Test Generation to Expose Changes in Evolving Programs

Dawei Qi, Abhik Roychoudhury, Zhenkai Liang
National University of Singapore
{dawei,abhik,liangzk}@comp.nus.edu.sg

ABSTRACT

Software constantly undergoes changes throughout its life cycle, and thereby it evolves. As changes are introduced into a code base, we need to make sure that the effect of the changes is thoroughly tested. For this purpose, it is important to generate test cases that can stress the effect of a given change. In this paper, we propose an automatic test generation solution to this problem. Given a change c, we use dynamic symbolic execution to generate a test input t, which stresses the change. This is done by ensuring (i) the change c is executed by t, and (ii) the effect of c is observable in the output produced by the test t. To construct a change-reaching input, our technique uses distance in control-dependency graph to guide path exploration towards the change. Then, our technique identifies the common programming patterns that may prevent a given change from affecting the program's output. For each of these patterns we propose methods to tune the change-reaching input into an input that reaches the change and propagates the effect of the change to the output. Our experimental results show that our test generation technique is effective in generating change-exposing inputs for real-world programs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools, Symbolic execution*

General Terms

Experimentation, Reliability

Keywords

Software Evolution, Test Generation, Symbolic Execution

1. INTRODUCTION

Regression testing is one of the most commonly known software engineering activities for developing reliable software. In simple terms, it stresses "program changes" as a program evolves from one version to another, checking whether new functionality introduced by the changes is correct and whether the changes result in errors in existing functionality. Often, regression testing involves re-testing using a new test-suite containing both existing test cases and new test cases as the program evolves.

For re-testing with existing test cases, because the test-suite of a program is often huge, it is inefficient to test the changed program with all existing test cases. Most of the past research efforts in regression testing focus on this inefficiency issue and provide solutions via test selection [2, 12] (selecting a subset of the tests to be run) or test prioritization [4, 15] (changing the order in which a set of given tests is run).

However, the evolution of a program often involves addition of new functionality, and thus the test-suite should also evolve with the evolution of the program. In this aspect, the key challenge is to generate test cases related to the changes. Recent work [13, 16, 19] has studied test-suite augmentation for evolving software. The main task in test-suite augmentation is to find new test cases that stress the program changes and affect the program output. Suppose a program P (with a test-suite T) evolves to a program P', i.e., P is *changed* to produce P'. A test-suite augmentation method should generate test cases that make the effect of the *changes* visible in terms of observable program output. If these test cases do not appear in the existing test-suite T, we add them to T.

Let us now examine an intuitive way of generating test cases for stressing program changes. Consider an output variable *out* in programs P and P', and let the inputs of P and P' be in_1, in_2, ..., and in_k. By performing a strongest post-condition computation (using symbolic execution) on program P, we represent the output variable *out* in P as a formula $\varphi(in_1, in_2, \ldots, in_k)$. Similarly, by performing a strongest post-condition computation on program P', we represent the variable *out* in P' as another formula $\varphi'(in_1, in_2, \ldots, in_k)$. We can then solve

$$\varphi(in_1, in_2, \ldots, in_k) \neq \varphi'(in_1, in_2, \ldots, in_k)$$

and the solutions are test cases (assignments of values to inputs $in_1, in_2, \ldots, in_k$) that make the output values different in the two programs.

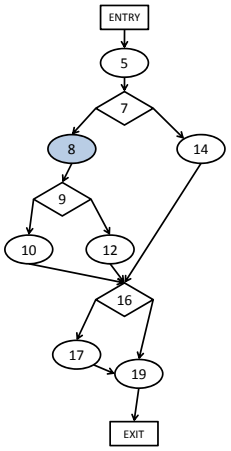
Although the above approach is straightforward, it does not scale. Since we need to perform static symbolic execution on the program (rather than dynamic symbolic execution on an execution path of the program), it is difficult for the approach to scale up to large real-world programs. In this paper, we develop a scalable approach for test-suite augmentation. Our approach builds on the *execute-infect-propagate* (PIE) paradigm [18]: the new tests should (i) execute the program changes, (ii) infect the program state, and (iii) propagate the infection to the output.

From a high level, our approach works in two steps. The first step is to generate an input satisfying the *execute* property in the

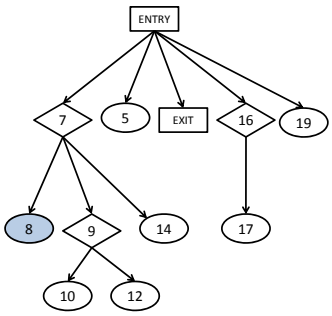
ASE'10, September 20–24, 2010, Antwerp, Belgium.

```
1 int x; /* Input variable */
2 int y;
3 int o; /* Output variable */
4
5 input(x);
6
7 if (x > 0) {
8     y = 3; //change: y = 2;
9     if (x - y > 0)
10        o = y;
11    else
12        o = 0;
13 } else
14    o = -1;
15
16 if (x > 20)
17    o = 10;
18
19 output(o);
```

(a) Example Program



(b) Control Flow Graph



(c) Control Dependence Graph

Figure 1: A motivating example to illustrate our approach.

PIE paradigm. Given a change c (in source code from one program version to another), we find a path that reaches c in the control flow graph. We then perform symbolic execution along the path to find an input t that makes the program execute the path leading to c. The second step of our approach aims to generate test cases that satisfy the *infect* and *propagate* properties, in addition to the *execute* property. Since any infection in program states is reflected as different variable values (after the change) in the two program versions, we observe that state infection and propagation may be avoided if (i) variables affected (directly/indirectly) by a program change are defined but not used, or (ii) the uses of affected variable cannot propagate the change effect forward (by affecting other variables). If an assignment of some affected variable v is not used in the execution of test t, we find a new test t' (aided by symbolic execution along a path) that can execute the uses of v. If the use of variable v does not propagate the change effect forward, we find a new test t' that can propagate the effect in v.

The preceding describes our method in a nutshell. The key to the method's efficiency lies in our strategy in avoiding symbolic execution on programs. Our approach performs every symbolic execution along a program path. Note that symbolic execution along a program path has additional overhead in enumerating and searching for the "right" path. We use various analysis methods to guide us to the "right" path. When trying to execute the change, the shortest path in the control dependency graph guides us to efficiently locate and construct a path to the change. When trying to propagate the change effect, we identify the reasons for which the propagation terminates, and propagate the change effect to program output while detecting branch correlations on-the-fly (which allows us to avoid infeasible program paths).

