SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–30
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

DEMINER: Test Generation for High Test Coverage through Mutant Exploration

Yunho Kim[1], Shin Hong[2*]

[1]*School of Computing, KAIST, South Korea*
[2]*School of CSEE, Handong Global University, South Korea*

SUMMARY

Most software testing techniques test a target program *as it is*, and fail to utilize valuable information of *diverse test executions on many variants/mutants* of the original program in test generation. This paper proposes a new test generation technique DEMINER which utilizes mutant executions to guide test generation on the original program for high test coverage. DEMINER first generates various mutants of an original target program, and then extracts runtime information of mutant executions which covered unreached branches by the mutation effects. Using the obtained runtime information, DEMINER inserts *guideposts*, artificial branches to replay the observed mutation effects, to the original target programs. Finally, DEMINER runs automated test generation on the original program with guideposts and achieves higher test coverage. We implemented DEMINER for C programs through software mutation and guided test generation such as concolic testing and fuzzing. We have shown the effectiveness of DEMINER on six real-world target programs: `Busybox-ls`, `Busybox-printf`, `Coreutils-sort`, `GNU-find`, `GNU-grep`, and `GNU-sed`. The experiment results show that DEMINER improved branch coverage by 63.4% and 19.6% compared to those of the conventional concolic testing techniques and the conventional fuzzing techniques on average, respectively.

Received . . .

KEY WORDS: automated test generation, concolic testing, fuzzing, mutation analysis, test coverage

1. INTRODUCTION

Although test generation techniques have advanced significantly over several decades, most of them still test a target program *as it is* without modifying the target program structure. In contrast, other disciplines use testing methods (e.g., *invasive tests* in medical science, or *destructive tests* in mechanical engineering) that analyze a target object by inducing physical forces to transform its structure to induce unusual conditions to obtain information which is difficult to provide by non-invasive analyses.

A software program is an ideal object to analyze by modifying/breaking its structure, because it is free to make identifical copies, to induce various alternative changes, and revert the modification after the analysis. Based on this nature of software, *mutation testing*, which systematically generates various structural variants of a target program and utilizes them for different kinds of software analyses, has been proposed and widely studied in the last two decades [1]. Although mutation testing methods lead many advances in software testing techniques, their applications to testing

*Correspondence to: Shin Hong, School of CSEE, Handong Global University, South Korea. E-mail: hongshin@handong.edu

techniques are still limited to evaluating existing test suites, rather than generating new test inputs by utilizing runtime information obtained from mutation testing results. This is because it is non-trivial to apply various dynamic information from many mutants to analyze the original target program due to high complexity of the target programs. For example, it is difficult to draw a conclusion on an original program p by observing crashes on one of its variants m_1 with a test case t_1, because a crash of m_1 with t_1 does not necessarily mean that p will crash with t_1.

To resolve the aforementioned limitation, we propose a new approach to utilize mutant executions for test generation, which takes the following three stages:

1. Extracting/learning useful information on an original target program p for high test coverage through diverse test exploration of diverse variants/mutants of p
2. Generating *guideposts* by using the extracted information, which can guide test generation for high test coverage
3. Generating test executions on p following/satisfying the guideposts inserted in p

A core idea of our approach is two-fold. First, our approach explores new behaviors of a target programs by changing the structure of a target program to know under which conditions an execution can cover new branches. We conjecture that, if many mutants cover new branches under certain condition, an original target program can also cover these new branches under the same condition.

Second, our approach captures such a coverage-increasing condition as a *guidepost*. A guidepost is a new side-effect-free branch condition inserted to a certain location of a target program and it can effectively lead test generation to cover various executions of p that achieve high test coverage.

We developed DEMINER (gui**DE**ed test generation using **M**utat**I**o**N** **E**xplo**R**ation) to improve test coverage achievement of automated test generation techniques (i.e., concolic testing, fuzz testing). DEMINER operates as follows:

1. DEMINER 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, DEMINER executes $m_1, \ldots, m_n$ with T and records test runs (i.e., mutant explorations) covering new lines (i.e., lines not covered on p with T) and corresponding mutation instances.
3. Based on the recorded information, DEMINER constructs *guideposts* to guide test generation on p to improve test coverage.
4. DEMINER applies automated test generation techniques such as concolic testing or fuzzing on p with an inserted guidepost to generate new test executions that follow the guideposts and achieve high test coverage.

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 μ_1 (i.e., replace l_2 with l_2'), and t covers l_6 on m_1. From this mutant execution, DEMINER captures a partial state c_1 infected by μ_1 as a key for t to cover l_6, and then DEMINER inserts `guidepost(`c_1`)` right before the mutation site of μ_1 in p. Finally, DEMINER automatically generates new tests (e.g., t_1) on p with the inserted guidepost so that the new tests follow/satisfy the guidepost condition c_1. DEMINER repeats this process for all other coverage-increasing mutants (i.e., mutants whose executions cover lines that were not covered on p with T).

The guidepost generation strategy is important for code coverage and the execution time cost of DEMINER. We have developed five types of guideposts to study how much the different types of guidepost affect the code coverage and the execution time of DEMINER. We have developed two types of guideposts which utilize a single value monitored from mutant executions and three types of guideposts which utilize multiple values monitored from mutant executions (see Section 3.3).

We have studied the effectiveness of DEMINER on six real-world target programs `Busybox-ls`, `Busybox-printf`, `Coreutils-sort`, `GNU-find`, `GNU-find`, `GNU-grep`, and `GNU-sed`. The experiment results show that DEMINER improved branch coverage by 63.4% and 19.6% compared to those of the conventional concolic testing techniques and the conventional fuzzing on average, respectively (see Section 5.5).

DOI: 10.1002/stvr

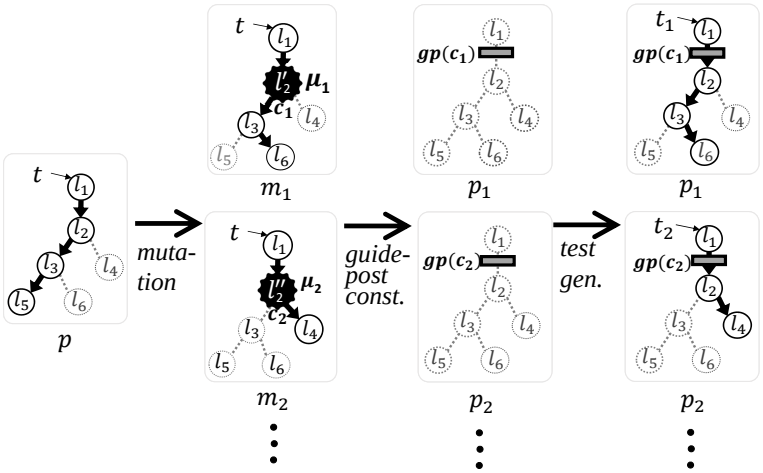


Figure 1. DEMINER overview

The idea of utilizing mutant executions to guide test input generation was first proposed by Kim et al. [2], which proposed DEMINER that utilized concolic testing as a test input generation technique. To improve applicability and test effectiveness of DEMINER, this paper extends Kim et al. [2] on the following points:

1. Now, as a new guided test generator, DEMINER has DEMINER-FUZZ which employs a fuzz testing using a modified version of AFL (in addition to DEMINER-CT).
2. We propose to construct guideposts of five different types (in addition to the guideposts of the two types proposed by Kim et al. [2]). We found that the guideposts of the new types (i.e., Single-value exclusion and Multi-value range guideposts) improve overall performance of DeMiner.
3. Through the new experiments, we showed that DEMINER-FUZZ outperforms a state-of-art fuzzing technique T-FUZZ [3] due to the new search strategy towards inserted guideposts of DEMINER.

The remainder of this paper is organized as follows. Section 2 shows a motivating example of a limitation of a current automated test generation technique and how DEMINER can overcome it. Section 3 explains DEMINER, and Section 4 describes research questions and the experiment setup to study the effectiveness of DEMINER. Sections 5 and 6 report and discuss the experiment results. Section 7 presents related work. Finally, Section 8 concludes this paper with future work.

2. MOTIVATING EXAMPLE

Although automated test generation techniques such as concolic testing [4–7] 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 [8–10], symbolic pointers [11–13], loop conditions with symbolic bound variables [14–16]).

For example of `get_max` in Figure 2, concolic testing with the DFS (Depth-First Search) concolic search strategy fails to cover the branch consisting of Lines 9–10. Figure 2 shows `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 as a concolic search strategy.

Suppose that an initial input for `get_max` has

- `sz=100`, and

DOI: 10.1002/stvr

```
01  get_max(int* a, unsigned int sz) {
02    max = 0 ;
03    for (i = 0 ; i < sz ; i++) {
04      if (max < a[i])
05        max = a[i] ;
06    }
07    if (max > 0)
08      printf ("%d\n", max) ;
09    else
10      error() ;
11  }
```

Figure 2. Example whose Lines 9–10 are difficult to cover by concolic testing

- `a` is sorted in a strictly ascending order, and
- every element of `a` is positive

Then, the symbolic path formula ϕ_1 obtained from an execution with the initial input is as follows:

$$\phi_1 = (0 < sz) \wedge (0 < a[0]) \wedge (1 < sz) \wedge (a[0] < a[1]) \wedge \ldots \wedge (99 < sz) \wedge (a[98] < a[99]) \wedge (100 \not< sz) \wedge (a[99] > 0)$$

Note that the subsequent concolic executions with DFS (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 ϕ_1' is unsatisfiable. This is because $a[0]$ should be positive (i.e., $0 < a[0]$ in ϕ_1') and a is sorted in a strictly ascending order (i.e., $a[0] < a[1] < \ldots < a[99]$ in ϕ_1'). Subsequently, concolic testing negates second last condition in ϕ_1 (i.e.,$100 \not< sz$) and generates the second input with `sz=101`, which still *does not* cover Lines 9–10. The symbolic path formula ϕ_2 obtained from the second input is longer than ϕ_1 by iterating the for-loop (Lines 3–5) one more time with $i = 100$ as follows:

$$\phi_2 = (0 < sz) \wedge (0 < a[0]) \wedge (1 < sz) \wedge (a[0] < a[1]) \wedge \ldots \wedge (100 < sz) \wedge (a[99] < a[100]) \wedge (101 \not< sz) \wedge (a[100] > 0)$$

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 < max`. The execution of m_3 with the initial input does not enter the loop and covers Lines 9–10 because `max` is zero and `i < 0` is false. Then, DEMINER learns from this mutant execution and generates a guidepost `guidepost(sz == 0)` between Line 2 and Line 3 of `get_max` (see Section 3.3 for the detail of guidepost construction).

Since `guidepost(c)` is a macro of `if (!c) exit(0);`, the initial execution terminates at the guidepost because the execution does not satisfy the guidepost condition (i.e., `sz == 0`). 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 `sz == 0`). Finally, `get_max` reaches Lines 9–10 with this test input generated with the guide of the guidepost.

DOI: 10.1002/stvr

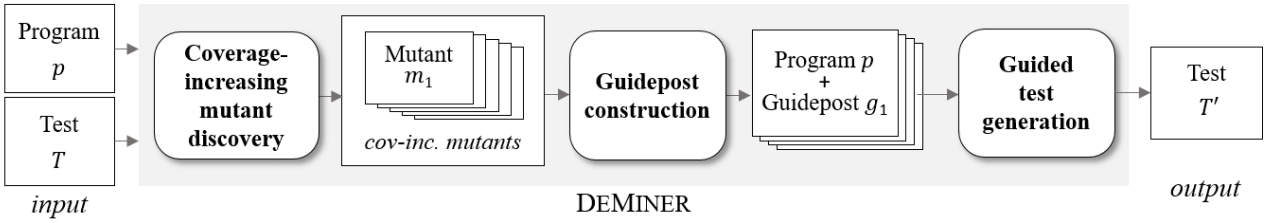


Figure 3. The overall process of DEMINER

3. DEMINER FRAMEWORK

3.1. Overview

DEMINER employs mutation to generate diverse variants or 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 and fuzzing techniques to guide their test generations to cover these new lines in p.

