ABSTRACT
We present template-guided concolic testing, a new technique for effectively reducing the search space in concolic testing. Addressing the path-explosion problem has been a significant challenge in concolic testing. Diverse search heuristics have been proposed to mitigate this problem but using search heuristics alone is not sufficient to substantially improve code coverage for real-world programs. The goal of this paper is to complement existing techniques and achieve higher coverage by exploiting templates in concolic testing. In our approach, a template is a partially symbolized input vector whose job is to reduce the search space. However, choosing a right set of templates is nontrivial and significantly affects the final performance of our approach. We present an algorithm that automatically learns useful templates online, based on data collected from previous runs of concolic testing. The experimental results with open-source programs show that our technique achieves greater branch coverage and finds bugs more effectively than conventional concolic testing.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging;

KEYWORDS
Concolic Testing, Online Learning

1 INTRODUCTION
Concolic testing [11, 22] is a popular software testing method that effectively and systematically achieves high code coverage and finds bugs. The key idea of concolic testing is to simultaneously execute a program concretely and symbolically, where new test cases are systematically generated by symbolic execution enhanced with concrete execution. Recently, concolic testing has been used in diverse application domains such as operating systems [18], firmware [8, 16, 31], and binary code [1, 25] among many others.

A major open challenge in concolic testing is how to effectively explore the search space. As the number of execution paths in a realistic program grows exponential, concolic testing must be able to favor and explore the paths that are most likely to benefit the final testing results. However, guiding concolic testing effectively is nontrivial and many different approaches exist with the goal of mitigating the path-explosion problem: e.g., path pruning [2, 3, 17, 28], search heuristics [4, 5, 19, 23, 29], and so on.

In this paper, we present template-guided concolic testing, a new technique for adaptively reducing the search space of concolic testing. The key idea is to guide concolic testing with templates, which restrict the input space by selectively generating symbolic variables. Unlike conventional concolic testing that tracks all input values symbolically, our technique treats a set of selected input values as symbolic and fixes unselected inputs with particular concrete inputs, thereby reducing the original search space. A challenge, however, is choosing input values to track symbolically and replacing the remaining inputs with appropriate values. To address this challenge, we develop an algorithm that performs concolic testing while automatically generating, using, and refining templates. The algorithm is based on two key ideas. First, by using the sequential pattern mining [9], we generate the candidate templates from a set of effective test-cases, where the test-cases contribute to improving code coverage and are collected while conventional concolic testing is performed. Second, we use an algorithm that learns effective templates from the candidates during concolic testing. Our algorithm iteratively ranks the candidates based on the effectiveness of templates that were evaluated in the previous runs. Our technique is orthogonal to the existing techniques and can be fruitfully combined with them, in particular with the state-of-the-art search heuristics.

Experimental results show that our approach outperforms conventional concolic testing in term of branch coverage and bug-finding. We have implemented our approach in CREST [7] and compared our technique with conventional concolic testing for open-source C programs of medium size (up to 165K LOC). For all benchmarks, our technique achieves significantly higher branch coverage compared to conventional concolic testing. For example, for vim-5.7, we have performed both techniques for 70 hours, where our technique exclusively covered 883 branches that conventional concolic testing failed to reach. Our technique also succeeded in finding real bugs that can be triggered in the latest versions of three open-source C programs: sed-4.4, grep-3.1 and gawk-4.21.
This paper makes the following contributions:

- We present template-guided concolic testing, a new technique for reducing the input space by selectively generating symbolic values without any prior domain knowledge.
- We present an online learning algorithm to select useful templates from previous runs of concolic testing.
- We extensively compare our technique with conventional concolic testing on open-source C programs. We make our tool, called ConTest, and data publicly available.\(^1\)

## 2 OVERVIEW

In this section, we illustrate our approach with an example.

### 2.1 Motivating Example

Fig. 1 shows a code snippet simplified from tree-1.6.8, where we assume that the body of strncmp is not available. Function f takes as input two arrays of characters, namely input1 and input2, where the size of each array is 4. The program execution is determined by the contents of these arrays. At line 5, Xflag is set to 1 if the first two characters of input1 are ‘-‘ and ‘X’. At line 9, duflag is set to 1 if input2 contains the string ‘--du’. Thus, the error location (line 12) is reachable when the function is executed with the following inputs:

\[
\text{input1: } \alpha_1 \alpha_2 \alpha_3 \alpha_4 \quad \text{input2: } \alpha_5 \alpha_6 \alpha_7 \alpha_8
\]

where * means an arbitrary character. The goal of concolic testing is to generate such inputs that drive program execution to hit the error location.

