Invasive Software Testing: Mutating Target Programs to Diversify Test Exploration for High Test Coverage

Yunho Kim*, Shin Hong†, Bongseok Ko*, Loc Duy Phan*, Moonzoo Kim*
*School of Computing, KAIST, South Korea
yunho.kim03@gmail.com, bsko5006@gmail.com, duyloc_1503@kaist.ac.kr, moonzoo@cs.kaist.ac.kr
†School of CSEE, Handong Global University, South Korea
hongshin@handong.edu

Abstract—Software testing techniques have advanced significantly over several decades; however, most of current techniques still test a target program as it is, and fail to utilize valuable information of diverse test executions on many variants of the original program in test generation.

This paper proposes a new direction for software testing – Invasive Software Testing (IST). IST first generates a set of target program variants \( m_1, \ldots, m_n \) from an original target program \( p \) by applying mutation operations \( \mu_1, \ldots, \mu_n \). Second, given a test suite \( T \), IST executes \( m_1, \ldots, m_n \) with \( T \) and records the test runs which increase test coverage compared to \( p \) with \( T \). Based on the recorded information, IST generates guideposts for automated test generation on \( p \) toward high test coverage. Finally, IST generates test inputs on \( p \) with the guideposts and achieves higher test coverage. We developed DE\textsc{Miner} which implements IST for C programs through software mutation and concolic testing. Further, we showed the effectiveness of DE\textsc{Miner} on three real-world target programs Busybox-1s, Busybox-printf, and GNU-find. The experiment results show that the amount of the improved branch coverage by DE\textsc{Miner} is 24.7% relatively larger than those of the conventional concolic testing techniques on average.

Index Terms—automated test generation, concolic testing, mutation analysis, test coverage

I. INTRODUCTION

Software testing has been a de-facto standard method to assess and improve software quality. Although software testing techniques have advanced significantly over several decades, most of them (either manual or automated ones) still test a target program as it is without modifying the target program. In contrast, other engineering disciplines such as mechanical engineering have used invasive testing (or destructive testing) as a standard technique to obtain information on a target object (i.e., analyzing a target object by inducing physical forces on it) which cannot be provided by a non-invasive testing.

Note that a software program has the following ideal characteristics for invasive testing:

- it is almost free to make copies of a software program.
- it is easy to modify/destroy a software program (in a source code or binary level).

However, owing to high complexity of software, it is non-trivial to re-interpret/re-analyze various dynamic information obtained from many modified variants of the original target program (e.g., what conclusion can we make on an original program \( p \) by observing crashes on a variant \( m_1 \) of \( p \) with a test case \( t_1 \)? For example, crash of \( m_1 \) with \( t_1 \) does not necessarily mean that \( p \) will crash with \( t_1 \)). Consequently, software testing so far has failed to improve testing effectiveness further by utilizing useful information from diverse test executions on many variants of a target program.

To resolve the aforementioned problem, this paper proposes a new direction for software testing: Invasive Software Testing (IST). This technique consists of the following three stages:

1) extract/learn useful information of an original target program \( p \) for high test coverage through the diverse test exploration of various variants of \( p \)
2) create guideposts by using the extracted information, which can guide test generation for high test coverage
3) generate test executions on \( p \) following/satisfying the guideposts inserted in \( p \)

A core idea of IST is to create guideposts by learning knowledge on an original target program \( p \) from diverse test executions of many mutated versions \( m_1, \ldots, m_n \) of \( p \), which can effectively lead test generation to cover various executions of \( p \) that achieve high test coverage.

We developed DE\textsc{Miner} (gu\textsc{i}D\textsc{E}ed test generation using Mutat\textsc{Ion} Explor\textsc{ation}) which realizes IST to improve test coverage. DE\textsc{Miner} operates as follows:

1) DE\textsc{Miner} generates target program variants \( m_1, \ldots, m_n \) from an original target program \( p \) by applying mutation \( \mu_1, \ldots, \mu_n \), respectively.
2) given a test suite \( T \), DE\textsc{Miner} executes \( m_1, \ldots, m_n \) with \( T \) and records test runs covering new lines not covered on \( p \) with \( T \) and corresponding mutation operator instances.
3) Based on the recorded information, DE\textsc{Miner} constructs guideposts to guide test generation on \( p \) to improve test coverage.
4) DE\textsc{Miner} applies concolic testing on \( p \) with an inserted guidepost to generate new test executions that follow the guideposts and achieve high test coverage.
The contributions of this paper are as follows:

1) As far as the authors know, this paper is first to technically proposes invasive software testing to explicitly generate highly effective test inputs.

2) We developed DEMINER which realizes IST by generating diverse test executions guided by guideposts created by learning coverage-increasing conditions from various mutant executions.

3) We performed case studies to demonstrate that DEMINER can increase branch and line coverage compared to the conventional concolic testing through three real-world C programs (i.e., Busybox-\(\text{ls}\), Busybox-\(\text{printf}\), and GNU-find). The experiment results show that the amount of the improved branch coverage by DEMINER is 24.7% relatively larger than those of the conventional concolic testing techniques on average. (Section V-D).

II. MOTIVATING EXAMPLE

Although automated test generation techniques such as concolic testing [1]–[3] generate test inputs achieving high test coverage, they sometimes fail to cover target branches due to several limitations of the techniques (e.g., external binary library APIs [4]–[6], symbolic pointers [7]–[9], loop conditions with symbolic bound variables [10]–[12]).

