Automatically Generating Search Heuristics for Concolic Testing

Sooyoung Cha
Korea University
sooyoungcha@korea.ac.kr

Seongjoon Hong
Korea University
seongjoon@korea.ac.kr

Junhee Lee
Korea University
junhee_lee@korea.ac.kr

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

ABSTRACT

We present a technique to automatically generate search heuristics for concolic testing. A key challenge in concolic testing is how to effectively explore the program's execution paths to achieve high code coverage in a limited time budget. Concolic testing 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 concolic testing by automatically generating search heuristics. We define a class of search heuristics, namely a parameterized heuristic, and present an algorithm that efficiently finds an optimal heuristic for each subject program. Experimental results with open-source C programs show that our technique successfully generates search heuristics that significantly outperform existing manually-crafted heuristics in terms of branch coverage and bug-finding.

CCS CONCEPTS

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

ACM Reference Format:
Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180166

1 INTRODUCTION

Concolic testing [15, 28] has emerged as an effective software-testing method with diverse applications [1, 7, 21, 30, 33]. The idea of concolic testing is to symbolically execute a program alongside the concrete execution, where the main job of the symbolic execution is to collect path conditions. Initially, the program is executed with a random input. After the program finishes, a branch of the current path is selected and negated to find an input that drives the next program execution to follow a previously unexplored path. This way concolic testing systematically explores the execution paths of the program, greatly improving random testing.

A key component of concolic testing is the so-called search heuristic. Because of the path-explosion problem, exploring all execution paths of a nontrivial program is simply impossible. Instead, concolic testing relies on a search heuristic to maximize code coverage in a limited time budget. A search heuristic has a criterion and steers concolic testing by choosing the best branch to negate according to the criterion. For example, the CFDS (Control-Flow Directed Search) heuristic [3] picks the branch that is closest to the uncovered regions of the program and the CGS (Context-Guided Search) heuristic [29] selects a branch only if it is in a new context. It is well-known that the effectiveness of concolic testing depends heavily on the choice of the search heuristic [3, 21, 27, 29].

However, manually designing such a heuristic is challenging. It is not only nontrivial but also likely to deliver sub-optimal and unstable results. As we demonstrate in this paper, no manually-designed existing heuristics consistently achieve good code coverage in practice. For example, the CGS heuristic is arguably a state-of-the-art and outperforms existing approaches for a number of programs [29]. However, we found that CGS is sometimes brittle and inferior even to a random heuristic. Furthermore, existing search heuristics came from a huge amount of engineering effort and domain expertise. The difficulty of manually coming up with a good search heuristic is a major remaining challenge in concolic testing.

To address this challenge, this paper presents a new approach that automatically generates search heuristics for concolic testing. To this end, we use two key ideas. First, we define a *parameterized search heuristic*, which creates a large class of search heuristics. The parameterized heuristic reduces the problem of designing a good search heuristic into a problem of finding a good parameter value. Second, we present a search algorithm specialized to concolic testing. The search space that the parameterized heuristic poses is intractably large. Our algorithm effectively guides the search by iteratively refining the search space based on the feedback from previous runs of concolic testing.

Experimental results show that automatically-generated heuristics by our approach outperform existing manually-crafted heuristics for a range of C programs. We have implemented our technique in CREST [3] and evaluated it on 10 C programs (0.5–150KLoC). 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.

This paper makes the following contributions:

- We present a new approach for automatically generating search heuristics for concolic testing. Our work represents a significant departure from prior work; while existing work (e.g. [3, 21, 27, 29]) focuses on manually developing a particular search heuristic, our goal is to automate the very process of generating such a heuristic.

*Corresponding author

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180166

- We present a parameterized search heuristic and an efficient algorithm for finding good parameter values.
- We extensively evaluate our approach with open-source C programs. We make our tool and data publicly available.[1]

2 PRELIMINARIES

In Section 2.1, we define a generic concolic testing algorithm. Section 2.2 discusses existing search heuristics and their limitations.

2.1 Concolic Testing

Concolic testing is a hybrid software testing technique that combines symbolic [24] and concrete executions to systematically explore the program's execution paths.

Concolic testing begins with executing the subject program P with an initial input v_0. During the concrete execution, concolic testing maintains a *symbolic memory state* S and a *path condition* Φ. 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 α and $\beta + 1$ where α and β 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 an assume statement *assume*(e) is encountered. For instance, when $S = [x \mapsto \alpha]$ and $e = x < 1$, the path condition Φ gets updated by $\Phi \wedge (\alpha < 1)$.