Performing symbolic execution along a path also helps us avoid the memory alias problem: since the symbolic execution is along a program path, all memory references are disambiguated.

In our experiments, we tried our test-case generation approach on two programs: `tcas`, a small program with multiple versions (each encoding a different change) from the SIR repository [3] and `libPNG`, a large-scale library for manipulating PNG images (27977 lines of code). For both the subject programs, our method successfully generated test cases that stress the changes (by producing different program outputs) for almost all the program versions.

2. MOTIVATING EXAMPLE

In this section, we motivate the problem of test case generation in evolving programs with an example.

Figure 1 shows an example of evolving programs. In the example program shown in Figure 1(a), the variable x is the input variable and the variable o is the output variable. The change is at line 8, where the assignment of variable y is changed from 3 to 2. We now need to synthesize a test case that stresses this change. Figure 1(b) and Figure 1(c) show the control-flow graph (CFG) and control-dependency graph (CDG) of the example program, respectively, in which the changed statement is marked in dark color.

In order to test the change, we need test cases that can drive the program to the changed statement in P', and can result in an output of P' different from that of P.

Reaching the change.

Suppose we have the following test case for the original program P: *x = -1*. The execution trace of the test case *x = -1* in the changed program is {*5, 7, 13, 14, 16, 19*}, which does not cover the change at line 8. Note that there are two branches in the trace, namely line 7 and line 16. If we execute these branches differently, we may drive P' to reach the change. However, some of the branches, such as line 16, do not help to reach the change. We use the CDG to identify such branches. From the CDG in figure 1(c), we can see that node 7 can determine whether the change (node 8) is executed, while node 16 cannot. So we want to execute branch 7 differently. This analysis (for finding which branch to evaluate differently) can be automated via traversal of the CDG. In this example, we construct the formula $(x > 0)$ by flipping the evaluation of the branch in line 7. Solving this formula $x > 0$, we get an input, say *x = 1*, that stresses the change.

Affecting program output.

Using the new input *x = 1* to test both program versions, we get the same program output 0. Even though the change is executed, its effect is not propagated into the program output. We can see that

propagation of the change effect (*y = 2*) stops at the branch in line 9 — this branch is evaluated to false in both the program versions for *x = 1*. To propagate the effect of the program change past this branch, we need the branch to be evaluated differently in P and P', which is expressed in the following symbolic formula.

$$(x > 0) \wedge (x - 2 > 0) \wedge \neg(x - 3 > 0)$$

An x satisfying this formula will execute the change, *and* evaluate the branch in line 9 differently in the two program versions. This gives us the answer *x = 3*. Executing P and P' using the input *x = 3*, we find that the program output is different in both program versions. Thus, we have generated a test case which executes the change, and propagates its effect to the program output.

3. OUR APPROACH

Our goal is to generate test cases that make the effect of software changes observable through difference in program outputs. A test case for regression testing should drive the changed program to execute the changes, and the program states affected by the changes should result in difference in program outputs.

To meet the above requirements, our approach uses symbolic execution on program traces to guide the exploration of program paths. The exploration in our approach is guided by the execute-infect-propagate philosophy (execute the change, infect program state and propagate the infection to output), instead of program path coverage. Our approach is divided into two steps. First, given a program change, we use symbolic execution to find the constraints on program input variables that need be satisfied to execute the change. Second, given the constraints (and sample test inputs) generated in the first step, we tune the sample test inputs into test cases that not only execute the change but also propagate the effect of the change to the program output.

We now explain these two steps in Section 3.1 and Section 3.2, respectively.

3.1 Driving the program to reach the changes

Our approach iteratively constructs the change-reaching test inputs using symbolic execution. In each iteration, a new test input is generated to drive the program execution closer to the targeted change. This process continues until we get a test input that executes the given program change.

The basic intuition of this step is as follows. We run the test-suite T_P of the old program P on the changed program P'. If any test $t \in T_P$ executes the changed statement $stmt$ in the changed program P', we return t. Otherwise, we collect the path condition of P' when it processes t and manipulate the path condition to generate a new input that "advances" the execution in P' towards the changed statement $stmt$.

Assume the path corresponding the test input t is π, which does not reach the changed statement $stmt$, and π's path condition is $\psi_1 \wedge \psi_2 \wedge \ldots \psi_m$. We look for inputs that can make P' deviate from π to reach $stmt$. The deviation can be made at any of the branches along π. If an input makes P' deviate from π at the k-th branch, it must satisfy the following condition:

$$\psi_1 \wedge \ldots \wedge \psi_{k-1} \wedge \neg\psi_k$$

That is, the new input satisfies the first $k-1$ branching conditions of π, but does not satisfy the k-th branching condition. If it is satisfiable[1], we can generate an input t_i satisfying the formula. The new input t_i leads to a different path, which can potentially make P' reach the target.

[1]Note that this formula may be unsatisfiable.

Algorithm 1 Reaching the change

```
1: Input:
2: P P' : original and modified program
3: T_p : The existing test-suite for P
4: Output:
5: t_new: A test case that reaches the difference between P and P'
6: unexpanded = ∅
7: S = ∅
8: stmt = ... // this is the changed statement
9: CDG_p' = computeCDG(P')
10: G_stmt = computeDistGraph(CDG_P', stmt)
11:
12: // step1: run the existing test-suite
13: for all t ∈ T_p do
14:     ret = Execute(P', t)
15:     if ret ≠ null then
16:         return ret
17:     end if
18: end for
19: // step2: construct new test case
20: while unexpanded ≠ ∅ and not timeout do
21:     select φ ∈ unexpanded with minimum distance
22:     remove φ from unexpanded
23:     let φ = (ψ_1 ∧ ψ_2 ∧ ... ∧ ψ_{k-1} ∧ ψ_k)
24:     construct θ = (ψ_1 ∧ ψ_2 ∧ ... ∧ ψ_{k-1} ∧ ¬ψ_k)
25:     solve θ
26:     if θ is satisfiable then
27:         let t_θ be an input that satisfies θ
28:         ret = Execute(P', t_θ)
29:         if ret ≠ null then
30:             return ret
31:         end if
32:     end if
33: end while
34: return null
35:
36: procedure Execute(P', t)
37:     execute P' with input t
38:     let f = (ψ_1 ∧ ψ_2 ∧ ... ∧ ψ_m) be the path condition
39:     for all i from 1 to m do
40:         φ_i =def ψ_1 ∧ ... ψ_i ∧ ψ_{i+1}
41:         if dist(ψ_{i+1}) = 0 then
42:             return t
43:         end if
44:         if dist(ψ_{i+1}) ≠ ∞ and φ_i ∉ S then
45:             S ⋃= φ_i
46:             unexpanded ⋃= φ_i
47:         end if
48:     end for
49:     return null
50: end procedure
```

