How We Get There: A Context-Guided Search Strategy in Concolic Testing

Hyunmin Seo and Sunghun Kim

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{hmseo, hunkim}@cse.ust.hk

ABSTRACT

One of the biggest challenges in concolic testing, an automatic test generation technique, is its huge search space. Concolic testing generates next inputs by selecting branches from previous execution paths. However, a large number of candidate branches makes a simple exhaustive search infeasible, which often leads to poor test coverage. Several search strategies have been proposed to explore high-priority branches only. Each strategy applies different criteria to the branch selection process but most do not consider *context*, how we got to the branch, in the selection process.

In this paper, we introduce a context-guided search (CGS) strategy. CGS looks at preceding branches in execution paths and selects a branch in a new context for the next input. We evaluate CGS with two publicly available concolic testing tools, CREST and CarFast, on six C subjects and six Java subjects. The experimental results show that CGS achieves the highest coverage of all twelve subjects and reaches a target coverage with a much smaller number of iterations on most subjects than other strategies.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Concolic testing, symbolic execution, search strategies

1. INTRODUCTION

Recently, an automatic test generation technique called concolic testing [31] or Directed Automated Random Testing (DART) [16] has received much attention due to its low false positives and high code coverage [11]. Concolic testing runs a subject program with a random or user-provided input vector, then it generates additional input vectors by analysing previous *execution paths*. Specifically, concolic testing *selects* one of the *branches* in a previous execution path and generates a new input vector to steer the next execution toward the *opposite* branch of the selected branch. By carefully selecting branches for the new inputs, concolic testing can avoid generating redundant inputs for the same path and achieve high code coverage.

However, the huge *search space* is one of the biggest challenges in concolic testing [10, 11, 3]. The search space in concolic testing is the branches in the execution paths. To generate the next input, concolic testing has to select one branch among a large number of candidate branches. As concolic testing proceeds by generating more input vectors, the search space gets even bigger as more branches are added from new execution paths. Given a limited testing budget, exploring all branches even for a medium-sized application is not practical [10].

To alleviate the search space challenge, search heuristics or search strategies have been proposed [10, 11, 3]. Instead of exploring all branches in the candidate list, search heuristics prioritise branches according to some criteria and only explore high priority branches. For example, the CarFast strategy always selects a branch whose opposite branch is not yet covered, and has the highest number of statements control-dependent on that branch [29]. The CFG strategy calculates the distance from the branches in an execution path to any of the uncovered statements and selects a branch that has the minimum distance first [6]. The Generational strategy measures the incremental coverage gain of each branch in an execution path and guides the search by expanding the branch with the highest coverage gain [17].

However, most strategies do not consider how an execution reaches a branch in the branch selection criterion even though covering the branch may depend on this information. Figure 1 shows an example code snippet consisting of three conditional statements with three possible execution paths over the CFG (Control Flow Graph) of the code. Each diamond in the CFG represents a conditional statement and the left and right edges correspond to the `TRUE` and `FALSE` branches of the conditional. Selecting b_6 from π_1 to generate the next input to cover b_5 is unsuccessful because b_4, the branch taken right before b_6, sets a constraint such that `total` is less than 100. However, `total` must be bigger than or equal to 200 to cover b_5. This introduces conflict between the two constraints ($\texttt{total} < 100 \wedge \texttt{total} \geq 200$). As a result, concolic testing gets an `UNSAT` result from the SMT solver and cannot generate an input vector. This is the same for π_2. However, if b_6 was selected from π_3, concolic testing

FSE'14 , November 16–22, 2014, Hong Kong, China

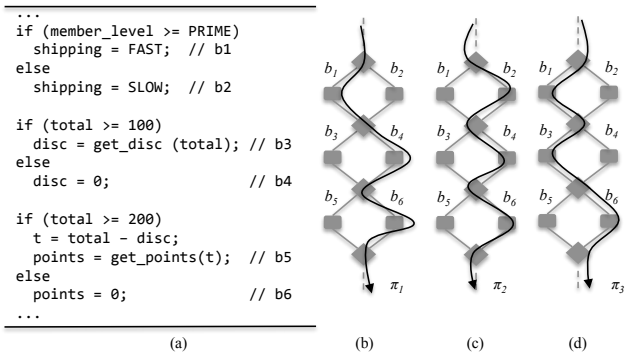


Figure 1: An example code snippet consisting of three conditional statements and three execution paths over the CFG. The left and right branches in the CFG correspond to TRUE and FALSE branches respectively.

could generate an input vector for b_5 because π_3 has taken b_3 and the constraint for b_3 (`total` ≥ 100) does not conflict with the constraint for b_5 (`total` ≥ 200).

Oftentimes, branches in a program have *dependencies* on other branches. In Figure 1, b_5 has a dependency on b_3 such that b_5 can only be covered from the execution paths taking b_3.

Selecting branches by considering their *context*, how the execution reached the branches, can help cover such branches. For example, after selecting b_6 in π_1 and getting an `UNSAT` result, we can see that π_1 has taken b_4 before it came to b_6. There are two more execution paths, π_2 and π_3, which also go through b_6, but they have a different context for b_6 (π_2 has taken b_4 but π_3 has taken b_3 before they came to b_6). Then, we can select b_6 in π_3 instead of selecting b_6 in π_2 because we have already selected a branch having the same context as b_6 in π_2.

In this paper, we introduce the context-guided search (CGS) strategy in which the search is guided by the context of branches. CGS selects a branch under a *new* context for the next input. We define the context of b as a sequence of preceding branches in the execution path. In addition, CGS excludes *irrelevant* branches in the context information by calculating *dominators* of branches.

We implement the CGS strategy on two publicly available concolic testing tools, CREST and CarFast, and evaluate them on six C subjects and six Java subjects. The evaluation results show that CGS achieves a higher coverage than any other strategy used in the evaluation on all twelve subjects. In addition, CGS achieves such coverage with a much smaller number of iterations than other strategies.

Our paper makes the following contributions:

- **Use of context information in branch selection:** We look at how an execution reached b in the branch selection process. Irrelevant branches are excluded in the context information by analysing the static structure of the program.
- **Level-based CGS strategy:** We consider preceding branches located close to b as more important than preceding branches located far from b, and define different levels of context. In addition, we incrementally increase the consideration level of the context.
- **Evaluation on six C subjects six Java subjects:** We evaluate CGS on two publicly available concolic testing tools on six C subjects and six Java subjects.

The remainder of the paper is organised as follows. Section 2 introduces concolic testing and describes several representative search strategies found in the literature. We explain CGS in Section 3. Section 4 shows the evaluation plan and the experimental results are shown in Section 5. Related work is described in Section 6 and we conclude this paper in Section 7.

