Generating Succinct Test Cases using Don't Care Analysis

Cuong Nguyen*, Hiroaki Yoshida†, Mukul Prasad†, Indradeep Ghosh† and Koushik Sen*
*Department of Computer Science, University of California, Berkeley, CA, USA
Email: {nacuong, ksen}@cs.berkeley.edu
†Fujitsu Laboratories of America, Inc., Sunnyvale, CA, USA
Email: {hyoshida, mukul, ighosh}@us.fujitsu.com

***Abstract*—We study the problem of reducing test cases generated by bit vector based symbolic execution test generators. In particular, we first consider a guileless test case generation approach that generates assignment statements for each symbolic scalars, array and structure elements and object fields. We show that test cases generated by this approach can be significantly verbose. We then propose a method for making the generated test cases more succinct using a novel analysis entitled *don't care analysis*. Don't care analysis identifies assignment statements that can be safely removed from the test cases without affecting the overall code coverage. Our algorithm is based on binary and delta-debugging search. Because it exploits the knowledge of the internal SAT solver, it is effective and efficient in practice. To our knowledge, this is the first fully automatic approach that reduces the sizes of test cases generated using symbolic execution.**

We implement our test case reduction technique for the KLEE test generation tool and evaluate on 295 programs and functions. Our results are encouraging: in average, the reduced test cases are 50 times smaller than the test cases generated by the guileless test case generator. In addition, since our don't care analysis is tightly integrated into the test case generation tool, its overhead to the overall test generation process is negligible.

I. Introduction

Our growing dependence on computing systems in many aspects of life has shown the significant reliance of society in software functionality. The universal and omnipresent character of computing systems has made not only their performance but also their reliability important. Traditionally, software quality has been assured through manual testing, which is tedious, time consuming, and provides poor code coverage. Recent work in the literature has found success in employing *symbolic execution* to automate the test generation process, with high structural coverage, for the programs under test [17], [26], [12], [8], [21]. At a high level, symbolic execution symbolically explores all paths in a function or programs under test while collecting path constraints along the way. They then employ a constraint solver to generate test inputs that satisfy the constraints, thus providing test cases to concretely traverse through the associated execution paths.

While many recent work in the literature have focused on the ability to find bugs using symbolic execution [11], [28], [22], less is known about the quality and usability of test cases generated using symbolic execution. In particular, we desire to generate test cases that are human friendly, easy to maintain and debug. The practice of making test cases as small as possible is well recognized in practice as a mean of supporting of these goals. For example, as quoted in the GNU bug reporting instructions, "smaller test cases make debugging easier", "GCC developers prefer bug reports with small, portable test cases" and "minimized test cases can be added to the GCC test suites" [1]. Aside from the GNU website, minimizing test cases are also recommended in the guidelines of other code bases, such as LLVM [2], Mozilla [3] or Webkit [4][1].

Nevertheless, a common issue pertaining to symbolic execution engines is that their generated input assignments are not easy to be interpreted by human. In particular, in bit vector based symbolic execution engines [12], [21], the memory model is represented as a bit vector array, and thus the input assignments are produced as a sequence of bytes. The natural approach for generating test cases from the generated test inputs is to interpret these sequences of bytes as values depending on the type of the symbolic variables. For aggregate types, scalar values are generated first and then aggregated to form the whole object value. However, we find that this approach generates verbose test cases. In particular, we studied this approach using 100 programs from the GNU Coreutils and 195 functions from SQLite3 and found that the generated test cases can be made in average 50 times smaller (Section IV).

How can we effectively reduce the sizes of test cases generated using symbolic execution without affecting the overall code coverage? In this paper, we propose a method for reducing test cases using a novel analysis entitled don't care analysis. Essentially, don't care analysis infers a set of *don't care symbolic variables*: variables that can be assigned to any values, *within the context of a specific satisfying assignment*, without affecting the overall code coverage. Using don't care analysis, symbolic execution engines can remove assignment statements for don't care variables from the generated test cases, thus make the test cases more succinct while achieving the same code coverage. Concretely, we reduce the problem of reducing test cases generated by symbolic execution based test generators to the problem of inferring don't care variables in a constraint formula, i.e., variables that, within the context

[1]We note that this problem is different from test suite reduction, in which the goal is to reduce, from a large set of test cases, a smaller subset that likely reveals faults in the program under test.

of a satisfiable assignment, do not affect the satisfiability of the constraint. The analysis is performed at two levels: SMT level and SAT level. At the SMT level, during the constraint formula simplification phase [15], we infer a set of don't care SMT variables that do not appear in the simplified formula. At the SAT level, we attempt to find a *locally maximal* set of SAT variables that can be assigned to `undefined` such that the SAT formula is still satisfiable. The notion of `undefined` will be formally defined in Section III. Our algorithm exploits the *solution trail* generated by the SAT solver algorithm, which ranks the variables in the order of their importance using the *variable state independent decaying sum* (VSIDS) heuristic [24]. We then apply binary search to find a cut that is closest to the beginning of the trail such that all variables after the cut are don't care variables. Finally, we apply an optional post-processing step based on delta-debugging search to tune the set of care variables into a local maximum set.

We implemented our test reduction method for the KLEE test generation tool and evaluated it on a set of 295 programs and functions, including 100 programs from the GNU COREUTILS and 195 API functions from SQLITE3. Our experiment showed encouraging results: the reduced test cases were 50 times smaller in average compared to the test cases generated by the guileless test case generator. In addition, don't care analysis introduced negligible overhead to the overall test generation process.

Our main contributions are as follows:

1) We study the problem of reducing test cases generated using symbolic execution. We reduce this problem to the problem of inferring don't care variables in an SMT constraint formula. To our knowledge, our paper is the first work that looks into this issue.
2) We propose a novel analysis, entitled don't care analysis, for determining don't care variables from a constraint formula. We use this analysis to reduce the generated test cases. The resulting test case sizes are smaller, yet the test cases achieve the same code coverage as the original test cases.
3) We demonstrate the effectiveness of our analysis on a large number of programs and functions, thus reducing the chance of per-dataset bias.