We conjecture that, even with limited test inputs, program mutation can effectively diversify program executions, 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 as mutant execution generation does not require sophisticated semantic analysis. DEMINER's mutant execution analysis overhead is not much high because DEMINER just extracts line coverage information and evaluation results of the mutated expression from the mutant executions. In addition, mutant execution generation 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 $T = \{t_1, t_2, \ldots, t_k\}$ 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. DEMINER uses all the generated mutants to discover the coverage-increasing mutants.
2. *Guidepost construction*
   Based on the mutant executions that cover unreached lines, DEMINER infers program conditions of the executions covering the unreached lines as *guidepost*s. Then, DEMINER generates multiple copies of p each of which has one guidepost.
3. *Guided test generation*
   DEMINER runs concolic testing or fuzzing 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. DEMINER-CT and DEMINER-FUZZ are instances of DEMINER which use concolic testing and fuzzing as a test input generation technique for guided test generation, respectively.

The remainder of this section describes each phase in detail.

3.2. 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 $m_1, m_2, \ldots, m_n$ by mutating expressions $e_1, e_2, \ldots, e_l$ in p respectively, and then runs the mutants with T.

DOI: 10.1002/stvr

Table I. Mutation operators used by DEMINER

Category	Mutation operator names
Change a value or constant	CRCR, VTWD
Change a variable or memory access	VGAR, VLAR, VGSR, VLSR, VSCR OAAA, OAAN, OABA, OABN, OAEA, OALN, OARN, OASA, OASN, OBAA, OBAN, OBBA, OBBN, OBEA, OBLN, OBNG, OBRN, OBSA, OBSN, OEAA,
Change an operator in an expression	OEBA, OESA, OLAN, OLBN, OLLN, OLNG, OLRN, OLSN, ORAN, ORBN, ORLN, ORRN, ORSN, OSAA, OSAN, OSBA, OSBN, OSEA, OSLN, OSRN, OSSA, OSSN, OIPM
Change a branching condition	OCNG, OCOR

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 a line only if the line is reached by at least one test input in T.

Table I shows names and categories of the mutation operators [17] 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 `break` with `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).

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

Finally, DEMINER selects a subset of the coverage-increasing mutants $M \subseteq M_{All}$ that are passed to the next phase (see Section 3.3). 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., $C_M - C_p = C_{M_{All}} - C_p$).

*The line coverage of m_i is compatible with that of p as DEMINER carefully mutates p to keep the line numbers the same (see Section 4.6).

DOI: 10.1002/stvr

3.3. 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., `guidepost(`exp`)` $\equiv$ `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 test generation on a target program p to generate tests toward the observed coverage-increasing executions.

To infer guideposts from the selected mutants $M = \{m_1, m_2, \ldots, m_{n'}\}$, DEMINER first re-runs the original program p, and 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 than the evaluation of e_i at the original program.

For each mutant m_i, DEMINER identifies all runtime evaluations of the mutated expression as *coverage-increasing values* $V_i = \{v_1^i, v_2^i, \ldots, v_{w_i}^i\}$. 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. If there exist multiple unreached lines that m_i additionally covers, $C_{m_i} - Cp = \{l_1, l_2, \ldots, l_h\}$, DEMINER defines $V_i^{l_j} \subseteq V_i$ as a set of the coverage-increasing values observed when m_i covers l_j, such that $\bigcup_{1 \leq j \leq h} V_i^{l_j} = V_i$. In addition, DEMINER identifies all runtime evaluations of the original expression at the mutation site as $U_i = \{u_1^i, u_2^i, \ldots, u_{w'_i}^i\}$.

Once the coverage-increasing values of m_i are captured, DEMINER first infers preconditions of the observed coverage-increases as predicates over e_i with V_i and U_i. After that, for each inferred precondition, a guidepost is inserted immediately before the mutation site of m_i (i.e., riget before e_i is evaluated) such that the guidepost enforces an execution to continue only if e_i is evaluated to satisfy the inferred precondition.

DEMINER constructs five types of guideposts from V_i as follows:

- *Sinle-value equality guidepost (Seq)*
  For each $v_j^i \in V_i$, a single-value equality guidepost is created to check if e_i is evaluated to v_j^i for the first time. This condition is encoded as `guidepost(`$e_i == v_j^i$`)` for $v_j^i \in V_i$.
- *Single-value exclusion guidepost (Sx)*
  For each runtime evaluation observed in the original program $u_j^i \in U_i$, a single-value exclusion guidepost is created to check if e_i is evaluated to a different value than u_j^i for the first time. This condition is encoded as `guidepost(`$e_i != u_j^i$`)` for $u_j^i \in U_i$.
- *Multi-value equality guidepost (Meq)*
  For m_i with multiple coverage-increasing values (i.e., $|V_i| > 1$), DEMINER creates a multi-value equality 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 `guidepost(`$(e_i == v_1^i) \vee \ldots \vee (e_i == v_{w_i}^i)$`)`.
- *Coverage-based guidepost (Mcov)*
  For m_i that covers multiple unreached lines (i.e., $|C_{m_i} - C_p| > 1$), DEMINER creates a guidepost for each $l_j \in C_{m_i} - C_p$ that checks e_i is always evaluated to one of the coverage-increasing values in $V_i^{l_j}$ that covers l_j. This condition is encoded as `guidepost(`$(e_i == v_1^{i,l_j}) \vee \ldots \vee (e_i == v_x^{i,l_j})$`)` for $V_i^{l_j} (= \{v_1^{i,l_j}, v_2^{i,l_j}, \ldots, v_x^{i,l_j}\})$.
- *Multi-value range guidepost (Mrange)*
  For m_i with multiple coverage-increasing values (i.e., $|V_i| > 1$), DEMINER creates a guidepost that checks e_i is always evaluated to one of the values between $Min(V_i)$ and $Max(V_i)$. The guidepost of this type is formed as `guidepost(`$Min(V_i)$ `<=` $e_i \wedge e_i$ `<=` $Max(V_i)$`)`.

DEMINER constructs various guideposts from a same set of coverage-increasing values V_i. This is because guideposts of a certain type with a limited set of runtime values may fail to capture

DOI: 10.1002/stvr

```
   func(x,y) {
1:   z = x + y ;
2:   if (z < 0)
3:     print(x) ;
4:   else if (z ==0)
5:     print(y) ;
```

(a) target program p

```
   func(x,y) {
1:   z = x - y ;
2:   if (z < 0)
3:     print(x) ;
4:   else if (z ==0)
5:     print(y) ;
```

(b) a mutant m_1

```
   func(x,y) {
1:   z = probe(x - y);
2:   if (z < 0)
3:     print(x) ;
4:   else if (z ==0)
5:     print(y) ;
```

(c) m_1 with a monitoring probe

```
   func(x,y) {
1':  guide(x + y == -1);
1:   z = x + y;
2:   if (z < 0)
3:     print(x) ;
4:   else if (z ==0)
5:     print(y) ;
```

(d) p augmented with guidepost g_0

Figure 4. Example of guidepost construction

actual preconditions of covering unreached lines. Also, the guidepost types can affect efficiency of DEMINER because DEMINER generates a different number of guideposts according to the type of guideposts (see Section 5.2).

Single-value equality guideposts (Seq) guide that e_i should be evaluated to one specific coverage-increasing value in V_i. Seq works effectively when a specific value v_j^i is essential to cover unreached lines. A single-value exclusion guidepost (Sx) is generated for each runtime value of the original expression at a mutation site (i.e., U_i). Sx is intended to alternate the evaluations of a target expression with new values in test generation. Thus, Sx is effective when a target expression should be evaluated to values other than ones in U_i to cover more unreached lines.

Multi-value equality guideposts (Meq) guide that e_i is evaluated to one of the observed coverage-increasing values in V_i. Unlike Seq, Meq allows a target expression to have multiple value choices. Coverage-based guideposts (Mcov) are created to target each unreached line. From V_i, DEMINER identifies a group of coverage-increasing values related to each unreached line, and then generates an individual guidepost for each group. Multi-value range guideposts (Mrange) are generated to accept a range of values (a number of the Mrange guideposts is usually much less than the Meq guideposts).

For each guidepost g, DEMINER generates a version of p which inserts g to right before the target expression e to reproduce the mutation effect on the original program. Figure 4 illustrates how DEMINER generates guideposts with monitored runtime values and inserts them to a target program p. Suppose that DEMINER created a mutant m_1 from p by changing an operator at Line 1 (i.e., e is `x+y`). With three test inputs that execute `func(0,1)`, `func(0,2)`, `func(1,1)`, Lines 3 and 5 are not covered on p, but covered on m_1 (i.e., $C_{m_1} - C_p = \{l_3, l_5\}$). Once DEMINER finds that m_1 covers unreached lines, it inserts a probe to m_1 to monitor values of the mutated expression (i.e., `x - y`) (Figure 4-(c)). By re-running m_1 with a monitoring probe, DEMINER finds that the mutated expression is evaluated to -1, -2, and 0 (i.e., $V_1 = \{v_0^1 = -1, v_1^1 = -2, v_2^1 = 0\}$). DEMINER identifies that an execution of p may cover Line 3 if e is evaluated to -1 or -2 (i.e., $V_1^{l_3} = \{v_0^1, v_1^1\}$) and p may cover Line 5 if e is evaluated to 0 (i.e., $V_1^{l_5} = \{v_2^1\}$). In addition, DEMINER inserts a probe to p and extracts the original runtime values at the mutation site. When p runs with the three test inputs, DEMINER finds e is evaluated to 1, 2, and 2, respectively (i.e., $U_1 = \{u_0^1 = 1, u_1^1 = 2\}$).

DEMINER generates total nine guideposts (i.e., $g_0, g_2, \ldots, g_8$) with the m_1 results as follows:

DOI: 10.1002/stvr

- Three Seq guideposts:
  DEMINER generates the following three guideposts using the three coverage-increasing values:

  g_0 : `guide(x + y == -1)` (i.e., v_0^1)
  g_1 : `guide(x + y == -2)` (i.e., v_1^1)
  g_2 : `guide(x + y == 0)` (i.e., v_2^1)
- Two Sx guideposts:
  DEMINER generates two single-value exclusion guideposts with the two runtime values observed at the original program (i.e., U_1), as follows:

  g_3 : `guide(x + y != 1)` (i.e., u_0^1)
  g_4 : `guide(x + y != 2)` (i.e., u_1^1)
- One Meq guidepost:
  A multi-value equality guidepost is generated to guide that the target expression must be evaluated to one of the three coverage-increasing values, as follows:

  g_5 : `guide(x + y == -1 || x + y == -2 || x + y == 0)` (i.e., V_1)
- Two Mcover guideposts:
  As m_1 covers Lines 3 and 5 which were not covered on p, DEMINER generates two coverage-based guideposts as follows:

  g_7 : `guide(x + y == -1 || x + y == -2)` (i.e., $V_1^{l_3}$)
  g_8 : `guide(x + y == 0)` (i.e., $V_1^{l_5}$)
- One Mrange guidepost:
  DEMINER generates a multi-value range guidepost for mutant m_1, which guides that an evaluation of the target expression must be in the range of the coverage-increasing values.

  g_6 : `guide(-2<=x + y && x + y<=0)` (i.e., $Min(V_1) = -2$ and $Max(V_1) = 0$)

A guidepost g_i is inserted to p such that it is guaranteed to be executed right before the target expression. Each guidepost g_i is used for constructing one version of p. Figure 4-(d) shows a new version of p generated with g_0.