2. PRELIMINARY

This section describes concolic testing and its search space challenge. In addition, we introduce several representative search strategies proposed to address the challenge in the literature.

2.1 Concolic Testing

Concolic testing [31] or Directed Automated Random Testing (DART) [16] is an automatic test generation technique based on symbolic execution.

The key idea behind symbolic execution [12, 22] is to represent program variables with *symbolic values* instead of concrete values. Symbolic execution maintains a symbolic memory state σ which is a mapping from program variables to symbolic expressions and a symbolic path constraint PC which is a conjunction of conditions collected at each conditional statement along an execution path [10].

Initially, symbolic execution starts with an empty mapping as σ and *true* as PC[1]. For each input variable `v`, a symbolic value s_0 is introduced into the mapping $\{v \mapsto s_0\}$. The symbolic memory state is updated at each assignment statement by representing variables in the program as *symbolic expressions* over the symbolic values. For example, after executing `w = 2 * v` under the current memory state of $\{v \mapsto s_0\}$, the memory state is updated to $\{v \mapsto s_0, w \mapsto 2s_0\}$. When a conditional statement `if (e) S1 else S2` is executed, the symbolic execution follows both branches by forking another symbolic execution. The condition e is evaluated under the current symbolic memory state as $\sigma(e)$ and PC is updated as $PC \wedge \sigma(e)$ for the execution taking the `TRUE` branch and $PC \wedge \neg\sigma(e)$ for the execution taking the `FALSE` branch.

When the symbolic execution reaches the end of the program, concrete input vectors for each execution path can be generated by solving the collected PC with an SMT solver.

Concolic testing performs symbolic execution dynamically by running the target program with a concrete input vector and performing symbolic execution along the execution path of the input. After finishing the symbolic execution, the PC consists of the symbolic constraints at each conditional statement encountered along the execution path ($PC = pc_1 \wedge pc_2 \wedge \ldots \wedge pc_n$).

Each pc_i corresponds to either the `TRUE` or `FALSE` branch taken at the i-th conditional statement. Concolic testing *selects* a pc_i (or the corresponding branch) and formulates a new path constraint by negating pc_i while maintaining the same pc_j for $0 < j < i$ ($PC' = pc_1 \wedge pc_2 \wedge \ldots \wedge \neg pc_i$). The new PC' represents an execution path taking the exact same branches as the previous execution path until pc_{i-1}, however it takes the opposite branch at pc_i. A new input vector for this execution path is generated by solving PC' with an SMT solver.

[1]We explain symbolic execution following the terms and notations used in [10].

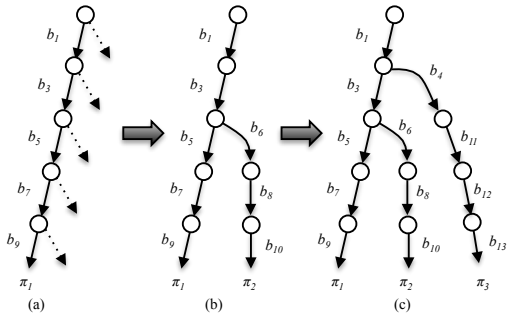


Figure 2: Concolic testing selects a branch for the next input. The new execution paths are added to build the execution tree.

Concolic testing runs the target program with the newly generated input vector and performs symbolic execution again along the new execution path. If PC' is unsatisfiable, the SMT solver returns an `UNSAT` result and concolic testing selects a different *pc*. This process repeats for a given number of iterations or until a target coverage goal is achieved.

One of the key benefits of concolic testing over pure symbolic execution is that concolic testing can use concrete values to overcome the limitations of symbolic execution [10, 11]. For example, when a PC contains complex constraints such as non-linear expressions or floating point operations of which SMT solvers cannot deal with efficiently [15, 21], concolic testing can simplify the constraints with concrete values to solve it. Even though this concretisation may introduce imprecision, it allows concolic testing to generate input values where symbolic execution may not [17, 10].

2.2 Search Space in Concolic Testing

The execution paths explored in concolic testing form a tree. Figure 2 shows three execution paths building a tree up. The set of all feasible execution paths is called an *execution tree*. Concolic testing can be viewed as building the complete execution tree starting from an initial execution path. The search space in concolic testing is the branches in the current execution tree. The branch selected for the next input determines which path to add to the tree. If all branches are chosen without missing any branch, concolic testing eventually builds the complete execution tree.

One of the biggest challenges in concolic testing is that there are often too many branches to select for the next input. This is referred to as the *path explosion* problem [10, 11, 3]. The number of paths in the execution tree increases exponentially with the number of branches in the program. Visiting only the top twenty branches in the execution tree in a breadth first search (BFS) order requires more than one million concolic runs (2^{20}). However, programs usually have far more than twenty branches, for example, an execution path of `grep`, a 15K line of code program, contains more than 8,000 branches. Therefore, exploring all paths in an execution tree in a reasonable amount of time is not feasible.

To mitigate this challenge, search heuristics or search strategies have been proposed [10, 11, 3]. Instead of exploring all branches in an execution tree, search strategies prioritise some branches and only explore those high priority branches.

Algorithm 1 shows a generic search strategy [7, 19]. The algorithm starts with an initial execution path as the execution tree (line 1). A branch is selected from the currently built execution tree T (line 3). If there is an input vector

Algorithm 1: Generic Search Strategy

Input: A target program P and an execution path p
Output: A set of test input and coverage information

```
1 T ← p  //initialise execution tree with p
2 while termination conditions are not met do
3 |   b ← pick a branch from T
4 |   p ← get execution path of b from T
5 |   if ∃I that forces P toward b_0, ..., b̄ along p then
6 |   |   q ← ConcolicRun(P,I)
7 |   |   Add q to T
8 |   end if
9 end while
```

guiding the execution toward the opposite branch of the selected branch (line 5), the program is run concolically with the new input (line 6) and the new path is added to T (line 7). The algorithm repeats till the termination conditions are met (line 2). In general, the algorithm terminates when a coverage goal is achieved or a given testing budget is reached. Search strategies decide which branch to choose in line 3.

2.3 Search Strategies

In this section, we introduce several representative search strategies which are trying to improve coverage in general in the literature. Search strategies focusing on covering specific branches are introduced in Section 6.

2.3.1 DFS and BFS

A typical tree traversal algorithm such as DFS has been used to explore an execution tree [16, 31]. However, the DFS strategy has several limitations. First, when the program contains a loop or recursion whose termination condition is dependent on symbolic input, DFS may be trapped in the loop or recursion by continuously generating input vectors which only increase the number of iterations of the loop or recursion during the execution. Bounding the depth is one way to overcome this problem [31, 6]. However, the maximum depth is set arbitrarily and the branches located beyond the maximum depth cannot be selected using this approach. In addition, since it selects branches in the increasing order of depth, generating input vectors becomes harder as the depth increases as the number of involved constraints also increases.

