Making Symbolic Execution Promising by Learning Aggressive State-Pruning Strategy

Sooyoung Cha
Korea University
Republic of Korea
sooyoungcha@korea.ac.kr

Hakjoo Oh*
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

ABSTRACT

We present Homi, a new technique to enhance symbolic execution by maintaining only a small number of promising states. In practice, symbolic execution typically maintains as many states as possible in a fear of losing important states. In this paper, however, we show that only a tiny subset of the states plays a significant role in increasing code coverage or reaching bug points. Based on this observation, Homi aims to minimize the total number of states while keeping *promising* states during symbolic execution. We identify promising states by a learning algorithm that continuously updates the probabilistic pruning strategy based on data accumulated during the testing process. Experimental results show that Homi greatly increases code coverage and the ability to find bugs of KLEE on open-source C programs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Dynamic Symbolic Execution, Online Learning

ACM Reference Format:
Sooyoung Cha and Hakjoo Oh. 2020. Making Symbolic Execution Promising by Learning Aggressive State-Pruning Strategy. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409755

1 INTRODUCTION

Symbolic execution [6, 7, 14, 23] is an effective software testing method to increase code coverage and find subtle bugs. The key idea of this method is to systematically explore program's diverse paths by substituting program inputs with symbolic ones to execute the program symbolically. At a high-level, symbolic execution iteratively selects, executes, and forks a state while maintaining a set of states during its testing process. In particular, it forks the state into one or two separate states according to the feasibility of branch conditions encountered during the symbolic execution. Thanks to the systematic process, symbolic execution has been actively used in a variety of applications: operation systems [17], smartphone apps [1], neural networks [26], and smart contracts [20, 22].

However, performing symbolic execution on real-world programs inevitably faces the infamous state-explosion problem that exponentially increases the number of states to be maintained, leading to significant increases of memory usage. Hence, in practice, symbolic executor (e.g., KLEE [5]) takes as input the memory budget to prevent unexpected memory usage, and maintains as many states as possible within the memory budget to reduce the risk of losing important states during testing. This reasonable behavior causes the symbolic executor to suffer from two practical problems. First, preserving as many states as possible increases the total number of candidate states, which makes it difficult for symbolic executor to decide proper states in a sense of increasing code coverage or finding bugs. Second, since the accumulated states easily exceed a given memory budget, numerous states are randomly pruned to reduce the memory usage. As we demonstrate in Section 2, when performing KLEE [5] with the default memory budget (2GB) on C open-source programs, the number of states to maintain is tens of thousands, and the number of blindly pruned states ranges from tens of thousands to hundreds of thousands on average.

To resolve this state-explosion problem, we aim to minimize the total number of states but to keep *promising* states during symbolic execution. Of the preserved states, we observed that there exist a very few promising states to effectively increase the code coverage or to reach the bug points; thus, symbolic execution becomes more effective and efficient if we only maintain those small number of promising states in a sense of resolving the state-explosion problem. To achieve our goal, the technical challenges we need to address are (1) to estimate how promising each state is and (2) to determine how many states we need to prune. Although diverse approaches exist with the goal of reducing the search space of symbolic execution [2, 3, 15, 27, 30, 31], their goals are not to maintain a small number of promising states; the existing approaches aim to identify and prune only the redundant states that meet the predefined criteria from the total ones. For instance, post-conditioned symbolic execution [30, 31] is to prune only the states having the same path suffixes as previously explored states, and Jaffar et al. [15] discard program paths guaranteed to be unreachable to bug points.

We present a new technique to adaptively maintain only a small number of promising states during symbolic execution via online learning. To achieve our goal, we introduce two key ideas: a *probabilistic* pruning strategy and a learning algorithm. First, we

*Corresponding author

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409755

define the probabilistic pruning strategy that contains both continuous and discrete probability distributions. We use the former distribution to score how promising each state is, and the latter one to decide how many states are pruned. That is, we reduce the problem of solving the two technical challenges into the problem of learning both probabilistic distributions. Second, we present a learning algorithm that continuously updates the two probabilistic distributions online based on data accumulated during symbolic execution.

Experimental results show that symbolic execution with our technique significantly improves branch coverage while maintaining a relatively small number of states compared to the general symbolic execution on open-source C programs. We implemented our technique in a tool, Homi, on top of KLEE [5] and evaluated it on 9 C programs (10-61KLoC). Symbolic execution with Homi succeeds in covering more branches and finding more real bugs than conventional symbolic execution on 9 benchmarks. For instance, our technique is able to generate the bug-triggering inputs that cause abnormal termination and segmentation fault in `grep-2.6` and `combine-0.4.0`, respectively, while conventional symbolic execution failed to do so.

Contributions. Our contributions are as follows:

- We present a new technique to maintain only promising states by continuously learning the probabilistic pruning strategy online during symbolic execution.
- We demonstrate the effectiveness of Homi on 9 open-source C programs by comparing symbolic execution with vs. without Homi.
- We make our tool, Homi, and data publicly available.[1]

2 PRELIMINARIES

In this section, we describe a general algorithm and limitation of symbolic execution, and explain our observation to present the goal of this paper.

Symbolic Execution. The main idea of symbolic execution [5, 7, 8] is to systematically explore program's diverse paths by replacing program inputs with symbolic ones to execute a program symbolically. Algorithm 1 presents a generic algorithm for symbolic execution, except for a few change, line 6, that stems from our main approach. Generally, symbolic execution maintains a set S of program states until the time budget expires, where a single state consists of a tuple $(instr, store, \Phi)$. Each element of a tuple respectively denotes the next instruction to be executed ($instr$), a symbolic store ($store$) which maps the program variables into symbolic values, and a path-condition (Φ) which is a conjunction of branch conditions evaluated symbolically in the state. The symbolic execution generates test-cases by iteratively selecting, executing and updating the states in S during its testing process.

The Run procedure in Algorithm 1 takes a program P under test and the time budget (N) as input, and returns a set T of test-cases generated within the time budget. At line 2, the algorithm initializes a set S as an initial state $(instr_0, store_0, true)$, where $instr_0$ is the very first instruction executed in the program P, $store_0$ is the initial mapping information, and Φ is set to $true$. For instance, suppose that there exists a small program P under test as follows:

```
void main(int x, int y){
  if(x>89) printf("good");
  if(x==2 && y>25) assert("bad");}
```

With this program as an input, $instr_0$ is set to the first instruction of the program, if(x>89), $store_0$ is $[x \mapsto \alpha, y \mapsto \beta]$, and Φ is $true$. At line 3, the algorithm initializes a set T of test-cases to an empty set. At line 5, the $\textsc{Prune}_M$ function decides a set of states to be pruned; that is, it randomly selects a subset S_P of S with the size of $|S| * r$ (e.g., r=0.1) when $|S|$, the size of S, exceeds the given memory capacity M. Otherwise, it returns an empty set (i.e., $S_P = \emptyset$). At line 7, the algorithm updates the set S with the difference set between S and S_P. For every state in the pruning set S_P, the algorithm generates a test-case t which is a model of the path-condition Φ in the state $(instr, store, \Phi)$ at line 8-9.