3.4. Phase 3. Guided test generation

This phase takes multiple versions of p each of which is augmented with one guidepost and runs guided test generation techniques to generate test inputs. DEMINER uses concolic testing and fuzzing as guided test generation techniques.

3.4.1. Guided test generation using concolic testing Algorithm 1 shows the algorithm of DEMINER-CT test generation. The algorithm takes a target program with a guidepost P_{guide}, a regression test suite T_{reg}, and a number of test cases to generate n as inputs and generates a set of test cases T_{new}. DEMINER applies concolic testing to P_{guide} with each test input $t \in T_{reg}$ as an initial test input (Lines 3–15). DEMINER uses a new prioritized concolic search strategy which applies depth-first search (DFS) until the guidepost condition c is satisfied and applies random branch negation (RND) after c is satisfied (Lines 5–14).

- The search algorithm first performs concolic testing using DFS until the guidepost condition c is satisfied (Line 5–9). If 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 another test execution. DEMINER repeats these steps until it generates an test input that satisfies c.
- If concolic testing generates a test execution satisfying c, DEMINER uses RND to negate only those branches executed after the guidepost (Lines 10–14). This is to focus on the execution space satisfying the guidepost condition c and, thus, to have high probability to increase coverage.

Finally, the algorithm returns T_{new} as an output (Line 16).

DOI: 10.1002/stvr

Algorithm 1: DEMINER-CT test generation algorithm

Input: P_{guide}: A target program with a guidepost, T_{reg}: a regression test suite ($\neq \emptyset$), n: a number of test cases to generate per guidepost and initial test input by DEMINER-CT
Output: T_{new}: A set of generated test cases

```
1  DeMiner_CT(P_guide, T_reg, n){
2  T_new = ∅
3  foreach t_init ∈ T_reg do
4      t = t_init
5      repeat
6          path = execute(P_guide, t)
7          t = next_test_DFS(path)
8          T_new = T_new ∪ t
9      until path satisfies the guidepost condition c in P_guide or |T_new| ≥ n;
10     while |T_new| < n do
11         path = execute(P_guide, t)
12         t = next_test_RND(path)
13         T_new = T_new ∪ t
14     end
15 end
16 return T_new }
```

3.4.2. Guided test generation using fuzzing Coverage-guided fuzzing maintains a pool of seed test inputs and iterates a fuzzing loop as follows:

1. pick a seed test input from the test input pool according to a policy of a fuzzer
2. mutate the seed test input to generate new test inputs
3. run a target program with these new test inputs
4. monitor the instrumented program execution to track code coverage as well as crashes and assert violations
5. put the test inputs contributing to code coverage into the pool and go to step 1.

DEMINER-FUZZ uses a modified version of AFL [18] which is a coverage-guided fuzzing technique to guide test generation using the guidepost. Algorithm 2 shows the algorithm of DEMINER-FUZZ test generation. The algorithm takes a target program with a guidepost P_{guide}, a regression test suite T_{reg}, and a number of test cases to generate n as inputs and generates a set of test cases T_{new}. First, T_{new} and T_{used} are initialized as an empty set and T_{seed} are initialized as T_{reg} (Lines 2–4). Then, the algorithm iterates a fuzzing loop (Lines 5–17) until it generates n new test cases. In the fuzzing loop, a seed test is selected using `ChooseSeed` function (Line 6) and the selected seed test is added to T_{used} to avoid repeated selection (Line 7). Then, a timeout for the mutation-execution loop (i.e., Lines 7–14) is calculated (Line 8). The mutation-execution loop (Lines 9–16) mutates a seed test using AFL's mutation operators (Line 10) and executes the target program with the mutated test cases (Line 11). If the execution path of the mutated test case t' is newly covered one (Line 12), the algorithm adds t' into T_{new} and T_{seed} (Line 13–14). If the algorithm generates n test cases, it returns T_{new} (Line 18).

`ChooseSeed`(P_{guide}, T_{seed}) (Lines 19–31) tries to select a test case in T_{seed} that satisfies the guidepost P_{guide} and that has both small input size and low execution time. For that purpose, the function computes the AFL score and DEMINER score for each seed test (Lines 22–25) as follows and selects a test case which has the smallest normalized score:

- *AFL score* (`compute_score`$_{\text{AFL}}$ at Line 23): AFL score is used by AFL to prioritize test inputs. AFL prioritizes test inputs that are both small and executed fast. AFL assigns a score to a test input which is proportional to its execution time and its size and selects a test input with the lowest score.

DOI: 10.1002/stvr

Algorithm 2: DEMINER-FUZZ test generation algorithm

Input: P_{guide}: A target program with a guidepost, T_{reg}: a regression test suite ($\neq \emptyset$), n: a number of test cases to generate by DEMINER-FUZZ
Output: T_{new}: A set of generated test cases

```
1  DeMiner_fuzz(P_guide, T_reg, n){
2  T_new = ∅
3  T_used = ∅
4  T_seed = T_reg
5  repeat
6      t = choose_seed(P_guide, T_seed \ T_used)
7      T_used = T_used ∪ t
8      timeout = assign_mutation_time(P_guide, t)
9      repeat
10         t' = mutate(t)
11         path = execute(P_guide, t')
12         if path is new then
13             T_new = T_new ∪ t'
14             T_seed = T_seed ∪ t'
15         end
16     until |T_new| ≥ n or timeout is reached;
17 until |T_new| ≥ n;
18 return T_new }
19 choose_seed(P_guide, T_seed){
20 score_AFL = []
21 score_DEMINER = []
22 foreach t ∈ T_seed do
23     score_AFL[t] = compute_score_AFL(P_guide, t)
24     score_DEMINER[t] = compute_score_DEMINER(P_guide, t)
25 end
26 norm_score_AFL = normalize_score(score_AFL)
27 norm_score_DEMINER = normalize_score(score_DEMINER)
28 foreach t ∈ T_seed do
29     norm_score[t] = α · norm_score_AFL[t] + β · norm_score_DEMINER[t]
30 end
31 return a seed test t whose normalized score (i.e., norm_score[t]) is lowest }
```

- DEMINER *score* (`compute_score`$_{\text{DEMINER}}$ at Line 24): DEMINER score is used to prioritize test inputs that satisfy (or are likely to satisfy) the guideposts. First, from an inter-procedural control-flow graph of a target program, it identifies the node which is executed by the given test input and nearest to the node representing the `then` branch of the guidepost P_{guide} in terms of the branch distance in an inter-procedural control-flow graph (i.e., approach level in search-based test generation [19]). Then, DEMINER assigns the distance between the two nodes as DEMINER score of a test input. DEMINER prefers test inputs with lower scores.

Then, the function computes the normalized score based on the AFL and DEMINER scores. Normalization converts the highest AFL score to 1 and the lowest AFL score to 0 (Line 24) (applying the same process to DEMINER score too (Line 25)) and lower normalized value is preferred. DEMINER-FUZZ uses $\alpha \cdot norm\_score_{AFL}[t] + \beta \cdot norm\_score_{\text{DEMINER}}[t]$ as a score of a test input t (Line 28–30). $norm\_score_{AFL}[t]$ and $norm\_score_{\text{DEMINER}}[t]$ are normalized values of AFL score and DEMINER score, respectively. For the weight of α and β, we use α as one and β

DOI: 10.1002/stvr

as 10 because our exploratory study shows that this ratio is better than the others (see Section 6.2). Finally, `choose_seed` returns a seed test t whose normalized score is the lowest one (Line 31).

4. EXPERIMENT SETUP

We have designed the following research questions to evaluate DEMINER mainly regarding its increased coverage.

- **RQ1.** *With given test inputs, how many unreached lines/branches of an original program are covered by mutant executions?*
- **RQ2.** *To what extent do the guidepost generation strategies of* DEMINER *affect line/branch coverage and execution time?*
- **RQ3.** *How many unreached lines/branches of an original program are covered by* DEMINER*?*
- **RQ4.** *How many unreached lines/branches covered by the mutant executions are also covered by* DEMINER*?*
- **RQ5**. *How many unreached lines/branches of an original program does* DEMINER *cover, compared to the conventional concolic testing and fuzzing techniques?*
- **RQ6**. *To what extent does the mutant selection of* DEMINER *affect execution time and line/branch coverage?*

RQ1 is to validate our conjecture that mutant executions cover a meaningfully large amount of unreached lines/branches that given test inputs do not cover on an original program p.

RQ2 evaluates the effectiveness and the efficiency of DEMINER with different guidepost generation strategies: two single-value strategies (single-value equality (Seq) strategy and single-value inequality (Sieq) strategy) and three multi-values strategies (multi-values equality (Meq), multi-values per coverage (Mcov) and multi-values range (Mrange)) (see Section 3.3). We compare the number of the generated guideposts, line and branch coverage, and execution time. Note that, after we choose the most cost-effective single-value guidepost generation strategy and the most cost-effective multi-value guidepost generation strategy, we use the selected two guidepost generation strategies as a representative DEMINER technique for RQ3 to RQ6.

RQ3 is to check the coverage improvement of DEMINER over the coverage achieved by the given test suites.

RQ4 is to check how effectively the guideposts guide concolic testing and fuzzing on p to cover the target unreached lines/branches covered by the coverage-increasing mutant executions.

RQ5 is to compare DEMINER with the conventional concolic testing and fuzzing techniques. The conventional concolic testing techniques use CROWN [20] with three search strategies (i.e., DFS, RND, and control-graph based heuristic for fast branch coverage increase (CFG) [21]) as well as KLEE [22] with the interleaved search strategy of the random path search and the coverage-guided search used by Cadar et al. [22]. The conventional fuzzing technique uses AFL [18] and T-Fuzz [3].

RQ6 evaluates the efficiency and the effectiveness of the greedy mutant selection method of DEMINER (see Section 3.2). We compared DEMINER with a variant that randomly selects the same number of mutants selected by DEMINER and another variant that selects all mutants.

To answer RQ1 to RQ6, we performed the experiments on the six real-world C programs shown in Table II. The following subsections explain the details of the experiment setup.

DOI: 10.1002/stvr

Table II. Study objects

Program	Lines	Branches	Num. TCs	Covered lines	Covered branches
`Busybox-ls`	404	303	6	257 (63.6%)	135 (44.6%)
`Busybox-printf`	169	105	17	140 (82.8%)	75 (71.4%)
`Coreutils-sort`	1811	1276	168	1607 (88.7%)	901 (70.6%)
`GNU-find`	3616	2091	120	2192 (60.6%)	1061 (50.7%)
`GNU-grep`	6832	4763	176	5062 (74.1%)	2817 (59.1%)
`GNU-sed`	5614	3981	86	4317 (76.9%)	2476 (62.2%)

4.1. Study Objects

We used recent versions of three real-world C programs as study objects. `Busybox-ls` and `Busybox-printf` are a file listing utility and a formatted data printer in BusyBox version 1.24.0 [*] respectively. `Coreutils-sort` is a text line sort program in Coreutils version 8.30 [†]. `GNU-find` is a file search utility in GNU FindUtils version 4.6 [‡]. `GNU-grep` and `GNU-sed` are a text pattern matching program and a stream editor, respectively. We use `GNU-grep` version 3.1 [§] and `GNU-sed` version 4.5 [¶]. These six programs are utilities for UNIX-like operating systems. Table II shows the size of the target code in executable lines (LoC) and branches, a 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 the branch coverage throughout the experiments.

4.2. Mutant Generation Setup

DEMINER generates mutants of a target program using a C source code mutation tool MUSIC [23]. After mutant generation, DEMINER eliminates trivially equivalent and duplicated mutants [24]. An equivalent mutant is identified by checking whether or not the MD5 checksum of the compiled binary object is the same to 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`, `Busybox-printf`, and `Coreutils-sort`. To save experiment time, the experiments with `GNU-find`, `GNU-grep`, and `GNU-sed` randomly select and use five mutants per code line, because the total amount of time spent for the experiments on `GNU-find`, `GNU-grep`, and `GNU-sed` will be significantly large.

