Concolic Testing with Adaptively Changing Search Heuristics

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

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

ABSTRACT

We present Chameleon, a new approach for adaptively changing search heuristics during concolic testing. Search heuristics play a central role in concolic testing as they mitigate the path-explosion problem by focusing on particular program paths that are likely to increase code coverage as quickly as possible. A variety of techniques for search heuristics have been proposed over the past decade. However, existing approaches are limited in that they use the same search heuristics throughout the entire testing process, which is inherently insufficient to exercise various execution paths. Chameleon overcomes this limitation by adapting search heuristics on the fly via an algorithm that learns new search heuristics based on the knowledge accumulated during concolic testing. Experimental results show that the transition from the traditional non-adaptive approaches to ours greatly improves the practicality of concolic testing in terms of both code coverage and bug-finding.

CCS CONCEPTS

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

KEYWORDS

Concolic Testing, Dynamic Symbolic Execution, Online Learning

ACM Reference Format:
Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338964

1 INTRODUCTION

Concolic testing [11, 27] is a promising software testing technique popular in both academia and industry [1, 5, 6, 19, 20, 30, 32, 33]. The technique aims to increase code coverage as quickly as possible, ultimately enabling effective bug-finding in a limited time budget. To do so, unlike random testing or fuzzing, concolic testing systematically generates test-cases by repeating the following process: (1) it *concolically* executes the subject program to collect the path condition, i.e., the sequence of symbolic branch conditions exercised by the current program execution, (2) it produces a new path condition by selecting and negating a branch of the current path condition, and (3) it solves the resulting path condition to generate a new test-case that guides the next program execution towards the opposite of the selected branch. Because of this systematic nature, concolic testing is increasingly used in diverse domains, including operating systems [19], embedded systems [10, 14], and even neural networks [30], among others.

Concolic testing includes search heuristics as a critical ingredient. To be practical for real-world applications, concolic testing must be able to adequately address the path-explosion problem; because real-world programs exhibit infinitely many different paths, it is impossible to exercise all of them by testing. To address this challenge, concolic testing uses a search heuristic, a branch selection strategy that takes a path condition and selects a branch based on its own criterion (it is used in the second step of the concolic testing process described in the preceding paragraph). Search heuristics allow concolic testing to preferentially explore particular classes of execution paths that they think are most effective to maximize code coverage within a given time limit. It has been well-known that how to choose and use search heuristics is critically important, and diverse approaches have been proposed to improve concolic testing in practice over the past decade [3–5, 19, 22, 26, 28].

In this paper, we propose a new approach, called Chameleon, for effectively employing search heuristics during concolic testing. The key novelty of Chameleon is *adaptively* changing search heuristics on the fly, so that the branch-selection criterion changes as necessary throughout concolic testing in a way that maximizes the final performance. By contrast, all of the existing approaches for employing search heuristics [3–5, 19, 22, 26, 28] are not adaptive as they use the same search heuristics over the whole process of concolic testing. In this paper, we demonstrate that this is a key limiting factor of the existing approaches, and we can make concolic testing much more practical for real-world applications by being adaptive. We illustrate the limitation of existing search heuristics in more detail in Section 2.

To enable adaptation, we present an algorithm that automatically learns and switches search heuristics during concolic testing. The algorithm maintains a set of search heuristics and continuously changes them during the testing process. To do so, we first define the space of possible search heuristics using the idea of parametric search heuristic recently proposed in prior work [5]. A technical challenge is how to adaptively switch search heuristics in the predefined space. We address this challenge with a new concolic testing algorithm that (1) accumulates the knowledge about the previously evaluated search heuristics, (2) learns the probabilistic distributions of the effective and ineffective search heuristics from the accumulated knowledge, and (3) samples a new set of search heuristics

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338964

from the distributions. The algorithm iteratively performs these three steps until it exhausts a given time budget.

Experimental results demonstrate that shifting from the classical non-adaptive approaches to ours is essential for improving the practicality of concolic testing. We implemented Chameleon on top of CREST [8] and compared it with six existing approaches on 8 open-source C programs (up to 165KLoC). For all benchmarks, Chameleon outperformed all existing non-adaptive search heuristics in terms of both branch coverage and bug-finding in a practical setting. In particular, Chameleon was highly effective in finding various types of bugs, including segmentation faults, abnormal termination, and memory exhaustion. For the latest versions of `vim`, `gawk`, and `grep`, Chameleon succeeded to trigger those bugs whereas all non-adaptive techniques failed to do so.

Contributions. Our contributions are as follows:

- We present Chameleon, a new approach for performing concolic testing, which adaptively learns and changes search heuristics online. To our knowledge, our work is the first that raises the need for adapting search heuristics. Existing works have focused on coming up with new but non-adaptive search heuristics [3–5, 19, 22, 26, 28].
- We provide extensive evaluation by comparing Chameleon with six existing search heuristics in terms of branch coverage and bug-finding. We make our tool[1] and data publicly available.

2 CONVENTIONAL CONCOLIC TESTING

In this section, we describe traditional concolic testing and explain in what sense it is non-adaptive. Algorithm 1 and 2 describe a conventional method for performing concolic testing, which encapsulates the commonality of the approaches used in prior work [3, 5, 11, 12, 28].

The procedure Concolic in Algorithm 1 takes as input a program (P) under test and a search heuristic (Heuristic), runs the program *concolically* with the given search heuristic, and produces as output the set (B) of branches covered during the concolic execution. We assume that an initial input $\mathbf{v_0}$ is fixed and given for the subject program P (line 2). The algorithm initially sets B to the empty set (line 3) and repeats the body of the loop at lines 4–9 for N times, where N determines the number of times to execute the program with the current search heuristic. At line 5, the program is concolically executed with the current input vector $\mathbf{v}$ (i.e., Execute(P, $\mathbf{v}$)), which produces the path condition $\Phi = \phi_1 \wedge \cdots \wedge \phi_n$, i.e., a conjunction of symbolic branch conditions taken in the current execution. For instance, assume that the two branch conditions exercised by the execution are $(x > 1)$ and $(x > 10)$. When the symbolic variable for x is α, the path condition Φ is $\phi_1 \wedge \phi_2$, where $\phi_1 = (\alpha > 1)$ and $\phi_2 = (\alpha > 10)$. At line 6, the algorithm accumulates the covered branches in the set B (where we write Branches(Φ) for the branch ids covered by the current execution path). At line 7, the algorithm uses the search heuristic (Heuristic) to choose a branch ϕ_i to be negated in the next iteration. Then, at line 8, we generate a new input vector $\mathbf{v}$ by finding a model of the constraint $\bigwedge_{j<i} \phi_j \wedge \neg\phi_i$ via an SMT solver.[2] The algorithm repeats the process described so far, and returns the set B upon termination.