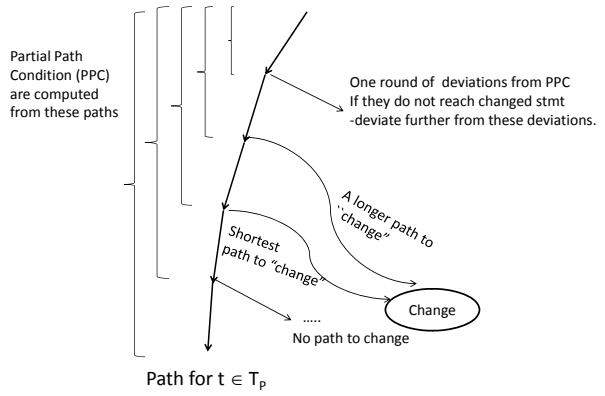


Figure 2: Summary of Algorithm 1

One of the major challenges faced by this intuitive solution is to handle the large number of branching conditions. The intuitive solution cannot handle large-scale software with limited time and computing resources. To address this problem, we observe that the negation of certain branching conditions cannot help to reach the change. For example, in our motivating example described in Section 2 (illustrated in Figure 1), negating the branch condition at line 16 will not help to drive the execution closer to the change. Therefore, we need to measure whether the negation of a branch condition can drive the execution in P' closer to the change. In our approach, the measure of closeness or proximity of a path π (w.r.t. the change $stmt$) is given by the *length of the chain of control dependencies that need to be traversed from* π *in order to reach the change* $stmt$.

In this step, we are working with program paths that have not yet reached the change $stmt$. The path conditions of such paths are referred to as *partial path conditions* (abbreviated as PPC) in our terminology. The term "partial" specifically emphasizes that the change has not yet been reached. Also, note that the path condition of any program path π is a quantifier-free first order logic formula satisfied by all test inputs that drive P' to execute along π. The notion of partial path conditions is central to our approach. Since we are trying to construct a feasible program path (*i.e.*, exercised by at least one program input) in the changed program P' that terminates at the given change $stmt$, our method works by constructing a path that gets "closer" to the change. These partial path conditions are then explored to see whether they can reach the changed statement $stmt$, otherwise we explore further deviations from these partial path conditions of t_i in a similar way (*i.e.*, by negating the last branch condition). This, in essence, is the heart of our algorithm for reaching the changed statement.

Algorithm 1 captures the core method for generating a change-reaching test case. The inputs to the algorithm are: the old program P, the changed program P', and the test-suite T_P for program P. There is a single change between the programs P and P', that is, P and P' differ via a unit change statement $stmt$. The output of the algorithm is a test input t_{new} that executes this changed statement $stmt$. The elements in set S and $unexpanded$ are PPCs (partial path conditions). Set S is used to maintain all the PPCs we have seen to avoid redundancy. Set $unexpanded$ contains all the PPCs that have not been tried out. In the algorithm, the CDG of P' is first computed, then the distance from each node to $stmt$ is computed from the CDG of P'. This is shown by the function $dist$ in the algorithm. Thus, let b be a program branch in the changed program P' whose condition is ψ. Then $dist(\psi)$ is the shortest path from b to the changed statement in the static inter-procedural control dependency graph of P'. The distance for a PPC φ_i ($\varphi_i = \psi_1 \wedge \ldots \psi_i \wedge \psi_{i+1}$) is the same as the distance of the last branch condition in φ_i, that is, $dist(\varphi_i) = dist(\psi_{i+1})$.

The algorithm first looks for a change-reaching test case in the existing test-suite T. If no existing test case can reach the target, our algorithm iteratively constructs such a test case in the second step. In each iteration of the second step, we choose a PPC that is closest to the target from the PPCs that have not been tried out. By deviating at the last branch of the path, we get closer to the target in each iteration.

From the description of Algorithm 1, we see that it is similar to the generational search strategy used in SAGE [7]. In SAGE, given a path π for a test, new paths are explored by negating each branch in π, similar to our algorithm. The main difference between our method and SAGE is in the way that we choose the branches to negate. Given a set of partial path conditions (obtained from the path condition $\psi_1 \wedge \ldots \wedge \psi_m$ of a program path)

$$\{\psi_1 \wedge \ldots \wedge \psi_i \wedge \psi_{i+1} \mid 0 \leq i < m\}$$

we use the *distance* between ψ_k and the changed statement $stmt$ to prioritize the selection of the branch condition to negate. The distance between ψ_k and $stmt$ is defined as the weighted shortest path from the program branch contributing to ψ_k to the changed statement $stmt$ in the static inter-procedural control dependency graph of the changed program P'. If the k-th branch condition has the smallest distance from $stmt$, it will be negated first.

3.2 Propagating the effect of a change to program outputs

Executing the changed statement is not sufficient for reflecting the change in the output. A change should first affect some program states, and the effect of the change should be seen from the output (via propagation of the affected states). In reality, a program can have a large number of paths. Therefore, any propagation technique is either path insensitive and hence imprecise, or path sensitive but not scalable to large programs. In this section, we propose a practical technique for propagating the effect of the change in an iterative way.

Assumptions. Before we describe our method for propagating effects of the program change to output, we make some assumptions about the programs we work with.

- *Assumption 1*: A variable defined in a program must be used somewhere in the program.

  If we have a program that does not satisfy this assumption, we can use def-use analysis to eliminate all the variables that are never used. Since we only eliminate the variables that are defined but never used, this transformation will not change the program behavior.

- *Assumption 2*: All statements are reachable.

  If there are unreachable statements, we can eliminate these statements without affecting the program behavior.

Why a change may not propagate to output. To build a method that propagates the effect of a change to output, we investigate common reasons for which propagation failed to reach the output. We use P and P' to denote the original program and changed program respectively. The execution trace of input t in program

P is denoted as $trace(P, t)$. For a variable definition statement instance s', we say the defined variable is "*affected*" when it has different value than the value defined in s. Statements s and s' are aligned statements in $trace(P, t)$ and $trace(P', t)$ respectively. Note that the effect of a change could have propagated to certain distance before the propagation stops. So we do not need to start the propagation from the change each time, we only need to intervene when the effect stops propagating in the execution. The change cannot affect the output when none of the affected variables can affect the output. Suppose an affected variable v is defined in s' in $trace(P', t)$, the reasons that the different value in v stops propagating, may now be enumerated as follows.