The breadth-first search (BFS) strategy traverses the execution tree according to a BFS order. The BFS strategy prefers branches that appear early in the execution paths, therefore generating new input vectors is easier because a smaller number of constraints will be involved for those branches. On the other hand, branches that only appear later in the execution path have little chance of being selected during a given testing budget.

In theory, both DFS and BFS strategies can cover all execution paths in the execution tree. However, as described in the previous section, real world programs have a non-trivial number of execution paths and neither strategy scales to even medium-sized programs [10, 11, 3]. In practice, both strategies may end up with a skewed search area as shown in Figure 3a and 3b which may result in low coverage.

2.3.2 Random Search

To overcome the skewed search area limitation, random strategies have been proposed to provide a scattered search area as in Figure 3c. Uniform Random Search traverses the

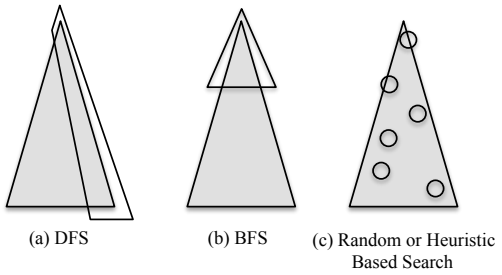


Figure 3: The search area of each strategy. The large coloured triangle represents the execution tree. DFS and BFS have a skewed search area and random or heuristic based strategies have a scattered search area.

execution tree from the root and randomly selects a branch to follow [6, 10]. For example, at the first branch in the initial execution path, it flips a coin. If the result is heads, it follows the current execution path and moves onto the second branch. If the result is tails, it tries to follow the opposite branch by generating an input vector for that branch. It then moves onto the next branch in the new execution path. The strategy repeats this process by flipping a coin at the next branch it visits.

Instead of traversing the execution tree randomly, Random Branch Search [6] selects a branch randomly from the last execution path only. After executing the program with a new input vector, the strategy selects another branch randomly from the new execution path. Evaluation results show that Random Branch Search is more effective than Uniform Random Search or DFS [6]. Even though random strategies have shown better coverage than DFS, covering new branches relies on randomness.

2.3.3 CarFast

To improve the coverage further, strategies exploiting coverage information have been introduced. CarFast [29] is a prioritised greedy strategy. A greedy strategy always selects a branch whose opposite branch is not yet covered so that additional coverage gain is achieved with the new input vector. In addition, CarFast prioritises those branches according to a score value which is the expected number of statements coverable when the branch is selected. This is measured by calculating the number of statements transitively control-dependent on each branch. CarFast strategy selects a branch whose opposite branch has the highest score value and is not yet covered.

2.3.4 CFG-Directed Search

CFG-directed search combines the coverage information with the static structure of the program to guide the search [6]. For each branch in an execution path, it calculates the *distance* from the opposite branch to any of the currently uncovered statements. The distance is measured by the summation of the weight of the shortest path in the control flow graph (CFG) after assigning each branch-edge weight one and all other edges weight zero. The strategy selects a branch with the minimum distance first. The intuition behind this approach is that an uncovered statement located close to the current execution path is easier to cover than a branch located far from the current path.

2.3.5 Generational Search

Generational search uses the incremental coverage gain of each branch to guide the search. It is a strategy used in SAGE, a white-box fuzz testing tool based on dynamic symbolic execution [17]. Instead of selecting only one branch for the next input, generational search selects all the branches in an execution path and generates a set of input vectors. These inputs become a *generation*. The program under test is run with each of the new input vectors and the incremental coverage gain is measured for each input. The execution path of the input with the largest coverage gain is chosen for the next generation. Again, all the branches in the new execution path are selected and the newly generated input vectors become the next generation. Generational search repeats this process by selecting an execution path with the largest coverage gain.

2.4 Limitations

Due to the path explosion, DFS and BFS cannot search the whole space within the limited testing budget and may end up with a skewed search area. Branches located within the search area may be selected several times while branches only located outside the search area may not be selected. This typically results in low coverage. Random strategies have scattered search areas but the branch selection relies on randomness. A greedy strategy does not select branches if their opposite branches are already covered but this may put limitations on the search area.

Moreover, most heuristic-based strategies such as CarFast [29], CFG-directed [6] and generational search [17] focus on coverage information in the branch selection process but do not consider how the execution reaches the branch. However, as we showed in Section 1, certain branches have dependencies on other branches and looking at how the execution reaches the branch can help cover such branches efficiently.

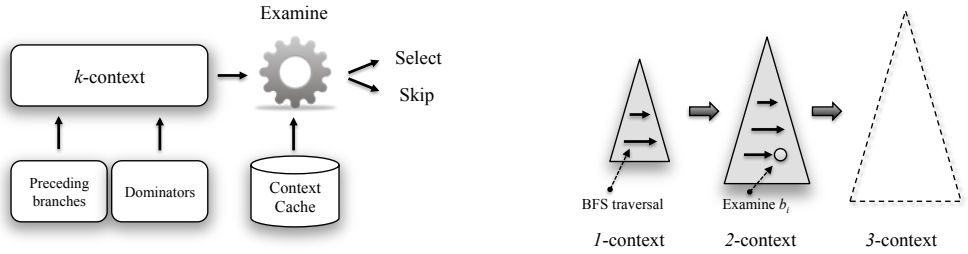
3. CONTEXT-GUIDED STRATEGY

This section describes our context-guided search (CGS) strategy. We first show an overview of CGS. We then define *context* and *dominator* and explain the strategy in detail.

3.1 Overview

CGS explores branches in the current execution tree. For each visited branch, CGS *examines* the branch and decides whether to *select* the branch for the next input or *skip* it. Figure 4a shows the branch selection process. CGS looks at how the execution reaches the current branch by calculating *k-context* of the branch from its preceding branches and dominator information. We explain context and dominator in the following sections. Then, the *k-context* is compared with the context of previously selected branches which is stored in the *context cache*. If the *k-context* is new, the branch is selected for the next input. Otherwise, CGS skips the branch.

Figure 4b shows how CGS builds the execution tree. CGS visits branches according to a BFS order under different *levels* of contexts. First, CGS examines branches based on their *1-context*. After examining the last branch of the current execution tree, CGS increases the context level to *2-context* and traverses the tree again to examine previously skipped branches. Figure 4b shows CGS finished the traversal under *1-context* and is currently examining b_i at the *2-context* level. As CGS increases the context level, more branches are selected for the next input and the execution tree grows further with the new execution paths.