For example, of \(\text{get\_max}\) in Figure 2, concolic testing (almost) fails to cover the branch consisting of Lines 9–10. Figure 2 shows \(\text{get\_max}\) which receives an array of integers \(a\) and an unsigned integer \(sz\) that represents the number of valid target elements in \(a\). Suppose that concolic testing declares every element of \(a\) and \(sz\) as symbolic values, and uses DFS (Depth First Search) as a concolic search strategy.

Suppose that an initial input for \(\text{get\_max}\) has

- \(sz=100\), and
- \(a\) is sorted in a strictly ascending order, and
- every element of \(a\) is positive

Then, the symbolic path formula \(\phi_1\) obtained from an execution with the initial input is as follows:

\[
\phi_1 = (0 < sz) \land (0 < a[0]) \land (1 < sz) \land (a[0] < a[1]) \land \ldots \land (99 < sz) \land (a[98] < a[99]) \land (100 < sz) \land (a[99] > 0)
\]

Note that the subsequent concolic executions (almost) fail to cover Lines 9–10 for the following reason.

After the initial execution, concolic testing negates the last branch condition (i.e., \(a[99] > 0\)) and the resulting symbolic path constraint \(\phi'_1\) is unsatisfiable. This is because \(a[0]\) should be positive (i.e., \(0 < a[0]\) in \(\phi'_1\)) and \(a\) is sorted in a strictly ascending order (i.e., \(a[0] < a[1] < \ldots < a[99]\) in \(\phi'_1\)). Subsequently, concolic testing negates second last condition in \(\phi_1\) (i.e.,\(100 < sz\)) and generates the second input with \(sz=101\), which still does not cover Lines 9–10. The symbolic path formula \(\phi_2\) obtained from the second input is longer than \(\phi_1\) by iterating the for-loop (Lines 3–5) one more time with \(i = 100\) as follows:

\[
\phi_2 = (0 < sz) \land (0 < a[0]) \land (1 < sz) \land (a[0] < a[1]) \land \ldots \land (100 < sz) \land (a[99] < a[100]) \land (101 < sz) \land (a[100] > 0)
\]

Figure 1 shows an overview of DEMINER. Suppose that nodes \(l_4\) and \(l_6\) in the leftmost box indicate uncovered lines in a target program \(p\) with a test case \(t \in T\). Also, suppose that DEMINER generates a mutant \(m_1\) from \(p\) via mutation \(\mu_1\) (i.e., replace \(l_1\) with \(l'_1\)), and \(t\) covers \(l_6\) on \(m_1\). From this mutant execution, DEMINER captures a partial state \(c_1\) infected by \(\mu_1\) as a key for \(t\) to covering \(l_6\), and then DEMINER inserts guidepost \(\phi_1\) right before the mutation site of \(\mu_1\) in \(p\).

Finally, DEMINER automatically generates new tests on \(p\) (e.g., \(t_1\)) with the inserted guidepost which follow/satisfy the guidepost condition \(c_1\). DEMINER repeats this process for all other coverage-increasing mutants each of which increases line coverage by covering lines that were not covered on \(p\) with \(t\).

We studied the effectiveness of DEMINER on three real-world target programs Busybox-\(\text{ls}\), Busybox-\(\text{printf}\), and GNU-find. The experiment results show that the amount of the improved branch coverage by DEMINER is 24.7% relatively larger than those of the conventional concolic testing techniques on average. (Section V-D).

The remainder of this paper is organized as follows. Section II shows a motivating example showing a limitation of a current automated test generation technique and how IST can overcome it. Section III explains DEMINER, and Section IV describes research questions and the experiment setup to study the effectiveness of DEMINER. Sections V and VI report and discuss the experiment results. Finally, Section VII concludes this paper with future work.
Similarly, concolic testing keeps increasing the loop bound $sz$ and generates a large number of test inputs but fails to cover Lines 9–10.

In contrast, DeMINer can cover Lines 9–10 by generating a guidepost by learning from mutant executions as follows. Suppose that DeMINer generates a mutant of get_max (saying $m_3$) that replaces the loop condition at Line 3 (i.e., $i < sz$) with $i == sz$. The execution of $m_3$ with the initial input does not enter the loop and covers Lines 8–9. Then, DeMINer learns from this mutant execution and generates a guidepost guidepost(0==sz) between Line 2 and Line 3 of get_max (see Section III-C for the detail of guidepost construction).

Since \texttt{guidepost($c$)} is a macro of \texttt{if (!c) exit(0);}, the initial execution terminates at the guidepost because the execution does not satisfy the guidepost condition (i.e., 0==sz). After that, the concolic testing generates a next test input which has sz==0 by solving the symbolic path constraint obtained by negating the last branch condition (i.e., the guidepost condition 0==sz). Finally, get_max reaches Lines 8–9 with this test input generated with the guide of the guidepost.

### III. DeMINer Framework

#### A. Overview

DeMINer employs mutation to generate diverse variants/mutants $m_1, ..., m_n$ of a target program $p$ to generate various mutant executions to reach corner-case unreached statements of $p$. By using the information on mutant executions that reach new lines, DeMINer infers a precondition to cover the unreached lines. Then, it feeds these program conditions in the form of a guidepost to concolic testing to guide symbolic executions to cover these new lines in $p$.

We conjecture that, even with limited test inputs, program mutation can effectively diversify program executions by applying various mutation operations at different execution points, because there exists a large set of mutation operators that induce diverse program changes at various program locations. The generation of mutant executions is scalable, because mutant generation does not require sophisticated semantic analysis and it can be parallelized over a large number of computing nodes.

Figure 3 describes the DeMINer process generating new test inputs $T'$. Initially, DeMINer takes source code of an original target program $p$, and a set of test inputs $T = \{t_1, t_2, ..., t_k\}$ as inputs. Then, DeMINer operates in the following three phases:

1) **Discovery of coverage-increasing mutants**

DeMINer generates and runs various mutants of a target program with $T$ so that some mutant executions cover unreached lines of $p$ as the mutation turns a program state to a new one leading to the unreached lines by chance.