After the test-case generation, the Select function, namely a search heuristic [9, 19, 24, 28], chooses a single state $(instr, store, \Phi)$ from the set S based on its own selection criteria (line 10). With the selected state, the Execute function executes the instruction $instr$, and returns the updated state $(instr', store', \Phi)$. At line 13, if the instruction $instr'$ is an if/else statement, Algorithm 1 first checks the feasibility of the path-conditions corresponding to both two branches. If the path-condition of the if statement, $(\Phi \wedge \phi)$, is satisfiable, the algorithm adds the new state $(instr_1, store', \Phi \wedge \phi)$ into the set S (line 14). Likewise, the algorithm adds the new state into the set S if the path-condition of the else statement is satisfiable (line 15). When both sides of the branch are feasible, the algorithm forks the single state into two states, where this forking process causes the state-explosion problem in symbolic execution. On the other hand, if $instr'$ is the halt statement, the algorithm generates

Algorithm 1 Symbolic Execution

Input: Program (P), time budget (N), and the probabilistic data ($\mathcal{P}$).
Output: A set of test cases (T)

1: **procedure** Run(P, N, $\mathcal{P}$)
2: $S \leftarrow \{(instr_0, store_0, true)\}$ ▷ initial states
3: $T \leftarrow \emptyset$ ▷ initial test cases
4: **repeat**
5: $S_P \leftarrow \textsc{Prune}_M(S, M, r)$ ▷ M is memory and r is ratio.
6: $S_P \leftarrow S_P \cup \textsc{Prune}(S, \mathcal{P}, \eta_t)$ ▷ prune states
7: $S \leftarrow S \setminus S_P$
8: **for each** $(\_, \_, \Phi) \in S_P$ **do** ▷ generate test cases
9: $T \leftarrow T \cup \{(\Phi, \text{Model}(\Phi))\}$
10: $(instr, store, \Phi) \leftarrow \text{Select}(S)$ ▷ choose a state
11: $S \leftarrow S \setminus \{(instr, store, \Phi)\}$
12: $(instr', store', \Phi) \leftarrow \text{Execute}(instr, store, \Phi)$
13: **if** $instr'$ = (if (ϕ) then $instr_1$ else $instr_2$) **then**
14: **if** SAT$(\Phi \wedge \phi)$ **then** $S \leftarrow S \cup \{(instr_1, store', \Phi \wedge \phi)\}$
15: **if** SAT$(\Phi \wedge \neg\phi)$ **then** $S \leftarrow S \cup \{(instr_2, store', \Phi \wedge \neg\phi)\}$
16: **else if** $instr'$ = halt **then**
17: $T \leftarrow T \cup \{(\Phi, \text{Model}(\Phi)\}$ ▷ generate test cases
18: **until** budget N expires (or $S = \emptyset$)
19:
20: **for each** $(\_, \_, \Phi) \in S$ **do** ▷ generate test cases
21: $T \leftarrow T \cup \{(\Phi, \text{Model}(\Phi))\}$
22: **return** T

[1] Homi: https://github.com/kupl/HOMI_public

Table 1: The number of states and pruned states on C open-source programs (time budget:5h, memory budget:2GB)

	gawk	grep	vdir	ginstall	trueprint
#states ($\|S\|$)	37K	41K	43K	60K	49K
#pruned states	34K	112K	115K	587K	155K

a test-case t and adds a pair of path-condition and a test-case, (Φ, t), to the set T of test-cases. For simplicity, we have omitted the cases when *instr'* is the other instructions such as load, store, and call instructions. Algorithm 1 repeats this process until the time budget N expires or the set S becomes an empty set. When the loop ends, at line 20-21, the algorithm generates test-cases using the path-conditions of all remaining states in the set S, which have not yet reached the halt statement. Lastly, the algorithm returns the generated test-cases T as an output.

Limitation. The general symbolic execution attempts to maintain as many states as possible within the memory budget to reduce the loss of critical states during testing. This behavior, however, significantly degrades the performance of symbolic execution applied to real-world programs as the number of states in both S and S_P grows. The greater the number of states in the set S, the harder it is for the Select function to choose meaningful states which are likely to increase the code coverage or to reach the buggy locations. Furthermore, the increases in the size of the set of pruned states, S_P, may lead to the loss of promising states as they are forcibly pruned from the set S of candidate states.

Table 1 shows the average number of candidate states (S) and pruned total states when performing KLEE [5], a popular symbolic execution tool, on open-source C programs for 5 hours with the default memory capacity, 2GB. Overall, the size of the candidate set is tens of thousands, and the number of pruned states ranges from tens of thousands to hundreds of thousands. For instance, when performing KLEE on grep, the Select function should choose a state from about 41,000 candidate states on average for each iteration at line 4-18; the $\textsc{Prune}_{\textsc{M}}$ function blindly prunes about 112,000 states due to exceeding the memory budget even though the promising states may exist among the pruned ones.

Goal. The goal of this paper is to maintain only promising states via aggressive state-pruning during symbolic execution. In our work, we define the promising states as having the potential to effectively increase branch coverage when they are further explored, and observe that there are a very few promising states among the total candidate states. Hence, if we succeed in performing symbolic execution while keeping them only, we are able to maximize code coverage and to find many bugs. That is, the Prune function in Algorithm 1 enables the symbolic execution to maintain the minimized set S of candidate states while preserving the promising states, and prevents situations where the candidate states are blindly pruned due to memory overrun. To achieve this goal, the technical challenges we must address are as follows:

(1) How promising each state is?
(2) How many states do we need to prune?

In this paper, we address the challenges via the probabilistic pruning strategy learned online during symbolic execution.

3 OUR TECHNIQUE

In this section, we describe our technique, Homi, in detail. Section 3.1 defines the probabilistic pruning strategy (Prune) used in Algorithm 1. Section 3.2 describes our symbolic execution algorithm (Algorithm 2) with the online learning technique for the probabilistic pruning strategy.

3.1 Probabilistic Pruning Strategy

The pruning function (Prune) in Algorithm 1 decides the set S_P of pruned states based on the probabilistic data $\mathcal{P}$. This function takes as input the set S of all states, the probabilistic data $\mathcal{P}$, and the time cycle η_t. For every η_t seconds, the pruning strategy selects the set S_P of "unpromising" states in two steps: sampling and pruning. In the experiments, we set the hyper-parameter η_t to 30 seconds based on our observation that the short pruning cycle (e.g., 30) is generally more effective than the large one (e.g., 300) when testing real-world benchmarks.

Sampling. The first step, sampling, is to obtain the two important values from the probabilistic data $\mathcal{P}$, where $\mathcal{P}$ consists of a tuple (F, $\mathcal{P}_{stgy}$, $\mathcal{P}_{ratio}$). F denotes a set of n features to represent each state in the set S as an n-dimensional boolean vector. $\mathcal{P}_{stgy}$ is the distribution of an n-dimensional vector θ to calculate how promising each state is, and $\mathcal{P}_{ratio}$ is the distribution of the ratio r to decide the number of states to be pruned. For simplicity, we assume that the parameter vector θ and ratio r are given by sampling step from the learned distribution $\mathcal{P}_{stgy}$ and $\mathcal{P}_{ratio}$, respectively. We explain how we obtain these two values in Section 3.2.3 and 3.2.4.

Pruning. The second step is to select the states to be pruned by using the two sampled values, θ and r, and the set F of features in $\mathcal{P}$. We define the probabilistic pruning strategy as the following Prune function:

$$\textsc{Prune}(S, \mathcal{P}, \eta_t) = \begin{cases} \underset{S_P \subseteq S \wedge |S_P| = |S| * r}{\operatorname{argmin}} \sum_{s \in S_P} score(s, \theta) & \text{if } (F \neq \emptyset) \\ \emptyset & \text{otherwise} \end{cases}$$

where the function returns an empty set when the set F is empty. If not, the function scores each state in the set S, and returns the set S_P of the k states with the lowest scores in S (e.g., $k = |S| * r$).