(a) Branch selection **(b) Incremental search**

Figure 4: Overview of CGS

3.2 Context

We define the context of b as a sequence of preceding branches appearing in an execution path including b itself. Context information shows us how the execution reaches b. In addition, we define *k-context* of b as a sequence of k preceding branches in an execution path. For example, π_1 in Figure 1b consists of branches (b_1, b_4, b_6). Then, *2-context* of b_6 becomes (b_4, b_6).

CGS examines each branch by its context and selects a branch having a *new* context only for the next input, for example, when CGS visits b_6 in π_1 in Figure 1, CGS calculates its context as (b_4, b_6)[2]. Since this context is new, CGS selects the branch and tries to generate a new input vector. CGS stores the context into the context cache and moves on to examine the next branch. When CGS visits b_6 from π_2, CGS skips it since it has the same context, (b_4, b_6), which has been explored before. However, when CGS visits b_6 from π_3, CGS selects b_6 for the next input since it has the new context of (b_3, b_6).

The size of the context affects the performance of CGS. Under ∞-*context*, the full context of each branch is considered and CGS becomes the same as the BFS strategy, selecting every branch for the next input since each branch has a different context under ∞-*context*. On the other hand, under *1-context*, each branch itself becomes its own context and CGS selects each branch only once. This causes CGS to skip a large number of branches but might result in low coverage.

The optimal k may vary depending on the characteristics and size of the test subjects and testing budget. Instead of setting k to a fixed number, CGS incrementally increases k by considering preceding branches located close to b as more important than preceding branches located far from b.

For example, CGS starts with *1-context* and traverses the execution tree. Since CGS skips a large number of branches under *1-context*, it quickly finishes the traverse. Then, CGS increases k to *2-context* and traverses the execution tree again from the top. At this time, CGS examines the previously skipped branches and selects branches based on their *2-context*. CGS continuously increases k after the end of each traversal of the execution tree within the given testing budget.

Incrementally increasing k has the benefit that it can improve coverage faster than starting with a fixed k even though both may yield similar coverage in the end. We compare the coverage differences between incremental-k and fixed-k strategy in Section 5.3.

[2]Suppose we consider *2-context* currently and b_6 has never been visited before.

3.3 Dominator

Depending on the structure of CFG, it is possible that all the execution paths reaching a branch have the same *k-context* when k is small. Figure 5 shows part of the CFG of function `regex_compile` in `grep`. Due to the structure with deeply nested conditional statements, all the execution paths going to b_{11} have the same *k-context* if k is smaller than or equal to five. For example, b_{11} has the same *5-context* of $(b_3, b_5, b_7, b_9, b_{11})$ in both execution paths π_1 and π_2[3].

However, the branches in this *5-context* are *irrelevant* for finding a *different* context for b_{11} since every execution path to b_{11} must go through them. To exclude irrelevant branches in the context, we calculate *dominator* information.

In CFG, node d dominates node n, written as d *dom* n, if every path from the entry node to node n must go through node d [1]. From the definition, it follows that if $p_1, p_2, \ldots, p_k$ are all predecessors of n, and $d \neq n$, then d *dom* n if and only if d *dom* p_i for each i [1]. Therefore, finding dominators of node n can be formulated as finding the maximal fixed-point solution to the following data flow equation [2, 20].

$$Dom(n_0) = \{n_0\} \tag{1}$$

$$Dom(n) = \left(\bigcap_{p \in preds(n)} Dom(p) \right) \cup \{n\} \tag{2}$$

The dominator concept is defined for nodes in the CFG but we can apply it to edges (branches) also such that b_x *dom* b_y if every path from the entry node to b_y must go through b_x. For example, in Figure 5, b_3 dominates b_{11} since all the execution paths heading for b_{11} must go through b_3.

After calculating dominators, we consider non-dominating branches only in the context information. For example, the *2-context* of b_{11} in π_1 in Figure 5 becomes (b_1, b_{11}) instead of (b_9, b_{11}) since b_3, b_5, b_7 and b_9 are dominators of b_{11}. With dominator information, CGS can find execution paths reaching b_{11} with different context by increasing the context level to *2-context* only instead of increasing it to *6-context*.

3.4 CGS Algorithm

Algorithm 2 shows CGS in detail. We use a list of branches at each depth in the tree for the BFS traversal (line 6). The depth is initialized to one (line 4) and increased after examining all branches at the current depth (line 17).

For each randomly selected branch in the list, *k-context* of the branch is calculated and the context is checked for

[3]In other words, b_{11} is control-dependent on b_3, b_5, b_7 and b_9.

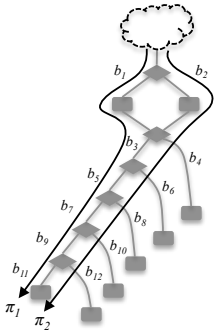


Figure 5: An example complex CFG which is part of the CFG of function `regex_compile` in `grep`.

whether it is new or not (line 8). The branch with a new context is selected for the next input and if it has a satisfying input vector for the negated path condition (line 12), the target program is run with the new input vector (line 13) and the new path is added to the execution tree (line 14).

The algorithm starts with the minimum context size (line 2). After the BFS traversal is over, we increase the size of context by one (line 19) and start the traversal again from the top of the tree (line 4). From the second traversal, we examine previously skipped branches (line 8). The algorithm repeats until the termination conditions are met (line 3).

4. EVALUATION

This section describes the evaluation plan for CGS including research questions, concolic testing tools, evaluation strategies and subjects.

4.1 Research Questions

We designed our evaluation to address the following research questions.

- **RQ1. Given the same testing budget, how many branches can each strategy cover?** We measure the number of branches covered by each strategy given the same testing budget. A strategy achieving a higher coverage is better.
- **RQ2. Give a target coverage goal, how many iterations does each strategy require to achieve that goal?** How fast a strategy reaches a target coverage is another common criterion to evaluate the performance of search strategies [29, 24]. When enough testing budget is given, different strategies may ultimately reach a similar coverage. However, the number of iterations required to reach that coverage might be different. A strategy achieving the same coverage with a smaller number of iterations is better.

Table 1: Subjects used in the experiments.

