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

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

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

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

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

I. INTRODUCTION

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

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

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

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

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

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

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

We developed DEMINER (gui**DE**ed test generation using **M**utat**I**oN **E**xplo**R**ation) which realizes IST to improve test coverage. 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 covering new lines not covered on p with T and corresponding mutation operator instances.
3) Based on the recorded information, DEMINER constructs *guideposts* to guide test generation on p to improve test coverage.
4) DEMINER applies concolic testing on p with an inserted guidepost to generate new test executions that follow the guideposts and achieve high test coverage.

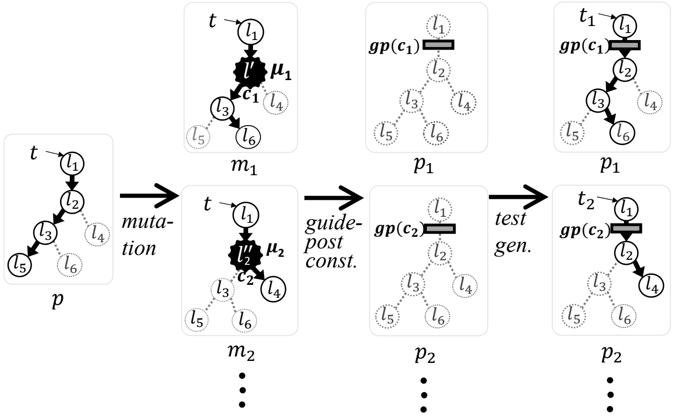


Fig. 1. DeMiner overview

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_1 with l'_1), and t covers l_6 on m_1. From this mutant execution, DeMiner captures a partial state c_1 infected by μ_1 as a key for t to covering 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 on p (e.g., t_1) with the inserted guidepost which follow/satisfy the guidepost condition c_1. DeMiner repeats this process for all other coverage-increasing mutants each of which increases line coverage by covering lines that were not covered on p with t.

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

The contributions of this paper are as follows:

1) As far as the authors know, this paper is first to technically proposes *invasive software testing* to explicitly generate highly effective test inputs.
2) We developed DeMiner which realizes IST by generating diverse test executions guided by *guideposts* created by learning coverage-increasing conditions from various mutant executions.
3) We performed case studies to demonstrate that DeMiner can increase branch and line coverage compared to the conventional concolic testing through three real-world C programs (i.e., `Busybox-ls`, `Busybox-printf`, and `GNU-find`).

The remainder of this paper is organized as follows. Section II shows a motivating example showing a limitation of a current automated test generation technique and how IST can overcome it. Section III explains DeMiner, and Section IV describes research questions and the experiment setup to study the effectiveness of DeMiner. Sections V and VI report and discuss the experiment results. Finally, Section VII concludes this paper with future work.

```
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  }
```

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

II. Motivating Example

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

For example of `get_max` in Figure 2, concolic testing (almost) 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 (Depth First Search) as a concolic search strategy.

Suppose that an initial input for `get_max` has

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

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

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

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

After the initial execution, concolic testing negates the last branch condition (i.e., $a[99] > 0$) and the resulting symbolic path constraint 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:

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

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==sz`. The execution of m_3 with the initial input does not enter the loop and covers Lines 8–9. Then, DeMiner learns from this mutant execution and generates a guidepost `guidepost(0==sz)` between Line 2 and Line 3 of `get_max` (see Section III-C for the detail of guidepost construction).

Since `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., `0==sz`). After that, the concolic testing generates a next test input which has `sz==0` by solving the symbolic path constraint obtained by negating the last branch condition (i.e., the guidepost condition `0==sz`). Finally, `get_max` reaches Lines 8–9 with this test input generated with the guide of the guidepost.

III. DeMiner Framework

A. Overview

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

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

Figure 3 describes the DeMiner process generating new test inputs T'. Initially, DeMiner takes source code of an original target program p, and a set of test inputs $T = \{t_1, t_2, \ldots, t_k\}$ 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.
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 on the copies of p with the guideposts to generate test executions that follow/satisfy the guideposts to achieve the unreached lines covered at Phase 1.

The remainder of this section describes each phase in detail.

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

B. Phase 1. Discovery of coverage-increasing mutants

This phase aims at finding mutants whose executions cover some code lines that are not covered by running the original program p with a set of test input T. DeMiner constructs mutants $m_1, m_2, \ldots, m_n$ by mutating expressions $e_1, e_2, \ldots, e_n$ in p respectively, and then runs the mutants with T.