However, conventional concolic testing is unlikely to trigger the error due to the huge search space. In order to reach the error location, the program execution must hit lines 5 and 9. To do so, concolic testing initially runs the program with random inputs while simultaneously executing the program with the symbolic inputs:

\[
\text{input1: } a_1 \ a_2 \ a_3 \ a_4 \quad \text{input2: } a_5 \ a_6 \ a_7 \ a_8
\]

During the execution, constraints on the symbolic variables \((a_1, \ldots, a_8)\) are collected and used to generate the next input. For example, when the initial execution follows the true branches of the conditional statements at line 4 and the false branches of the statements at lines 7 and 11, the following constraints are collected:

\[
a_4 = \alpha \land a_2 = \alpha \land a_5 \neq \alpha
\]

Negating, for example, the last conjunct will produce input that makes the program execution to exercise the true branch of the first conditional statement at line 7. Then, assuming that the new input does not satisfy the second condition at line 7, the following path condition will be newly generated:

\[
a_3 = \alpha \land a_2 = \alpha \land a_5 = \alpha \land a_6 \neq \alpha
\]

That is, all input values except for the last two are fixed by concrete values, so that concolic testing no longer needlessly attempts to explore execution paths that cannot reach line 8. In other words, our technique is able to enforce the necessary condition to reach the error location, enabling concolic testing to focus on generating the inputs ‘d’ and ‘u’ for \(a_7\) and \(a_8\), respectively. With this template, concolic testing is able to generate the error-triggering input more effectively, up to 9 times faster than the conventional method for the example program.

### 2.2 Template-Guided Concolic Testing with Online Learning

Fig. 2 illustrates our technique for performing concolic testing while automatically generating templates online. Our technique is able to generate effective templates without any prior domain knowledge. The algorithm repeats the following five procedures until a given testing budget is exhausted.

#### 2.2.1 Conventional Concolic Testing

We first perform conventional concolic testing (without template) to generate a set of effective test cases. We say a test case is effective if it enables to exercise previously uncovered branches during concolic testing. We run concolic testing for a certain amount of time and collect effective

---

\(\text{Concolic Testing: https://github.com/kupl/ConTest}\)
Concolic Testing
without template

Sequential Pattern Mining

Pattern Ranking

Pattern to Template

Concolic Testing
with template

Figure 2: Overview of our technique

2.2.2 Sequential Pattern Mining. Once a dataset of effective test cases is collected, we try to capture common patterns in those input vectors. Specifically, we aim to extract a partial sequence of characters that frequently appear in the effective test cases. To do so, we use a recent algorithm for sequential pattern mining [9], which finds out the following four patterns from 40,000 test cases collected during the previous phase:

\[ P_1 : \neg X, \quad P_2 : \neg s, \quad P_3 : \neg X \neg s, \quad P_4 : \neg X \neg s \neg P \]

For example, pattern \( P_1 \) says that effective test cases are likely to involve characters ‘\( \neg \)’, ‘\( X \)’, and ‘\( \neg \)’ in order.

2.2.3 Pattern Ranking. After generating the candidate patterns via sequential pattern mining, we choose the top-\( k \) patterns that are most likely to maximize unique branch coverage; the coverage is calculated as the number of branches that conventional concolic testing has not discovered. In our example, to quickly cover the unique branch (e.g., the true branch at line 8 in Figure 1), pattern \( P_3 \) in Figure 2 is required. However, pinpointing the effective pattern among the candidates is nontrivial, as running the algorithm on real-world programs usually discovers thousands of patterns. Even worse, only a small fraction of the candidate patterns is effective for increasing branch coverage. We address this challenge by ranking candidate patterns based on the effectiveness of similar patterns that were evaluated in the previous runs. We accumulate sets of good and bad patterns during the algorithm and use them to estimate the effectiveness of the newly generated patterns. For the example program, we choose \( P_3 \) and \( P_2 \) when \( k = 2 \).

