# Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics

Sooyoung Cha, Seongjoon Hong, Jiseong Bak, Jingyoung Kim, Junhee Lee, and Hakjoo Oh

**Abstract**—We present a technique to automatically generate search heuristics for dynamic symbolic execution. A key challenge in dynamic symbolic execution is how to effectively explore the program's execution paths to achieve high code coverage in a limited time budget. Dynamic symbolic execution employs a search heuristic to address this challenge, which favors exploring particular types of paths that are most likely to maximize the final coverage. However, manually designing a good search heuristic is nontrivial and typically ends up with suboptimal and unstable outcomes. The goal of this paper is to overcome this shortcoming of dynamic symbolic execution by automatically learning search heuristics. We define a class of search heuristics, namely a parametric search heuristic, and present an algorithm that efficiently finds an optimal heuristic for each subject program. Experimental results with industrial-strength symbolic execution tools (e.g., KLEE) show that our technique can successfully generate search heuristics that significantly outperform existing manually-crafted heuristics in terms of branch coverage and bug-finding.

**Index Terms**—Dynamic Symbolic Execution, Concolic Testing, Execution-Generated Testing, Search Heuristics, Software Testing

## 1 Introduction

Dynamic symbolic execution [1], [2], [3], [4] has emerged as an effective method for software testing with applications to diverse domains [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. The idea of classical symbolic execution [15], [16], [17], [18] is to run a program by using symbolic values as input and representing the values of variables in the program as symbolic expressions. Dynamic symbolic execution is a modern variant of classical symbolic execution, which combines symbolic and concrete execution to mitigate the inherent limitations of purely symbolic approaches to program execution (e.g., handling non-linear arithmetic, external code). There are two major flavors of dynamic symbolic execution [19], namely "concolic testing" [1], [2] and "execution-generated testing" [3], [4]. Both approaches have been used in several tools. For instance, CREST [20] and SAGE [21] are well-known concolic testing tools and KLEE [22] is a representative symbolic executor based on execution-generated testing.

Search heuristics are a key component of dynamic symbolic execution. Because of the path-explosion problem, it is infeasible for dynamic symbolic execution tools to explore all execution paths of any nontrivial programs. Instead, symbolic execution relies on a search heuristic to maximize its effectiveness in a limited time budget. A search heuristic has a criterion and guides symbolic execution to preferentially explore certain types of execution paths of the subject program according to its criterion. In CREST, for example, the CFDS (Control-Flow Directed Search) heuristic [20] is used to prioritize the execution paths that are close to the uncovered regions of the program and the CGS (Context-Guided Search) heuristic [23] prefers to explore paths in a new context. In KLEE, more than 10 search heuristics are implemented, one of which is the depth heuristic that prefers to explore the paths having the lowest number of executed branches. It is well-known that the choice of search heuristics critically affects the effectiveness of symbolic execution [8], [19], [20], [22], [23], [24], [25], [26], [27].

However, developing a good search heuristic remains an open challenge. Manually designing a search heuristic is not only nontrivial but also likely to deliver sub-optimal and unstable results. As we demonstrate in Section 2.3, no existing search heuristics consistently achieve good code coverage for real-world programs. For instance, in concolic testing, the CGS heuristic is arguably the state-of-the-art and outperforms existing approaches for a number of programs [23]. However, we found that CGS is sometimes brittle and inferior even to a random heuristic. Likewise, in execution-generated testing, the performance of each search heuristic significantly varies depending on the program under test (Figure 5). Furthermore, existing search heuristics came from a huge amount of engineering effort and domain expertise.

To address this challenge, we present ParaDySE, a new approach that automatically generates search heuristics optimized for each program under test. Unlike existing work that focused on manually developing a universal, program-independent search heuristic, we aim to automatically generate program-specific heuristics for each target program. To this end, we use two key ideas. First, we define a *parametric search heuristic*, which creates a large class of search heuristics based on common features of symbolic execution. The parametric heuristic reduces the problem of designing a good search heuristic into a problem of finding a good parameter vector. Second, we present a learning algorithm specialized for dynamic symbolic execution. The search space that the parametric heuristic poses is intractably large. Our learning algorithm effectively guides the search by iteratively refining the search space based on the feedback

• *S. Cha, S. Hong, J. Bak, J. Kim, J. Lee, and H. Oh are with the Department of Computer Science and Engineering, Korea University, Seoul, Korea. (Corresponding author: Hakjoo Oh.)*

from previous runs of dynamic symbolic execution.

Experimental results confirm the usefulness of our approach for a range of C programs. To demonstrate the effectiveness for both flavors of dynamic symbolic execution, we have implemented PARADYSE in CREST [20] (concolic testing) and KLEE [22] (execution-generated testing). For the former, we evaluated it on 10 open-source C programs (0.5–150KLoC). For the latter, we assessed it on 6 GNU open-source C programs (5–89KLoC). For every benchmark program, our technique has successfully generated a search heuristic that achieves considerably higher branch coverage than the existing state-of-the-art techniques. We also demonstrate that the increased coverage by our technique leads to more effective finding of real bugs. Lastly, we show that our approach remains useful when considering the overhead of learning a search heuristic for each program; the learned heuristic can be reused as the subject program evolves (Section 5.3.1) and our approach also performs better than existing techniques even in the training phase (Section 5.3.2).

### Contributions

In this paper, we make the following contributions:

- We present PARADYSE, a new approach for automatically generating search heuristics for dynamic symbolic execution. Our work represents a fundamental departure from prior work; while existing work (e.g. [8], [20], [22], [23], [24], [25]) focuses on manually developing a particular search heuristic, our goal is to automate the very process of generating such a heuristic.
- We present a parametric search heuristic and a learning algorithm for finding good parameter vectors.
- We extensively evaluate our approach with open-source C programs. We make our tool, called ParaDySE, and data publicly available.[1]

This paper is an extension of the previous work [26] presented at the 40th International Conference on Software Engineering (ICSE 2018). The major extensions are summarized as follows:

- The present paper describes our technique in a generalized setting and applies it to both approaches (concolic testing and execution-generated testing) to dynamic symbolic execution. The use of the previous technique [26] was limited to the concolic testing approach and it was not clear how to apply the idea to another major approach to dynamic symbolic execution (e.g., KLEE [22]). In Sections 2.2 and 3.2, we explain how to use our technique to the KLEE-style symbolic execution.
- In Section 5, we provide new experimental results with KLEE [22] and conduct additional experiments for CREST [20]. We also discuss limitations of our approach. For execution-generated testing, we make our tool, called dd-klee, and data publicly available.[2]

## 2 PRELIMINARIES

In this section, we describe two major approaches to dynamic symbolic execution, namely concolic testing (Section 2.1) and execution-generated testing (Section 2.2), and their limitations (Section 2.3).

1. https://github.com/kupl/ParaDySE
2. https://github.com/kupl/dd-klee

**Algorithm 1:** Concolic Testing

**Input :** Program $P$, budget $N$, initial input $v_0$
**Output:** The number of branches covered

1: $T \leftarrow \langle\rangle$
2: $v \leftarrow v_0$
3: **for** $m = 1$ to $N$ **do**
4: $\quad \Phi_m \leftarrow \mathsf{RunProgram}(P, v)$
5: $\quad T \leftarrow T \cdot \Phi_m$
6: $\quad$ **repeat**
7: $\quad\quad (\Phi, \phi_i) \leftarrow \mathsf{Choose}(T) \qquad (\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$
8: $\quad$ **until** $\mathsf{SAT}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
9: $\quad v \leftarrow \mathsf{model}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
10: **return** $|\mathsf{Branches}(T)|$

Both concolic testing and execution-generated testing attempt to mix concrete and symbolic execution. However, their specific mechanisms for combining the two executions are different; that is, concolic testing is driven by *concrete execution* and explores each program path one-by-one while execution-generated testing is driven by *symbolic execution* and explores the program paths simultaneously.

### 2.1 Concolic Testing

Concolic testing is a hybrid software testing technique that combines concrete and symbolic execution [1], [2]. DART [1], CUTE [2], SAGE [21], and CREST [20] are well-known concolic testing tools.

#### 2.1.1 Algorithm

The idea of concolic testing is to repeat the following process until a given time budget is exhausted: (1) a program is executed with an initial input; (2) the exercised path condition is collected during the concrete execution; and (3) a prefix of the path condition with the last branch condition negated is solved to generate the next input.

Concolic testing begins with executing the subject program $P$ with an initial input $v_0$. During the concrete execution, the technique maintains a *symbolic memory state* $S$ and a *path condition* $\Phi$. The symbolic memory is a mapping from program variables to symbolic values. It is used to evaluate the symbolic values of expressions. For instance, when $S$ is $[x \mapsto \alpha, y \mapsto \beta + 1]$ (variables $x$ and $y$ are mapped to symbolic expressions $\alpha$ and $\beta + 1$ where $\alpha$ and $\beta$ are symbols), the statement $z := x + y$ transfers the symbolic memory into $[x \mapsto \alpha, y \mapsto \beta + 1, z \mapsto \alpha + \beta + 1]$. The path condition represents the sequence of branches taken during the current execution of the program. It is updated whenever a conditional statement $if(e)$ is encountered. For instance, when $S = [x \mapsto \alpha]$ and $e = x < 1$, the path condition $\Phi$ gets updated to $\Phi \wedge (\alpha < 1)$.

Let $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ be the path condition that results from the initial execution. To obtain the next input value, concolic testing chooses a branch condition $\phi_i$ and generates the new path condition $\Phi'$ as follows: $\Phi' = (\bigwedge_{j<i} \phi_j) \wedge \neg\phi_i$. That is, the new condition $\Phi'$ has the same prefix as $\Phi$ up to the $i$-th branch with $\phi_i$ negated, so the input values that satisfy $\Phi'$ drive the program execution

to follow the opposite branch of $\phi_i$. Such concrete input values can be obtained from an SMT solver. This process is repeated until a fixed testing budget runs out.

Algorithm 1 presents a generic concolic testing algorithm[3]. The algorithm takes a program $P$, an initial input vector $v_0$, and a testing budget $N$ (i.e., the number of executions of the program). The algorithm maintains the execution tree $T$ of the program, which is the list of previously explored path conditions; note that an efficient implementation for the execution tree is to use a tree, but we represent the execution tree as a list for simplicity of Algorithm 1. The execution tree $T$ and input vector $v$ are initially empty and the initial input vector, respectively (lines 1 and 2). At line 4, the program $P$ is executed with the input $v$, resulting in the current execution path $\Phi_m$ explored. The path condition is appended to $T$ (line 5). In lines 6–8, the Choose function takes $T$ as input[4], and chooses a branch to negate; more precisely, the function first chooses a path condition $\Phi$ from $T$, then selects a branch, i.e., $\phi_i$, from $\Phi$. Once a branch $\phi_i$ is chosen, the algorithm generates the new path condition $\Phi' = (\bigwedge_{j<i} \phi_j) \wedge \neg\phi_i$. If $\Phi'$ is satisfiable, the next input vector is computed (line 9), where $\mathsf{SAT}(\Phi)$ returns true iff $\Phi$ is satisfiable and $\mathsf{model}(\Phi)$ finds an input vector $v$ which is a model of $\Phi$, i.e., $v \models \Phi$. Otherwise, if $\Phi'$ is unsatisfiable, the algorithm repeatedly tries to negate another branch until a satisfiable path condition is found; as an exception, if the algorithm totally fails to find any satisfiable conditions in $\Phi$, it escapes the inner loop at lines 6–8, and then executes the program with a random input at line 4. We omitted this exception handling in Algorithm 1 for its simplicity. Algorithm 1 repeats for the given budget $N$ and the final number of covered branches $|\mathsf{Branches}(T)|$ is returned.

#### 2.1.2 Search Heuristic

The performance of Algorithm 1 varies depending on the choice of the function Choose, namely a search heuristic. Since the number of execution paths in a program is usually exponential in the number of branches, exploring all possible execution paths is infeasible. To address this problem, concolic testing relies on the search heuristic that steers the execution in a way to maximize code coverage in a given limited time budget [19]. In prior work, the search heuristic was developed manually. Below, we describe three search heuristics [20], [23], which are known to perform comparatively better than others.

The most simple search heuristic, called Random Branch Search (Random) [20], is to randomly select a branch from the last execution path. That is, the Choose function in Algorithm 1 is defined as follows:

$$\mathsf{Choose}(\langle \Phi_1 \Phi_2 \cdots \Phi_m \rangle) = (\Phi_m, \phi_i)$$

3. Although Algorithm 1 generalizes existing search heuristics (e.g., CGS [23], CFDS [20]) but it does not cover some algorithm such as one implemented in SAGE [21]. Incorporating SAGE into Algorithm 1 makes the algorithm unnecessarily complicated.

4. For simplicity, though some search heuristics require additional arguments, we assumed they are passed to Choose implicitly. For example, when Choose corresponds to the CFDS heuristic, it additionally takes a control-flow graph as input; when corresponding to the CGS heuristic, it takes a dominator tree as input.

**Algorithm 2:** Execution-Generated Testing

**Input :** Program $P$, budget $N$
**Output:** The number of branches covered

1: $States \leftarrow \{(instr_0, M_0, true)\}$
2: $T \leftarrow \emptyset$
3: **repeat**
4: $\quad(instr, M, \Phi) \leftarrow \mathsf{Choose}(States)$
5: $\quad States \leftarrow States \setminus \{(instr, M, \Phi)\}$
6: $\quad(instr', M', \Phi) \leftarrow \mathsf{Execute}((instr, M, \Phi))$
7: $\quad$**if** $instr'$ = (if $(e)$ then $s_1$ else $s_2$) **then**
8: $\quad\quad$**if** $\mathsf{SAT}(\Phi \wedge e)$ **then**
9: $\quad\quad\quad States \leftarrow States \cup \{(s_1, M', \Phi \wedge e)\}$
10: $\quad\quad$**if** $\mathsf{SAT}(\Phi \wedge \neg e)$ **then**
11: $\quad\quad\quad States \leftarrow States \cup \{(s_2, M', \Phi \wedge \neg e)\}$
12: $\quad$**else if** $instr'$ = halt **then**
13: $\quad\quad T \leftarrow T \cup \mathsf{model}(\Phi)$
14: **until** budget $N$ expires or $States = \emptyset$
15:
16: **for all** $(instr, M, \Phi) \in States$ **do**
17: $\quad T \leftarrow T \cup \mathsf{model}(\Phi)$
18: **return** $|\mathsf{Coverage}(T)|$

where $\phi_i$ is a randomly chosen branch from $\Phi_m = \phi_1 \wedge \cdots \wedge \phi_n$. Although very simple, the Random heuristic is typically a better choice than simple deterministic heuristics such as DFS and BFS [20]. In our experiments, the Random heuristic was sometimes better than sophisticated techniques (Figure 3).

Control-Flow Directed Search (CFDS) [20] is based on the natural intuition that uncovered branches near the current execution path would be easier to be exercised in the next execution. This heuristic first picks the last path condition $\Phi_m$, then selects a branch whose opposite branch is the nearest from any of the unseen branches. The distance between two branches is calculated by the number of branches on the path from the source to the destination. To calculate the distance, CFDS uses control flow graph of the program, which is statically constructed before the testing.

Context-Guided Search (CGS) [23] performs the breath-first search (BFS) on the execution tree, while reducing the search space by excluding branches whose "contexts" are already explored. Given an execution path, the context of a branch in the path is defined as a sequence of preceding branches. During search, it gathers candidate branches at depth $d$ from the execution tree, picks a branch from the candidates, and the context of the branch is calculated. If the context has been already considered, CGS skips that branch and continues to pick the next one. Otherwise, the branch is negated and the context is recorded. When all the candidate branches at depth $d$ are considered, the search proceeds to the depth $d+1$ of the execution tree and repeats the process explained above.