Algorithm 1 Basic Concolic Testing Procedure

Input: A program (P) under test and a search heuristic (Heuristic)
Output: The set (B) of covered branches

```
1:  procedure Concolic(P, Heuristic)
2:      v ← v0                                   ▷ initial input v0
3:      B ← ∅
4:      for m = 1 to N do
5:          Φ ← Execute(P, v)                    ▷ Φ = (φ1 ∧ ··· ∧ φn)
6:          B ← B ∪ Branches(Φ)
7:          φi ← Heuristic(Φ)                    ▷ choose a branch
8:          v ← model(⋀_{j<i} φj ∧ ¬φi)
9:      end for
10:     return B
11: end procedure
```

Algorithm 2 Conventional Method for Running Concolic Testing

Input: Program P, A set H of search heuristics
Output: The set T of covered branches

```
1:  procedure Run(P, H)
2:      T ← ∅
3:      repeat
4:          for each h ∈ H do
5:              B ← Concolic(P, h)
6:              T ← T ∪ B
7:          end for
8:      until timeout
9:      return T
10: end procedure
```

Algorithm 2 describes how the procedure Concolic is used in practice. The procedure Run takes a program P under test. Also, in order to generalize existing approaches [3–5, 11, 12, 22, 28], it takes a finite set H of search heuristics as input. Then, the algorithm repeats the following process: 1) it performs Concolic with each heuristic h in H (line 5), and 2) it adds covered branches (B) to the set T of total branches (line 6). When the given time budget is exhausted, the algorithm returns the set of branches covered so far. Readers might wonder why we use Algorithm 2 instead of simply using Algorithm 1 with larger N. In practice, running Algorithm 2 typically performs better than running Algorithm 1 alone because of the randomness of search heuristics. We empirically corroborate this claim in Section 4.5.

Existing approaches for performing concolic testing can be understood as instances of Algorithm 2. Most of the existing approaches to concolic testing use the algorithm with a single search heuristic. For example, Burnim and Sen [3] perform concolic testing by running Run(P, {CFDS}), where CFDS is a search heuristic that exploits the control-flow information of the program. Seo and Kim [28] propose to run Run(P, {CGS}), where CGS is a search heuristic that performs the context-guided breadth-first search on the execution tree. Cha et al. [5] also use the algorithm with a single heuristic, i.e., Run(P, {Param}), where Param is a search

[1] **Chameleon**: https://github.com/kupl/Chameleon

[2] If the constraint is unsatisfiable, the algorithm uses the search heuristic again to choose another branch, which we omit in Algorithm 1 for simplicity.

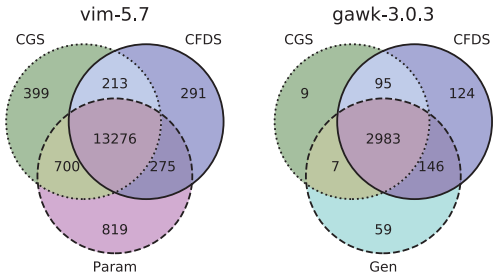

Figure 1: Venn diagrams of the number of branches covered by each search heuristic.

heuristic generated automatically by a learning algorithm. A few approaches [4, 22] use the algorithm with a number of search heuristics (e.g., Run(P, {CFDS, CGS})) so as to combine existing heuristics in a round-robin fashion.

Note that the conventional approach to concolic testing (i.e., Algorithm 2) is *non-adaptive* in that it uses the same set H of search heuristics in every iteration of the outer loop at lines 3–8. In this paper, we argue that this is a key limiting factor in existing approaches. Using a fixed set of search heuristics implies fixed branch-selection criteria, which is essentially limited to favoring specific areas of the program only. In other words, different search heuristics are largely incomparable in terms of the branch sets that they can cover during concolic testing. For example, Figure 1 shows that there is no clear winner among the top three heuristics for each program, where we ran Algorithm 2 for 24 hours per heuristic to compare the sets of branches covered by them. In this paper, we aim to mitigate this problem by *adaptively* changing search heuristics during concolic testing.

3 OUR APPROACH TO CONCOLIC TESTING

Unlike conventional concolic testing, our approach is adaptive and changes the set H of search heuristics over the course of the testing process. To achieve this, we need to define a space of possible search heuristics and to develop an algorithm that can continuously learn a new set of search heuristics from the space during the concolic testing process. The latter constitutes the key contribution of this paper (Section 3.2). For the former, we use the idea of parametric search heuristic recently proposed in prior work [5].

3.1 Parametric Search Heuristics

Our work builds on the idea of parametric search heuristics [5], which defines the space of possible search heuristics used in our approach. Cha et al. [5] defined a search heuristic, denoted $\text{Heuristic}_{\mathbf{w}}$, which has a parameter $\mathbf{w}$ as follows:

$$\text{Heuristic}_{\mathbf{w}}(\Phi) = \underset{\phi_j \in \Phi}{\text{argmax}}\ score_{\mathbf{w}}(\phi_j) \quad (\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$$

where the parameter $\mathbf{w} = \langle \theta_1, \ldots, \theta_d \rangle$ is a d-dimensional vector of real numbers. $\text{Heuristic}_{\mathbf{w}}$ takes a path-condition Φ and selects a branch ϕ_j with the highest score. To compute scores of branches, each branch ϕ is represented by a feature vector. A feature $feat_i$ denotes a predicate describing characteristics of branches:

$$feat_i : \mathbb{B} \to \{0, 1\}.$$

where $\mathbb{B}$ is the set of branches in the program. For instance, a feature may describe whether the branch ϕ is located inside a loop body or not. If true, the $feat_i(\phi)$ is 1; otherwise, it is 0. With a predefined set of d features, we are able to represent a branch by a d-dimensional boolean vector as follows:

$$feat(\phi) = \langle feat_1(\phi), feat_2(\phi), \ldots, feat_d(\phi) \rangle.$$

In this paper, we reused the 40 features (i.e., d=40) presented in [5], where these are divided into 12 static and 28 dynamic features. Using the predefined features, we transform each branch in a path-condition into a feature vector. Then, the score for each branch ϕ is calculated by a linear combination of the feature vector $feat(\phi)$ and a given d-dimensional weight vector $\mathbf{w}$:

$$score_{\mathbf{w}}(\phi) = feat(\phi) \cdot \mathbf{w}.$$

Lastly, we choose a branch ϕ_j with the highest score in Φ.

With the parametric search heuristic described above, a search heuristics corresponds to a d-dimensional weight vector. Thus, in the rest of this paper, we will call the d-dimensional real-number vectors search heuristics when there is no confusion. With this convention, we write $\mathbb{H} = \mathbb{R}^d$ for the space of possible search heuristics, where $\mathbb{R}$ denotes real numbers between −10 and 10.

Algorithm 3 Our Approach to Concolic Testing

Input: Program P
Output: The set T of covered branches
1: $\langle K, T \rangle \leftarrow \langle \emptyset, \emptyset \rangle$
2: $H \leftarrow \{(\epsilon, h_1), \ldots, (\epsilon, h_{\eta_1}) \mid h_i \sim \mathcal{U}([-10, 10]^d)\}$
3: **repeat**
4: $\quad G \leftarrow \emptyset$
5: $\quad$ **for each** $(\_, h) \in H$ **do**
6: $\quad\quad B \leftarrow \text{Concolic}(P, \text{Heuristic}_h)$
7: $\quad\quad T \leftarrow T \cup B$
8: $\quad\quad G \leftarrow G \cup \{(h, B)\}$
9: $\quad$ **end for**
10: $\quad K \leftarrow$ **if** $K = \emptyset$ **then** G **else** $\text{Refine}(K, G, H)$
11: $\quad (K_1, K_2) \leftarrow \text{Select}(K)$
12: $\quad H \leftarrow \text{Switch}(K_1, K_2)$
13: **until** timeout
14: **return** T

3.2 Overall Algorithm

Our approach reuses Algorithm 1 without modification but replaces Algorithm 2 by Algorithm 3. Unlike Algorithm 2, our algorithm does not take search heuristics as input; instead, it adaptively learns and changes them throughout the process of concolic testing. At each iteration of the outer loop (i.e., the repeat-until loop at lines 3–13), the algorithm evolves three sets: $H \subseteq \mathbb{H} \times \mathbb{H}$ is a set of search heuristics, $K \subseteq \mathbb{H} \times \wp(\mathbb{B})$ the accumulated knowledge about previous search heuristics from which we learn new heuristics, and $T \subseteq \mathbb{B}$ the set of branches covered so far. Our algorithm represents a search heuristic by a pair (h', h) in order to keep track of the birthplace information; the second component h is the actual heuristic that we are interested in the current iteration while the first component h' is the *parent* of h that gave rise to h in the previous iteration.

