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

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.
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\footnote{Chameleon: https://github.com/kupl/Chameleon} 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 $v_0$ 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 $v$ (i.e., $\text{Execute}(P, v)$), which produces the path condition $\Phi = \phi_1 \land \cdots \land \phi_n$, 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 $a$, the path condition $\Phi$ is $\phi_1 \land \phi_2$, where $\phi_1 = (a > 1)$ and $\phi_2 = (a > 10)$. At line 6, the algorithm accumulates the covered branches in the set $B$ (where we write $\text{Branches}(\Phi)$ for the branch ids covered by the current execution path). At line 7, the algorithm uses the search heuristic (Heuristic) to choose a branch $\phi_i$ to be negated in the next iteration. Then, at line 8, we generate a new input vector $v$ by finding a model of the constraint $\land_{j<i} \phi_j \land \neg \phi_i$ via an SMT solver. The algorithm repeats the process described so far, and returns the set $B$ upon termination.

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 $\text{Run}(P, \{\text{CFDS}\})$, where CFDS is a search heuristic that exploits the control-flow information of the program. Seo and Kim [28] propose to run $\text{Run}(P, \{\text{CGS}\})$, 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., $\text{Run}(P, \{\text{Param}\})$, where Param is a search

\begin{algorithm}
\caption{Basic Concolic Testing Procedure}
\begin{algorithmic}
  \State Input: A program $(P)$ under test and a search heuristic (Heuristic)
  \State Output: The set $(B)$ of covered branches
  \Procedure{Concolic}{$P$, Heuristic} \Comment{initial input $v_0$}
  \State $v \leftarrow v_0$
  \State $B \leftarrow \emptyset$
  \For{$m = 1$ to $N$}
    \State $\Phi \leftarrow \text{Execute}(P, v)$ \Comment{$\Phi = (\phi_1 \land \cdots \land \phi_n)$}
    \State $B \leftarrow B \cup \text{Branches}(\Phi)$
    \State $\phi_i \leftarrow \text{Heuristic}(\Phi)$ \Comment{choose a branch}
    \State $v \leftarrow \text{model}(\land_{j<i} \phi_j \land \neg \phi_i)$
  \EndFor
  \State return $B$
  \EndProcedure
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{Conventional Method for Running Concolic Testing}
\begin{algorithmic}
  \State Input: Program $P$, A set $H$ of search heuristics
  \State Output: The set $T$ of covered branches
  \Procedure{Run}{$P$, $H$} \Comment{initial input $v_0$}
  \State $T \leftarrow \emptyset$
  \Repeat
    \For{each $h \in H$}
      \State $B \leftarrow \text{Concolic}(P, h)$
      \State $T \leftarrow T \cup B$
    \EndFor
    \Until timeout
  \EndRepeat
  \State return $T$
  \EndProcedure
\end{algorithmic}
\end{algorithm}
heuristically 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 \( \text{Heuristic}_w \), which has a parameter \( w \) as follows:

\[
\text{Heuristic}_w(\Phi) = \arg\max_{\phi_j \in \Phi} \text{score}_w(\phi_j) \quad (\Phi = \phi_1 \land \cdots \land \phi_n)
\]

where the parameter \( w = (\theta_1, \ldots, \theta_d) \) is a \( d \)-dimensional vector of real numbers. \( \text{Heuristic}_w \) takes a path-condition \( \Phi \) and selects a branch \( \phi_j \) with the highest score. To compute scores of branches, each branch \( \phi \) is represented by a feature vector. A feature \( \text{feat}_i \) denotes a predicate describing characteristics of branches:

\[
\text{feat}_i : \mathbb{B} \rightarrow \{0, 1\}.
\]

where \( \mathbb{B} \) is the set of branches in the program. For instance, a feature may describe whether the branch \( \phi \) is located inside a loop body or not. If true, the \( \text{feat}_i(\phi) \) 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:

\[
\text{feat}(\phi) = (\text{feat}_1(\phi), \text{feat}_2(\phi), \ldots, \text{feat}_d(\phi)).
\]

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 \( \phi \) is calculated by a linear combination of the feature vector \( \text{feat}(\phi) \) and a given \( d \)-dimensional weight vector \( w \):

\[
\text{score}_w(\phi) = \text{feat}(\phi) \cdot w.
\]

Lastly, we choose a branch \( \phi_j \) with the highest score in \( \Phi \).