- The uses of v are never executed before v is redefined in $trace(P', t)$. As an example, consider the program in our example.

  ```
  y = 2; /* originally y = 3; */
  if (x > 0){ o = y;} else { o = 0;}
  ```

  Here, the changed statement affects the definition of y. However, in the execution trace for the input $x = 0$ (and thus $x > 0$ is false), the use of this definition is never executed. The output variable *o* is not affected by the change.

- Uses of v are executed before v is redefined in $trace(P', t)$, but the use cannot result in other affected variable. As an example, consider the following program. Let x be the input, and o be the output of this program fragment.

  ```
  y = 2; /* originally y = 3; */
  if (x - y > 0){ o = y;} else { o = 0;}
  ```

  Here y is the variable whose definition is affected by the change. The use of the definition is also executed in the form of the condition `(x-y > 0)`. However, for input $x = 0$, it does not make a difference in the control flow and the subsequently calculated output value for o.

Propagating changes to the output. We handle the two reasons for which the effect of a change may not propagate to the program output as follows. If the uses of the affected variables are not executed in an execution trace π, we drive the program execution towards the use statements. This is achieved via a method similar to Algorithm 1, where we set the use statements as the target, instead of the changed statement.

In the second case, the uses of the affected variables are executed in the trace but no program variables are affected by the use. Suppose the affected variable v is defined in statement instance s', and v is subsequently used in statement instance m'. Obviously, m' was not able to propagate the effect of v forward. According to the type of m', we use the following steps to propagate the effect in v.

- If m' is a variable definition statement, we compute the so-called "*transfer condition*" [11] (Definition 1 below) of the statement. Intuitively, the transfer condition of an expression is the condition under which the value of the expression will be different if one of its operands is different. Given the transfer condition, we use it in symbolic execution to compute a test input which propagates the effect of the change, while following the same path.

  DEFINITION 1 (TRANSFER CONDITION). *The transfer condition for $exp = oprand1\ op\ oprand2$ with respect to $oprand1$ is the condition under which exp has different value if $oprand1$ has different value.*

  For example the transfer condition of `x + y` is *true* since if either operand is different, the sum is different. On the other hand, the transfer condition of `x * y` is $y \neq 0$ for a change in x.

- If m' is an branch and v is used as the condition in m', we compute the condition which makes m and m' to be evaluated *differently* in the two programs P and P' in order to produce different outputs in P and P' (m and m' are aligned statements in $trace(P, t)$ and $trace(P', t)$). We use symbolic execution to find an input that reaches m in programs P and m' in P' and then evaluates m and m' differently in the two programs.

Algorithm for Propagating Change. The algorithm for propagating change effects is shown in Algorithm 2. The algorithm iteratively calls procedure *Propagate* to construct a new input that can propagate the change effect forward. The procedure *Propagate* first executes P and P' using the input t. Then it analyzes the execution traces of P and P'. An important concept here is the *change effect propagation tree (CEPT)* calculated by **CPTree** at line 11.

DEFINITION 2 (CHANGE EFFECT PROPAGATION TREE). *Given a unit change, a CEPT CT is defined as follows. The nodes of a CEPT are statement instances in the changed program P'. There is an edge from α' to β' in CT if and only if (i) β' is dependent on α' (either control dependence or data dependence) (ii) the operands of β' have different values than those of β, where β is the corresponding statement of β' in P. Each leaf node in the CEPT is a place where the change effect propagation terminates.*

Note that the CEPT defined above is a polytree [8]. A polytree is a restricted DAG (Directed Acyclic Graph). While a DAG allows multiple undirected paths between two nodes as long as they do not form directed cycle, a polytree allows at most one undirected path between any two nodes. Compared to nodes in a tree, a node in a polytree can have more than one parent node, which represents that more than one operand in a statement are affected by the change.

Identifying terminating locations of effect propagation. The CEPT is computed by dynamic forward slicing. During slicing, our approach compares the operand values of corresponding statement instances in both programs to determine whether the change effect has stopped propagating. If the operand values of a statement instance s' in P' are the same as those of the corresponding statement instance s in P, s' will not be included in the propagation tree, because it will not cause differences in the output. In other words, the change effect does not propagate to s'.

When comparing the operand values of corresponding statement instances in both P and P', if s' of P' has no corresponding statement instance in P, the operands for s' are treated as different from those in P. To compare the operand values, we use trace alignment to find the corresponding statement instance s (in execution trace of P) of s' (in execution trace of P'), which is the $align(P, P', t)$ function in Algorithm 2. For simplicity, we use $sat(\varphi)$ to represent an input instance that satisfies φ.

Propagating change effects further. According to the type of the leaf nodes in the change effect propagation tree, we use different methods to drive the propagation of the change effect forward. If an affected variable is defined but never used, the procedure *PropNouse* is called. In this procedure, we first use def-use analysis to identify all the use locations of the defined variable, and use our Algorithm 1 to reach at least one of these locations.

If an affected variable is used but does not propagate the effect forward, according to the type of the use statement s', two different procedures are used for propagation. If the statement s' is a variable-definition statement and the defined variable is used, it can only become leaf node (of CEPT) when the transfer condition is not satisfied. Procedure *PropTransfer* is invoked in this case. In this procedure, we first use symbolic execution to compute the transfer condition TC. Then we get a new input that satisfies TC by solving the formula $f' \wedge TC$ where f' is the partial path condition up to s'. If an affected variable is used as the condition in a branch, it becomes a leaf node when the branch is evaluated the same in both versions. In this case, we use *PropCjmp* to execute the branch differently in two versions. If the procedure *Propagate* cannot generate a change stressing test input by analyzing the new program version, we apply *Propagate* to the old program version.

4. IMPLEMENTATION

We implemented our approach on the x86 platform based on the BitBlaze [14] binary analysis framework. We show the component view of our implementation in Figure 3, where the components used in our solution are shown as boxes, and the data used by the components are shown as italic labels of edges. Some of the important intermediate data are shown in ovals.

SMT formula solving is used extensively throughout our approach. We used the Boolector SMT solver [1] for all our formula solving tasks.

4.1 Architecture of our implementation

Reaching changes.