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 ϕ_i and generates the new path condition Φ' as follows: $\Phi' = \bigwedge_{j<i} \phi_j \wedge \neg\phi_i$. That is, the new condition Φ' has the same prefix as Φ up to the i-th branch with ϕ_i negated, so that input values that satisfy Φ' drive the program execution to follow the opposite branch of ϕ_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 the concolic testing algorithm. 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. 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 Φ_m explored. The path condition is appended to T (line 5). In lines 6–8, the algorithm chooses a branch to negate. The function Choose first chooses a path condition Φ from T, then selects a branch, i.e., ϕ_i, from Φ. Once a branch ϕ_i is chosen, the algorithm generates the new path condition $\Phi' = \bigwedge_{j<i} \phi_j \wedge \neg\phi_i$. If Φ' is satisfiable, the next input vector is computed (line 9), where SAT(Φ) returns true iff Φ is satisfiable and model(Φ) finds an input vector v which is a model of Φ, i.e., $v \models \Phi$. Otherwise, if Φ' is unsatisfiable, the algorithm repeatedly tries to negate another branch until a satisfiable path condition is found. This procedure repeats for the given budget N and the final number of covered branches |Branches(T)| is returned.

[1] https://github.com/kupl/ParaDySE

Algorithm 1: Concolic Testing

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

```
1:  T ← ⟨⟩
2:  v ← v_0
3:  for m = 1 to N do
4:      Φ_m ← RunProgram(P, v)
5:      T ← T · Φ_m
6:      repeat
7:          (Φ, φ_i) ← Choose(T)        (Φ = φ_1 ∧ ··· ∧ φ_n)
8:      until SAT(⋀_{j<i} φ_j ∧ ¬φ_i)
9:      v ← model(⋀_{j<i} φ_j ∧ ¬φ_i)
10: end for
11: return |Branches(T)|
```

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 concolic testing in a way to maximize code coverage in a given limited time budget [6]. The goal of this paper is to automatically generate an effective heuristic for a given program.

2.2 Existing Search Heuristics

Before presenting our technique, we describe two notable search heuristics. These heuristics are known to perform comparatively better than other heuristics [3, 29].

Control-Flow Directed Search (CFDS) [3]. CFDS 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 Φ_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) [29]. CGS basically 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. The search 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.

Limitations. Existing search heuristics have a key limitation; they rely on a fixed heuristic 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 significantly varies depending on the target programs. For example, CGS outperforms other existing heuristics for several benchmarks: e.g., `expat-2.1.0` and `grep-2.2` (Figure 1). However, we found that the CGS heuristic is sometimes inferior even to the random heuristic (e.g., `tree-1.6.0`). That is, the key feature, *contexts*, of CGS is not appropriate for some programs.

Besides their sub-optimality, another key 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 concolic testing depends heavily on the search heuristic, ordinary developers cannot fully benefit from concolic testing. These observations motivated us to develop a technique that automatically generates search heuristics.

3 OUR TECHNIQUE

In this section, we present our technique for automatically generating search heuristics for concolic testing. We define a family of search heuristics, namely parameterized search heuristics (Section 3.1), and present an algorithm to choose the best heuristic from the family for a given subject program (Section 3.2).

3.1 Parameterized Search Heuristic

Let $P \in \mathit{Program}$ 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:

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

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.

We define a family $\mathcal{H} \subseteq \mathit{SearchHeuristic}$ of search heuristics as a parameterized heuristic Choose_θ, where θ 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 parameterized search heuristic is defined as follows:

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

Intuitively, the heuristic first chooses the last path condition Φ_m from the execution tree T, then selects a branch ϕ_j from Φ_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 ϕ in Φ_m with respect to a given parameter θ:

(1) We represent the branch by a feature vector. We designed 40 boolean features describing properties of branches in concolic testing. A feature π_i is a boolean predicate on branches: $\pi_i : \mathit{Branch} \to \{0, 1\}$. 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 θ, a branch ϕ is represented by a boolean vector as follows:

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

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	true branch of a loop
3	false branch of 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	true branch of a case statement
12	false branch of a 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

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

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

(3) Finally, we choose the branch with the highest score. That is, among the branches $\phi_1, \ldots, \phi_n$ in Φ_m, we choose the branch ϕ_j such that $\mathit{score}_\theta(\phi_j) \geq \mathit{score}_\theta(\phi_k)$ for all k.

Branch Features. 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.

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 true and false branches of loops have different roles; by giving a high score to a true branch we can explicitly steer concolic testing away from the loop (i.e. negating the true branch) while giving a high score to a false branch leads to getting into the loop.

On the other hands, we designed dynamic features (13-40) to capture the dynamics of concolic testing. For instance, feature 24 checks whether the branch has been negated more than 10 times during concolic testing. That is, during the execution of 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.

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 [29] while features 30-31 are based on the idea of the CFDS heuristic [3] that calculates the distance to uncovered branches.