2) **Guidepost construction**

Based on the mutant executions that cover unreached lines, DeMINer infers program conditions of the executions covering the unreached lines as guideposts. Then, DeMINer generates multiple copies of $p$ each of which has one guidepost.

3) **Guided test generation**

DeMINer runs concolic testing on the copies of $p$ with the guideposts to generate test executions that follow/satisfy the guideposts to achieve the unreached lines covered at Phase 1.

The remainder of this section describes each phase in detail.

#### B. Phase 1. Discovery of coverage-increasing mutants

This phase aims at finding mutants whose executions cover some code lines that are not covered by running the original program $p$ with a set of test input $T$. DeMINer constructs mutants $m_1, m_2, ..., m_n$ by mutating expressions \( \epsilon_1, \epsilon_2, ..., \epsilon_n \) in $p$ respectively, and then runs the mutants with $T$.

As the first step, DeMINer runs $p$ with $T$ to measure baseline coverage $C_p$ of $p$ with $T$. Then, total 52 mutation operators (see Table I) are applied to every mutation point of $p$ to generate mutants. DeMINer generates mutants at line $l$ only if $l$ is reached by at least one test input in $T$.

Table I shows names and categories of the mutation operators [13] used by DeMINer. DeMINer uses only expression-level mutation operators because DeMINer focuses on a single expression change that increases coverage. DeMINer does not use statement-level mutation operators (e.g., SSDL(statement deletion), SBRC(replacement of \texttt{break} with \texttt{return})). This is because they may change evaluations of multiple expressions/variables at the same time, which makes monitoring and formulating mutation effect as a guidepost difficult. Also, DeMINer does not employ mutation operators on pointer dereference or pointer arithmetics. This is because a corresponding guidepost condition will be an expression on a pointer variable but concolic testing may not generate test inputs to satisfy such guidepost condition (i.e., concolic testing tools do not support a general symbolic pointer).

<table>
<thead>
<tr>
<th>Category</th>
<th>Mutation operator names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change a value or constant</td>
<td>CRCR, VTWD</td>
</tr>
<tr>
<td>Change a variable or memory access</td>
<td>VGAR, VLAR, VGSR, VLSR, VSCR</td>
</tr>
<tr>
<td></td>
<td>OAAA, OAAN, OABA, OABN, OAEEA, OALN, OARN, OASA, OASN, OBAA, OBAN, OBBA, OBBN, OBEA, OBLN, OBNG, OBRN, OBSA, OBSN, OEEA, OESA</td>
</tr>
<tr>
<td>Change an operator in an expression</td>
<td>OEBR, OESR, OLRN, OLNR, OLSN, ORAN, ORBN, ORLN, ORRN, ORSN, OSAA, OSAN, OSBA, OSBN, OSEA, OSLN, OSRN, OSSA, OSSN, OIPM</td>
</tr>
<tr>
<td>Change a branching condition</td>
<td>OCNG, OCOR</td>
</tr>
</tbody>
</table>

### TABLE I

**Mutation operators used by DeMINer**
DeMINER runs each mutant $m_i$ with test inputs $T$ to measure $C_{m_i}$ (i.e., lines of $m_i$ covered by $T$) \(^1\). After the mutant executions, it collects all mutants whose executions cover at least one unreached line as a set of coverage-increasing mutants $M_{All} = \{m_i | C_{m_i} - C_p \neq \emptyset\}$.

Finally, DeMINER selects a subset of the coverage-increasing mutants $M \subseteq M_{All}$ that are passed to the next phase (see Section III-C). DeMINER tries to select $M$ as a minimal set of the coverage-increasing mutants which covers the same set of the unreached lines covered by $M_{All}$. We found that many coverage-increasing mutants redundantly cover the same set of unreached lines. Thus, we believe that this mutant selection method reduces the runtime cost of the subsequent analyses while not hurting testing effectiveness much.

The mutant selection is made by a greedy heuristic algorithm, which initially defines $M$ and $C_M$ as empty sets. $M$ holds selected mutants and $C_M$ contains the unreached lines covered by the mutants in $M$. After initialization, the algorithm selects a mutant $m$ in $M_{All} - M$ that covers the most unreached lines, and then updates $M$ and $C_M$ by including $m$, correspondingly (i.e., $M \leftarrow M \cup \{m\}$ and $C_M \leftarrow C_M \cup C_m$). If ties exist, the algorithm randomly picks one of them. The selection continues until the set of the unreached lines covered by the mutants in $M$ is equal to that of $M_{All}$ (i.e., $C_M - C_p = C_{M_{All}} - C_p$).

### C. Phase 2. Guidepost construction

From the executions of the coverage-increasing mutants obtained from Phase 1, DeMINER infers a precondition at a program location to cover the unreached lines. DeMINER expresses such a precondition as a *guidepost* encoded as an if-statement that continues the execution if the condition is satisfied, or terminates the execution otherwise (i.e., \(\text{guidepost}(\text{exp}) \equiv \text{if(!exp) exit(0);}\)). A guidepost embeds the knowledge on the coverage-increasing executions of the mutants. Note that a guidepost prunes executions without changing the behaviors of a target program. Thus, a guidepost guides concolic testing on a target program $p$ to generate tests toward the observed coverage-increasing executions.

To infer guideposts from the selected mutants $M = \{m_1, m_2, \ldots, m_l\}$, DeMINER first re-runs each mutant $m_i \in M$ to inspect the mutation effects (i.e., infection) to cover the unreached lines. Since $m_i$ has a mutation on a single expression $e_i$, we suspect that a cause of the coverage increase is the evaluation of $e_i$ to a different value.