The top portion of Figure 3 illustrates the implementation of the first step of our approach: finding inputs to reach the change. Our approach first computes the static control-flow dependency graph (CDG): it uses the ERESI tool [5] to generate the static CFG, and then uses our module *CDG builder* to compute the inter-procedural CDG and distance graph from the static CFG. The distance from a node v to the target is defined as the shortest path from v to $target$ in weighted CDG. In a weighted CDG, auxiliary edges (such as function call to the start of the called function) are associated with weight 0. All other edges have weight of 1. We use Dijkstra's algorithm to compute the distance of all nodes (to the change) using one pass of the algorithm.

Next, our approach iteratively constructs an input to reach the change. Given the binary P' and a test case t, our approach generates an execution trace of P' using BitBlaze's TEMU component. TEMU is a whole-system emulator based on QEMU [10]. It emulates a PC system, which runs operating systems such as Windows and Linux. TEMU supports logging instructions executed in the emulated PC and tracking instruction operands that are dependent on program inputs (tainted operands) using taint analysis. Next, our approach uses BitBlaze's analysis component, VINE, to generate the path condition of the execution trace. The path condition is represented by VINE's intermediate language.

With the CDG and path condition, our approach uses our *change-reaching input generation* module to select a branching condition to negate based on the distance to the change statement in the CDG. It then generates a new input that drives P' to execute closer to the change. If P' reaches the change using the new input, our approach continues to the next step. Otherwise, the above process is repeated using the new input until it generates an input that leads P' to execute the changed statement.

Algorithm 2 Propagate the change effect

Input:
t: a change reaching input
P P': original and modified program
Output:
t_{new}: a input that have different output in P and P'
procedure $Propagate(P, P', t)$
  $align(P, P', t)$
  let $stmt$ be the difference between P and P'
  execute t in P' and P to get the execution traces
  $T = CPTree(stmt, P', P, t)$
  for all leaf node s' in CT **do**
    if s' is a variable definition statement **then**
      if variable defined by s' is not used in t's trace **then**
        $ret = PropNouse(s')$
      else
        $ret = PropTransfer(s')$
      end if
    else
      $ret = PropCjmp(s')$
    end if
    if $ret \neq null$ **then**
      return ret
    end if
  end for
  return $null$
end procedure
procedure $PropNouse(s')$
  $U = useSet(s')$ // All *first* uses of the definition of s'
  for all $u \in U$ **do**
    execute Algorithm 1 using u as the target
    let ret be the return value from Algorithm 1
    if $ret \neq null$ **then**
      return ret
    end if
  end for
  return $null$
end procedure
procedure $PropTransfer(s')$
  compute the transfer condition TC for s'
  let the partial path condition up to s' be f'
  if $f' \wedge TC$ is satisfiable **then**
    return $sat(f' \wedge TC)$
  else
    return $null$
  end if
end procedure
procedure $PropCjmp(s')$
  let s be the corresponding statement instance for s' in P
  let the partial path condition up to s' for t in P' be $f' = \psi'_1 \wedge \ldots \psi'_{i'} \wedge \psi'_{i'+1}$, the path condition up to s for t in P be $f = \psi_1 \wedge \ldots \psi_i \wedge \psi_{i+1}$
  let $\varphi = f \wedge \psi'_1 \wedge \ldots \psi'_{i'} \wedge \neg\psi'_{i'+1}$
  let $\varphi' = f' \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$
  if φ is satisfiable **then**
    return $sat(\varphi)$
  else if φ' is satisfiable **then**
    return $sat(\varphi')$
  else
    return $null$
  end if
end procedure

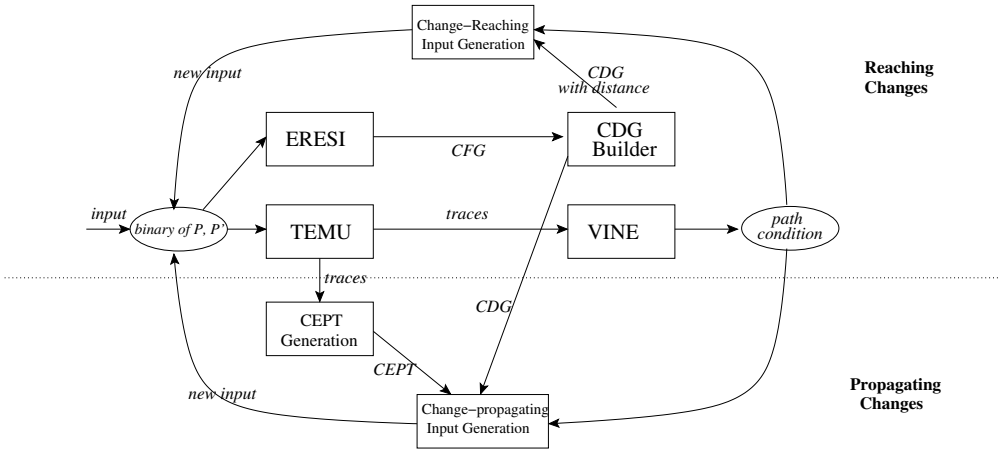


Figure 3: Component view of our approach (CEPT = Change Effect Propagation Tree, CDG = Control Dependency Graph).

Propagating effects of changes.

The bottom portion of Figure 3 illustrates the implementation of the second step of our approach: finding inputs to propagate the effects of the change to the program output. If the input generated in the previous step (the input that reaches and executes the change) cannot affect the output, our approach uses it to generate an execution trace of P' and compute the path condition. Then, our approach generates the Change Effect Propagation Tree or CEPT (refer Def. 2) to decide how to further propagate the change effect towards the output.

As described in our algorithm 2, trace alignment is needed to compute the CEPT. Our approach views each trace as an instruction sequence, and aligns the traces of P and P' using minimum editing distance. From the alignment result, we compare the operand values of P' with the corresponding instructions in P, which is required by our approach to decide whether a node is a leaf node in computing CEPT. For def-use analysis, we implemented a simple def-use analysis module without pointer aliasing support.

With the above information, our approach constructs the CEPT by performing forward slicing on the x86 instruction trace. Each instruction is treated as a statement in the slice. One of the practical challenges is caused by memory and register allocation in different program traces. The memory addresses of the same variable in two binary versions are often allocated differently by compiler or loader. Therefore, when such addresses become the instruction operands, the aligned instructions will have different operand values. For example, `mov EAX, [EBX]` will have different EBX value if the variable pointed to by the EBX is allocated at different addresses in the traces of P and P'. However, the difference in memory address does not imply a different program state, which is defined by the contents of variables, instead of addresses of variables. Similar issue happens with the stack registers ESP and EBP. To address this problem, our approach does not treat difference in memory address operands and stack register operands as different program states.