With the parametric search heuristic described above, a search heuristic 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 \( H = \mathbb{R}^d \) for the space of possible search heuristics, where \( \mathbb{R} \) denotes real numbers between \(-10 \) and \( 10 \).

#### 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: \( H \subseteq \mathbb{R} \times \mathbb{H} \) is a set of search heuristics, \( K \subseteq \mathbb{H} \times \mathbb{P}(\mathbb{B}) \) 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 η1 randomly generated heuristics (line 2) (we fixed η1 = 100 in experiments):

\[ H = \{(e, h_1), (e, h_2), \ldots, (e, h_\eta_1)\} \]

where \( h_1, \ldots, h_\eta_1 \) are independent random samples from the uniform distribution \( U([-10, 10]^d) \) and \( \epsilon \) 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., \( \text{Concolic}(P, \text{Heuristic}_h) \)) with each heuristic in \( H \) and generates the data \( G \) as follows:

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

where \( h_1 \) is the current heuristic (i.e., \( (h', h_1) \)) 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 \( \text{Refine} \), and at lines 11 and 12, a new set of search heuristics is generated from the knowledge using the procedures \( \text{Select} \) and \( \text{Switch} \). 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:

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

\[ \bigcup_{(h, K) \in H_0} \text{Concolic}(P, h) \cup \bigcup_{(h, K) \in H_1} \text{Concolic}(P, h) \cup \cdots \]

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 \( \text{Refine} \), \( \text{Select} \), and \( \text{Switch} \) described below.

### 3.3 Select

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

\[ \text{Select}(K) = (K_1, K_2). \]

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 \( \eta_2 \), a predetermined hyperparameter of our algorithm. In practice, we set \( \eta_2 \) to be \([|K| \times 0.03]\), 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 \( K' \subseteq K \) s.t. \(|K'_1| = \eta_2\),
   \[ \left| \bigcup_{(h, K) \in K_1'} B \right| \leq \left| \bigcup_{(h, B) \in K_1} B \right|. \]

Similarly, \( K_2 \) is a subset of \( K \) such that \(|K_2| = \eta_2\) and \( \bigcup_{(h, K) \in K_2} B \) is minimized. These top-\( \eta_2 \) and bottom-\( \eta_2 \) 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, \( \text{Select}(K) \) produces the following \( K_1 \) and \( K_2 \) when \( \eta_2 = 2 \):

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

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:

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

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

\[ \text{Offspring}(h, K_2) = \{(h, h_1), \ldots, (h, h_{\eta_1})\}. \]

\( \eta_1 \) is a hyperparameter that determines the number of offspring of each parent \( h \in K_1 \) (we set \( \eta_1 = 10 \) in experiments). To generate \( h_1 \)s that are similar to \( h \) but dissimilar to those in \( K_2 \), we randomly sample each heuristic \( h_1 \), 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:

\[ S_j = \text{Sample}(h^j) \setminus \text{Sample}(\{h'^j \mid (h', \_ \in K_2\}). \]

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

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

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:

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

and \( S \cap S' \) computes the following:

\[ S \cap S' = \{\{e \mid e \in S\} \setminus \{e \mid e \in S'\} \}. \]
The notation $\| \|$ indicates that the sets are multisets allowing duplicated elements. For instance, for $S = \{2.7, 3.1, 3.4, 5.2\}$ and $S' = \{1.6, 2.4, 3.3, 4.9\}$, $S \setminus S' = \{2.3, 3.5\} \setminus \{1.2, 3.4\} = \{3.5\}$.

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., $\text{Sample}(\{h'[j] \mid (h'_j, \ldots) \in K_2\})$) 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 $\text{Switch}(K_1, K_2)$ is defined as follows:

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

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

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

where $\eta_4$ is the hyperparameter that controls the tradeoff between exploitation and exploration. We set $\eta_4$ to 0.8 in experiments.