3.2 Parameter Optimization Algorithm

Now we describe our algorithm for finding a good parameter value of the parameterized search heuristic. We formally define the optimization problem, and then present our algorithm.

Optimization Problem. In our approach, finding a good search heuristic corresponds to solving an optimization problem. We model the concolic testing procedure in Algorithm 1 by the function:

$$C : \mathit{Program} \times \mathit{SearchHeuristic} \to \mathbb{N}$$

which takes a program and a search heuristic, and returns the number of covered branches. Given a program P and a search heuristic Choose, $C(P, \mathsf{Choose})$ performs concolic testing (Algorithm 1) using the heuristic for a fixed number of executions (i.e. N). We assume that the initial input (v_0) and the number of executions (N) are fixed for the program.

Given a program P to test, our goal is to find a parameter θ that maximizes the performance of the concolic testing algorithm with respect to P. Formally, our objective is to find θ^* such that

$$\theta^* = \underset{\theta \in \mathbb{R}^k}{\operatorname{argmax}}\, C(P, \mathsf{Choose}_\theta). \tag{1}$$

That is, we aim to find a parameter θ^* that causes the concolic testing algorithm C with the search heuristic Choose_θ to maximize the number of covered branches in P.

Optimization Algorithm. We propose an algorithm that efficiently solves the optimization problem in (1). A simplistic approach to solve the problem would be random sampling, which randomly samples parameter values and returns the best parameter 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 of concolic testing (Section 4.3). This is mainly because of two reasons. First, the search space is intractably large and therefore blindly searching for good parameters without any guidance is hopeless. Second, a single evaluation of a parameter value is generally unreliable and does not represent the average performance in concolic testing. This performance variation arises from the inherent nondeterminism in concolic testing (e.g. branch prediction failure) [15].

In response, we designed an optimization algorithm (Algorithm 2) specialized to efficiently finding good parameter values of search heuristics. The key idea behind this algorithm is to iteratively refine the sample space based on the feedback from previous runs of concolic testing. 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, $\mathbb{R}_i$ ($i \in [1, k]$), where k is the dimension of the parameters (i.e., the number of branch features in our parameterized heuristic). In our algorithm, the i-th components of the parameters are sampled from $\mathbb{R}_i$, independently from other components. For all i, $\mathbb{R}_i$ is initialized to the space $[-1, 1]$.

In the first phase (*Find*), we randomly sample n parameter values: $\theta_1, \theta_2, \ldots, \theta_n$ from the current sample space $\mathbb{R}_1 \times \mathbb{R}_2 \times \cdots \times \mathbb{R}_k$ (line 7), and their performance numbers (i.e., the number of branches covered) are evaluated (lines 9–11). In experiments, we set n to 1,000 (300 for vim). Among the 1,000 parameters, we choose the top K parameters according to their branch coverage. In our experiments, K is set to 10 because we observed that parameters with good qualities are usually found in the top 10 parameters. This first step of executing a program 1,000 times can be run in parallel.

In the next phase (*Check*), we choose the top 2 parameters that show the best average performance. At lines 16–17, the K parameters chosen from the first phase are evaluated again to obtain the average code coverage over 10 trials, where B_i^* represents the average performance of parameter θ_i'. At line 19, we choose two parameters θ_{t_1} (top 1) and θ_{t_2} (top 2) with the best average performance. This step (*Check*) is needed to rule out unreliable parameters. Because of the nondeterminism of concolic testing, the quality of a search heuristic must be evaluated over multiple executions.

In the third step (*Refine*), we refine the sample spaces $\mathbb{R}_1, \ldots, \mathbb{R}_k$ based on θ_{t_1} and θ_{t_2}. Each $\mathbb{R}_i$ is refined based on the values of the i-th components ($\theta_{t_1}^i$ and $\theta_{t_2}^i$) of θ_{t_1} and θ_{t_2}. When both $\theta_{t_1}^i$ and $\theta_{t_2}^i$ are positive, we modify $\mathbb{R}_i$ by $[\min(\theta_{t_1}^i, \theta_{t_2}^i), 1]$. When both $\theta_{t_1}^i$ and $\theta_{t_2}^i$ are negative, $\mathbb{R}_i$ is refined by $[-1, \max(\theta_{t_1}^i, \theta_{t_2}^i)]$. Otherwise, $\mathbb{R}_i$ remains the same. Then, our algorithm goes back to the first phase (*Find*) and randomly samples n parameter values 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 30–31). This way, we iteratively refine each sample space $\mathbb{R}_i$ and guide the search to continuously find and climb the hills toward top in the parameter space.

