim, M. Kim, Y. Kim, and Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 1143–1152.

[32] S. Cha and H. Oh, "Concolic testing with adaptively changing search heuristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 235–245.

[33] S. Cha, S. Hong, J. Lee, and H. Oh, "Automatically generating search heuristics for concolic testing," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 1244–1254.

[34] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 350–360.

[35] Q. Yi, Y. Yu, and G. Yang, "Compatible branch coverage driven symbolic execution for efficient bug finding," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1633–1655, 2024.

[36] D. Liew, D. Schemmel, C. Cadar, A. Donaldson, R. Zähl, and K. Wehrle, "Floating-point symbolic execution: A case study in n-version programming," in *IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, 11 2017, pp. 601–612.

[37] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08, 2008, pp. 443–446.

[38] D. A. Ramos and D. Engler, "{Under-Constrained} symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[39] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, pp. 842–853.

[40] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, "Learning to accelerate symbolic execution via code transformation," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2018.

[41] S. So, S. Hong, and H. Oh, "{SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1361–1378.

[42] C. Zhang, A. Groce, and M. A. Alipour, "Using test case reduction and prioritization to improve symbolic execution," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 160–170.