The outline of the paper is as follows: In Section II, we study a concrete example to motivate how a test case generated by symbolic execution can be verbose. We then provide a background on symbolic execution and constraint solver in Section II-A, II-B and II-C. This is followed by a detailed description of our don't care analysis approach (Section III), an evaluation of its effectiveness (Section IV) and discussion (Section V). Finally, we discuss related works in Section VI before concluding in Section VII.

II. BACKGROUND

A. Symbolic Execution

We begin with a brief overview on how a symbolic execution test generator can generate test cases automatically given a function under test (FUT). Consider the following code snippet.

```c
1  // assume c has 100 elements
2  int foo(unsigned char i, char c[]) {
3    if (c[i] == i)
4      return 0;
5    else if (c[i] == i+1)
6      return 1;
7
8    return 2;
9  }
```

Listing 1: Example of a function under test.

First, the symbolic execution test generator can automatically extract the test harness, which is the interface of the function under test together with its external environment. In this example, the test harness includes two variables that will be made symbolic: variable `i` of type `unsigned char` and variable `c` of type `char[]`. The test harness will also include information about the size and the type for each symbolic variable. Aggregate type variables can be initialized with a random size, e.g. variable `c` of array type has length 100.

Second, the symbolic execution test generator invokes an executor that symbolically explores all paths in the function under test. The executor accumulates the constraints appeared at each control statement it encounters along each path and report them at the end of the symbolic execution of that path. The resulting constraint is called the path condition (PC). A path condition is a constraint on the symbolic variables, which, when solved, produces an assignment that follows the associated path. Figure 1 shows the execution tree of the FUT in Listing 1 along with the generated path conditions. Finally, the path conditions are fed into a constraint solver [15], to generate satisfying assignments for the symbolic variables if they exist. Figure 2 depicts the overall architecture of a symbolic execution test generator.

In bit vector based symbolic execution engines, the memory model is naturally represented as a sequence of bytes. Thus the test inputs are also produced in the format of byte sequences. To make this output more human readable, the test generator typically includes a script rendering component that renders the values nicely and makes the test cases executable. Concretely, the test script renderer generates values for symbolic variables based on their types. For aggregate types, it first generates values for each scalar element and then aggregates

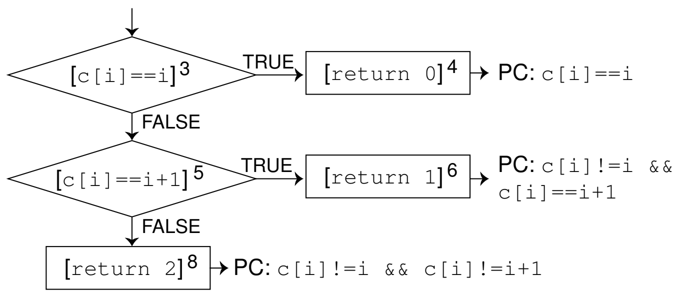


Fig. 1: Execution tree of the function under test.

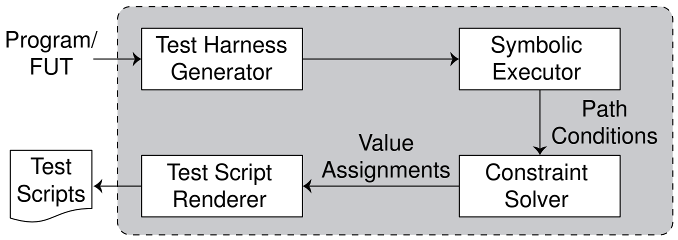


Fig. 2: Architecture of a symbolic execution based test generator.

them to form the whole object value. Finally, it sets up the test harness if necessary, and invokes the FUT with the generated input. The test cases in Listings 2, 3 and 4 are examples of generated test cases for the FUT in Listing 1. The corresponding path condition is annotated as a comment in each test case.

```
// PC: c[i] == i
char c[100];
unsigned char i = 0;
c[0] = 0;
c[1] = 0;  // don't care
...        // don't care
c[99] = 0; // don't care
foo(i, c);
```

Listing 2: Generated Test Case 01.

```
// PC: c[i] != i && c[i] ==  i+1
char c[100];
unsigned char i = 1;
c[0] = 0;  // don't care
c[1] = 2;
...        // don't care
c[99] = 0; // don't care
foo(i, c);
```

Listing 3: Generated Test Case 02.

```
// PC: c[i] != i && c[i] !=  i+1
char c[100];
unsigned char i = 0;
c[0] = 2;
c[1] = 0;  // don't care
...        // don't care
c[99] = 0; // don't care
foo(i, c);
```

Listing 4: Generated Test Case 03.

As noted in the test case code, in each test case, 99 of the 103 lines of code are irrelevant to the path condition (noted as “don't care”) and can be safely removed without affecting the code coverage or integrity of the test case. When the don't care assignment statements are removed, the reduced test cases are 25 times smaller. The problem is rooted in the SMT constraint solver phase, where the variable `c` is represented as an array of 100 bytes. The solver generates a satisfying assignment

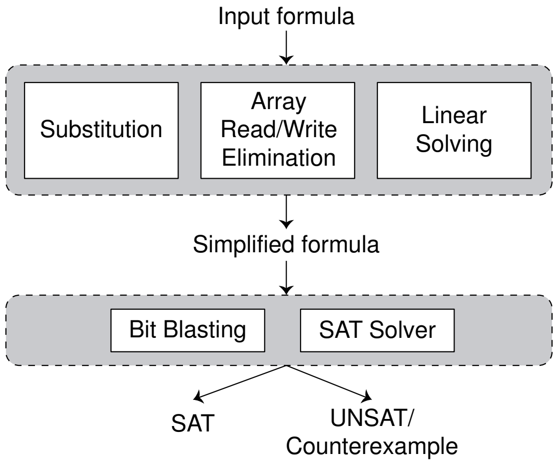


Fig. 3: Architecture of a bit-vector and array SMT solver.

for each byte without knowing which bytes are irrelevant to the path condition. To address this problem, in this paper, we propose a novel analysis entitled don't care analysis to infer don't care variables from a constraint formula, and as a consequence, help the test script generator remove irrelevant assignment statements from the generated test scripts.