Algorithm 2: Our 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 $\mathbb{R}_i = [-1, 1]$ for $i \in [1, k]$
3: $\langle max, \text{converge} \rangle \leftarrow \langle 0, false \rangle$
4: **repeat**
5: /* Step 1: Find */
6: /* sample n parameters: $\theta_1, \ldots, \theta_n$ (e.g., n=1,000) */
7: $\{\theta_i\}_{i=1}^n \leftarrow$ sample from $\mathbb{R}_1 \times \mathbb{R}_2 \times \cdots \times \mathbb{R}_k$
8: /* evaluate the sampled parameters */
9: **for** $i = 1$ **to** n **do**
10: /* B_i: branch coverage achieved with θ_i */
11: $B_i \leftarrow C(P, \text{Choose}_{\theta_i})$
12: **end for**
13: pick top K parameters $\{\theta'_i\}_{i=1}^K$ from $\{\theta_i\}_{i=1}^n$ with highest B_i
14:
15: /* Step 2: Check */
16: **for all** K parameters θ'_i **do**
17: $B^*_i \leftarrow \frac{1}{10} \sum_{j=1}^{10} C(P, \text{Choose}_{\theta'_i})$
18: **end for**
19: pick top 2 parameters $\theta_{t_1}, \theta_{t_2}$ with highest B^*_i
20:
21: /* Step 3: Refine */
22: **for** $i = 1$ **to** k **do**
23: **if** $\theta^i_{t_1} > 0$ **and** $\theta^i_{t_2} > 0$ **then**
24: $\mathbb{R}_i = [\min(\theta^i_{t_1}, \theta^i_{t2}), 1]$
25: **else if** $\theta^i_{t_1} < 0$ **and** $\theta^i_{t_2} < 0$ **then**
26: $\mathbb{R}_i = [\text{-}1, \max(\theta^i_{t_1}, \theta^i_{t_2})]$
27: **end if**
28: **end for**
29:
30: /* Check Convergence */
31: **if** $B^*_{t_1} < max$ **then**
32: converge $\leftarrow$ *true*
33: **else**
34: $\langle max, \theta_{max} \rangle \leftarrow \langle B^*_{t_1}, \theta_{t_1} \rangle$
35: **end if**
36: **until** converge
37: **return** θ_{max}

4 EXPERIMENTS

In this section, we experimentally evaluate our approach that automatically generates search heuristics of concolic testing. We implemented our approach in CREST [9], a concolic testing tool widely used for C programs [3, 12, 23, 29]. We conducted experiments to answer the following research questions:

- **Effectiveness of generated heuristics**: Does our approach generate effective search heuristics? How do they perform compared to the existing state-of-the-art heuristics?
- **Time for obtaining the heuristics**: How long does our approach take to generate the search heuristics? Is our approach useful even considering the training effort?
- **Efficacy of optimization algorithm**: How does our optimization algorithm perform compared to the naive algorithm by random sampling?
- **Important features**: What are the important features to generate effective search heuristics for concolic testing?

Table 2: 10 benchmark programs

Program	# Total branches	LOC	Source
vim-5.7	35,464	165K	[3]
gawk-3.0.3	8,038	30K	ours
expat-2.1.0	8,500	49K	[29]
grep-2.2	3,836	15K	[3]
sed-1.17	2,656	9K	[22]
tree-1.6.0	1,438	4K	ours
cdaudio	358	3K	[29]
floppy	268	2K	[29]
kbfiltr	204	1K	[29]
replace	196	0.5K	[3]

Evaluation Setting. We have compared our approach with five existing heuristics: CGS (Context-Guided Search) [29], CFDS (Control-Flow Directed Search) [3], Random branch search [3], DFS (Depth-First Search) [15], and Generational search [16]. We chose these heuristics for comparison because they have been commonly used in prior work [3, 10, 15, 16, 29]. In particular, CGS and CFDS are arguably the state-of-the-art search heuristics that often perform the best in practice [3, 29]. The implementation of CFDS, Random, and DFS heuristics are available in CREST. The implementations of CGS and Generational search came from the prior work [29].[2]

We used 10 open-source benchmark programs (Table 2).[3] The benchmarks are divided into the large and small programs. The large benchmarks include `vim`, `expat`, `grep`, `sed`, `gawk`, and `tree`. The first four are standard benchmark programs in concolic testing for C, which have been used multiple times in prior work [2, 3, 5, 22, 29]. The last two programs (`gawk` and `tree`) were prepared by ourselves, which are available with our tool. Our benchmark set also includes 4 small ones: `cdaudio`, `floppy`, `kbfiltr`, and `replace`, which were used in [3, 22, 29].

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 (4000 executions, i.e., $N = 4000$ in Algorithm 1). Note that the performance of concolic testing generally 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 Random 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 same input used in prior work [29]. For `grep`, we selected an input on which the random heuristic was effective. The initial inputs 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 [15], the randomness of search heuristics, and so on. We repeated the experiments 100 times for all benchmarks

[2] We obtained the implementation from authors via personal communication.
[3] Henceforth, the version numbers will be omitted when there is no confusion.