### 2.2 Execution-Generated Testing

Another major flavor of dynamic symbolic execution is execution-generated testing [3], which has been implemented in popular symbolic execution tools such as EXE [4] and KLEE [22].

#### 2.2.1 Algorithm

Like concolic testing, the main idea of execution-generated testing is to mix concrete and symbolic execution. Unlike concolic testing, however, execution-generated testing is not driven by concrete execution. Instead, it works in a manner similar to classical symbolic execution and implicitly switches to concrete execution only when path conditions are unsolvable or the current instruction does not involve symbolic values.

Execution-generated testing basically maintains a set of states, where each state is a tuple $(instr, M, \Phi)$ of an instruction ($instr$) to evaluate next, a symbolic memory state ($M$), and a path condition ($\Phi$).[5] Similar to classical symbolic execution, execution-generated testing forks the execution whenever it evaluates a conditional statement.

Algorithm 2 shows an algorithm for execution-generated testing. The algorithm takes as input a program $P$ and a testing budget $N$ (e.g., 1 hour); unlike concolic testing (Algorithm 1), it does not take the initial input. At lines 1 and 2, the sets of explored states $States$ and generated test-cases $T$ are set to the initial state $(instr_0, M_0, true)$ and the empty set, respectively. For instance, consider the program:

```
1 void foo(int x){
2   if(x == 20)
3     assert("error"); }
```

The initial state for the program above is as follows:

$$(if(x{==}20), [x \mapsto \alpha], true) \quad (1)$$

where the first element represents the next instruction to evaluate, the symbolic state maps the argument $x$ to symbolic value $\alpha$, and the path condition is initially $true$. At line 4, using the procedure Choose, the algorithm selects a state to explore from the set $States$. At line 6, the algorithm executes the instruction $instr$ of the selected state, and returns the updated state $(instr', M', \Phi)$. If $instr'$ is a conditional statement (line 7), the algorithm checks whether the new path-conditions for the true (line 8) and false branches (line 10) are satisfiable. If both conditions are satisfiable, the algorithm forks the state into two states: $(s_1, M', \Phi \wedge e)$ and $(s_2, M', \Phi \wedge \neg e)$. For instance, the initial state in (1) is split into the states below:

$$\begin{aligned} state_1 &= (assert(\text{“}error\text{”}), [x \rightarrow \alpha], \alpha == 20) \\ state_2 &= (\mathsf{halt}, [x \rightarrow \alpha], \alpha \neq 20) \end{aligned}$$

When $instr'$ is the halt statement (e.g., exit), the algorithm generates a test-case which is a model of $\Phi$ of the state, and then adds it to the set $T$. The algorithm repeats the process described above until the time budget $N$ expires or the set $States$ is empty (line 14). Additionally, at lines 16–17, the algorithm generates test-cases using the path-conditions of states $States$, where the instruction of each state has not yet finished. Finally, using the test-cases $T$, the algorithm returns the number of covered branches (line 18).

We illustrated Algorithm 2 from the perspective that focuses on the role of search heuristic (Choose) in EGT. In this context, it is sufficient to treat EGT as classical symbolic execution. However, classical symbolic execution and execution-generated testing are different [19]. The key difference is that classical symbolic execution always fails to generate the test-case corresponding to the path-condition $(\Phi \wedge e)$ where $e$ involves a non-linear arithmetic that the constraint solver cannot handle. On the other hand, in the same situation, execution-generated testing is able to simplify the constraint $e$ by replacing some symbolic values by concrete ones. By doing so, execution-generated testing often succeeds in generating test-cases.

5. The definitions of the symbolic memory state and path condition are given in Section 2.1.1.

#### 2.2.2 Search Heuristic

Like concolic testing, the effectiveness of execution-generated testing depends on the choice of search heuristic, i.e., the Choose procedure in Algorithm 2. In this case, Choose is a function that takes a set of states and selects a state to explore next. Below, we describe one representative search heuristic, called RoundRobin, which is the default search heuristic of KLEE [22] and has been widely used in prior work (e.g., [22], [28], [29], [30], [31]).

The RoundRobin heuristic combines two search heuristics in a round robin fashion: Random-Path Search (Random-Path) and Coverage-Optimized Search (CovNew). The Random-Path heuristic selects a state by randomly traversing the execution tree on explored instructions of the subject program from the root. The leaves of the execution tree correspond to the candidate states to choose from, and the internal nodes denote the locations where the states forked. Compared to the purely random state selection heuristic (called Random-State), the Random-Path heuristic prioritizes the states located higher in the execution tree; the intuition is that the states with fewer constraints are easier to solve. The CovNew heuristic first calculates the weights of candidate states and then stochastically selects the state with high weight. The weight of each candidate is calculated by two factors; the first one is the minimum distance from the uncovered instructions, and the second one is the number of executed instructions since the heuristic most recently covered new instructions.

### 2.3 Limitations of Existing Search Heuristics

Existing search heuristics for both approaches of dynamic symbolic execution have a key limitation; they rely on a fixed strategy and fail to consistently perform well on a wide range of target programs. Our experience with these heuristics is that they are unstable and their effectiveness varies significantly depending on the target programs.

Figure 1 shows that no existing search heuristics perform consistently in concolic testing. The branch coverage achieved by each search heuristic fluctuates with subject programs. For example, the CFDS heuristic outperforms other existing heuristics for `vim-5.7` while the heuristic does not perform well for `expat-2.1.0`. Conversely, the CGS heuristic achieves the highest coverage for `expat-2.1.0`, but is inferior even to the random heuristic for `vim-5.7`. This is not a coincidence. Similar situations are observed in other programs (see Figure 3); for example, CGS is better than other heuristics for `grep-2.2`, but fails to beat the naive random heuristic on `tree-1.6.0`. That is, the main feature, *contexts*, of CGS is not appropriate for some programs such as `vim-5.7` and `tree-1.6.0`.

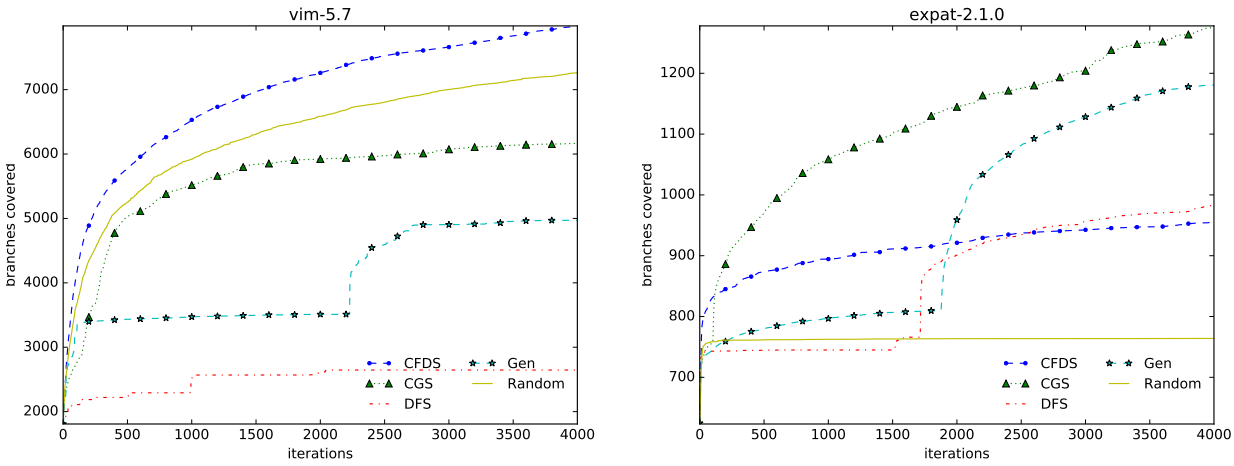



Fig. 1: Limitations of existing search heuristics for concolic testing: No search heuristic performs well consistently.

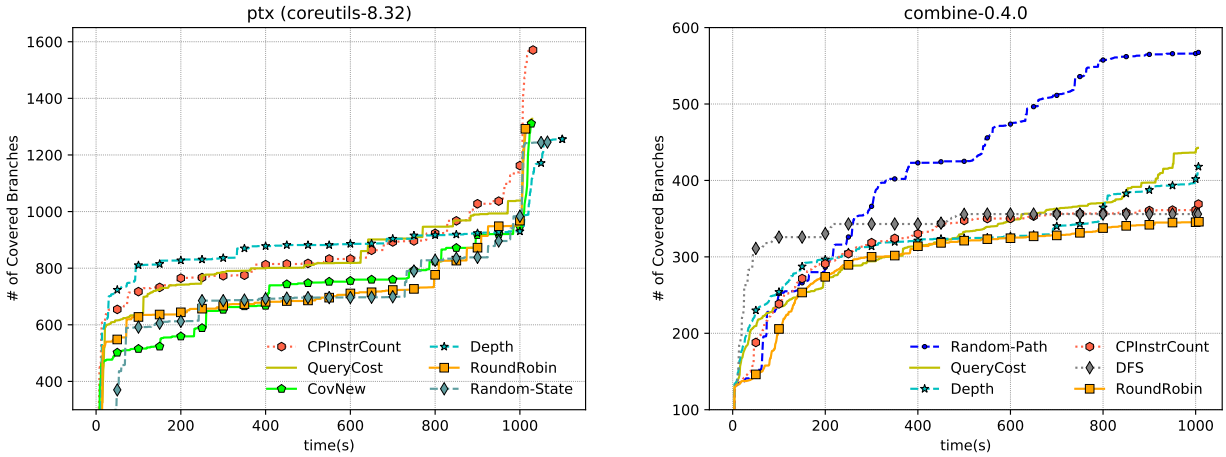



Fig. 2: Limitations of existing search heuristics for execution-generated testing (KLEE)

For execution-generated testing, we also obtained similar results. We evaluated 11 different search heuristics, including RoundRobin, implemented in KLEE [22] and Figure 2 shows the branch coverage achieved by the top 6 search heuristics for `ptx(coreutils-8.32)` and `combine-0.4.0`. The CPInstrCount heuristic was better than other search heuristics for `ptx` while the heuristic took the forth place for `combine-0.4.0`. On the other hand, the Random − Path heuristic succeeded in achieving the highest branch coverage for `combine-0.4.0`, but is not even ranked in the sixth place for `ptx`. More surprisingly, as we demonstrate in the experiments (Figure 5), when collecting the search heuristics with the highest coverage on each of 6 benchmark programs, we obtained 3 distinct heuristics. That is, for each program under test, the most effective search heuristic is likely to be different.

Besides their sub-optimality, another major limitation of existing approaches is that developing a good search heuristic requires a huge amount of engineering effort and expertise. Given that the effectiveness of dynamic symbolic execution depends heavily on the search heuristic, ordinary developers who lack the expertise on search heuristics cannot fully benefit from dynamic symbolic execution. These observations motivated us to develop a technique that generates search heuristics automatically.

## 3 Parametric Search Heuristic

Our first idea for automatically generating search heuristics is to define a parametric search heuristic, which defines a space of search heuristics from which our learning algorithm in Section 4.2 aims to choose the best one for each subject program.

In this section, we describe how we parameterize search heuristics for concolic testing (Section 3.1) and execution-generated testing (Section 3.2). The same idea is used for both approaches with slight variations due to the different types of search heuristics.

### 3.1 Parameterization for Concolic Testing

Let $P \in Program$ be a subject program under test. Recall that a search heuristic, the Choose function in Algorithm 1, is a function from execution trees to pairs of a path condition and a branch:

$$\mathsf{Choose} \in Heuristic_c = ExecutionTree \to PathCond \times Branch$$

where $ExecutionTree$ is the set of all execution trees of the program, $PathCond$ the set of all path conditions in the trees, $Branch$ the set of all branches in $P$. $Heuristic_c$ denotes the set of all search heuristics for concolic testing.

We define a family $\mathcal{H} \subseteq Heuristic_c$ of search heuristics as a parametric heuristic $\mathsf{Choose}_\theta$, where $\theta$ is the parameter which is a $k$-dimensional vector of real numbers: $\mathcal{H} = \{\mathsf{Choose}_\theta \mid \theta \in \mathbb{R}^k\}$. Given an execution tree $T = \langle \Phi_1, \Phi_2, \cdots, \Phi_m \rangle$, our parametric search heuristic is defined as follows:

$$\mathsf{Choose}_\theta(\langle \Phi_1, \cdots, \Phi_m \rangle) = (\Phi_m, \operatorname*{argmax}_{\phi_j \in \Phi_m} score_\theta(\phi_j))$$

Intuitively, the heuristic first chooses the last path condition $\Phi_m$ from the execution tree $T$, then selects a branch $\phi_j$ from $\Phi_m$ that gets the highest score among all branches in that path. Except for the CGS heuristic, all existing search heuristics choose a branch from the last path condition. In this work, we follow this common strategy but our method can be generalized to consider the entire execution tree as well. We explain how we score each branch $\phi$ in $\Phi_m$ with respect to a given parameter vector $\theta$:

1) We represent the branch by a feature vector. We designed 40 boolean features describing properties of branches in concolic testing. A feature $\pi_i$ is a boolean predicate on branches:

   $$\pi_i : Branch \to \{0, 1\}.$$

   For instance, one of the features checks whether the branch is located in the main function or not. Given a set of $k$ features $\pi = \{\pi_1, \ldots, \pi_k\}$, where $k$ is the length of the parameter vector $\theta$, a branch $\phi$ is represented by a boolean vector as follows:

   $$\pi(\phi) = \langle \pi_1(\phi), \pi_2(\phi), \ldots, \pi_k(\phi) \rangle.$$

2) Next we compute the score of the branch. In our method, the dimension $k$ of the parameter vector $\theta$ equals to the number of branch features. We use the simple linear combination of the feature vector and the parameter vector to calculate the branch:

   $$score_\theta(\phi) = \pi(\phi) \cdot \theta.$$

3) Finally, we choose a branch with the highest score. That is, among the branches $\phi_1, \ldots, \phi_n$ in $\Phi_m$, we choose branch $\phi_j$ such that $score_\theta(\phi_j) \geq score_\theta(\phi_k)$ for all $k$.

We have designed 40 features to describe useful properties of branches in concolic testing. Table 1 shows the features, which are classified into 12 static and 28 dynamic features. A static feature describes a branch property that can be extracted without executing the program. A dynamic feature requires to execute the program and is extracted during concolic testing. In the feature engineering phase, we did not aim to find statistical evidence of the features' usefulness. Instead, we simply tried to obtain as many features as possible and let the learning algorithm to identify the statistical importance of each feature. In Section 5.5, we discussed the usefulness of each feature based on the outcome of the learning algorithm.

The static features 1–12 describe the syntactic properties of each branch in the execution path, which can be generated by analyzing the program text. For instance, feature 8 indicates whether the branch has a pointer expression in its conditional expression. We designed these features to see how much such simple features help to improve branch coverage, as there is no existing heuristic that extensively considers the syntactic properties of branches. At first glance features 2 and 3 seem redundant, but not so. the role of branch continuing a loop is different from the one of branch exiting a loop; by giving a high score to branch continuing a loop we can explicitly steer concolic testing away from the loop (i.e. negating the branch) while giving a high score to a branch exiting a loop leads to getting into the loop.