2.2.4 Pattern to Template. The next step is to transform patterns to templates. Note that a pattern is simply an ordered sequence of meaningful input values (e.g., characters); to be a template, we need to decide the position of each value contained in a given pattern. To do so, we first collect the test-cases containing the pattern and then identify the positions where the template values appear most frequently. For instance, suppose that the concrete value ‘\( X \)’ appeared the most at the second index in the test-cases. Then, we replace the symbolic value \( a_2 \) at the second index in input 2 with the value ‘\( X \)’. By applying this rule to patterns \( P_3 \) and \( P_2 \), which were selected in the previous phase, we obtain the following two templates:

\[ \text{input1: } \neg X \neg s, \quad \text{input2: } \neg X \neg s \neg P \]
\[ \text{input1: } \neg a_2, \quad \text{input2: } \neg a_2 \neg a_3 \neg a_4 \]

In the rest of this paper, we also represent a template by a set of concrete values and their positions. For example, the first template can be represented as follows:

\( \{ (0, \neg \), (1, \( \neg X \)), (4, \neg \), (5, \neg \) \}. \)

2.2.5 Concolic Testing with Template. The final step is to run concolic testing with the generated templates (\( T_1 \) and \( T_2 \)). For example, when using the template \( T_1 \), we only generate four symbolic values (\( a_3, a_4, a_7, a_8 \)) and replace the rest with concrete values in the template \( T_1 \). Note that the concrete values are not arbitrary but are effectively guiding the concolic testing to reach the error location (e.g., true branch at line 11 in Figure 1) by forcing the program execution to follow the specific path, taking all true branches of the conditional statements at lines 4 and 7.

After performing concolic testing with the templates for a certain amount of time, we evaluate the qualities of the generated templates in terms of the number of unique branches. As a result, we classify the corresponding patterns into good and bad patterns in Figure 2, which will be used by the ranking algorithm in the next iteration of the algorithm. As the entire procedure is going on, our algorithm accumulates the evaluation data and therefore the ranking algorithm is able to pick more effective patterns based on the increased knowledge.

3 Template-Guided Concolic Testing

Algorithm 1 presents our template-guided concolic testing. We first describe conventional concolic testing and then explain how to modify it to our algorithm.

3.1 Conventional Concolic Testing

Without line 6, Algorithm 1 becomes conventional concolic testing, which takes a program \( P \) and returns covered branches as well as the set of generated input vectors. At line 2, the sets of covered branches \( B \) and generated input vectors \( V \) are initialized. At line 3, \( v \) denotes the initial concrete input vector, which is assumed to be given for each program. At line 4, the algorithm initializes the symbolic input vector: \( s = (a_1, \ldots, a_{|W|}) \), where each \( a_i \) denotes a fresh symbol representing the \( i \)-th input. At line 7, the program \( P \) is “concolically” executed; \( P \) is executed with the concrete input \( v \) while it is at the same time executed symbolically with \( s \). Once the
Our algorithm differs from conventional concolic testing in that the technique that interleaves conventional and template-guided concolic testing while automatically generating effective input data can achieve the benefit of concolic testing. The main contribution of this paper is to propose a new approach to generate effective input data automatically.

### 3.2 Concolic Testing with Template

Our algorithm differs from conventional concolic testing in that some input values are fixed according to the given template. A template \( T \) is a set of pairs of indices and values:

\[
T = \{(i_1, v_1), \ldots, (i_m, v_m)\}
\]

Intuitively, a pair \((i, v)\) in \( T \) indicates that the \( i \)-th input of \( v \) and \( s \) is fixed by the concrete value \( v \), so that concolic testing should not symbolically track those inputs in \( T \). We assume that for every \((i, v)\) \( T \), \( i \) is unique and \( 0 \leq i < |v| \).

The template is instantiated at line 6. Before running the program, both concrete and symbolic input vectors are modified, where the Instantiate function replaces a given vector \( a \) according to the template \( T \) as follows:

\[
\text{Instantiate}(v, T) = (v_{i_1}, \ldots, v_{i_m})
\]

where \( v_{i_j} \) is the value \( v \) in the template if \((i, v)\) \( T \). Otherwise, if \((i, v)\) \( T \), \( v_{i_j} \) is not changed, i.e., \( v_{i_j} = a_{i_j} \). That is, given a vector \( a \) and a template \( T \), \text{Instantiate}(v, T) \ replaces the \( i \)-th element of \( a \) by the value in \( T \). As a result, concolic execution of \( P \) at line 7 generates constraints only for a subset of the original symbolic variables \((a_{i_1}, \ldots, a_{i_m})\). We assume that the model function at line 15 produces arbitrary values for unconstrained symbols.

Our template-guided concolic testing poses a significant challenge. That is, the effectiveness of our approach depends on the given template \( T \). For example, when \( T = \emptyset \), the algorithm becomes the ordinary concolic testing that tracks all input variables symbolically, which often suffers from the path-explosion problem. On the other hand, when the template is too specific (e.g., \( T = \{(0, v_0), (1, v_1), \ldots, (|v| - 1, v_{|v|-1})\} \) in the extreme), the algorithm becomes more like random testing and is likely to lose the benefit of concolic testing. The main contribution of this paper is the technique that interleaves conventional and template-guided concolic testing in a way that automatically generates effective templates and maximizes the final code coverage in the long run.

