FeatMaker: Automated Feature Engineering for Search Strategy of Symbolic Execution

ANONYMOUS AUTHOR(S)

We present FeatMaker, a novel technique that automatically generates state features to enhance the search strategy of symbolic execution. Search strategies, designed to address the well-known state-explosion problem, prioritize which program states to explore. These strategies typically depend on a "state feature" that describes a specific property of program states, using this feature to score and rank them. Recently, search strategies employing multiple state features have shown superior performance over traditional strategies that use a single, generic feature. However, the process of designing these features remains largely manual. Moreover, manually crafting state features is both time-consuming and prone to yielding unsatisfactory results. The goal of this paper is to fully automate the process of generating state features for search strategies from scratch. The key idea is to leverage path-conditions, which are basic but vital information maintained by symbolic execution, as state features. A challenge arises when employing all path-conditions as state features, as it results in an excessive number of state features. To address this, we present a specialized algorithm that iteratively generates, clusters, and refines state features based on data accumulated during symbolic execution. Experimental results on 15 open-source C programs show that FeatMaker significantly outperforms existing search strategies that rely on manually-designed features, both in terms of branch coverage and bug detection. Notably, FeatMaker achieved an average of 29.4% higher branch coverage compared to state-of-the-art strategies and discovered 15 unique bugs. Of these, six were detected exclusively by FeatMaker.

ACM Reference Format:
Anonymous Author(s). 2024. FeatMaker: Automated Feature Engineering for Search Strategy of Symbolic Execution. 1, 1 (March 2024), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Symbolic execution [9, 10, 20, 38] is a powerful software testing technique that automatically generates test-cases to increase code coverage and trigger bugs. The core idea of symbolic execution is to substitute program inputs with symbolic variables and exercise the program's unique paths with these variables. More specifically, symbolic execution repeatedly chooses, executes, and updates a "state" while maintaining a set of states during a given time period. In practice, performing symbolic execution on real-world programs leads to the notorious state-explosion problem, in which the number of states to be maintained increases exponentially with the number of branches in the program. To mitigate this problem, symbolic execution adopts a search strategy [7, 8, 21, 33, 39, 43, 44] that prioritizes states deemed more effective for uncovering bugs or enhancing code coverage.

The effectiveness of search strategies heavily depends on "state features" that describe the properties of states. These features play a pivotal role in distinguish promising states from the entirety of states maintained during symbolic execution. Each search strategy incorporates its unique set of state features to identify and prioritize these promising states. For example, the Minimum-Distance-to-Uncovered (md2u) strategy [8] employs a singular state feature: the minimum distance between each state and the uncovered instructions of the program. Leveraging this feature, the strategy prioritizes states closest to uncovered regions. State-of-the-art search strategies, such as

XXXX-XXXX/2024/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ParaDySE [11] and Learch [27], utilize multiple state features to capture a diverse range of state properties. Specifically, ParaDySE [11] incorporates 26 boolean state features, while Learch [27] employs 10 state features with various types to convert each state into a 47-dimension feature vector. Notably, recent strategies harnessing multiple features [11, 27], have significantly surpassed traditional strategies that rely on a single feature [7, 8, 21, 33] in terms of both code coverage and bug-finding capability.

However, the design process for state features of search strategies remains predominantly manual. In other words, all existing search strategies have depended on either single or multiple state features that are manually designed by domain experts. Furthermore, these predefined state features are program-agnostic, meaning that they remain constant regardless of the target program to be tested. As we demonstrated in our experiments, even search strategies that utilize multiple state features, as well as those relying on a single feature, often produce inconsistent and unsatisfactory results across a variety of programs. This inconsistency arises from the manual nature of the state feature design process and their program-agnostic characteristics. For example, some state features defined in ParaDySE [11] indicate that 99.8% of states maintained during symbolic execution have exactly the same properties for the target program, which provides little help in distinguishing promising states.

In this paper, we present FeatMaker, a new technique that automates the process of designing state features tailored to each target program from scratch, thereby presenting a truly automated search strategy for symbolic execution. The core idea is to use path-conditions, a common yet invaluable type of information that symbolic execution must maintain, as state features. Specifically, we define a state feature as a boolean predicate that checks whether each state contains a specific branch condition (ϕ), given that a path-condition is a conjunction of exercised branch conditions (e.g., $\phi_1 \land \phi_2 \cdots \phi_n$). However, performing symbolic execution on real-world programs often encounters tens of thousands of symbolic branch conditions. A simplistic approach that uses all branch conditions as state features results in an overwhelming number of features, which ironically makes states indistinguishable. To address this, we present a customized algorithm that iteratively generates, clusters, and refines state features tailored to individual programs based on data gradually accumulated while interacting with a symbolic execution tool.

Experimental results demonstrate that our approach, FeatMaker, which leverages automatically-generated state features, significantly outperforms existing state-of-the-art strategies that use either single or multiple manually-designed features. We implemented FeatMaker on top of KLEE [8], a widely-used symbolic execution tool, and assessed its efficacy on 15 open-source C programs, ranging from 2K to 29K LoC, in terms of bug detection and branch coverage. In the realm of bug-finding, FeatMaker identified 15 unique bugs. Notably, six of these bugs were undetected by two cutting-edge search strategies, ParaDySE [11] and Learch [27], both of which employ multiple state features. Moreover, we confirmed that these six bugs could be reliably reproduced using the same state features that identified them. Regarding branch coverage, FeatMaker surpassed ParaDySE and Learch by covering 29.4% and 30.3% more branches, respectively. The experimental results also underscore the superiority of FeatMaker over a simplistic approach that uses all branch conditions discovered during symbolic execution as state features; FeatMaker covered an impressive 166% more unique branches than this naive approach, using only 19.7% of the state features that the naive method employs during symbolic execution.

Contributions. We summarize our contributions below:

- We introduce a pioneering approach that automatically generates state features for the search strategy of symbolic execution. FeatMaker is, to our knowledge, the first initiative

that emphasizes the need to transition from existing search strategies, which rely on manually-crafted state features, to our strategy utilizing machine-tuned features.
- We present a customized algorithm that continuously generates, clusters, and refines state features based on the data generated during symbolic execution without any prior knowledge.
- We conducted a comprehensive evaluation of the effectiveness of FEATMAKER on 15 real-world programs, comparing it with the current state-of-the-art search strategies. We plan to make our tool, FEATMAKER, and associated data, including the benchmarks, publicly accessible.[1]

2 PRELIMINARIES

2.1 Symbolic Execution

Symbolic execution [8] aims to automatically generate test-cases that explore unique program paths by introducing the concept of "states". A state is typically represented as a tuple $(instr, mem, \Phi)$. More specifically, the first element, *instr*, denotes the next instruction to be executed. The second element, *mem*, represents a symbolic memory that maps program variables to symbolic ones. The last element, Φ, denotes a path-condition, which is a conjunction of branch conditions executed by the state so far. At a high-level, symbolic execution continuously chooses, executes, and updates a state while maintaining multiple states during the testing period.

Algorithm 1 depicts a simplified symbolic execution algorithm. It accepts a program (*pgm*), a time budget (*time*), a set of state features (F), and a scoring function (SCORE) as input. For now, we will disregard the last two inputs, F and SCORE, which will be further explained in Section 2.2. At line 2, the algorithm first initializes the set S of states as a singleton set containing an initial state $(instr_0, mem_0, true)$, where $instr_0$ represents the first instruction of the program (*pgm*), and mem_0 denotes the symbolic memory at the entry point of *pgm*. At line 3, it also initializes the set TC of test-cases as an empty set.

The main process of Algorithm 1 involves an iterative selection, execution, and updating of the state while simultaneously maintaining the set S of states. Initially, the algorithm selects a state, $(instr, mem, \Phi)$, from the set S at line 5. Subsequently, the algorithm updates the selected state by executing its next instruction (*instr*). For instance, when *instr* is an if/else statement with the branch condition ϕ, the algorithm forks the selected state into two states when both true and false conditions of the statement are satisfiable. More precisely, if the conjunction of the true branch condition and the path-condition of the state, $(\Phi \wedge \phi)$, is satisfiable, the algorithm incorporates the updated state, $(instr_1, mem, \Phi \wedge \phi)$, into the set S (line 8). Similarly, if the opposite path-condition, $(\Phi \wedge \neg\phi)$, is also satisfiable, the algorithm adds another updated state into the set S (line 9). This forking process introduces the infamous state-explosion problem that the size of the set S grows exponentially with the number of conditional statements in the program (*pgm*).

The algorithm generates the test-case tc by finding a model of Φ of the selected state through the SMT solver [16, 19] when the next instruction *instr* is the halt statement. It then adds a pair of tc and Φ into the set TC of test-cases at line 11. When *instr* is the other instructions such as store and load instructions, the algorithm appropriately updates the selected state at line 13; we have omitted the details for simplicity. In this way, Algorithm 1 repeats a cycle of state selection, execution, and updating until the given time budget is exhausted, and returns a set TC of test-cases.