[*] https://busybox.net
[†] https://www.gnu.org/software/coreutils
[‡] https://www.gnu.org/software/findutils
[§] https://www.gnu.org/software/grep
[¶] https://www.gnu.org/software/sed
[‖] Nine test cases were excluded as they re-execute the main routine multiple times, which complicates concolic testing with seeding initial test input (see Section 4.3). Also, to save testing time, two test cases were excluded since they consume more than 15 times longer execution times than the other test cases, while the two test cases do not increase the total line/branch coverage.

DOI: 10.1002/stvr

Table III. Symbolic input setup for the study objects

Program	Max. # sym-args	Max. len. sym-arg	Max. # sym. file-metadata	Max. len. sym-file
`Busybox-ls`	4	6	5×13	-
`Busybox-printf`	7	11	-	-
`Coreutils-sort`	4	15	-	2498
`GNU-find`	6	10	6×13	-
`GNU-grep`	4	33	-	5495
`GNU-sed`	5	28	-	49686

4.3. Concolic Test Generation Setup

We declared command-line arguments and file-metadata such as file mode, file size, permission, modification time, and file contents 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. The third column shows the maximum length of each symbolic command-line argument of the experiment setup. The fourth column shows the maximum number of symbolic file-metadata structures in the experiments (only `Busybox-ls` and `GNU-find` use such file-metadata). The number of the symbolic file-metadata structures is same to the number of files used in a given regression test input. Each symbolic file-metadata consists of 13 symbolic integer variables (e.g., `st_mode` and `st_size` in `struct stat`). The last column shows the maximum size of symbolic file in bytes which is same to the maximum size of a file included in initial seed test inputs (`Busybox-ls`, `Busybox-printf` and `GNU-find` do not read file contents).

For each pair of a generated guidepost g_i and a given test input t_j, DEMINER applies concolic testing (i.e., DEMINER-CT) to a target program having g_i with t_j as an initial seed test input for generating 500 test inputs further (we selected to generate 500 test inputs because coverage increase saturates around 1,000 test inputs (see Figure 6)).

In addition, we compared DEMINER-CT with four conventional concolic testing techniques (i.e., CROWN [20] with three different search strategies: DFS, RND, and CFG (a greedy strategy for branch coverage based on a control flow graph of a target program) as well as KLEE with the interleaved search strategy of the random path search and the coverage-guided search used by Cadar et al. [22]). Conventional concolic testing techniques also use the given test inputs as initial seed test inputs. For fair comparison, we run CROWN and KLEE for the same amount of time that DEMINER-CT consumes which includes the followings:

1. mutant generation and selection time
2. mutant executions time with the given test inputs
3. guidepost construction time
4. guided concolic testing time for 500 test inputs per guidepost and initial test input.

For example of `Busybox-ls`, as DEMINER-CT spent total 26,644 seconds, each conventional concolic testing technique is executed for 26,644 seconds with the six initial test inputs (i.e., for each initial test input, a conventional technique runs for 4,440 seconds (=26,644/6)).

4.4. Fuzzing Test Generation Setup

We set the command-line arguments and file-metadata such as file mode, file size, permission, modification time, and file contents as inputs for fuzzing test generation as we did for concolic testing (see Section 4.3). AFL and T-Fuzz mutate command-line arguments and file contents, but not file-metadata. To generate diverse file-metadata using fuzzing, we created a new file f_m whose content is file-metadata of a target file f. Then, we modified the `Busybox-ls` and `GNU-find` to read f_m instead of f and set AFL and T-Fuzz to mutate f_m.

DOI: 10.1002/stvr

For each pair of a generated guidepost g_i and a given test input t_j, DEMINER applies weighted fuzzing (i.e., DEMINER-FUZZ) to a target program having g_i with t_j as an initial seed test input and generate test inputs that cover 500 *unique* execution paths as DEMINER-CT does. Note that fuzzing may generate many redundant test inputs that cover the same path while Concolic testing algorithm of DEMINER-CT guarantees that a newly generated test input explores a new execution path (see Section 3.4.1).

In addition, we compared DEMINER-FUZZ with the conventional fuzzing techniques (i.e., the AFL fuzzer and T-Fuzz without guideposts). The conventional fuzzing techniques also use the given test inputs as initial seed test inputs. For fair comparison, we run the conventional fuzzing techniques for the same amount of time that DEMINER-FUZZ consumes which includes the followings:

1. mutant generation and selection time
2. mutant executions time with the given test inputs
3. guidepost construction time
4. guided fuzzing for 500 test inputs each of which covers unique execution path per guidepost and initial test input.

For example of `Busybox-ls`, as DEMINER-FUZZ spent total 9,722 seconds, the conventional fuzzing techniques are executed for 9,722 seconds with the six initial test cases (i.e., for each initial test input, the conventional fuzzing technique runs for 1,620 seconds (=9722/6)).

4.5. 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 larger than the average execution time of each regression test execution time.

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

We used `gcov` to measure line and branch coverage. Note that covered line/branch information of a generated mutant is comparable to that of the original target program, since DEMINER uses only expression level mutation operators (Section 3.2) and MUSIC performs *line-preserving* mutation (Section 4.6).

4.6. Implementation

The DEMINER implementation is written in C++ and Python. The component for mutation analyses (i.e., in Phase 1, see Section 3.2) consists of the mutant generation part and the mutant execution part. For mutant generation, we used MUSIC (MUtation analySIs tool with high Configurability and extensibility) [23]. * MUSIC implements 73 expression-level and statement-level mutation operators for modern C programs (63 of them are defined in Agrawal et al. [17]). 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 2,130 lines of C++ code using Clang/LLVM 6.0 [27]. The test generation components for guided concolic testing and fuzzing (i.e., Phase 3) are implemented upon CROWN [20] and AFL [18], respectively. 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 [28]. It supports complex C features such as bitwise operators, floating point arithmetic, bitfields and so on. We

*We failed to use Proteum [25] and MILU [26] for the experiments. Proteum (last release on 2001) does not recognize C99 standard and often fails to parse target C programs. MILU also frequently generated uncompilable mutants from the target C programs.

DOI: 10.1002/stvr

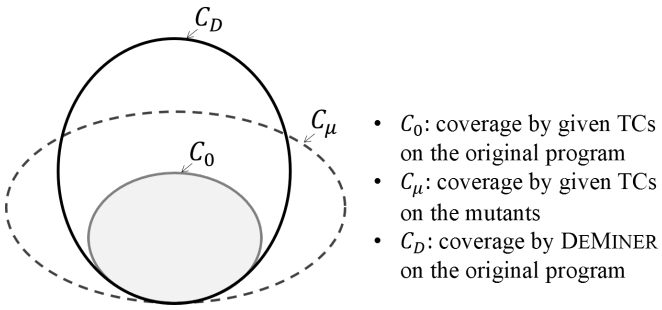


Figure 5. Diagrams showing the relation among coverages by regression test cases (C_0), mutant executions (C_μ), and DEMINER (C_D)

Table IV. The number of the lines and branches covered by the mutant executions

Program	Total mutants	Cov-incr. mutants	# of covered lines (line cov.%)	# of covered branches (br. cov.%)
`Busybox-ls`	17313	1280	363 (89.9%)	247 (81.5%)
`Busybox-printf`	5728	118	163 (96.4%)	99 (94.3%)
`Coreutils-sort`	30673	1389	1698 (93.8%)	1081 (84.7%)
`GNU-find`	5501	172	2505 (69.3%)	1257 (60.1%)
`GNU-grep`	24389	916	5414 (79.2%)	3104 (65.2%)
`GNU-sed`	17435	791	4576 (81.5%)	2735 (68.7%)

added 230 lines of C/C++ code to implement the prioritized search strategy which hybridizes DFS and RND (see Section 3.4.1). AFL is a coverage-based fuzzer which implements a genetic algorithm to evolve test inputs to increase code coverage. We add 130 lines of C code to implement the guided genetic algorithm to to satisfy guideposts (see Section 3.4.2).

4.7. 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 six C programs. We believe that this threat is limited because the target programs are non-trivial real-world C programs which have different characteristics. Another external threat involves the representativeness of the conventional concolic testing and fuzzing techniques that we compared with DEMINER. We think that this threat is also limited because these three search strategies are representative ones for concolic testing and AFL is a widely used fuzzing technique.

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

5. EXPERIMENT RESULTS

This sections describes the results of the experiments to answer the research questions in Section 4. Note that, in the following subsections, we denote the sets of lines/branches covered by the given test cases as C_0, and that covered by all mutant executions as C_μ, and that covered by DEMINER as C_D. Figure 5 shows the relation among C_0, C_μ, and C_D.

DOI: 10.1002/stvr

5.1. *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 (increasing line coverage and branch coverage by 5.7% to 41.2% and 10.2% to 83.0%, respectively). In other words, many unreached lines and branches are covered on the mutants with the given test cases (i.e., $|C_\mu - C_0| >> 0$ 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 coverage-increasing mutants. 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 table shows that, with the same given test cases, the executions on the 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 on the original program). The mutant executions increase line coverage by 5.7% (=(1698-1607)/1607)(`Coreutils-sort`) to 41.2% (=(363-257)/257)(`Busybox-ls`). Also, the mutant executions improve branch coverage by 10.2% (=(3104-2817)/2817) (`GNU-grep`) to 83.0% (=(247-135)/135)(`Busybox-ls`). Note that these coverage increments appear on the mutants, *not* on the original target program.

> Mutant executions with the given test suites generate diverse program executions that cover unreached lines/branches. Mutant executions with the given test suites increase line/branch coverage by 15.1% and 29.0% on average, respectively, compared to the original program executions with the given test suites.

5.2. *RQ2. To what extent do the guidepost generation strategies of* DEMINER *affect line/branch coverage and execution time?*

The experiment results on RQ2 show that the guidepost generation strategies significantly affect the line/branch coverage and the execution time of DEMINER-CT and DEMINER-FUZZ. Thus, the guidepost generation strategy is important for effectiveness and efficiency of DEMINER. We choose single-value exclusion (Sx) and multi-value range (Mrange) as representative DEMINER guidepost generation strategy. This is because Sx and Mrange are the most cost-effective single-value and multi-value guidepost generation strategies in terms of both line coverage and branch coverage, respectively.

Table V shows the number and the ratio of unreached lines covered by DEMINER-CT with the five guidepost generation strategies. For each guidepost generation strategy, we report the number of unreached lines covered by DEMINER-CT (#New cov. lines columns), the ratio of the number of the unreached lines newly covered by DEMINER-CT using one of the five guidepost generation strategies over the number of the unreached lines newly covered by DEMINER-CT using all of the five guidepost generation strategies, the amount of time spent to cover new lines (in hours), and the ratio of the unreached lines covered by DEMINER-CT per hour.

For example of `Busybox-ls` (shortened to `BB-ls` in Table V), DEMINER-CT with the single-value equality guidepost generation strategy (Seq) covers 78 unreached lines in 62.1 hours. In other words, DEMINER-CT with Seq covers 82.1% of the total number of the newly covered lines (95 lines) by DEMINER-CT with all the five guidepost generation strategies and DEMINER-CT with Seq covers 1.3% of the newly covered lines per hour.

We compare the ratio of the newly covered lines per hour to select the most cost-effective guidepost generation strategy among the single-value guidepost generation strategies and the multi-value guidepost generation strategies.

- For the single-value guidepost generation strategy, Sx is more cost-effective than Seq (6.6%/h vs 1.5%/h), on average.
- For the multi-value guidepost generation strategy, Mrange is the most cost-effective strategy (17.9%/h vs. 10.2%/h).