To estimate how promising a state is, we first transform each state into a feature vector. Each feature denotes a boolean predicate that checks whether the path-condition Φ of the state s contains a specific branch condition ϕ. For instance, a feature describes whether the path-condition Φ of the state s involves (α>10), the branch condition. If true, the feature, feat(s), is 1; otherwise, it is 0. Formally, the i-th feature is defined as:

$$\text{feat}_i(s) = \begin{cases} 1 & \text{if } (\phi_i \in \Phi) \wedge ((\_, \_, \Phi) \in s) \\ 0 & \text{otherwise} \end{cases}$$

where it takes a single state s as input and returns 1 or 0. Using the set F of n features in the given probabilistic data $\mathcal{P}$, we can convert a state into an n-dimensional boolean vector as follows:

$$\text{feat}(s) = \langle \text{feat}_1(s), \text{feat}_2(s), \ldots, \text{feat}_n(s) \rangle.$$

Algorithm 2 Our Approach

Input: A program (P), time budget (N)
Output: The set of test-cases (T)

```
1:  procedure Homi(P, N)
2:      ⟨T, D⟩ ← ⟨∅, ∅⟩
3:      initialize two sample spaces (S_time and S_ratio)
4:      N′ ←sample from U(S_time)
5:      P ← (∅, U([−1, 1]^n), U(S_ratio))
6:      repeat
7:          T′ ← Run(P, N′, P)
8:          for each (Φ, t) ∈ T′ do
9:              D ← D ∪ {(Φ, t, B)}                    ▷ B = Branches(t)
10:         GoodD ← Extract(D)
11:         NewF ← FGenenator(GoodD)
12:         P_stgy, P_ratio, N′ ← PGenerator(GoodD, NewF)
13:         P ← (NewF, P_stgy, P_ratio)
14:         T ← T ∪ T′
15:     until budget N expires
16:     return T
```

The features are automatically generated online by the data accumulated during symbolic execution. We explain how these features are obtained in Section 3.2.2.

After transforming each state in the set S into a feature vector, we calculate the score of each state s using the inner product of the feature vector feat(s) and the sampled n-dimensional vector θ as:

$$score(s, \theta) = \text{feat}(s) \cdot \theta.$$

For example, when n is 3, θ and feat(s) can be $\langle 0.4, -0.82, -0.3\rangle$ and $\langle 1, 0, 0\rangle$, respectively, where the output of $score(s, \theta)$ is 0.4. The feature vector, $\langle 1, 0, 0\rangle$, denotes that the path-condition of the state s only contains the branch condition corresponding to 1st feature. In the vector θ, $\langle 0.4, -0.82, -0.3\rangle$, the i-th value represents the importance of the i-th feature.

Finally, the Prune function returns the set S_P of the $|S| * r$ states with the lowest scores in the set S as output, where the pruning ratio r is obtained from the learned distribution $\mathcal{P}_{ratio}$. We remark that the selection for the set S_P can be done efficiently; after calculating the score of each state in the total set S, it ranks the states according to their scores, and then picks the bottom-k states, where k is $|S| * r$.

Note that we reduce the problem of solving the two technical problems discussed in Section 2 into the problem of learning probabilistic distributions, $\mathcal{P}_{stgy}$ and $\mathcal{P}_{ratio}$.

3.2 Homi

The key point of our approach, Algorithm 2, is to continuously update the features and the two probabilistic distributions, $\mathcal{P}_{stgy}$ and $\mathcal{P}_{ratio}$, via online learning during symbolic execution. Except for the probabilistic data ($\mathcal{P}$), the input and output of our algorithm are the same as the ones of Algorithm 1. Unlike the Algorithm 1 which performs the Run procedure only once within the time budget N, our algorithm performs the Run procedure n times by dividing N into n smaller budgets N'. This is because our algorithm terminates numerous states early through the probabilistic pruning strategy in the Run procedure; thereby, our algorithm performs the Run procedure multiple times with the updated data $\mathcal{P}$ to recover the terminated promising states. Note that we can also perform Algorithm 1 in the same way. However, without our state-pruning strategy, we experimentally observed that it usually performs better to run Algorithm 1 once for a long time period than to do multiple times for a short time period (Section 4.4).

We explain the workflow of how Algorithm 2 works in detail. At line 2, Algorithm 2 initializes the set T of test-cases and accumulated data D to an empty set, respectively. At line 3, the algorithm initializes two sample spaces, S_{time} and S_{ratio}; the former S_{time} denotes the sample space for the time budget N' to run the Run procedure at line 7, and the latter, S_{ratio}, represents the sample space for the pruning ratio used in the probabilistic pruning strategy Prune. S_{time} and S_{ratio} are defined as:

$$S_{time} = [\tau_{max}, \tau_{itv}], \quad S_{ratio} = [\eta_{max}, \eta_{itv}]$$

where the two hyperparameters, τ_{max} and τ_{itv}, are to define the discrete space of τ_{itv} equal intervals with τ_{max} as the maximum time budget. Likewise, the discrete space S_{ratio} is defined equally by the two hyperparameters, η_{max} and η_{itv}. For instance, if S_{time} and S_{ratio} are [600, 6] and [0.8, 4], their discrete spaces are as follows:

$$S_{time} = [100, 200, 300, 400, 500, 600], \quad S_{ratio} = [0.2, 0.4, 0.6, 0.8]$$

In the experiments, for the space S_{time}, we set τ_{max} and τ_{itv} to 800 seconds and 4. We respectively set η_{max} and η_{itv} to 0.6 and 3 for the space S_{ratio}; that is, our strategy prunes 20% or 40% or 60% of total states for aggressive state-pruning.

At line 4, the algorithm samples the initial time budget N' from the uniform distribution $\mathcal{U}(S_{time})$. It initializes the probabilistic data $\mathcal{P}$ consisting of a triplet (F, $\mathcal{P}_{stgy}$, $\mathcal{P}_{ratio}$) at line 5; initially, the set F of features is an empty set and each of two probabilistic distributions, $\mathcal{P}_{stgy}$ and $\mathcal{P}_{ratio}$, is a uniform distribution. As the set F in $\mathcal{P}$ is set to an empty set, the algorithm performs the Run procedure without any state-pruning on the first iteration of the loop at lines 6-15. Our algorithm repeats the following two main processes until the time budget N expires: 1) performing symbolic execution with the probabilistic pruning strategy based on the data $\mathcal{P}$ (line 7) and 2) updating the data $\mathcal{P}$ (line 13). On the first iteration of the loop, Homi performs the Run procedure (Algorithm 1) with the time budget N' and initial probabilistic data $\mathcal{P}$, and returns the set T' of generated test-cases at line 7. After the algorithm calculates the set B of branches covered by each test-case t in the set T', we accumulate the tuple (Φ, t, B) in the set D (line 8-9).

3.2.1 Collecting Promising Data. At line 10, we run the Extract function to extract the most "promising" but minimal set of data, *GoodD*, from the set D of accumulated data. Conceptually, the set *GoodD* is the smallest subset of D where the unions of B in *GoodD* is the same with the set of branches covered by all the test-cases in D. To formally define the set *GoodD*, we first calculate D^* as:

$$D^* = \underset{D' \subseteq D}{\text{argmax}} \left| \bigcup_{(\_,\_,B) \in D'} B \right|.$$

where the notation 'argmax' returns the set D^* of all arguments that maximize the objective. The set D^* is the set of all subsets of D which collectively maximize the set of covered branches. Then, the set *GoodD* is defined as:

$$GoodD = \underset{D' \in D^*}{\text{argmin}} |D'|$$

where the notation 'argmin' returns one of the arguments that minimize the objective. In practice, calculating the set *GoodD* corresponds to solving the set cover problem [16], the well-known np-complete problem. In this paper, we obtain the minimal set *GoodD* by applying the greedy algorithm which iteratively selects the element having the largest number of uncovered branches at each stage.