4.2 SMT solving optimizations

We note that in Algorithm 1 (for constructing an input which reaches the change), many of the formulae $\psi_1 \wedge \ldots \wedge \psi_{k-1} \wedge \neg\psi_k$ constructed for reaching the changed statement may be unsatisfiable. For example, when there is no dynamic data dependence chain between ψ_k and input variables (along the path which results in the partial path condition $\psi_1 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$), the formula $\psi_1 \wedge \ldots \wedge \psi_{k-1} \wedge \neg\psi_k$ is unsatisfiable. Naturally there is no point in submitting such formulae to the SMT solver. In this way, we reduce a significant amount of SMT queries, which leads to more than 90% reduction of SMT solving queries in our experiments.

4.3 Handling branch correlations

Most of the SMT formulae used in our algorithms are based on path condition. We noticed some branches along the path could be correlated. Suppose the branch we are trying to negate is b_β, there is an earlier branch b_α that is correlated with b_β. This correlation could make the constructed formula unsatisfiable. We handle the common cases of "immediate conflicts" between (branch, branch) and (assignment, branch) pairs as follows.

1. `if(x>2){ ... // x is not modified here}`
   `if(x<0){ //target }`

   Given a trace for say `x == 3` which evaluates the first branch to true and the second branch to false, suppose we want to flip the evaluation of the second branch statement to reach the target. However evaluating `x > 2` to true and then `x < 0` to true constitutes an infeasible path in the control flow graph.

   Solution: When we solve formula $\theta = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \neg\psi_k)$, if the branch corresponding to ψ_k is b_k, we perform a backward slicing on the trace from b_k. All the branch conditions that are not in the slice are removed from the path condition θ. In this way, we keep all the branches that are relevant for reaching b_k, and we also keep all the statements that are used for computing the branch condition.

2. `if(x>0){ y = 1; } else{ y = 0; }`
   `if(y){ //target }`

   For the input $x = 0$, we can see y is set to 0 from the execution trace. To reach the target, we need to negate the first branch that the definition of y is dependent on.

   Solution: If we find a branch condition ψ_k is not "tainted" (dependent on the input via a chain of data dependencies), we cannot directly negate this branch. Suppose the last definition of ψ_k's variables in the trace is def, we use backward slicing to find all tainted branches that def is dependent on.

By negating one of these branches, we may evaluate ψ_k differently.

5. EVALUATION

To examine the efficacy of our approach, we evaluated our approach using two real-world programs. In this section, we report our empirical evaluation results.

5.1 Experience with `tcas`

The first program we used to evaluate our approach was `tcas` from the SIR repository [3]. `Tcas` is an aircraft collision avoidance system. It has an original version (the golden program) and 41 changed versions with seeded bugs, exactly one line of a change for each bug. The `tcas` program from SIR reads inputs from command line, we modified the program to read inputs from a file. To stress our test generation method fully, we took an initial test-suite with only one randomly generated test case in this case study. Then we apply Algorithm 1 on this test-suite to generate test cases reaching the change.

Our technique uses 80 runs to reach all the 41 changes, about two runs to reach one change on average. Out of the 41 versions, in 8 versions, the inputs generated by Algorithm 1 already produced different program outputs; thus change effect propagation is not needed in these cases. These inputs are returned by our approach as the test cases to augment the test-suite.

The remaining 33 buggy versions of `tcas` needs to go through change effect propagation (as shown in Algorithm 2), to generate test cases. Our approach successfully generated test cases that show different program outputs (w.r.t the original `tcas` program) in 31 out of the 33 program versions. In the remaining two program versions, because of incorrect program alignment at line 7 of algorithm 2, the CEPT was not computed correctly. The node where change effect propagation terminates was not identified as a leaf node in the incorrect CEPT.

Now we discuss two cases where change effect propagation are needed in `tcas`. Through these examples, we show that how our techniques propagate the change effect forward towards the output.

Affected variable defined but not used.. Figure 4 shows an example in version 3 of `tcas`, in which the change effect cannot propagate because the affected variable $intent\_not\_known$ is defined but not used. Note that in the example code, one line of source code is treated as multiple instructions in our technique. In the example, because the operator is changed from $\&\&$ to $||$, the variable $intent\_not\_known$ is evaluated to different values in two different versions in execution. However, because the value of $tcas\_equipped$ is false, the variable $intent\_not\_known$ is never used after its definition (note that a $\&\&$ is defined using short circuit evaluation, that is, if the first operand is false, the second operand is not used). To propagate the effect of the change, we employ our algorithm 1 to reach the statements where $intent\_not\_known$ is used. Algorithm 1 negated the value of $tcas\_equipped$ to execute the condition test on $intent\_not\_known$.

Propagation stops because of branches.. From our experience in the experiments, it is very common that the propagation terminates because of a branch is evaluated similarly in both versions. Figure 5 shows the case in version 13 of `tcas`. The value compared with $Own\_Tracked\_Alt\_Rate$ is changed from 600 to 700. Only when $Own\_Tracked\_Alt\_Rate$ is in $(600, 700]$, the effect of the change propagates. Through symbolic execution, our technique found a value 604 for $Own\_Tracked\_Alt\_Rate$ such

```
//original version
intent_not_known = Two_of_Three_Reports_Valid
 && Other_RAC == NO_INTENT;
alt_sep = UNRESOLVED;
if (enabled && ((tcas_equipped
 && intent_not_known) || !tcas_equipped))

//changed version
intent_not_known = Two_of_Three_Reports_Valid
 || Other_RAC == NO_INTENT; /* logic change */
alt_sep = UNRESOLVED;
if (enabled && ((tcas_equipped
 && intent_not_known) || !tcas_equipped))
```

Figure 4: Variable $intent\_not\_known$ defined but not used

```
//orginal version
enabled = High_Confidence
 &&(Own_Tracked_Alt_Rate <= 600)
 &&(Cur_Vertical_Sep > MAXALTDIFF);

//changed version
enabled = High_Confidence
 &&(Own_Tracked_Alt_Rate <= 700)
 &&(Cur_Vertical_Sep > MAXALTDIFF);
```

Figure 5: Propagation stops because of branches

that variable $enabled$ is evaluated to different values in two versions.

Comparison with [16].. We compare our results with [16] — the only work that generates test cases to stress program changes. Other research efforts on this topic, such as [13], generate criteria for propagating effects of changes, but they do not generate test cases to reach *and* stress a program change. Therefore, we cannot compare our experimental results directly with those of [13].