[1] FEATMAKER: https://github.com/anonymousfeatmaker/featmaker

Algorithm 1 Symbolic Execution

```
Input: Program ($pgm$), budget ($time$), features ($F$), scoring function (SCORE).
Output: Test cases ($TC$)
 1: procedure SYMEXECUTE($pgm, time, F$, SCORE)
 2:     $S \leftarrow \{(instr_0, mem_0, true)\}$                          ▷ A set of states
 3:     $TC \leftarrow \emptyset$                                          ▷ A set of test cases
 4:     repeat
 5:         $s \leftarrow$ SEARCHSTRATEGY($S, F$, SCORE)                   ▷ $s = (instr, mem, \Phi)$
 6:         $S \leftarrow S \setminus \{s\}$
 7:         if $instr$ = (if($\phi$) then $instr_1$ else $instr_2$) then
 8:             if ISSAT($\Phi \wedge \phi$) then $S \leftarrow S \cup \{(instr_1, mem, \Phi \wedge \phi)\}$
 9:             if ISSAT($\Phi \wedge \neg\phi$) then $S \leftarrow S \cup \{(instr_2, mem, \Phi \wedge \neg\phi)\}$
10:         else if $instr$ = halt then
11:             $TC \leftarrow TC \cup \{(tc, \Phi)\}$                      ▷ $tc$ = MODEL($\Phi$)
12:         else
13:             $S \leftarrow S \cup \{$UPDATE($s$)$\}$                   ▷ UPDATE($s$) = $(instr', mem', \Phi)$
14:     until $time$ expires (or $S = \emptyset$)
15:     return $TC$
```

2.2 Search Strategy

The SEARCHSTRATEGY function in Algorithm 1 plays a crucial role for the performance of symbolic execution. Its primary responsibility is to select a state to be explored further from the set S of states. Owing to its capacity to effectively mitigate the state-explosion problem, a variety of search strategies [8, 12, 27, 33, 39] have been developed through two implicit but common stages.

The first stage involves designing a set F of state features to represent each state in the set S into a feature vector. A state feature describes any property of a state, where its value can have an arbitrary value such as a real or boolean value. Each search strategy has its own predefined state features to distinguish promising states in terms of code coverage and bug detection from the numerous states in the set S. Formally, we can define a state feature, feat, as a function that takes a state as input and returns a real value corresponding to the feature as:

$$\text{feat} : \mathbb{S} \longrightarrow \mathbb{R}$$

where $\mathbb{S}$ represents the set of all possible states in the program. That is, using a set F of n predefined features, a search strategy converts the set S of states into the set of FV of pairs of the state and its feature vector as:

$$FV = \{(s, \langle \text{feat}_1(s), \text{feat}_2(s), \cdots, \text{feat}_n(s)\rangle) \mid s \in S\}$$

The second stage is to score each state in S based on its feature vector derived from the first stage. Each search strategy employs a unique scoring function to decide the state selection. Generally, the SCORE function, which takes a feature vector and returns a score, can be defined as:

$$\text{SCORE} : \mathbb{V} \longrightarrow \mathbb{R}$$

where $\mathbb{V}$ denotes the set of all feature vectors for all states $\mathbb{S}$.

Based on the set FV and the SCORE function obtained from the first and second stages, the search strategy (SEARCHSTRATEGY) is to prioritize the selection of the most promising state s^* as:

$$s^* = \underset{(s,\, fv) \in FV}{\text{argmax}}\ \text{SCORE}(fv)$$

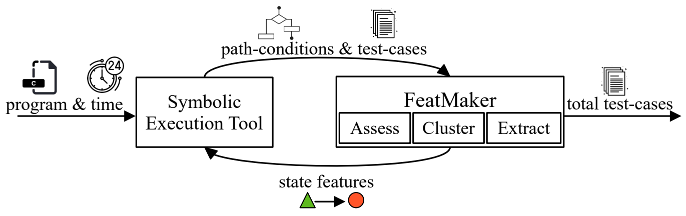

Fig. 1. Overview of FeatMaker

It is well-known that the performance of the search strategy significantly depends on the set F of state features and the state scoring function [8, 11, 33].

The existing search strategies [8, 11] can be expressed in the two stages. For instance, the Minimum-Distance-to-Uncovered (md2u) strategy [8] first designs only a state feature that measures the minimum distance between the next instruction, *instr*, of the state (*instr*, *mem*, Φ) and the instructions not yet covered. Using the single feature, it transforms each state in the set S into a 1-dimensional integer vector. Second, its scoring function is defined as:

$$\textsc{Score}(s) = \frac{1}{(\text{feat}(s))^2}$$

Intuitively, this strategy prioritizes the state closest to an instruction that has not been reached yet. For another example, the state-of-the-art parameterized search strategy [11] defines 26 boolean state features that describe the atomic properties of each state. This strategy converts each state in S into a 26-dimensional boolean vector. It also employs a scoring function that calculates a linear combination of the feature vector and a weight vector w as:

$$\textsc{Score}_w(s) = \langle \text{feat}_1(s), \text{feat}_2(s), \cdots, \text{feat}_{26}(s) \rangle \cdot w$$

where the weight vector w is a 26-dimensional real number vector. The design mechanism can be used to explain other existing search strategies for symbolic execution [7, 27, 33, 39].

2.3 Limitations

Existing search strategies have relied on a set of state features that were manually designed based on the intuition or insight of domain experts. However, these hand-crafted features often failed to effectively represent states depending on the program under test. As we demonstrated in Section 4, search strategies using a state feature failed to consistently achieve high code coverage on open-source C programs. This is because they focus on explaining a specific property of the state. State-of-the-art search strategies such as ParaDySE [11] and Learch [27] have attempted to use multiple state features to express various state properties simultaneously. Yet, these hand-crafted features also struggle to distinguish the states depending on the target program. For instance, some features (e.g., The number of initially defined symbolic variables) implemented in ParaDySE [11] calculate the same value for 99.8% of the total states, thus losing their effectiveness as features. These shortcomings have motivated us to present a new technique that automatically generates both state features and scoring functions for designing effective search strategies of symbolic execution.

3 OUR APPROACH

Our approach, FeatMaker, aims to automatically generate state features tailored to each program under test based on the data accumulated during symbolic execution. Unlike existing search strategies that rely on manually-designed state features, FeatMaker takes a fundamentally different direction, eliminating the need for human intervention in state feature design.

3.1 Overview

Figure 1 illustrates how FeatMaker automatically creates state features and scoring functions in interaction with a symbolic execution tool. Initially, due to a lack of data for automatic feature generation, FeatMaker performs symbolic execution with a simple search strategy, such as random selection, multiple times. It then collects informative data (e.g., path-conditions) generated during symbolic execution and constructs state features and scoring functions based on this data. Once the state features are generated, FeatMaker continuously runs symbolic execution on the program with our search strategy using these features and scoring policies and updates them based on the newly gathered data during testing.

FeatMaker automates the generation of state features and scoring functions through four stages: Collect, Cluster, Extract, and Learn. The Collect stage gathers data by executing the program with test-cases generated during symbolic execution. The next two stages, Cluster and Extract, aim to create promising state features from this data. In this work, we define promising features as those that effectively distinguish states that are able to increase code coverage and discover bugs. The final stage, Learn, designs suitable scoring functions for these newly created state features.

3.2 Our Search Strategy

State Feature. Our key idea is to leverage path-conditions, common yet valuable pieces of information maintained by all symbolic execution tools, as state features. More specifically, as a path-condition consists of a conjunction of exercised branch conditions (e.g., $\phi_1 \wedge \phi_2 \cdots \phi_n$), we define a state feature, feat, as a boolean predicate that checks whether each state s, $(instr, mem, \Phi)$, contains a specific branch condition (ϕ) as:

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

where it takes a state s and a symbolic branch condition ϕ as input and returns 1 when ϕ is in the path-condition Φ of s. That is, assuming that we use the set F of k state features, we represent a state into k-dimensional feature vector as:

$$\langle \text{feat}_{\phi_1}(s), \text{feat}_{\phi_2}(s), \cdots, \text{feat}_{\phi_k}(s) \rangle$$

Note that we decided to use a branch condition ϕ as a state feature because it directly affects the value of test-case, meaning that it has the potential to effectively distinguish states without any prior knowledge. For simplicity, we denote a state feature, feat_ϕ, as ϕ in the rest of the paper when there is no confusion.

The technical challenge is that performing symbolic execution on real-world programs typically generates thousands to tens of thousands of symbolic branch conditions during testing. That is, a naive approach using all branch conditions as features would result in an excessively large feature vector for each state. For instance, a representative symbolic executor, klee [8], generated about 10,000 symbolic branch conditions when testing an open-source C program, grep-3.8, for an hour; it implies a transformation of each state into a 10,000-dimensional feature vector. Hence, our approach, FeatMaker, aims to select only the most promising symbolic branch conditions as features among all candidate branch conditions generated during symbol execution.