For each $m_i$, DeMINER identifies all runtime evaluations of the mutated expression as *coverage-increasing values* $V_i = \{v^1_i, v^2_i, \ldots, v^u_i\}$. To extract $V_i$ from the executions of $m_i$, DeMINER instruments $m_i$ by inserting a probe exporting evaluation results of the mutated expression.

Once the coverage-increasing values of $m_i$ are captured as $V_i$, DeMINER constructs guideposts in $p$. Each guidepost is inserted immediately before the mutation site of $m_i$ (i.e., $e_i$) such that a guidepost executes immediately before $e_i$ is evaluated. A guidepost checks if $e_i$ is evaluated to one of the values in $V_i$. DeMINER constructs two types of guideposts from $V_i$ as follows:

- **Single-value guidepost**
  
  For each $v^j_i \in V_i$, a single-value guidepost is created to check if $e_i$ is evaluated to $v^j_i$ for the first time. This condition is encoded as \(\text{guidepost}(e_i == v^j_i)\) for $v^j_i \in V_i$.

- **Multi-value guidepost**
  
  For $m_i$ with multiple coverage-increasing values (i.e., $|V_i| > 1$), DeMINER additionally creates a multi-value guidepost that checks $e_i$ is always evaluated to one of the coverage-increasing values in $V_i$. Thus, the condition of a multi-value guidepost is formed as \(\text{guidepost}(\text{or}(e_i == v^1_i, \ldots, e_i == v^u_i))\). A multi-value guidepost allows a target expression to have multiple value choices when it is evaluated multiple times (e.g., in a loop).

Figure 4 illustrates how DeMINER generates guideposts from monitoring mutant executions and inserts them to a target program $p$. Suppose that DeMINER created a mutant $m$ from $p$ by changing an operator at Line 1 (i.e., $e = x + y$). With a test case $t$ that executes $\text{func}(0,1)$, Line 3 was not covered on $p$, but covered on $m$ as $z$ is -1 on $m$. Once DeMINER finds that $m$ covers an unreached Line 3, it inserts a probe to $m$ to monitor values of the mutated expression (i.e., $x - y$) (Figure 4-(c)). By re-running $m$ with a monitoring probe, DeMINER finds that the mutated expression is evaluated to -1 ($v_1 = -1$) and expects that an execution of $p$ may cover Line 3 if $e$ is evaluated to -1. To reproduce the mutation effect, DeMINER inserts a single-value guidepost to constraint $x + y$ (i.e., $e$) to become -1 (i.e., $v_1$) right before $e$.
IV. Experiment Setup

We have designed the following four research questions to evaluate DeMINER in terms of increased coverage and execution time.

- **RQ1.** With given test inputs, how many unreached lines/branches of an original program are covered by mutant executions?

- **RQ2.** How many unreached lines/branches of an original program are covered by DeMINER?

- **RQ3.** How many unreached lines/branches covered by the mutant executions are also covered by DeMINER?

- **RQ4.** How many unreached lines/branches of an original program does DeMINER cover, compared to conventional concolic testing techniques?

- **RQ5.** To what extent does the mutant selection of DeMINER affect execution time and line/branch coverage?

Suppose that there is another test case \( t' \) that executes \( \text{func}(1, 1) \). Then, DeMINER will find 0 as the second coverage-increasing value for \( c \) (i.e., \( v_2 = 0 \)). From the two coverage-increasing values associated, DeMINER generates a multi-value guidepost that constraints \( x + y \) to become either -1 or 0 (Figure 4-(e)).

For each guidepost \( g \), DeMINER generates a version of \( p \) which embeds \( g \).

**D. Phase 3. Guided concolic testing**

The last phase takes multiple versions of \( p \) each of which is augmented with a guidepost and runs concolic testing to generate test inputs. For the multiple versions of \( p \) with guideposts (e.g., \( g_1, \ldots, g_k \)), DeMINER applies concolic testing to \( p_i \) (\( p \) augmented with \( g_i \)) with each test input in \( T \) as an initial test input. DeMINER uses a new prioritized concolic search strategy which hybridizes depth-first search (DFS) and random branch negation (RND) strategies as follows:

- The search algorithm first performs concolic testing using DFS until the inserted guidepost is reached. If the guidepost condition \( c \) is violated, the target program execution immediately terminates. Then, the search algorithm negates the last branch condition (i.e., unsatisfied guidepost condition \( \neg c \)) and generates a test execution that satisfies \( c \). If concolic testing fails to generate a test execution satisfying \( c \), the concolic testing continues to use DFS until reaching the guidepost again through a different test execution.

- If concolic testing generates a test execution satisfying \( c \), the concolic testing uses RND to negate only those branches executed after the guidepost. This is to focus on execution space satisfying a guidepost condition \( c \) and, thus, has high probability to increase coverage.