Table 3: Average branch coverage on 4 small benchmarks

	OURS	CFDS	**CGS**	Random	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 4: Effectiveness in terms of maximum branch coverage

	OURS	CFDS	CGS	Random	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 5: Effectiveness in terms of finding bugs

	OURS	CFDS	CGS	Random	Gen	DFS
gawk-3.0.3	**100/100**	0/100	0/100	0/100	0/100	0/100
grep-2.2	**47/100**	0/100	5/100	0/100	0/100	0/100

except for vim for which we averaged over 50 trials as its execution takes much longer time. The experiments were done on a linux machine with two Intel Xeon Processor E5-2630 and 192GB RAM.

4.1 Effectiveness of Generated Heuristics

For each benchmark program, we ran our algorithm (Algorithm 2) to generate our search heuristic (ours), and compared its performance with that of the existing heuristics. We evaluate the effectiveness with two measures: branch coverage and capability to find bugs.

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

Figure 1 compares the average branch coverage achieved by different search heuristics on 6 large benchmarks. 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 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 CFDS. On small benchmarks, we obtained similar results; ours (together with CGS) consistently achieved the highest average coverage (Table 3). 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 1, 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 1, the p value was less than 0.01 according to Wilcoxon signed-rank test. In Figure 1, the standard deviations for each heuristic are as follows: (1) OURS: vim(258), expat(42), gawk(0), grep(51), sed(22), tree(7); (2) CFDS: vim(252), expat(44), gawk(120), grep(33), sed(24), tree(13); (3) CGS: vim(200), expat(24), gawk(57), grep(29), sed(27), tree(15). Other search heuristics also have similar standard deviations.

In Figure 1, we compared the effectiveness of search heuristics over iterations (# of executions)[4], but our approach was also superior to others over execution time. For example, given the same time budget (1,000 sec), ours and Random (the second best) covered 8,947 and 8,272 branches, respectively, for vim (Figure 2). The results were averaged over 50 trials.

Table 4 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. 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 Random 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 Generational and Random heuristics surpassed CFDS and CGS in expat and tree, respectively. On the other hand, our approach is able to consistently produce the best search heuristics in terms of both coverage metrics.

Bug Finding. We found that the increased branch coverage by our approach leads to more effective finding of real bugs (not seeded ones). Table 5 reports the number of trials that successfully generate test-cases, which trigger the known performance bugs in gawk and grep [13, 18]. 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.

[4] Evaluating the performance of search heuristics over iterations is a common practice [3, 29], as the execution time of a program may vary considerably depending on the input.