On the other hand, we designed dynamic features (13–40) to capture the dynamics of concolic testing. For instance, feature 40 checks whether the branch is inside the most recently reached function. That is, during the execution of

TABLE 1: Branch features for concolic testing. Features 1–12 are static, and Features 13–40 are dynamic.

| # | Description |
|---|---|
| 1 | branch in the main function |
| 2 | branch continuing a loop |
| 3 | branch exiting a loop |
| 4 | nested branch |
| 5 | branch containing external function calls |
| 6 | branch containing integer expressions |
| 7 | branch containing constant strings |
| 8 | branch containing pointer expressions |
| 9 | branch containing local variables |
| 10 | branch inside a loop body |
| 11 | branch entering the switch case statement |
| 12 | branch entering the default case statement |
| 13 | first 10% branches of a path |
| 14 | last 10% branches of a path |
| 15 | branch appearing most frequently in a path |
| 16 | branch appearing least frequently in a path |
| 17 | branch newly covered in the previous execution |
| 18 | branch located right after the just-negated branch |
| 19 | branch whose context ($k = 1$) is already visited |
| 20 | branch whose context ($k = 2$) is already visited |
| 21 | branch whose context ($k = 3$) is already visited |
| 22 | branch whose context ($k = 4$) is already visited |
| 23 | branch whose context ($k = 5$) is already visited |
| 24 | branch negated more than 10 times |
| 25 | branch negated more than 20 times |
| 26 | branch negated more than 30 times |
| 27 | branch near the just-negated branch |
| 28 | branch failed to be negated more than 10 times |
| 29 | the opposite branch failed to be negated more than 10 times |
| 30 | the opposite branch is uncovered (depth 0) |
| 31 | the opposite branch is uncovered (depth 1) |
| 32 | branch negated in the last 10 executions |
| 33 | branch negated in the last 20 executions |
| 34 | branch negated in the last 30 executions |
| 35 | branch in the function that has the largest number of uncovered branches |
| 36 | the opposite branch belongs to unreached functions (top 10% of the largest func.) |
| 37 | the opposite branch belongs to unreached functions (top 20% of the largest func.) |
| 38 | the opposite branch belongs to unreached functions (top 30% of the largest func.) |
| 39 | the opposite branch belongs to unreached functions (# of branches $> 10$) |
| 40 | branch inside the most recently reached function |

the program, the boolean value of each dynamic feature for the same branch may change while the static feature values of the branch do not. Note that some of features seem arbitrary, but each feature has its own purpose. For instance, we designed the features 24–26 to learn how many times the same branch should be explored in terms of increasing code coverage. The threshold value, 10, in the feature 24 was determined by trial and error; for example, the value was initially 10% of the total number of executions, but it was not effective.

We also incorporated the key insights of the existing search heuristics into the features. For example, dynamic features 19–23 were designed based on the notion of contexts used in the CGS heuristic [23] while features 30–31 are based on the idea of the CFDS heuristic [20] that calculates the distance to uncovered branches.

### 3.2 Parameterization for Execution-Generated Testing

In execution-generated testing, note that a search heuristic, the Choose function in Algorithm 2 is a function from the power set of all states to the set of all states:

$$\mathsf{Choose} \in \mathit{Heuristic}_e = \wp(\mathit{States}) \to \mathit{States}$$

where $\mathit{States}$ denotes the set of all states in the program. We define $\mathit{Heuristic}_e$ as the set of all search heuristics for execution-generated testing. We also define a family $\mathcal{H}$ of search heuristics as a parametric heuristic $\mathsf{Choose}_\theta$, where $\mathcal{H}$ is as follows:

$$\mathcal{H} = \{\mathsf{Choose}_\theta \mid \theta \in \mathbb{R}^k\} \subseteq \mathit{Heuristic}_e$$

The parametric heuristic for execution-generated testing is defined as:

$$\mathsf{Choose}_\theta(S) = \operatorname*{argmax}_{s \in S} \mathit{score}_\theta(s).$$

That is, the parametric heuristic selects a state $s$ with the highest score from the set $S$ of states. Scoring each state with a given parameter vector $\theta$ is similar to the scoring function for concolic testing (Section 3.1). The difference is that we need features for describing properties of states instead of branches of path conditions. The scoring function $\mathit{score}_\theta$ works as follows:

1) It transforms each state in $S$ into a feature vector. We designed 26 boolean features describing properties of states in execution-generated testing. A feature $\pi_i$ is a boolean predicate on states:

   $$\pi_i : \mathit{States} \to \{0, 1\}.$$

   With the predefined 26 features, a state $s$ is represented by a boolean vector as follows:

   $$\pi(s) = \langle \pi_1(s), \pi_2(s), \ldots, \pi_{26}(s) \rangle.$$

2) Second, we compute the state score. The dimension of the parameter vector $\theta$ is equal to 26, the number of state features. Using the linear combination, We calculate the score as follows:

   $$\mathit{score}_\theta(\phi) = \pi(\phi) \cdot \theta.$$

3) We choose the state with the highest score from $S$.

We have designed 26 new features to describe useful properties of states in execution-generated testing. In the feature engineering phase, we tried to represent most of the useful information that KLEE [22] have already maintained in each state $(\mathit{instr}, M, \Phi)$ as features; that is, to reduce the effort for designing the features, we did not use any information beyond that maintained in KLEE.

Table 2 shows the features, which are systematically categorized into three folds: 8 features for the instruction ($\mathit{instr}$), 8 features for the symbolic memory ($M$), and 10 features for the path-condition ($\Phi$). For each feature, we marked the 10% states satisfying a specific property among the total states as 1 while marking the rest as 0. First, the features 1–8 describe the information about the instructions executed in each state. Specifically, the information includes the number of executed instructions (#1–2), the number of instructions executed while satisfying a specific condition (#3–6), the distance to uncovered instructions (#7–8).

Second, the features 9–16 describe the useful information about the symbolic memory of the state. The memory information includes the address space (#9–10), the number of variables mapped with concrete values (#13–14) and symbolic expressions (#15–16). Features 11 and 15 (or Features 12 and 16) seem similar, but not so. For instance, when the initial symbolic memory $M$ is $[x \mapsto \alpha]$, both the number of initially defined symbolic variables and variables mapped with symbolic expressions are 1. However, after executing the statement $y := x + 3$, the memory $M$ is transferred into $[x \mapsto \alpha, y \mapsto \alpha{+}3]$. Then, the latter number becomes 2 while the former number remains 1. That is, while performing symbolic execution, the latter number keeps changing, but the former number does not change.

Lastly, the features 17–26 contain the information of the path-condition in the state. More concretely, for each path-condition, the features describe the solving cost (#17–18), the number of exercised branches (#19–20), and the loop depth (#23–26). Note that feature 19 looks similar to feature 21, but the number of exercised branches (#19) is always greater than or equal to the number of symbolic constraints (#21). This is because the former includes both exercised concrete and symbolic branches while the latter only includes the symbolic branches.

The designed features also reflected the key insights of the effective search heuristics implemented in KLEE [22]. Specifically, the instruction features 1–8 were inspired from the four search heuristics: InstrCount (#1–2), CovNew (#3–4), MinDistance (#5–6) and CPInstrCount (#7–8). Second, the path-condition features 17–18 and features 19–20 are based on the idea of the QueryCost and Depth heuristics, respectively.

## 4 Parameter Learning Algorithm

Now we describe our algorithm for finding a good parameter value of the parametric search heuristic in Section 3. We define the optimization problem, and then present our algorithm. Our optimization algorithm is general and can be used for both approaches to dynamic symbolic execution.

### 4.1 Optimization Problem

In our approach, finding a good search heuristic corresponds to solving an optimization problem. We model dy-

TABLE 2: State features for execution-generated testing.

| # | Description |
|---|---|
| 1 | 10% states with the smallest number of executed instructions |
| 2 | 10% states with the highest number of executed instructions |
| 3 | 10% states with the smallest number of executed instructions since the last instruction was newly covered |
| 4 | 10% states with the highest number of executed instructions since the last instruction was newly covered |
| 5 | 10% states with the smallest number of executed instructions in currently executing function |
| 6 | 10% states with the highest number of executed instructions in currently executing function |
| 7 | 10% states closest to the uncovered instructions |
| 8 | 10% states farthest from the uncovered instructions |
| 9 | 10% states with the smallest global/heap address space |
| 10 | 10% states with the largest global/heap address space |
| 11 | 10% states with the smallest number of initially defined symbolic variables |
| 12 | 10% states with the highest number of initially defined symbolic variables |
| 13 | 10% states with the smallest number of variables mapped with concrete values |
| 14 | 10% states with the highest number of variables mapped with concrete values |
| 15 | 10% states with the smallest number of variables mapped with symbolic expressions |
| 16 | 10% states with the highest number of variables mapped with symbolic expressions |
| 17 | 10% states with the lowest query solving cost |
| 18 | 10% states with the highest query solving cost |
| 19 | 10% states with the smallest number of exercised branches |
| 20 | 10% states with the highest number of exercised branches |
| 21 | 10% states having the smallest number of symbolic constraints |
| 22 | 10% states having the highest number of symbolic constraints |
| 23 | 10% states with the shallowest loop depth in currently executing function |
| 24 | 10% states with the deepest loop depth in currently executing function |
| 25 | 10% states with the shallowest loop depth in the caller of currently executing function |
| 26 | 10% states with the deepest loop depth in the caller of currently executing function |

namic symbolic execution algorithms (i.e., Algorithm 1 and Algorithm 2) by the function:

$$\mathcal{S} : \mathit{Program} \times \mathit{Heuristic} \to \mathbb{N}$$

where $\mathit{Program}$ is the set of all programs, and $\mathit{Heuristic}$ corresponds to $\mathit{Heuristic}_c$ in Section 3.1 when performing concolic testing, and $\mathit{Heuristic}_e$ in Section 3.2 when performing execution-generated testing. Intuitively, the function $\mathcal{S}$ takes a program and a search heuristic, and returns the number of covered branches. Given a program $P$ and a search heuristic $\mathsf{Choose}$, $\mathcal{S}(P, \mathsf{Choose})$ performs dynamic symbolic execution using the heuristic for a fixed testing budget (i.e. $N$). We assume that the initial input ($v_0$) for concolic testing and the testing budget ($N$) are fixed for each subject program.

Given a program $P$ to test, our goal is to find a parameter vector $\theta$ that maximizes the performance of $\mathcal{S}$ with respect to $P$. Formally, our objective is to find $\theta^*$ such that

$$\theta^* = \underset{\theta \in \mathbb{R}^k}{\operatorname{argmax}} \mathcal{S}(P, \mathsf{Choose}_\theta). \quad (2)$$

That is, we aim to find a parameter vector $\theta^*$ that causes the function $\mathcal{S}$ with the search heuristic $\mathsf{Choose}_\theta$ to maximize the number of covered branches in $P$.

### 4.2 Optimization Algorithm

We propose an algorithm that efficiently solves the optimization problem in (2). A simplistic approach to solve the problem would be the random sampling method defined as follows:

1: initialize the sample space $\mathbb{S} = [-1, 1]$
2: **repeat**
3: $\quad\theta \leftarrow$ sample from $\mathbb{S}^k$
4: $\quad B \leftarrow \mathcal{S}(P, \mathsf{Choose}_\theta)$
5: **until** timeout
6: **return** best $\theta$ found

which randomly samples parameter vectors and returns the best parameter vector found for a given time budget. However, we found that this naive algorithm is extremely inefficient and leads to a failure when it is used for finding a good search heuristic (Section 5.4). This is mainly because of two reasons. First, the search space is intractably large and therefore blindly searching for good parameter vectors without any guidance is hopeless. Second, a single evaluation of a parameter vector is generally unreliable and does not represent the average performance in dynamic symbolic execution. For example, the performance of concolic testing can vary due to the inherent nondeterminism (e.g. branch prediction failure) [2].

In response, we designed an optimization algorithm (Algorithm 3) specialized to efficiently finding good parameter vectors of search heuristics. The key idea behind this algorithm is to iteratively refine the sample space based on the feedback from previous runs of dynamic symbolic execution. The main loop of the algorithm consists of the three phases: *Find*, *Check*, and *Refine*. These three steps are repeated until the average performance converges.

At line 2, the algorithm initializes the sample spaces. It maintains $k$ sample spaces, $\mathbb{S}_i$ ($i \in [1, k]$), where $k$ is the dimension of the parameter vectors (i.e., the number of features in our parametric heuristic). In our algorithm, the $i$-th components of the parameter vectors are sampled from $\mathbb{S}_i$, independently from other components. For all $i$, $\mathbb{S}_i$ is initialized to the space $[-1, 1]$.

In the first phase (*Find*), we randomly sample $n$ parameter vectors: $\theta_1, \theta_2, \ldots, \theta_n$ from the current sample space $\mathbb{S}_1 \times \mathbb{S}_2 \times \cdots \times \mathbb{S}_k$ (line 7), and their performance numbers (i.e., the number of branches covered) are evaluated (lines 9–11). In experiments, we set $n$ depending on the given program $P$ (Table 10). Among the $n$ parameter vectors, we choose the top $K$ parameter vectors according to their branch coverage. In our experiments, $K$ is set to 10 because we observed that parameter vectors with good qualities are usually found in

**Algorithm 3:** Parameter Optimization Algorithm

**Input :** Program $P$
**Output:** Optimal parameter $\theta \in \mathbb{R}^k$ for $P$

```
1:  /* k: the dimension of θ */
2:  initialize the sample spaces S_i = [−1, 1] for i ∈ [1, k]
3:  ⟨max, converge⟩ ← ⟨0, false⟩
4:  repeat
5:    /* Step 1: Find */
6:    /* sample n parameters: θ_1, ..., θ_n (e.g., n=1,000) */
7:    {θ_i}_{i=1}^n ← sample from S_1 × S_2 × ··· × S_k
8:    /* evaluate the sampled parameters */
9:    for i = 1 to n do
10:     /* B_i: branch coverage achieved with θ_i */
11:     B_i ← S(P, Choose_{θ_i})
12:   pick top K parameters {θ'_i}_{i=1}^K from {θ_i}_{i=1}^n with
      highest B_i
13:
14:   /* Step 2: Check */
15:   for all K parameters θ'_i do
16:     B*_i ← (1/T) Σ_{j=1}^T S(P, Choose_{θ'_i})
17:   pick top 2 parameters θ_{t1}, θ_{t2} with highest B*_i
18:
19:   /* Step 3: Refine */
20:   for i = 1 to k do
21:     if θ^i_{t1} > 0 and θ^i_{t2} > 0 then
22:       S_i = [min(θ^i_{t1}, θ^i_{t2}), 1]
23:     else if θ^i_{t1} < 0 and θ^i_{t2} < 0 then
24:       S_i = [-1, max(θ^i_{t1}, θ^i_{t2})]
25:
26:   /* Check Convergence */
27:   if B*_{t1} < max then
28:     converge ← true
29:   else
30:     ⟨max, θ_max⟩ ← ⟨B*_{t1}, θ_{t1}⟩
31: until converge
32: return θ_max
```

the top 10 parameter vectors. This first step of performing the symbolic execution function $n$ times (line 11) can be run in parallel.

In the next phase (*Check*), we choose the top 2 parameter vectors that show the best average performance. At lines 15–16, the $K$ parameter vectors chosen from the first phase are evaluated again to obtain the average code coverage over $T$ trials, where $B_i^*$ represents the average performance of parameter vector $\theta_i'$. In experiments, we set $T$ to at least 5 to 10 depending on the cost of a single evaluation (e.g, $\mathcal{S}(P, \mathsf{Choose}_{\theta_i})$). At line 17, we choose two parameter vectors $\theta_{t_1}$ (top 1) and $\theta_{t_2}$ (top 2) with the best average performance. This step (*Check*) is needed to rule out unreliable parameter vectors. Because of the nondeterminism of dynamic symbolic execution, the quality of a search heuristic must be evaluated over multiple executions.

In the third step (*Refine*), we refine the sample spaces $\mathbb{S}_1, \ldots, \mathbb{S}_k$ based on $\theta_{t_1}$ and $\theta_{t_2}$. Each $\mathbb{S}_i$ is refined based on the values of the $i$-th components ($\theta^i_{t_1}$ and $\theta^i_{t_2}$) of $\theta_{t_1}$ and $\theta_{t_2}$. When both $\theta^i_{t_1}$ and $\theta^i_{t_2}$ are positive, we modify $\mathbb{S}_i$ by $[\min(\theta^i_{t_1}, \theta^i_{t_2}), 1]$. When both $\theta^i_{t_1}$ and $\theta^i_{t_2}$ are negative, $\mathbb{S}_i$ is refined by $[-1, \max(\theta^i_{t_1}, \theta^i_{t_2})]$. Otherwise, $\mathbb{S}_i$ remains the same. Then, our algorithm goes back to the first phase (*Find*) and randomly samples $n$ parameter vectors from the refined space.