In [16], they build their work based on PIE model. They provide heuristics to avoid exploring paths that (i) cannot lead to *execution* of the change (ii) cannot lead to state *infection* (iii) cannot lead to affected state *propagation*.

To compare with [16], we used the result from first 11 changed versions of the `tcas` program, the same versions used in the evaluation of [16]. For each version, our technique started with a random generated input, and interactively generated new inputs to reach the change. The technique in [16] used 95 runs in total to reach all the 11 changes. In contrast, our technique used only 32 runs to reach all the 11 changes. Note that we compare the number of runs for reaching the change, not the number of runs for propagating the change effect to the output. This is because, [16] does not report the number of runs for propagating the effect of the change to the output. Our technique can generate change-reaching inputs with much fewer runs, because the path exploration in our technique is guided by a target. We use the notion of distance in the control dependency graph to prioritize exploring shorter paths to the change.

In addition to the number of runs reported above, our technique also made less number of calls to the SMT solver. We used data tainting method to identify branches that cannot be executed differently, as is described in Section 4.2. In our experiments, we found that this optimization led to significant reduction in the number of calls to the SMT solver. More than 90% branch conditions are not tainted, and thus can be eliminated, in both of our case studies.

5.2 Experience with `libPNG`

In our second case study, we studied the changes between two versions of the `libPNG` program. `LibPNG` is a open-source li-

```
//original version
png_byte red_high =
 (trans_value->red > 8) & 0xff;

//changed version
png_byte red_high =
 (trans_value->red » 8) & 0xff;
```

Figure 6: An example change from libPNG

brary for manipulating PNG image files. It supports almost all the features of PNG file format. We used two consecutive versions from libPNG, v1.2.20 and v1.2.21. *Each version has a large code base, running into around 28000 lines of code.*

We first remove all the obvious syntactic changes that do not affect the semantics of the program. After removing these changes, we are left with 10 changes. Each of these 10 changes are independent. Therefore, we can construct intermediate versions of libPNG by applying the 10 changes to version v1.2.20 one by one. We use c_i to denote change i and use v_0 to v_{10} to denote the intermediate versions (Version v_0 is v1.2.20 and v_{10} is v1.2.21). Version v_i is obtained by applying c_i on Version v_{i-1}. Because the changes are not correlated, if a change c_i can affect the output in v_{i-1} and v_i, it can also affect the output in v1.2.20 and v1.2.21.

Instead of randomly generating test inputs, we used the PngSuite [17] as the test-suite for evaluating Algorithm 1 on libPNG. PngSuite is a large collection of PNG files to test PNG applications. The creator of PngSuite aims to represent all the PNG formats when the suite was created in 1998. Because libPNG has been evolving with the evolution of PNG specification, some new features in libPNG cannot be fully tested by PngSuite any more. This is the exact situation where test-suite augmentation is needed due to program evolution.

LibPNG comes with a test driver to show how the library should be used. We modified the test driver to make the changes statically reachable. We tried to make the changes to the test driver minimal.

Eight out of all ten changes were reached by existing test cases in the PngSuite. For the remaining two changes, our algorithm was able to construct new PNG files that can drive the execution to the changes. A PNG file consists of multiple chunks with different information. Each chunk contains chunk type, chunk length, checksum, and the chunk data. Most functions in libPNG are chunk-specific. For example, a function for handling chunk of type iTXt is only called when there is a chunk of type iTXt in the input PNG file. These two changes appeared in functions that handle the iTXt chunk type. Since iTXt only appears in v1.2 of PNG specification, which was released in 1999, no PNG files in PngSuite (created in 1998) were able to test these two changes. Our algorithm automatically generated test inputs of the iTXt type.

After finishing Algorithm 1 for all the ten changes, we got ten change-reaching inputs (one for each change). Each of these ten inputs can only guarantee the corresponding change being executed, but they may not necessarily affect the program output. In fact, we found that out of the ten change-reaching inputs, seven affect the program output (that is, the output is different for these inputs in the two program versions), and the remaining three do not. Among these seven changes, six changes are bug fixes, and the other is a content change in the output message. We show one example of a bug fix in Figure 6. In the example, the bit-wise right shift operator $\gg$ was mistyped as greater than operator $>$ in the buggy version (original version). With the bug fix, the variable red_high had different values in two versions. This variable was later used to compute a PNG file as the program output. The output was already different because of this change, so change effect propagation was not needed in this case.

Three out of the ten change reaching test inputs needed to go through change effect propagation (Algorithm 2 in our approach). By employing Algorithm 2, we succeeded in altering two of these three change-reaching inputs, to produce test inputs which execute the change, *and* propagate its effect to the program output. In other words, we constructed a PNG file which executes the change and manifests its effect by producing different outputs in the two versions of libPNG.

For the last program change, we did not succeed in generating a change-stressing input. The change-effect propagation stopped at a conditional jump in this case. The formula constructed was not satisfiable because of branch correlation. The heuristics proposed by us to handle branch correlation (see Section 4.3) did not allow us to construct a satisfiable formula in this case. This is because, our heuristics handle "immediate" (branch, branch) and (assignment, branch) conflicts. It does not handle "transitive conflicts" where the branch correlation cannot be explained by a *pair* of assignments/branches. For example consider the code fragment `x = 1; y = x; if (y > 2)` Here the direction of evaluation of `y > 2` is fixed by the past two assignments, but there does not exist any pair of statements which can explain the infeasibility of the sequence of statements being executed for any input.

In summary, there are ten changes in our experiment with libPNG, our technique succeeded for nine of them. In the first step of our technique, we successfully get ten change-reaching inputs for all the changes. Eight of these ten inputs are from existing test-suite, the other two inputs are generated by our Algorithm 1. In the second step of our technique, our Algorithm 2 modifies the results from the first step to get inputs that have different output because of the changes. The second step succeeded on nine changes and failed on only one change.

6. RELATED WORK

To test evolving programs, there have been several research efforts under the banner of "regression testing." Even though regression testing in general refers to any testing process intended to detect software regressions (where a program's functionality stops working after some change), often regression testing amounts to re-testing of tests from an existing test-suite. In the past, there have been several research directions that go beyond re-testing all of the tests of an existing test-suite. One stream of work has espoused test selection [2, 12] — selecting a subset of tests from existing test-suite for running on the modified program. Another stream of work proposes test prioritization [4, 15] — ordering tests in an existing test-suite to better meet testing objectives of the changed program.