DOI: 10.1002/stvr

Table V. The number of the unreached lines covered by and the execution time of DEMINER-CT with the different guidepost generation strategies (Ratio columns show the ratio of the number of unreached lines covered by each guidepost generation strategy over the number of unreached lines covered by using all of the five guidepost generation strategies (last column))

Program	Seq			Sx			Meq			Mcov			Mrange			Total
	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines
BB-ls	78	82.1%	62.1	1.3%	76	80.0%	14.8	5.4%	58	61.1%	2.9	20.9%	82	86.3%	2.7	31.7%	74	77.9%	2.2	35.9%	95
BB-printf	15	65.2%	9.3	7.0%	14	60.9%	1.9	32.8%	11	47.8%	1.2	38.7%	18	78.3%	1.3	59.8%	18	78.3%	1.1	68.4%	23
Core-sort	81	71.7%	6447.9	0.0%	94	83.2%	1095.8	0.1%	39	34.5%	351.8	0.1%	66	58.4%	288.3	0.2%	87	77.0%	421.4	0.2%	113
GNU-find	139	56.7%	361.9	0.2%	122	49.8%	70.9	0.7%	62	25.3%	63.8	0.4%	183	74.7%	40.7	1.8%	215	87.8%	55.7	1.6%	245
GNU-grep	361	74.3%	1335.8	0.1%	334	68.7%	266.9	0.3%	381	78.4%	161.8	0.5%	396	81.5%	102.2	0.8%	412	84.8%	199.7	0.4%	486
GNU-sed	271	83.9%	593.0	0.1%	255	78.9%	132.0	0.6%	228	70.6%	75.5	0.9%	244	75.5%	49.6	1.5%	256	79.3%	89.7	0.9%	323
Average	157.5	72.3%	1468.3	1.5%	149.2	70.3%	263.7	6.6%	129.8	52.9%	109.5	10.2%	164.8	75.8%	80.8	16.0%	177.0	80.8%	128.3	17.9%	214.2

Table VI. The number of the unreached lines covered by and the execution time of DEMINER-FUZZ with the different guidepost generation strategies (Ratio columns show the ratio of the number of unreached lines covered by each guidepost generation strategy over the number of unreached lines covered by using all of the five guidepost generation strategies (last column))

Program	Seq			Sx			Meq			Mcov			Mrange			Total
	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines	Ratio	Time (h)	Ratio /h	#New cov. lines
BB-ls	94	88.7%	23.1	3.8%	94	88.7%	5.1	17.5%	94	88.7%	1.0	88.7%	93	87.7%	0.9	96.3%	98	92.5%	1.1	83.7%	106
BB-printf	21	91.3%	5.0	18.1%	21	91.3%	1.0	87.2%	20	87.0%	0.7	128.2%	20	87.0%	0.7	128.2%	22	95.7%	0.6	168.8%	23
Core-sort	98	86.7%	756.3	0.1%	103	91.2%	134.8	0.7%	91	80.5%	38.0	2.1%	96	85.0%	33.5	2.5%	99	87.6%	28.9	3.0%	113
GNU-find	162	61.4%	247.2	0.2%	188	71.2%	49.6	1.4%	207	78.4%	34.5	2.3%	220	83.3%	28.1	3.0%	229	86.7%	30.0	2.9%	264
GNU-grep	462	84.3%	650.8	0.1%	493	90.0%	152.7	0.6%	405	73.9%	88.8	0.8%	429	78.3%	55.7	1.4%	482	88.0%	86.0	1.0%	548
GNU-sed	262	67.4%	376.2	0.2%	291	74.8%	93.6	0.8%	256	65.8%	51.7	1.3%	309	79.4%	29.1	2.7%	329	84.6%	40.7	2.1%	389
Average	183.2	80.0%	343.1	3.8%	198.3	84.5%	72.8	18.0%	178.8	79.0%	35.8	37.2%	194.5	83.5%	24.7	39.0%	209.8	89.2%	31.2	43.6%	240.5

We do not report the branch coverage results of the five guidepost generation strategies since they are similar to the line coverage results (i.e., Sx and Mrange are the most cost-effective guidepost generation strategies).

Table VI shows the number and the ratio of the unreached lines covered by DEMINER-FUZZ with the five guidepost generation strategies. The structure of the table is same to Table V. For DEMINER-FUZZ, Sx and Mrange are the most cost-effective single-value and multi-value guidepost generation strategies as for DEMINER-CT, respectively. From now on, the experiment results of DEMINER are the ones obtained by using the Sx and Mrange guidepost generation strategies.

Note that we choose only one of the single-value guidepost generation strategies and one of the multi-value guidepost generation strategies. This is because choosing multiple strategies among the single-value strategies and multiple strategies among the multi-value strategies is not cost-effective to increase line coverage as the single-value guidepost generation strategies have large overlap of the covered lines and the multi-values guidepost generation strategies also have large overlap. The overlap ratio of the lines covered by DEMINER-CT with Seq and DEMINER-CT with Sx is 98.2% on average. The overlap ratio of the lines covered by DEMINER-CT with multi-values guidepost generation strategies is 95.4% (Mcov and Mrange) to 97.9% (Meq and Mrange). Similarly, the overlap ratio of the lines covered by DEMINER-FUZZ with Seq and Sx is 97.9% on average and the overlap ratio of the lines covered by DEMINER-FUZZ using multi-values guidepost generation strategies is 94.6% (Mcov and Mrange) to 98.3% (Meq and Mrange).

Table VII shows the number of the guideposts generated by the guidepost generation strategies. The second and third columns show the number of generated guideposts with single-value equality and single-value exclusion guidepost generate strategies, respectively. The fourth, fifth, and sixth columns show the number of the generated guideposts with multi-values equality, multi-values per coverage, and multi-values range, respectively. Seq generates the largest number of the guideposts

DOI: 10.1002/stvr

Table VII. The number of the guideposts generated by the guidepost generation strategies

Program	Seq	Sx	Meq	Mcov	Mrange
`Busybox-ls`	2959	690	128	119	125
`Busybox-printf`	163	30	20	20	19
`Coreutils-sort`	1327	228	92	101	98
`GNU-find`	413	76	62	47	57
`GNU-grep`	687	155	91	60	97
`GNU-sed`	591	147	85	53	85

Table VIII. The number of covered lines and branches by DEMINER (using Sx and Mrange)

Program	Cov	DEMINER-CT		DEMINER-FUZZ	
		# of unreached lines and br. covered (%cov.)	# of all lines and br. covered (%cov.)	# of unreached lines and br. covered (%cov.)	# of all lines and br. covered (%cov.)
`Busybox-ls`	Line	94 (23.3%)	351 (86.9%)	106 (26.2%)	363 (89.9%)
	Br.	96 (31.7%)	231 (76.2%)	104 (34.3%)	239 (78.9%)
`Busybox-printf`	Line	21 (12.4%)	161 (95.3%)	23 (13.6%)	163 (96.4%)
	Br.	18 (17.1%)	93 (88.6%)	23 (21.9%)	98 (93.3%)
`Coreutils-sort`	Line	96 (5.3%)	1703 (94.0%)	113 (6.2%)	1720 (95.0%)
	Br.	206 (16.1%)	1107 (86.8%)	228 (17.9%)	1129 (88.5%)
`GNU-find`	Line	266 (7.4%)	2458 (68.0%)	252 (7.0%)	2444 (67.6%)
	Br.	158 (7.6%)	1219 (58.3%)	169 (8.1%)	1230 (58.8%)
`GNU-grep`	Line	450 (6.6%)	5512 (80.7%)	516 (7.6%)	5578 (81.6%)
	Br.	316 (6.6%)	3133 (65.8%)	363 (7.6%)	3180 (66.8%)
`GNU-sed`	Line	299 (5.3%)	4616 (82.2%)	341 (6.1%)	4658 (83.0%)
	Br.	241 (6.1%)	2717 (68.2%)	275 (6.9%)	2751 (69.1%)
Average	Line	204.3 (10.0%)	2466.8 (84.5%)	225.2 (11.1%)	2487.7 (85.6%)
	Br.	172.5 (14.2%)	1416.7 (74.0%)	193.7 (16.1%)	1437.8 (75.9%)

because Seq strategy generates one guidepost for each monitored value from the coverage-increasing mutant executions. As the number of the generated guideposts increases, the execution time of DEMINER also increases because DEMINER runs concolic testing or fuzzing for each guidepost-inserted program and an initial seed test.

The guidepost generation strategies significantly affect test coverage and execution time of DEMINER-CT and DEMINER-FUZZ. For both of the DEMINER-CT and DEMINER-FUZZ, single-value exclusion guidepost (Sx) and multi-value range guidepost (Mrange) are the most effective single-value and multi-value guidepost generation strategy, respectively. DEMINER-CT and DEMINER-FUZZ use Sx and Mrange as their guidepost generation strategies.

5.3. RQ3. How many unreached lines/branches of an original program are covered by DEMINER*?*

The experiment results on RQ3 show that DEMINER effectively increases test coverage by utilizing the knowledge on the mutant executions.

In Table VIII, the third column represents numbers of the unreached lines and branches covered by the 500 test inputs generated by DEMINER-CT (using the Sx and Mrange guidepost generation strategies) for each pair of a guidepost and an initial test input. The fourth column represents numbers of all lines and branches covered by the 500 test inputs generated by DEMINER-CT. Similarly, the fifth column represents numbers of the unreached lines and branches covered by the 500 unique executions paths explored by DEMINER-FUZZ and the sixth column represents numbers of all lines and branches covered by DEMINER-FUZZ for each pair of a guidepost and an initial test case (see Section 4.4).

DOI: 10.1002/stvr

Table IX. Coverage increase by the mutant executions and DEMINER

Program	Cov	$\lvert C_\mu - C_0\rvert$	DEMINER-CT			DEMINER-FUZZ		
			$\lvert C_D - C_0\rvert$	$\lvert(C_D - C_0)\cap (C_\mu - C_0)\rvert$	$\lvert(C_D - C_0)\cap (C_\mu - C_0)\rvert / \lvert(C_\mu - C_0)\rvert$	$\lvert C_D - C_0\rvert$	$\lvert(C_D - C_0)\cap (C_\mu - C_0)\rvert$	$\lvert(C_D - C_0)\cap (C_\mu - C_0)\rvert / \lvert(C_\mu - C_0)\rvert$
`Busybox-ls`	Line	106	94	94	88.7%	106	106	100.0%
	Br.	112	96	96	85.7%	104	104	92.9%
`Busybox-printf`	Line	23	21	21	91.3%	23	22	95.7%
	Br.	24	18	18	75.0%	23	23	95.8%
`Coreutils-sort`	Line	91	96	89	97.8%	113	88	96.7%
	Br.	180	206	169	93.9%	228	168	93.3%
`GNU-find`	Line	313	266	266	85.0%	252	211	67.4%
	Br.	196	158	158	80.6%	169	142	72.4%
`GNU-grep`	Line	352	450	332	94.3%	516	341	96.9%
	Br.	287	316	276	96.2%	363	233	81.2%
`GNU-sed`	Line	259	299	247	95.4%	341	219	84.6%
	Br.	229	241	218	95.2%	275	202	88.2%
Average	Line	190.7	204.3	174.8	92.1%	225.2	164.5	90.2%
	Br.	171.3	172.5	155.8	87.8%	193.7	145.3	87.3%

For example of `Busybox-ls`, DEMINER-CT increases coverage by 36.6% (=94 /257). Also, for `Busybox-ls`, DEMINER-FUZZ increases line coverage by 41.2% (=106/257). DEMINER-CT and DEMINER-FUZZ increased line coverage by 10.0% and 11.1% on average, respectively. Also, DEMINER-CT and DEMINER-FUZZ increases branch coverage by 14.2% and 16.1% on average, respectively. DEMINER-CT and DEMINER-FUZZ spend 199.9 and 104.0 hours on average, respectively (see the sixth and twelfth columns of Table XI for details).