Finally, our algorithm terminates when the best average coverage ($B^*_{t_1}$) obtained in the current iteration is less than the coverage ($max$) from the previous iteration (lines 27–28). This way, we iteratively refine each sample space $\mathbb{S}_i$ and guide the search to continuously find and climb the hills toward top in the parameter space.

TABLE 3: Benchmark programs for concolic testing (CREST)

| Program | # Total branches | LOC |
|---|---|---|
| vim-5.7 | 35,464 | 165K |
| gawk-3.0.3 | 8,038 | 30K |
| expat-2.1.0 | 8,500 | 49K |
| grep-2.2 | 3,836 | 15K |
| sed-1.17 | 2,656 | 9K |
| tree-1.6.0 | 1,438 | 4K |
| cdaudio | 358 | 3K |
| floppy | 268 | 2K |
| kbfiltr | 204 | 1K |
| replace | 196 | 0.5K |

TABLE 4: Benchmark programs for execution-generated testing (KLEE)

| Program | # Total branches | LOC |
|---|---|---|
| gcal-4.1 | 15,799 | 89K |
| combine-0.4.0 | 2,357 | 32K |
| trueprint-5.4 | 2,518 | 12K |
| du (coreutils-8.32) | 6,653 | 8K |
| ptx (coreutils-8.32) | 5,270 | 6K |
| ls (coreutils-8.32) | 3,776 | 5K |

## 5 Experiments

In this section, we experimentally evaluate our approach, called ParaDySE. We demonstrate the effectiveness of ParaDySE for both flavors of dynamic symbolic execution: concolic testing and execution-generated testing. For the former, we implemented our approach in CREST [32], a concolic testing tool widely used for C programs [20], [23], [27], [33]. For the latter, we implemented ParaDySE in KLEE[6] [22], one of the most popular symbolic execution tools widely used in previous work [27], [34], [35], [36], [37], [38], [39]. We conducted experiments to answer the following research questions:

- **Effectiveness of generated heuristics**: Does our approach generate effective search heuristics for dynamic symbolic execution? How do they perform compared to the existing state-of-the-art heuristics?
- **Cost for learning heuristics**: How long does our approach take to learn search heuristics? Is our approach useful in practice even considering the learning cost?
- **Efficacy of learning algorithm**: How does our learning algorithm perform compared to the naive algorithm by random sampling?
- **Important features**: What are the important features to generate effective search heuristics for both approaches to dynamic symbolic execution?

All experiments were done on a Linux machine with two Intel Xeon Processors E5-2630 and 192GB RAM; the machine has a total of 16 cores and 32 threads. Note that although we conducted all experiments on a Linux with 192GB RAM, the average RAM usage per thread was not so high (e.g., 4GB).

6. We used KLEE-2.1 released in March 2020.

### 5.1 Evaluation Setting

#### 5.1.1 Concolic Testing

We have compared our approach with pure random testing (PureRandom) and concolic testing with the following five existing search heuristics: CGS (Context-Guided Search) [23], CFDS (Control-Flow Directed Search) [20], RandomBranch (Random Branch Search) [20], DFS (Depth-First Search) [2], and Gen (Generational Search) [21]. We chose these heuristics for comparison because they have been commonly used in prior work [2], [20], [21], [23], [40]. In particular, CGS and CFDS are arguably the state-of-the-art search heuristics that often perform the best in practice [20], [23]. We obtained the implementations of CFDS, RandomBranch, PureRandom, and DFS heuristics from CREST without any modification. The implementations of CGS and Gen came from the prior work [23].[7] We clarify that "RandomBranch" performs concolic testing with a search heuristic that randomly selects a branch to negate while "PureRandom" is a random testing that generates test cases without leveraging the path information. Thus, PureRandom is much faster than concolic testing as it does not require to manipulate path conditions on-the-fly (see Section 5.2.1).

We used 10 open-source benchmark programs (Table 3). The benchmarks are divided into the large and small programs. The large benchmarks include `vim`, `expat`, `grep`, `sed`, `gawk`, and `tree`.[8] These are well-known benchmark programs in concolic testing for C, which have been used multiple times in prior work [20], [22], [23], [41], [42], [43], [44]. Our benchmark set also includes 4 small ones: `cdaudio`, `floppy`, `kbfiltr`, and `replace`, which were used in [20], [23], [42].

We conducted all experiments under the same evaluation setting; the initial input (i.e. $v_0$ in Algorithm 1) was fixed for each benchmark program and a single run of concolic testing used the same testing budget (4,000 executions, i.e., $N = 4,000$ in Algorithm 1). We used the number of executions as the testing budget because the previous works on search heuristics [20], [23] used the number of executions instead of execution time. However, it may not be the best decision, so we also report results based on the execution time for two large programs.

Note that the performance of concolic testing also depends on the initial input. We found that in our benchmark programs, except for `grep` and `expat`, different choices of initial input did not much affect the final performance, so we generated random inputs for those programs. For `grep` and `expat`, the performance of concolic testing varied significantly depending on the initial input. For instance, with some initial inputs, CFDS and RandomBranch covered 150 less branches in `grep` than with other inputs. We avoided this exceptional case when selecting the input for `grep` and `expat`. For `expat`, we chose the input used in prior work [23]. For `grep`, we selected an input value on which the RandomBranch heuristic was reasonably effective. The initial input values we used are available with our tool.

The performance of each search heuristic was averaged over multiple trials. Even with the same initial input, the search heuristics have coverage variations for several reasons: search initialization in concolic testing [2], the randomness of search heuristics, and so on. We repeated the experiments 100 times for all benchmarks except for `vim` for which we averaged over 50 trials as its execution takes much longer time.

#### 5.1.2 Execution-Generated Testing

We have compared our approach with 11 existing search heuristics implemented in KLEE [22]: DFS (Depth-First Search), BFS (Breath-First Search), Random-State, Random−Path, CovNew, QueryCost, MinDistance (Minimal-Distance to Uncovered), Depth, InstrCount (Instruction-Count), CPInstrCount (CallPath-Instruction-Count), and RoundRobin using Random − Path and CovNew in a round robin fashion.

We used 6 GNU open-source C programs as benchmarks (Table 4). First, these benchmarks include the three largest programs in GNU Coreutils-8.32. We used GNU Coreutils as it is the most commonly used benchmark for evaluating KLEE (e.g., [25], [30], [35], [36], [37], [38], [45]). We excluded small programs (e.g., `cat`, `rm`, and `pwd`) in Coreutils as the existing search heuristics already achieve high branch coverage on those programs, as their sizes are quite small (e.g., `pwd` is of 0.4KLoC). Second, we also include the three other GNU benchmarks, `gcal`, `combine`, and `trueprint`, so that our experimental results are not specific to the programs in Coreutils.

We used the same evaluation settings in all experiments. We allocated 1,000 seconds for testing budget (i.e., $N = 1,000s$ in Algorithm 2). We used the maximum running time as the testing budget instead of the number of program executions. This is because, unlike concolic testing, execution-generated testing does not involve the time to actually execute the program with the generated inputs; hence, previous works on KLEE [22], [25], [27], [35] used the time budget for evaluation. We repeated the experiments for all benchmarks 10 times and reported the average branch coverage over 10 trials because of the randomness of search heuristics.

### 5.2 Effectiveness of Generated Heuristics

For each benchmark program in Table 3 and 4, we ran our algorithm (Algorithm 3) to generate our search heuristic (OURS)[9], and compared its performance with that of the existing heuristics in both approaches of dynamic symbolic execution. We evaluate the effectiveness in terms of branch coverage. For concolic testing, we also compare the heuristics in terms of bug detection.

#### 5.2.1 Concolic Testing

For branch coverage, we measured the average and maximum coverages. The average branch coverage is obtained by averaging the results over the 100 trials (50 for `vim`). The maximum coverage refers to the highest coverage achieved during the 100 trials (50 for `vim`). The former indicates the

7. We obtained the implementation from authors via personal communication.

8. We omitted the program version when there is no confusion.

9. OURS denotes our automatically-generated search heuristic.

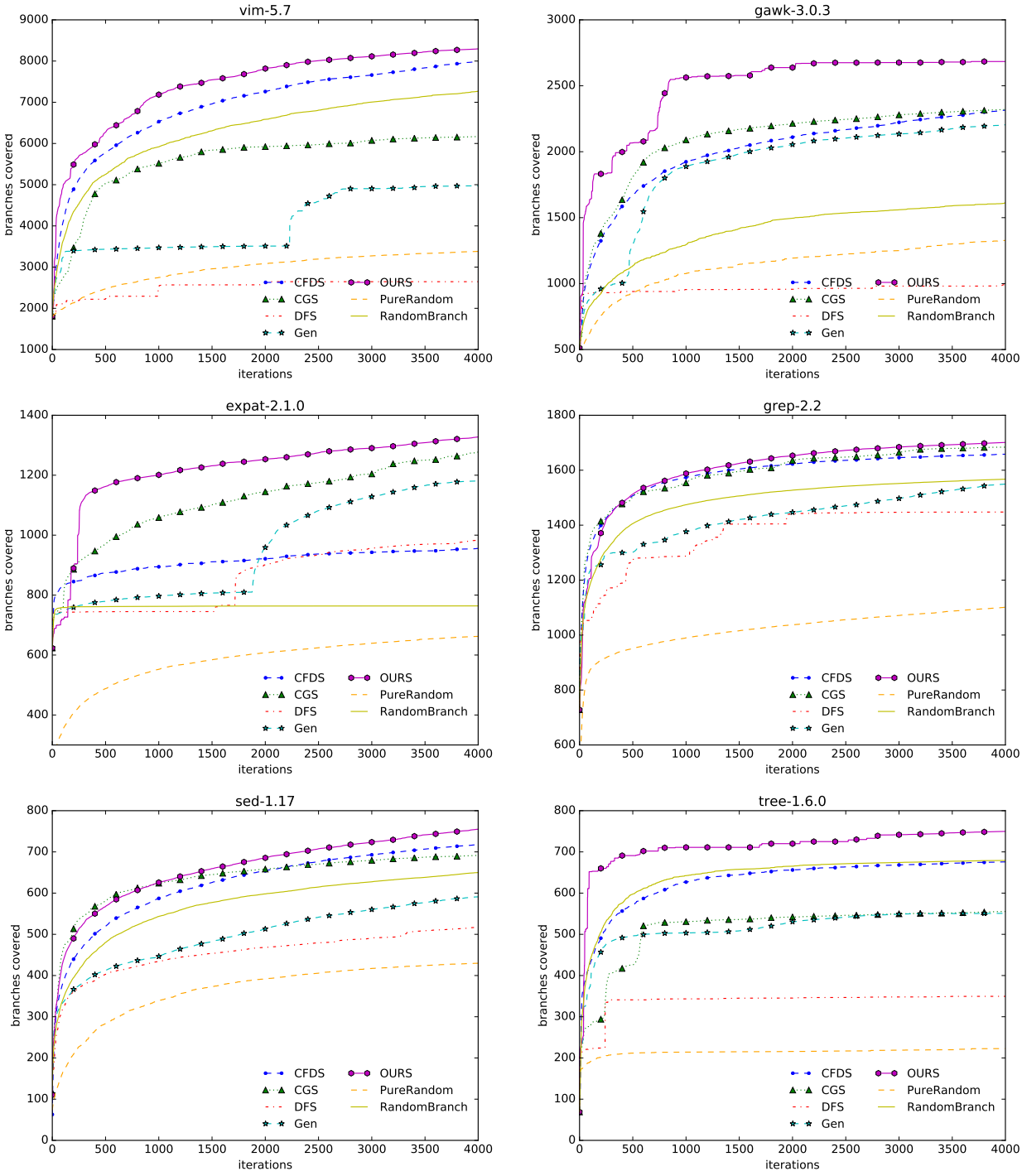



Fig. 3: Average branch coverage achieved by each search heuristic on 6 large benchmarks for concolic testing

average performance while the latter the best performance achievable by each heuristic.

Figure 3 compares the average branch coverage achieved by different search heuristics and pure random testing on 6 large benchmarks in Table 3. The results show that the search heuristics generated by our approach (OURS) achieve the best coverage on all programs. In particular, OURS significantly increased the branch coverage on the two largest benchmarks: `vim` and `gawk`. For `vim`, OURS covered 8,297 branches in 4,000 executions while the CFDS heuristic, which took the second place for `vim`, covered 7,990 branches. Note that CFDS is already highly tuned and therefore outperforms the other heuristics for `vim` (for instance, CGS covered 6,166 branches only). For `gawk`, OURS covered 2,684 branches while the CGS heuristic, the second best one, managed to cover 2,321 branches. For `expat`, `sed`, and `tree`, our approach improved the existing heuristics considerably. For example, OURS covered 1,327 branches for `expat`, increasing the branch coverage of CGS by 50. For `grep`, OURS also performed the best followed by CGS and

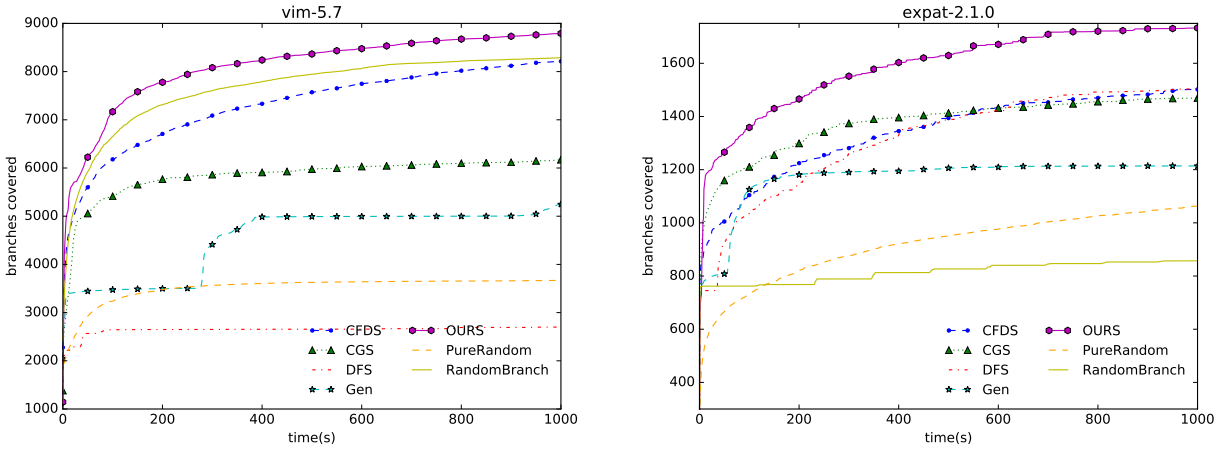



Fig. 4: Performance w.r.t. execution time on vim-5.7 and expat-2.1.0

CFDS. On small benchmarks, we obtained similar results; OURS (together with CGS) consistently achieved the highest average coverage (Table 5). In the rest of the paper, we focus only on the 6 large benchmarks, where existing manually-crafted heuristics fail to perform well.

On all benchmarks in Figure 3, OURS exclusively covered branches that were not covered by other heuristics. For example, in `vim`, a total of 504 branches were exclusively covered by our heuristic. For other programs, the numbers are: `expat`(14), `gawk`(7), `grep`(23), `sed`(21), `tree`(96).