**Feature Selection.** To reduce the space of candidate search heuristics, we can optimize the procedure $\text{Switch}$ 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:

$$G_i = \{\theta_i \mid \langle \theta_i, \theta_2, \ldots, \theta_d \rangle \in K_1\}$$

$$B_i = \{\theta_i \mid \langle \theta_i, \theta_2, \ldots, \theta_d \rangle \in K_2\}$$

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 $K_2$:

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

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

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

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

### 3.5 Refine

The role of $\text{Refine}$ refines the current knowledge $K$ to make learning more effective. The procedure $\text{Refine}$ 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 $(K, G, H)$, $\text{Refine}(K, G, H)$ produces the refined knowledge $K'$ as follows:

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

It first augments the previous knowledge $K$ with the new one $G$ and then removes the set $\text{Kill}$ from the result. Intuitively, $\text{Kill}$ denotes the parent heuristics that are turned out to be no longer useful at the current iteration of the algorithm; $\text{Kill}$ 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, $\text{Kill}$ is defined as follows:

$$\text{Kill} = \{(h', B') \in K \mid (h'_j, \ldots) \in H, \bigcup_{(h', B') \in G} B \subseteq \bigcup_{(h, B) \in K} B\}$$

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

**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 $\{(h_5, \{3, 4, 6\}), (h_6, \{1, 2, 3, 4, 5, 6\}), (h_7, \{5, 7\}), (h_8, \{3, 8\})\}$ with the following profiles:

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

Then, the set $\text{Kill}$ is as follows:

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

because the offspring of $h_3$ 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:

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

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 ($\eta_1$, $\eta_2$, $\eta_3$, and $\eta_4$) for which appropriate values are assumed to be given beforehand. The first hyperparameter $\eta_1$ determines the pool size of search heuristics. $\eta_2$ in the $\text{Select}$ 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 $\text{Switch}$ procedure; $\eta_3$ determines the number of offspring to be generated from each effective heuristic and the last one $\eta_4$ is the exploitation rate. In experiments, we set $\eta_1 = 100$, $\eta_2 = |K| \times 0.03$, $\eta_3 = 10$, 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 $\eta_4$, 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 $\eta_4$.

### 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

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

(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 included 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.

**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. 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,
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
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\|\W*\)’ 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\|\W*\)’ 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
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 of 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: $I_2(146), I_3(300), I_4(417), \ldots, I_{13}(588)$. 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.
would be possible to use algorithms for tuning hyperparameters. We found hyperparameters by trial and error. To be systematic, it was crucial for obtaining the desired results. For example, Table 5 Algorithm 1 covered 14,394 branches only.

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

<table>
<thead>
<tr>
<th>Exploitation rate</th>
<th>0%</th>
<th>20%</th>
<th>40%</th>
<th>60%</th>
<th>80%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td># branches</td>
<td>1,612</td>
<td>1,611</td>
<td>1,638</td>
<td>1,679</td>
<td>1,696</td>
<td>1,672</td>
</tr>
</tbody>
</table>

4.5 Discussions

Exploration and Exploitation. In our algorithm (Section 3), the hyperparameter $\eta$ 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 $\eta$ 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 latter 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 algorithm 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).

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

<table>
<thead>
<tr>
<th>Exploitation rate</th>
<th>0%</th>
<th>20%</th>
<th>40%</th>
<th>60%</th>
<th>80%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td># branches</td>
<td>1,612</td>
<td>1,611</td>
<td>1,638</td>
<td>1,679</td>
<td>1,696</td>
<td>1,672</td>
</tr>
</tbody>
</table>

Table 6: Average branch coverage achieved by each heuristic on Algorithm 1 and 2 (24h).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>grep-2.2</th>
<th>sed-1.17</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A2</td>
<td>A1</td>
</tr>
<tr>
<td>gawk-3.0.3</td>
<td>3,350</td>
<td>3,349</td>
</tr>
<tr>
<td>CFDS [3]</td>
<td>2,767</td>
<td>3,095</td>
</tr>
<tr>
<td>CGS [28]</td>
<td>3,113</td>
<td>3,091</td>
</tr>
<tr>
<td>Random [5]</td>
<td>2,336</td>
<td>3,184</td>
</tr>
<tr>
<td>Gen [12]</td>
<td>2,828</td>
<td>2,939</td>
</tr>
<tr>
<td>Param [5]</td>
<td>9,789</td>
<td>10,172</td>
</tr>
</tbody>
</table>

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