Algorithm 2 Symbolic Execution with FeatMaker

```
Input: Program ($pgm$), time budget ($time$).
Output: Test cases ($TC$)
 1: $\langle D, NewF, PrevB \rangle \leftarrow \langle \emptyset, \emptyset, \emptyset \rangle$
 2: $\{w_i\}_{i=1}^{\eta_k} \leftarrow \langle\rangle$                          ▷ $\eta_k = 20,\ w_i = \textsc{Score}_{w_i}$
 3: $times_s \leftarrow \eta_{time}$                                   ▷ $\eta_{time} = 120$
 4: repeat
 5:     for $i = 1$ to $\eta_k$ do
 6:         $TC' \leftarrow \textsc{SymExecute}(pgm, times_s, NewF, w_i)$         ▷ $(tc, \Phi) \in TC'$
 7:         $D \leftarrow D \cup \textsc{Collect}(pgm, TC')$            ▷ $(tc, \Phi, B, NewF, w_i) \in D$
 8:     $\langle F, W \rangle \leftarrow \langle NewF, \{w_i\}_{i=1}^{\eta_k} \rangle$
 9:
10:     /* Automatic state feature generation */
11:     $\{RD_1, RD_2, \cdots, RD_n\} = \textsc{Cluster}(D)$                 ▷ $RD_i \subseteq D$
12:     $NewF \leftarrow \textsc{Extract}(RD)$                 ▷ $RD = \{RD_1, RD_2, \cdots, RD_n\}$
13:
14:     /* Automatic scoring function generation */
15:     $\{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_{|NewF|}\} \leftarrow \textsc{Learn}(F, W, PrevB, NewF, D)$
16:     $\{w_i\}_{i=1}^{\eta_k} \leftarrow$ sample from $\mathcal{P}_1 \times \mathcal{P}_2 \times \cdots \times \mathcal{P}_{|NewF|}$
17:     $PrevB \leftarrow \bigcup\{B \mid (F = F') \land (\_, \_, B, F', \_) \in D\}$
18: until $time$ expires
19: return $TC$                           ▷ $TC = \{(tc, \Phi) \mid (tc, \Phi, \_, \_, \_) \in D\}$
```

Scoring Function. We define our scoring function, similar to the one used in the parameterized search strategy [11], as follows:

$$\textsc{Score}_w(s) = \langle \phi_1(s), \phi_2(s), \cdots, \phi_k(s) \rangle \cdot w$$

The scoring function takes a state s and a k-dimensional vector w of real numbers, where k equals the number of state features. It scores the state by calculating a linear combination of its feature vector and the weight vector w. In this work, we not only generate state features automatically but also aim to iteratively update a set of suitable weight vectors based on data accumulated from previous runs of symbolic execution. This is crucial as the value of each weight vector w also greatly influences the prioritization of states. For simplicity, we will refer to $\textsc{Score}_w$, as a weight vector w when there is no risk of confusion.

3.3 FeatMaker

Algorithm 2 describes how FeatMaker automatically generates and updates state features and scoring functions during testing. It only requires a target program (pgm) and a total testing budget ($budget$) as input and returns a set TC of test-cases generated within the total budget. Initially, it sets up three empty sets: a set D of data, a set $NewF$ of state features, and the set $PrevB$ of covered branches (line 1). It also initializes η_k scoring functions, more precisely η_k weight vectors (line 2). The rationale for employing multiple scoring functions is based on our observation that each function tends to cover different code areas (e.g., branches). That is, utilizing a variety of scoring functions allows FeatMaker to effectively prioritize different states, enabling it to cover various branches. The total budget ($budget$) is then divided into smaller budgets, each defined by η_{time}, to perform symbolic execution (Algorithm 1) multiple times on the program (line 3). This iterative approach allows FeatMaker to interact with a symbolic execution tool multiple times, thereby finding more effective state features and weight vectors based on accumulated data. For our experiments, we determined the hyper-parameters η_k and η_{time} to be 20 and 120 seconds, respectively, through trial and error. Intuitively, by setting these values, we aimed to give FeatMaker ample opportunities to update state features and scoring functions, recognizing that developing effective ones from scratch is inherently challenging.

After initialization, Algorithm 2 performs symbolic execution repeatedly, using the state features ($NewF$) and scoring policies ($\{w_1, \cdots, w_{\eta_k}\}$). These are updated in each iteration of the outer loop at lines 4–18. Initially, with no state features (i.e., $NewF = \emptyset$), symbolic execution is run with a naive search strategy, such as a random strategy. This produces the set TC', where each element in TC' is a pair of a test-case and its path-condition. Based on TC', Algorithm 2 proceeds through the Collect, Cluster, Extract, and Learn stages to generate $NewF$ and W for the next iteration of symbolic execution runs.

Collect. In the Collect stage, the goal is to collect the necessary elements for automatic feature generation. To achieve this, we perform symbolic execution using state features ($NewF$) and i-th scoring function (w_i), producing the set TC' of test-cases (line 6). The Collect function then generates the data D' from TC' by executing the program with each test-case tc in TC' and calculating the set B of branches covered by tc. The data D' is represented as:

$$D' = \{(tc_1, \Phi_1, B_1, NewF, w_i), \cdots, (tc_p, \Phi_p, B_p, NewF, w_i)\}$$

Each element in D' is a 5-tuple: a test-case tc, its path-condition Φ, a set B of branches covered by tc, the set $NewF$ of state features, and the i-th weight vector w_i. After D' corresponding to TC' is collected, it is added to the set D (line 7). This process is repeated for every scoring function. At line 8, we store the currently used state features and weight vectors in F and W, respectively. These will act as valuable data for refining the weight vectors.

Cluster. During the Cluster stage at line 11, our objective is to first group the set D into subsets, $\{SD_1, SD_2, \cdots, SD_m\}$, where each subset has the same potential, specifically the same set of covered branches. We then identify the most representative subsets, $\{RD_1, RD_2, \cdots, RD_n\}$, from these grouped subsets; these representative subsets will be used for generating new state features. To achieve this, we first define i-th subset SD_i, which has the same branch coverage from the set D as follows:

$$SD_i = \{(tc, \Phi, B, F, w) \mid (B_i = B) \land (tc, \Phi, B, F, w) \in D\}$$

Here, B_i represents the set of covered branches of i-th element in D. That means that all elements of the subset SD_i cover the same set of branches as B_i of the i-th element $(tc_i, \Phi_i, B_i, F, w)$ in D.

After generating the set SD of subsets from SD_1 to SD_m, we choose a set RD of representative subsets from SD. In this work, RD is defined as the smallest subset of SD that can cover the same branches as those collectively covered in all subsets from SD_1 to SD_m. To construct RD, we define the Cov function that takes i-th subset SD_i and returns the set of all branches covered in the subset as:

$$\mathrm{Cov}(SD_i) = \bigcup_{(\_,\_,B,\_,\_) \in SD_i} B$$

Using the Cov function, we can define the set SD^* of candidate representative subsets as:

$$SD^* = \operatorname*{argmax}_{SD' \subseteq SD} \Big| \bigcup_{1 \le i \le |SD'|} \mathrm{Cov}(SD'_i) \Big|$$

In this work, we assume that the 'argmax' notation returns the set of all candidate arguments that maximize the objective, which is the total number of exercised branches. From the set SD^*, we finally define RD as the set of representative subsets as:

$$RD = \underset{RD^* \in SD^*}{\operatorname{argmin}} |RD^*|$$

Unlike the 'argmax' notation, we assume that the 'argmin' notation returns an arbitrary argument that minimizes the objective. Therefore, RD is the smallest subset of SD that can equally reach all branches covered by all subsets in SD. This problem of calculating the set RD corresponds to the set-cover problem [31], which is NP-complete. In our experiments, we solve the problem by using a simple greedy approximate algorithm [30]. We are now ready to extract state features in the next stage.

Example 3.1. Suppose that we obtain the set D from the previous COLLECT stage as:

$$\begin{aligned} D = \{&(tc_1, \Phi_1, \{b_1, b_2, b_3\}, F, w_1), (tc_2, \Phi_2, \{b_1, b_2, b_3\}, F, w_2), (tc_3, \Phi_3, \{b_2, b_4\}, F, w_3), \\ &(tc_4, \Phi_4, \{b_2, b_4\}, F, w_4), (tc_5, \Phi_5, \{b_4, b_5\}, F, w_5), (tc_6, \Phi_6, \{b_4, b_5\}, F, w_6)\} \end{aligned}$$

Recall that each element in D consists of a 5-tuple: a test-case, its path-condition, the set of branches covered by the test-case, the set of state features, and the weight vector. Then, in the CLUSTER stage, we first divide the set D into three subsets, SD_1, SD_2, and SD_3, as:

$$\begin{aligned} SD_1 &= \{(tc_1, \Phi_1, \{b_1, b_2, b_3\}, F, w_1), (tc_2, \Phi_2, \{b_1, b_2, b_3\}, F, w_2)\} \\ SD_2 &= \{(tc_3, \Phi_3, \{b_2, b_4\}, F, w_3), (tc_4, \Phi_4, \{b_2, b_4\}, F, w_4)\}, \\ SD_3 &= \{(tc_5, \Phi_5, \{b_4, b_5\}, F, w_5), (tc_6, \Phi_6, \{b_4, b_5\}, F, w_6)\} \end{aligned}$$

where all elements of each subset SD_i cover the same set of covered branches. Using the set SD that consists of these three subsets (e.g., $SD = \{SD_1, SD_2, SD_3\}$), we then construct a set RD of representative subsets as follows:

$$RD = \{SD_1, SD_3\}$$

where RD denotes the minimum subset of SD that can exercise all branches, i.e., $\{b_1, b_2, b_3, b_4, b_5\}$, covered by three subsets in SD.