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

Table I shows names and categories of the mutation operators [13] used by DeMiner. DeMiner uses only *expression-level mutation operators* because DeMiner focuses on a *single expression change* that increases coverage. DeMiner does not use statement-level mutation operators (e.g., SSDL(statement deletion), SBRC(replacement of `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).

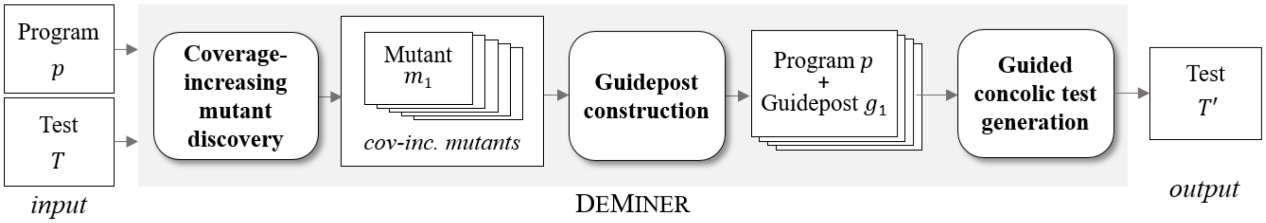


Fig. 3. The overall process of DEMINER

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

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

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

C. Phase 2. Guidepost construction

From the executions of the coverage-increasing mutants obtained from Phase 1, DEMINER infers a precondition at a program location to cover the unreached lines. DEMINER expresses such a precondition as a *guidepost* encoded as an if-statement that continues the execution if the condition is satisfied, or terminates the execution otherwise (i.e., `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 concolic testing on a target program p to generate tests toward the observed coverage-increasing executions.

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

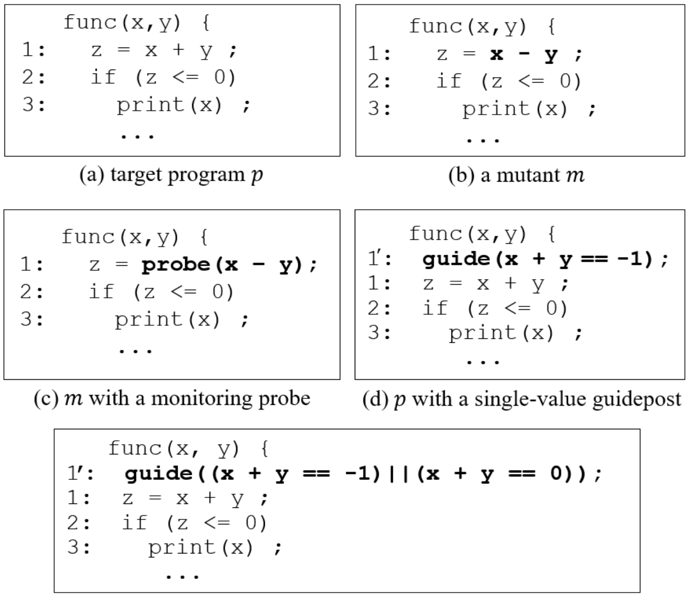
For each m_i, DEMINER identifies all runtime evaluations of the mutated expression as *coverage-increasing values* $V_i = \{v^i_1, v^i_2, \ldots, v^i_u\}$. To extract V_i from the executions of m_i, DEMINER instruments m_i by inserting a probe exporting evaluation results of the mutated expression.

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

- *Single-value guidepost*
  For each $v^i_j \in V_i$, a single-value guidepost is created to check if e_i is evaluated to v^i_j for the first time. This condition is encoded as `guidepost(`e_i `==` v^i_j`)` for $v^i_j \in V_i$.
- *Multi-value guidepost*
  For m_i with multiple coverage-increasing values (i.e., $|V_i| > 1$), DEMINER additionally creates a multi-value guidepost that checks e_i is always evaluated to one of the coverage-increasing values in V_i. Thus, the condition of a multi-value guidepost is formed as `guidepost(`$(e_i == v^i_1) \vee \ldots \vee (e_i == v^i_u)$`)`. A multi-value guidepost allows a target expression to have multiple value choices when it is evaluated multiple times (e.g., in a loop).

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

[1] 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 IV-D).