For the remainder of this section, we will give an overview of the constraint solving algorithm for theory of bit vectors and arrays (Section II-B). We then review the algorithm used in typical SAT solvers (Section II-C). These backgrounds serve as a foundation for our don't care analysis, which will be presented in Section III.

B. Bit-Vector and Array SMT Solver

In this section, we will give an overview of a decision procedure for satisfiability of quantifier-free formulas in the theory of bit-vectors and arrays. A detailed discussion of this algorithm can be found at [15]. A bit-vector and array constraint solver is typically used in test case generation because it can naturally represent the program memory model as an array of bytes. A typical implementation consists of two core components: the formula simplification component and the SAT solver component. We depict this architecture in Figure 3.

The formula simplification component employs several high-level structure simplification strategies to reduce the number of symbolic variables and the complexity of the constraint. These strategies include substitution, linear solving, and array optimizations. In particular, the substitution phase represents the formula using only a subset of the symbolic variables. Linear solving partially solves the constraints using arithmetic and Boolean simplification. Finally, array optimization eliminates the array read and write operations, and simplifies the transformed formula. The simplified formula is then transformed into a conjunctive normal form (CNF) using a bit-blasting algorithm and solved by a SAT solver [14]. During the transformation process, the SMT solver maintains a substitution map, which can later be used to re-construct a

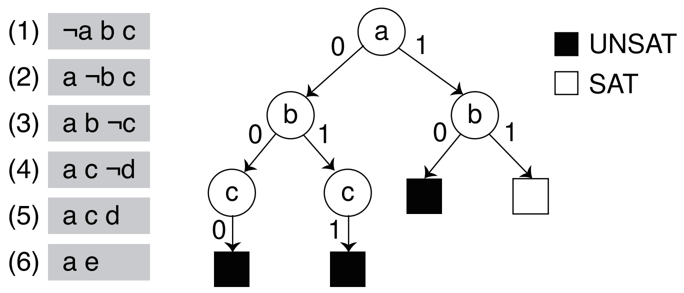


Fig. 4: An example of DPLL solution.

high-level structural representation from the SAT solution.

C. SAT Solver

Many modern SAT solvers are instances of a CHAFF-like SAT solver, which employs a two-literal watching scheme for fast Boolean constraint propagation (BCP) and clause learning by conflict analysis [14], [24]. At the core of these SAT solvers is the basic DPLL backtracking-based search algorithm. In this section, we will first review the classic DPLL algorithm. We then learn how the modern implementation of DPLL differs from the classic implementation.

At a high level, DPLL is a search and backtracking algorithm. The set of Boolean literals in the formula are ordered in some chronological order and are sequentially assigned in that order. If an assignment causes the formula to be unsatisfiable, DPLL backtracks to a literal where an alternative assignment can be made and continues from that assignment. Figure 4 demonstrates the DPLL algorithm on a Boolean satisfiability problem. In the first step, literal `a` is picked and randomly assigned the value `0`. DPLL then performs BCP, which identifies and removes all clauses that become true (which is clause 1 in this case). BCP also removes the literal `a` from other clauses, including clauses 2, 3, 4, 5 and 6. At this point, clause 6 only consists of one literal `e` which indicates that `e` must be 1. We note here that the state in which a variable is picked and assigned a random value is called a *decision level*. At a decision level, other Boolean variables can be influenced and several assignments can be determined, e.g. values of `a` and `e` are determined at the same decision level in this example. Similarly, `b` is chosen at the next decision levels and randomly assigned the value 0. At this point, BCP identifies that `c` must be 0 at clause 3 but finds a conflict at clauses 4 and 5 because they both imply that `d` needs to be both `0` and `1`. The algorithm then backtracks to a decision level that has not been tried both ways and continues from that point. The algorithm stops when all clauses have become true (satisfied), otherwise when all literals have been tried both ways (unsatisfiable). In this example, the formula is satisfied when `a` and `b` are assigned to `1`.

Modern SAT implementations of DPLL differ from the classic implementation in the following significant ways.

- Modern SAT implementations use conflict-driven backtracking, rather than chronological backtracking, to backtrack to the proper decision level. The process of BCP *does not maintain a set of satisfied clauses*, but employs two literal watching scheme and clause learning by conflict analysis for fast BCP. A detailed discussion on these two schemes can be found in [15]. As a result, in case of satisfiability, the algorithm does not terminate when all clauses have become satisfied, but rather when all Boolean literals have been assigned to some values. In case of unsatisfiability, the algorithm terminates when conflicts cannot be resolved.
- Variable ordering in modern SAT implementations is dynamic and is determined using the VSIDS heuristic [24]. At a high level, this ordering favors the variables actively involved in a conflict, since it deems that these variables are more important in determining the satisfiability of the problem. Therefore, our don't care analysis also utilizes this order as a heuristic to identify the maximum set of don't care variables.

III. DON'T CARE ANALYSIS

We discuss our realization of don't care analysis as a practical and effective method to infer don't care variables. Generally, don't care analysis can be implemented as a plug-in to an SMT solver. Given a constraint formula, SMT solver with don't care analysis attempts to find an assignment and a *maximal* set of don't care variables within the context of that assignment such that the formula is satisfiable. We consider two definitions of maximum.

Finding a Global Maximum: Finding a global maximum set of don't care variables may require evaluation of an exponential number of constraint formulas. To be precise, we may be required to evaluate as many as 2^{8n} constraint formulas, where n is the total number of bytes in the set of symbolic variables. This naive approach evaluates all possible variable assignments by changing the value of one byte at a time. The shortcoming of this approach is that it does not scale to constraints that have hundreds to thousands of bytes, as typically observed in real-world applications.

Finding a Local Maximum: If we are interested in finding a local maximal set of don't care variables, we could find a set of don't care variables so that, if we add any other variable into this set, the input formula will become unsatisfiable. Such a set is a local 1-maximum. We have found that finding a local 1-maximum don't care set is efficient, and at the same time, effective in making test cases more succinct. We target this definition of maximum in this work.