3.2.2 Generating Features. At line 11, Homi generates n features to transform each state into a feature vector in the probabilistic pruning function, Prune. Intuitively, a feature is a core branch condition that contributes to determining the value of a test case that effectively increases branch coverage. To generate the features, we use the core branch conditions in the path-condition Φ corresponding to each promising test-case in the set *GoodD*. We define a core branch condition ϕ as a condition that can be expressed in the predefined language L as follows:

$$\begin{aligned} \phi &::= cond \mid cond \wedge cond \mid cond \vee cond \\ cond &::= lv = n \\ lv &::= \alpha \mid \alpha[i] \end{aligned}$$

where the language is small yet sufficient to represent the minimum branch conditions that are necessary to directly determine the value of each test-case. An l-value (lv) denotes a symbolic value (α) or the value of i-th index of an array α ($\alpha[i]$). A condition ($cond$) consists of a boolean condition to express that the l-value equals to a constant value n. A core branch condition (ϕ) is a single $cond$ or a conjunction (disjunction) of $cond$. To generate a set of features from the promising data *GoodD*, we first collect the set *PC* of all path-conditions from *GoodD* as follows:

$$PC = \{\Phi \mid (\Phi, \_, \_) \in GoodD\}$$

Second, we collect the set *NewF* of new features by extracting core branch conditions in the set *PC* as:

$$NewF = \{\phi \in L \mid \phi \in \Phi, \Phi \in PC\}$$

That is, we extract only the conditions that can be expressed in the language L among the branch conditions of each Φ in the set *PC*. For instance, suppose that the set *PC* contains two sets of path-conditions as follows:

$$PC = \{\{(\alpha == 3), (\alpha > 1)\}, \{(\alpha[2] \neq 3), (\alpha[2] == 8)\}\}.$$

where the two branch conditions, $(\alpha == 3)$ and $(\alpha[2] == 8)$, in *PC* can be expressed in the language L (e.g., $lv = n$) while the remaining conditions, $(\alpha > 1)$ and $(\alpha[2] \neq 3)$, cannot be. Hence, we can define the set *NewF* of features from *PC* as follows:

$$NewF = \{(\alpha == 3), (\alpha[2] == 8)\}$$

where the two features in the set *NewF* are the minimal conditions to determine a model of each path-condition; for instance, the model of the first path-condition, $(\alpha == 3) \wedge (\alpha > 1)$, in *PC* is equals to the one of the minimal condition $(\alpha == 3)$. In short, the set *NewF* of generated features at line 11 represents the key evidences of the minimal test-cases that contribute to maximizing branch coverage until the current state.

3.2.3 Learning Distribution. At line 12, the function PGenerator learns the probabilities of two values, n-dimensional weight vector θ and the pruning ratio r, and returns a tuple $(\mathcal{P}_{stgy}, \mathcal{P}_{ratio}, N')$. The first element $\mathcal{P}_{stgy}$ denotes the probability for the weight vector θ that scores how promising each state is, and the second, $\mathcal{P}_{ratio}$, is the probability for the pruning ratio r that determines the number of states to be pruned. The time budget N' is the newly allocated time budget for the Run procedure on the next iteration of the loop.

The probability $\mathcal{P}_{stgy}$ consists of n distributions as:

$$\mathcal{P}_{stgy} = \mathcal{P}_1 \times \mathcal{P}_2 \times \cdots \times \mathcal{P}_n.$$

where $\mathcal{P}_i$ denotes the probability of the weight value θ^i corresponding to the i-th feature in the set *NewF*. To define the i-th distribution $\mathcal{P}_i$, we first collect the set of promising test-cases *GoodT* from *GoodD* as follows:

$$GoodT = \{t \mid (\_, t, \_) \in GoodD\}.$$

Whenever each test-case t is generated during symbolic execution, our algorithm additionally maintains a quadruple of information used to generate each test-case t as follows:

$$t = (F, \theta, r, N')$$

where F is the set of features, θ is the weight vector, r is the pruning ratio, and N' is the time budget. Using this additional information, we collect the set *GoodF* of the features which are used at least once when generating the promising test-case in the set *GoodT* as follows:

$$GoodF = \bigcup_{(F,\_,\_,\_) \in GoodT} F.$$

That is, the set *GoodF* contains the features that contribute to generating effective test-cases in terms of code coverage. Finally, we can define the i-th distribution $\mathcal{P}_i$ as:

$$\mathcal{P}_i = \begin{cases} \mathcal{N}(\mu(W_i), \sigma(W_i), -1, 1) & \text{if } (\phi_i^{new} \in GoodF) \\ \mathcal{U}([-1, 1]) & \text{otherwise} \end{cases} \quad (1)$$

where ϕ_i^{new} denotes the i-th feature in the set *NewF* that has been generated at line 11 in Algorithm 2.

If the i-th new feature (ϕ_i^{new}) belongs to the set *GoodF* of promising features, we learn the probability $\mathcal{P}_i$ which is the truncated normal distribution with median $\mu(W_i)$, standard deviation $\sigma(W_i)$, minimum value (-1), and maximum value (1). We define the set W_i as:

$$W_i = \{\theta^k \mid (\phi_i^{new} = \phi_k) \wedge (\{\phi_1, \cdots, \phi_n\}, \theta, \_, \_) \in GoodT\}.$$

Intuitively, W_i denotes the set of weight values corresponding to the i-th new feature ϕ_i^{new} that has already been used for each test-case in *GoodT*. Given the set W, the median $\mu(W_i)$ and standard deviation $\sigma(W_i)$ are calculated as:

$$\mu(W) = \sum_{w \in W} \frac{w}{|W|}, \quad \sigma(W) = \sqrt{\frac{\sum_{w \in W} (w - \mu(W))^2}{|W|}}.$$

On the other hand, if the i-th new feature (ϕ_i^{new}) does not belong to *GoodF*, we fix the probability $\mathcal{P}_i$ to a uniform distribution between -1 and 1 since there is no accumulated data corresponding to the i-th new feature for learning. In this way, we learn the probability $\mathcal{P}_{stgy}$ that consists of the n distributions from $\mathcal{P}_1$ to $\mathcal{P}_n$ based on the most promising data *GoodD*.

After the learning process of the probability $\mathcal{P}_{stgy}$, we calculate the probability $\mathcal{P}_{ratio}$ of the given pruning ratio r' which is one of the values in the predefined discrete space S_{ratio} as follows:

$$\mathcal{P}_{ratio}(X = r') = \frac{|\{(\_, \_, r, \_) \in GoodT \mid r' = r\}|}{|GoodT|} \quad (2)$$

The intuition is that the more the pruning ratio r is used to generate promising test-cases $GoodT$, the higher the probability of the ratio.

Lastly, we sample the new budget N' based on the following probability $\mathcal{P}_{time}$:

$$\mathcal{P}_{time}(X = N') = \frac{|\{(\_, \_, \_, N) \in GoodT \mid N' = N\}|}{|GoodT|} \quad (3)$$

The intuition is the same as the probability $\mathcal{P}_{ratio}$ above.

3.2.4 Sampling Values. We describe how to sample the weight vector (θ) and ratio (r) from the two learned distributions, $\mathcal{P}_{stgy}$ and $\mathcal{P}_{ratio}$, in the first 'sampling' step of the probabilistic pruning strategy. First, we sample the weight vector θ by using one of the three sampling methods: exploitation, reverse exploitation, and exploration. The first two methods are to exploit the learned distribution $\mathcal{P}_{stgy}$ as it is or reversely. The last method is to explore purely random weight vector.

Exploitation. We sample the new weight vector θ from the learned distribution $\mathcal{P}_{stgy}$ itself as:

$$\mathsf{Sample}_{\mathsf{exploit}}(\mathcal{P}_1 \times \mathcal{P}_2 \times \cdots \times \mathcal{P}_n) = \langle \theta^1, \theta^2, \cdots, \theta^n \rangle$$

where the i-th weight value θ^i is sampled from the i-th probability $\mathcal{P}_i$ in (1). Our expectation is that the new weight vector θ statistically similar to the promising weight vectors in the set $GoodD$ will likely increase the code coverage on the next iteration of the loop.