Recent research projects [13, 16, 19] have proposed test-suite augmentation — developing new tests to stress the effect of the program changes, and these are the works that are closest to our method. We now compare our work with these methods.

The technique in [13] focuses on generating criteria for test-suite augmentation. Their technique starts from the change and guarantees the change effect is propagated up to certain distance. Our technique differs from [13] in several aspects. The work of [13] can either be used to select test cases from a large test pool or can be used as the criteria to drive test generation techniques. In contrast, our technique *generates* test cases, and the test cases generated by our technique are guaranteed to satisfy the criteria of [13]. The technique of [13] performs static symbolic execution (on a program), whereas we perform symbolic execution on a dynamic execution trace. Generally the scalability of static symbolic execution is always an issue, and for this reason we believe [13] restricts the

use of symbolic execution to certain length of dependencies. Our technique is not restricted by the length of dependence chains, and we can handle change effect propagation to any length.

Overall, the work of [13] requires an input which reaches the change, but how such an input is obtained is not considered in their paper. Automatically constructing a change-reaching input is difficult for large programs with huge number of control flow paths. In our work, we construct an input that reaches the change *and* propagates its effect to the output, whereas the method of [13] starts from the change itself (how to reach the change is not studied).

White-box concolic testing techniques from Directed Automated Random Testing or DART (*e.g.*, see [6]) can be used to generate test cases using symbolic execution. The original works on DART focused on test generation to systematically explore program paths. There was no study on generating tests to stress program changes. However, recently some test generation methods [16, 19] for evolving programs have been proposed, which are based on DART. The technique in [16] integrates heuristics to avoid paths that cannot lead to a change and paths that cannot propagate the change-effect. Since they do not consider the distance to the program change, if there are multiple paths leading to the change (as is often the case), their technique would randomly choose one, whereas our technique would prioritize shorter paths. For change effect propagation, the work of [16] only provides heuristics to avoid trying out paths which are unlikely to propagate effects of changes. In contrast, we use dynamic symbolic execution to *guarantee* the advancement in propagation of change effect.

The work in [19] identifies "dangerous edges" affected by the change in control flow graph. Subsequently, heuristics (built on top of symbolic execution using DART) are used to stress these edges. However, after the dangerous edges are identified, the entire analysis is carried out in the changed program. It is important to note that any test generation that analyzes only one program version is unlikely to find the test cases to stress changes. As an example we can consider the two versions of Figure 1 as an example. Consider the test-suite *{x = 0, x = 1, x = 4, x = 100}*. These tests stress the set of all feasible control flow paths in the changed program of Figure 1. Thus, it even achieves path coverage! Whatever dangerous edges are identified, clearly they will be stressed by the tests in this test-suite. However, this test-suite still does not contain the only test input *x = 3* that will stress the program change in this example. In contrast, our technique automatically constructs the input *x = 3*, which behaves differently in these two versions.

Finally, differential symbolic execution (DSE) [9] uses static symbolic execution to characterize the effect of changes. Thus they form a summary of changes across program versions. The summaries are achieved by abstracting large same portions between the original version and the changed version to reduce cost and improve efficiency. Such a technique, while useful for change comprehension, cannot be directly used for test-suite augmentation. If we employ the technique for test-suite augmentation we need to employ symbolic execution of programs for large code-bases (the similar portions across program versions can be abstracted in a summary, but it is not clear how to avoid symbolic execution of this common code-base while generating new tests).

7. DISCUSSION

In this paper, we present a test-suite augmentation method which stresses program changes. To stress a change c in a program, our technique automatically generates a new test case t that gives different outputs in two versions. Our technique works mainly in two steps. In the first step, we use distance in Control Dependency Graph to guide our path exploration towards the change. After a change-reaching input is constructed, our technique use *change effect propagation tree* to identify why a change cannot affect output, and then propagate its effect accordingly.

In the case of correct refactorings, we expect the technique to not generate any new inputs. Since such a change does not result in different program state, our test-suite augmentation method indeed does not generate any change-stressing inputs in such a case.

We have implemented our technique in a toolset based on BitBlaze [14]. To test the efficacy of our technique, we performed two case studies on `tcas` and `libPNG`. For almost all the changes we studied, our tool was able to generate a new test case that stresses the change and causes difference in program outputs. Compared with existing test-suite augmentation techniques, our technique is more goal-directed since we: (i) employ metrics like distance in the control dependency graph to reach the change quickly (this enables us to find a short program path to the change), and (ii) employ heuristics to handle correlated branches while propagating the effect of program change to output (this enables us to avoid searching through many infeasible program paths).

In terms of future work, we can extend our method to handle test programs which stress multiple changes. If the changes are independent (as was the case in our experiments with `libPNG`), the method proposed in this paper can handle multiple changes. However, for inter-dependent program changes (such as when the variable sets affected by two changes have a non-empty intersection), our method needs to be further augmented. This remains an important avenue of future work.

Acknowledgments. This work was partially supported by a Defence Innovative Research Programme (DIRP) grant (R-252-000-393-422) from Defence Research and Technology Office (DRTech).

8. REFERENCES

[1] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, 2009.

[2] Y. Chen, D. Rosenblum, and K. Vo. Testtube: a system for selective regression testing. In *ICSE*, 1994.

[3] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4), 2005.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, 2000.

[5] ERESI. http://www.eresi-project.org/, 2009.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[7] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[8] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[9] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE*, 2008.

[10] QEMU. QEMU emulator. `http://www.qemu.org`, 2009.

[11] D. Richardson and M. Thompson. The relay model of error detection and its application. In *Workshop on Software Testing, Verification, and Analysis*, 1988.

[12] G. Rothermel and M. J. Harrold. A safe efficient regression test selection technique. *TOSEM*, 6(2), 1997.

[13] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE*, 2008.

[14] D. Song et al. BitBlaze: A new approach to computer securityvia binary analysis. In *ICISS, Keynote paper*, 2008.

[15] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in a development environment. In *ISSTA*, 2002.

[16] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Guided path exploration for regression test generation. In *ICSE, New Ideas and Emerging Results*, 2009.

[17] W. van Schaik. Pngsuite. http://www.schaik.com/pngsuite/, December 1998.

[18] J. M. Voas. PIE: a dynamic failure-based technique. *IEEE TSE*, 18(8):717–727, Aug. 1992.

[19] Z. Xu and G. Rothermel. Directed test suite augmentation. In *APSEC*, 2009.