The algorithm begins with η_1 randomly generated heuristics (line 2) (we fixed $\eta_1 = 100$ in experiments):

$$H = \{(\epsilon, h_1), (\epsilon, h_2), \ldots, (\epsilon, h_{\eta_1})\}$$

where $h_1, \ldots, h_{\eta_1}$ are independent random samples from the uniform distribution $\mathcal{U}([-10, 10]^d)$ and ϵ indicates that the initial search heuristics do not have parents. Initially, K and T are empty (line 1). With the inner loop at lines 5–9, the algorithm performs concolic testing (i.e., Concolic(P, Heuristic$_h$)) with each heuristic in H and generates the data G as follows:

$$G = \{(h_1, B_1), \ldots, (h_{|H|}, B_{|H|})\}$$

where h_i is the current heuristic (i.e., (h', h_i)) in H and B_i is the set of branches covered by running concolic testing with h_i. At the first iteration, K becomes G at line 10 since K is initially empty. Starting with this initial knowledge and search heuristics, the algorithm keeps updating them. The knowledge is refined at line 10 using the procedure Refine, and at lines 11 and 12, a new set of search heuristics is generated from the knowledge using the procedures Select and Switch. The algorithm repeats the process above until a given time budget is exhausted. Upon termination, it returns the set T of covered branches.

Example 3.1. Suppose that we have a set H of four initial heuristics, h_1, h_2, h_3 and h_4, and running the Concolic procedure with each heuristic produces the following data:

$$G = \{(h_1, \{1, 2, 3, 4\}), (h_2, \{1, 2, 3\}), (h_3, \{5, 6\}), (h_4, \{2, 3\})\} \quad (1)$$

The set G means that the heuristic h_1 succeeds in covering branches 1, 2, 3 and 4, the heuristic h_2 covered branches 1, 2, and 3, and so on. Note that at the end of the first iteration, the knowledge K is identical to G. This way, the algorithm accumulates K that will be used in later iterations to adaptively produce new search heuristics.

In essence, our algorithm aims to continuously switch the current set H of search heuristics to a new one H', so that concolic testing with H' can exercise new branches that were not explored in previous iterations. That is, we would like to find a sequence of sets of search heuristics $H_0, H_1, H_2, \ldots$ such that

$$\bigcup_{(\_,h)\in H_0} \text{Concolic}(P, h) \cup \bigcup_{(\_,h)\in H_1} \text{Concolic}(P, h) \cup \cdots$$

is maximized within a given time budget. Algorithm 3 can be understood as a practical solution for this problem, which does so by combining the three procedures Refine, Select, and Switch described below.

3.3 Select

Let us first describe the procedure Select. The goal of Select is to select two sets, namely K_1 and K_2, of search heuristics from K:

$$\textsc{Select}(K) = (K_1, K_2).$$

Intuitively, K_1 and K_2 represent the most effective and most ineffective combinations of search heuristics in K that collectively achieve the highest and lowest coverages, respectively, where the sizes of K_1 and K_2 are fixed to η_2, a predetermined hyperparameter of our algorithm. In practice, we set η_2 to be $\lfloor |K| \times 0.03 \rfloor$, selecting 3% of K. Formally, K_1 is defined to be a set satisfying the two conditions:

(1) K_1 is a subset of K such that $|K_1| = \eta_2$, and

(2) for all $K_1' \subseteq$ s.t. $|K_1'| = \eta_2$,

$$\Big|\bigcup_{(h,B)\in K_1'} B\Big| \leq \Big|\bigcup_{(h,B)\in K_1} B\Big|.$$

Similarly, K_2 is a subset of K such that $|K_2| = \eta_2$ and $|\bigcup_{(h,B)\in K_2} B|$ is minimized. These top-η_2 and bottom-η_2 heuristics will be used for adaptively learning the distributions of the effective and ineffective search heuristics in the next step.

Example 3.2. Consider Example 3.1, where the current knowledge K is identical to the set G in (1). Then, Select(K) produces the following K_1 and K_2 when $\eta_2 = 2$:

$$K_1 = \{(h_1, \{1, 2, 3, 4\}), (h_3, \{5, 6\})\}, K_2 = \{(h_2, \{1, 2, 3\}), (h_4, \{2, 3\})\}$$

In words: h_1 and h_3 are top-2 heuristics that can cover as diverse branches as possible. On the other hand, h_2 and h_4 are bottom-2 heuristics that cover the least number of branches. The branches covered by K_1 and K_2 are $\{1, 2, 3, 4, 5, 6\}$ and $\{1, 2, 3\}$, respectively.

Finding the sets K_1 and K_2 corresponds to solving the maximum coverage problem (MCP), which is NP-hard. We use a simple greedy algorithm [15] that progressively selects set elements that collectively maximize (or minimize) the number of branches covered at each step.

3.4 Switch

Once we select K_1 and K_2, we learn new search heuristics based on the distributions of K_1 and K_2. The idea is to produce search heuristics that are statistically similar to those in K_1 but dissimilar to those in K_2. To do so, we collect the following set:

$$\bigcup_{(h,B)\in K_1} \text{Offspring}(h, K_2). \quad (2)$$

That is, we consider each heuristic $h \in K_1$ in turn, and produce its offspring as follows:

$$\text{Offspring}(h, K_2) = \{(h, h_1), \ldots, (h, h_{\eta_3})\}.$$

η_3 is a hyperparameter that determines the number of offspring of each parent $h \in K_1$ (we set $\eta_3 = 10$ in experiments). To generate h_is that are similar to h but dissimilar to those in K_2, we randomly sample each heuristic h_i, which is a d-dimensional vector of weights, from the sample space $S_1 \times S_2 \times \cdots \times S_d$, where S_j is a set of real numbers defined as follows:

$$S_j = \text{Sample}(\{h^j\}) \setminus\setminus \text{Sample}(\{h'^j \mid (h', \_) \in K_2\}) \quad (3)$$

where h^j denotes the j-th component of vector h and Sample(R) samples real numbers from the truncated normal distribution with mean $\mu(R)$, standard deviation $\sigma(R)$, and the interval $[-10, 10]$:

$$\text{Sample}(R) = \{r_1, r_2, \ldots, r_n \mid r_i \sim \mathcal{N}(\mu(R), \sigma(R), -10, 10)\}.$$

where the number (n) of samples, unless too small, does not matter and $\mu(R)$ and $\sigma(R)$ denote the median and standard deviation of the set R of real numbers:

$$\mu(R) = \sum_{r\in R} \frac{r}{|R|}, \quad \sigma(R) = \begin{cases} \sqrt{\frac{\sum_{r\in R}(r-\mu(R))^2}{|R|}} & \text{if } (|R| > 1) \\ 1 & \text{otherwise} \end{cases}$$

and $S \setminus\setminus S'$ computes the following:

$$S \setminus\setminus S' = \{\!\{\lfloor e \rfloor \mid e \in S\}\!\} \setminus \{\!\{\lfloor e \rfloor \mid e \in S'\}\!\}.$$