DEMINER-CT and DEMINER-FUZZ effectively increased test coverage by using the guideposts. DEMINER-CT and DEMINER-FUZZ achieved 84.5% and 85.6% line coverage by increasing line coverage by 10.0%p and 11.1%p, on average, respectively. DEMINER-CT and DEMINER-FUZZ achieved 74.0% and 75.9% branch coverage by increasing branch coverage by 14.2%p and 16.1%p, on average, respectively.

5.4. *RQ4. How many unreached lines/branches covered by the mutant executions are also covered by* DEMINER*?*

The experiment results on RQ4 confirm that the guideposts generated from the coverage increasing mutant executions effectively guide test generation to cover most of the target unreached lines and branches covered by the mutant executions.

Table IX compares the coverage of the mutant executions and DEMINER. The third column shows a number of the unreached lines/branches covered by the mutant executions (i.e., $|C_\mu - C_0|$). The fourth and fifth columns show a number of the unreached lines/branches covered by DEMINER-CT (i.e., $|C_D - C_0|$) and by both mutant executions and DEMINER-CT (i.e., $|(C_D - C_0) \cap (C_\mu - C_0)|$), respectively. Similarly, the sixth and seventh columns show a number of the unreached lines/branches covered by DEMINER-FUZZ and by both mutant executions and DEMINER-FUZZ, respectively.

For example of `Busybox-ls`, DEMINER-CT covers 88.7% (= $\frac{|(C_D - C_0)\cap(C_\mu - C_0)|}{|C_\mu - C_0|}$=94/106) of the unreached lines covered by the mutant executions (see the third and fifth columns). Also, for `Busybox-ls`, DEMINER-FUZZ covers 100.0% (=106/106) of the unreached lines covered by the mutant executions of `Busybox-ls`. On average, DEMINER-CT and DEMINER-FUZZ cover 92.1% and 90.2% of the unreached lines covered by the mutant executions, respectively. Similarly, DEMINER-CT and DEMINER-FUZZ cover 87.8% and 87.3% of the unreached branches covered by mutant executions, respectively.

Note that for `Coreutils-sort`, `GNU-grep`, and `GNU-sed`, DEMINER-CT and DEMINER-FUZZ cover the lines and branches which are covered neither by the initial test cases nor by the mutant executions. For example of `GNU-sed`, DEMINER-CT covers 40 more new lines (= 299-259 = $|C_D - C_0| - |C_\mu - C_0|$) than the mutant executions. In addition, DEMINER-FUZZ cover

DOI: 10.1002/stvr

Table X. The number of the unreached lines and branches covered by the conventional concolic testing and DEMINER

Program	Cov	Conventional concolic				DEMINER -CT	Conventional fuzzing		DEMINER -FUZZ
		CROWN			KLEE		AFL	T-Fuzz	
		DFS	RND	CFG					
Busybox-ls	Line	80	75	74	83	94	82	101	106
	Br.	72	78	77	84	96	88	97	104
Busybox-printf	Line	18	18	19	19	21	23	23	23
	Br.	14	14	15	15	18	23	23	23
Coreutils-sort	Line	72	76	76	74	96	67	91	113
	Br.	130	130	126	132	206	131	193	228
GNU-find	Line	196	218	203	205	266	134	221	252
	Br.	79	85	83	82	158	89	137	169
GNU-grep	Line	344	349	338	332	450	232	313	516
	Br.	174	165	165	152	316	149	218	363
GNU-sed	Line	236	234	222	228	299	311	331	341
	Br.	125	125	128	123	241	255	268	275

such lines and branches for Busybox-printf and GNU-find. This fresh coverage increase is because DEMINER can explore diverse execution space beyond the ones reached by the mutant executions.

The guideposts effectively guide test generation to cover the target unreached lines and branches covered by the mutant executions. DEMINER-CT covers 92.1% of the lines and 87.8% of the branches covered by the mutant executions. DEMINER-FUZZ covers 90.2% of the lines and 87.3% of the branches covered by the mutant executions.

5.5. *RQ5. How many unreached lines/branches of an original program does* DEMINER *cover, compared to the conventional concolic testing and fuzzing techniques?*

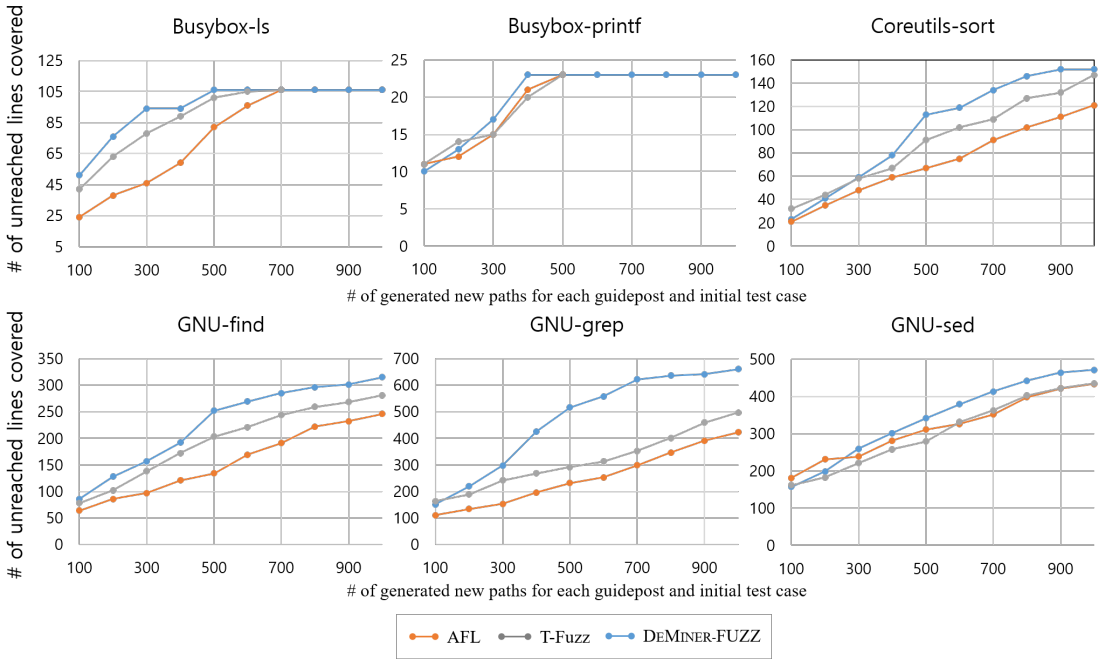
The experiment results show that DEMINER-CT covers more lines and branches than all four studied concolic testing techniques (i.e., CROWN with DFS, RND, and CFG search strategies as well as KLEE) when it generates 500 test inputs (see Section 4.3). Also, DEMINER-FUZZ covers more lines and branches than the conventional fuzzing techniques (i.e., AFL and T-Fuzz) when it explores 500 unique execution paths (see Section 4.4).

Table X compares the coverage achievements of the conventional test generation techniques with that of DEMINER-CT and DEMINER-FUZZ. The third to fifth columns show the numbers of unreached lines/branches covered by CROWN with DFS, RND, and CFG search strategies, respectively. The sixth column shows the result of KLEE. The seventh column shows the result of DEMINER-CT. The eighth and ninth columns report the number of the unreached lines and branches covered by the conventional fuzzing techniques, AFL and T-Fuzz, respectively. The last column show the result of DEMINER-FUZZ.

For example of Busybox-ls, DEMINER-CT covered 94 unreached lines which are 17.5% (=(94-80)/80) more than the ones covered by the conventional concolic testing using DFS (the best conventional concolic testing technique). Similarly, Busybox-ls, DEMINER-FUZZ covered 106 unreached lines which are 29.3% (=(106-82)/82) more than the ones covered by the conventional fuzzing technique AFL. For all six target programs, DEMINER-CT outperformed the four concolic testing techniques by covering 24.5% and 63.4% more lines and branches than CROWN with RND which is the best conventional concolic testing technique on average, respectively. DEMINER-FUZZ also outperformed T-Fuzz which is the best conventional fuzzing technique by covering 18.5% and 19.6% more lines and branches on average, respectively.

In addition, Figure 6 shows the number of the unreached lines covered by the four conventional techniques and DEMINER-CT using up to 1,000 test inputs. The X-axis represents a number of test inputs generated by DEMINER-CT for each guided concolic testing instance (i.e., concolic testing of a target program with one guidepost and one initial test case). The Y-axis represents

DOI: 10.1002/stvr


Busybox-ls
Busybox-printf
Coreutils-sort
GNU-find
GNU-grep
GNU-sed
# of unreached lines covered
# of generated new paths for each guidepost and initial test case
AFL
T-Fuzz
DeMiner-FUZZ

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 `GNU-find` graph in the figure).

Similar to Figure 6, Figure 7 shows the number of the unreached lines covered by the conventional fuzzer and DEMINER-FUZZ exploring up to 1,000 unique execution paths. The result shows that, for all target programs and all levels of the numbers of the explored unique execution paths (except 100 and 200 unique explored execution paths), DEMINER-FUZZ always covers more unreached lines than the conventional fuzzing techniques.

> DEMINER-CT and DEMINER-FUZZ cover more lines and branches than the conventional concolic testing and fuzzing techniques, respectively. DEMINER-CT covers 26.4% more lines and 63.4% more branches than the best of conventional concolic testing technique (CROWN with RND). DEMINER-FUZZ covers 18.5% more lines and 19.6% more branches than the best of the conventional fuzzing technique (T-Fuzz).

5.6. *RQ6. To what extent does the mutant selection of* DEMINER *affect execution time and line/branch coverage?*

The experiment results show that the greedy selection of the mutants effectively reduces the runtime cost without hurting coverage much for DEMINER-CT and DEMINER-FUZZ.

Table XI shows the effects of the coverage-increasing mutant selection. The third to fifth columns show the line and branch coverage that DEMINER-CT 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 sixth to eighth columns show the execution time in hours spent by DEMINER-CT with the corresponding methods. Similarly, the ninth to the 11th columns show the line and branch coverage that DEMINER-FUZZ achieves with the mutant selection methods. The 12th to the last columns show the execution time in hours spent by DEMINER-FUZZ with the mutant selection methods.

DEMINER-CT with the greedy mutant selection consumes only 2.4% (=126.7/5379.5) (`GNU-find`) to 4.5% (=466.7/10316.2) (`GNU-grep`) of the execution time of DEMINER-CT with all coverage-increasing mutants. DEMINER-FUZZ with the greedy mutant selection consumes only 2.9% (=6.2/212.4) (`Busybox-ls`) to 6.5% (=1.6/24.9) (`Busybox-printf`) of the execution time of DEMINER-FUZZ with all coverage-increasing mutants.

Also, for all target programs, DEMINER-CT and DEMINER-FUZZ with the greedy selection covers more (or equal numbers of) lines and branches than DEMINER-CT and DEMINER-FUZZ with random selection, respectively. For example of `Busybox-ls`, DEMINER-CT with the greedy mutant selection covers 94 unreached lines while DEMINER-CT with the random selection covers only 86 lines.

Note that DEMINER-CT and DEMINER-FUZZ are easy-to-parallelize. Mutant executions and running concolic testing or fuzzing on a program with a guidepost can be distributed to multiple machines because they are independent from each other. For example of `GNU-grep`, using 80 CPU cores (20 machines each of which is equipped with 4 cores), DEMINER-CT and DEMINER-FUZZ spend only 5.8 and 3.0 hours, which are practical in industry.

> The greedy selection of coverage-increasing mutants reduce the execution time of DEMINER-CT and DEMINER-FUZZ by at least 95.5% and 93.5%, respectively. Also, the greedy selection covers more lines and branches than random selection of coverage-increasing mutants.