Reverse Exploitation. We sample the new weight vector θ_r by exploiting the learned distribution $\mathcal{P}_{stgy}$ reversely. We first generate the set of 100 real-numbers, U, by sampling the uniform distribution between -1 and 1 as:

$$U = \{r_1, r_2, \ldots, r_{100} \mid r_i \sim \mathcal{U}(-1, 1)\}.$$

The sampling method takes the probability $\mathcal{P}_{stgy}$ and the set U as input and returns the new weight vector θ_r as:

$$\mathsf{Sample}_{\mathsf{reverse}}(\mathcal{P}_1 \times \mathcal{P}_2 \times \cdots \times \mathcal{P}_n, U) = \langle \theta_r^1, \theta_r^2, \cdots, \theta_r^n \rangle$$

We assume that the i-th weight value θ^i is sampled from the probability $\mathcal{P}_i$ defined in (1). Then the i-th reverse weight value θ_r^i is calculated as follows:

$$\theta_r^i = \underset{u \in U}{\operatorname{argmax}} |u - \theta^i|$$

where the reverse value θ_r^i in U is the farthest one from the value θ^i sampled from the distribution $\mathcal{P}_i$. Hence, θ_r^i represents the value that is the most unlikely to be sampled in the learned distribution $\mathcal{P}_{stgy}$. We expect that this weight vector would lead the symbolic execution to explore the branches uncovered in previous iterations.

Table 2: 9 benchmark programs

Programs	LOC	# of Branches
gawk-3.1.4	60,904	11,934
grep-2.6	56,931	7,021
combine-0.4.0	35,756	2,359
trueprint-5.4	12,229	2,518
ginstall (8.31)	22,290	3,652
ptx (8.31)	22,148	5,262
vdir (8.31)	19,378	3,830
pr (8.31)	12,156	1,991
dd (8.31)	10,531	1,547

Exploration. For the last method, we generate a weight vector θ by sampling from the uniform distribution $\mathcal{U}([-1, 1]^n)$, where the i-th value θ^i is a random real-number between -1 and 1. In the experiments, to accumulate enough data D for learning the distribution $\mathcal{P}_{stgy}$, Algorithm 2 repeats the loop, using only the exploration method m times (e.g., m=10). After enough data is collected, we set the same probabilities for the three sampling methods.

Finally, we sample the pruning ratio r based on the probability in (2) when sampling the weight vector θ by exploitation or reverse exploitation. Otherwise, when sampling the weight vector θ by exploration, the pruning ratio is randomly sampled in the uniform distribution $\mathcal{U}(S_{ratio})$. Likewise, we obtain the next testing budget N' on the same basis as sampling the pruning ratio; that is, we sample the budget from the uniform distribution $\mathcal{U}(S_{time})$ for the exploration case only. If not, we sample the one in (3).

As the loop at lines 6-15 in Algorithm 2 iterates, our technique, Homi, is able to make smarter decisions on how to represent each state (F), how promising each state is ($\mathcal{P}_{stgy}$), and how many states are pruned ($\mathcal{P}_{ratio}$).

4 EXPERIMENTS

In this section, we experimentally evaluate our approach, Homi, to answer the following research questions:

- **Effectiveness**: How effectively does Homi improve branch coverage? How many branches and bugs are reachable by Homi only? (Section 4.2)
- **The number of states**: How many states does Homi maintain during testing compared to general symbolic execution? (Section 4.3)
- **Comparison with naive approach**: How well does Homi (Algorithm 2) perform compared to symbolic execution with random state-pruning? (Section 4.4)

We implemented our approach in a tool, Homi, on top of KLEE [5], a publicly available symbolic execution tool for testing C programs. We conducted all experiments on a Linux machine equipped with two Intel Xeon Processors E5-2630 and 192GB RAM, where it has a total of 16 cores and 32 threads.

4.1 Settings

Benchmarks. We used 9 GNU open-source C programs for evaluation. Table 2 shows the total number of lines and branches for each benchmark, where the largest benchmark, gawk, has about 12K

branches. The last five benchmarks in Table 2 are among the larger programs in GNU Coreutils-8.31. To construct our benchmark suite, we used two criteria: 1) the benchmarks have been widely used in prior work on dynamic symbolic execution [4, 5, 9–11, 21, 24, 29], and 2) they are relatively larger and more challenging than those often used in existing work on KLEE.

Baselines. We compared our approach with the general symbolic execution (Algorithm 1) without state-pruning but with 9 different search heuristics. Specifically, we used the following 9 search heuristics: CPICount (CallPath Instruction Count), CovNew, MinDistance (Minimal Distance to Uncovered), InstrCount (Instruction Count), QueryCost, RandomPath, Depth, RandomState, and RoundRobin; the last heuristic is the default heuristic of KLEE that uses CovNew and RandomPath in a round robin fashion. All these heuristics are implemented in KLEE [5]. Note that we deliberately used 9 search heuristics instead of using only the default heuristic. This is because, as demonstrated in Section 4.2, the performance of the general symbolic execution varies greatly depending on both the subject program and search heuristic.

We applied Homi on top of the best search heuristic that achieves the highest branch coverage for each program. For instance, we applied Homi on top of the MinDistance heuristic for `gawk` while applying Homi on top of the CPICount heuristic for `grep`. Note that Homi and search heuristics are orthogonal, and they are naturally combined since Homi works regardless of search heuristics; in a generic symbolic execution algorithm (Algorithm 1), Homi decides which states to prune at line 6 while search heuristics determine which states to explore further at line 10.

Other Settings. For all evaluations, we maintained the same experimental environments: symbolic arguments, time budget, and memory capacity. First, we used the symbolic arguments used in [5] (e.g., "--sym-args 0 1 10 --sym-args 0 2 2 --sym-files 1 8 --sym-stdin 8 --sym-stdout"). Second, we used the same memory capacity, 2GB, where it is the default setting of KLEE. Lastly, we allocated 5 hours to both the baselines (Algorithm 1) and our technique (Algorithm 2) for all benchmarks as time budget. We repeated all experiments five times and reported the average results.

4.2 Effectiveness

We evaluate the effectiveness of our approach, Homi, from two perspectives: branch coverage and bug-finding capability. In summary, Homi is able to significantly increase branch coverage and exclusively find bug-triggering inputs, compared to the general symbolic execution.

4.2.1 Branch Coverage. For each benchmark in Table 2, we report the average number of total covered branches (Figure 1) and exclusively covered branches (Table 3), by Homi and top-5 search heuristics, respectively. As both the general symbolic execution (Algorithm 1) and Homi (Algorithm 2) return as output the set of test-cases, we plotted the number of branches covered by all preceding test-cases to depict the coverage graph in Figure 1. In particular, when the time budget (5h) expired, we re-executed the binary of the program with each test-case in the set T sequentially, where the 'sequence' denotes the time each test-case was created. We calculated the cumulative number of covered branches corresponding to the creation time of the test-case; we used `gcov`, one of the most popular tools for measuring code coverage. As we mentioned in Section 3.2, Homi performs the general symbolic execution without state-pruning on the first iteration of the loop. Hence, to demonstrate the benefits of state-pruning only, we have plotted the accumulated number of covered branches after the time for the first iteration of Algorithm 2 elapsed. In our experiments, we first perform the general symbolic execution for 800 seconds, record the calculated number of covered branches on the graph, and then run Algorithm 1 for the remaining time period (e.g., 5h - 800 seconds). In other words, the graphs in the Figure 1 can clearly demonstrate the comparison of the performance of symbolic execution with and without state-pruning technique.