The notation $\{\!\{\}\!\}$ indicates that the sets are multisets allowing duplicated elements. For instance, for $S = \{2.7,\ 3.1,\ 3.4,\ 5.2\}$ and $S' = \{1.6,\ 2.4,\ 3.3,\ 4.9\}$, $S \setminus\!\setminus S' = \{\!\{2, 3, 3, 5\}\!\} \setminus \{\!\{1, 2, 3, 4\}\!\} = \{\!\{3, 5\}\!\}$.

Note that, when we construct the sample space in (3), we generate distributions by considering *all* weights of the j-th feature vector h' in K_2 (i.e., $\mathsf{Sample}(\{h'^j \mid (h', \_) \in K_2\})$) whereas we treat heuristics in K_1 separately. The intuition is to maintain the relationships between the features that each top heuristic in K_1 *may* have, while maintaining the relationships between the features that all bottom ones in K_2 *must* have. We found that this is an important choice for our algorithm to fully exploit the current knowledge; it enables the algorithm to produce new heuristics that resemble good ones while effectively avoiding bad ones.

With the set collected in (2), the procedure $\textsc{Switch}(K_1, K_2)$ is defined as follows:

$$\textsc{Switch}(K_1, K_2) = \mathsf{Exploit} \cup \mathsf{Explore}$$

where Exploit is the set that contains $\eta_1 \times \eta_4$ heuristics selected from the set in (2) and Explore is the set of $\eta_1 \times (1 - \eta_4)$ randomly generated heuristics to enable exploration:

$$\mathsf{Explore} = \{h_1, \ldots, h_{\eta_1 \times (1-\eta_4)} \mid h_i \sim \mathcal{U}([-10, 10]^d)\}$$

where η_4 is the hyperparameter that controls the tradeoff between exploitation and exploration. We set η_4 to 0.8 in experiments.

Feature Selection. To reduce the space of candidate search heuristics, we can optimize the procedure Switch via feature selection. When we construct the sample space S_j for the j-th weights in (3), we simply define $S_j = \{0\}$ if the j-th feature is uninformative. We consider the i-th feature is uninformative if the weights of that feature in K_1 are statistically similar to those in K_2. To calculate the similarity, we first define the two sets, G_i and B_i, as follows:

$$G_i = \{\theta^i \mid (\langle \theta^1, \theta^2, \cdots, \theta^d \rangle, \_) \in K_1\}$$
$$B_i = \{\theta^i \mid (\langle \theta^1, \theta^2, \cdots, \theta^d \rangle, \_) \in K_2\}$$

where G_i and B_i are sets consisting of the i-th components of the weight vectors in K_1 and K_2, respectively. Second, we collect the features whose weights are similar in K_1 and K_2:

$$F = \{i \in [1, d] \mid \mathsf{similar}(G_i, B_i)\}$$

where $\mathsf{similar}(G_i, B_i)$ is true when the distributions of G_i and B_i are similar in the following sense:

$$\mathsf{similar}(G_i, B_i) \iff |\mu(G_i) - \mu(B_i)| + |\sigma(G_i) - \sigma(B_i)| < 1.$$

Once we compute the set F of uninformative features, we define $S_j = \{0\}$ if $j \in F$.

3.5 Refine

The role of Refine refines the current knowledge K to make learning more effective. The procedure Refine takes three sets: K, G, and H, where K is the knowledge from the previous iteration, G is the newly generated knowledge from the current iteration, and H is the current set of search heuristics. Given (K, G, H), $\textsc{Refine}(K, G, H)$ produces the refined knowledge K' as follows:

$$K' = (K \cup G) \setminus \mathit{Kill}$$

It first augments the previous knowledge K with the new one G and then removes the set *Kill* from the result. Intuitively, *Kill* denotes the parent heuristics that are turned out to be no longer useful at the current iteration of the algorithm; *Kill* is the set of parents whose offspring totally failed to cover new branches. We remove those heuristics in K in order not to exploit them in vain again in later iterations, which makes the overall learning process smarter. Formally, *Kill* is defined as follows:

$$\mathit{Kill} = \{(h', B') \in K \mid (h', \_) \in H, \bigcup_{(h',h)\in H,(h,B)\in G} B \subseteq \bigcup_{(h,B)\in K} B\}.$$

In words: (h', B') in K is removed if h' is a parent of some current search heuristics in H, i.e., $(h', \_) \in H$, and the offspring of h' fail to exercise new branches over the current knowledge, i.e., $\bigcup_{(h',h)\in H,(h,B)\in G} B \subseteq \bigcup_{(h,B)\in K} B$.

Example 3.3. Consider the second iteration of Algorithm 3 and the set in (1) is the previous knowledge:

$$K = \{(h_1, \{1, 2, 3, 4\}), (h_2, \{1, 2, 3\}), (h_3, \{5, 6\}), (h_4, \{2, 3\})\}$$

and the current H (with $\eta_3 = 2$) is $\{(h_1, h_5), (h_1, h_6), (h_3, h_7), (h_3, h_8)\}$ with the following profiles:

$$G = \{(h_5, \{1, 3, 4, 6\}), (h_6, \{1, 2, 3, 4, 5, 6\}), (h_7, \{5, 7\}), (h_8, \{3, 8\})\}$$

Then, the set *Kill* is as follows:

$$\mathit{Kill} = \{(h_1, \{1, 2, 3, 4\})\}$$

because the offspring of h_1 are h_5 and h_6, and the set $\{1, 2, 3, 4, 5, 6\}$ of branches covered by h_1 and h_5 according to G is subsumed by the set of branches contained in the previous knowledge K. The refined knowledge is:

$$K' = \{(h_2, \{1, 2, 3\}), (h_3, \{5, 6\}), (h_4, \{2, 3\}), (h_5, \{1, 3, 4, 6\}),$$
$$(h_6, \{1, 2, 3, 4, 5, 6\}), (h_7, \{5, 7\}), (h_8, \{3, 8\})\}.$$

Note that h_1 is removed from K, so it will not be selected for exploitation in the future iterations of Algorithm 3.

Hyperparameters. Our algorithm involves four hyperparameters (η_1, η_2, η_3, and η_4) for which appropriate values are assumed to be given beforehand. The first hyperparameter η_1 determines the pool size of search heuristics. η_2 in the Select procedure denotes the number of effective (and ineffective) search heuristics to be selected from the knowledge K. The remaining two hyperparameters are required in the Switch procedure; η_3 determines the number of offspring to be generated from each effective heuristic and the last one η_4 is the exploitation rate. In experiments, we set $\eta_1 = 100$, $\eta_2 = \lfloor |K| \times 0.03 \rfloor$, $\eta_3 = 10$, and $\eta_4 = 0.8$. Basically, we decided these hyperparameters by trial and error but found that most of them require no fine tuning. An exception was η_4 , for which choosing a right value was important for the performance. In Section 4.5, we discuss how the performance changes with different values of η_4.