Figure 5 illustrates our integration of don't care analysis into a bit-vector and array SMT solver. First, a set of don't care variables is inferred during the formula simplification phase (Section III-A). Second, a 1-maximal set of don't care variables is inferred during the SAT solving phase (Section III-B). The final set of don't care variables is the combination of these two sets, but is represented in the format of the input symbolic variables.

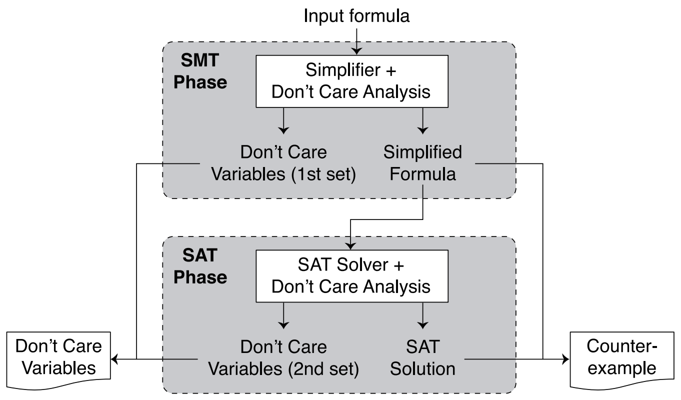

Input formula
SMT Phase
Simplifier + Don't Care Analysis
Don't Care Variables (1st set)
Simplified Formula
SAT Phase
SAT Solver + Don't Care Analysis
Don't Care Variables (2nd set)
SAT Solution
Don't Care Variables
Counter-example

Fig. 5: Overview of Don't Care Analysis.

A. *Don't Care Analysis at the Simplification Phase*

This section illustrates how the first set of don't care variables are inferred during the formula simplification phase. Essentially, at the simplification phase, the original formula is transformed into another simplified formula that is more suitable for the SAT solver. During this process, several new temporary variables are introduced into the formula. A substitution map is created to represent the correspondence of the original variables to the temporary variables. Variables that do not have any substitutions and do not participate in the simplified formula are determined to be don't care. Indeed, those variables neither need to have any values nor affect the simplified formula in any way. As an example, consider the following input formula. In this formula, `x` is an array of 3 bit-vectors of length 10, and `y` is a bit-vector of length 10.

```
x : ARRAY OF 3 BITVECTOR(10);
y : BITVECTOR(10);
ASSERT(y[1:3] = 011
   AND y[4:5] = 11
   AND x[1][2:5] = 1111
   AND x[1][1:3] = y[1:3]
);
```

Listing 5: Example of a constraint input formula.

After the simplification process, the formula is transformed into the formula given in Listing 6. The corresponding substitution map is given in Listing 7.

```
array_x_1[2:5] = 1111
AND
array_x_1[1:3] = 110
```

Listing 6: The simplified constraint formula.

```
b -> 011
c -> 11
y -> a CONCAT b CONCAT c CONCAT d
x[1] -> array_x_1
```

Listing 7: The substitution map.

Five variables are newly introduced: `a`, `b`, `c`, `d` and `array_x_1`. Variable `array_x_1` is a substitution for `x[1]` and is involved in the simplified formula. Variables `a` and `d` do not have any substitutions and do not appear in the simplified formula, thus are don't care variables. They imply that the first one bit and the last 4 bits of `y` are *don't care*. Likewise, `x[0]` and `x[2]` are don't care variables.

B. *Don't Care Analysis at the SAT Solver Phase*

In the SAT solver phase, the simplified formula is transformed into a CNF formula of Boolean literals. This section illustrates how a locally maximal set of don't care Boolean literals are inferred. A substitution map that maps each bit-vector variable to a vector of Boolean literals is maintained, and is used to infer which bit-vector variables are don't care given the set of don't care Boolean literals. Listing 8 shows the CNF formula and the substitution map corresponding to the formula in Listing 6. Each Boolean literal is denoted as an integer number. A negative number denotes the negation of a Boolean literal. Each line in the CNF formula is a clause of disjunction of Boolean literals. All clauses need to be true in order for the formula to be satisfiable. Conceptually, each Boolean literal can only be assigned to either `true` or `false` value. For the purpose of don't care analysis, we introduce a third special value, called `undef`. Value `undef` is a special value in which the evaluations of both `undef` and `¬undef` return the value `false`. As a consequence, `undef` literals can be removed from the clauses without affecting the satisfiability of the entire formula.

```
// CNF formula
2 -7 -8 -9 -3
-2 3
-2 7
-2 9
-2 8
3 5 -6
-3 -5
-3 6
14
2

// Bit-blasting map
array_x_1 -> 4 5 6 7 8 9 10 11 12 13
```

Listing 8: The bit-blasted CNF formula and the bit-blasting map.

Given such a CNF formula, our goal is to find an assignment that maximizes the number of `undef` Boolean variables. If the DPLL algorithm maintains a set of satisfiable clauses at each decision level, we can simply terminate the algorithm at the decision level when all the clauses become satisfiable and assign the rest of variables to the `undef` value. However, as discussed earlier in Section II-C, the set of satisfiable clauses are not maintained in modern DPLL implementations. Instead, our algorithm starts with a satisfiable solution produced by the SAT solver and attempts to switch as many variables to `undef` as possible. In particular, the algorithm operates on the solution trail returned by DPLL. It then uses binary search to find the *highest cut* (the cut that is closest to the beginning of the trail) such that all variables appeared after the cut are don't care variables. Recall that the solution trail orders variables based on their degrees of involvements in conflicts, in other words, their importance to the resolution of the formula. Therefore, such a cut also attempts to maximize the number of variables that may be irrelevant in the context of the given solution trail. Concretely, our algorithm is effective and efficient for the following two reasons.

- It utilizes the variable order inferred by the SAT solver. This heuristic guides the search effectively to find a maximum set of don't care variables.
- In each iteration, it operates on a decision level as a whole rather than on a single literal. Operating on decision levels is effective because in practice the number of decision levels can often be less than the number of literals by several orders of magnitude. It is also accurate because of the following reason: the cut can only been found at the end of each decision level. Indeed, if the cut is found in the middle of a decision level, it means that the decision literal is a care literal, but some of its BCP propagation literals are don't cared literals. This is a contradiction because the BCP literals' values are deterministic based on the decision literal value.