These results are statistically significant; on all benchmark programs in Figure 3, the $p$ value was less than $0.01$ according to Wilcoxon signed-rank test. Table 6 also shows the standard deviations for each search heuristic in Figure 3. The standard deviation of our heuristic varies depending on each benchmark program, but overall it is similar to the standard deviations of the existing heuristics.

Although we focused on comparing the effectiveness of search heuristics over iterations (# of executions) in Figure 3, we conducted the additional experiments over execution time. Given the same time budget (1,000 sec), Figure 4 compares the average branch coverage achieved by each heuristic and pure random testing on the two large benchmarks: `vim` and `expat`. The results show that the coverage gap between our heuristic and the second best heuristic is greater than the ones in Figure 3. For example, given the same time budget, OURS and RandomBranch (the second best) covered 8,947 and 8,272 branches, respectively, for `vim`. Note that on the real-world program, it is very challenging to cover about 700 more branches on average within 1000 seconds. The results were averaged over 50 trials.

Figure 3 and 4 also show that pure random testing (PureRandom) is much inferior to several search heuristics, including OURS, on 6 large benchmarks even though PureRandom is generally faster than concolic testing with search heuristics. For instance, during the same time budget (1,000s), the average number of executions for PureRandom on `vim` is 6.1 times larger than the one for CGS. For another example, the average number for Gen on `expat` is 7,462 while the number for PureRandom is 86,950; that is, PureRandom is 11.7 times faster than concolic testing with the Gen heuristic, but it does not lead to a higher coverage. Figure 3 and 4 show that PureRandom is worse than concolic testing right from the start. This is because most of the inputs generated by PureRandom fail to pass through the input-validation routines of each application program, failing to executing the core functionality of the program. The input validation routines often involved equality conditions (e.g., `if (input == 52)`), for which random testing is unlikely to generate satisfying inputs. For instance, for `tree-1.6.0` in Figure 3, about 99.3% of inputs generated by PureRandom were invalid (e.g., "error opening dir") while only 15.1% were invalid in concolic testing with our search heuristic.

Table 7 compares the heuristics in terms of the maximum branch coverage on 6 large benchmarks. The results show that our approach in this case also achieves the best performance on all programs. For instance, in `vim`, we considerably increased the coverage of CFDS, the second best strategy; OURS covered 8,788 branches while CFDS managed to cover 8,585. For `expat`, OURS and CGS (the second best) have covered 1,422 and 1,337 branches, respectively.

Note that there is no clear winner among the existing search heuristics. In Figure 3, except for OURS, CFDS took the first place for `vim` and `sed` in terms of average branch coverage. For `gawk`, `expat`, and `grep`, the CGS heuristic was the best. For `tree`, the RandomBranch heuristic was better than CFDS and CGS. In terms of the maximum branch coverage, CFDS was better than the others for `vim` and `gawk` while CGS was for `grep` and `sed`. The Gen and RandomBranch heuristics surpassed CFDS and CGS in `expat` and `tree`, respectively.

We found that our approach also leads to more effective finding of real bugs (not seeded ones). Table 8 reports the number of trials that successfully generate test-cases, which trigger the known performance bugs in `gawk` and `grep` [46], [47]. During the 100 trials (where a single trial consists of 4,000 executions), our heuristic always found the bug in `gawk` while all the other heuristics completely failed to find it. In `grep`, OURS succeeded to find the bug 47 times out of 100 trials, which is much better than CGS does (5 times). Other heuristics were not able to trigger the bug at all.

Our heuristics are good at finding bugs because they

TABLE 5: Average branch coverage on 4 small benchmarks

| | **OURS** | CFDS | **CGS** | RandomBranch | Gen | DFS |
|---|---|---|---|---|---|---|
| cdaudio | **250** | **250** | **250** | 242 | **250** | 236 |
| floppy | **205** | **205** | **205** | 170 | **205** | 168 |
| replace | **181** | 177 | **181** | 174 | 176 | 171 |
| kbfiltr | **149** | **149** | **149** | **149** | **149** | 134 |

TABLE 6: Standard deviations on 6 large benchmarks

| | OURS | CFDS | CGS | RandomBranch | Gen | DFS |
|---|---|---|---|---|---|---|
| vim | 258.26 | 251.48 | 197.01 | 285.98 | 187.23 | 0.00 |
| expat | 41.47 | 43.98 | 24.32 | 48.40 | 95.68 | 20.47 |
| gawk | 0.00 | 120.16 | 56.92 | 216.81 | 115.14 | 9.64 |
| grep | 51.22 | 33.08 | 28.68 | 17.16 | 27.84 | 2.52 |
| sed | 21.65 | 23.65 | 26.83 | 17.49 | 42.72 | 17.02 |
| tree | 7.42 | 12.50 | 15.44 | 9.29 | 15.89 | 2.79 |

are much better than other heuristics in exercising diverse program paths. We observed that other heuristics such as CGS, CFDS, and Gen also covered the branches where the bugs originate. However, note that even though several heuristics are able to cover the same branches, those heuristics can easily fail to generate the bug-triggering inputs. This is because in the real-world program, there are thousands of execution paths passing through the branch where the bug originated. However, among those candidate paths, only a few paths can cause the bugs. Table 8 shows that our heuristic has often succeeded in preferentially exploring such few paths compared to the existing heuristics.

We remark that we did not specially tune our approach towards finding those bugs. In fact, we were not aware of the presence of those bugs at the early stage of this work. The bugs in `gawk` and `grep` [46], [47] cause performance problems; for example, `grep-2.2` requires exponential time and memory on particular input strings that involve back-references [47]. During concolic testing, we monitored the program executions and restarted the testing procedure when the subject program ran out of memory or time. Those bugs were detected unexpectedly by a combination of this mechanism and our search heuristic.

#### 5.2.2 Execution-Generated Testing

PARADYSE also succeeded in generating the most effective search heuristic for each program in Table 4, compared to all 11 search heuristics implemented in KLEE. Here, we

TABLE 7: Effectiveness in terms of maximum coverage

| | **OURS** | CFDS | CGS | RandomBranch | Gen | DFS |
|---|---|---|---|---|---|---|
| vim | **8,788** | 8,585 | 6,488 | 8,143 | 5,161 | 2,646 |
| expat | **1,422** | 1,060 | 1,337 | 965 | 1,348 | 1,027 |
| gawk | **2,684** | 2,532 | 2,449 | 2,035 | 2,443 | 1,025 |
| grep | **1,807** | 1,726 | 1,751 | 1,598 | 1,640 | 1,456 |
| sed | **830** | 780 | 781 | 690 | 698 | 568 |
| tree | **797** | 702 | 599 | 704 | 600 | 360 |

TABLE 8: Effectiveness in terms of finding bugs (concolic testing)

| | **OURS** | CFDS | CGS | RandomBranch | Gen | DFS |
|---|---|---|---|---|---|---|
| gawk | **100/100** | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 |
| grep | **47/100** | 0/100 | 5/100 | 0/100 | 0/100 | 0/100 |

TABLE 9: Effectiveness in terms of finding bugs (execution-generated testing)

| | Crash | OURS | CPInstrCount | RandomPath | DFS | BFS | InstrCount | QueryCost | Depth |
|---|---|---|---|---|---|---|---|---|---|
| gcal | SIGSEGV | 10/10 | 10/10 | 10/10 | 10/10 | 10/10 | 10/10 | 9/10 | 10/10 |
| | SIGABRT | 10/10 | 9/10 | 9/10 | 0/10 | 10/10 | 10/10 | 3/10 | 10/10 |
| combine | SIGABRT | 10/10 | 10/10 | 10/10 | 10/10 | 10/10 | 7/10 | 10/10 | 10/10 |

calculated the branch coverage with respect to running time as follows:

1) While running KLEE on a program, we recorded the creation time of each test-case generated by Algorithm 2. This step produces the data $D = \{(T_i, t_i)\}_{i=1}^{M}$, where $T_i$ is a test-case, $t_i$ is its creation time ($t_j < t_k$ if $j < k$), and $M$ is the number of generated test-cases.
2) When the time budget expires, we re-ran the original binary of the program with each test-case $T_i$ in order. We computed the accumulated branch coverage $c_i$ of $T_i$ including the branches covered by all preceding test-cases $T_j (j < i)$. We plotted the data $\{(t_i, c_i)\}_{i=1}^{M}$ to depict the coverage graph. To measure the branch coverage, we used `gcov`, a well-known tool for analyzing code coverage.

Figure 5 shows the average branch coverage over the 10 trials achieved by top 6 heuristics on the 6 benchmarks. In particular, our automatically-generated heuristic (OURS) significantly increased the average branch coverage for the largest benchmark `gcal`; OURS covered 1,822 branches while the second best heuristic (CPInstrCount) only covered 1,603 branches. That is, OURS is able to cover about 219 more branches than the second best one on average during the same time period. For GNU Coreutils, OURS also outperformed the existing heuristics; OURS succeeded in increasing the number of covered branches by 192 and 144, compared to the second best heuristic of each benchmark: CovNew (`du`) and Random − Path (`ls`).

One interesting point in execution-generated testing is that there is a significant increase in branch coverage at the end of the testing. For example, Figure 5 shows that all the search heuristics, including OURS, suddenly increase the branch coverage on `gcal`, `ptx`, and `ls` when the testing budget (1,000s) expires. This is due to the test-cases generated after the testing budget is over; more precisely, it is caused by lines 16–17 of Algorithm 2. The results indicate that the remaining states that have not yet reached the end of the subject program (e.g., halt instruction in Algorithm 2) contribute significantly to increasing branch coverage.

As with concolic testing, there is no obvious winner among 11 search heuristics for execution-generated testing. Except for OURS, there are the 3 distinct heuristics which achieve the highest branch coverage at least in one of the 6 benchmarks; the Random − Path heuristic succeeded in achieving the highest branch coverage for `combine`, `trueprint`, and `ls`, but it was not included in the top-6 heuristics for `ptx`. On the other hand, CovNew and CPInstrCount took the first place for `du` and `ptx`, respectively. The best heuristic (CovNew) for `du`, however, was not even ranked in the sixth place for all the remaining benchmarks except `ptx`. These results on the existing heuristics for dynamic symbolic execution demonstrate our claim that manually-crafted heuristics are likely to be suboptimal and

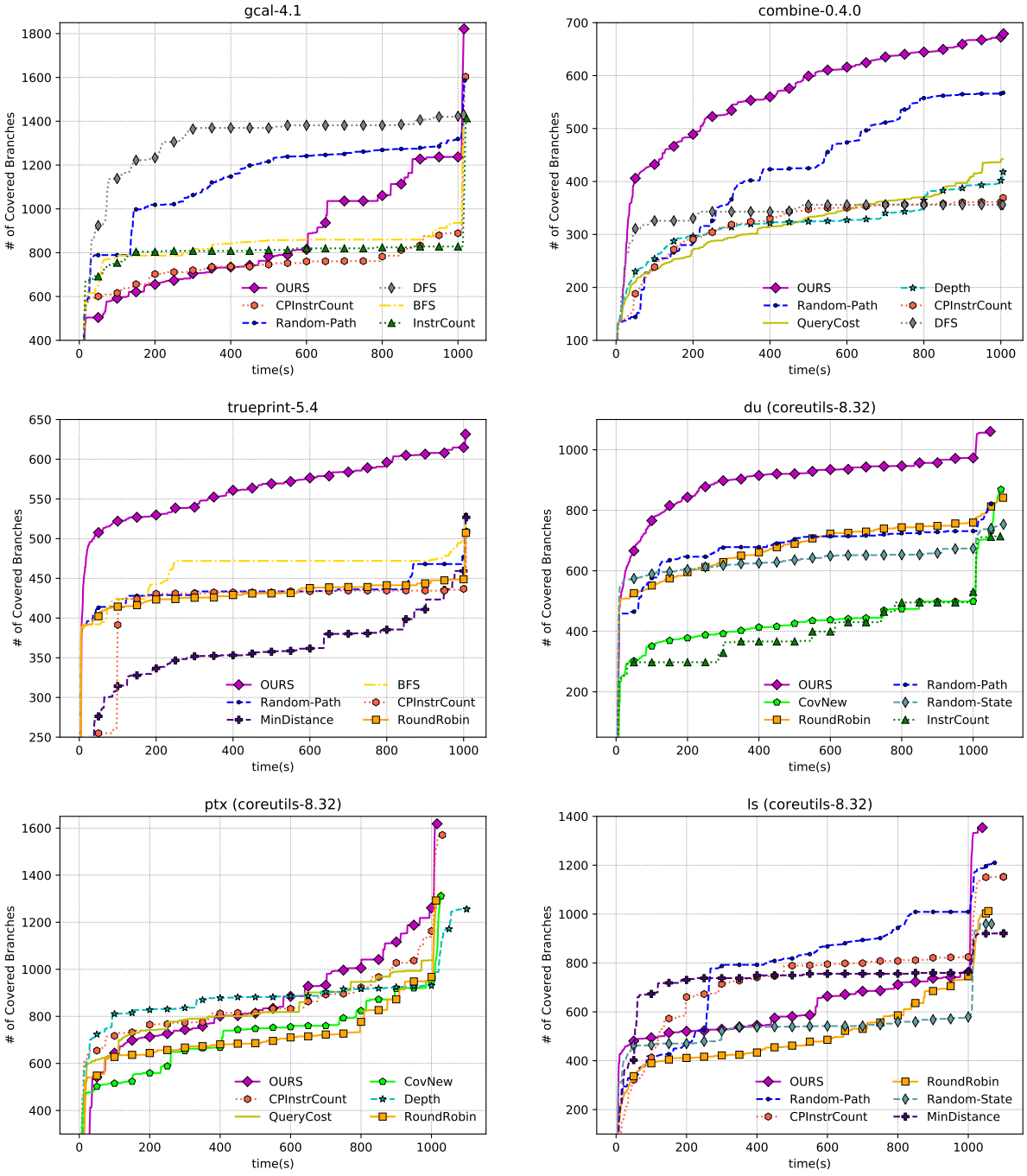



Fig. 5: Average branch coverage achieved by top 6 search heuristics on 6 large benchmarks for execution-generated testing

unstable. On the other hand, our approach, ParaDySE, is able to consistently produce the best search heuristics for both approaches of dynamic symbolic execution in terms of branch coverage.

Our heuristic (OURS) for execution-generated testing also consistently found real bugs such as segmentation fault (SIGSEGV) and abnormal termination (SIGABRT). Table 9 reports the number of trials that successfully generated test-cases, which cause the program to crash. During the 10 trials (where a single trial consists of 1,000 seconds), OURS (together with BFS and Depth) always generated bug-triggering inputs in `gcal` and `combine` while other search heuristics were unstable in finding these bugs; for example, the DFS heuristic totally failed to generate test-cases that cause abnormal termination of `gcal`.