6. DISCUSSION

6.1. *Hard-to-reach Lines which* DEMINER*-CT can cover*

Figure 8 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

DOI: 10.1002/stvr

Table XI. Effect of the coverage-increasing mutant selection methods

Program	Cov	DeMiner-CT						DeMiner-FUZZ					
		# of unreached lines and branches covered			Time (in hour)			# of unreached lines and branches covered			Time (in hour)		
		Greedy	Rand.	All	Greedy	Rand.	All	Greedy	Rand.	All	Greedy	Rand.	All
Busybox-ls	Line	94	86	94	16.9	19.7	522.3	106	106	106	6.2	6.6	212.4
	Br.	96	91	96				104	104	104			
Busybox-printf	Line	21	19	21	3.0	4.8	69.6	23	23	23	1.6	2.1	24.9
	Br.	18	15	18				23	23	23			
Coreutils-sort	Line	96	88	99	364.3	351.6	8645.3	113	104	122	163.7	151.0	3746.8
	Br.	206	189	211				228	215	238			
GNU-find	Line	266	231	281	126.7	115.8	5379.5	252	213	288	79.6	92.1	2267.1
	Br.	158	106	173				169	143	198			
GNU-grep	Line	450	401	489	466.7	521.5	10316.2	516	518	581	238.7	244.1	5099.6
	Br.	316	253	335				363	365	413			
GNU-sed	Line	299	229	351	221.7	238.3	5319.2	341	319	411	134.3	112.3	2908.0
	Br.	241	208	298				275	245	317			

```
// format points to a character array whose
// elements are symbolic input characters
160 static void print_direc(char *format, ...){
...
168   // guide(format[fmt_length] == 'X')
169   ch = format[fmt_length]; //M1:fmt_length
...                            //-> precision
172   have_prec = strstr(format, ".*");
173   have_width = strchr(format, '*');
...
179   switch (ch) {
...
201   // hard-to-cover for conventional concolic
202   case 'X':
203     llv = my_xstrtoull(argument);
```

Figure 8. An example of hard-to-reach lines of `Busybox-printf` for conventional concolic testing

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 `ch` 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 3.0 hours, due to the loops inside `strstr` at Line 172 and `strchr` at Line 173. Both loops have symbolic variables in their loop conditions and they iterate over a symbolic input string `format` until they reach any specified character (i.e., '.', '*', or '\0'). Thus, conventional concolic testing keeps increasing the loop bound and fails to cover Lines 202–203 within a given time bound.

Note that DeMiner-CT covers these hard-to-cover lines as follows. DeMiner-CT generates a mutant m which mutates a variable `fmt_length` to another variable `precision`. One of the mutant executions of m covers Lines 202–203 and the monitoring probe for m (i.e., `probe(format[precision])` at right before Line 169) reports 'X' as a value of the mutated target expression. Using the reported value 'X', DeMiner-CT inserts a guidepost at Line 168 and DeMiner-CT tries to satisfy the guidepost condition with high priority instead of negating branches of the loops inside `strstr` and `strchr` at Lines 172–173 (see Section 3.4.1). As a result, DeMiner-CT effectively covers Lines 202–203.

Furthermore, for `GNU-find`, DeMiner-CT covers 16 lines that seem 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.,

DOI: 10.1002/stvr

Table XII. The numbers of unreached lines and branches covered by DEMINER-FUZZ with different ratios of DEMINER score to AFL score for selecting seed test inputs

Program	Cov	0:1	1:1	10:1	20:1
`Busybox-ls`	Line	106	106	106	106
	Br.	104	104	104	104
`Busybox-printf`	Line	23	23	23	23
	Br.	23	23	23	23
`Coreutils-sort`	Line	113	113	113	113
	Br.	228	228	228	228
`GNU-find`	Line	162	200	252	202
	Br.	110	147	169	145
`GNU-grep`	Line	322	424	516	458
	Br.	221	301	363	316
`GNU-sed`	Line	213	273	341	301
	Br.	173	205	275	245

execution time is 120 weeks in total). Then, we found that those 16 lines covered by DEMINER-CT were never covered by conventional concolic techniques with 120 weeks of the testing time. DEMINER-CT can cover these 16 lines successfully because the guideposts prune the search space that is not relevant to cover these new lines, and effectively guide concolic testing toward execution paths that reach unreached lines observed from diverse coverage-increasing mutant executions.

6.2. *Impact of Guidepost-guided Fuzzing Strategy on Coverage Achievement*

DEMINER-FUZZ effectively utilizes guideposts in a target program to guide test generations to achieve unreached lines and branches. Table XII presents the numbers of unreached lines and branches covered by DEMINER-FUZZ with different ratios of DEMINER score to AFL score for selecting seed test inputs (see Section 3.4.2). The third column shows the coverage results when the original AFL is used for test generation without counting DEMINER scores (i.e., the ratio of DEMINER score to AFL score is 0:1). The forth to the sixth columns show the coverage results that DEMINER-FUZZ counts DEMINER score with the equal weight to AFL score (1:1), with 10 time higher (10:1), and 20 time higher weights (20:1).

The results show that the guided test generation with high weight on DEMINER score (i.e., 10:1) effectively improves the line and branch coverage. For the three GNU utilities `GNU-find`, `GNU-grep`, and `GNU-sed`, DEMINER-FUZZ with the original AFL (i.e., 0:1) covers the least lines and branches. DEMINER-FUZZ shows the highest coverage achievements when the ratio is 10:1. For `Busybox-ls`, `Busybox-print`, and `Coreutils-sort`, different weights on DEMINER score do not affect coverage, since these programs are relatively small and even more focus on guideposts through higher weight on DEMINER score does not change the coverage results.

6.3. *Comparison of Unreached Lines Covered by* DEMINER-*CT and* DEMINER-*FUZZ*

We found that DEMINER-CT covers a different set of unreached lines from DEMINER-FUZZ, and vice versa. Thus, it is a good idea to apply these two test generation techniques together to maximize overall coverage achievement. In Table XIII, the third column shows the number of unreached lines that both DEMINER-CT and DEMINER-FUZZ covered. The fourth column presents the number of unreached lines that DEMINER-CT covers, but DEMINER-FUZZ does not. The fifth column shows the number of unreached lines that DEMINER-FUZZ covers but DEMINER-CT does not. For each target program, the #line row shows the number of covered lines and the %line row shows the ratio of the lines covered by corresponding techniques over the total covered lines by DEMINER-CT and/or DEMINER-FUZZ. For example of `GNU-find`, total 277 lines (=241+25+11) are covered by DEMINER-CT and/or DEMINER-FUZZ and among the 277 lines, 87.0% of lines (=241/277) are covered by both DEMINER-CT and DEMINER-FUZZ.

DOI: 10.1002/stvr

Table XIII. The number and ratio of unreached lines covered by DEMINER-CT and DEMINER-FUZZ

Program	Cov	Both DeMiner-CT and DeMiner-FUZZ	DeMiner-CT only	DeMiner-FUZZ only
`Busybox-ls`	#line	94	0	12
	%line	88.7%	0.0%	11.3%
`Busybox-printf`	#line	21	0	2
	%line	91.3%	0.0%	8.7%
`Coreutils-sort`	#line	96	0	17
	%line	85.0%	0.0%	15.0%
`GNU-find`	#line	241	25	11
	%line	87.0%	9.0%	4.0%
`GNU-grep`	#line	419	31	97
	%line	76.6%	5.7%	17.7%
`GNU-sed`	#line	261	38	80
	%line	68.9%	10.0%	21.1%
Average	#line	188.7	15.7	36.5
	%line	82.9%	4.1%	13.0%

For the three GNU target programs, on average 22.5% of unreached lines are covered by either DEMINER-CT (8.2%) or by DEMINER-FUZZ (14.3%) but not by both. For example of `GNU-find`, among the 277 lines covered by DEMINER-CT and/or DEMINER-FUZZ, DEMINER-CT exclusively covers 9.0% of lines (=25/277). For `GNU-grep` and `GNU-sed`, DEMINER-CT exclusively cover 5.7% and 10.0% of the total new covered lines, respectively. For the other three relatively small programs, `Busybox-ls`, `Busybox-printf`, and `Coreutils-sort`, DEMINER-FUZZ covers all lines covered by DEMINER-CT, because DEMINER-FUZZ achieves almost maximal coverage of the target programs.

7. RELATED WORK

7.1. 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 [29–31] mainly focuses on evaluating bug finding effectiveness of a test suite by measuring how many mutants the test suite can kill.

Fraser and Zeller [32] 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 differences and coverage differences between a target program and its mutants. The difference between DEMINER and Fraser and Zeller [32] is that DEMINER uses the dynamic information on mutants to infer *internal conditions (i.e., guideposts)* of a target program to increase test coverage, rather than measuring the output difference between the target program and a mutant (i.e., Fraser and Zeller [32]).

As mentioned by Papadakis et al. [1], recently a few researchers mutate program code or runtime states to explore various mutant behavior for fault localization [33–35], for automated program repairs [36], and for automated program improvements [37, 38]. There exist few test generation techniques [39, 40] that utilize runtime values and execution paths observed from mutated program executions. Unlike our approach, these techniques have a limitation that they may produce unsound analysis results because mutated program executions may be infeasible on the original program.

Also, program mutation has 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 [32] utilizes mutation analysis to infer test oracles from the mutant execution information. Jahangirova

DOI: 10.1002/stvr

et al. [41] presents a method to assess and improve the quality of a given test oracle by utilizing program mutation. In addition, mutation analysis has been used to precisely localize a fault in a program. Mutation-based fault localization [33–35, 42, 43] locates a fault in the target program code by observing how the behaviors of the faulty program change according to the program code mutation.

7.2. Concolic Testing

Several techniques have been proposed to increase code coverage of concolic testing using the cost-effective search strategy. For example, the coverage-guided search strategy in KLEE [22] computes a weight for each state, which is later used to select states to explore. The weight is obtained by considering how far the nearest uncovered instruction is located. Similarly, Burnim and Sen [21] proposed control-flow guided search strategy to negate a branch which is the nearest one to an uncovered branch. Context-guided search strategy [44] selects a branch to negate by excluding the branches whose contexts (i.e., a sequence of preceding branches) are already explored.

Shortest-distance symbolic execution [45] does not target coverage, but aims at identifying program inputs that execute a specific point in a program by measuring and minimizing distance of inputs to the target point. As a fitness function [46] to measure how close an explored path is to achieve the target test coverage, FITNEX [47] introduces a strategy for flipping branches in concolic testing that prioritizes paths likely closer to take a specific branch. Cha et al. [48, 49] propose a machine learning-based search strategy generation technique. This technique learns the features of the branches which lead to high code coverage when the branches are negated and generates a search strategy using the learned features of branches.

The main difference between the existing concolic testing search strategies and DEMINER is that DEMINER prioritizes concolic testing search strategies to satisfy the guideposts while the existing concolic testing search strategies give the same priority to the uncovered code coverage elements. Since the guideposts have rich information (learned from the diverse mutant executions) to increase code coverage, satisfying the guideposts can increase the code coverage more than the conventional concolic testing search strategies.

7.3. Fuzzing

Fuzzing [50, 51] has been developed to detect vulnerability of target programs. Fuzzers generate the test inputs by mutating existing seed test cases [18, 52, 53] or following the grammar of valid inputs given by a user [54, 55]. Due to the simplicity and scalability, the mutation-based solution is widely adopted in practice. Coverage-guided fuzzing techniques [18, 56–58] employ an evolving algorithm to drive fuzzers towards high code coverage. In a fuzzing loop, coverage-guided fuzzing techniques monitor the program execution to track code coverage as well as crashes or assert violations. Then, they select test inputs that contribute to code coverage and insert the test inputs into a seed test input pool for high code coverage. The difference between the coverage-guided fuzzing and DEMINER is that DEMINER identifies and constructs *guideposts* from diverse mutant executions which lead to high code coverage and guides fuzzers to generate test inputs that satisfies the guideposts (see Section 5.5).