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

(a) target program p

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

(b) a mutant m

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

(c) m with a monitoring probe

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

(d) p with a single-value guidepost

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

(e) p with a multi-value guidepost

Fig. 4. Example of guidepost construction

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

For each guidepost g, DEMINER generates a version of p which embeds g.

D. Phase 3. Guided concolic testing

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

- The search algorithm first performs concolic testing using DFS until the inserted guidepost is reached. If the guidepost condition c is violated, the target program execution immediately terminates. Then, the search algorithm negates the last branch condition (i.e., unsatisfied guidepost condition $\neg c$) and generates a test execution that satisfies c. If concolic testing fails to generate a test execution satisfying c, the concolic testing continues to use DFS until reaching the guidepost again through a different test execution.
- If concolic testing generates a test execution satisfying c, the concolic testing uses RND to negate only those branches executed after the guidepost. This is to focus on execution space satisfying a guidepost condition c and, thus, has high probability to increase coverage.

IV. EXPERIMENT SETUP

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

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

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 is to check the coverage improvement of DEMINER.

RQ3 is to check how effectively the guideposts guide concolic testing on p to cover the target unreached lines/branches covered by the coverage-increasing mutant executions.

Regarding RQ4, we compared DEMINER with the conventional concolic testing techniques that use three search strategies depth-first search (DFS), random branch negation (RND), and control-graph based heuristic for fast branch coverage increase (CFG) [14].

RQ5 evaluates the efficiency and the effectiveness of the greedy mutant selection method of DEMINER. 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 RQ5, we performed experiments on the three real-world C programs (see Table II). The following subsections explain the details of the experiment setup.

A. Study Objects

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

`Busybox-ls` is a file listing utility and `Busybox-printf` is a formatted data printer in BusyBox version 1.24.0 [2]. `GNU-find` is a file search utility in GNU FindUtils version 4.6 [3]. These three programs are utilities for UNIX-like operating systems. Table II shows the size of the target code in executable lines (LoC) and branches, a

[2] https://busybox.net

[3] https://www.gnu.org/software/findutils

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%)
GNU-find	3616	2091	120	2192 (60.6%)	1061 (50.7%)

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 [4]. We used `gcov` to measure LoC, and the line and the branch coverage throughout the experiments.

B. Mutant Generation Setup

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

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

C. Test Generation Setup

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

TABLE III
SYMBOLIC INPUT SETUP FOR THE STUDY OBJECTS

Program	Max. # sym-args	Max. len. sym-args	Max. # sym. file-metadata
Busybox-ls	4	6	5 × 13
Busybox-printf	7	11	-
GNU-find	6	10	6 × 13

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

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

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

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

D. Implementation

The DeMiner implemenentation is written in C++ and Python. The component for mutation analyses (i.e., in Phase 1, see Section III-B) consists of the mutant generation part and the mutant execution part. For mutant generation, we used MUSIC (MUtation analySIs tool with high Confiurability and extensibility) [15]. [5] MUSIC implements 73 expression-level and statement-level mutation operators for modern C programs (63 of them are defined in Agrawal et al. [13]). MUSIC preserves the source code line numbering in an expression-level mutation to make coverage information on mutants and the original program comparable. The mutant execution part is implemented in 540 lines of Python script code.

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

[4] Nine test cases were excluded as they re-execute the main routine multiple times, which complicate concolic testing with initial test case seeding (see Section IV-C). Also, to save testing time, two test cases were executed since they consume more than 15 longer execution times than the other test cases, while the two test cases do not increase the total line/branch coverage.

[5] We failed to use Proteum [17] and MILU [18] 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.

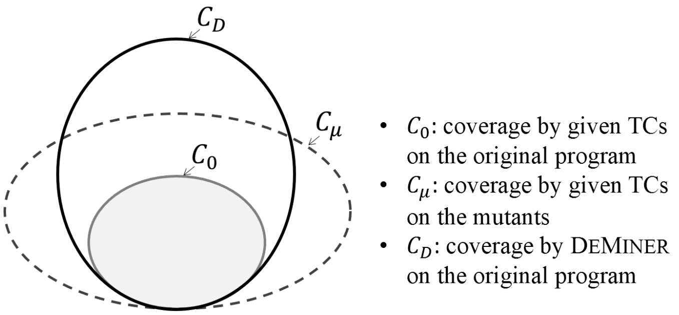

C_D
C_μ
C_0
- C_0: coverage by given TCs on the original program
- C_μ: coverage by given TCs on the mutants
- C_D: coverage by DeMiner on the original program

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

point arithmetic, bitfields and so on. We added 230 lines of C/C++ code to implement the prioritized search strategy which hybridizes DFS and RND (see Section III-D).