**Fig. 4. Example of guidepost construction**

\[
\text{func}(x, y) \{
\begin{align*}
1: & \quad z = x + y; \\
2: & \quad \text{if } (z \leq 0) \\
3: & \quad \text{print}(x); \\
\ldots
\end{align*}
\]

(a) target program \( p \)

\[
\text{func}(x, y) \{
\begin{align*}
1: & \quad z = x - y; \\
2: & \quad \text{if } (z \leq 0) \\
3: & \quad \text{print}(x); \\
\ldots
\end{align*}
\]

(b) a mutant \( m \)

\[
\text{func}(x, y) \{
\begin{align*}
1: & \quad z = \text{probe}(x - y); \\
2: & \quad \text{if } (z \leq 0) \\
3: & \quad \text{print}(x); \\
\ldots
\end{align*}
\]

(c) \( m \) with a monitoring probe

\[
\text{func}(x, y) \{
\begin{align*}
1: & \quad z = x + y; \\
2: & \quad \text{if } (z \leq 0) \\
3: & \quad \text{print}(x); \\
\ldots
\end{align*}
\]

(d) \( p \) with a single-value guidepost

\[
\text{func}(x, y) \{
\begin{align*}
1: & \quad g(x + y = -1) || (x + y = 0); \\
2: & \quad z = x + y; \\
3: & \quad \text{print}(x); \\
\ldots
\end{align*}
\]

(e) \( p \) with a multi-value guidepost

**A. Study Objects**

We used recent versions of three well-known, real-world C programs as study objects.

- **Busybox-ls** is a file listing utility and
- **Busybox-printf** is a formatted data printer in BusyBox version 1.24.0.
- **GNU-find** is a file search utility in GNU FindUtils version 4.6. These three programs are utilities for UNIX-like operating systems. Table II shows the size of the target code in executable lines (LoC) and branches, a

---

2https://busybox.net
3https://www.gnu.org/software/findutils
number of given test cases used, and line and branch coverage achieved by running the all given test cases for each study object.

All test cases were obtained from the regression test suites in the program packages. The experiments use all test cases given in the package, except 11 test cases of GNU-find due to technical difficulties. We used gcov to measure LoC, and the line and branch coverage throughout the experiments.

B. Mutant Generation Setup

DEMINER generates mutants of a target program using a C source code mutation tool MUSIC [15] (see Section IV-D). After mutant generation, DEMINER eliminate trivially equivalent and duplicated mutants [16]. An equivalent mutant is identified by checking whether or not the MD5 checksum of the compiled binary object is the same as that of the original target program. Similarly, two mutants are identified as duplicated if their compiled binary objects have the same MD5 checksum value.

The experiments used all generated mutants for Busybox-ls and Busybox-printf. To save experiment time, the experiments with GNU-find randomly select and use five mutants per code line, because a total amount of time spent for the experiments on GNU-find will be significantly larger than that of Busybox-ls or Busybox-printf.

C. Test Generation Setup

We declared command-line arguments and file-metadata such as file mode, file size, permission, modification time as symbolic inputs for concolic testing. Table III shows the symbolic input setup for the study objects. The second column shows the maximum number of symbolic command-line arguments, and the third column shows the maximum length of each symbolic command-line argument of the experiment setup. A number of symbolic file-metadata structures is same to a number of files used in a given regression test case. The last column shows the maximum number of symbolic file-metadata structures in the experiments. Each symbolic file-metadata consists of 13 symbolic integer variables (Busybox-printf does not use symbolic file-metadata because it does not take a file as an input).

For each pair of a generated guidepost \( g_i \) and a given test case \( t_j \), DEMINER applies concolic testing to a target program having \( g_i \) with \( t_j \) as an initial seed test case for generating 500 test cases further.

In addition, we compared DEMINER with the conventional concolic testing techniques with three search strategies DFS, RND, and CFG. They also use the given test cases as initial seed test cases. For fair comparison, we run each of the three concolic testing techniques for the same amount of time that DEMINER consumes which includes the following:

1) mutant generation and selection
2) mutant executions with the given test cases
3) guidepost constructions
4) guided concolic testing for 500 test cases per guidepost and initial test case.

For example of Busybox-printf, as DEMINER spent total 23,394 seconds, each of the three conventional concolic testing techniques is executed for 23,394 seconds with the 17 initial test cases (i.e., for each initial test case, a conventional technique is executed for 1,376 seconds (=23494/17)).

D. Implementation

The DEMINER implementation is written in C++ and Python. The component for mutation analyses (i.e., in Phase 1, see Section III-B) consists of the mutant generation part and the mutant execution part. For mutant generation, we used MUSIC (MUtation analySIs tool with high Confiurability and extensibility) [15]. MUSIC implements 73 expression-level and statement-level mutation operators for modern C programs (63 of them are defined in Agrawal et al. [13]). MUSIC preserves the source code line numbering in an expression-level mutation to make coverage information on mutants and the original program comparable. The mutant execution part is implemented in 540 lines of Python script code.

The component for guidepost construction (i.e., Phase 2) is implemented in 1,620 lines of C++ code using Clang/LLVM 3.4 [19]. The component for guided concolic testing (i.e., Phase 3) is implemented upon CROWN [20]. CROWN (Concolic testing for Real-world softWare aNalysis) is a lightweight easy-to-customize concolic testing tool for real-world C programs, which is extended from CREST-BV [21]. It supports complex C features such as bitwise operators, floating

---

**TABLE II**

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Branches</th>
<th>Num. TCs</th>
<th>Covered lines</th>
<th>Covered branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busybox-ls</td>
<td>404</td>
<td>303</td>
<td>6</td>
<td>257 (63.6%)</td>
<td>135 (44.6%)</td>
</tr>
<tr>
<td>Busybox-printf</td>
<td>169</td>
<td>105</td>
<td>17</td>
<td>140 (82.8%)</td>
<td>75 (71.4%)</td>
</tr>
<tr>
<td>GNU-find</td>
<td>3616</td>
<td>2091</td>
<td>120</td>
<td>2192 (60.6%)</td>
<td>1061 (50.7%)</td>
</tr>
</tbody>
</table>

**TABLE III**

<table>
<thead>
<tr>
<th>Program</th>
<th>Max. # sym-args</th>
<th>Max. len. sym-args</th>
<th>Max. # sym. file-metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busybox-ls</td>
<td>4</td>
<td>6</td>
<td>5 × 13</td>
</tr>
<tr>
<td>Busybox-printf</td>
<td>7</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>GNU-find</td>
<td>6</td>
<td>10</td>
<td>6 × 13</td>
</tr>
</tbody>
</table>
point arithmetic, bitfields and so on. We added 230 lines of C/C++ code to implement the prioritized search strategy which hybridizes DFS and RND (see Section III-D).