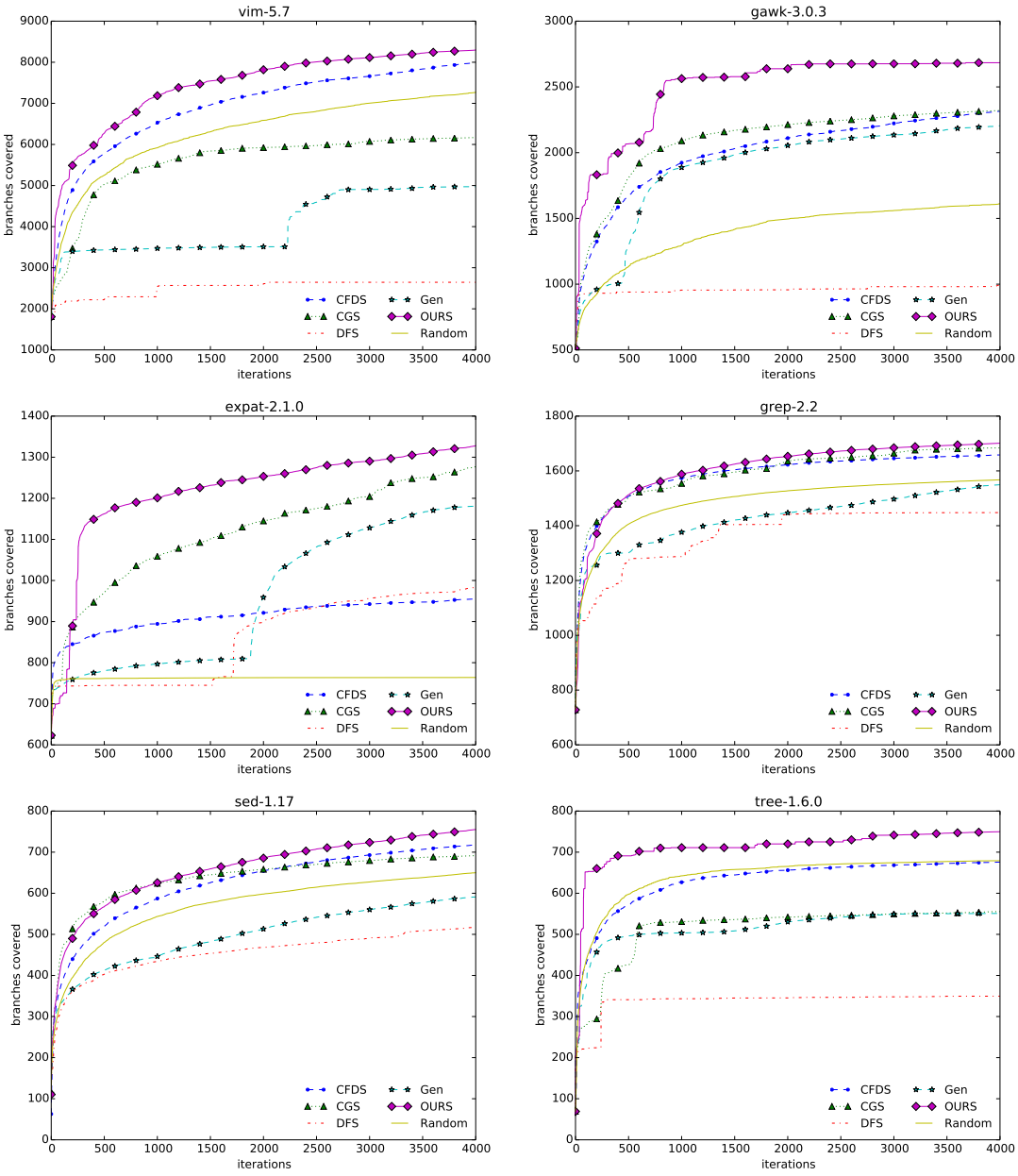

Figure 1: Average branch coverage achieved by each search heuristic on 6 large benchmarks

Our heuristics are good at finding bugs because they 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, the bugs are caused only by some specific path conditions and the existing heuristics could not generate inputs that satisfy the conditions.

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` [13, 18] cause performance problems; for example, `grep-2.2` requires exponential time and memory on particular input strings that involve back-references [18]. 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.

4.2 Time for Obtaining the Heuristics

Table 6 reports the running time of our algorithm to generate the search heuristics evaluated in Section 4.1. To obtain our heuristics,

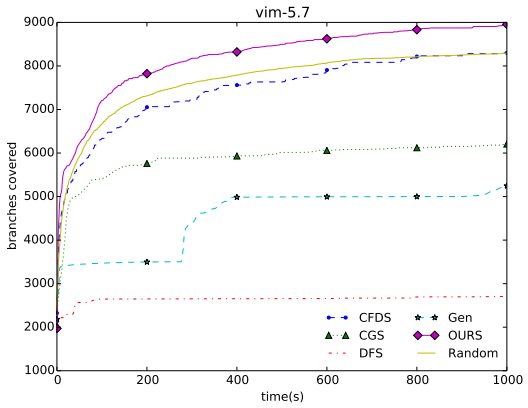

Figure 2: Average branch coverage over execution time

Table 6: 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

Table 7: Effectiveness in the training phase

	OURS	CFDS	CGS	Random	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

we ran the optimization algorithm (Algorithm 2) in parallel using 20 cores. Specifically, in the first phase ('Find') of the algorithm, we sampled 1,000 parameters, where each core is responsible for evaluating 50 parameters. For `vim`, we set the sample size to 300 as executing `vim` is expensive. The results show that our algorithm converges within 4–6 iterations of the outer loop of Algorithm 2, taking 3–24 hours depending on the size of the subject program.

Our approach requires training effort but it is rewarding because 1) our approach enables effective concolic testing even in the training phase; and 2) the learned heuristic can be reused multiple times as the subject program evolves.

Effectiveness in the training phase. Note that running Algorithm 2 is essentially running concolic testing on the subject program. We compared the number of branches covered during this training phase with the branches covered by other search heuristics given the same time budget reported in Table 6. Table 7 compares the results: except for `sed`, running Algorithm 2 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 cores and amount of time. For instance, in 24 hours, Algorithm 2 covered 14,003 branches of `vim` while concolic testing with the CFDS and CGS heuristics covered 13,706 and 7,934 branches, respectively.

Reusability over program evolution. More interestingly, the learned heuristic can be reused over multiple subsequent program variations. To validate this hypothesis, 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 4 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`.

4.3 Efficacy of Optimization Algorithm

We compared the performance of our optimization algorithm (Algorithm 2) with a naive approach based on random sampling. Because both approaches involve randomness, we statistically compare the qualities of parameters found by our algorithm and the random sampling method.