### 4 TEMPLATE-GUIDED CONCOLIC TESTING WITH ONLINE LEARNING

In this section, we present our algorithm (Algorithm 2) for performing template-guided concolic testing while automatically generating effective templates online. Algorithm 2 consists of four main stages: conventional concolic testing, sequential pattern mining,
The algorithm has three hyperparameters ($\eta_1$, $\eta_2$, and $\eta_3$). The first parameter $\eta_1$ is used at line 6 and determines the number of conventional concolic executions in the first phase. The second parameter $\eta_2$, which is used at line 23, denotes the number of concolic executions with each template. The last parameter $\eta_3$ represents the threshold value for the pattern $p$ to be a good pattern (i.e., included in the set $Good$). In experiments, we set $\eta_1 = 100$, $\eta_2 = 20$, $\eta_3 = 20$. In this work, we tuned these hyperparameters manually by trial-and-error, and found that the performance of Algorithm 2 depends on them substantially. An interesting future direction would be finding optimal hyperparameters automatically during the algorithm.

### 4.1 Exploration without Templates

The first phase of the algorithm (lines 5–9) is to run concolic testing without template (i.e., $T = \emptyset$) to explore and collect diverse input vectors that are effective in increasing branch coverage.

At line 5, the set $V$ of input vectors is initially empty. At lines 6–9, ConcolicTesting (Algorithm 1) is run for $\eta_1$ times. When concolic testing finishes, the sets $B_i$ and $V_i$ of covered branches and effective input vectors, respectively, are returned. We say input vectors are effective (i.e., effectiveinput at line 9 of Algorithm 1) if they satisfy the following two conditions. First, the input vectors should be able to increase branch coverage after the initial 10% of the budget $N$ for ConcolicTesting is exhausted. For example, when budget $N$ is 4,000 program executions, we ignore inputs generated during the first 400 executions. This is because branch coverage gets easily increased in the early stage of concolic testing, no matter what initial input vectors are used. Second, the input vectors should contribute to discovering branches that are new compared to previous program executions. Collecting effective inputs only is crucial because blindly collecting all inputs can cause serious performance degradation in the next stage, sequential pattern mining.

### 4.2 Mining Patterns

The second step of the algorithm is to mine common patterns from the collected set of effective input vectors (line 12). We observed that each effective input vector is likely to have meaningful subsequences that ultimately contribute to improve branch coverage. The goal of this stage is to quickly extract such subsequences that are common to many of the collected inputs and use them as the candidates to reduce the search space. Fortunately, for this purpose, we can use off-the-shelf techniques called sequential pattern mining in the data mining community, which can do the desired task efficiently. Numerous pattern mining algorithms have been proposed in the literature [9, 15, 27, 30]. We used a state-of-the-art algorithm, CloFast [9], which avoids generating redundant patterns. CloFast also outperforms the existing algorithms in terms of computation time and memory consumption [9]. For example, when CloFast takes 14,604 effective inputs collected from sed-1.17, it generates 6,176 candidate patterns in five minutes. In Algorithm 2, the algorithm is modeled by the SequentialPatternMining function, which takes a set of input vectors and returns a set of common patterns.

### 4.3 Ranking Patterns

The third step is to rank the candidate patterns according to their (estimated) effectiveness (line 15). We designed a ranking function (PatternRanking), which chooses the top-$k$ patterns from the candidates generated by the pattern mining algorithm. In experiments, we set $k = 20$. At line 15 in Algorithm 2, PatternRanking takes three pattern sets: patterns to rank ($Cand$), good patterns ($Good$), and bad patterns ($Bad$). Then, it returns the top-$k$ patterns ($Ranked$) that are most likely to cover new branches in the future.

The key ideas behind our ranking algorithm (Algorithm 3) is to reflect the experience with the patterns evaluated in the previous runs and try as many diverse patterns as possible. Hence, the main loop of the algorithm consists of the two phases: Reflection and Diversification. Initially, we rank the candidates $Cand$ by sorting them based on the frequency of each candidate calculated by the sequential pattern mining algorithm in ascending order. The hypothesis is that the patterns with high frequencies are unlikely to discover new branches. At lines 3–4, $Pat_{top}$ and $Pat_{mid}$ are initially empty vectors, and $Ranked$ is an empty set.