Extract. In the EXTRACT stage, we derive a set $NewF$ of state features from the set RD of representative subsets, $\{RD_1, RD_2, \cdots, RD_n\}$:

$$NewF = \{\phi \mid (\phi \in \Phi) \wedge ((\_, \Phi, \_, \_, \_) \in RD') \wedge (RD' \in RD)\}$$

where an element RD' in RD consists of a set of 5-tuples (tc, Φ, B, F, w). Intuitively, we simply use all the symbolic branch conditions in the representative data RD as state features (e.g., $\text{feat}_\phi = \phi$) in Section 3.2.

Example 3.2. Let us assume that the set RD from Example 3.1 is defined as:

$$\begin{aligned} RD = \{&\{(tc_1, \phi_1 \wedge \phi_2, \{b_1, b_2, b_3\}, F, w_1), (tc_2, \phi_1 \wedge \phi_3, \{b_1, b_2, b_3\}, F, w_2)\}, \\ &\{(tc_5, \phi_2 \wedge \phi_5, \{b_4, b_5\}, F, w_5), (tc_6, \phi_2 \wedge \phi_6, \{b_4, b_5\}, F, w_6)\}\} \end{aligned}$$

where the first element in RD corresponds to SD_1 and the second represents SD_3. In the EXTRACT stage, from RD, we derive the set $NewF$ of new state features as:

$$NewF = \{\phi_1, \phi_2, \phi_3, \phi_5, \phi_6\}$$

Intuitively, we utilize all the symbolic branch conditions used in RD as the new state features $NewF$.

Algorithm 3 Our Learning Algorithm

```
Input: State features ($F$), weight vectors ($W$), previously covered branches ($PrevB$), new state features ($NewF$),
    data ($D$)
Output: Probabilistic distributions ($\mathcal{P}$)
 1: procedure LEARN($F, W, PrevB, NewF, D$)
 2:     $\langle \mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_{|NewF|} \rangle \leftarrow \langle \mathcal{U}[-1,1], \mathcal{U}[-1,1], \cdots, \mathcal{U}[-1,1] \rangle$
 3:     $WD \leftarrow \emptyset$                                              ▷ $WD$ is learning data
 4:
 5:     /* Generate the learning data. */
 6:     for each $w \in W$ do
 7:         $B_w \leftarrow \bigcup \{B \mid (w = w') \wedge (\_, \_, B, \_, w') \in D\}$
 8:         $WD \leftarrow WD \cup \{(w, grade)\}$                          ▷ $grade = |B_w| + |(B_w \setminus PrevB)|$
 9:     $TopW, MidW, BotW \leftarrow$ CLASSIFY($WD$)
10:
11:     /* Update the probability distribution for $i$-th feature's weight. */
12:     for each $\phi_i \in (F \wedge NewF)$ do                            ▷ $\phi_i$ is i-th feature in $F$
13:         $TopW_i \leftarrow \{w^i \mid (\langle w^1, w^2, \cdots, w^{|F|} \rangle \in TopW)\}$
14:         $BotW_i \leftarrow \{w^i \mid (\langle w^1, w^2, \cdots, w^{|F|} \rangle \in BotW)\}$
15:         if NOTSIMILAR($TopW_i, BotW_i$) then $\mathcal{P}_i \leftarrow \mathcal{N}(\mu(TopW_i), \sigma(TopW_i), -1, 1)$
16:     return $\mathcal{P}$                                                ▷ $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_{|NewF|}\}$
```

Learn. In the LEARN stage, the goal of Algorithm 2 is to generate a set of scoring functions, $\{\text{SCORE}_{w_1}, \cdots, \text{SCORE}_{w_{\eta_k}}\}$, that will be used in conjunction with the newly generated state features in the EXTRACT stage. Recall that our scoring function, SCORE_w, scores each state based on the weight vector w of real numbers. Therefore, Algorithm 2 first learns the probabilistic distributions for sampling a suitable weight vector w and then generates a set of weight vectors from the learned distributions (lines 15–16). To accomplish this, the LEARN function evaluates a set of weight vectors used in previous symbolic execution runs, categorizes these vectors into good and bad weight vectors, and updates probability distributions based on both types of weight vectors.

The LEARN procedure (Algorithm 3) outlines the process of learning a set of probability distributions, $\mathcal{P}$, whose size is equal to the number of new state features, $NewF$. The algorithm takes as input 5 arguments: current state features (F), current weight vectors (W), previously covered branches ($PrevB$), newly generated state features ($NewF$), and data (D). It first initializes the sampling probability, $\mathcal{P}_i$, of i-th feature weight in $NewF$, to a uniform distribution between -1 and 1 (line 2). It sets the learning data, WD, to an empty set.

In the loop at lines 6–8, we first calculate B_w, which is the set of all branches covered while performing symbolic execution with the scoring function SCORE_w. We can define the set B_w as:

$$B_w = \bigcup \{B \mid (w = w') \wedge (\_, \_, B, \_, w') \in D\}$$

That is, B_w is the branch coverage achieved by the weight vector w. At line 8, we calculate the *grade* value corresponding to the performance of w, and accumulate the pair $(w, grade)$ into the set WD. The *grade* value of w is calculated by the sum of the following two values:

$$grade = |B_w| + |(B_w \setminus PrevB)|$$

where the former, $|B_w|$, denotes the total number of branches covered by w, and the latter represents the number of branches exclusively covered by w compared to $PrevB$, which is the set of branches collectively covered during the previous iteration of the loop at line 5–7 in Algorithm 2. That is, this grading policy awards high scores to weight vectors that not only cover the largest number of branches but also contribute to uncovering previously unseen branches. For simplicity, we have omitted the process that assigns equal weight to both factors, $|B_w|$ and $|(B_w \setminus PrevB)|$, through the normalization. After constructing the set *WD*, the Learn procedure is to divide the weight vectors in *WD* into three groups, *TopW*, *MidW*, and *BotW*, based on their *grade* values. To do so, we simply use the k-means clustering algorithm [32], where we set k to 3 (line 9). In this work, we use *TopW* and *BotW*, two sets of the most effective and ineffective weight vectors to learn the probability distributions.

Example 3.3. Assume that the set *WD* is defined as:

$$WD = \{(w_1, 2.0), (w_2, 1.8), (w_3, 0.2), (w_4, 0.3), (w_5, 0.8), (w_6, 0.9)\}$$

where each element in *WD* is a pair of a weight vector and its grade value. Our clustering algorithm categorizes the six weight vectors in *WD* into three groups based on their *grade* values as:

$$TopW = \{w_1, w_2\},\ MidW = \{w_5, w_6\},\ BotW = \{w_3, w_4\}$$

At line 12–16, the Learn procedure (Algorithm 3) is to update the sampling distribution $\mathcal{P}_i$ for the weight value of each i-th feature (ϕ_i) in F based on *TopW* and *BotW*. Basically, we update the distribution $\mathcal{P}_i$ only when the currently used i-th feature, ϕ_i, is in the set *NewF* of newly generated state features (line 12). Since our approach (FeatMaker) generates new state features every iteration of the outer loop at line 4–18 in Algorithm 2, FeatMaker is not able to learn the sampling probability for that newly appeared feature. In lines 13–14, we collect $TopW_i$, the set of effective weight values corresponding to the i-th feature, and $BotW_i$, the set of ineffective weight values. Then, the NotSimilar function checks whether the two sets, $TopW_i$ and $BotW_i$, have similar distributions, and returns true if they are not similar as:

$$\textsc{NotSimilar}(TopW_i, BotW_i) \iff |\mu(TopW_i) - \mu(BotW_i)| + |\sigma(TopW_i) - \sigma(BotW_i)| > 1$$

where $\mu(TopW_i)$ and $\sigma(TopW_i)$ are the median and standard deviation values of the set $TopW_i$, respectively. At lines 15–16, if the distributions of $TopW_i$ and $BotW_i$ are not similar, we update the probability distribution $\mathcal{P}_i$ to the truncated normal distribution with the median value $\mu(TopW_i)$, standard deviation $\sigma(TopW_i)$, and the interval [-1, 1].

The primary intuition behind these distribution updates is to sample new weight vectors that are statistically similar to the most promising weight vectors with the best performance (e.g., branch coverage) among all evaluated vectors so far. However, if the distributions of the top-performing vectors (*TopW*) and the least effective vectors (*BotW*) are too similar, it is challenging to ensure that vectors similar to the best ones are indeed superior. In such cases, we adopt a conservative approach by sampling new random weight vectors based on uniform distributions. This approach ensures that we only update the sampling distributions of scoring functions when there is clear statistical significance.