Figure 4 shows the distributions of final coverages achieved by those two algorithms on `grep-2.2` and `sed-1.17`. In the experiments, 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 parameters and the rest for the Find phase to evaluate the parameters 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 parameters. 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.

The result shows that our algorithm is much superior to random sampling method: the median of the samples increases while the variance 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 grows as the tasks repeat. In addition, the variance (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; the heuristics generated by random sampling failed to surpass the existing heuristics. For instance, for `grep`, our algorithm (Algorithm 2) 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.

4.4 Important Features

Winning Features. We discuss the relative importance of features by analyzing the learned parameters θ for each benchmark program. Intuitively, when the i-th component θ^i has a negative

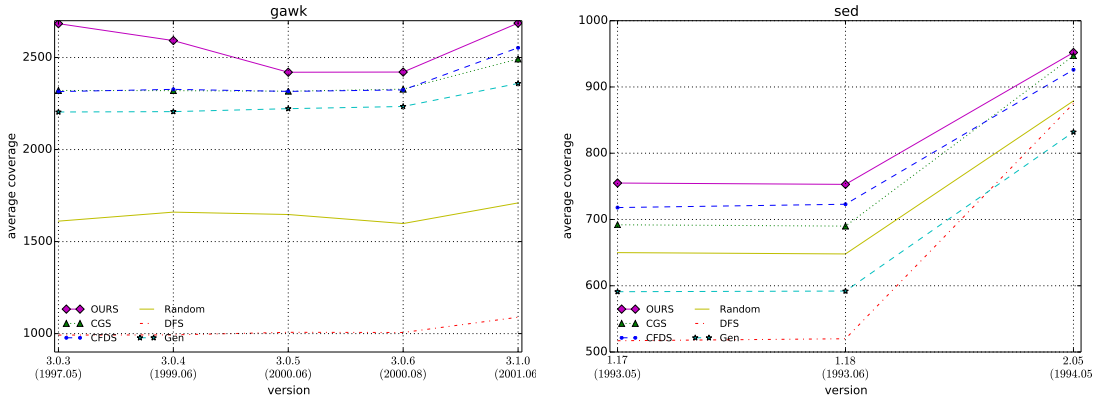

Figure 3: Average coverage of each search heuristic on multiple subsequent program variants

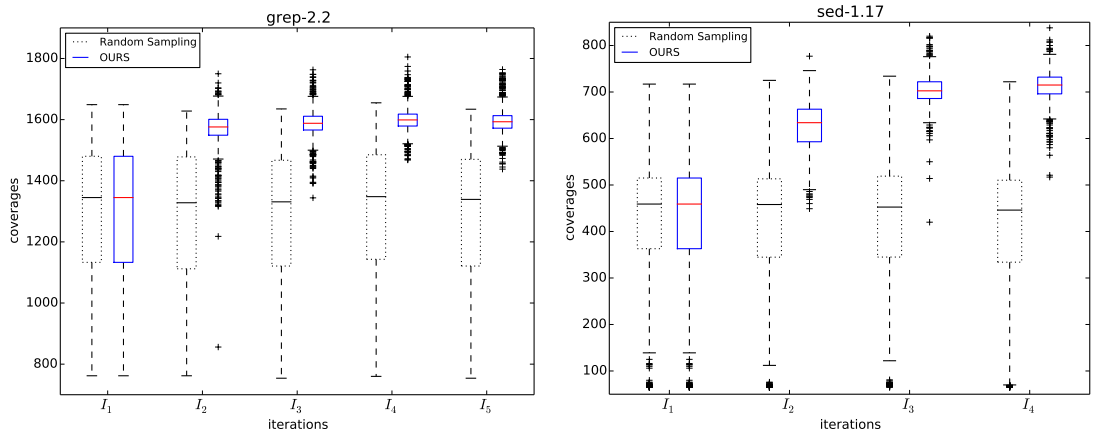

Figure 4: Comparison between our algorithm and random sampling method

Table 8: 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

Table 9: 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(-)**

number in θ, it indicates that the branch having i-th component should not be selected to be negated. Thus, both strong negative and positive features are equally important for our approach to improve the branch coverage. Table 8 and Table 9 show the top 10 positive and negative features for each benchmark, respectively.

The results show that there is no winning feature which always belongs to the top 10 positive or negative features. Nevertheless, 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.

Note that the features 10, 25, 35 and 38 appear in both Table 8 and Table 9. That is, depending on the program under test, the role of each feature changes from positive to negative (or vice versa). For instance, the feature 10 is used as the most positive feature in `gawk` while it is the most negative one for `tree`. This finding supports our claim that no single search heuristic can perform well for all benchmarks, and therefore it should be adaptively tuned for each target program.

Impact of Combining Static and Dynamic Features. The combined use of static and dynamic features was important. 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.