In the first stage (Reflection), we transform each pattern $p$ in $Cand$ into n-grams and check whether any n-grams in the pattern $p$ are included in any patterns in good or bad patterns sets (line 9–13). To do so, we define a function $ngram$ which takes a pattern and returns a set of n-grams for the pattern, where the number $n$
is half of the length \( p \) (\( n = \lceil |p|/2 \rceil \)). For example, when the pattern \( p \) is a string "s/b", \( \text{ngram}(p) \) returns the three 2-grams: "s/", "b/", "/b". Then we classify the patterns using predicate \( \text{Match} \) defined as follows:

\[
\text{Match}(p, P) \iff \exists g \in \text{ngram}(p). g \in \bigcup_{p' \in P} \text{ngram}(p')
\]

\( \text{Match} \) takes a pattern (\( p \)) and a set of patterns (\( P \)), and returns true iff any of the n-grams of \( p \) is included in the union of n-grams of the patterns in \( P \). We perform \( \text{Match} \) with both \( \text{Good} \) and \( \text{Bad} \). When \( \text{Match}(p, \text{Good}) \) and \( \text{Match}(p, \text{Bad}) \) are true and false, respectively, the pattern \( p \) is included in \( \text{Pat}_{\text{top}} \), a class with high priority (line 10). Intuitively, the pattern \( p \) gets high priority if it does not have any of the features having bad patterns \( \text{Bad} \) while the pattern \( p \) contains at least one feature of good patterns. At line 11, when the results of \( \text{Match}(p, \text{Good}) \) and \( \text{Match}(p, \text{Bad}) \) are both false, the pattern \( p \) is appended to \( \text{Pat}_{\text{md}} \), a class having middle priority (line 12). The patterns in the class \( \text{Pat}_{\text{md}} \) do not include at least the features of the bad patterns. Otherwise, the pattern \( p \) is removed from the candidate group of top-\( k \) patterns.

In the second step (\( \text{Diverse} \)), we aim to diversify the patterns by filtering out similar patterns in \( \text{Pat}_{\text{top}} \). To diversify the patterns, we use the following function:

\[
\text{Diverse}(p, P) \iff \text{ngram}(p) \not\subseteq \bigcup_{p' \in P} \text{ngram}(p')
\]

\( \text{Diverse} \) returns true iff \( \text{ngram}(p) \), a set of n-grams generated by the pattern \( p \), is not a subset of the union of all n-gram sets generated by each pattern \( p' \) in the given pattern set \( P \). At line 19, we pop a pattern \( p \). Then, we add the pattern \( p \) into the set \( \text{Ranked} \) only when \( \text{Diverse}(p, \text{Ranked}) \) returns true (line 22). Intuitively, this step makes each pattern in \( \text{Ranked} \) have at least one unique n-gram.

### 4.4 Exploitation with Templates

The last step of the algorithm is to exploit the patterns learned from the previous phase. However, the patterns in \( \text{Ranked} \) cannot be used immediately. Because a pattern is just a sequence of characters, we need to determine the appropriate position of each character. To transform a pattern into a template, the algorithm uses the function \( \text{PatternToTemplate} \), which takes a pattern and a set of input vectors, and creates a template for the pattern. We generate the template in two steps. First, we only collect the input vectors containing the pattern among the input vectors \( V \) accumulated in step 1 of Algorithm 2. Second, for each character in the pattern, we compute the position where the character appears most frequently. The resulting template is used to perform template-guided concolic testing.

At line 19 of Algorithm 2, we first pick a pattern \( p \) with the highest priority from the set \( \text{Ranked} \). Then, we transform the pattern \( p \) into the template \( T \) by using \( \text{PatternToTemplate} \) (line 21). Using the template, we perform \( \text{ConcolicTesting}(P, T) \) for \( n_2 \) times (lines 23-26). As we mentioned before, we set \( n_2 = 20 \), because we experimentally observed that a good template usually was able to cover new branches within 20 trials. Whenever we run \( \text{ConcolicTesting}(P, T) \), we accumulate the branches covered by each template \( T \) in the set \( B_T \). At lines 30-34, we evaluate the quality of the template \( T \) in terms of the number of uniquely covered branches, where the number is counted as the size of the difference set between the \( B_T \) and \( B \) sets. When the number is greater than the threshold (\( \eta_T \)), we add the pattern \( p \) corresponding to the template \( T \) to the set \( \text{Good} \). When the number is less than or equal to one, we add the pattern \( p \) to the set \( \text{Bad} \). Note that to rank the candidate patterns in the next iterations, we only use the patterns, which are definitely determined to be good or bad. The algorithm repeats the procedure until the time budget is exhausted. Then, it returns the total number of covered branches (\( |B \cup T_B| \) (line 37).