### 5.3 Time for Obtaining the Heuristics

Table 10 reports the running time of our algorithm to generate the search heuristics that were evaluated in Section 5.2. To obtain our heuristics, we ran the optimization algorithm (Algorithm 3) in parallel using 20 threads. For concolic testing, in the first phase ('Find') of the algorithm, we sampled 1,000 parameter vectors, where each thread is responsible for evaluating 50 parameter vectors. For `vim`, we set the

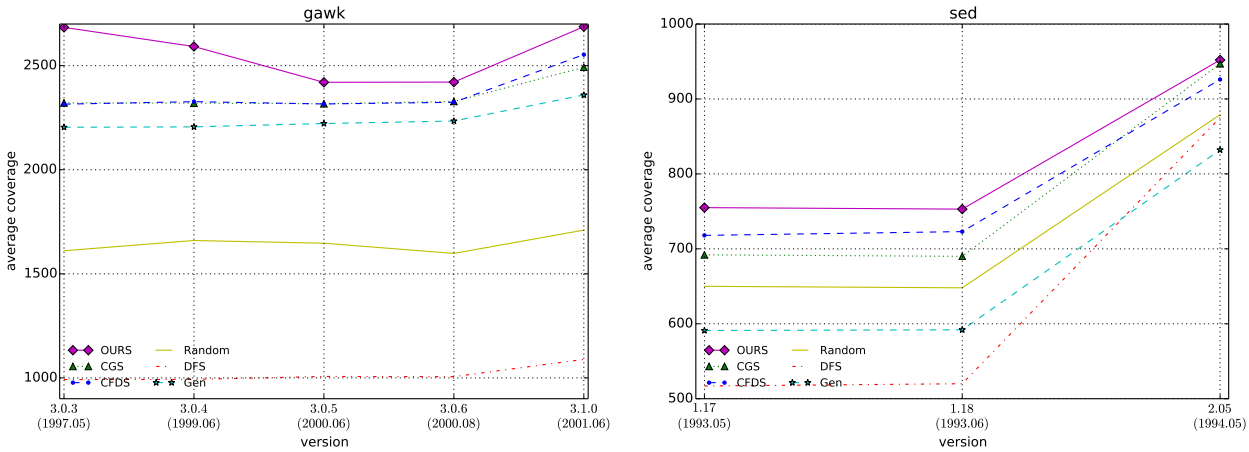



Fig. 6: Average coverage of each search heuristic for concolic testing on multiple subsequent program variants

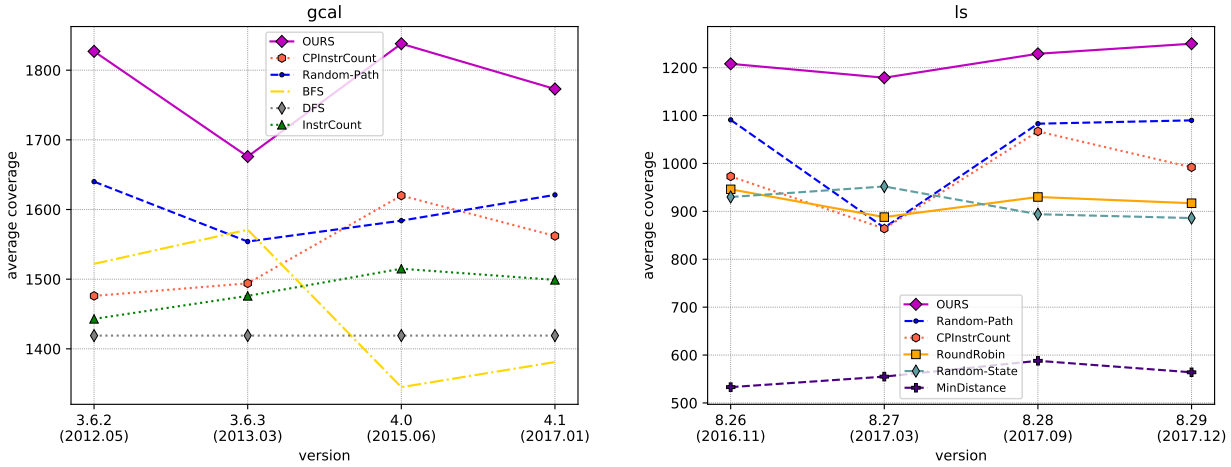



Fig. 7: Average coverage of each search heuristic for execution-generated testing on multiple subsequent program variants

TABLE 10: Time for generating the heuristics

| Benchmarks | # Sample | # Iteration | Total times |
|---|---|---|---|
| vim-5.7 | 300 | 5 | 24h 17min |
| expat-2.1.0 | 1,000 | 6 | 10h 25min |
| gawk-3.0.3 | 1,000 | 4 | 6h 28min |
| grep-2.2 | 1,000 | 5 | 5h 26min |
| sed-1.17 | 1,000 | 4 | 8h 55min |
| tree-1.6.0 | 1,000 | 4 | 3h 17min |
| gcal-4.1 | 300 | 4 | 20h 18min |
| combine-0.4.0 | 300 | 7 | 35h 15min |
| trueprint-5.4 | 300 | 8 | 40h 17min |
| du | 300 | 5 | 25h 58min |
| ptx | 300 | 4 | 20h 34min |
| ls | 300 | 3 | 15h 39min |

sample size to 300 as executing `vim` is expensive. For execution-generated testing, we equally fixed the sample size to 300 because a single evaluation (e.g, $\mathcal{S}(P, \text{Choose}_{\theta_i})$) is also expensive, where it took 1,000 seconds. The results show that our algorithm converges within 2–8 iterations of the outer loop of Algorithm 3, taking 3–40 hours depending on the size of the subject program and the number of iterations.

Our approach requires training effort but it is rewarding because the learned heuristic can be reused multiple times over a long period of time as the subject program evolves. Moreover, we show that our approach enables effective concolic testing even in the training phase; for execution-generated testing, we also demonstrate the practical benefit of our approach by comparing to the existing heuristics while involving the time of generating search heuristics.

#### 5.3.1 *Reusability of Learned Heuristics*

The learned heuristics for both approaches of dynamic symbolic execution can be reused over multiple subsequent program variations. To validate this hypothesis in concolic testing, we trained a search heuristic on `gawk-3.0.3` and applied the learned heuristic to the subsequent versions until `gawk-3.1.0`. We also trained a heuristic on `sed-1.17` and applied it to later versions. Figure 6 shows that the learned heuristics manage to achieve the highest branch coverage over the evolution of the programs. For example, OURS covered at least 90 more branches than the second best heuristic (CFDS) in all variations between `gawk-3.0.3` and `gawk-3.1.0`. The effectiveness lasted for at least 4 years for `gawk` and 1 year for `sed`.

For execution-generated testing, we trained a search heuristic on `gcal-3.6.2` (2012), and applied the learned

heuristic to the subsequent versions from `gcal-3.6.3` (2013) to `gcal-4.1` (2017). For another benchmark `ls` in Coreutils-8.26 (2016), we also trained the search heuristic, and re-used it to the subsequent, more precisely 3, versions from Coreutils-8.26 to 8.29. Figure 7 shows that the learned heuristics succeeded in achieving the highest branch coverage over all versions of `gcal` and `ls`; the results were averaged over 10 trials. For `gcal`, the difference in the number of covered branches between OURS and the second best heuristic is at least 105 and up to 218 on average. An interesting point is that the existing heuristics are likely to unstable even across different versions of the same program `gcal`; except for OURS, the Random − Path and BFS heuristics are the best one on `gcal-3.6.2` and `gcal-3.6.3`, respectively. For `gcal-4.0`, the another heuristic, CPInstrCount, becomes the winner. Likewise, for the different 4 versions of `ls`, OURS succeeded in covering at least 117 and up to 227 more branches. The usefulness of each learned heuristic continued for 5 years on `gcal` and 1 year on `ls`.

To show the reusability in a different perspective, we checked whether the learned heuristic worked well even when performing concolic testing with the different number of symbolic variables on vim-5.7. Originally, using the array with 20 symbolic values, we learned the heuristic on vim-5.7. To check the reusability of the heuristic, during the same time budget (1,000s), we compared the performance of our heuristic with the ones of the existing heuristics depending on the number of symbolic variables: 10, 15, 25, and 30. Figure 8 shows that our heuristic consistently achieves the highest branch coverage even over the different number of symbolic variables. The result also shows that we achieve the best performance in the number of symbolic variables (20) actually used for learning, but the performance decreases as the number of symbolic variables increases. That is, our learned heuristics are basically specialized for certain settings (e.g., # of symbolic variables, time budget), but still useful in other settings.

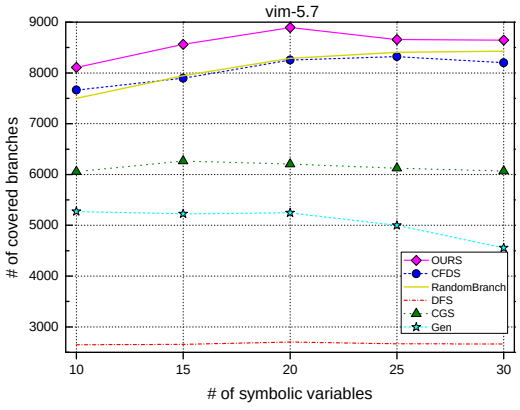



Fig. 8: Average coverage of each search heuristic on multiple symbolic variables. (Originally, we learned the heuristic with 20 symbolic variables)

### 5.3.2 *Practical Benefit of Our Approach*

Next, we show that our approach can be useful in practice, even considering the learning cost.

TABLE 11: Effectiveness in the training phase

| | **OURS** | CFDS | CGS | RandomBranch | Gen | DFS |
|---|---|---|---|---|---|---|
| `vim` | **14,003** | 13,706 | 7,934 | 13,835 | 7,290 | 2,646 |
| `expat` | **2,455** | 2,339 | 2,157 | 1,325 | 2,116 | 2,036 |
| `gawk` | **3,473** | 3,382 | 3,261 | 3,367 | 3,302 | 1,905 |
| `grep` | **2,167** | 2,024 | 2,016 | 2,066 | 1,965 | 1,478 |
| `sed` | 1,019 | 1,041 | **1,042** | 1,007 | 979 | 937 |
| `tree` | **808** | 800 | 737 | 796 | 730 | 665 |

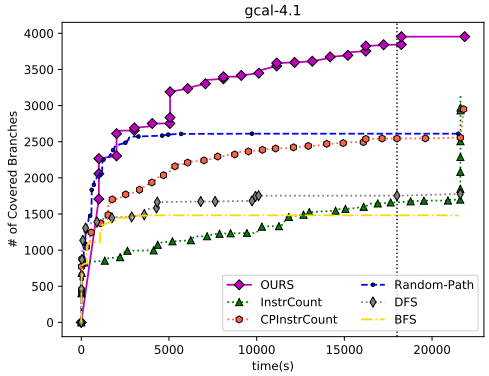
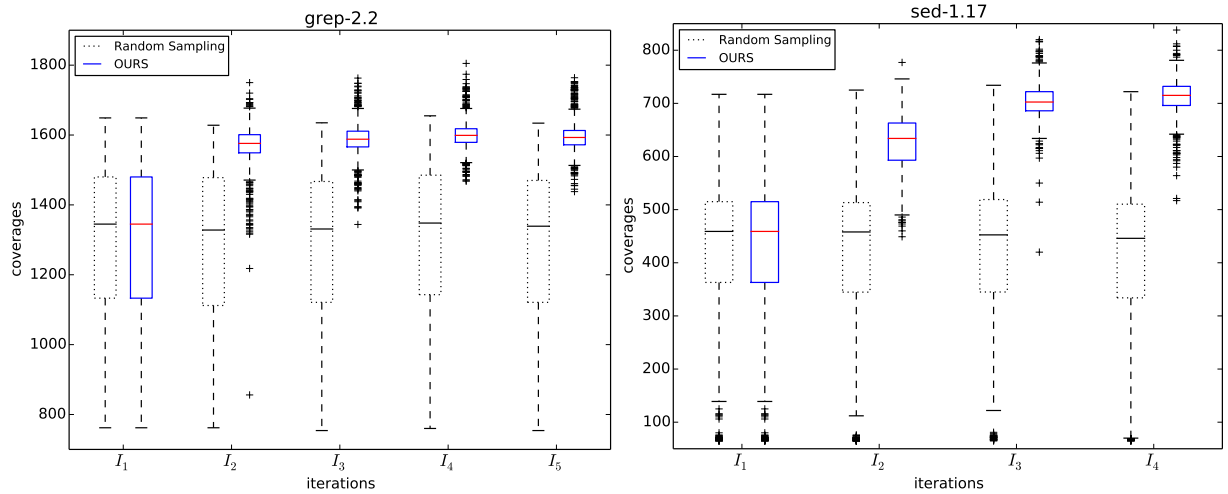


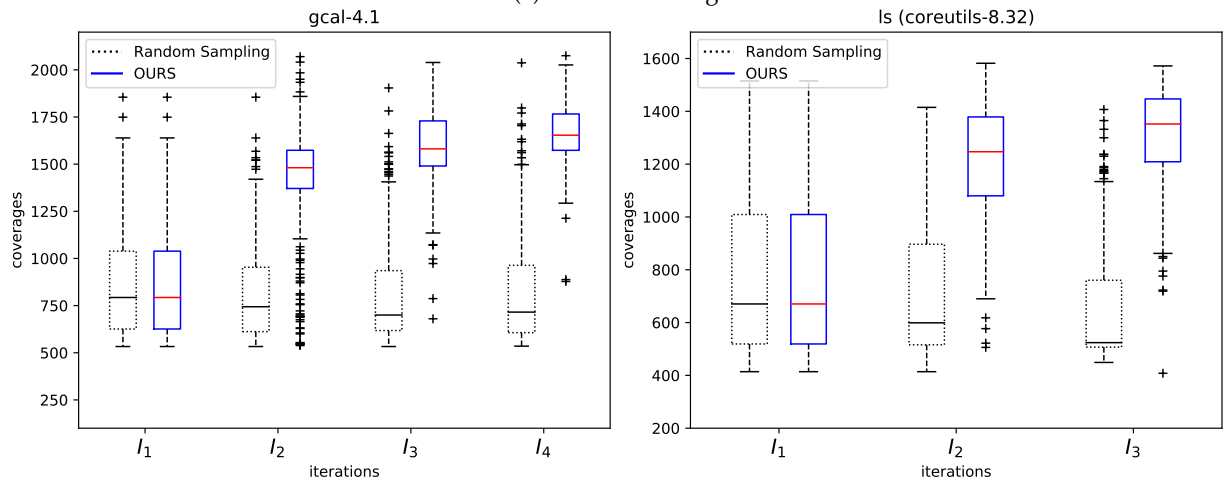

Fig. 9: Comparison with the existing heuristics when including the generation time on gcal-4.1 (the vertical dotted line indicates the generation time, 18,000s.)

First, the learning phase itself can be an effective testing process in practice. Note that running Algorithm 3 is essentially running dynamic symbolic execution on the subject program. Thus, we compared the number of branches covered during this training phase in concolic testing with the branches covered by other search heuristics given the same time budget reported in Table 10. Table 11 compares the results: except for `sed`, running Algorithm 3 achieves greater branch coverage than others. To obtain the results for other heuristics, we ran concolic testing (with $N = 4,000$) repeatedly using the same number of threads and amount of time. For instance, in 24 hours, Algorithm 3 covered 14,003 branches of `vim` while concolic testing with the CFDS and CGS heuristics covered 13,706 and 7,934 branches, respectively.

Second, for execution-generated testing, we check that our approach is still beneficial in practice, even considering the time of generating search heuristics. To do so, in parallel using 20 threads, we set 5 hours to learn the search heuristic for the largest program, `gcal`, and then applied the best heuristic learned in 5h, to the program for 1 hour; that is, we allocated for a total of 6 hours and 20 threads as a testing budget. Given the exactly same resources (6h and 20 threads) to the existing heuristics, we compared the number of branches covered by our approach with the ones covered by the existing heuristics. Figure 9 demonstrates that our approach outperforms the existing heuristics, even taking into account the generating time (5h=18,000s). Up to about 4,000 seconds, the performance of Random − Path and OURS were similar, but from then on, the branch coverage difference gradually increased. This indicates that our approach outperforms the existing search heuristics (e.g., Random − Path) even when our approach




(a) Concolic testing




(b) Execution-generated testing

Fig. 10: Comparison of our learning algorithm and random sampling method

is still trying to learn the best heuristic; in other word, the process of learning the best heuristic itself is also valuable. When the total budget, 6h, expires, OURS succeeded in covering a total of 3,952 branches while the second best heuristic (InstrCount) covered 3,135 branches. Note that when involving the generating time on `gcal`, the coverage difference between OURS and InstrCount is much larger compared to the difference in Figure 5 when only including the testing time (1,000s).