E. Data Collection

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

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

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

F. Threats to Validity

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

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

V. Experiment Results

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

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

Program	Total mutants	Cov-incr. mutants	# of cov. line (line cov.)	# of cov. br. (br. cov.)
`Busybox-ls`	17313	1280	363 (89.9%)	247 (81.5%)
`Busybox-printf`	5728	118	163 (96.4%)	99 (94.3%)
`GNU-find`	5501	172	2505 (69.3%)	1257 (60.1%)

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

The experiment results on RQ1 show that mutation effectively diversifies program executions in a large degree, so that many unreached lines and branches are covered on the mutants with the given test cases. (i.e., $|C_\mu - C_0| >> 0$ 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 results shows that, with the same given test cases, the executions on mutants cover meaningfully large amount of additional lines and branches compared to the execution on the original program. For example of `GNU-find`, the given 120 test cases cover 2,505 lines on the 172 mutants (i.e., covering 313 (=2505-2192) more lines than the original program). The mutant executions increase line coverage by 41.2% (=(363-257)/257), 16.4% (=(163-140)/140), and 14.3% (=(2505-2192)/2192) for `Busybox-ls`, `Busybox-printf`, and `GNU-find`, respectively. Also, the mutant executions improve branch coverage by 83.0% (=(247-135)/135), 32.0% (=(99-75)/75) and 18.5% (=(1257-1061)/1061) for `Busybox-ls`, `Busybox-printf`, and `GNU-find`, respectively. Note that these coverage increments appear on the mutants, *not* on the original target program.

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

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

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

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

TABLE V
THE NUMBER OF COVERED LINES AND BRANCHES BY DEMINER

Program	# of guideposts	# of cov. line (line cov.)	# of cov. br. (br. cov.)
Busybox-ls	3087	348 (86.1%)	231 (76.2%)
Busybox-printf	183	161 (95.3%)	93 (88.6%)
GNU-find	475	2458 (68.0%)	1198 (57.3%)

TABLE VI
COVERAGE INCREASE BY MUTANT EXECUTIONS AND DEMINER

Program	Cov	$\lvert C_\mu - C_0\rvert$	$\lvert C_D - C_0\rvert$	$\lvert(C_D - C_0)\cap (C_\mu - C_0)\rvert$
Busybox-ls	Line	106	91	91
	Br.	112	96	96
Busybox-printf	Line	23	21	21
	Br.	24	18	18
GNU-find	Line	313	266	251
	Br.	196	137	108

For Busybox-printf and GNU-find, DeMiner increases line coverage 15.0% (=(161-140)/140) and 12.1% ((2458-2192)/2192), respectively. In addition, it increases branch coverage of Busybox-ls, Busybox-printf, and GNU-find by 71.1% (=(231-135)/135), 24.0% (=(93-75)/75), and 12.9% (=(1198-1061)/1061), respectively.

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

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

For example of Busybox-ls, DeMiner covers 85.8% (= $\frac{|(C_D-C_0)\cap(C_\mu-C_0)|}{|C_\mu-C_0|}$=91/106) of the unreached lines covered by the mutant executions. For Busybox-printf and GNU-find, DeMiner covers 91.3% (=21/23) and 80.2% (=251/313) of the unreached lines that the mutant executions cover (see the fifth column). Similarly, for Busybox-ls, Busybox-printf, and GNU-find, it covers 85.7% (=96/112), 75.0% (=18/24), and 55.1% (=108/196) of the unreached branches that the mutant executions cover (see the third and fifth columns).