Subject	Testing tool	Language	LOC
`grep`	CREST	C	19K
`replace`	CREST	C	0.5K
`expat`	CREST	C	18K
`cdaudio`	CREST	C	2K
`floppy`	CREST	C	1.5K
`kbfiltr`	CREST	C	1K
`tp300`	CarFastTool	Java	0.3K
`tp600`	CarFastTool	Java	0.6K
`tp1k`	CarFastTool	Java	1.5K
`tp2k`	CarFastTool	Java	2.4K
`tp5k`	CarFastTool	Java	5.8K
`tp10k`	CarFastTool	Java	28K

Algorithm 2: Context-Guided Search Strategy

Input: A target program P and an execution path p
Output: A set of test input and coverage information

```
1  T ← p  //initialize execution tree with p
2  k ← 1 // size of context
3  while termination conditions are not met do
4      d ← 1 // BFS depth
5      while d ≤ depth of T do
6          blist ← get branches at depth d from T
7          for b in blist do
8              if (k-context of b is not new) or (b has been
               selected before) then
9                  continue
10             end if
11             p ← get execution path of b from T
12             if ∃I that forces P toward b_0, ..., b̄ along p then
13                 q ← ConcolicRun(P,I)
14                 Add q to T
15             end if
16         end for
17         increase d by 1
18     end while
19     increase k by 1
20 end while
```

4.2 Testing Tools and Evaluation Strategies

We evaluate CGS on top of two publicly available concolic testing tools, CREST [13] and CarFastTool [29].

CREST is an automatic test generation tool for programs written in C. We chose CREST since it has been widely used in previous work [6, 21, 14]. CREST's test driver comes with DFS, Random Branch Selection and CFG-directed strategies. In addition, we implemented CGS, CarFast and Generational strategies in the test driver of CREST.

CarFast is a search strategy and implemented in a Java concolic testing tool. To avoid confusion, we call the strategy CarFast and the testing tool CarFastTool. We chose CarFastTool as another testing tool since we wanted to test programs in a different language other than C, and CarFastTool is one of the most recently published concolic testing tools for Java programs. We implemented CGS in the test driver of CarFastTool. With CarFastTool, we only compare CGS with the CarFast strategy since the previous work [29] already showed that CarFast outperforms DART [16] which is based on the DFS strategy, and other random approaches on the same subjects.

We conducted the experiments on a linux machine equipped with Intel Xeon 2.67GHz CPU and 64GB RAM. Since the coverage depends on the initial input vector, we conducted the experiments 100 times with a random initial input[4] and calculated the average coverage. For CarFastTool, we conducted the experiments 10 times since CarFastTool took a much longer testing time than CREST.

4.3 Evaluation Subjects

We used six open-source C programs used in [6, 9, 5, 19] as the evaluation subjects for CREST. First is `grep`, a text search program supporting regular expressions, `replace` is a text processing program included in CREST while `expat` is an open-source XML parser library. The other three

[4]For `expat`, we used a sample XML file for the initial input.

Table 2: The number of branches covered by each strategy on six C subjects at different iterations. The numbers inside the parenthesis show the coverage improvements over the last 1,000 iterations.

Subject	Strategy	Iterations			
		1000	2000	3000	4000
grep	CGS	**1523.0**	**1643.3**	**1690.4**	**1721.6** (+1.8%)
	CFG	1404.5	1455.7	1479.8	1495.8 (+1.1%)
	Random Branch	1317.0	1371.9	1397.3	1412.7 (+1.1%)
	Generational	1032.5	1199.3	1224.6	1255.6 (+2.5%)
	DFS	948.1	989.9	1087.2	1099.9 (+1.2%)
	CarFast	1197.9	1223.1	1240.8	1253.9 (+1.1%)
replace	CGS	**180.0**	**180.8**	**181.0**	**181.0** (+0.0%)
	CFG	175.1	176.3	176.6	177.0 (+0.2%)
	Random Branch	167.4	171.9	173.2	174.2 (+0.5%)
	Generational	165.2	170.7	175.8	175.8 (+0.0%)
	DFS	84.3	157.0	169.6	170.6 (+0.6%)
	CarFast	151.0	152.9	154.0	155.8 (+1.2%)
expat	CGS	**1040.3**	**1131.6**	**1201.2**	**1248.0** (+3.9%)
	CFG	899.7	972.7	1036.8	1073.7 (+3.6%)
	Random Branch	677.2	677.4	677.5	677.6 (+0.0%)
	Generational	703.0	715.0	717.0	719.2 (+0.3%)
	DFS	670.0	670.0	670.0	670.0 (+0.0%)
	CarFast	739.8	764.9	789.7	819.0 (+3.7%)
cdaudio	CGS	**250.0**	**250.0**	**250.0**	**250.0** (+0.0%)
	CFG	246.0	249.0	249.6	249.7 (+0.0%)
	Random Branch	220.8	233.8	239.3	241.9 (+1.1%)
	Generational	250.0	250.0	250.0	250.0 (+0.0%)
	DFS	242.0	242.0	242.0	242.0 (+0.0%)
	CarFast	122.0	122.0	122.0	122.0 (+0.0%)
floppy	CGS	**205.0**	**205.0**	**205.0**	**205.0** (+0.0%)
	CFG	199.8	203.8	204.6	204.9 (+0.1%)
	Random Branch	133.4	150.4	159.2	165.0 (+3.7%)
	Generational	205.0	205.0	205.0	205.0 (+0.0%)
	DFS	186.3	186.3	186.3	186.3 (+0.0%)
	CarFast	49.0	49.0	49.0	49.0 (+0.0%)
kbfiltr	CGS	**149.0**	**149.0**	**149.0**	**149.0** (+0.0%)
	CFG	147.6	149.0	149.0	149.0 (+0.0%)
	Random Branch	143.9	147.7	148.6	148.8 (+0.1%)
	Generational	149.0	149.0	149.0	149.0 (+0.0%)
	DFS	137.0	137.0	137.0	137.0 (+0.0%)
	CarFast	109.0	109.0	109.0	109.0 (+0.0%)

programs, `cdaudio`, `floppy` and `kbfiltr` come from the SV-COMP [32] benchmark which is used in the competition for software verification.

For CarFastTool, we used six Java subjects included in the benchmark set coming with CarFastTool. They are synthesised Java programs of different sizes generated by a set of predefined rules. We could not add real-world programs into the evaluation subjects for CarFastTool, since the tool's symbolic execution and constraint solving technique are specialized for testing the benchmark programs and cannot handle real-world programs. Table 1 shows some statistics about the testing subjects.

5. RESULTS

We show the evaluation results of the search strategies for various subjects in this section. In addition, we discuss the effects of the increasing-k search and dominator.

5.1 Coverage in CREST

This section presents the coverage results of six search strategies on six C subjects experimented on CREST. We first show the coverage achieved by each strategy given the same testing budget to answer RQ 1.

Table 2 shows the number of covered branches on six C subjects at 1,000, 2,000, 3,000 and 4,000 iterations. We counted the number of unique branches in the CFG which are covered by the execution paths of the inputs generated in the experiment. For example, at 1,000 iterations, CGS covered 1523.0 branches on average on `grep` while CFG covered 1404.5 branches. Random Branch covered 1317.0 branches and DFS covered 948.1 branches only.