Figure 6 depicts our don't care cut search algorithm, entitled DCCSEARCH. The input to the algorithm is the *solution trail* Σ produced by SAT solver, the set of clauses $\mathbb{C}$ and the number of decision levels corresponding to the solution n. The solution trail Σ is an array of n assignments. Each element in the array is an assignment that maps the literals in the corresponding level to their Boolean values. $\mathbb{C}$ is the set of clauses in the CNF formula. The output of the algorithm is a *smallest* positive integer c less than or equal to n such that all variables from decision levels from c to n-1 are don't care variables. In the special case when $c = n$, all variables are care variables. DCCSEARCH first initializes low and up to be the lower bound and upper bound of the search space, and cut to be the middle point (line 3-4). It then sets the values of all variables in the decision levels from cut to up to $undef$ (line 7-9). DCCSEARCH then uses the predicate function $findunsat(\mathbb{C})$ to find a set of clauses that become unsatisfiable (line 10). This function checks for the satisfiability of each clause using the previously described semantic of $undef$ value. If the set of unsatisfiable clauses are empty, DCCSEARCH moves the cut closer to the lower bound by assigning up to cut, reset $\mathbb{C}_{\tt tmp}$ to the original set of clauses and recurs the algorithm from line 4 (line 11-14). Otherwise, it moves the cut closer to the upper bound by first resetting the variables from decision levels from cut to up to their original values (line 12-14) and then setting low to $cut + 1$ (line 15). It then performs an optimization by reducing the set of clauses for the next iteration to be $\mathbb{C}_{\tt unsat}$ (line 19). Finally, it recurs from line 4. The algorithm terminates when $low == up$ and returns cut as the output (line 20).

Procedure DCCSearch

Inputs

Σ	:	the solution trail
$\mathbb{C}$	:	the set of clauses
n	:	the number of decision levels

Outputs

The smallest c : Σ[c..n-1] are don't care variables

Algorithm

```
1  $\Sigma_{\tt tmp} = \Sigma$
2  $\mathbb{C}_{\tt tmp} = \mathbb{C}$
3  low, up = 0, n
4  cut = floor((low+up)/2)
5  while low < up:
6     cut = floor((low+up)/2)
7     for i in [cut..up-1]:
8        for v in $\Sigma_{\tt tmp}$[i]:
9           $\Sigma_{\tt tmp}$[i][v] = undef
10    $\mathbb{C}_{\tt unsat}$ = findunsat($\mathbb{C}_{\tt tmp}$)
11    if empty($\mathbb{C}_{\tt unsat}$)
12       up = cut
13       $\mathbb{C}_{\tt tmp} = \mathbb{C}$
14       continue
15    for i in [cut..up-1]:
16       for v in $\Sigma_{\tt tmp}$[i]:
17          $\Sigma_{\tt tmp}$[i][v] = $\Sigma$[i][v]
18    low = cut+1
19    $\mathbb{C}_{\tt tmp} = \mathbb{C}_{\tt unsat}$
20 return cut
```

Fig. 6: Don't Care Cut Search algorithm.

C. Finding a Local Maximum

To some extent, the DCCSEARCH algorithm finds the maximal set of don't care variables in the context of the solution trail when the VSIDS heuristic effectively ranks the variables in the order of their importance. Nevertheless, to guarantee that the analysis result is a 1-local maximum, optionally, we employ a delta-debugging based search on the set of care variables returned by DCCSEARCH. Delta-debugging is a well-known algorithm to find local maximum solutions for the test reduction problem [29]. Delta-debugging exhibits an $O(n^2)$ worst-case complexity where n is the number of test input elements. Essentially, given a test input and a predicate function, delta-debugging finds a locally maximal set of test elements that can be removed from the test input such that the predicate function remains true. In our context, the test input is a set of care variables and the predicate function is the satisfiability of the constraint formula. Delta-debugging helps switch as many care variables to don't care as possible such that the constraint formula is still satisfiable.

The algorithm starts by dividing the set of care variables Δ into two subsets of equal or almost equal size Δ_1 and Δ_2. It is also creates the complement set of these subsets $\nabla_1 = \Delta \setminus \Delta_1$ and $\nabla_2 = \Delta \setminus \Delta_2$ (lines 3 and 8). For each of these subsets, the algorithm verifies the satisfiability of the set of clauses $\mathbb{C}$, assuming all variables that do not belong to the subset are don't care variables (function `satisfied`($\mathbb{C}$,Δ)). If

Procedure LMSearch

Inputs

$\mathbb{C}$	:	the set of clauses
Δ	:	the set of care variables

Outputs

A minimal set of care variables

Algorithm

```
1   div = 2
2   for i in [1..div]:
3     $\Delta_i = \Delta[\frac{(i-1)|\Delta|}{\mathtt{div}}..\frac{i|\Delta|}{\mathtt{div}}]$
4     if satisfied($\mathbb{C}$, $\Delta_i$):
5       $\Delta$, div = $\Delta_i$, 2
6       goto 2
7   for i in [1..div]:
8     $\nabla_i = \Delta \setminus \Delta[\frac{(i-1)|\Delta|}{\mathtt{div}}..\frac{i|\Delta|}{\mathtt{div}}]$
9     if satisfied($\mathbb{C}$, $\nabla_i$):
10      $\Delta$, div = $\nabla_i$, div−1
11      goto 2
12  if div > $|\Delta|$:
13    return $\Delta$
14  else
15    div = 2 * div
16    goto 2
```

Fig. 7: Local Minimum Search algorithm.

a satisfiable subset of care variables exist, the algorithm recurs with that smaller subset (lines 5-6 and 10-11); otherwise it restarts the algorithm with a finer-grained partition (lines 15-16). In the special case when the granularity can no longer be increased, the algorithm returns the current Δ, which is a locally minimal set of care variables (lines 12-13). The complement of this set is the locally maximal set of don't care variables.