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

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

TABLE VII
THE NUMBER OF UNREACHED LINES AND BRANCHES COVERED BY THE CONVENTIONAL CONCOLIC TESTING AND DEMINER

Program	Cov	Conventional Concolic			DeMiner
		DFS	RND	CFG	
Busybox-ls	Line	81	77	77	91
	Br.	84	79	78	96
Busybox-printf	Line	18	18	19	21
	Br.	14	14	15	18
GNU-find	Line	204	233	210	266
	Br.	83	98	89	137

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

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

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

For example of Busybox-ls, DeMiner covered 96 unreached branches which are 14.3% (=(96-84)/84) more branches than the conventional concolic testing using DFS, which is the best conventional concolic testing technique for Busybox-ls. Similarly, for Busybox-printf and GNU-find, DeMiner outperformed the three concolic testing techniques by covering 20.0% (=(18-15)/15) and 39.8% (=(137-98)/98) more unreached branches than CFG and RND which are the best concolic testing techniques for Busybox-printf and GNU-find, respectively. [6] On average, the amount of the improved branch coverage by DeMiner is 24.7% (=(14.3+20.0+39.8)/3) relatively larger than those of the conventional concolic testing techniques.

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

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

[6] For GNU-find, we found that DeMiner generated test inputs that cover 22 unreached *empty* else branches (i.e., if without else) and, thus, increase branch coverage but not line coverage.

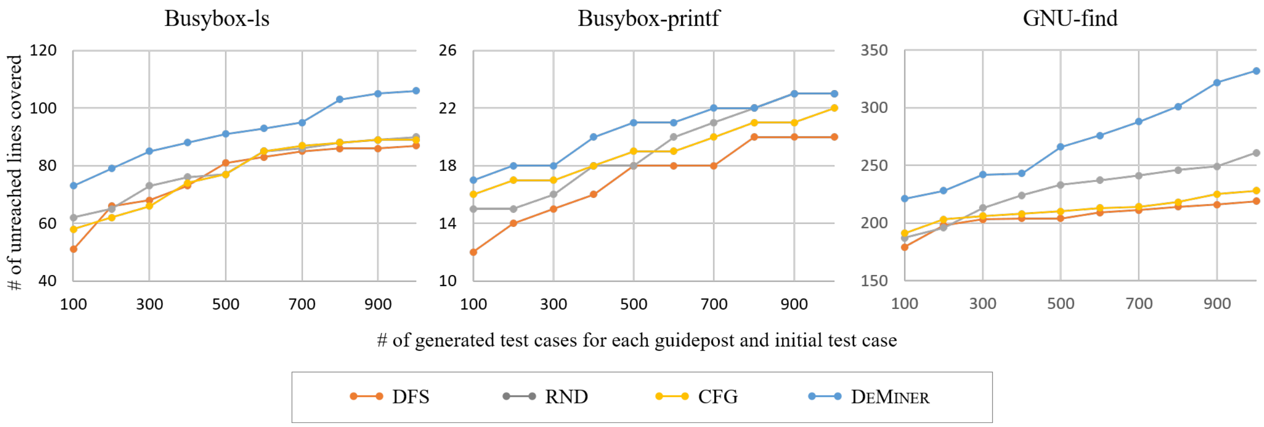


Fig. 6. The number of unreached lines covered with different numbers of test cases generated per different techniques

TABLE VIII
EFFECT OF COVERAGE-INCREASING MUTANT SELECTION METHODS

Program	Unreached lines covered (branches)			Time (in hour)		
	Greedy	Rand.	All	Greedy	Rand.	None
`Busybox-ls`	91 (96)	85 (90)	91 (96)	9.5	8.6	776.5
`Busybox-printf`	21 (18)	18 (14)	21 (18)	6.5	6.7	123.8
`GNU-find`	266 (137)	211 (91)	338 (162)	244.8	247.6	6960.0

covers the unreached lines around 27% (=(332-261)/261) more than the best conventional concolic testing technique (i.e., RND) (see the right end of the `find` graph in the figure).

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

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

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

Also, DEMINER with the greedy selection covers more lines and branches than DEMINER with random selection for all cases. For example of `Busybox-ls`, DEMINER with the greedy mutant selection covers 91 unreached lines while DEMINER with the random selection does only 85 lines.

```
// 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);
```

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

VI. DISCUSSION

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

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

Conventional concolic testing using DFS and RND could not cover Lines 202–203 in 23,394 seconds, due to the loops inside `strstr` at Line 172 and `strchr` at Line 173. Both of the loops have symbolic variables in their loop conditions,

[7] Because `GNU-find` has larger room to increase coverage than `Busybox-ls` and `Busybox-printf`, using more coverage-increasing mutants and more guideposts can effectively increase coverage.

because they iterate over a symbolic input string `format` until they reach any specified character (i.e., '.', '*', or '\0'). Therefore, conventional concolic testing keeps increasing the loop bound and fails to cover Lines 202–203 within a given time bound.