As the outer loop of Algorithm 2 is repeated, we gradually accumulate the learned knowledge, namely \( \text{Good} \) and \( \text{Bad} \) sets; the former represents the knowledge for effectively reducing the search space while the latter must be avoided. By iteratively updating these sets, our algorithm guides concolic testing towards maximizing branch coverage.

### 5 EVALUATION

In this section, we experimentally evaluate our approach. We implemented our approach in a tool, called ConTest, on the top of CREST [7], a publicly available concolic testing tool for C programs. We have conducted the experiments to address the following research questions:

- **Effectiveness of our approach**: How well does our approach perform compared to conventional concolic testing?
- **Efficacy of online learning**: Is online learning crucial for generating effective templates?
- **Learned patterns**: What lessons do the learned patterns provide about search space reduction?

#### 5.1 Settings

##### 5.1.1 Benchmarks

We have used 5 open-source C programs in Table 1: vim, gawk, grep, sed and tree. All benchmarks came from the prior works on concolic testing [4, 5, 19] with slight modifications on the annotations for three benchmarks (vim, grep and sed).

<table>
<thead>
<tr>
<th>Program</th>
<th># Total branches</th>
<th>LOC</th>
<th>Input type</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>vim-5.7</td>
<td>35,464</td>
<td>165K</td>
<td>unsigned char</td>
<td>[4]</td>
</tr>
<tr>
<td>gawk-3.0.3</td>
<td>8,038</td>
<td>30K</td>
<td>unsigned char</td>
<td>[5]</td>
</tr>
<tr>
<td>grep-2.2</td>
<td>3,836</td>
<td>15K</td>
<td>char</td>
<td>[4]</td>
</tr>
<tr>
<td>sed-1.17</td>
<td>2,650</td>
<td>9K</td>
<td>unsigned char</td>
<td>[19]</td>
</tr>
<tr>
<td>tree-1.6.0</td>
<td>1,440</td>
<td>4K</td>
<td>char</td>
<td>[5]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Org</th>
<th>Modify</th>
<th>Org</th>
<th>Modify</th>
</tr>
</thead>
<tbody>
<tr>
<td>vim-5.7</td>
<td>11,984</td>
<td>12,900</td>
<td>1,833</td>
<td>1,996</td>
</tr>
<tr>
<td>gawk-3.0.3</td>
<td>7,507</td>
<td>13,526</td>
<td>1,917</td>
<td>2,072</td>
</tr>
<tr>
<td>grep-2.2</td>
<td>11,142</td>
<td>11,842</td>
<td>1,767</td>
<td>1,851</td>
</tr>
<tr>
<td>sed-1.17</td>
<td>6,197</td>
<td>12,174</td>
<td>1,845</td>
<td>1,884</td>
</tr>
</tbody>
</table>
During this work, we found that the performance of concolic testing varies significantly depending on how benchmark programs are annotated, and tried to annotate the programs in ways that maximize the performance of the baseline concolic testing. For example, since vim is a text editor program, it is natural to take inputs of type ‘unsigned char’ (0–255). But the previous version of vim was annotated with the CREST_unsigned_short function, which needlessly generates inputs from the larger space (0–65,535). We replaced it with CREST_unsigned_char. For sed and grep, we also changed the annotations to make them more natural. For example, the original annotations of sed forced concolic testing to execute the program with the option ‘-f’ always turned on. We fixed this issue by symbolizing the arguments of the main function. The modified programs are available with our tool, ConTest.

Table 2 shows that the baseline concolic testing performs much better on the modified programs. We compared the performance of the conventional concolic testing on the original and modified programs with various search heuristics. The table reports the number of branches covered over 100 runs of concolic testing, where a single run consists of 4,000 program executions (i.e., \( N = 4000 \) in Algorithm 1). Overall, the performance of concolic testing is improved significantly with the modifications. For example, concolic testing with the CGS heuristic [23] for vim-5.7 covered 13,526 branches on the modified benchmark while the same method managed to cover 7,507 branches only on the original one. In summary, we modified the benchmark programs to make the baseline concolic testing much stronger.