Figure 1 demonstrates the average number of branches covered by the search heuristics over time in 9 benchmarks. We used a total of 6 heuristics for each benchmark, consisting of the top five of the nine original search heuristics and our technique applied to the best one among the five. The experimental results show that the search heuristic with Homi succeeds in achieving the highest branch coverage for all benchmarks. In particular, for the two largest programs, `gawk` and `grep`, Homi notably increases the number of covered branches compared to the best heuristic without it. For instance, in `gawk`, the search heuristic with Homi, MinDistance+Homi, covered about 2,884 branches while MinDistance heuristic itself managed to cover about 2,447 branches only. Likewise, in `grep`, when the time budget (5h) expired, the best heuristic (CPICount) with and without Homi covered about 2,851 and 2,505 branches, respectively. Moreover, as shown in a benchmark `trueprint`, the rate for the coverage increase over time of the heuristic equipped with Homi was noticeably higher than the ones of other five search heuristics. In the two benchmarks, `combine` and `vdir`, applying Homi to the best heuristic has successfully covered about 100 more branches.

Figure 1 shows that in most programs, the number of branches covered by each top-5 search heuristics rises sharply at the end of the time budget; this interesting fact is observed because we have reported the branch coverage covered over time. This phenomenon occurs since the algorithm (Algorithm 1) generates the test-cases, at lines 20-21, for each state that has not yet reached the halt statement after the time budget expires, and the generated test-cases contribute to increasing the total number of branch coverage a lot. This implicitly shows that the general symbolic execution fails to preferentially explore such promising states more.

Note that we applied Homi with the best search heuristic just because it is more challenging to improve the performance of the heuristic that has already achieved high code coverage. In fact, Homi performs well regardless of the search heuristic. For instance, applying Homi even with the 6th search heuristic (CPICount) on gawk in Figure 1 can cover more branches than applying CPICount without Homi; CPICount+Homi covered 2,356 branches on average while CPICount covered 2,093 only.

4.2.2 Exclusively Covered Branches. Table 3 shows the number of exclusively covered branches achieved by each technique. In Table 3, the i-th best heuristic on each benchmark corresponds to the one in Figure 1; for instance, the best (BestH) and second best heuristic (2ndH) on `combine` are RandomPath and CPICount,

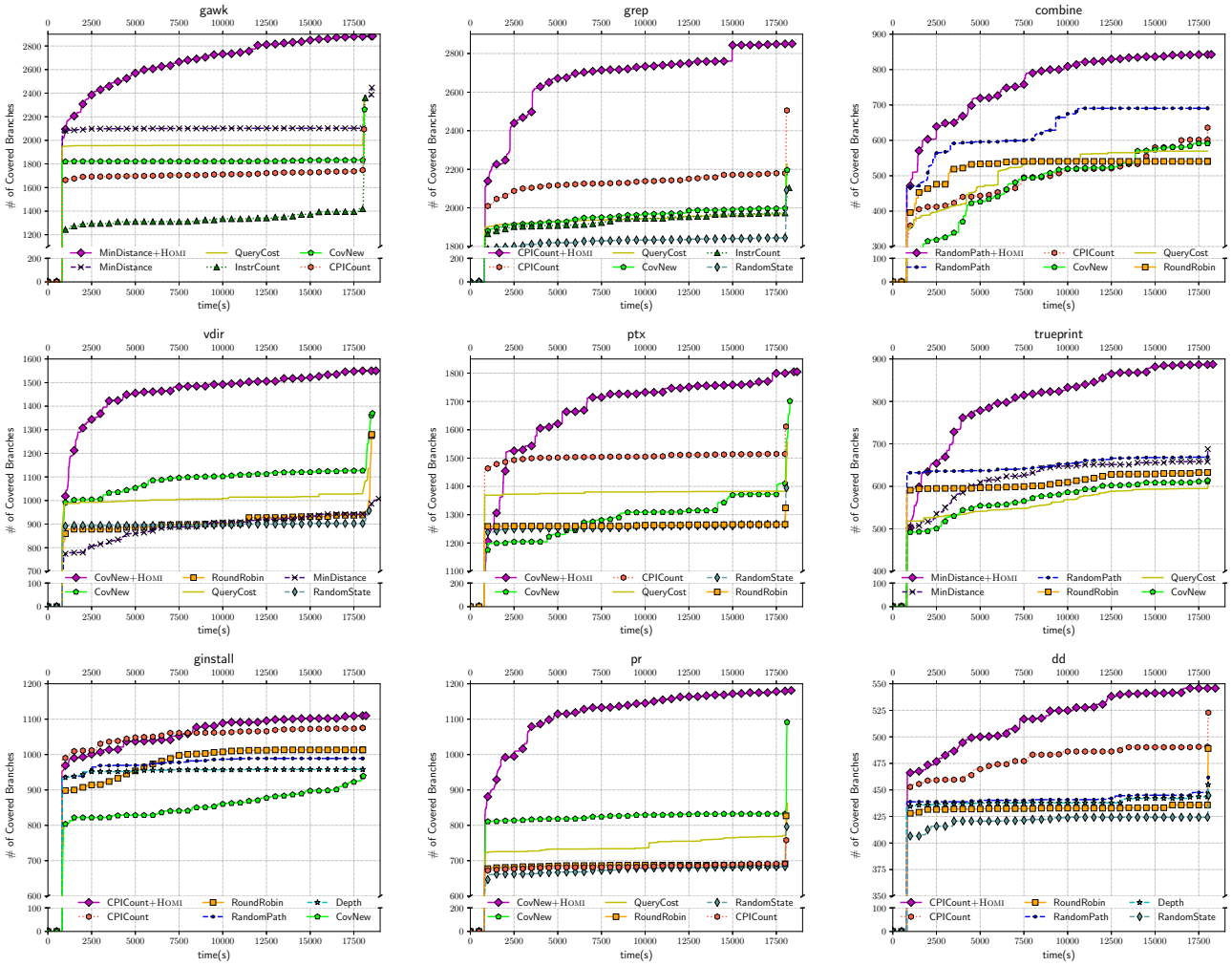

Figure 1: The average branch coverage achieved by top-5 heuristics and Homi on 9 benchmarks

Table 3: The average number of branches exclusively covered by top-5 heuristics and Homi on 9 benchmarks

	BestH+Homi	BestH	2ndH	3rdH	4thH	5thH
gawk	139	10	26	37	42	26
grep	208	53	22	116	14	1
combine	62	0	15	15	1	6
vdir	118	44	19	14	1	2
ptx	39	4	35	2	0	4
trueprint	147	17	0	0	3	52
ginstall	16	3	1	6	0	0
pr	61	3	1	2	0	3
dd	23	17	2	0	0	0
Total	813	151	121	192	61	94

respectively. The number achieved by our technique (BestH+Homi) denotes the number of branches that our technique covers but all the remaining top-5 heuristics fail to cover. The results show that our technique (BestH+Homi) is highly effective in increasing the number of exclusively covered branches. In total, the best heuristic with Homi was able to cover 813 branches while the best heuristic without Homi covered only 151 branches; in summary, the former exclusively covered 5.4 times more branches than the latter. For instance, in the largest program gawk, Homi significantly enhanced the performance of the best heuristic (MinDistance) by about 13.9 times. Likewise, our technique (CPICount+Homi) on grep exclusively covered 208 branches, but the best heuristic alone managed to exclusively cover 53 branches.

In addition, the number of exclusive branches covered by our technique (BestH+Homi) is even greater than the sum of the numbers of branches exclusively covered by each of top-5 heuristics, where the former is 813 and the latter is 619, i.e., 151+121+192+61+94. The number of branches that one of the top-5 heuristics can cover but our technique (BestH+Homi) fails is 816. In other words, the branch coverage achieved by applying Homi to the best-heuristic only for 5 hours is almost the same as the one achieved by applying

Table 4: Comparison of bug-finding ability of top-2 heuristics with vs. without Homi.