### 5.4 Efficacy of Learning Algorithm

We compared the performance of our optimization algorithm (Algorithm 3) with a naive approach based on random sampling when generating search heuristics. Because both approaches involve randomness, we statistically compare the qualities of parameter vectors found by our algorithm and the random sampling method. We conducted the comparison on `grep` and `sed` for concolic testing, and `gcal` and `ls` for execution-generated testing.

Figure 10 shows the distributions of final coverages achieved by those two algorithms instantiated in concolic testing and execution-generated testing, respectively. More specifically, in Figure 10, the height of the box denotes the interquartile range. The lower whisker is at the lowest data above $Q_1$ - 1.5*($Q_3$ - $Q_1$), and the upper whisker is at the highest data below $Q_3$ + 1.5*($Q_3$ - $Q_1$), where $Q_1$ and $Q_3$ denote the first and third quartiles; outliers denotes data beyond the whiskers. First, for `grep` and `sed`, our algorithm required a total of 1,100 trials of concolic testing to complete a single refinement task: 100 trials for the Check phase to select top 2 parameter vectors and the rest for the Find phase to evaluate the parameter vectors generated from the refined space. We compared the distributions throughout each iteration ($I_1, I_2, ..., I_N$) where 1,100 trials were given as budget for finding parameter vectors. Second, for `gcal` and `ls`, our algorithm needed a total of 350 trials of execution-generated testing as a single refinement task: 300 trials (the Find phase) and 50 trials (the Check phase). That is, we compared the distributions for each iteration with 350 trials. In both approaches of dynamic symbolic execution, the first refinement task of our algorithm begins with the initial samples in the first iteration $I_1$, which are prepared by random sampling method.

Figure 10a and 10b show that our algorithm is much superior to random sampling method for both approaches

of dynamic symbolic execution: (1) the median of the samples increases while (2) the interquartile range decreases, as the refinement task in our algorithm goes on. The median value (the band inside a box) of the samples found by our algorithm increases as the refinement task continues, while random sampling has no noticeable changes. The increase of median indicates that the probability to find a good parameter vector grows as the tasks repeat. In addition, the interquartile range (the height of the box, in simple) in our algorithm decreases gradually, which implies that the mix of *Check* and *Refine* tasks was effective.

We remark that use of our optimization algorithm was critical; in concolic testing, the heuristics generated by random sampling failed to surpass the existing heuristics. For instance, for `grep`, our algorithm (Algorithm 3) succeeded in generating a heuristic which covered 1,701 branches on average. However, the best one by random sampling covered 1,600 branches only, lagging behind CGS (the second best) by 83 branches.

## 5.5 Important Features

### 5.5.1 Top-k Features

We discuss the relative importance of features by analyzing the learned parameter vector $\theta$ for each program in Table 3 and 4. Intuitively, when the $i$-th component $\theta^i$ has a negative number in $\theta$, it indicates that the branch (or state) having $i$-th component should not be selected to be explored. Thus, both strong negative and positive features are equally important for our approach to improve the branch coverage. Table 12 and Table 13 show the top-$k$ positive and negative features for concolic testing and execution-generated testing, respectively; depending on the total number of features, we set $k$ to 10 and 5 for the former and the latter.

The results show that there is no winning feature which always belongs to the top-$k$ positive or negative features. Nevertheless, we mark the features that are used only as positive ones in at least three benchmarks as '(+)'. On the other hand, we mark the features that are used only as negative ones in at least three benchmarks as '(-)'. For instance, for concolic testing, the features 13 (front parts of a path) and 30-31 (distances of uncovered branches) are comparatively consistent positive ones. For 4 benchmarks, the feature 11 (case statement), 22 (context) and 26 (frequently negated branch) are included in the top 10 negative features. For designing effective search heuristics, the key ideas of CFDS heuristic (#30-31) and CGS (#19-20, #22) heuristics are generally used as good positive and negative features, respectively. In execution-generated testing, only a single feature 1 (# of executed instructions) is comparatively consistent positive. Meanwhile, the feature 6 and 12 are included in the top-5 negative features, where the former and the latter are about the executed instructions and the symbolic memory, respectively.

Many features for both approaches of dynamic symbolic execution simultaneously appear in both positive and negative feature tables. That is, depending on the program under test, the role of each feature changes from positive to negative (or vice versa). We mark those features that are used as both positive and negative ones in at least three benchmarks as '($\star$)'. In concolic testing, the features 10, 25, 35 and 38 appear in both Table 12a and Table 12b. In particular, the feature 10 is used as the most positive feature in `gawk` while it is the most negative one for `tree`. In execution-generated testing, the phenomenon is more prevalent; the features 10, 11, 15, 17, 20 and 22 serve as both positive and negative ones. For instance, the feature 11 (# of initially defined symbolic variables) is in top-5 positive one on `gcal` and `ptx` while it is also in top-5 negative one on `combine` and `trueprint`. This finding supports our claim that no single search heuristic can perform well for all benchmarks, and therefore it should be tuned for each target program.

### 5.5.2 Impact of Combining Static and Dynamic Features

The combined use of static and dynamic features was important. In concolic testing, we assessed the performance of our approach with different feature sets in two ways: 1) with static features only; and 2) with dynamic features only. Without dynamic features, generating good heuristic was feasible only for `grep`. Without static features, our approach succeeded in generating good heuristics for `grep` and `tree` but failed to do so for the remaining programs.

## 5.6 Limitations

One limitation of our approach is that it is difficult to clearly explain why our learned heuristics are able to achieve better results than the existing heuristics. We tried to analyze the sole strength of our heuristic through a case study on `tree-1.6.0`; our (program-specific) heuristic was able to infer the hidden relationships between branches in the program while the existing (program-independent) heuristics failed to reason them. Specifically, on `tree`, we collected and analyzed all the branches that were covered exclusively by our heuristic for every trial of the 100 trials in Figure 3 (where a single trial consists of 4,000 executions); in other words, only our heuristic can always cover these branches, but the existing heuristics always fail to cover these branches. We observed that all the branches belonged to the two functions: `getfulltree` and `unix_rlistdir`. To reach these functions in `tree`, the value of a variable named `duflag` should be true, where the value is true when the input involves the string '`--du`'. However, all the existing search heuristics do not know that the input '`--du`' makes it reach the `getfulltree` and `unix_rlistdir` functions.

Assuming that the body of '`strncmp`' is not available, consider the simplified example below:

```
1  void foo(char arr[4]){
2    for(i=0; i<4; i++){
3      switch(arr[i]){
4        case '-':
5          break;
6        case 'd':
7          break;
8        case 'u':
9          break;
10       default:
11         break;
12   }}
13   if(!strncmp('--du',arr,4))
14     unix_rlistdir(...);
15 }
```

TABLE 12: Top 10 features for concolic testing. (+): A comparatively positive feature. (-): A comparatively negative feature. (⋆): A feature used as both positive and negative features.

(a) Top 10 positive features

| Rank | Benchmarks | | | | | |
|---|---|---|---|---|---|---|
| | vim | gawk | expat | grep | sed | tree |
| 1 | # 15 | **# 10(⋆)** | # 27 | # 14 | **# 13(+)** | # 36 |
| 2 | # 18 | **# 13(+)** | **# 30(+)** | # 40 | # 2 | # 15 |
| 3 | **# 35(⋆)** | # 12 | # 23 | # 24 | # 29 | # 5 |
| 4 | # 40 | **# 38(⋆)** | **# 31(+)** | # 1 | # 3 | **# 25(⋆)** |
| 5 | **# 31(+)** | # 14 | # 4 | **# 30(+)** | # 8 | # 40 |
| 6 | # 7 | # 9 | # 9 | **# 38(⋆)** | **# 30(+)** | # 9 |
| 7 | **# 13(+)** | **# 35(⋆)** | # 8 | # 32 | **# 35(⋆)** | **# 13(+)** |
| 8 | # 3 | **# 31(+)** | # 15 | # 17 | # 6 | # 39 |
| 9 | # 12 | # 4 | **# 25(⋆)** | **# 31(+)** | # 21 | **# 30(+)** |
| 10 | **# 10(⋆)** | # 33 | # 7 | # 29 | # 16 | # 22 |

(b) Top 10 negative features

| Rank | Benchmarks | | | | | |
|---|---|---|---|---|---|---|
| | vim | gawk | expat | grep | sed | tree |
| 1 | # 17 | **# 26(-)** | # 39 | # 20 | **# 11(-)** | **# 10(⋆)** |
| 2 | **# 11(-)** | # 8 | **# 35(⋆)** | # 39 | # 32 | **# 35(⋆)** |
| 3 | # 34 | # 16 | # 33 | **# 22(-)** | # 19 | # 6 |
| 4 | # 33 | # 29 | # 37 | **# 25(⋆)** | # 40 | # 24 |
| 5 | **# 22(-)** | # 3 | **# 38(⋆)** | **# 26(-)** | **# 38(⋆)** | # 7 |
| 6 | # 21 | # 6 | # 2 | # 19 | # 18 | # 12 |
| 7 | **# 26(-)** | **# 22(-)** | # 24 | # 27 | # 5 | # 23 |
| 8 | **# 25(⋆)** | **# 11(-)** | **# 22(-)** | # 21 | # 20 | # 2 |
| 9 | # 37 | # 19 | **# 10(⋆)** | # 33 | # 34 | # 27 |
| 10 | # 20 | # 28 | # 32 | # 37 | **# 26(-)** | **# 11(-)** |

TABLE 13: Top 5 features for execution-generated testing. (+): A comparatively positive feature. (-): A comparatively negative feature. (⋆): A feature used as both positive and negative features.

(a) Top 5 positive features

| Rank | Benchmarks | | | | | |
|---|---|---|---|---|---|---|
| | gcal | combine | trueprint | du | ptx | ls |
| **1** | **# 1(+)** | **# 20(⋆)** | **# 1(+)** | # 14 | # 5 | **# 17(⋆)** |
| **2** | # 9 | # 5 | **# 10(⋆)** | **# 17(⋆)** | **# 1(+)** | # 19 |
| **3** | **# 22(⋆)** | # 21 | **# 15(⋆)** | # 16 | # 3 | **# 22(⋆)** |
| **4** | **# 11(⋆)** | # 2 | # 26 | # 7 | **# 17(⋆)** | **# 15(⋆)** |
| **5** | **# 10(⋆)** | **# 10(⋆)** | # 16 | **# 10(⋆)** | **# 11(⋆)** | # 14 |

(b) Top 5 negative features

| Rank | Benchmarks | | | | | |
|---|---|---|---|---|---|---|
| | gcal | combine | trueprint | du | ptx | ls |
| **1** | **# 15(⋆)** | **# 11(⋆)** | **# 17(⋆)** | # 19 | **# 15(⋆)** | **# 6(-)** |
| **2** | **# 20(⋆)** | **# 17(⋆)** | **# 12(-)** | **# 20(⋆)** | **# 23** | **# 12(-)** |
| **3** | **# 12(-)** | # 26 | **# 22(⋆)** | # 25 | # 2 | **# 24** |
| **4** | **# 3** | **# 22(⋆)** | **# 11(⋆)** | # 13 | **# 10(⋆)** | **# 23** |
| **5** | **# 21** | **# 12(-)** | # 25 | **# 6(-)** | **# 6(-)** | # 7 |

Function `foo` takes as input an array of characters, `arr`, where the array size is 4. The program is executed differently depending on the content of the array. In this code, the function `unix_rlistdir` (line 14) is reachable when the input, `arr`, becomes '`--du`'. As the body of the function '`strncmp`' is not available, all the existing heuristics could not infer the fact that generating the input with '`--du`' is to reach the function `unix_rlistdir`.

Our approach was able to learn the program-specific heuristic that preferentially explores the execution path that sequentially reaches the case statements in lines 4, 6, and 8. This is because our heuristic can control the priority of the execution paths in a fine-grained manner through a 40-dimensional vector of real-numbers. However, because the information is expressed at a low-level, it was difficult to interpret which feature and its corresponding weight value have allowed the execution path to be prioritized; the effect of our heuristic is not determined by a single feature and its weight value, but by the combination of 40 features and their weighted sum. It would be interesting future work to develop a more explainable learning algorithm for symbolic execution.

### 5.7 Threats to Validity

1) **Benchmarks**: For concolic testing, we collected 10 benchmarks from prior work [20], [22], [23], [41], [42], [43], [44]. For execution-generating testing, we used 6 GNU benchmark programs, including GNU Coreutils-8.32, where it is the representative benchmark in prior work [25], [30], [35], [36], [37], [38], [45]. However, our benchmarks are tailored to programs that accept strings as inputs; it is unclear that our technique works as well for programs that accept numeric inputs. That is, these 16 benchmarks may not be representative and not enough to evaluate the performance of the search heuristics in general.
2) **Testing budget**: For concolic testing, we chose 4,000 executions as the testing budget because it is the same criterion that was used for evaluating the existing heuristics (CGS, CFDS) in prior work [20], [23]. For execution-generated testing, we set 1,000 seconds to the testing budget because using the time budget is common in previous works on KLEE [22], [25], [27], [35]. However, our automatically-generated heuristic, which is specialized to maximize code coverage within a specific budget, might be less effective when given a longer testing budget; fortunately, Figure 4 shows the coverage gaps between our heuristic and the second best one are greater even when given the longer testing budget.
3) **Constraint solver**: The performance of dynamic symbolic execution may vary depending on the choice of the SMT solver. For concolic testing, we used Yices [48], the default SMT solver of CREST. For execution-generated testing, we used STP [49], the default SMT solver of KLEE.
4) **Hyper-Parameters**: We manually tuned several hyper-parameters (e.g., $K$ and $T$ in Algorithm 3) based on trial and error. More specifically, for concolic testing, we ran our tool, PARADYSE, with a few different hyper-parameter values (e.g., $K$=5, 10, 20) on three benchmarks (`vim`, `gawk`, `grep`), and chose an appropriate one (e.g., $K$=10) based on the experimental results. Then, we applied the same value for the remaining seven benchmarks. However, the values of these parameters might be ineffective for arbitrary programs.

## 6 Related work

We discuss existing works on improving the performance of dynamic symbolic execution. We classify existing techniques into the four classes: (1) improving search heuristics; (2) hybrid approaches; (3) reducing search space; (4) solving complex path conditions. Our work can also be seen as a combination of software testing and machine learning or search-based software testing.

### 6.1 Search Heuristics

As search heuristics are a critical component of dynamic symbolic execution, a lot of techniques have been proposed. However, all existing works on improving search heuristics focus on manually-designing a new strategy [8], [20], [22], [23], [24], [25], [50], [51]. In Section 2, we already discussed the CFDS [20] and CGS [23] heuristics. Another successful heuristic is generational search [21], which drives concolic testing towards the highest incremental coverage gain to maximize code coverage. For each execution path, all branches are negated and executed. Then, next generation branch is selected according to the coverage gain of each single execution. Xie et al. [50] designed a heuristic that guides the search based on the fitness values that measure the distance of branches in the execution path to the target branch. The CarFast heuristic [24] guides concolic testing based on the number of uncovered statements. The Subpath-Guided Search heuristic [25] steers symbolic execution to less explored areas of the subject program by using the length-$k$ subpath. WISE [52] aims to guide concolic testing to effectively generate the inputs with the worst-case complexity, rather than increasing code coverage; the key idea is to learn the guiding heuristic from all feasible execution paths of the program on small inputs, and then apply the heuristic for generating the worst-case inputs having large sizes. WISE can be effective in finding performance bugs, but it only works on small benchmarks (e.g., Quicksort) due to its scalability problem while our technique found performance bugs on real-world benchmarks (e.g., gawk-3.0.3). Our work is different from these works as we automate the heuristic-designing process itself.