IV. Evaluation

We have implemented the test case reduction technique described in this paper as a plugin for the KLEE symbolic executor. Don't care analysis is integrated as a part of STP, KLEE's SMT solver. DCCSearch and LMSearch are implemented for MiniSAT, the SAT engine used by STP solver. We evaluated our test reduction technique on a set of 295 programs and functions, including 100 programs from the GNU Coreutils and 195 API functions from SQLite3. The Coreutils suite consists of 100 well-known command line programs such as `ls`, `pwd`, `chmod`, etc. and is often used as a benchmark for evaluating symbolic execution test generation engines [12], [22], [9]. SQLite3 is also one of the most widely used SQL database engines in practice.

KLEE does not have test harness analysis component, therefore we setup the test harnesses manually. For GNU coreutils programs, we use the setting recommended in [5], which was previously used by many symbolic execution test generation work in the literature [12], [22], [9]. For SQLite3 API functions, we use the same setting as in [5], but removing the symbolic standard input and output parts. In addition, we make all function arguments symbolic. For aggregate type objects, we make all of their scalar fields symbolic. For pointer fields, we make the objects the pointers point to symbolic, and make the addresses concrete. We also concretize function pointers and set the addresses to point to empty body functions. We ran our experiments on an Intel Core i7-M620 2.67GHz machine with 8GB RAM running Ubuntu 14.04. KLEE and programs under test were compiled using LLVM 3.3. Unless otherwise noted, we run KLEE on each program and function under test in the suite up to 1 hour. Our evaluation addresses the following research questions:

1) How effective is don't care analysis in reducing test cases? We find that don't care analysis reduce the sizes of the test cases 50 times in average. (Section IV-A)
2) How efficient is don't care analysis compared to the overall test generation process? We find that DCCSearch in combination with LMSearch has an overhead rate at 48.32%. However, DCCSearch alone introduces negligible overhead. (Section IV-B)
3) How effective is LMSearch in finding more don't care variables (Section IV-C)? We find that DCCSearch is often efficient enough that the use of LMSearch can be optional.

A. Test Reduction Evaluation

GNU Coreutils: Using the setting recommended in [5], symbolic arguments to these programs are provided through the following KLEE option `--sym-args 0 1 10 --sym-args 0 2 2 --sym-files 1 8 --sym-stdout`, which states that the first argument has length 10, the next two arguments have length 2, the file input and standard input, each consists of a string of length 8, and the standard output consists of a string of length 1024. Since a test case for these programs typically consists of only one line, it is natural for us to use the length of the generated strings as a metric for test reduction. Indeed, for string inputs, the developers need to actually scan through the content of the string to locate defects. Thus the smaller the string input is, the easier it is for debugging. Throughout this section, we will also define *test reduction rate* to be the ratio of the test case size and the reduced test case size.

Figure 8 shows the test reduction rate for 100 Coreutils programs using logarithmic scale. Reduction rate is based on the length of the string inputs. On average, 24 test cases are generated for each program. The lowest reduction rate is 48 for the program unexpand, and in average the test reduction rate is 72. The test reduction rate for the entire test suite for all 100 programs is 72.78, as shown in Table I.

SQLite3: The SQLite3 API for C/C++ consists of 214 functions. After removing obsolete functions, functions that do not have any arguments, and functions that use only function pointers as inputs, we end up with 195 functions for evaluation. We use the same settings as suggested in [5] but removing the symbolic command line arguments from the input. Instead, we make all function parameters symbolic. Figure 9 shows the test reduction rate for these 195 functions using logarithmic scale.

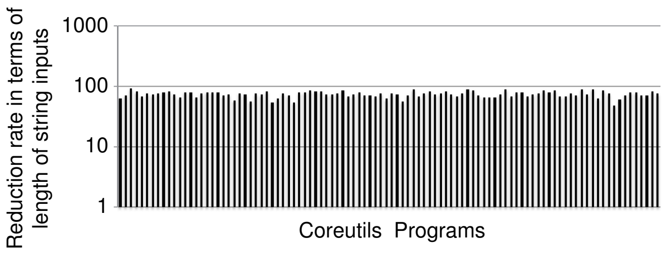


Fig. 8: Test reduction rate for 100 COREUTILS programs. Reduction metric is based on the length of the string inputs.

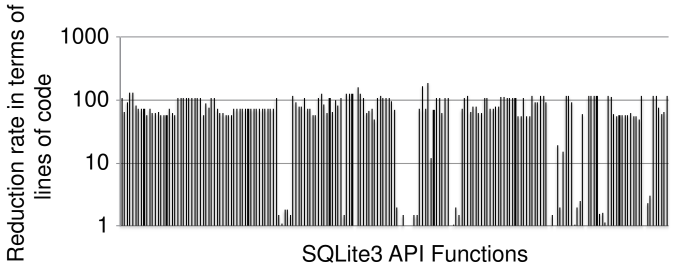


Fig. 9: Test reduction rate for 195 SQLITE3 API functions. Reduction metric is based on the size of generated test scripts.

TABLE I: COREUTILS entire test suite size vs. reduced test suite size. Size is measured in terms of length of string inputs.

Entire test suite size	Reduced test suite size	Reduction rate
2,540,105	34,897	72.78

TABLE II: SQLITE3 entire test suite size vs. reduced test suite size. Size is measured in term of number of lines of code.

Entire test suite size	Reduced test suite size	Reduction rate
152724	2944	51.87

Reduction rate is computed based on the number of lines of code of the generated test cases. As mentioned earlier, for each test case, we generate one assignment statement as one line of code. On average, KLEE generates test cases that consist of 207 lines, and generates 4 test cases for each function within the 1-hour time limit. The average test reduction rate for SQLITE3 API functions is 73 and can be as high as 190. The test reduction rate for the entire test suite of all 195 functions is 51.87, as shown in Table II.

B. Overhead Evaluation

We compare the analysis time of KLEE with don't care analysis and KLEE without don't care analysis using 195 SQLITE3 API functions. We find that the GNU COREUTILS programs involve too many I/O operations to have stable running time. We configure KLEE to terminate after generating 4 test cases using the command `--stop-after-n-tests 4 --dump-states-on-halt=0`. Four is the average number of test cases that KLEE generated for each SQLITE3 function in the 1-hour limit, as mentioned in Section IV-A. We report the running time overhead for two configurations of the don't care analysis algorithm: one uses the combination of DCCSEARCH and LMSEARCH as described in Section III-B, and one uses only DCCSEARCH.