The results show that CGS achieved the highest coverage on all six subjects and at all different iteration points. On `grep` and `expat`, CGS covered 1721.6 and 1248.0 branches respectively which are 225.8 and 174.3 more branches than CFG, the second best strategy. CGS also achieved the highest coverage on `replace`. We conducted the Mann-Whitney U test and found the coverage differences between CGS and the second best strategy on `grep`, `expat` and `replace` are statistically significant with p-value less than 0.01.[5] On `cdaudio`, `floppy` and `kbfiltr`, CGS and Generational covered the same number of branches being 250, 205 and 149 respectively. On `kbfiltr`, CFG also reached the same coverage at 2,000 iterations.

Unlike CGS, other strategies showed different performances depending on the subjects. For example, CFG was the second best strategy on `grep`, `replace` and `expat` but not on the other subjects. Generational search did not perform well on `grep`, `replace` and `expat` but was one of the best strategies

[5]The results of Shapiro-Wilk test [30] showed that we can reject normal distribution hypothesis with p-value lower than 0.01 on most cases.

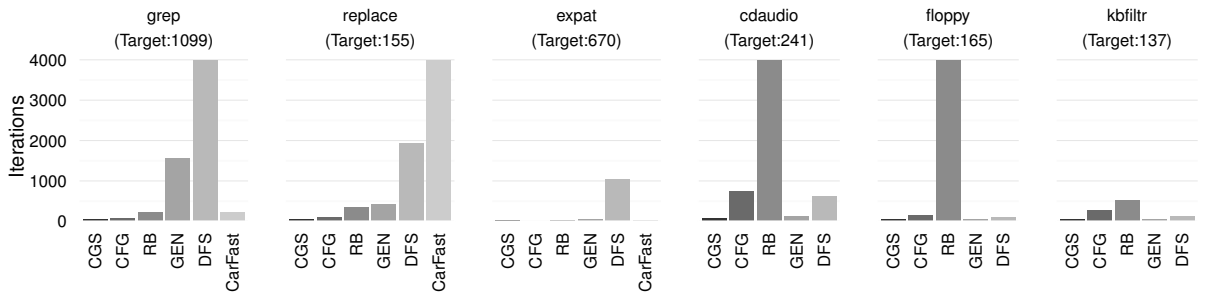


Figure 6: The number of iterations required to reach a target coverage on six subjects in CREST.

on `cdaudio`, `floppy` and `kbfiltr`. On the other hand, CGS consistently yielded the highest coverage on all six subjects.

Table 2 also shows that most strategies reached a coverage plateau [14] at 4,000 iterations. Except CGS and CFG, which continuously improved coverage by more than 3% on `expat`, the other strategies have about 1% or less improvement in coverage over the last 1,000 iterations. We conducted additional experiments on `expat` for CGS and CFG and counted the number of branches covered at 6,000 iterations, which is 50% more testing budget. Still, CGS covered 118.9 more branches than CFG by covering 1276.7 branches while CFG covered 1157.8 branches. On `cdaudio`, `floppy` and `kbfiltr`, CGS and Generational already reached a coverage plateau after 1,000 iterations.

We then compared the covered branch set between the strategies. For each strategy, we combined all the branches covered during 100 experiments, then we compared the covered branch set between CGS and other strategies. On `replace`, `cdaudio`, `floppy` and `kbfiltr`, CGS's covered branch set included all the branches covered by other strategies. On the other hand, on `grep`, we found 61 branches which were covered by CFG but not covered by CGS. Other strategies also covered a few number of branches which were not covered by CGS. For example, there were two and four branches covered by Generational and DFS respectively but not covered by CGS. Similarly, on `expat`, we found 191 branches covered by CFG but not covered by CGS. Generational and DFS also covered 63 and 27 branches respectively which were not covered by CGS. Even though CGS covered more branches given the same testing budget, the differences on covered branch set show that each strategy can explore different parts of the program. For better coverage, different strategies can be combined to complement each other.

To answer RQ 2, we set a target coverage goal and measured the number of iterations required by each strategy to reach the target. We set the lowest achieved coverage on each subject at 4,000 iterations as the target coverage. For `cdaudio`, `floppy` and `kbfiltr`, the difference between the lowest coverage and the others was too large so we chose second to the lowest coverage as the target.

Figure 6 shows the number of iterations by each strategy required to reach the target coverage. For example, we set 1099, the coverage achieved by DFS on `grep`, as the target coverage for `grep`. The bar graph in Figure 6 shows that DFS reached the target coverage at 3,981 iterations. CGS reached this coverage with the smallest number of iterations, 47, followed by CFG which reached it at 90 iterations. CarFast reached it at 239 iterations, RandomBranch at 223 and Generational at 1554 iterations.

CGS reached the target coverage with the smallest number of iterations on most subjects except `expat`. On `expat`, CFG reached the target coverage first after 12 iterations followed by CGS which took 19 iterations. However, Table 2 shows that CGS consistently achieved a much higher coverage than CFG after 1,000 iterations.

Even though CGS and Generational reached the same coverage on `cdaudio`, `floppy` and `kbfiltr` in Table 2, Figure 6 shows that CGS reached it faster than Generational. CGS reached the target coverage after 82, 42 and 45 iterations on `cdaudio`, `floppy` and `kbfiltr`, while Generational reached it after 137, 54 and 47 iterations respectively.

> Overall, the coverage results on C subjects show that CGS outperforms other strategies. CGS achieved the highest coverage on all six subjects and reached the target coverage first on five out of six subjects.

5.2 Coverage in CarFastTool

This section presents the evaluation results of CGS and CarFast on six Java subjects experimented on CarFastTool. We first show the coverage results given the same testing budget for RQ1.

Table 3 shows the number of branches covered by CGS and CarFast at different iterations. For example, at 500 iterations, CGS covered 982.6 branches on `tp300` while CarFast covered 966.3 branches on average. At 1,500 iterations, CGS covered 987.0 branches and CarFast covered 972.2 branches.

The results in Table 3 show that CGS achieved a higher coverage on all six subjects at all iteration points. On `tp300`, the smallest subject in the experiments, CGS covered 14.8 more branches than CarFast. The differences between CGS and CarFast become bigger as the size of the subjects increases. On `tp10k`, which is the largest subject in the ex-

Table 3: The number of branches covered by CGS and CarFast on six Java subjects at different iterations. The numbers inside parenthesis show the coverage improvements over the last 500 iterations.