Example 3.4. Suppose that *TopW* and *BotW* in Example 3.3 are defined as:

$$TopW = \{\langle 0.9, -0.2, 0.7\rangle, \langle 0.6, -0.7, -0.3\rangle\}, BotW = \{\langle -0.8, 0.4, -0.1\rangle, \langle -0.2, 0.8, 0.5\rangle\}$$

Initially, we extract $TopW_1$ and $BotW_1$ from *TopW* and *BotW*, representing the set of effective and ineffective weight vectors for the first feature, respectively:

$$TopW_1 = \{0.9, 0.6\}, BotW_1 = \{-0.8, -0.2\}$$

Using the NotSimilar function, we evaluate the distribution similarity between $TopW_1$ and $BotW_1$. If the sum of the differences in standard deviation and median difference between $TopW_1$ and $BotW_1$ exceeds 1, indicating statistical dissimilarity, we proceed to update the probability distribution

Table 1. 15 benchmark programs

Programs	LOC	# of Branches	Programs	LOC	# of Branches	Programs	LOC	# of Branches
sqlite-3.33.0	42K	29,337	diff (diffutils-3.7)	10K	7,688	expr (coureutils-8.32)	6K	4,608
gawk-5.1.0	25K	18,299	du (coureutils-8.32)	8K	6,707	csplit (coureutils-8.32)	6K	4,560
gcal-4.1	23K	15,799	make-4.3	8K	6,604	ls (coureutils-8.32)	5K	3,824
find (findutils-4.7.0)	13K	10,071	patch-2.7.6	8K	6,227	trueprint-5.4	4K	2,518
grep-3.4	11K	8,566	ptx (coureutils-8.32)	6K	5318	combine-0.4.0	4K	2,359

$\mathcal{P}_1$ to a truncated normal distribution with a median value of 0.75, a standard deviation of 0.15, and the interval [-1, 1]. This process is repeated for $TopW_2$ and $BotW_2$ through $TopW_3$ and $BotW_3$, updating the probability distributions from $\mathcal{P}_1$ to $\mathcal{P}_3$.

After the Learn procedure successfully updates a set $\mathcal{P}$ of probability distributions, FeatMaker (Algorithm 2) now samples a set of k weight vectors, $\{w_i\}_{i=1}^{\eta_k}$, to be used in the next iteration of the inner loop at line 5–7 based on $\mathcal{P}$ (line 16). In this way, FeatMaker aims to enhance the performance of symbol execution by continuously updating state features and scoring functions based on the accumulated data during testing.

4 EXPERIMENTS

In this section, we experimentally evaluate FeatMaker to answer the following research questions:

1. **Code coverage:** How effectively can FeatMaker increase code coverage in comparison to existing state-of-the-art search strategies that utilize manually-tuned state features?
2. **Bug-finding:** How effectively can FeatMaker find bugs? Can it identify unique bugs that all other existing strategies fail to find? Are these unique bugs reproducible?
3. **Comparison with naive approach:** How well does FeatMaker perform compared to a naive approach that employs all path-conditions as state features? How effectively does FeatMaker reduce the increase in state features compared to the naive approach?
4. **Impact of Hyper-parameters:** How do different hyper-parameter values influence the performance of FeatMaker?

We implemented FeatMaker on top of KLEE [8][2], a widely-adopted symbolic execution tool for testing C programs. All experiments were done on a Linux machine equipped with 52 CPUs and 256GB RAM, powered by two Intel Xeon Gold 6230R processors.

4.1 Experimental Settings

Baselines. We evaluated our approach, FeatMaker, against four existing search strategies: ParaDySE [11], Learch [27], SGS (Subpath-Guided Search) [33] and Depth [8]. Notably, ParaDySE and Learch, are arguably the state-of-the-art baselines that utilize multiple predefined state features to represent states. Specifically, ParaDySE transforms each state to a 26-dimension feature vector, while Learch converts it into a 47-dimension feature vector. For ParaDySE, which requires additional time to generate a learned search strategy for each benchmark program, we allocated its default budget [11, 40] of 72,000 seconds and 20 CPU cores. For Learch, which consists of four learned strategies, we have evaluated it by running each learned strategy for a quarter of the total time budget (24h) and then combined all the test-cases generated by each learned strategy.

The third baseline, SGS, uses a single state feature: the number of times that a k-length subpath of each state has been covered. The original authors of SGS employed four distinct k-subpath-guided strategies, setting k values to 1, 2, 4, and 8. We assessed SGS by running each of these strategies

[2]We used KLEE-2.1: https://github.com/klee/klee/releases/tag/v2.1.

with the four different k values and gathered the test-cases they generated. Lastly, we chose Depth, one of the search strategies implemented in KLEE [8], as a baseline because it was commonly used for evaluations in the first three baselines. Fortunately, all four search strategies were publicly available, enabling us to use their implementations without any additional effort.

Benchmarks. For our evaluation, we used 15 open-source C programs listed in Table 1. In constructing our benchmark suite, our objective was to ensure that our evaluation results were not biased by a specific benchmark suite used in prior work. To this end, we first gathered all benchmark programs previously used for evaluations in our four baselines [8, 11, 27, 33]. From this collection, we excluded smaller benchmark programs (e.g. `tty`, `true` in `coreutils-8.32` and `cjson-1.7.14`) that had already achieved high coverage with existing search strategies. Specifically, we included larger programs that had a minimum of 2,000 branches. Furthermore, we excluded programs (e.g., `readelf-2.36`) that failed to execute their core functionality, leading to very low branch coverage. For instance, for `readelf-2.36`, all four baselines achieved a branch coverage of less than approximately 1.2% over a 24-hour testing period, which indicates its unsuitability as a benchmark for evaluation. Note that, for our benchmark suite in Table 1, one of the four baselines achieved an average branch coverage of around 25% within a 24-hour period.

Other Settings. In each experiment, we allocated a 24-hour testing period for each program. We conducted each experiment 5 times and reported the average results. To better compare FeatMaker with existing search strategies, we allocated 24 hours, which exceeds the testing budgets used in evaluations of existing search strategies, such as 1,000 seconds in ParaDySE [12] and 8 hours in Learch [27]. This means our experiments were conducted more extensively than those of existing search strategies. For example, generating the results shown in Figure 2 required 9,000 hours when using a single core (e.g., 24 hours * 5 trials * 15 benchmarks * 5 search strategies). Second, we basically used KLEE's default parameter settings [8, 36] without any modifications. For certain benchmarks, such as `sqlite` and `make`, we used the hand-tuned parameter settings provided in Learch [27] to preserve its original performance. Third, as outlined in Algorithm 2, our approach, FeatMaker, incorporates two hyper-parameters: η_k and η_{time}. For our experiments, we set η_k to 20 and η_{time} to 120 seconds. Lastly, during the initial iteration of the loop at lines 5–7 in Algorithm 2, FeatMaker used KLEE's default search strategy due to the lack of data for automatic feature generation.

4.2 Code Coverage

In terms of branch coverage, FeatMaker surpasses all four baselines, including the state-of-the-art strategies, ParaDySE and Learch. Overall, FeatMaker covered 35.2% and 36.1% more branches than ParaDySE and Learch that were often the second best, respectively, across all benchmarks. Figure 2 illustrates the average branch coverage attained by FeatMaker and the four baselines across the 15 benchmark programs. These results were obtained using gcov [41], a popular tool for code coverage analysis used in prior work on search strategy [8, 11, 27, 33].

Figure 2 shows that FeatMaker consistently achieved the highest average branch coverage across all benchmarks. Particularly on larger benchmarks like `gawk`, `grep`, and `diff`, FeatMaker significantly surpassed the second best strategies. For instance, FeatMaker covered an average of 4,263 branches for `gawk` while the second best strategy, Learch, covered only 3,063 branches. Similarly, FeatMaker covered 3,545 and 2,007 branches on `grep` and `diff`, respectively, outperforming ParaDySE, the second-best strategy, by 1,101 and 648 branches. Notably, FeatMaker continues to break records, even where ParaDySE and Learch have already achieved high coverage. For example, on `diff`, FeatMaker covers 47.7% more branches than ParaDySE, which had already covered 45% more branches on average than the third best strategy, Learch. The success