DCCSEARCH+LMSEARCH: Figure 10a shows the running time overhead of DCCSEARCH+LMSEARCH compared to the test generation process for 195 SQLITE3 API functions. Per function, the average overhead is 26.5% and the median overhead is 59.49%. The overhead for the entire test suite is 48.32% (Table III).

DCCSEARCH: Figure 10b shows the overhead of don't care analysis using DCCSEARCH alone on 195 SQLITE3 API functions. Per function, the average overhead is 0.34% and median overhead is 0.0%. The overhead is only as high as 6.9% and is negligible ($< 2.0\%$) in 89% of the functions. The overhead for the entire test suite is 0.66% (Table III).

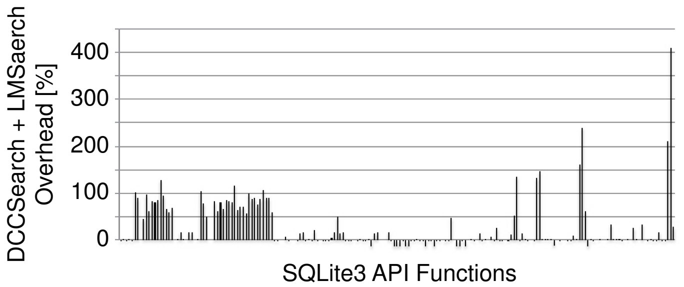
C. LMSEARCH Evaluation

The results of section IV-B show that don't care analysis using the combination of DCCSEARCH and LMSEARCH incurs fairly high overhead to the whole test generation process (48.32%), while using only DCCSEARCH incurs negligible overhead (0.66%). The results reported in this section will show however that DCCSEARCH is often efficient in finding don't care variables enough that LMSEARCH can be optional in many cases. Indeed, Figure 11 shows the relative differences in percentage in the number of care character inputs generated by DCCSEARCH+LMSEARCH and DCCSEARCH for 100 GNU COREUTILS programs. In average, DCCSEARCH alone generates more 2.12% care characters than DCCSEARCH+LMSEARCH. The highest difference is 9.88% for the `su` program, which translates into 0.81 care characters per test. Similarly for SQLITE3 benchmark, DCCSEARCH alone generates test cases that have 0.42% more lines of code than LMSEARCH (Figure 12). The highest difference is 10%, which translates into 0.33 more lines of code per test.

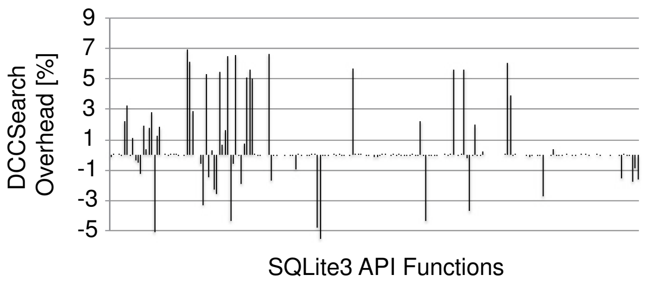
V. DISCUSSION

Our experiments so far show that our test reduction technique is effective on the two selected benchmarks. In this section, we discuss why it works well according to our collected data.

We observe that in the GNU COREUTILS benchmark, the standard input and input file are relevant parameters for only 32 of the 100 GNU COREUTILS programs, while the standard output does not impact test coverage at all. We then remove the standard output from KLEE setting and simply use `--sym-args 0 1 10 --sym-args 0 2 2`



(a) Overhead of DCCSearch+LMSearch



(b) Overhead of DCCSearch

Fig. 10: Overhead of different configurations of don't care analysis algorithm to the overall test generation process for 195 SQLITE3 API functions.

TABLE III: Overhead of different don't care analysis configurations on generating the entire SQLITE3 test suite.

DCCSEARCH+LMSEARCH	LMSEARCH
48.32%	0.66%

`--sym-files 1 8` as input. Figure 13 shows the test reduction rate for this setting using logarithmic scale. On average, don't care analysis still can reduce the test cases a factor of 2.25. We note however that removing standard output from the setting requires domain-specific knowledge. Without this knowledge, the developers can still use the original setting and let our don't care analysis infer this knowledge for free.

Regarding the SQLITE3 benchmark, we noticed that a majority of SQLITE3 functions receive SQLITE3 objects such as `sqlite3` or `sqlite3_stmt` as parameters. Since each of these objects has dozens of fields and only a few fields are actually used in the functions, the test reduction is fairly high. Test reduction in this case is very helpful as it helps to locate faster which fields might be the source of the bugs. We also notice that several functions do not have high test reduction rate, for example, `sqlite3_compileoption_used` has the reduction rate of 1.14 times smaller and `sqlite3_complete` has the reduction rate of 1.81 times smaller. All of these functions receive scalars as function parameters, thus the test reduction rate is smaller. The size of the generated test scripts is also smaller, which is 2 lines and 10 lines on average for the two functions mentioned earlier.

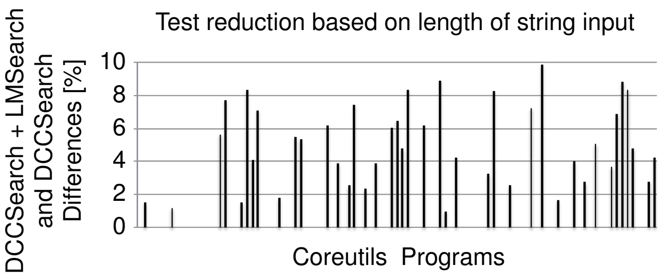


Fig. 11: Relative differences in number of don't care character inputs generated by DCCSEARCH+LMSEARCH and DCCSEARCH alone on GNU COREUTILS benchmark.