4.5 Threats to Validity

(1) We collected eight benchmarks from prior work [2, 3, 5, 22, 29] and created two new benchmarks (`gawk` and `tree`). However, these 10 benchmarks may not be representative and not enough to evaluate the performance of the search heuristics in general. (2) 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 [3, 29]. However, this might not be a best setting in practice. (3) The performance of search heuristics may vary when using different SMT solvers. We used Yices, the default SMT solver in CREST.

5 RELATED WORK

We discuss existing works on improving the performance of concolic testing. We classify existing techniques into the four classes: (1) improving search heuristics; (2) hybrid approaches; (3) reducing search space; and (4) solving complex path conditions.

Search Heuristics. All existing works on improving search heuristics focus on manually-designing a new strategy [3, 4, 21, 27, 29, 32]. In Section 2.2, we already discussed the CFDS [3] and CGS [29] heuristics. Another successful heuristic is generational search [16], 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. [32] 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 [27] guides concolic testing based on the number of uncovered statements. In [4], several concolic search heuristics are used in a round robin fashion. Our work is different from these works as we automate the heuristic-designing process itself.

Hybrid Approaches. Our approach is orthogonal to the existing techniques that combine concolic testing with other testing techniques. In [12, 26], techniques such as random testing are first used and they switch to concolic testing when the performance gains saturate. In [19], concolic testing is combined with evolutionary testing to be effective for object-oriented programs.

Reducing Search Space. Our work is also orthogonal to techniques that reduce the search space of concolic testing [2, 10, 14, 17, 20]. The read-write set analysis [2] identifies and prunes program paths that have the same side effects. Jaffar et al. [20] introduced an interpolation method that subsumes paths guaranteed not to hit a bug. Goderfroid et al. [14, 17] 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 [10] also reduces search space for concolic testing by using feedback from a model checker. Our work can be combined with these techniques to boost concolic testing further.

Solving Complex Path Conditions. Our technique can also be improved by incorporating existing techniques for solving complex path conditions. Conventional SMT solvers are not effective in handling constraints that involve non-linear arithmetic or external function calls, which often causes concolic testing to have poor coverage. In [11], an algorithm was introduced that can solve hard arithmetic constraints in path conditions. The idea is to generate geometric structures that help solve non-linear constraints with existing heuristics [8]. In [31], a technique to solve string constraints was proposed based on ant colony optimization. There are attempts to solve this problem by machine learning [25]. 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.

6 CONCLUSION

The difficulty of manually crafting good search heuristics has been a major open challenge in concolic testing. In this paper, we addressed this problem with a novel approach for automatically generating search heuristics. Given a program under test, our technique generates an optimal search heuristic for the subject program. Such a "machine-tuned" heuristic has been shown to outperform existing hand-tuned heuristics. To achieve this, we developed a parameterized search heuristic for concolic testing with an optimization algorithm to efficiently search for good parameter values. We hope that our technique can supplant the laborious and less rewarding task of manually tuning search heuristics of concolic testing.

ACKNOWLEDGMENTS

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00565, Development of Vulnerability Discovery Technologies for IoT Software Security). This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09.

REFERENCES

[1] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. 1083–1094.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 144–155.

[8] Philippe Codognet and Daniel Diaz. 2001. Yet another local search method for constraint solving. In *International Symposium on Stochastic Algorithms*. 73–90.

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

[10] Przemysław Daca, Ashutosh Gupta, and Thomas A. Henzinger. 2016. Abstraction-driven Concolic Testing. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583 (VMCAI '16)*. 328–347.

[11] Peter Dinges and Gul Agha. 2014. Solving Complex Path Conditions Through Heuristic Search on Induced Polytopes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. 425–436.

[12] Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. 2013. Feedback-directed Unit Test Generation for C/C++ Using Concolic Execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 132–141.

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

[14] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. 47–54.

[15] 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.

[16] 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.

[17] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 2010. 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 (POPL '10)*. 43–56.

[18] GNU Bug Report (grep). 2014. https://www.gnu.org/software/grep/manual/html_node/Reporting-Bugs.html. (2014).

[19] Kobi Inkumsah and Tao Xie. 2008. 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 (ASE '08)*. 297–306.

[20] 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.

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

[22] Yunho Kim and Moonzoo Kim. 2011. 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 (ESEC/FSE '11)*. 420–423.

[23] Yunho Kim, Moonzoo Kim, YoungJoo Kim, and Yoonkyu Jang. 2012. 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 (ICSE '12)*. 1143–1152.

[24] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

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

[26] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. 416–426.

[27] 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.

[28] 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.

[29] 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.

[30] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS '16)*. 1–16.

[31] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. 2017. Search-driven String Constraint Solving for Vulnerability Detection. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 198–208.

[32] 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.

[33] Yufeng Zhang, Zhenbang Clien, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. 643–653.