### 6.2 Hybrid Approaches

Our approach is orthogonal to the existing techniques that combine dynamic symbolic execution with other testing techniques [9], [33], [51], [53], [54], [55]. In [33], [53], techniques such as random testing are first used and the authors of [33], [53] switch to concolic testing when the performance gains saturate. In [54], concolic testing is combined with evolutionary testing to be effective for object-oriented programs. Munch [55] is a hybrid technique to combine symbolic execution (e.g., KLEE [22]) with fuzzing (e.g., AFL [56]) to maximize the function coverage. To find bugs in real-world applications, Driller [9] is also to properly apply fuzzing [56] and concolic execution [57] when each technique has its own benefits: the high speed of fuzzing, and the ability to generate complex inputs of concolic execution.

Recently, Wang et al. [51] proposed to define and compute the strategy that optimally decides when to apply random testing or symbolic execution (i.e., strategy for hybrid testing), and which execution path to explore first during symbolic execution (i.e., search heuristic for symbolic execution). Wang et al. also presented a greedy algorithm to approximate the optimal strategy as it is challenging to realize the optimal strategy in practice. From the perspective of search heuristics, the goals of ours and Wang et al's are different: while Wang et al. aim to present a new search heuristic for symbolic execution, which uses a branch-selection criteria that prioritizes execution paths with low constraint solving costs among the paths that are difficult to reach with random testing, our focus is not on developing a particular search heuristic but on automating the very process of designing such a heuristic.

We also note that Wang et al.'s work studies more theoretical aspects of concolic testing while our work is empirical. Thus, to define the optimal strategy, Wang et al. require specific assumptions on concolic testing that are unlikely to hold in practice. In particular, Wang et al.'s work assumes that encoding and solving path-conditions are perfect; for example, they assume there is no divergence during concolic testing. However, divergence exists when performing the concolic testing tools for real-world programs [21], [58]. Wang et al.'s work also does not consider the strategy that simplifies complex symbolic constraints using concrete values. However, the simplification is a key to making concolic testing more practical, compared to classical symbolic execution. By contrast, our approach does not require such assumptions.

We did our best to experimentally compare our approach with the search heuristic proposed by Wang et al. but failed due to insufficient information. The source code of the implementation is not publicly available[10]. We tried to use the binary executable, which is available in [59], but it was not possible to run the executable without specific build configurations with which the source code was compiled. We also tried to implement the Wang et al.'s search heuristic ourselves, but this was also nontrivial due to the lack of implementation details. For example, the key idea of the Wang et al.'s search heuristic is to prioritize the path with low solving costs, where the cost is calculated as the weighted sum of all primitive operations in each constraint. However, the specific weight values are unavailable. Other information such as the bound on the number of loop iterations is also unavailable. Thus, we could only make indirect comparison; the experimental results in Wang et al.'s work show that the performance of their greedy algorithm (i.e., search heuristic) is slightly better than that of pure random testing. However, in Section 5.2.1, we showed that our approach is far superior to pure random testing, which indirectly implies that our approach can be better than the greedy algorithm in practice.

### 6.3 Reducing Search Space

Our work is also orthogonal to techniques that reduce the search space of symbolic execution [29], [36], [37], [39], [40], [41], [60], [61], [62]. The read-write set analysis [41] identifies and prunes program paths that have the same side effects. Jaffar et al. [60] introduced an interpolation method that

10. We personally asked the authors for the source code, but they could not provide the code to us due to license issues.

subsumes paths guaranteed not to hit a bug. Godefroid et al. [61], [62] proposed to use function summarizes to identify equivalence classes of function inputs. It ensures that the concrete executions in the same class have the same side effect. Abstraction-driven concolic testing [40] also reduces search space for concolic testing by using feedback from a model checker. ConTest [63] aims to reduce the input space of concolic testing by selectively maintaining symbolic variables via online learning. Chopper [39] is a novel technique for performing symbolic execution while safely ignoring functions of the subject program that users do not want to explore. Postconditioned symbolic execution [36], [37] aims to prune redundant paths of the program by using the post conditions accumulated during symbolic execution.

State-merging is a promising technique to reduce the number of states in symbolic execution [5], [38], [64], [65]. Kuznetsov et al. [38] proposed a method to balance between reducing the number of states and increasing the burden on the constraint solver by statically and dynamically estimating the importance of the states. Veritesting [5] is to interleave between dynamic symbolic execution (DSE) and static symbolic execution (SSE), where the job of SSE is to merge the execution of multiple paths that do not contain specific statements (e.g., system calls). MultiSE [65] introduced a new technique to enable symbolic execution to merge states without generating any auxiliary symbolic variables. Thereby, MultiSE is able to perform symbolic execution even when it merges values that are unsupported by constraint solver in the states. Our work is orthogonal to state-merging techniques and can be combined with them to boost symbolic execution further. Zhang et al. [64] presented a technique to automatically choose whether to merge or fork states in terms of maintaining the benefit of concrete execution during dynamic symbolic execution.

### 6.4 Solving Complex Path Conditions

Our technique can also be improved by incorporating existing techniques for solving complex path conditions. This is because for real-world programs, solving constraints that involve a variety of theories and external function calls is often a very expensive part of dynamic symbolic execution. In [66], an algorithm was introduced that can solve hard arithmetic constraints in path conditions. The idea is to generate geometric structures that help solve nonlinear constraints with existing heuristics [67]. In [68], a technique to solve string constraints was proposed based on ant colony optimization. There are attempts to solve this problem by machine learning [69]. It encodes not only the simple linear path conditions, but also complex path conditions (e.g., function calls of library methods) into the symbolic path conditions. The objective function is defined by dissatisfaction degree. By iteratively generating sample solutions and getting feedback from the objective function, it learns how to generate solution for complex path condition containing even black-box function which cannot be solved by current solver. Perry et al. [70] aim to reduce the cost of solving array constraints in symbolic execution. To do so, they present a technique to transform complex constraints into simple ones while preserving the semantics.

### 6.5 Software Testing with Machine Learning

Similar to ours, a few existing techniques use machine learning to improve software testing [26], [44], [71], [72], [73], [74], [75], [76]. In Continuous Integration, RECTECS [75] uses reinforcement learning to preferentially execute failing test-cases. Likewise, in Android GUI testing, QBE [73] employs a standard reinforcement learning algorithm (Q-learning) to increase both activity coverage and the numbed of crashes. In web application testing, to achieve high statement coverage, Sant et al. [76] automatically build statistical models from logged data via machine learning techniques. In grammar-based fuzzing, Learn&Fuzz [71] leverages recurrent neural networks to automatically learn the complex structure of PDF objects, intending to maximize code coverage. In a broad sense, our work belongs to this line of research, where we use a learning algorithm to generate search heuristics of dynamic symbolic execution automatically.

### 6.6 Search-based Software Testing

Our work can be seen as an instance of the general framework of search-based software testing/engineering [77], [78], where a testing task is formulated as an optimization problem and solved by using a meta-heuristic algorithm (e.g., genetic algorithm). In this work, we formulated the problem of generating search heuristics of dynamic symbolic execution as an optimization problem and presented an effective algorithm to solve it. To our knowledge, this is a novel application from the search-based software testing perspective.

## 7 Conclusion

The difficulty of manually crafting good search heuristics has been a major open challenge in dynamic symbolic execution. In this paper, we proposed to address this challenge by automatically learning search heuristics. Given a program under test, our technique generates a search heuristic by using a parametric search heuristic and an optimization algorithm that searches for good parameter values. For two approaches to dynamic symbolic execution, namely concolic testing and execution-generated testing, we have shown that automatically-generated search heuristics are likely to outperform existing hand-tuned heuristics, greatly improving the effectiveness of dynamic symbolic execution. We hope that our technique can supplant the laborious and less rewarding task of manually tuning search heuristics of dynamic symbolic execution.

## 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 the MSIT (Ministry of Science and ICT), Korea, under the ICT Creative Consilience program (IITP-2021-2020-0-01819)

supervised by the IITP (Institute for Information & communications Technology Planning & Evaluation). This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1C1C2006410).

## 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] 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.

[3] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proceedings of the 12th International Conference on Model Checking Software*, ser. SPIN'05, 2005, pp. 2–23.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06, 2006, pp. 322–335.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 1083–1094.

[6] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 144–155.

[7] Y. Zhang, Z. Clien, J. Wang, W. Dong, and Z. Liu, "Regular property guided dynamic symbolic execution," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 643–653.

[8] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical concolic testing techniques for COTS operating systems," in *2017 USENIX Annual Technical Conference*, ser. USENIX ATC '17, 2017, pp. 689–701.

[9] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Symposium on Network and Distributed System Security*, ser. NDSS '16, 2016, pp. 1–16.

[10] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "SymCerts: Practical symbolic execution for exposing noncompliance in x.509 certificate validation implementations," in *2017 IEEE Symposium on Security and Privacy*, ser. S&P '17, 2017, pp. 503–520.

[11] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. Butler, "FirmUSB: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017, pp. 2245–2262.

[12] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *27th USENIX Security Symposium*, ser. USENIX Security '18, 2018, pp. 309–326.

[13] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018, pp. 109–119.

[14] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "DeepConcolic: Testing and debugging deep neural networks," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '19, 2019, pp. 111–114.

[15] J. C. King, "A new approach to program testing," *SIGPLAN Not.*, pp. 228–233, 1975.

[16] ——, "Symbolic execution and program testing," *Commun. ACM*, pp. 385–394, 1976.

[17] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT–a formal system for testing and debugging programs by symbolic execution," *SIGPLAN Not.*, pp. 234–245, 1975.

[18] W. E. Howden, "Symbolic testing and the dissect symbolic evaluation system," *IEEE Transactions on Software Engineering*, pp. 266–278, 1977.

[19] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, pp. 82–90, 2013.

[20] 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.

[21] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Symposium on Network and Distributed System Security*, ser. NDSS '08, 2008, pp. 151–166.

[22] 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.

[23] 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.

[24] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "CarFast: Achieving higher statement coverage faster," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012, pp. 35:1–35:11.

[25] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications*, ser. OOPSLA '13, 2013, pp. 19–32.

[26] 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.

[27] 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.

[28] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, 2011, pp. 183–198.

[29] 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.

[30] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "DASE: Document-assisted symbolic execution for improving automated software testing," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 620–631.

[31] O. S. Dustmann, K. Wehrle, and C. Cadar, "Parti: A multi-interval theory solver for symbolic execution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018, p. 430–440.

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

[33] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for c/c++ using concolic execution," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 132–141.

[34] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E Platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, pp. 2:1–2:49, 2012.

[35] P. D. Marinescu and C. Cadar, "Make Test-zesti: A symbolic execution solution for improving regression testing," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 716–726.

[36] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Post-conditioned symbolic execution," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation*, ser. ICST '15, 2015, pp. 1–10.

[37] ——, "Eliminating path redundancy via postconditioned symbolic execution," *IEEE Trans. Softw. Eng.*, pp. 25–43, 2018.

[38] 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, pp. 193–204.

[39] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 350–360.

[40] P. Daca, A. Gupta, and T. A. Henzinger, "Abstraction-driven concolic testing," in *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, ser. VMCAI '16, 2016, pp. 328–347.

[41] 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.

[42] Y. Kim and M. Kim, "SCORE: A scalable concolic testing tool for reliable embedded software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, pp. 420–423.

[43] S. Cha, S. Lee, and H. Oh, "Template-guided concolic testing via online learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018, pp. 408–418.

[44] 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*, ser. ESEC/FSE '19, 2019, p. 235–245.

[45] S. Mechtaev, A. Griggio, A. Cimatti, and A. Roychoudhury, "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*, ser. ESEC/FSE 2018, 2018, pp. 389–399.

[46] Gnu Bug Report (gawk), http://gnu.utils.bug.narkive.com/Udtl5IZR/gawk-bug, 2005.

[47] GNU Bug Report (grep), https://sourceware.org/bugzilla/show_bug.cgi?id=24269, 2018.

[48] B. Dutertre and L. D. Moura, "The yices smt solver," Tech. Rep., 2006.

[49] 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.

[50] 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.

[51] 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.

[52] J. Burnim, S. Juvekar, and K. Sen, "Wise: Automated test generation for worst-case complexity," in *2009 IEEE 31st International Conference on Software Engineering*, ser. ICSES '09, 2009, pp. 463–473.

[53] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, 2007, pp. 416–426.

[54] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08, 2008, pp. 297–306.

[55] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, "Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18, 2018, pp. 1475–1482.

[56] American Fuzzy Lop (AFL) Fuzzer, https://github.com/mirrorer/afl, 2017.

[57] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *Proceedings of the Symposium on Network and Distributed System Security*, ser. NDSS '15, 2015.

[58] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, pp. 50:1–50:39, 2018.

[59] J. Sun, http://sav.sutd.edu.sg/research/smartconcolic, 2018.

[60] 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.

[61] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07, 2007, pp. 47–54.

[62] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10, 2010, pp. 43–56.

[63] S. Cha, S. Lee, and H. Oh, "Template-guided concolic testing via online learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018, pp. 408–418.

[64] C. Zhang, W. Yin, and Z. Lin, "Boost symbolic execution using dynamic state merging and forking," in *International Workshop on Quantitative Approaches to Software Quality*, 2018, pp. 14–21.

[65] 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.

[66] P. Dinges and G. Agha, "Solving complex path conditions through heuristic search on induced polytopes," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 425–436.

[67] P. Codognet and D. Diaz, "Yet another local search method for constraint solving," in *International Symposium on Stochastic Algorithms*, 2001, pp. 73–90.

[68] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand, "Search-driven string constraint solving for vulnerability detection," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 198–208.

[69] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li, "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*, ser. ASE '16, 2016, pp. 554–559.

[70] 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*, ser. ISSTA '17, 2017, pp. 68–78.

[71] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17, 2017, pp. 50–59.

[72] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy*, ser. S&P '17, 2017, pp. 579–594.

[73] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, ser. ICST '18, 2018, pp. 105–115.

[74] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications*, ser. OOPSLA '13, 2013, pp. 623–640.

[75] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '17, 2017, pp. 12–22.

[76] J. Sant, A. Souter, and L. Greenwald, "An exploration of statistical models for automated test case generation," in *Proceedings of the Third International Workshop on Dynamic Analysis*, ser. WODA '05, 2005, pp. 1–7.

[77] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 153–163.

[78] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, pp. 11:1–11:61, 2012.

**Sooyoung Cha** received the PhD degree from Korea University, Korea, in 2021. He is a research professor at the center for software security and assurance (CSSA) from Korea University. His research interests include software testing and static analysis, especially in data-driven symbolic execution. He has published papers on software engineering in top conferences and journals such as TSE, ICSE, FSE, and ASE.

**Seongjoon Hong** is a Ph.D. student in Department of Computer Science and Engineering at Korea University, where he received his BS degree in computer science. His research interests are in programming languages and software engineering, particularly in the application of program analysis techniques for software debugging such as program testing and program repair.

**Jiseong Bak** received the BS degree in electrical and computer engineering from Korea University, in 2021. He is currently working toward the MS degree in Department of Computer Science and Engineering, Korea University. His research interests include software testing and static analysis.

**Jingyoung Kim** is working toward the MS degree in computer science at Korea University. Her research interests include static analysis and binary analysis, with a focus on improving the performance of disassemblers. She is also interested in software testing.

**Junhee Lee** is a PhD student from Korea University. He received BS degree in mathematics from Korea University. His research focuses on automating tedious and error-prone debugging which includes finding bugs, fixing bugs, and verifying patches especially by using static analysis. He has published papers on software engineering in top conferences such as ICSE and FSE.

**Hakjoo Oh** is an Associate Professor in the Computer Science Department at Korea University. He received his Bachelors degree in Computer Science from KAIST in 2005 and his PhD from Seoul National University in 2012. He was a research associate at the center of software analysis for error-free computing in SNU before joining Korea University in 2015. His research interest includes program analysis, synthesis, and repair.