We did not use the four small programs, which were used in [4, 5, 19, 23]: cdaudio, floppy, kbfl11tr and replace. This is because the conventional concolic testing already achieves high code coverage on those programs, as the sizes of these benchmarks are very small (e.g., replace is of 0.5KLoC).

5.1.2 Search Heuristics. In evaluation, we considered four search heuristics: CGS (Context-Guided Search) [23], CFDS (Control-Flow Directed Search) [4], Random branch search [4] and Gen (Generational search) [12]. We chose them because our technique requires search heuristics to be nondeterministic in order to generate diverse input patterns in the first step of Algorithm 2. We did not use deterministic techniques such as DFS (Depth-First Search) [11] and ParaDySE [5]. For each benchmark program, we applied our technique on top of the search heuristic that performs best. For example, we used CGS for vim and grep, and CFDS for the remaining three programs.

5.1.3 Other Settings. We used the same evaluation settings for both conventional and template-guided concolic testing. First, all experiments were conducted on a machine with two Intel Xeon Processors E5-2630 and 192GB RAM. Second, we performed concolic testing on all the benchmarks, using 10 cores in parallel. Third, the initial input was fixed for each benchmark. For vim, the largest benchmark, we allocated 70 hours for testing budget and 7 hours for the four smaller programs. We set \( N = 4,000 \) in Algorithm 1.

5.2 Effectiveness of Our Approach
We evaluated our technique and conventional concolic testing on 5 benchmarks in terms of branch coverage and bug detection.

5.2.1 Branch Coverage. Figure 3 shows that our approach (T-CGS) increases branch coverage significantly compared to conventional concolic testing on vim-5.7. The CGS heuristic is a robust baseline that covers 806 more branches compared to the Random heuristic, the second best of conventional concolic testing. Nevertheless, T-CGS (our template-guided concolic testing on top of the CGS heuristic) covered 16,197 branches, covering 552 more branches than CGS. More importantly, Table 3 shows that T-CGS exclusively covered 833 branches that CGS fails to reach over 70 hours, using 10 cores in parallel. The results show that our technique enables concolic testing to achieve significant performance gains in practice by effectively reducing the search space.

Figure 4 shows that our approach also achieves higher branch coverage than conventional concolic testing on the remaining 4 benchmarks. For example, ours (T-CGS) covered 2,252 branches for grep, while CGS covered only 2,157 branches during the same time period (7h). Our approach succeeded to cover the branches that conventional testing fails to reach on all benchmarks. Table 3 reports the number of unique branches and trials. The former denotes the number of branches only covered by each approach. For grep and tree, 98 and 80 branches were exclusively covered by T-CGS and T-CFDS, respectively. The latter is the total number of trials that each approach has performed concolic testing during the same time budget for each benchmark; as we mentioned above,
a single trial consists of 4,000 program executions ($N = 4,000$ in Algorithm 1).

Because our approach involves additional runtime overhead (e.g., sequential pattern mining), it is natural for our approach to have fewer runs of concolic testing than conventional approach within the same time budget. Table 3 shows that the number of trials by baseline is usually greater than the number of trials by our template-guided concolic testing. For example, for vim, the largest program in our benchmarks, the baseline (CGS) ran concolic testing 2,496 times for 70 hours, while our technique (T-CGS) performed it 2,054 times. One interesting point is that for grep, the number of trials for our technique is greater than that for conventional concolic testing. This is because the benefit of reducing the search space (e.g., constraint solving time) in grep is greater than the overhead (e.g., pattern mining time) caused by our approach.

5.2.2 Bug Finding. During experiments, we have found five bugs in sed, grep, and gawk, which are exploitable even in the latest versions of the programs. Table 4 shows the bug-triggering inputs and phenomenons when the programs are executed with the inputs.

Table 4: Bugs in benchmarks

<table>
<thead>
<tr>
<th>Phenomenons</th>
<th>Bug-Triggering Inputs</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>sed Memory Exhaustion</td>
<td>$H$</td>
<td>4.4(latest)</td>
</tr>
<tr>
<td>sed Infinite File Write</td>
<td>$w$</td>
<td>4.4(latest)</td>
</tr>
<tr>
<td>grep Segmentation Fault</td>
<td>$({1}1+)^+$$</td>
<td>3.1(latest)</td>
</tr>
<tr>
<td>grep Non-Terminating</td>
<td>$({1}1+)^+$$ {8957}'</td>
<td>3.1(latest)</td>
</tr>
<tr>
<td>gawk Memory Exhaustion</td>
<td>$6672467e2=E7$</td>
<td>4.21(latest)</td>
</tr>
</tbody>
</table>