Benchmarks	Crash-Types	Bug-Triggering Inputs	Error Locations	BestH+Homi	BestH	2ndH+Homi	2ndH
gawk-3.14	Abnormal-termination	`"--nostalgi" "-"`	'Line: 1044 in main.c'	✔	✔	✔	✔
	Abnormal-termination	`"--compat" "-m" "r "`	'Line: 526 in /libc/stdlib/stdlib.c'	✔	✘	✔	✘
grep-2.6	Abnormal-termination	`"\n\w*\'*\n" "-"`	'Line: 1432 in /src/dfa.c'	✘	✘	✔	✘
combine-0.4.0	Segmentation fault	`"--field=, ,"`	'Line: 385 in /src/field.c'	✔	✔	✔	✔
	Segmentation fault	`"--fi=r.o1'" "-r" ""`	'Line: 633 in /src/df_options.c'	✔	✘	✘	✘

each of top-5 heuristics for 5 hours (25 hours in total). Despite the obvious advantages of Homi, it is still not optimal since it also failed to cover 816 branches achieved by top-5 search heuristics. From this observation, selective decision on applying Homi would be an interesting future work.

4.2.3 Bug-Finding Capability. In Table 4, we compared the bug-finding capability of two best heuristics both with and without Homi, respectively, for the three largest benchmarks: `gawk`, `grep`, and `combine`. In summary, Homi found a total of five reproducible bugs on the three benchmarks. In particular, the three bugs were only detectable by Homi while the general symbolic execution failed to find these bugs.

Table 4 shows the benchmark, the crash-type, the bug-triggering input generated by Homi, the error-location, and success or failure of bug-finding for each technique in order. In particular, we marked each technique as 'success' (✔) when the technique succeeded in finding the bug at least once during five iterations of the time budget (5h). On the contrary, when each technique totally failed to find the bug during the time period (5h $*$ 5times), we marked it as 'failure' (✘). The results show that our technique was able to generate a total of four distinct bug-triggering inputs in `gawk` and `combine`, but the best heuristic without Homi only generated the two inputs. We confirmed that the first bug-triggering input ("--nostalgi" "-") found in `gawk` is reproducible in the latest version (gawk-5.0.1). One interesting point is that the discovered bugs are different when applying Homi to the best and the second best heuristics; the best heuristic with Homi caused a crash, abnormal-termination, on `combine` while the second best one caused a segmentation fault on `grep`. That is, we expect that applying Homi to diverse (new) search heuristics will allow more bug-detection.

4.3 The Number of Candidate States

For each benchmark in Table 2, we compare the number of states that our technique and the general symbolic execution maintain during the testing period. Figure 2 shows the average number of states that can be selected by each technique for every second; more precisely, the average number denotes the set size of states, $|S|$, at line 10 in Algorithm 1. The results show that our technique (BestH+Homi) maintains a relatively small number of states for most of the time period on all benchmarks compared to the general symbolic execution. When performing the general symbolic execution without the state-pruning at first, which is the first iteration of the loop in Algorithm 2, our technique also faced the state-explosion problem. After the first iteration, however, ours has successfully maintained a small number of states. For instance, in `gawk`, our technique (MinDistance+Homi) kept about 1,897 states per second on average while the MinDistance heuristic maintained about 37,315 states. In other words, our technique succeeded in achieving the highest branch coverage in Figure 1 while maintaining 19.7 times fewer states than the general symbolic execution. In `grep`, CPICount+Homi and CPICount retained 2,030 and 41,210 states on average, respectively. In `vdir`, even after the first iteration of the loop in Algorithm 2, our technique sometimes faced the state-explosion problem. We confirmed that this problem occurs because the number of states grows exponentially faster than the number pruned by our state-pruning strategy.

For the general symbolic execution without our technique, we observed that keeping the fewer number of states is not directly related to improving the branch coverage. For instance, in `ptx`, RoundRobin heuristic maintains about 4,660 states during the symbolic execution, which is almost the same number of states maintained by our technique (CovNew+Homi). However, Figure 1 shows that RoundRobin covered about 300 fewer branches on average than CovNew of which the number of states is about 60,053 during symbolic execution. In another benchmark, `gawk`, the numbers of candidate states maintained by CPICount and MinDistance are almost the same, where the former is 36,299 and the latter is 37,315. However, the difference in the number of covered branches by the two techniques is approximately about 350 branches. That is, the key answer for increasing code coverage is not to blindly maintain the number of states, but to smartly keep only the "promising" states.

Although Homi successfully maintains such promising states on our benchmark suite, we were not able to provide high-level insights into why those states are promising. In our approach, we determine how promising a state is based on its corresponding feature vector and weight vector. However, as our learning algorithm represents each state as low-level features that check whether it contains core branch conditions, it was difficult to decode the learning outcomes and describe the intuition behind promising states.

4.4 Comparison with Naive Approaches

We evaluate the efficacy of Homi (Algorithm 2) by comparing it with two naive methods. The first naive method is to replace the probabilistic pruning strategy (Prune) in Algorithm 1 with the random pruning strategy (RandomPrune), and then to perform

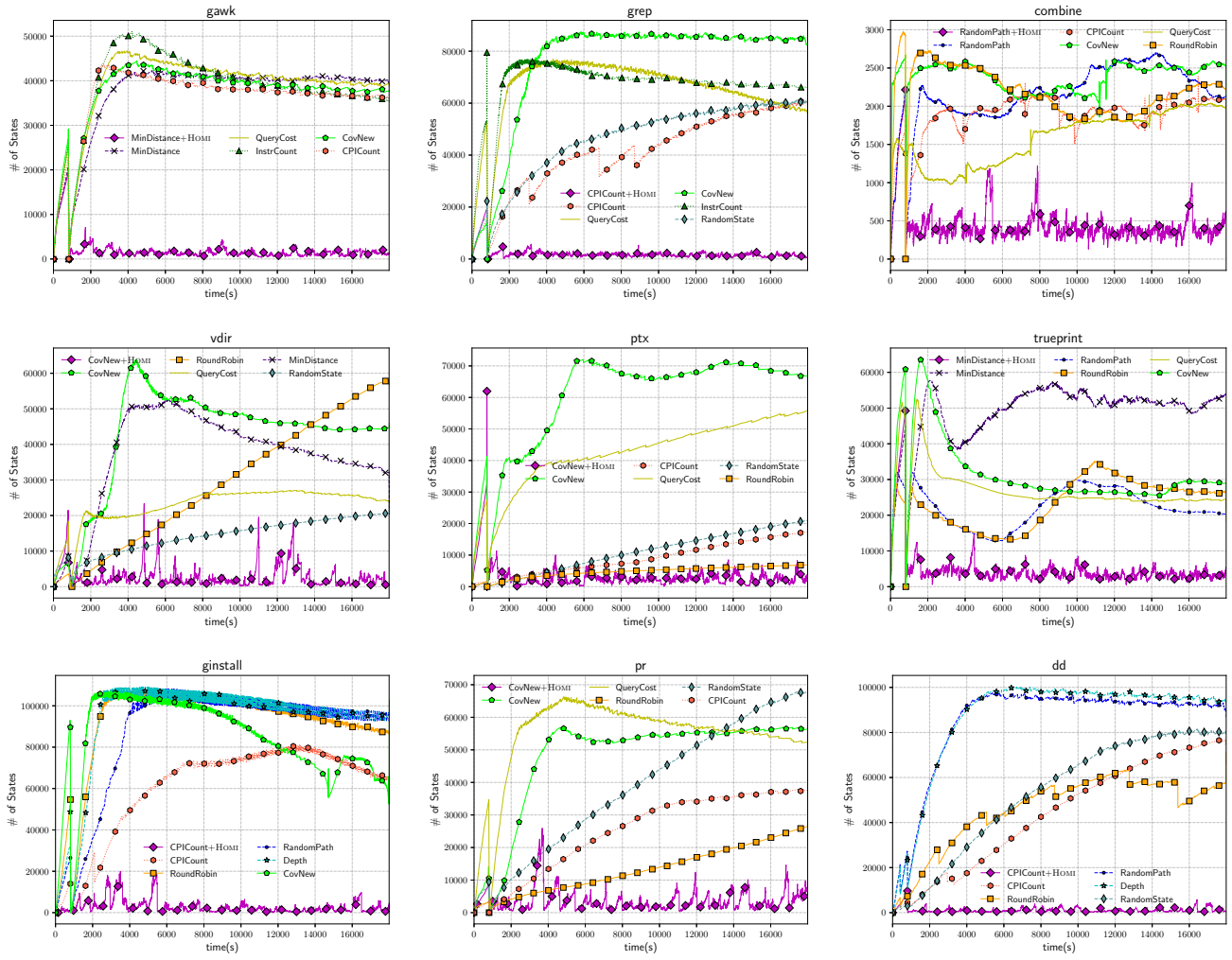

Figure 2: The average number of states for each technique to select on 9 benchmarks