4 EXPERIMENTS

In this section, we evaluate the effectiveness of our approach. We implemented our approach in a tool, called Chameleon, on top of CREST [8] and ParaDySE [5]. CREST is an open-source framework for concolic testing of C programs widely used in prior work (e.g., [3, 5, 6, 9, 22, 24, 28]). ParaDySE provides a publicly available implementation[3] of the parametric search heuristic in Section 3.1. We evaluate Chameleon from the three perspectives:

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

Table 1: 8 benchmark programs

Program	#Branches	LOC	Source
vim-5.7	35,464	165K	[3, 5, 6, 25]
gawk-3.0.3	8,038	30K	[5, 6]
grep-2.2	3,836	15K	[3, 5, 6, 28]
sed-1.17	2,650	9K	[5, 6, 21]
cdaudio	358	3K	[5, 17, 28]
floppy	268	2K	[5, 17, 28]
kbfiltr	204	1K	[5, 17, 28]
replace	196	0.5K	[3, 5, 21, 28]

(1) **Branch coverage:** How effectively does Chameleon increase branch coverage? How does it compare to conventional concolic testing with existing non-adaptive search heuristics? (Section 4.2)
(2) **Bug-finding:** How effectively does Chameleon find bugs? Does it find more bugs than conventional concolic testing? Does it find nontrivial bugs that are hard to fix? (Section 4.3)
(3) **Efficacy of learning algorithm:** Is our learning algorithm (Section 3) essential for achieving the desired results? How effective is it compared to simpler techniques? (Section 4.4)

4.1 Experimental Setup

Benchmarks. We evaluated Chameleon on 8 open-source programs in Table 1. We used these benchmarks because they were commonly used in previous works on concolic testing [3, 5, 6, 17, 21, 25, 28]. These benchmarks are divided into 4 large and 4 small programs. The former consists of `vim`, `gawk`, `grep`, and `sed`, which have at least 2,000 branches; the latter includes `cdaudio`, `floppy`, `kbfiltr`, and `replace`. We did not use expat-2.10, which is used in [5, 28], because we found it is less suitable for concolic testing without prior knowledge about the input format (XML).

Existing Search Heuristics. We compared Chameleon with six recent or well-known search heuristics: `Param` (Parametric Search) [5], `CGS` (Context-Guided Search) [28], `CFDS` (Control-Flow Directed Search) [3], `Gen` (Generational search) [12], and `Random` (Random branch search) [3], and `RoundRobin` (`RR`). `RoundRobin` is a combination of the first five heuristics, which uses them in a round-robin fashion. `CFDS` and `Random` are available in CREST, and `Param`, `Gen` and `CGS` are available in ParaDySE. We did not consider naive heuristics such as `DFS` and `BFS`, because their performance is not competitive as shown in the prior works [3, 5, 28].

Time Budget. We allocated 24 hours as a testing budget to the four large programs while allocating one hour to the four small ones. For the large programs, we gave enough time budget (i.e., 24 hours) to compare the performance of Chameleon and existing search heuristics in a truly practical setting. By contrast, we observed that the time budgets commonly used in previous works are not very realistic. For example, previous works on search heuristics [3, 5, 28] conducted experiments with small time budgets needed to execute each program 4,000 times, which corresponds to 1–30 minutes for the benchmark programs in Table 1 in our environment. According to our experience, these budgets are too small to appropriately evaluate the practical performance of concolic testing, especially for large programs such as `vim`.

Table 2: Average branch coverage achieved by Chameleon and 6 search heuristics on 4 small benchmarks

	Chameleon	RR	CFDS	CGS	Param	Gen	Random
cdaudio	250	250	250	250	250	250	250
floppy	205	205	205	205	205	205	196
replace	181	181	181	181	181	181	181
kbfiltr	149	149	149	149	149	149	149

Table 3: The number of branches exclusively covered by each technique on 4 large benchmarks

	Chameleon	RR	CFDS	CGS	Param	Gen	Random
vim-5.7	**364**	37	62	163	272	50	82
gawk-3.0.3	**136**	4	3	4	0	1	4
grep-2.2	**55**	0	4	3	0	0	0
sed-1.17	**43**	0	9	3	7	3	0
Total	**598**	41	78	173	279	54	86

Others. All experiments were conducted under the same settings. First, we used the same initial inputs provided together with each benchmark program. Second, we conducted all experiments on the same machine with two Intel Xeon Processors E5-2630 and 192GB RAM. Third, we performed concolic testing on a single core for all benchmarks except for `vim`. This is because we found that the branch coverage did not converge within 24 hours for `vim`. We accelerated the convergence by running concolic testing for `vim` using 10 cores in parallel, which means a total of 240 hours are in fact spent for testing `vim`.[4] Forth, we set N in Algorithm 1 to 4,000. Finally, to calculate the average performance of the six existing heuristics and Chameleon, we repeated all the experiments 3 times and averaged the results.

4.2 Branch Coverage

Let us first compare Chameleon and conventional concolic testing in terms of branch coverage. We use two metrics, average branch coverage and exclusively covered branches. In both cases, Chameleon performs much better than existing approaches.

Average Branch Coverage. Figure 2 compares average branch coverage achieved by Chameleon and conventional approaches on four large benchmarks. The results show that Chameleon impressively outperforms the existing approaches in all cases. In particular, the results for the two largest programs (`vim` and `gawk`) are noteworthy: Chameleon was able to reach 15,468 branches covering 399 more branches than `Param`, a state-of-the-art that already covers 283 more branches over `RoundRobin`. For `gawk`, Chameleon covered 3,564 branches while the second best heuristic (`RoundRobin`) managed to exercise 3,350 branches within the same time budget. For `grep` and `sed`, Chameleon was the clear winner as well, covering 2,271 and 1,696 branches, respectively. For the small benchmarks,

[4] Algorithms 2 and 3 are easily parallelizable without dependency between parallel tasks.