T-Fuzz [3] uses program transformation to increase effectiveness of coverage-guided fuzzing. T-Fuzz removes the sanity check code that blocks the invalid inputs and runs the fuzzing to generate crashing inputs. When a crashing input is generated, T-Fuzz performs symbolic execution on the transformed program and the original program following the execution path of the crashing input to check the execution path is feasible in the original program. Although T-Fuzz removes only sanity-check code, DEMINER performs general program transformation using various mutation operators to explore diverse execution space. Also, unlike T-Fuzz, DEMINER does not generate any infeasible test inputs.

DOI: 10.1002/stvr

8. CONCLUSION

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

We plan to extend DEMINER in the following directions. First, to enhance the knowledge discovery, we will improve DEMINER 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 [59–62] which can generate the coverage-increasing conditions at an entry of a function. Third, we will leverage program invariant inference techniques, such as Daikon [63], to generate various kinds of guideposts from mutant execution information. Fourth, we will apply the proposed idea to enhance other automated test generation techniques such as method sequence generations [64, 65] and concurrent program testing [66]. Finally, we will apply DEMINER to more real-world programs to show that DEMINER is generally effective in finding unknown bugs in real-world programs.

ACKNOWLEDGMENTS

This research was supported by Next-Generation Information Computing Development Program (No. 2017M3C4A7068177 and No. 2017M3C4A7068179) and Basic Science Research Program (No. 2017R1C1B1008159, No. 2017R1D1A1B03035851, and No. 2019R1A2B5B01069865) through the National Research Foundation of Korea (NRF).

REFERENCES

1. Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M. Chapter six - mutation testing advances: An analysis and survey. *Advances in Computers*, vol. 112. Elsevier, 2019; 275 – 378.
2. Kim Y, Hong S, Ko B, Phan DL, Kim M. Invasive software testing: Mutating target programs to diversify test exploration for high test coverage. *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, 2018; 239–249.
3. Peng H, Shoshitaishvili Y, Payer M. T-fuzz: Fuzzing by program transformation. *2018 IEEE Symposium on Security and Privacy (SP)*, 2018; 697–710, doi:10.1109/SP.2018.00056.
4. Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.
5. Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, ACM: New York, NY, USA, 2005; 263–272.
6. Sen K, Agha G. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. *Proceedings of Computer Aided Verification (CAV)*, 2006.
7. Kim Y, Kim M. SCORE: a scalable concolic testing tool for reliable embedded software. *Proceedings of the Joint Meeting of European Software Engineering Conference and the ACM Symposium of Foundations of Software Engineering (ESEC/FSE)*, 2011.
8. Xiao X, Xie T, Tillmann N, de Halleux J. Precise identification of problems for structural test generation. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.
9. Godefroid P. Higher-order test generation. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.
10. Godefroid P, Taly A. Automated synthesis of symbolic instruction encodings from i/o samples. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.
11. Elkarablieh B, Godefroid R, Levin M. Precise pointer reasoning for dynamic test generation. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.
12. Trtík M, Strejček J. Symbolic memory with pointers. *Proceedings of the Automated Technology for Verification and Analysis (ATVA)*, 2014.
13. Romano A, Engler DR. symMMU: Symbolically executed runtime libraries for symbolic memory access. *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014.
14. Godefroid P, Luchaup D. Automatic partial loop summarization in dynamic test generation. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011.
15. Christakis M, Muller P, Wustholz V. Guiding dynamic symbolic execution toward unverified program executions. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016.

DOI: 10.1002/stvr

16. Alatawi E, Søndergaard H, Miller T. Leveraging abstract interpretation for efficient dynamic symbolic execution. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2017.
17. Agrawal H, DeMillo RA, Hathaway B, Hsu W, Hsu W, Krauser EW, Martin RJ, Mathur AP, Spafford E. Design of mutant operators for the C programming language. *Technical Report SERC-TR-120-P*, Purdue University 1989.
18. AFL: American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2018-12-29.
19. McMinn P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 2004; **14**(2):105–156, doi:10.1002/stvr.294. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294.
20. CROWN: Concolic testing for Real-wOrld softWare aNalysis. https://github.com/swtv-kaist/CROWN. Accessed: 2018-12-29.
21. Burnim J, Sen K. Heuristics for scalable dynamic test generation. *Technical Report UCB/EECS-2008-123*, EECS Department, University of California, Berkeley Sep 2008.
22. Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, USENIX Association: Berkeley, CA, USA, 2008; 209–224.
23. Phan DL, Kim Y, Kim M. MUSIC:Mutation Analysis Tool with High Configurability and Extensibility. *Proceedings of the International Workshop on Mutation Analysis (MUTATION)*, 2018.
24. Kintis M, Papadakis M, Jia Y, Malevris N, Traon YL, Harman M. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering* 2017; doi:10.1109/TSE.2017.2684805.
25. Maldonado JC, Delamaro ME, Fabbri SCPF, da Silva Simão A, Sugeta T, Vincenzi AMR, Masiero PC. Proteum: A family of tools to support specification and program testing based on mutation. *Mutation Testing for the New Century*, Wong WE (ed.). Springer US: Boston, MA, 2001; 113–116, doi:10.1007/978-1-4757-5939-6_19. URL https://doi.org/10.1007/978-1-4757-5939-6_19.
26. Jia Y, Harman M. MILU : A customizable, runtime-optimized higher order mutation testing tool for the full c language. *Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, 2008.
27. Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.
28. Kim Y, Kim M, Kim Y, Jang Y. Industrial application of concolic testing approach: A case study on libExif by using CREST-BV and KLEE. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.
29. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)* 2011; **37**(5):649–678.
30. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering (TSE)* Aug 2006; **32**(8):608–624.
31. Hong S, Staats M, Ahn J, Kim M, Rothermel G. Are concurrency coverage metrics effective for testing: A comprehensive empirical investigation. *Software Testing, Verification and Reliability (STVR)* Jun 2015; **25**(4):334–370.
32. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 2011; **38**:278–292, doi:doi.ieeecomputersociety.org/10.1109/TSE.2011.93.
33. Papadakis M, Traon YL. Metallaxis-FL: mutation-based fault localization. *Journal of Software Testing, Verification, and Reliability (STVR)* 2015; **25**(5-7):605–628.
34. Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2014.
35. Hong S, Kwak T, Lee B, Jeon Y, Ko B, Kim Y, Kim M. MUSEUM: Debugging real-world multilingual programs using mutation analysis. *Information and Software Technology (IST)* Feb 2017; **82**:80–95.
36. Le Goues C, Nguyen T, Forrest S, Weimer W. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* Jan 2012; **38**(1):54–72, doi:10.1109/TSE.2011.104. URL http://dx.doi.org/10.1109/TSE.2011.104.
37. Jia Y, Wu F, Harman M, Krinke J. Genetic improvement using higher order mutation. *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, 2015; 803–804.
38. Langdon WB, Lam BYH, Modat M, Petke J, Harman M. Genetic improvement of GPU software. *Genetic Programming and Evolvable Machines* 2017; **18**(1):5–44.
39. Zhang J, Lou Y, Zhang L, Hao D, Zhang L, Mei H. Isomorphic regression testing: Executing uncovered branches without test augmentation. *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016.
40. Jaygarl H, Kim S, Xie T, Chang CK. OCAT: Object capture-based automated testing. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2010.
41. Jahangirova G, Clark D, Harman M, Tonella P. Test oracle assessment and improvement. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2016.
42. Hong S, Lee B, Kwak T, Jeon Y, Ko B, Kim Y, Kim M. Mutation-based fault localization for real-world multilingual programs. *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2015.
43. Kim Y, Lam SM, Yoo S, Kim M. Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2019; To appear.
44. Seo H, Kim S. How we get there: A context-guided search strategy in concolic testing. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, ACM: New York, NY, USA, 2014; 413–424, doi:10.1145/2635868.2635872. URL http://doi.acm.org/10.1145/2635868.2635872.
45. Ma KK, Phang KY, Foster JS, Hicks M. Directed symbolic execution. *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, Springer-Verlag: Berlin, Heidelberg, 2011; 95–111. URL http://dl.

acm.org/citation.cfm?id=2041552.2041563.
46. Gross F, Fraser G, Zeller A. Search-based system testing: High coverage, no false alarms. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2012.
47. Xie T, Tillmann N, de Halleux J, Schulte W. Fitness-guided path exploration in dynamic symbolic execution. *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, 2009; 359–368, doi:10.1109/DSN.2009.5270315.
48. Cha S, Hong S, Lee J, Oh H. Automatically generating search heuristics for concolic testing. *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, ACM: New York, NY, USA, 2018; 1244–1254, doi:10.1145/3180155.3180166. URL http://doi.acm.org/10.1145/3180155.3180166.
49. Cha S, Lee S, Oh H. Template-guided concolic testing via online learning. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, ACM: New York, NY, USA, 2018; 408–418, doi:10.1145/3238147.3238227. URL http://doi.acm.org/10.1145/3238147.3238227.
50. Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z. Collafl: Path sensitive fuzzing. *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, 2018; 660–677, doi:10.1109/SP.2018.00040. URL doi.ieeecomputersociety.org/10.1109/SP.2018.00040.
51. Li J, Zhao B, Zhang C. Fuzzing: a survey. *Cybersecurity* Jun 2018; **1**(1):6, doi:10.1186/s42400-018-0002-y. URL https://doi.org/10.1186/s42400-018-0002-y.
52. Serebryany K. Continuous fuzzing with libfuzzer and addresssanitizer. *2016 IEEE Cybersecurity Development (SecDev)*, 2016; 157–157, doi:10.1109/SecDev.2016.043.
53. Householder AD, Foote JM. Probability-based parameter selection for black-box fuzz testing. *Technical Report*, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST 2012.
54. Godefroid P, Kiezun A, Levin MY. Grammar-based whitebox fuzzing. *Programming Language Design and Implementation (PLDI)*, 2008.
55. Eddington M. Peach fuzzing platform. *Peach Fuzzer* 2011; .
56. Serebryany K. Continuous fuzzing with libfuzzer and addresssanitizer. *2016 IEEE Cybersecurity Development (SecDev)*, 2016; 157–157, doi:10.1109/SecDev.2016.043.
57. honggfuzz. https://github.com/google/honggfuzz. Accessed: 2019-07-30.
58. Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. Vuzzer: Application-aware evolutionary fuzzing. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
59. Kim Y, Kim Y, Kim T, Lee G, Jang Y, Kim M. Automated unit testing of large industrial embedded software using concolic testing. *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering*, 2013; 519–528.
60. Kim Y, Choi Y, Kim M. Precise concolic unit testing of C programs using extended units and symbolic alarm filtering. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018.
61. Kim Y, Lee D, Baek J, Kim M. Concolic testing for high test coverage and reduced human effort in automotive industry. *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '10, IEEE Press: Piscataway, NJ, USA, 2019; 151–160, doi:10.1109/ICSE-SEIP.2019.00024. URL https://doi.org/10.1109/ICSE-SEIP.2019.00024.
62. Kim Y, Hong S, Kim M. Target-driven compositional concolic testing with function summary refinement for effective bug detection. *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '19, 2019.
63. Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* Dec 2007; **69**(1-3):35–45.
64. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, IEEE Computer Society: Washington, DC, USA, 2007; 75–84.
65. Fraser G, Arcuri A. While test suite generation. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.
66. Hong S, Ahn J, Park S, Kim M, Harrold MJ. Testing concurrent programs to achieve high synchronization coverage. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, ACM: New York, NY, USA, 2012; 210–220, doi:10.1145/2338965.2336779. URL http://doi.acm.org/10.1145/2338965.2336779.