Sub	Strategy	Iterations		
		500	1000	1500
tp300	CGS	**982.6**	**987.0**	**987.0** (+0.0%)
	CarFast	966.3	971.0	972.2 (+0.1%)
tp600	CGS	**1658.3**	**1666.5**	**1668.1** (+0.1%)
	CarFast	1620.1	1632.6	1639.3 (+0.4%)
tp1k	CGS	**3777.5**	**3831.9**	**3832.2** (+0.0%)
	CarFast	3741.9	3757.3	3764.0 (+0.2%)
tp2k	CGS	**5862.1**	**5905.6**	**5913.7** (+0.1%)
	CarFast	5737.0	5800.5	5818.2 (+0.3%)
tp5k	CGS	**15092.8**	**15318.3**	**15487.1** (+1.1%)
	CarFast	15063.1	15310.0	15392.2 (+0.5%)
tp10k	CGS	**66525.4**	**67402.0**	**67750.9** (+0.5%)
	CarFast	66215.0	67089.5	67404.7 (+0.5%)

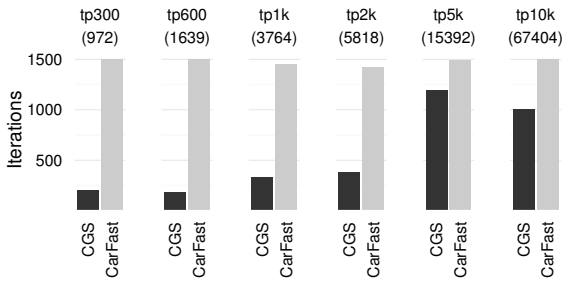


Figure 7: Number of iterations required to reach a target coverage in CarFastTool.

periments, CGS covered 346.2 more branches than CarFast by covering 67750.9 branches. To see if the coverage differences between the two strategies are significant, we conduced the Mann-Whitney U test and found that the coverage differences are statistically significant with the p-value lower than 0.01 on all six subjects. Since CarFastTool only reported the coverage summaries, we could not compare the covered branch sets directly.

Table 3 also shows that both strategies reached a coverage plateau at 1,500 iterations having less than 1% coverage gain over the last 500 iteration on most subjects.

To see how fast each strategy improves coverage, we set a goal coverage and measured the number of iterations required to reach that goal. We set the coverage achieved by CarFast at 1,500 iterations as the goal coverage.

Figure 7 shows the number of iterations required by each strategy to reach the goal. For example, we set 972 as the goal coverage for `tp300`. Figure 7 shows that CGS reached the goal after 206 iterations while CarFast reached it after 1500 iterations. On `tp10k`, the goal coverage was 67404 and CGS reached it after 1002 iterations while CarFast reached it after 1500 iterations.

On all six subjects, CGS reached the target coverage with a significantly less number of iterations than CarFast. For `tp300` and `tp600`, CGS needed only 13.7% and 11.9% of the iterations needed for CarFast to reach the goal. As the subject size becomes bigger, the difference between the required number of iterations becomes smaller. Even though, CGS reaches the goal much faster than CarFast on `tp1k` and `tp2k` with only 22.3% and 26.6% of the number of iterations needed for CarFast. On `tp5k` and `tp10k`, CGS reached the target with 79.7% and 66.8% of the number of iterations needed for CarFast.

> Overall, CGS outperforms CarFast. CGS achieved the highest coverage and reached the target coverage with a much smaller number of iterations on all six Java subjects.

5.3 Discussion

5.3.1 Increasing-k VS. Fixed-k

CGS increases the context level incrementally starting from one. To see the effects of the increasing-k search, we compared it with the fixed-k search. We ran CGS again for six C subjects by fixing k to five from the beginning and compared the coverage with the results of the original CGS which increases k from one to five.

Figure 8 shows the results on `grep`. Both increasing-k and fixed-k reached a similar coverage after 2,500 iterations. However, increasing-k improved coverage faster than fixed-k. At 1,500 iterations, increasing-k covered about 200 more branches than fixed-k. In particular, increasing-k had a sharp coverage gain between 850 and 1,050 iterations while fixed-k had a similar gain but later, between 2,000 and 2,500 iterations, resulting in slower coverage improvement. In addition, the duration of the sharp gain is different. Increasing-k improved coverage quickly during approximately 200 iterations while fixed-k increased during approximately 500 iterations, again resulting in slower coverage improvement. Evaluation results on the other C subjects showed similar trends where increasing-k improved coverage faster than fixed-k. Moreover, fixed-k did not reach the coverage achieved by increasing-k at 4,000 iterations on all six subjects.

5.3.2 Dominators

CGS uses dominator information to exclude irrelevant branches in the context information. To see if this is helpful for improving coverage, we ran CGS by calculating the context without dominator information (CGS-NoDom).

The results on `grep` are also depicted in Figure 8. CGS-NoDom achieved much lower coverage than CGS at 4,000 iterations. When the context level is one, dominator information is not considered in CGS, therefore CGS and CGS-NoDom showed the same coverage in the beginning. However, as the context level is increased, the dominator information became effective and CGS increased the coverage faster than CGS-NoDom. The results on `replace` and `cdaudio` were similar. CGS-NoDom showed the same coverage in the beginning, but had lower coverage in the end. On the other hand, the effect of dominator was not clear on the other subjects where CGS and CGS-NoDom showed a similar coverage improvement.

5.4 Threats to Validity

We identify the following threats to the validity of our experiment:

- **The subjects and search strategies used in the experiment may not be representative**. We used `grep` and `replace` in the experiment since they come with CREST and have been used to evaluate other strategies in [6]. We chose `expat`, a text parsing program, as another subject to minimise the effects of floating-points and non-linear constraints since CREST's symbolic execution does not support them. We used three more subjects from the SV-COMP benchmark which have been used in other work [19]. Six Java subjects coming with CarFast tool have been chosen to evaluate CGS in the same environment where CarFast has been evaluated [29]. Even though we used six open-source C programs with diverse sources and six synthesised Java programs, they may not be representative of other programs. In addition, there are many other search strategies, and we only compared CGS with five of them. Our approach may yield different results on other subjects and other search strategies.
- **More precise symbolic execution and constraint solving may yield different results.** CREST's symbolic execution does not support floating-points and non-linear constraint solving. CarFast's symbolic execution only deals with integer values. The experimental results might be different with more precise symbolic execution and different constraint solving techniques.