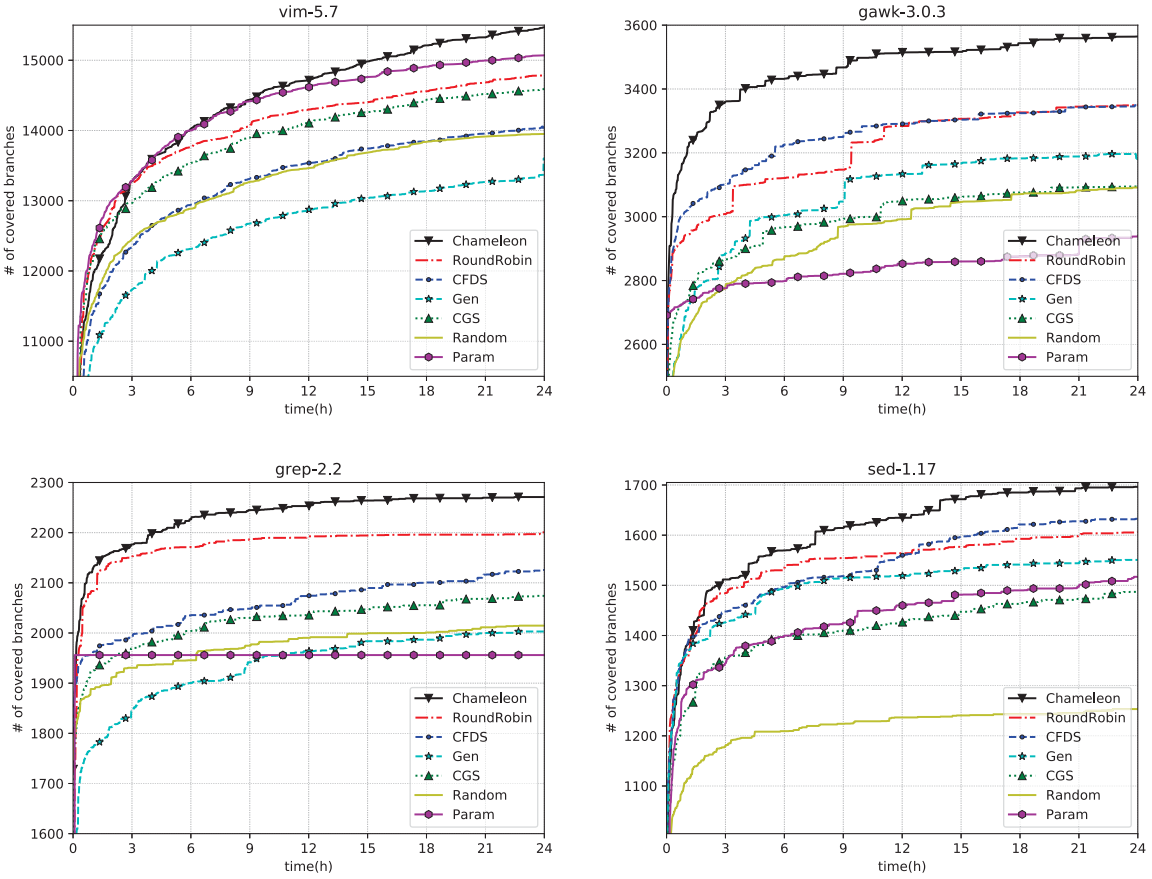

Figure 2: Average branch coverage achieved by CHAMELEON and 6 search heuristics on 4 large benchmarks

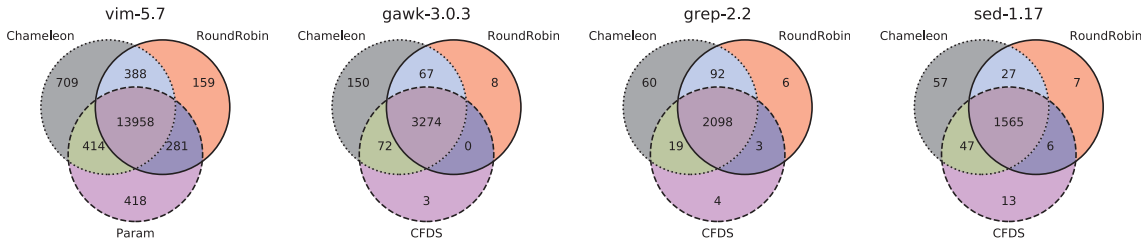

Figure 3: Venn-diagrams depicting the sets of branches covered by the top-3 heuristics for each large benchmark

CHAMELEON and others, except for Random, achieved exactly the same branch coverage within the 1 hour time budget (Table 2).

Exclusively Covered Branches. We also compared CHAMELEON and the existing approaches in terms of the set of covered branches. Table 3 reports the number of branches that each technique *exclusively* covered over the other 6 techniques. In this metric as well, CHAMELEON is much better than the existing search heuristics. In total, 598 branches were covered exclusively by CHAMELEON. In particular, note that, for all benchmarks except for `vim`, the number of unique branches covered by CHAMELEON alone is greater than the number of unique branches covered by *all* the other techniques, which implies that CHAMELEON is better than *any* combinations of the six existing heuristics. For example, for `gawk`, the former is 136 while the latter is 16. Similarly, for `grep`, the number of unique branches covered by CHAMELEON is about 8 times more than the number of branches that all the other techniques exclusively can cover. For `vim`, CHAMELEON is still the best but it is not enough to say it is a clear winner. This is because the size of `vim` is so large that all the techniques, including CHAMELEON, have not converged yet even though we performed concolic testing for 24 hours using

Table 4: Comparison of bug-finding ability of ours (CHAMELEON) and existing approaches on 4 large benchmarks.

Benchmarks	Versions	Error Types	Bug-Triggering Inputs	OURS	Param	RR	CGS	CFDS	Gen	Random
vim	8.1*	Non-termination	`K1!1000100100111110(`	✓	✗	✗	✗	✗	✗	✗
	5.7	Abnormal-termination	`H:w>>`"`\` [press 'Enter']	✓	✓	✗	✗	✗	✓	✓
		Segmentation fault	`=ipI\-9~qOqw`	✓	✓	✓	✓	✗	✗	✓
		Non-termination	`v(ipaprq&T$T`	✓	✓	✓	✗	✗	✗	✓
gawk	4.2.1*	Memory-exhaustion	`'+E_Q$h+w$8==++$6E8#'`	✓	✗	✗	✗	✗	✗	✗
	3.0.3	Abnormal-termination	`'f[][][][][y]^/#['`	✓	✗	✓	✓	✓	✓	✓
		Non-termination	`'$g?E2^=-E-2"?^+$=":/?/#["'`	✓	✓	✗	✗	✓	✗	✗
grep	3.1*	Abnormal-termination	`'\(\)\1*?*?|\W*\1W*'`	✓	✗	✗	✗	✗	✗	✗
		Segmentation fault	`'\(\)\1^*@*\?\1*\+*\?'`	✓	✗	✗	✓	✗	✗	✗
	2.2	Segmentation fault	`"_^^*9|^\(\)\'\1*$"`	✓	✓	✓	✓	✓	✓	✓
		Non-termination	`'\({*****\)*\+**\1*\+'`	✓	✓	✓	✓	✓	✓	✗
sed	1.17	Segmentation fault	`'{:};:C;b'`	✓	✗	✓	✗	✓	✓	✓

10 cores in parallel. Figure 3 shows the Venn-diagrams that depict the relationships between the branches covered by each technique, where we only consider top-3 techniques for each benchmark.

4.3 Bug-Finding

Now we compare CHAMELEON and conventional concolic testing in terms of bug-finding. In short, CHAMELEON is highly effective in finding real-bugs; for the latest versions of `vim`, `gawk`, and `grep`, CHAMELEON succeeded to generate bug-triggering inputs while all the other techniques failed to do so.

Setup. While conducting the experiments in Section 4.2, we monitored program execution and collected bug-triggering inputs generated by CHAMELEON and other six techniques. Specifically, we considered two types of bugs: program crashes and performance bugs. First, to collect crashing inputs, we monitored the system signals (e.g., SIGSEGV) after executing the program with each input that CHAMELEON and other techniques generated. Second, we collected the performance bugs by checking if the program execution with each input would exhaust a time or memory bound. After collecting the bug-triggering inputs for each technique, we filtered the genuine bugs that are reproducible on the original binary of each benchmark program without annotations for concolic testing and excluded irreproducible ones. Finally, we further classify the collected bugs into 4 categories: segmentation fault (SIGSEGV), abnormal-termination (SIGABRT), non-termination, and memory-exhaustion.