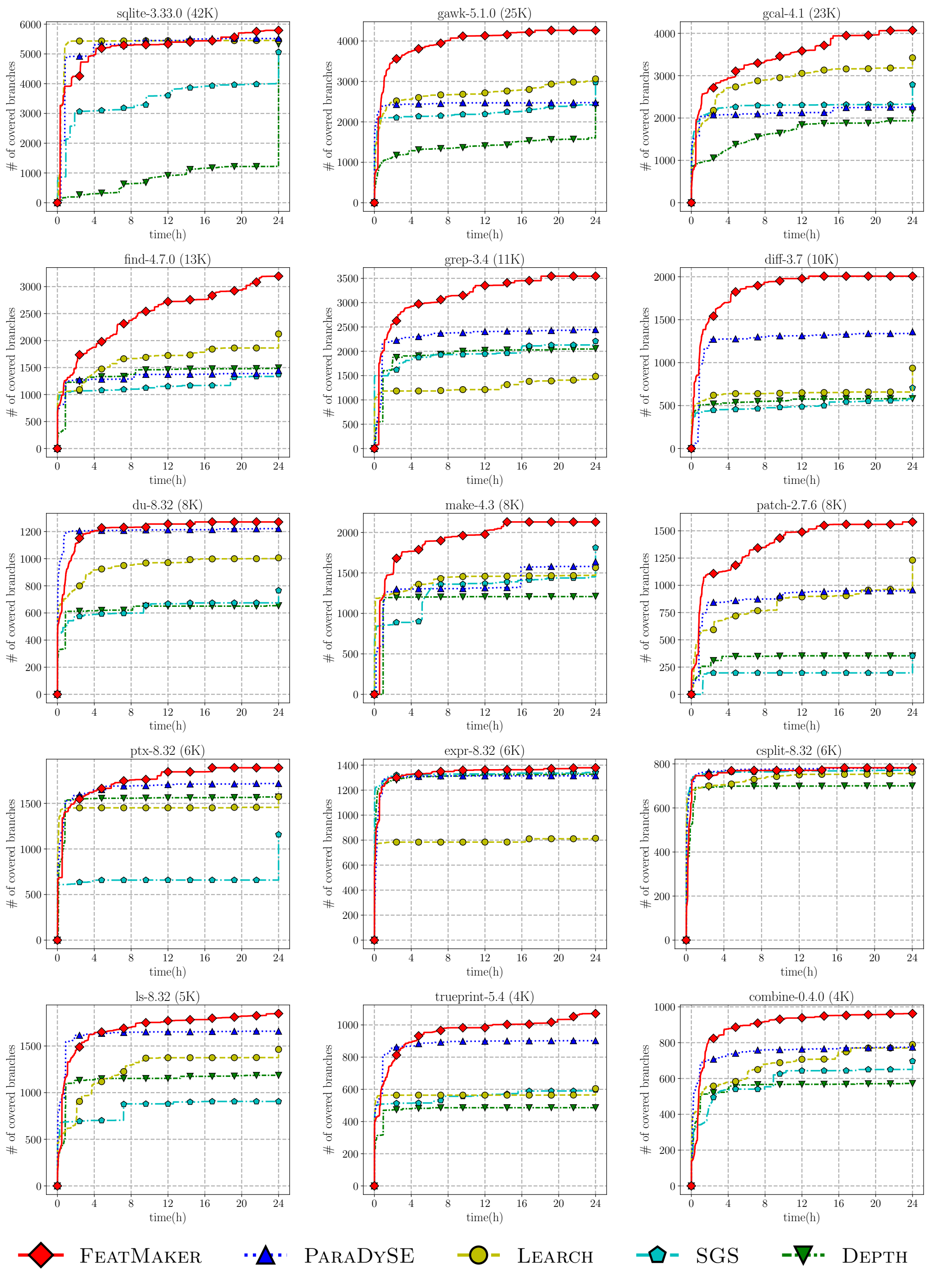

Fig. 2. The average number of branches covered by FeatMaker and the four baselines on 15 benchmarks

of FeatMaker lies in its use of both program-adaptive state features and scoring functions, which are continuously refined based on data accumulated during the testing of each target program. This approach contrasts with existing search strategies that utilize a fixed set of state features and scoring functions, regardless of the target program.

Excluding our strategy, the two baseline strategies, ParaDySE and Learch, which use multiple predefined state features, generally outperform the other two baselines that employ a single state feature across 15 benchmark programs. More specifically, ParaDySE and Learch ranked second in eight and five programs, respectively. However, these state-of-the-art strategies occasionally demonstrated inconsistent performance in terms of branch coverage. For instance, in the `grep` and `expr` programs, Learch unexpectedly recorded the lowest branch coverage among the four baseline strategies. Likewise, for the `gcal` and `find` programs, ParaDySE ranked third out of the four baselines in performance. Notably, for the `make` and `expr` programs, the SGS strategy, which uses a single feature, covered more branches than both ParaDySE and Learch. This highlights the potential limitations of manually-crafted multiple state features.

The standard deviations for the average branch coverage achieved by each search strategy across all 15 benchmarks were as follows: FeatMaker (75), ParaDySE (63), Learch (62), SGS (50), and Depth (26). Given the marked increase in branch coverage by FeatMaker compared to the four baseline search strategies, these deviation differences might not seem particularly significant. The coverage results in Figure 2 were also statistically significant according to a Mann-Whitney U test [34] for the majority of our benchmark suite. Specifically, the p-values of FeatMaker were below 0.05 when compared to ParaDySE and Learch for 12 of the benchmark programs. Exceptions included a few programs, such as `du` and `csplit`, where the difference in the number of covered branches between FeatMaker and these two baselines was small.

Case Study. To demonstrate how FeatMaker achieves higher branch coverage compared to two state-of-the-art strategies, ParaDySE and Learch, we conducted a case study on the `find` program. We analyzed branches that were exclusively covered by FeatMaker, which were unreachable by ParaDySE and Learch. Our finding revealed that FeatMaker uniquely reached several functions within the program, thereby covering an additional 555 branches on average. This success is attributed to FeatMaker's ability to navigate through various case statements in specific switch-case statements within the 'peek_token' function. The function includes 33 case statements across two switch-case structures, executing different functions (e.g., duplicate_node, calc_eclosure_ite) based on the token type (e.g., OP_ALT) in each case as follows:

```
static  int peek_token(token *token,  string  *input){
  unsigned char  c  = input [0];
  token->opr.c  = c;
  switch  (c){
    case  '\n':
      token->type = OP_ALT;
    /*  18  more case  statements  */
   }
  unsigned char  c2  = input [1];
  token->opr.c  = c2;
  switch  (c2)   {
     case  '*':
       token->type = OP_DUP_ASTERISK;
     /*  13  more case  statements  */
  }
  return  1;
}
```

On average, FeatMaker succeeded in exploring about 27 out of 33 case statements, whereas ParaDySE and Learch managed to reach only 2 and 4 case statements, respectively. The effectiveness of FeatMaker stems from its strategic employment of branch conditions within specific switch-case statements as state features, along with scoring functions that enhance the weights corresponding to these features.

Table 2. Comparison of bug-finding ability of FeatMaker and the four baselines.

Benchmarks	Error-Types	Error Locations	FeatMaker	ParaDySE	Learch	SGS	Depth
gawk-5.1.0	Abnormal termination	Line: 1337 in main.c	✔	✕	✕	✕	✕
gcal-4.1	Segmentation fault	Line: 740 in /src/file-io.c	✔	✔	✔	✔	✕
	Segmentation fault	Line: 3956 in /src/gcal.c	✔	✔	✕	✕	✕
	Segmentation fault	Line: 4934 in /src/gcal.c	✔	✕	✕	✕	✕
	Segmentation fault	Line: 5001 in /src/gcal.c	✔	✕	✕	✕	✕
	Segmentation fault	Line: 72 in /libc/string/strncasecmp.c	✔	✔	✔	✕	✕
	Abnormal termination	Line: 27 in /libc/string/strcpy.c	✔	✕	✔	✕	✕
	Abnormal termination	Line: 29 in /libc/string/memcpy.c	✔	✕	✕	✕	✕
find-4.7.0	Segmentation fault	Line: 131 in /find/tree.c	✔	✕	✔	✕	✕
	Abnormal termination	Line: 3143 in /find/parser.c	✔	✕	✕	✕	✕
patch-2.7.6	Segmentation fault	Line: 2396 in /src/pch.c	✔	✔	✔	✕	✕
combine-0.4.0	Segmentation fault	Line: 385 in /src/field.c	✔	✔	✔	✔	✔
	Segmentation fault	Line: 458 in /src/field.c	✔	✔	✕	✕	✕
	Segmentation fault	Line: 633 in /src/df_options.c	✔	✔	✕	✕	✕
	Segmentation fault	Line: 646 in /src/df_options.c	✔	✕	✕	✕	✕

Table 3. Reproducibility of bugs discovered by FeatMaker using state features and weight vectors.

Benchmarks	Error Locations	# of (F,w)	Reproducibility(%)	Benchmarks	Error Locations	# of (F,w)	Reproducibility(%)
gcal-0.4.0	4934 in /src/gcal.c	2	100% (200/200)	gawk-5.4.0	1337 in main.c	8	100% (800/800)
	5001 in /src/gcal.c	2	100% (200/200)	find-4.7.0	131 in /find/tree.c	8	86% (690/800)
	29 in /libc/string/memcpy.c	9	87% (780/900)	combine-0.4.0	646 in /src/df_options.c	4	52% (208/400)

4.3 Bug-Finding

In terms of bug-finding capabilities, FeatMaker significantly surpasses all four baselines reliant on manually-tuned state features. It succeeded in finding 15 unique bugs across five benchmarks: `gawk`, `gcal`, `find`, `patch`, and `combine`. In contrast, the second best strategy, ParaDySE, identified only seven of them. Notably, our approach (FeatMaker) discovered six distinct bugs missed by all four baselines.