Note that DeMiner covers these hard-to-cover lines as follows. DeMiner 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 inserts a guidepost at Line 168 and guided concolic testing of DeMiner 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 III-D). Therefore, DeMiner effectively covers Lines 202–203.

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

B. Application of Mutation Analysis

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

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

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

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

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

VII. 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 three 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 converage-increasing conditions more efficiently and effectively, we will utilize the automated unit test generation technique using concolic testing [31] which can generate the coverage-increasing conditions at an entry of a function. Third, we will leverage program invariant inference techniques, such as Daikon [32], to generate various kinds of guideposts from mutant execution information. In addition, we will explore what kinds of guidepost condition structures are effective for capturing useful knowledge from mutant executions. Finally, we will apply 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, No. 2017M3C4A7068179), Basic Science Research Program (No. 2016R1A2B4008113, No. 2017R1C1B1008159) through the National Research Foundation (NRF) funded by the Ministry of Science and ICT of Korea, Basic Science Research Program (No. 2017R1D1A1B03035851) through NRF funded by the Ministry of Education of Korea, and KAIST High Risk High Return Project (HRHRP).

References

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[2] K. Sen and G. Agha, "CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools," in *Proceedings of Computer Aided Verification (CAV)*, 2006.

[3] Y. Kim and M. Kim, "SCORE: a scalable concolic testing tool for reliable embedded software," in *Proceedings of the Joint Meeting of European Software Engineering Conference and the ACM Symposium of Foundations of Software Engineering (ESEC/FSE)*, 2011.

[4] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux, "Precise identification of problems for structural test generation," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[5] P. Godefroid, "Higher-order test generation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[6] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[7] B. Elkarablieh, R. Godefroid, and M. Levin, "Precise pointer reasoning for dynamic test generation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[8] M. Trtík and J. Strejček, "Symbolic memory with pointers," in *Proceedings of the Automated Technology for Verification and Analysis (ATVA)*, 2014.

[9] A. Romano and D. R. Engler, "symMMU: Symbolically executed runtime libraries for symbolic memory access," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014.

[10] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[11] M. Christakis, P. Muller, and V. Wustholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016.

[12] E. Alatawi, H. Søndergaard, and T. Miller, "Leveraging abstract interpretation for efficient dynamic symbolic execution," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2017.

[13] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Purdue University, Tech. Rep. SERC-TR-120-P, 1989.

[14] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-123, Sep 2008.

[15] D. L. Phan, Y. Kim, and M. Kim, "MUSIC:Mutation Analysis Tool with High Configurability and Extensibility," in *Proceedings of the International Workshop on Mutation Analysis (MUTATION)*, 2018.

[16] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, 2017.

[17] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. d. S. Simão, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero, "Proteum: a family of tools to support specification and program testing based on mutation," in *Mutation testing for the new century*, 2001.

[18] Y. Jia and M. Harman, "MILU : A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, 2008.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[20] "CROWN: Concolic testing for Real-wOrld softWare aNalysis," http://github.com/swtv-kaist/CROWN, accessed: 2018-02-14.

[21] Y. Kim, M. Kim, Y. Kim, and Y. Jang, "Industrial application of concolic testing approach: A case study on libExif by using CREST-BV and KLEE," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[22] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 5, pp. 649–678, 2011.

[23] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, pp. 278–292, 2011.

[24] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test oracle assessment and improvement," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[25] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2014.

[26] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2015.

[27] S. Hong, T. Kwak, B. Lee, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "MUSEUM: Debugging real-world multilingual programs using mutation analysis," *Information and Software Technology (IST)*, vol. 82, pp. 80–95, Feb. 2017.

[28] M. Papadakis and Y. L. Traon, "Metallaxis-FL: mutation-based fault localization," *Journal of Software Testing, Verification, and Reliability (STVR)*, vol. 25, no. 5-7, pp. 605–628, 2015.

[29] J. Zhang, Y. Lou, L. Zhang, D. Hao, L. Zhang, and H. Mei, "Isomorphic regression testing: Executing uncovered branches without test augmentation," in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[30] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: Object capture-based automated testing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[31] Y. Kim, Y. Choi, and M. Kim, "Precise concolic unit testing of C programs using extended units and symbolic alarm filtering," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018.

[32] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.