Results. Table 4 shows the results on two versions of each benchmark program: the original version used in Section 4.2 (on which we found bugs) and the latest version at the time of writing. For each benchmark, the table shows the program version (Versions), the error type (Error Types), one of the bug-triggering inputs generated by CHAMELEON (Bug-Triggering Inputs), and the success (✓) and failure (✗) results for each technique. The success mark (✓) for a technique indicates that the technique succeeded to generate at least one input that causes the corresponding error type, whereas the failure mark (✗) means the technique totally failed to trigger the error type.

The results show that CHAMELEON outperforms the existing techniques in terms of bug-finding. In particular, CHAMELEON was unique in finding bugs that can be triggered in the latest versions of `vim`, `gawk`, and `grep`. Furthermore, CHAMELEON was able to find various types of errors, including non-termination (`vim-8.1`), memory-exhaustion (`gawk-4.21`), and abnormal termination (`grep-3.1`). In total, CHAMELEON could trigger 12 different types of errors across all programs and their versions. On the other hand, the other techniques managed to trigger 6 types of errors at best. The performance of existing techniques varied depending on the benchmark while CHAMELEON consistently performed well on 4 large benchmarks.

We found that CHAMELEON is effective in finding hard-to-find bugs. For example, the input `'\(\)\1*?*?\|\W*\1W*'` generated by CHAMELEON causes a segmentation fault in `grep-3.1`. Surprisingly, this bug survived over the last 20 years from `grep-2.2` (1998) to `grep-3.1` (2018). CHAMELEON also found deadly bugs. For example, on `gawk-4.21`, the input `'+E_Q$h+w$8==++$6E8#'` found by CHAMELEON causes a serious performance bug that may consume all the memory of the machine. All the bug-triggering inputs in Table 4 are easily reproducible. For instance, on `grep-3.1`, the command `./grep '\(\)\1*?*?\|\W*\1W*' file` (where `file` is an arbitrary file) immediately aborts the program execution.

Figure 4 also adds to evidence that CHAMELEON is good at finding difficult bugs. The figures show how many bug-triggering inputs found by each technique in the initial programs survive as programs evolve, where the hypothesis is that difficult bugs would survive longer than shallow bugs. In the case of `grep`, CHAMELEON consistently achieves the highest number of reproducible bug-triggering inputs over the subsequent program versions. Meanwhile, all bugs found by other techniques, except for `CGS`, did not survive after `grep-2.4`, and only a single bug-triggering input found by `CGS` remains in `grep-2.6`. For `gawk-3.0.3` (the initial version), note

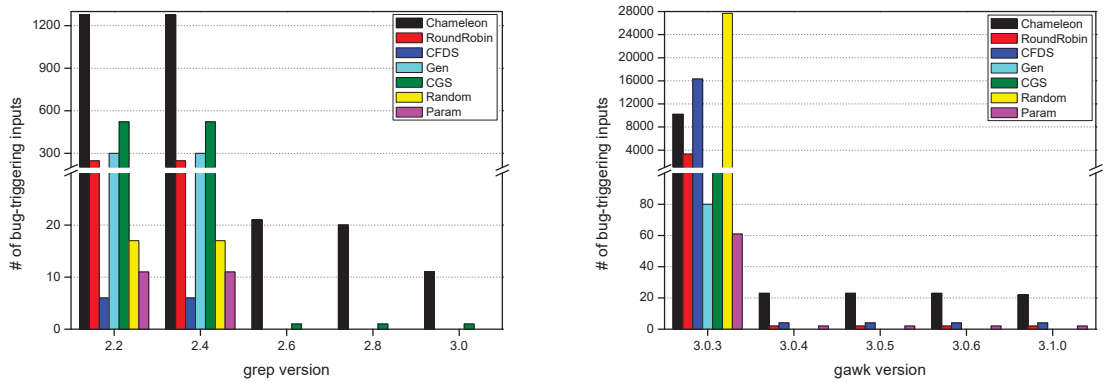

Figure 4: Comparison of the number of bug-triggering inputs that survive over program evolution

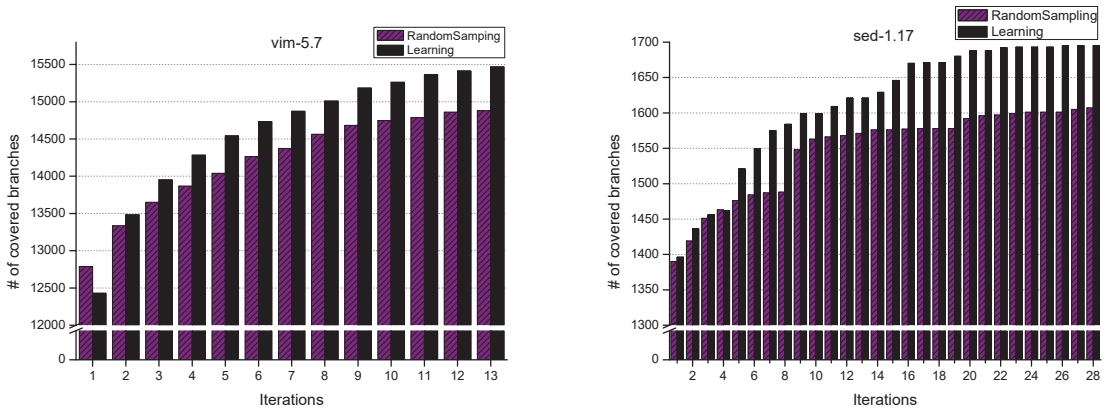

Figure 5: Comparison between random-sampling and our learning algorithm

that Chameleon is not the winner as `Random` and `CFDS` find more bug-triggering inputs. However, as the program evolves, the situation is completely reversed; all of the 28,000 bug-triggering inputs generated by `Random` in the original version failed to survive in the next version (`gawk-3.0.4`). That is, `Random` is likely to find bugs that are comparatively easy to fix. On the other hand, 22 inputs discovered by Chameleon are reproducible until the version 3.1.0 without being fixed for more than 4 years.

4.4 Efficacy of Learning Algorithm

We evaluated the efficacy of our algorithm by comparing it with a much simpler algorithm that randomly changes search heuristics. The naive algorithm can be easily implemented by sampling the set H randomly before line 5 of Algorithm 3 and ignoring the lines 10–12 for Refine, Select, and Switch. For each iteration of the outer loop of Algorithm 3, we compared the cumulative branch coverage achieved by our algorithm and the naive algorithm for `vim-5.7` and `sed-1.17`.