E. Data Collection

All the experiments were performed on machines equipped with Intel quad-core i5 4670K and 8GB ram, running Ubuntu 16.04.3 64 bit version. For each mutant execution, we setup timeout as 0.5 seconds which is almost 10 times of the average execution time of each regression test execution time.

To limit random effects of the greedy coverage-increasing mutant selection and the RND and CFG concolic search strategies, we repeated the same experiment five times and report the average of the results.

We used gcov to measure line and branch coverage. Since MUSIC performs line-preserving mutation (Section IV-D) and DE Miner uses only expression level mutation operators (Section III-B), covered line/branch information of a generated mutant is comparable to that of an original target program.

F. Threats to Validity

A primary threat to external validity is the representativeness of the study objects used for the experiments, because we have examined only three C programs. We believe that this threat is limited because the target programs are non-trivial real-world C programs which have different characteristics. We will address this threat to external validity in future work by applying DE Miner to more target programs. Another external threat involves the representativeness of the conventional concolic testing techniques that we compared with DE Miner (DE Miner might yield different results compared to concolic testing techniques other than DFS, RND, and CFG). We think that this threat is limited because these three search strategies are representative ones for concolic testing.

A primary threat to internal validity is possible bugs in the implementation of DE Miner. Since we have spent significant effort for testing and debugging the implementation, we believe that this threat is limited.

V. EXPERIMENT RESULTS

This sections describes the results of the experiments to answer the research questions in Section IV. Note that, in the following discussion, we denote the sets of lines/branches of an original program as \( C_0 \), and that of all mutant executions as \( C_\mu \), and that achieved by DE Miner as \( C_D \).

Figure 5 shows the relation among \( C_0 \), \( C_\mu \), and \( C_D \).

A. RQ1. With given test inputs, how many unreached lines/branches of an original program are covered by mutant executions?

The experiment results on RQ1 show that mutation effectively diversifies program executions in a large degree, so that many unreached lines and branches are covered on the mutants with the given test cases. (i.e., \( |C_\mu - C_0| \gg 0 \) in Figure 5).

Table IV shows the results on coverage-increasing mutants. The second column shows a number of all mutants used for the experiments, and the third column shows the number of mutant executions. The forth and the fifth columns report a number of the lines and branches that are covered by at least one mutant execution, respectively (i.e., \( |C_\mu| \)).

The results shows that, with the same given test cases, the executions on mutants cover meaningfully large amount of additional lines and branches compared to the execution on the original program. For example of GNU-find, the given 120 test cases cover 2,505 lines on the 172 mutants (i.e., covering 313 (=2505-2192) more lines than the original program). The mutant executions increase line coverage by 41.2% (=363-257)/257), 16.4% (=163-140)/140, and 14.3% (=2505-2192)/2192) for Busybox-printf, Busybox-printf, and GNU-find, respectively. Also, the mutant executions improve branch coverage by 83.0% (=247-135)/135), 32.0% (=99-75)/75) and 18.5% (=1257-1061)/1061) for Busybox-printf, Busybox-printf, and GNU-find, respectively. Note that these coverage increments appear on the mutants, not on the original target program.

B. RQ2. How many unreached lines/branches of an original program are covered by DE Miner?

The experiment results on RQ2 show that DE Miner effectively increases test coverage by utilizing the knowledge on the mutant executions.

In Table V, the second column represents a number of generated guideposts for each study object. The third and forth columns show a total numbers of lines and branches that DE Miner covers in generating 500 test cases for each guidepost with each initial test case, respectively.

For example of Busybox-printf, DE Miner generates 3,087 guideposts and covers 348 lines, which increases coverage of all given test cases by 35.4% (=348-257)/257).

\[
\begin{array}{|l|c|c|c|c|}
\hline
\text{Program} & \text{Total} & \text{Cov-incr.} & \text{# of cov. line} & \text{# of cov. br.} \\
& \text{mutants} & \text{mutants} & (\text{line cov.}) & (\text{br. cov.}) \\
\hline
\text{Busybox-printf} & 17313 & 1280 & 363 (89.9\%) & 247 (81.5\%) \\
\text{Busybox-printf} & 5728 & 118 & 163 (96.4\%) & 99 (94.3\%) \\
\text{GNU-find} & 5501 & 172 & 2505 (69.3\%) & 1257 (60.1\%) \\
\hline
\end{array}
\]

Fig. 5. Diagrams showing the relation among coverages by regression test cases (\( C_0 \)), mutant executions (\( C_\mu \)), and DE Miner (\( C_D \))
For Busybox-printf and GNU-find, DE MinER increases line coverage 15.0% (=(161-140)/140) and 12.1% ((2458-2192)/2192), respectively. In addition, it increases branch coverage of Busybox-printf, Busybox-printf, and GNU-find by 71.1% (=(231-135)/135), 24.0% (=(93-75)/75), and 12.9% (=(1198-1061)/1061), respectively.

C. RQ3. How many unreached lines/branches covered by the mutant executions are also covered by DE MinER?

Table VI compares the coverage of the mutant executions and DE MinER. The third and fourth columns show a number of newly covered lines/branches covered by both mutant executions and DE MinER, respectively. The fifth column shows a number of the unreached lines/branches covered by both mutant executions and DE MinER (i.e., |(C_D − C_0) ∩ (C_M − C_0)|).