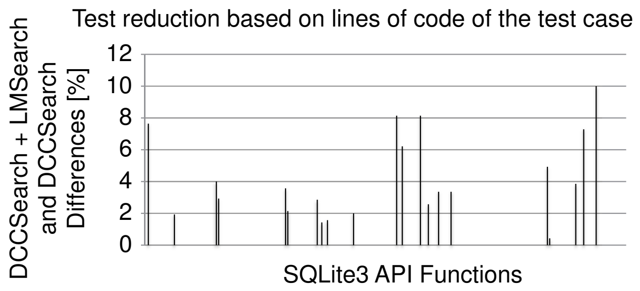


Fig. 12: Relative differences in number of lines of code of test cases generated by DCCSEARCH+LMSEARCH and DCCSEARCH alone on SQLITE3 benchmark.

VI. RELATED WORK

The topic of automated test case generation using symbolic execution has attracted much interest from both academic research and industrial studies in the last decade. An online bibliography [7] currently lists more than 190 publications related to symbolic execution and its applications. Notable work that was pioneer in addressing the major limitations of symbolic execution include DART [17], CUTE [26] and KLEE [12]. DART and CUTE combined concrete execution with symbolic execution to alleviate the limitations in constraint solving and uninterpreted functions, and allow unit test case generation to be completely automated. KLEE introduced the idea of system modeling and was the first tool to show that symbolic execution can generate high structural coverage test cases on real world programs. Industrial implementations of symbolic execution include KLOVER [21] for C/C++, PEX [27] for .NET, SAGE [10] and MAYHEM [9] for x86 code. Recent work including those by SMART [16], TRACER [18] and veritesting [9] has been devoted to addressing the problem of path explosion. The approaches used are based on path merging techniques using function summaries, interpolation, and a combination of static and dynamic symbolic execution. Finally, a different set of work that focus on heuristics for efficient path traversal are those by CREST [11], T. Xie et al. [28] and Y. Li et al. [22].

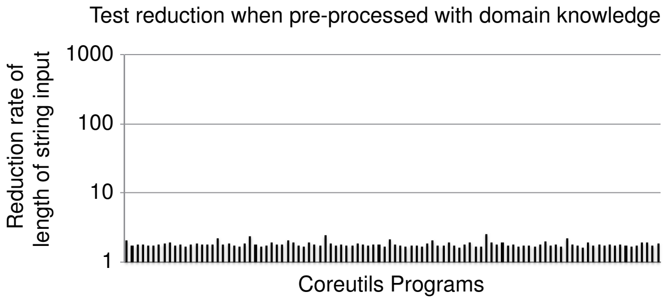


Fig. 13: Test reduction rate for 100 GNU COREUTILS programs without symbolic standard output. Reduction metric is based on length of string inputs.

Our work is also closely related to other work in test case reduction. Zeller and Hildebrandt [29] developed delta debugging, a well-known algorithm for general test case reduction problem and demonstrated it usefulness on string based inputs. Subsequently, Misherghi and Su proposed hierarchical delta debugging, which reduced tree-structural test cases [23]. Domain-specific test case reduction algorithms for compiler bugs include those by BUGFIND [13], LLVM BUGPOINT [6] and C-REDUCE [25]. These algorithms essentially look at the program as a black box and use a predicate function to verify the validity of the test cases. Our work differentiates in the sense that our algorithm is based on analyzing the program structure rather than merely the generated test inputs. Test case reduction work for unit test cases generated from random testing includes those by Lei and Andrews [19] and Leitner et al. [20]. Our work is similar to those in the sense that we also reduce generated test scripts. However, while in other work, the test reduction is conducted after test case generation process, our test reduction approach is integrated inside the test generation process.

VII. CONCLUSION

In this paper, we have presented an automated method for reducing test cases generated by bit vector based symbolic execution test generators. The method is based on a novel analysis entitled don't care analysis, which infers a locally maximal set of don't care variables from a constraint formula. We implemented don't care analysis as an efficient and practical plugin for the KLEE symbolic executor. Initial evaluation on a set of 284 programs and functions showed encouraging result: we were able to generate test cases that are 50 time smaller in size, compared to a guileless test case generation algorithm.

In the future, we would like to apply our analysis to a wider range of applications to gain better statistical results. We also want to explore different techniques to make generated test cases friendlier to human. Generated test cases in general need to be easy to understand, maintain and locate defects.

REFERENCES

[1] https://gcc.gnu.org/bugs/minimize.html.

[2] http://llvm.org/docs/HowToSubmitABug.html.

[3] https://developer.mozilla.org/en-US/docs/Mozilla/QA/Reducing_testcases.

[4] https://www.webkit.org/quality/reduction.html.

[5] http://klee.github.io/klee/CoreutilsExperiments.html.

[6] http://llvm.org/docs/CommandGuide/bugpoint.html.

[7] A. Alipour and S. Bucur. A Bibliography of Papers on Symbolic Execution Technique and its Applications. https://sites.google.com/site/symexbib/, 2014.

[8] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *TACAS'07*, pages 134–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with Veritesting. In *ICSE 2014*, pages 1083–1094, New York, NY, USA, 2014. ACM.

[10] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ICSE '13*, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.

[11] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE '08*, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[12] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[13] J. M. Caron and P. A. Darnell. Bugfind: A tool for debugging optimizing compilers. *SIGSOFT Softw. Eng. Notes*, 15(1):64–65, Jan. 1990.

[14] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[15] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.

[16] P. Godefroid. Compositional dynamic test generation. In *POPL '07*, pages 47–54, New York, NY, USA, 2007. ACM.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[18] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *CAV'12*, pages 758–766, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *ISSRE '05*, pages 267–276, Washington, DC, USA, 2005. IEEE Computer Society.

[20] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE '07*, pages 417–420, New York, NY, USA, 2007. ACM.

[21] G. Li, I. Ghosh, and S. P. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *CAV'11*, pages 609–615, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *OOPSLA '13*, pages 19–32, New York, NY, USA, 2013. ACM.

[23] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *ICSE '06*, pages 142–151, New York, NY, USA, 2006. ACM.

[24] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

[25] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI '12*, pages 335–346, New York, NY, USA, 2012. ACM.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[27] N. Tillmann and J. de Halleux. White-box testing of behavioral web service contracts with PEX. In *ISSTA'08*, pages 47–48, 2008.

[28] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN '09*, pages 359–368, June 2009.

[29] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.