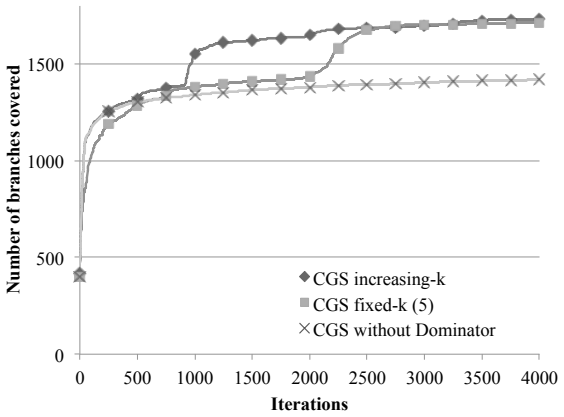


Figure 8: Comparison of CGS increasing-k and fixed-k. The graph also shows the result of CGS without dominator information.

6. RELATED WORK

6.1 Other Search Strategies

Several heuristic-based approaches have been proposed to guide an execution toward a specific branch, which is generally undecidable [33]. Xie et al. [34] introduced a fitness guided path exploration technique. Their technique calculates fitness values of execution paths and branches to guide the next execution towards a specific branch. The use of fitness functions is similar to a traditional search-based testing technique which uses fitness functions to generate input vectors [27], however they combined it with concolic testing. Marinescu et al. [26] introduced a system called KATCH to guide the symbolic execution towards the code of the software patches. It exploits the existing test suite to find a good starting input and uses symbolic execution with several heuristics to generate more inputs to test the patches. For example, to deal with branches having concrete values, it guides the execution toward a different definition location of the variables used in the branches. Our approach focuses on improving coverage in general rather than focusing on specific branches. However, these techniques can be incorporated with our strategy to guide the execution to hard-to-cover branches.

Li et al. [23] introduced a technique which steers symbolic execution to less traveled paths, which is the most similar approach to ours. Whenever a symbolic execution comes to a branch, it forks another state to follow both `TRUE` and `FALSE` branches. To select which state to follow, they used the *subpath* of each state and chose a state having the least frequent subpath. The subpath is a similar concept to context in our approach. However, they used a fixed-size subpath while our approach incrementally increases the size of the context. In addition, they did not consider dominator information in the subpath.

On the other hand, there are techniques combining concolic testing with other testing techniques to explore the search space effectively. Hybrid concolic testing [25] combines random testing and concolic testing. The technique starts from random testing to quickly reach a deep state of a subject program by executing a large number of random inputs. When the random testing *saturates*, without improving coverage for a while, it switches to concolic testing to exhaustively search the state space from the current program state. However, as the authors mentioned, hybrid concolic testing works best for reactive programs that receive inputs periodically while our search strategy best suits programs that have a fixed-sized initial input. Garg et al. [14] introduced a technique combining feedback-directed unit test generation with concolic testing. The strategy starts with a unit testing similar to Randoop [28] and switches to concolic testing when the unit testing reaches a coverage plateau. Since it combines random testing with concolic testing, our strategy can be used in the concolic testing part. KLEE [8] used a meta-strategy which combines several search strategies in a round robin fashion to avoid cases where one strategy gets stuck. CGS selects branches in a new context and this can help prevent the continuous selecting of the same branch.

6.2 Techniques for Path Explosion

Pruning redundant paths is another way to deal with path explosion. Boonstoppel et al. [5] introduced the RWset technique to prune redundant paths during exploration, which is based on two key ideas. First, if an execution reaches a program point in the same state as some previous executions, then the execution produces the same results as before, therefore the exploration can stop at the program point. Second, if two states only differ in program values that are not subsequently read, then the two states produce the same results and one state can be discarded. Jaffar et al. [19] introduced a technique using *interpolation* to subsume execution paths that are guaranteed not to hit a buggy location. Interpolation succinctly represents the reason any branch cannot be covered. They introduced a technique to find full interpolants quickly so that next executions satisfying the interpolant can be subsumed. *Summary* can also be used to alleviate the path explosion problem [18, 4]. A function summary Φ_f is defined as a disjunction of formula Φ_w, where Φ_w is defined as $pre_w \wedge post_w$. pre_w is a conjunction of constraints of the inputs to f and $post_w$ is *effect* of f or a conjunction of constraints of the outputs from f. Since a summary succinctly represents the execution of a function, the summary can greatly reduce the number of paths during a symbolic execution if the function is called frequently from many different locations. However, calculating the summary for a complex function is non-trivial [11].

7. CONCLUSION

An efficient search strategy is a key component in concolic testing to overcome the search space challenge. While most strategies focus on coverage information in the branch selection process, we introduce CGS which considers context information, that is, how the execution reaches the branch. Our evaluation results show that CGS outperforms other strategies. However, we believe further coverage improvement can be achieved. Specifically, a more precise dependency analysis would allow a search strategy to focus on more important branches. It is our future plan to investigate more deeply into such a strategy.

8. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 2nd edition, Sept. 2006.

[2] F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report

IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, 1972.

[3] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. J. Li, and H. Zhu. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 2013.

[4] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 4963 in Lecture Notes in Computer Science, pages 367–381. Springer Berlin Heidelberg, Jan. 2008.

[5] P. Boonstoppel, C. Cadar, and D. Engler. RWset: attacking path explosion in constraint-based test generation. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 4963 in Lecture Notes in Computer Science, pages 351–366. Springer Berlin Heidelberg, Jan. 2008.

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 443–446, Sept. 2008.

[7] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report EECS-2008-123, Berkeley University, 2008.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, Dec. 2008.

[10] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[11] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 2012.

[12] L. A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

[13] CREST. Automatic test generation tool for C. https://code.google.com/p/crest/.

[14] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 132–141, Piscataway, NJ, USA, 2013. IEEE Press.

[15] P. Godefroid and J. Kinder. Proving memory safety of floating-point computations by combining static and dynamic program analysis. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 1–12, New York, NY, USA, 2010. ACM.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[17] P. Godefroid, M. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, 2008.

[18] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 43–56, New York, NY, USA, 2010. ACM.

[19] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 48–58, New York, NY, USA, 2013. ACM.

[20] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, Jan. 1976.

[21] Y. Kim, M. Kim, Y. J. Kim, and Y. Jang. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1143–1152, 2012.

[22] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[23] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 19–32, New York, NY, USA, 2013. ACM.

[24] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[25] R. Majumdar and K. Sen. Hybrid concolic testing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 416–426, May 2007.

[26] P. D. Marinescu and C. Cadar. KATCH: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[27] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[28] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[29] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. CarFast: achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International*

Symposium on the Foundations of Software Engineering, FSE '12, pages 35:1–35:11, New York, NY, USA, 2012. ACM.

[30] N. M. Razali and Y. B. Wah. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of Statistical Modeling and Analytics*, 2(1):21–33, 2011.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[32] SVCOMP. 2014 - competition on software verification. http://sv-comp.sosy-lab.org/2014/benchmarks.php.

[33] E. J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4):587–598, Nov. 1979.

[34] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 359–368, July 2009.