Table 2 highlights the superiority of FeatMaker in bug detection as it found a superset of the total bugs discovered by all four baselines. This table provides four key details: the version of each program, the error-type, the error location, and the bug-finding result of each search strategy. A "✔" signifies that the strategy successfully generated at least one bug-triggering test case during five iterations of 24 hours. Conversely, a "✕" indicates the failure to generate any such test case over the same 120-hour testing period (24 hours × 5 times). FeatMaker found 15 unique inputs that triggered 12 segmentation faults (SIGSEGV), and three abnormal terminations (SIGABRT) across five benchmarks. For example, on `gcal-4.1`, the command `./gcal "--pEriO=-$"`—exclusively discovered by FeatMaker—causes a segmentation fault, terminating the program execution abnormally. Similarly, the command `./find "!" "-amin" "NAN"`—generated by FeatMaker— immediately aborts the `find` program. Interestingly, this bug survived from `find-4.7.0` (2019) to `find-4.9.0` (2022), which is the latest version of the program.

Table 4. The ratio of killed mutants to total mutants by FeatMaker and the four baselines.

Benchmarks	# of total mutants	FeatMaker	ParaDySE	Learch	SGS	Depth
find-4.7.0	38,852	**89.64% (34,828)**	44.74% (17,382)	74.52% (28,954)	44.82% (17,415)	43.35% (16,843)
combine-0.4.0	10,906	**99.95% (10,901)**	96.85% (10,562)	96.87% (10,565)	96.57% (10,532)	93.33% (10,179)

Excluding FeatMaker, the results for bug-finding are consistent with the branch coverage results in Figure 2. The two state-of-the-art strategies, ParaDySE and Learch, outperformed the other two search strategies that rely on a state feature, taking 2nd and 3rd places, respectively. However, neither ParaDySE nor Learch emerged as a clear winner, as each missed bugs that the other found. Specifically, ParaDySE failed to detect two bugs in `gcal` and `find` that Learch did find, while Learch missed three bugs in `gcal` and `combine` that ParaDySE discovered. The remaining strategies, SGS and Depth, found only two and one distinct bugs, respectively, which implies that relying on only a single state feature is insufficient for effective bug-finding.

Reproducibility. We investigated the reproducibility of the six unique bugs found solely by FeatMaker, as listed in Table 2. To achieve this, we first gathered all the pairs used when finding these six bugs, where each pair (F, w) consists of state features and a weight vector. Intuitively, the number of these pairs listed in Table 3 indicates how many times each bug was discovered. Then, using each pair (F, w), we performed symbolic execution for 120 seconds to determine whether FeatMaker could consistently rediscover the bug within the time period. Each experiment was conducted 100 times, and we reported the average bug reproducibility.

Table 3 reveals that FeatMaker achieved an average bug reproducibility of approximately 89.3% across 33 pairs used to identify six unique bugs, suggesting that these findings are not coincidental. Remarkably, FeatMaker attained a 100% recall rate for the 12 pairs used to detect three bugs in `gcal` and `gawk`. In other words, when symbolic execution was run 100 times for 120 seconds using each of these 12 pairs of state features and a weight vector, FeatMaker succeeded in finding all the corresponding bugs without a single failure during a total of 1200 trials. For the remaining three bugs, FeatMaker also demonstrated acceptable reproducibility rates. For instance, it achieved 87% and 86% reproducibility for bugs found in `gcal` and `find`, respectively. These results show an unexpected advantage of FeatMaker: its ability to deterministically reach a specific point in the target program, such as a bug point.

Bug-Finding Capability for Injected Faults. We further evaluated the bug-finding capability of FeatMaker and four baseline techniques through mutation testing [17, 22, 29], which involved injecting artificial faults (e.g., simple syntactic changes) into our benchmark programs. Initially, we generated "mutants" for the original program using the open-sourced mutant generation tool, MART [14], where a mutant represents a faulty program with a minor syntactic alteration from the original. The effectiveness of each technique in uncovering these faults was determined by the number of mutants it successfully "killed"; a mutant is considered killed if its execution result (e.g., EXIT STATUS) differs from that of the original program for any of the test-cases generated by the technique. We focused on two benchmarks, `find` and `combine`, for which MART successfully generated mutants among all the benchmarks in Table 2. Specifically, MART generated 388,520 mutants for `find` and 109,060 mutants for `combine`. To streamline the evaluation process, we randomly sampled 10% of these mutants for each benchmark [46, 47]. We then assessed the effectiveness of FeatMaker and the four baselines by calculating the number of mutants each killed within 24 hours.

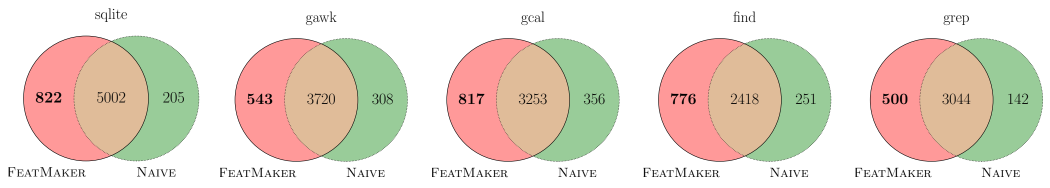

Fig. 3. The average number of covered branches achieved by FeatMaker and naive approach on 5 benchmarks

Table 4 shows that FeatMaker consistently surpassed the four baselines in killing mutants across both benchmarks. Notably, for `find`, FeatMaker killed an average of 34,828 mutants, significantly outperforming ParaDySE (17,382 mutants) and Learch (28,954 mutants). Similarly, for `combine`, the test-cases of FeatMaker killed 99.95% of the sampled mutants, surpassing the second-best performance of 96.87%. These results highlight the ability of FeatMaker to produce high-quality test-cases, demonstrating its superiority over existing search strategies in mutation testing.

4.4 Comparison with A Naive Approach

We experimentally evaluate the effectiveness of FeatMaker by comparing it with a naive approach. Specifically, the naive approach (Naive) employs all branch conditions found during symbolic execution as state features and uses weight vectors sampled from the uniform distributions. In other words, it omits the Cluster and Learn steps outlined in FeatMaker (Algorithm 2). For our assessment, we performed symbolic execution using both FeatMaker and the naive approach on five of our largest benchmarks: `sqlite`, `gawk`, `gcal`, `find`, and `grep`. We allocated a 24-hour testing budget for each benchmark and repeated the process five times to obtain average results.

Figure 3 presents Venn diagrams that illustrate the unique branches covered by FeatMaker and the naive approach (Naive), respectively. The results show that FeatMaker significantly outperforms Naive. On average, FeatMaker covered 193% more unique branches than Naive across the five largest benchmarks. For instance, in the largest benchmark, `sqlite`, FeatMaker exclusively covered 822 branches while Naive reached only 205, making FeatMaker about four times more effective over the same 24-hour period. Similar results were obtained in the `grep` benchmark, where FeatMaker covered approximately 3.5 times more unique branches. Overall, FeatMaker succeeded in covering an average of 12.4% more branches than Naive on all the five benchmark programs. These findings underscore the importance of the main components of FeatMaker, including Cluster and Learn, in efficiently generating state features and scoring functions.

The number of State Features. We further evaluated how effectively FeatMaker mitigates the surge in state features compared to the naive approach (Naive). For our five largest benchmarks, we compared the state features used by FeatMaker and Naive at the end of the 24-hour testing period. Figure 4 depicts the average number of state features employed by each approach.

Figure 4 reveals that FeatMaker consistently uses fewer state features than Naive. On average, FeatMaker retains only 17.6% of the number of state features that Naive does. For instance, in the largest benchmark, `sqlite`, Naive generated 18,124 state features within 24 hours, whereas FeatMaker produced just 782—a mere 4.3% of the number of Naive's state features. Even in the smallest benchmark, `grep`, FeatMaker reduced the state features by approximately 85.4% compared to Naive.

Considering the results from Figures 3 and 4 together, there seems to be a correlation between the reduction in state features achieved by FeatMaker and its exclusive branch coverage. For

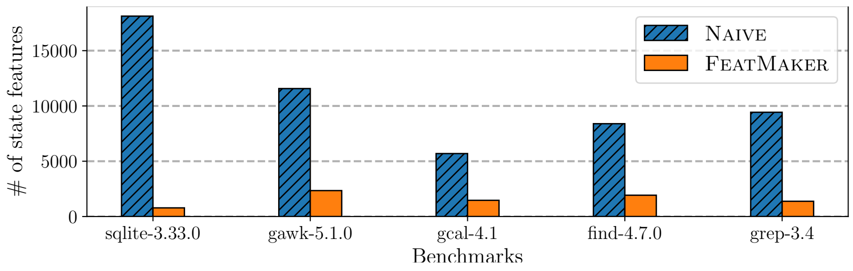

Fig. 4. The average number of state features generated by FeatMaker and naive approach on 5 benchmarks

Table 5. Performance depending on different values for each hyper-parameter

hyperparameters	η_k			η_{time}		
benchmarks	10	20	30	60	120	180
sqlite-3.33.0	5744.6	**5823.6**	5645.8	5754.4	**5823.6**	5680.2
gawk-5.1.0	4232.0	**4263.4**	4031.4	4132.8	**4263.4**	4136.2
gcal-4.1	3945.4	**4069.8**	3810.8	3958.8	**4069.8**	3976.4
find-4.7.0	2935.4	**3193.6**	3012.0	2945.8	3193.6	**3224.4**
grep-3.4	3551.6	3544.0	**3572.0**	3504.8	**3544.0**	3522.0

`sqlite`, where FeatMaker achieved the most significant reduction at 95.7%, it also quadrupled the exclusive branch coverage compared to Naive. Similarly, for `grep`, which had the second highest reduction rate at 85.4% with FeatMaker, there was a 3.5-fold increase in the number of exclusively covered branches.