For example of Busybox-ls, DE MinER covers 85.8% (= 206/239) of the unreached lines covered by the mutant executions. For Busybox-printf and GNU-find, DE MinER covers 91.3% (=21/23) and 80.2% (=251/313) of the mutant executions cover (see the fifth column). Similarly, for Busybox-printf and GNU-find, it covers 85.7% (=96/112), 75.0% (=18/24), and 55.1% (=108/196) of the unreached branches that the mutant executions cover (see the third and fifth columns).

This result shows that the guideposts effectively guide concolic testing in DE MinER to cover most unreached lines and branches covered by the mutation executions.

Note that, for GNU-find, DE MinER covers 15 lines (=266-251) and 29 branches (=137-108) which are covered neither by the initial test cases nor by the mutant executions. This fresh coverage increase is because DE MinER can explore diverse execution space beyond the ones reached by the mutant executions. Regarding Busybox-printf and Busy-printf, which has relatively smaller execution space than GNU-find, we conjecture that there exist only too few lines/branches left for DE MinER to cover beyond the mutant executions.

D. RQ4. How many unreached lines/branches of an original program does DE MinER cover, compared to conventional concolic testing techniques?

Table VII compares the coverage achievements of DE MinER with the three conventional concolic testing techniques. The third to fifth columns show the number of newly covered lines/branches (compared to the initial test cases) by concolic testing with DFS, RND, and CFG search strategies, respectively. The last column represents the result of DE MinER.

The experiment results show that DE MinER covers more lines and branches than all three studied concolic testing techniques.

For example of Busybox-ls, DE MinER covered 96 unreached branches which are 14.3% (=96-84)/84) more branches than the conventional concolic testing using DFS, which is the best conventional concolic testing technique for Busybox-ls. Similarly, for Busybox-printf and GNU-find, DE MinER outperformed the three conventional testing techniques by covering 20.0% (=18-15)/15) and 39.8% (=137-98)/98) more unreached branches than CFG and RND which are the best conventional testing techniques for Busybox-printf and GNU-find, respectively. On average, the amount of the improved branch coverage by DE MinER is 24.7% (=14.3+20.0+39.8)/3) relatively larger than those of the conventional concolic testing techniques.

In addition, Figure 6 shows the increase of covering unreached lines for 1,000 test input generations of the three conventional concolic testing techniques and DE MinER. The X-axis represents a number of test cases generated by DE MinER for each guided concolic testing instance (i.e., concolic testing of a target program having one guidepost using one initial test case). The Y-axis represents a number of unreached lines covered.

The result shows that, for all target programs and all levels of the numbers of generated test cases, DE MinER always covers the largest number of unreached lines among the conventional concolic testing techniques. For example of GNU-find with 1,000 test cases generation, DE MinER

---

**TABLE V**

<table>
<thead>
<tr>
<th>Program</th>
<th># of guideposts</th>
<th># of cov. line (line cov.)</th>
<th># of cov. br. (br. cov.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busybox-ls</td>
<td>3087</td>
<td>348 (86.1%)</td>
<td>231 (76.2%)</td>
</tr>
<tr>
<td>Busybox-printf</td>
<td>183</td>
<td>161 (95.3%)</td>
<td>93 (88.6%)</td>
</tr>
<tr>
<td>GNU-find</td>
<td>475</td>
<td>2458 (68.0%)</td>
<td>1198 (57.3%)</td>
</tr>
</tbody>
</table>

**TABLE VI**

<table>
<thead>
<tr>
<th>Program</th>
<th>Cov</th>
<th>[C_M − C_0]</th>
<th>[C_D − C_0]</th>
<th>([C_D − C_0] ∩ [C_M − C_0])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busybox-ls</td>
<td>Line</td>
<td>106</td>
<td>91</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Br.</td>
<td>112</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>Busybox-printf</td>
<td>Line</td>
<td>23</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Br.</td>
<td>24</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>GNU-find</td>
<td>Line</td>
<td>313</td>
<td>266</td>
<td>251</td>
</tr>
<tr>
<td></td>
<td>Br.</td>
<td>196</td>
<td>137</td>
<td>108</td>
</tr>
</tbody>
</table>
covers the unreached lines around 27% (= (332-261)/261) more than the best conventional concolic testing technique (i.e., RND) (see the right end of the find graph in the figure).

E. RQ5. To what extent does the mutant selection of DEMINER affect execution time and line/branch coverage?

Table VIII shows the effects of the coverage-increasing mutant selection. The second to fourth columns show the line and branch coverage that DEMINER achieves with the greedy mutant selection algorithm, a random mutant selection which selects the same number of mutants selected by the greedy one, and using all coverage-increasing mutants, respectively. The fifth to seventh columns show the execution time in hours taken by DEMINER with the corresponding methods.

The experiment results show that the greedy selection of the mutants effectively reduces the runtime cost without hurting coverage much (except GNU-find) 7. DEMINER with the greedy mutant selection consumes only 1.2% (= 9.5/776.5) (Busybox-ls) to 5.3% (= 6.5/123.8) (Busybox-printf) of the execution time of DEMINER with all coverage-increasing mutants.

Also, DEMINER with the greedy selection covers more lines and branches than DEMINER with random selection for all cases. For example of Busybox-1s, DEMINER with

Fig. 7. An example of hard-to-reach lines of Busybox-printf for conventional concolic testing

the greedy mutant selection covers 91 unreached lines while DEMINER with the random selection does only 85 lines.