The two error-triggering inputs for sed could consume all of our Linux machine’s memory and hard disk, respectively. The template used for generating the former input is as follows: \{(1, \'\n\'), (3, \'\n\'), (5, \'D\'), (6, \'\n\')\}. The template guides concolic testing to find the bug effectively by concretizing 4 of the 6 characters required to
in concolic testing [2, 4, 5, 10, 12, 13, 17, 23, 28, 29], we discuss two
main approaches that are closely related to our approach: search

5.5 Threats to Validity

- Benchmarks: We used 5 benchmark programs which were widely used from prior work on concolic testing [4, 5, 19]. However, the benchmarks, accepting strings as input, may not be sufficient to evaluate the performance of our technique and conventional concolic testing in general.
- A budget for ConcolicTesting: We set $N$ in Algorithm 1 to 4,000, the same value used in prior work [4, 5, 19]. However, the performance of our technique and conventional concolic testing may vary depending on the value.

6 RELATED WORK

Among existing works on mitigating the path-explosion problem in concolic testing [2, 4, 5, 10, 12, 13, 17, 23, 28, 29], we discuss two main approaches that are closely related to our approach: search
heuristics and search-space reduction. We also discuss recent works that improve software testing with learning [5, 14, 20, 21, 24, 26].

**Search Heuristics.** Our technique is orthogonal to the existing works for search heuristics [4, 5, 12, 23, 29]. To achieve the goal of maximizing code coverage, search heuristics focus on selecting one of the candidate branches in the path, whereas our technique reduces the number of the candidates by using template. A heuristic selects the branches that are most likely to maximize code coverage according to its own criterion. For example, the CFDS heuristic [4] selects a branch closest to any of uncovered branches nearby the current execution path. The CGS heuristic [23] selects a branch by performing the breath-first search on execution tree while excluding branches with the same “contexts” from the branch selection. The context of each branch is calculated as a sequence of preceding branches. The Generational heuristic [12] first selects all the branches once in the current path, and measures the coverage gain for each branch selection. Then, the heuristic selects the branch with the highest gain as the next-generation branch. Our technique can be used in combination with these search heuristics.

**Search-Space Reduction.** Our work can be seen as a new approach for reducing the search space [2, 3, 17, 28]. DASE (Document Assisted Symbolic Execution) [28] is a technique that allows symbolic execution to exercise core functionalities of the program by extracting input constraints from program documents (e.g., manual pages). Our technique is different from DASE as we do not require any prior domain knowledge (i.e., documents). Jaffar et al. [17] aim to prune the execution paths guaranteed to not trigger a bug by using interpolation. Boonstoppel et al. [2] proposed the technique, read-write set analysis, for pruning the number of execution paths that produce the same effects. Bugrara et al. [3] introduced the technique to discard the paths that are similar to previously executed paths. Our technique differs from these works in that we apply online learning to adaptively reduce the search space of concolic testing.

**Learning-based Software Testing.** At a high-level, our work belongs to the techniques that combine software testing with machine learning [5, 6, 14, 20, 24, 26]. Learn&Fuzz [14] aims to generate input grammars (e.g., PDF object) for fuzzing by using character-level recurrent neural networks. Skyfire [26] aims to learn a probabilistic context-sensitive grammar from the existing samples to generate seed inputs for fuzzing. QBE [20] learns the kinds of GUI actions to detect crashes or increase activity coverage in Android GUI testing via Q-learning. RETECs [24] employs reinforcement learning to automatically prioritize test cases that are likely to detect bugs in Continuous Integration (CI). Lastly, ParaDySE [5] aims to automatically learn search heuristics for concolic testing. In this work, we use online learning to select good templates, effectively reducing the search space of concolic testing.

**7 CONCLUSION**

Coping with the path-explosion problem continues to be the long-standing challenge in concolic testing. In this paper, we presented a new approach, which mitigates the path-explosion problem by reducing the search space using templates. In our approach, concolic testing uses a set of templates to exploit common input patterns that improve coverage effectively, where the templates are automatically generated through online learning algorithm based on the feedback from past runs of concolic testing. Experimental results demonstrate that our template-guided concolic testing with online learning outperforms conventional concolic testing significantly in both branch coverage and bug-finding.

**ACKNOWLEDGMENTS**

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B2014062).

**REFERENCES**


[27] Jianyong Wang, Jiawei Han, and Chun Li. 2007. Frequent closed sequence mining without candidate maintenance. *IEEE Transactions on Knowledge and Data Engineering* 19, 8 (2007), 1042–1056.