Algorithm 2 without online learning (line 10-13) as:

$$\textsc{RandomPrune}(S, r) = \{s \in S_P \mid S_P \subseteq S \wedge |S_P| = |S| * r\}$$

where r is sampled from the uniform distribution $\mathcal{U}(S_{ratio})$ defined in Section 3.2. The second naive method is to perform the general symbolic execution (Algorithm 1) multiple times by dividing the total budget (5h) into smaller budgets N', where N' is sampled from $\mathcal{U}(S_{time})$ defined in Section 3.2. In `grep`, we compared branch coverage achieved by ours (CPICount+Homi), the best heuristic (CPICount), the first naive approach (CPICount+`RandomPrune`), and the second one (CPICount[Divide]), respectively.

Figure 3 shows that Homi (Algorithm 2) is essential to effectively improve branch coverage. For example, ours covered at least 300 more branches than the second best method (CPICount+`RandomPrune`). The second and the third best methods (CPICount +`RandomPrune` and CPICount) achieved nearly identical branch coverage when time budget expired. For the general symbolic execution (Algorithm 1) without state-pruning, it is much better to

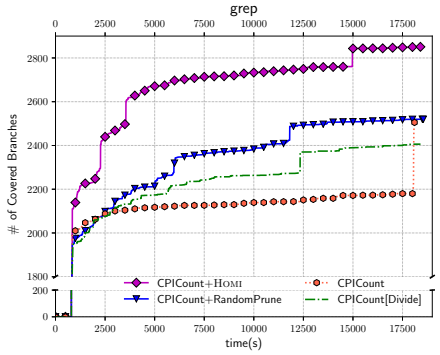

Figure 3: Comparison with two naive approaches on grep

perform symbolic execution for a long time than to perform symbolic execution several times with small budget; the former and the latter was able to cover 2,505 and 2,406 branches, respectively.

4.5 Threats to Validity

(1) We manually tuned the several hyper-parameters: η_t, η_{max}, and η_{itv}. To determine each value, we ran Homi with a few different values (e.g., 30, 300) on three benchmarks, and chose an appropriate one achieving the highest coverage during the experiments. Then, we applied the same value for the remaining six benchmarks. However, the tuned values may not be suitable for larger open-source C programs (e.g., LOC > 100K). (2) In evaluation, we used both the default SMT solver of KLEE (STP [13]) and the default memory budget (2GB). However, the performance of Homi may vary for different SMT solvers and memory budgets. (3) We used 9 C open-source programs extensively used in previous works [4, 5, 9–11, 21, 24, 29]. But these may not be representative.

5 RELATED WORK

In this section, we discuss existing works that are closely related to our goal and approach, respectively. At a high level, our goal is to prune the search space of symbolic execution [2, 3, 15, 27, 30, 31], and our approach belongs to the techniques that combine symbolic execution with machine learning [9–12, 18, 25].

Reducing Search Space of Symbolic Execution. Homi is different from and orthogonal to the existing techniques [2, 3, 15, 27, 30, 31]. These techniques aim to *conservatively* prune redundant states based on some *predefined* criteria. On the other hand, Homi aims to *aggressively* prune the states based on *adaptive* criteria learned online during symbolic execution. The read-write set (RWset) analysis [2] aims to prune program paths that will execute the same basic blocks as previously explored paths. Likewise, the goal of post-conditioned symbolic execution [30, 31] is to discard the states having the same path suffixes as previously explored states during testing. Jaffar et al. [15] aims to subsume the paths guaranteed to be unreachable to the annotated assertions in the program. Chopper [27] presents a novel technique to perform symbolic execution while safely excluding the irrelevant functions in the program which are not the targets of users to test. Note that our tool, Homi, can further enhance symbolic execution by combining these techniques that safely prune redundant paths.

Combining Symbolic Execution with Learning. Our work aligns with this line of research that employs machine learning to boost symbolic execution [9–12, 18, 25]. ParadySE [9] presents a new approach to automatically generate search heuristics of symbolic execution via offline learning. Chameleon [11] is a novel symbolic execution that adaptively switches search heuristics for better performance via online learning. MLB [18] uses machine learning to effectively handle the complex path-conditions that involve external function calls or floating point arithmetic in symbolic execution. LEO [12] is a machine-learning based approach to boost symbolic execution by transforming the program under test into an easy-to-analyze program while preserving its semantics. ConTest [10] aims to learn useful templates that reduce the input space of the program under test by selectively generating symbolic variables during testing. On the other hand, we use a learning algorithm to aggressively prune unpromising states online during symbolic execution.

6 CONCLUSION

We present a new approach, Homi with the goal of maintaining promising states only via aggressive state-pruning. The key idea is to continuously learn the probabilistic pruning strategy based on the cumulative data during the testing period. Experimental results on 9 open-source C projects show that symbolic execution with Homi is able to notably increase branch coverage and find real bugs while keeping a relatively small set of states. We believe that minimizing candidate states in symbolic execution will emerge as a new solution against the state-explosion problem.

ACKNOWLEDGMENTS

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-51. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337, (SW STAR LAB) Research on Highly-Practical Automated Software Repair) and Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (2017M3C4A7068175).

REFERENCES

[1] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. 1–11.

[2] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. 351–366.

[3] Suhabe Bugrara and Dawson Engler. 2013. Redundant State Detection for Dynamic Symbolic Execution. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. 199–212.

[4] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. 443–446.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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 (OSDI '08)*. 209–224.

[6] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the 12th International Conference on Model Checking Software (SPIN'05)*. 2–23.

[7] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2 (2008), 10:1–10:38.

[8] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

[9] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 1244–1254.

[10] Sooyoung Cha, Seonho Lee, and Hakjoo Oh. 2018. Template-guided Concolic Testing via Online Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. 408–418.

[11] Sooyoung Cha and Hakjoo Oh. 2019. 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 (ESEC/FSE '19)*. 235–245.

[12] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP '18)*.

[13] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. 519–531.

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 213–223.

[15] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. 2013. Boosting Concolic Testing via Interpolation. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. 48–58.

[16] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. 85–103.

[17] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*. 689–701.

[18] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. 554–559.

[19] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA '13)*. 19–32.

[20] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 254–269.

[21] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic Execution with Existential Second-Order Constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. 389–399.

[22] Ivica Nikoliundefined, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. 653–663.

[23] Koushik Sen, Darko Marinov, and Gul Agha. 2005. 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 (ESEC/FSE '05)*. 263–272.

[24] Hyunmin Seo and Sunghun Kim. 2014. 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 (FSE '14)*. 413–424.

[25] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints.. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS '19)*.

[26] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. 109–119.

[27] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 350–360.

[28] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards Optimal Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 291–302.

[29] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan. 2015. DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15)*. 620–631.

[30] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2015. Postconditioned Symbolic Execution. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST '15)*. 1–10.

[31] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2018. Eliminating Path Redundancy via Postconditioned Symbolic Execution. *IEEE Transactions on Software Engineering* (2018), 25–43.