VI. DISCUSSION

A. Hard-to-reach Lines which DEMINER can cover

Figure 7 shows an example of hard-to-reach lines of Busybox-printf for conventional concolic testing. print_direc takes a character pointer format which points to a character array whose elements are symbolic input characters as the first parameter (Line 160). print_direc assigns a symbolic input character of format to ch (Line 169). Then, it calls strstr to check if format contains ‘.’ or ‘*’ (Line 172) and calls strchr to get a position of ‘*’ in format (Line 173). print_direc converts input data according to the format string character using switch statement on Line 179. Lines 202–203 are not covered by the regression test inputs provided in Busybox-printf 1.24.0.

Conventional concolic testing using DFS and RND could not cover Lines 202–203 in 23,394 seconds, due to the loops inside strstr at Line 172 and strchr at Line 173. Both of the loops have symbolic variables in their loop conditions,
because they iterate over a symbolic input string \texttt{format} until they reach any specified character (i.e., '.', '*', or '\0'). Therefore, conventional concolic testing keeps increasing the loop bound and fails to cover Lines 202–203 within a given time bound.

Note that DE MINER covers these hard-to-cover lines as follows. DE MINER generates a mutant \( m \) which mutates a variable \( \text{fmt\_length} \) to another variable \( \text{precision} \). One of the mutant executions of \( m \) covers Lines 202–203 and the monitoring probe for \( m \) (i.e., \texttt{probe(format[precision])}) at right before Line 169) reports ‘\( x \)’ as a value of the mutated target expression. Using the reported value ‘\( x \)’, DE MINER inserts a guidepost at Line 168 and guided concolic testing of DE MINER tries to satisfy the guidepost condition with high priority instead of negating branches of the loops inside \texttt{strstr} and \texttt{strchr} at Lines 172–173 (see Section III-D). Therefore, DE MINER effectively covers Lines 202–203.

Furthermore, for GNU-find, DE MINER covers 16 lines that seems not reachable by the conventional concolic testing by any means. We ran each of the conventional concolic testing techniques using DFS, RND and CFG for one week (i.e., 168 hours) per each initial test case (i.e., execution time is 120 weeks in total). Then, we found that there are 16 lines which are covered by DE MINER, but they were never covered by any test executions by conventional concolic techniques with 120 weeks of the testing time. DE MINER can cover these 16 lines successfully because the guideposts prune the search space that is not relevant to cover new lines, and effectively guide concolic testing toward execution paths that reach unreached lines observed from diverse coverage-increasing mutant executions.

\section*{B. Application of Mutation Analysis}

Program mutation has been a popular method for evaluating how given test cases detect subtle program changes. Traditional research on software mutation [22] mainly focuses on evaluating bug finding effectiveness of a test suite by measuring how many mutants the test suite can kill.

Fraser and Zeller [23] presents a search-based unit test generation technique that targets mutants as a way to generate diverse unit tests. The technique directs test case generation toward finding output and coverage differences between a target program and its mutants. Both our approach and that by Fraser and Zeller [23] are common in that both analyze mutant executions. The difference is that in our approach, the dynamic information on mutants is used for inferring internal conditions of a target program to increase test coverage, rather than measuring the difference between the target program and a mutant.

Program mutation has also been used to generate and evaluate test oracles. Mutation analysis is used to examine which properties are invariants of the correct program. Fraser and Zeller [23] utilizes mutation analysis to infer test oracles from the mutant execution information. Jahangirova et al. [24] presents a method to assess and improve the quality of a given test oracle by utilizing program mutation. Recently, mutation analysis has been used to precisely localize a fault in a program. Mutation-based fault localization [25]–[28] locates a fault in the target program code by observing how the behaviors of the faulty program code change according to the program code chances.

A few research work have focused on mutating program code or runtime states to derive diverse executions for better dynamic analyses [29], [30]. These techniques utilize values and execution paths observed from mutated program executions directly during the analyses of the original program without validating whether these observations also hold for the original program. Thus, these techniques may produce unsound analysis results because mutated program executions may be infeasible on the original program.

Our approach alleviates this false positive problem by generating only feasible test inputs by leveraging concolic testing techniques with the discovered knowledge from the mutation analyses.

\section*{VII. CONCLUSION}

This paper presents DE MINER which is an automated test generation technique that realizes the invasive software testing paradigm by utilizing information from diverse mutant executions. We demonstrated that DE MINER can effectively increase test coverage by applying DE MINER to three real-world C programs.

We plan to extend DE MINER in the following directions. First, to enhance the knowledge discovery, we will improve DE MINER to employ more mutation operators, including statement-level and higher-order mutation operators. Second, to learn the coverage-increasing conditions more efficiently and effectively, we will utilize the automated unit test generation technique using concolic testing [31] which can generate the coverage-increasing conditions at an entry of a function. Third, we will leverage program invariant inference techniques, such as Daikon [32], to generate various kinds of guideposts from mutant execution information. In addition, we will explore what kinds of guidepost condition structures are effective for capturing useful knowledge from mutant executions. Finally, we will apply DE MINER to more real-world programs to show that DE MINER is generally effective in finding unknown bugs in real-world programs.

\section*{ACKNOWLEDGMENTS}

This research was supported by Next-Generation Information Computing Development Program (No. 2017M3C4A7068177, No. 2017M3C4A7068179), Basic Science Research Program (No. 2016R1A2B4008113, No. 2017R1C1B1008159) through the National Research Foundation (NRF) funded by the Ministry of Science and ICT of Korea, Basic Science Research Program (No. 2017R1D1A1B03035851) through NRF funded by the Ministry of Education of Korea, and KAIST High Risk High Return Project (HRHRP).
REFERENCES