4.5 Impact of Hyper-parameters

We evaluated how the performance of FeatMaker varies with different values of its two hyper-parameters. FeatMaker (Algorithm 2) incorporates two hyper-parameters: η_k and η_{time}. By default, these are set to 20 and 120 seconds, respectively. The η_k parameter determines the update frequency for state features and scoring functions, while η_{time} specifies the time allocated for a single symbolic execution run. Intuitively, reducing the values of these parameters leads to more frequent updates. For each hyper-parameter, we evaluated the performance of FeatMaker when using values both below and above the default on our five largest benchmarks.

Table 5 shows that the default hyper-parameter values chosen for our experiments yielded overall good performance across the five benchmarks. Specifically, for η_k, a value of 20 provided the most stable results. While a η_k value of 10 achieved the highest branch coverage for `gawk`, it lagged behind in the coverage for `gcal` and `find`. In contrast, with η_k set to 20, even the benchmarks where it did not secure the top position showed reasonable performance. As for η_{time}, a duration of 120 seconds appeared optimal for four benchmarks. A shorter duration of 60 seconds compromised performance due to the limited time for a single symbolic execution run, while extending it to 180 seconds did not allow for sufficient updates, resulting in inconsistent outcomes for some programs.

4.6 Threats to Validity

(1) We selected to implement our approach, FeatMaker, on top of KLEE. This choice is because it is a representative symbolic execution tool. Furthermore, existing state-of-the-art search strategies, including ParaDySE [11] and Learch [27], are built on KLEE. However, our experimental

results obtained from KLEE might not be generalizable to other symbolic execution tools, such as CREST [15]. (2) FeatMaker involves two hyper-parameters that we manually adjusted based on trial and error. As highlighted in Section 4.5, these values yielded stable results for our benchmark suite, but they might not be the optimal settings. (3) For evaluation, we utilized 15 benchmark programs from prior work that aimed to enhance the search strategies of symbolic execution. However, these benchmark programs might not be enough to draw definitive conclusions.

5 RELATED WORK

In this section, we discuss existing works that aim to enhance the performance of symbolic execution by mitigating the notorious state-explosion problem. Specifically, we focus on two main approaches closely related to ours: search strategies [7, 11, 12, 27, 33, 37, 39] and pruning strategies [5, 6, 13, 28, 42, 45]. Search strategies aim to preferentially select promising program states from the candidates maintained during symbolic execution. On the other hand, pruning strategies aim to eliminate unpromising program states from the candidates. Additionally, we discuss search-based software testing [3, 4, 18, 23, 24, 26, 35] as our technique can be seen as a specific instance of SBST.

5.1 Search Strategies

To the best of our knowledge, FeatMaker is the first technique to fully automate the process of generating search strategies of symbolic execution, including the design of state features. Existing search strategies [7, 11, 12, 27, 33, 37, 39] have been designed either manually or semi-automatically, as they all rely on a fixed set of manually-tuned state features. Traditional search strategies [7, 8, 33, 37] primarily focused on designing a single general feature expected to perform well across various programs. The Control-Flow-Directed Search strategy [7] employed a minimum distance from uncovered branches as a feature. The Subpath-Guided Search strategy [33] considered the number of times that a k-length subpath of each state was reached. More recently, state-of-the-art search strategies [11, 12, 27] have shifted from focusing only on a specific property of program states. Instead, they manually design multiple atomic features, represent program states into multiple feature vectors, and learn which state to explore first. While these strategies have partially automated the design process, they still leave the labor-intensive task of feature engineering to manual efforts. In contrast, FeatMaker determines which states to explore first based on automatically-generated state features, achieving a truly automated approach to search strategies.

5.2 Pruning Strategies

Our technique (FeatMaker) is orthogonal to techniques that prune redundant program states or paths for symbolic execution [5, 6, 13, 28, 42, 45]. While FeatMaker prioritizes the selection of promising states from the pool of candidate states maintained during symbolic execution, these pruning techniques focus on eliminating states deemed unpromising for bug-finding and code coverage. Each pruning strategy employs specific criteria to determine the redundancy of program paths or states. Boonstoppel et al. [5] introduced a criterion that prunes program paths that will exercise the exactly same basic blocks as paths already explored. Similarly, Yi et al. [45] defined a criterion that discards program states sharing the exact same path suffixes as states previously explored during symbolic execution. Jaffar et al. [28] presented a path-pruning strategy for concolic testing, which omits execution paths guaranteed not to reach annotated bug points in the target program. Cha et al. [13] introduced learning-based state-pruning strategies aiming to minimize the total number of maintained states while preserving promising states for symbolic execution. Our approach, FeatMaker, can be integrated with these pruning strategies, potentially further boosting the efficiency of symbolic execution.

5.3 Search-Based Software Testing

At a high level, our technique, FeatMaker, can be categorized as a specific instance of Search-Based Software Testing (SBST) [3, 4, 18, 23, 35] and Search-Based Software Engineering (SBSE) [1, 2, 24–26]. Designing a testing technique in SBST and SBSE can be viewed as solving an optimization problem. The goal is to effectively find optimal solutions in a vast search space within a limited time budget using meta-heuristic search techniques. In our work, we formulated FeatMaker as a multi-object optimization problem. It aims to find subsets from the set of all candidate state features generated during symbolic execution and to identify scoring functions from n-dimensional real-number vector spaces. We present a customized algorithm that iteratively generates and refines both state features and scoring functions based on data accumulated during symbolic execution.

6 CONCLUSION

Recent search strategies that utilize multiple state features have significantly surpassed traditional strategies that rely on a single state feature. However, the process of designing these features remains predominantly manual. In this paper, we aim to automatically generate state features tailored to individual programs, setting the stage for a truly automated and optimized search strategy. Our key idea is to utilize path-conditions, which are fundamental pieces of information inherent in states maintained during symbolic execution. Leveraging this, we present a specialized algorithm that iteratively generates and refines both state features and scoring functions, aiming to maximize the performance of symbolic execution. Experimental results on open-source C programs indicate that FeatMaker, with its automatically generated state features, substantially outperforms existing search strategies that rely on manually-crafted multiple state features, both in terms of branch coverage and bug detection. We believe that FeatMaker has the potential to revolutionize the challenging task of creating new search strategies for symbolic execution.

REFERENCES

[1] Andrea Arcuri and Gordon Fraser. 2011. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*. 33–47.

[2] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* (2013), 594–623.

[3] Andrea Arcuri and Xin Yao. 2008. Search based software testing of object-oriented containers. *Information Sciences* (2008), 3075–3095.

[4] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. 2011. Symbolic search-based testing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011) (ASE '11)*. 53–62.

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

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

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

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. 209–224.

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

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

[11] Sooyoung Cha, Seongjoon Hong, Jiseong Bak, Jingyoung Kim, Junhee Lee, and Hakjoo Oh. 2022. Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics. *IEEE Transactions on Software Engineering* (2022), 3640–3663.

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

[13] Sooyoung Cha and Hakjoo Oh. 2020. Making Symbolic Execution Promising by Learning Aggressive State-Pruning Strategy. In *The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*.

[14] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: a mutant generation tool for LLVM. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1080–1084. https://doi.org/10.1145/3338906.3341180

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

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. 337–340.

[17] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* (1978), 34–41.

[18] Gordon Fraser and Andrea Arcuri. 2012. The seed is strong: Seeding strategies in search-based software testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*. 121–130.

[19] Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification (CAV '07)*. 519–531.

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

[21] Patrice Godefroid, Michael Y Levin, and David A Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS '08)*. 151–166.

[22] R.G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering* (1977), 279–290.

[23] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST '15)*. 1–12.

[24] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* (2001), 833–839.

[25] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* (2012), 1–61.

[26] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. 2010. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*. 1–59.

[27] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. 2526–2540.

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

[29] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering (TSE)* (2011), 649–678.

[30] David S. Johnson. 1973. Approximation Algorithms for Combinatorial Problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. 38–49.

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

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

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

[34] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[35] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 153–163.

[36] OSDI'08_Coreutil_Experiments. 2008. https://klee.github.io/docs/coreutils-experiments.

[37] Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. 2012. CarFast: Achieving Higher Statement Coverage Faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. 35:1–35:11.

[38] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*. 263–272.

[39] Hyunmin Seo and Sunghun Kim. 2014. How We Get There: A Context-guided Search Strategy in Concolic Testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. 413–424.

[40] ParaDySE. A tool for automatically generating search strategy of symbolic execution. 2022. https://github.com/kupl/dd-klee/tree/master/paradyse.

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

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

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

[44] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 359–368.

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

[46] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 92–102. https://doi.org/10.1109/ASE.2013.6693070

[47] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. 2010. Is operator-based mutant selection superior to random mutant selection?. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 435–444. https://doi.org/10.1145/1806799.1806863