Figure 5 shows that our learning algorithm for *adaptively* changing search heuristics is essential. For `vim-5.7`, when the testing budget (24h) is exhausted, our algorithm is able to cover 15,468 branches, covering 588 more branches than the random sampling method. In the first iteration where both algorithms relied on random sampling, our algorithm unfortunately started with initial search heuristics with lower quality compared to the naive algorithm. However, in the next iteration, our algorithm immediately succeeded in switching search heuristics that can cover more branches than the naive one. As the iteration of both algorithms goes on, the difference in branch coverage achieved by each algorithm becomes larger as follows: $I_2(146)$, $I_3(300)$, $I_4(417)$, $\cdots$, $I_{13}(588)$. For `sed-1.17`, we obtained the similar conclusion; until the fourth iteration at which the knowledge (K) was not accumulated sufficiently, our algorithm had similar performance compared to the random sampling method. However, ours covered around 1,700 branches in the end, where it learns to increase the branch coverage over the random method by around 100.

Table 5: Coverage variation by exploitation rate (sed-1.17)

exploitation rate	0%	20%	40%	60%	**80%**	100%
# branches	1,612	1,611	1,638	1,679	**1,696**	1,672

Table 6: Average branch coverage achieved by each heuristic on Algorithm 1 and 2 (24h). We set N to ∞ and 4,000 for Algorithm 1 (A_1) and Algorithm 2 (A_2), respectively.

	gawk-3.0.3		grep-2.2		sed-1.17	
	A_1	A_2	A_1	A_2	A_1	A_2
CFDS [3]	3,350	3,349	2,132	2,125	1,548	1,632
CGS [28]	2,767	3,095	1,922	2,074	1,208	1,487
Random [3]	3,113	3,091	1,924	2,014	1,481	1,253
Gen [12]	2,336	3,184	1,797	2,003	1,106	1,550
Param [5]	2,828	2,939	2,014	1,956	1,031	1,517
Total	14,394	**15,658**	9,789	**10,172**	6,374	**7,439**

4.5 Discussions

Exploration and Exploitation. In our algorithm (Section 3), the hyperparameter η_4 for balancing exploration and exploitation was crucial for obtaining the desired results. For example, Table 5 shows that Chameleon achieves the highest branch coverage on `sed-1.17` when the exploitation rate is around 80%. We obtained similar results for other programs and set η_4 to 0.8. In experiments, we found hyperparameters by trial and error. To be systematic, it would be possible to use algorithms for tuning hyperparameters automatically from the machine-learning community (e.g., [2]).

Algorithm 1 vs Algorithm 2. In practice, within the same time budget, performing concolic testing with a small budget multiple times (i.e., Algorithm 2) is more effective than performing Algorithm 1 alone with large N until timeout. Table 6 shows that using Algorithm 1 with $N = \infty$ is far inferior to using Algorithm 2 with small N (4,000) on 3 large benchmarks. For instance, for `gawk`, running Algorithm 2 covered 15,658 branches in total, while running Algorithm 1 covered 14,394 branches only.

Threats to Validity. First, our evaluation used 8 benchmark programs that have been commonly used in prior works [3, 5, 6, 17, 21, 25, 28]. However, these programs may not be sufficient to draw a firm conclusion in general. Second, to run Chameleon, we manually tuned the hyper-parameters that work well on our benchmarks. However, these values may not suit arbitrary programs.

5 RELATED WORK

In this section, we discuss two lines of researches that are most related to ours: techniques for employing search heuristics [3–5, 12, 22, 28] and combining learning and software testing [6, 7, 13, 16, 18, 23, 29, 31]. The former aims to mitigate the path-explosion problem of concolic testing by presenting search heuristics. The latter aims to solve various problems of software testing with learning.

Search Heuristics. All previous works on search heuristics [3–5, 12, 22, 28] have focused on coming up with a new branch selection strategy. However, in this paper, we claim that any single search heuristics or their limited combinations are not sufficient. The selection criterion of `CFDS` [3] is to randomly pick one of the branches that are closest to uncovered branches in the current execution path. The `CGS` [28] heuristic is to randomly select one of the branches at the same depth of execution tree by `BFS` heuristic, while excluding branches with already explored "context". The strategy of `Param` [5] is to select the branch with the highest score in the current path, where each branch score is calculated as a linear combination of the branch feature vector and a given weight vector. To do so, the technique works in two steps: offline and online phases. In the offline phase, a learning algorithm is run to produce a search heuristic that is optimal for a subject program. Then, the learned heuristic (`Param`) is used for testing the subject program (the online phase). Note that the `Param` heuristic does not change during the online phase and therefore we call it non-adaptive. In contrast, our work focuses on adapting search heuristics during concolic testing (i.e., Chameleon can be used without the offline learning phase).

Combining Testing and Learning. At a high-level, our work belongs to the techniques that combine software testing and learning [6, 7, 13, 16, 18, 23, 29, 31], which leverage machine-learning technologies to solve various problems of software testing. ConTest [6] aims to reduce the search space of concolic testing by online learning, where the goal is to selectively generate symbolic variables. In Continuous Integration (CI), RECTECS [29] first uses a reinforcement learning to effectively select and prioritize failing test cases. In Android GUI testing, QBE [23] also employs a reinforcement learning algorithm (Q-learning) to learn the GUI actions that are likely to increase activity coverage, enabling crash detection. In fuzzing, Learn&Fuzz [13] aims to learn the structure of PDF objects to increase the effectiveness of input fuzzing by using neural-network-based learning techniques. Similarly, for fuzzing, Skyfire [31] aims to generate well-distributed seed inputs, thereby achieving the highest code coverage. To do so, it learns a probabilistic context-sensitive grammar from large amount of existing samples. Unlike the previous works, our work employs a learning algorithm to adaptively change search heuristics online in concolic testing.

6 CONCLUSION

Designing effective ways of employing search heuristic is an ongoing challenge in concolic testing. In this paper, we presented Chameleon to adaptively learn and change search heuristics during concolic testing. Experiments with open-source programs show that Chameleon outperforms a number of state-of-the-art, yet non-adaptive, approaches in both code coverage and bug detection. Our results suggest that, unlike existing approaches that rely on specific heuristics, search heuristics should be changed adaptively during concolic testing.

ACKNOWLEDGMENTS

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. This work was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068175).

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] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-parameter Optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*. 2546–2554.

[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] 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.

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

[7] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. 623–640.

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

[9] 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.

[10] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security '13)*. 463–478.

[11] 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.

[12] 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.

[13] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. 50–59.

[14] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R.B. Butler. 2017. FirmUSB: Vetting USB Device Firmware Using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 2245–2262.

[15] Dorit S. Hochbaum (Ed.). 1997. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., Boston, MA, USA.

[16] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. 269–282.

[17] 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.

[18] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. 2015. Learning Combinatorial Interaction Test Generation Strategies Using Hyperheuristic Search. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. 540–550.

[19] 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.

[20] Yunho Kim, Yunja Choi, and Moonzoo Kim. 2018. Precise Concolic Unit Testing of C Programs Using Extended Units and Symbolic Alarm Filtering. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 315–326.

[21] 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.

[22] 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.

[23] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST '18)*. 105–115.

[24] Hongbo Li, Sihuan Li, Zachary Benavides, Zizhong Chen, and Rajiv Gupta. 2018. COMPI: Concolic Testing for MPI Applications. *2018 IEEE International Parallel and Distributed Processing Symposium* (2018), 865–874.

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

[26] 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.

[27] 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.

[28] 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.

[29] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. 12–22.

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

[31] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (S&P '17)*. 579–594.

[32] 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.

[33] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security '18)*. 745–761.