Selecting Peers for Execution Comparison

William N. Sumner, Tao Bao, Xiangyu Zhang
Dept. of Computer Science, Purdue University
West Lafayette, Indiana, USA
{wsumner,tbao,xyzhang}@cs.purdue.edu

ABSTRACT

Execution comparison is becoming more common as a means of debugging faulty programs or simply explaining program behavior. Oftentimes, such as when debugging, the goal is to understand particular aspects of a single execution, and it is not immediately clear against what we should compare this execution. Prior work has led to approaches for acquiring a second execution, or peer, with which to compare the first. The earliest of these involved searching a test suite for suitable candidates. More recently, the focus has been on synthesizing a new execution, either by generating new input for the program or by directly mutating the execution itself. In spite of these proposals, it is not clear what advantages these different techniques for finding peers might have over each other. In this paper, we implement five different existing techniques and examine their impact on 20 real bugs. These bugs represent the full set of reported bugs for three programs during one year. We propose a metric to evaluate the quality of the peers. It is based on the similarity of the peers to the executions of the patched programs. We also discuss in detail the different scenarios where these techniques hold advantages.

1. INTRODUCTION

Execution comparison provides a means for developers to better understand a program's behavior. Given two executions of a program, a developer may compare and contrast the control flow paths taken and the values operated upon during these executions in order to better understand how and why the program behaved differently for each. When one of the executions fails and the other is correct, such comparison forms a basis for automated techniques toward fault localization and debugging [20, 23, 27, 30]. These techniques extract the behaviors and properties of the respective executions that correspond with the correct and incorrect program behaviors. The differences in behavior provide evidence for what parts of a program might be incorrect and why. This helps to reduce the developer burden in finding, understanding, and fixing a bug.

The utility of such techniques depends on precisely which executions they compare against each other [2, 20, 23, 30]. When the correct execution does not follow the same path as the failing one, automated analyses can derive little more than the fact that the executions are different because there are no fine-grained differences from which comparison can derive more information. One might then try using an execution that follows the exact same path as the failing one. Unfortunately, the failing execution's behavior is incorrect, so executions that are similar to it may also behave as if they were incorrect. As a result, the differences between the executions don't provide insight on why the failing execution is buggy. Deciding which executions should be compared is a significant problem when using execution comparison.

The difficulty in the problem arises because of a mismatch between the respective objectives of execution comparison and of debugging in general. Debugging involves comparing the failing execution with the programmer's model or specification of how the execution should have behaved. Differences between the model and the actual behavior of the execution precisely capture an explanation where the program is not behaving correctly, why it is not behaving correctly, and ideally how the developer may change the underlying program to remove the fault. In contrast, execution comparison finds the differences between two actual executions. These differences explain how and why those particular executions behaved differently. For example, if one execution receives an option '`-a`' as input and the other receives an option '`-b`' as input, execution comparison can explain how the semantically different options lead to semantically different program behavior.

The mismatch between debugging and execution comparison lies in the fact that the semantic differences that debugging strives to infer should explain the failure. This inherently requires comparing the failing execution against the correct, intended execution. Unfortunately, only the correct version of the program can generate that correct execution, and the correct program is exactly what automated debugging should help the programmer create. Because this execution is unavailable, we must settle for an execution that is instead similar to the correct one. In order for execution comparison to be useful, the execution against which we compare the failing execution must *approximate* the correct execution without knowing a priori how the correct execution should behave. We call this the *peer selection* problem. Given a particular failing execution, another execution, called the *peer*, must be chosen for comparison against the failing one. This peer must be as similar as possible to the unknown correct execution.

For example, consider the program snippet in Figure 1 a. The `if` statement on line 5 is incorrect; instead of `x>1`, it should instead be `x>2`. As a result, if the input is `2`, the program prints 'two' when it should print 'three'. If we use execution comparison to debug the failing execution when the input is `2`, we must first find a suitable peer. Figure 1 b presents some possible correct executions along with the input that yields each execution and the path, or the trace of

```
1  x = input()
2  if x > 5:
3    print('one')
4  elif x > 0:
5    if x > 1:
6      print('two')
7    else:
8      print('three')
```

(a)

Executions

input	2	6	4	1
path	1	1	1	1
	2	2	2	2
	4	3	4	4
	5		5	5
	6		6	8

(b)

Figure 1: (a) A trivial faulty program. The `x>1` on line 5 should instead be `x>2`. This causes the program to fail when the input is `2`. (b) Example executions of the program on different inputs. Input 2 yields a failing execution; the others are correct.

the statements in the execution.

First consider the execution with input `6`. Of those listed, this execution differs the most from the failing execution. They are different enough that execution comparison doesn't provide much insight. We can see that they take different paths at line 2 because they have different inputs and because `6>5` is true, but `2>5` is not. However, this doesn't imply anything about the correct behavior when the input is `2`. Similarly, if we consider input `4`, the execution follows the exact same path as the failing execution except the execution is correct. This also yields no insights on why the original execution failed. Finally, let us consider the input `1`. This execution follows the exact path that the failure inducing input `2` would induce if the program were correct, and it produces the output that the failing execution should have. Comparing this execution with the failing execution tells us that lines 2 and 4 are likely correct because they evaluate the same in both executions. Line 5, however, evaluates differently, so we should suspect it and examine it when debugging the program. Because this execution behaved most similarly to how the failing execution *should have* behaved, it was able to provide useful insight on why the program was faulty.

This paper considers 5 techniques that can select or create a peer execution when provided with a failing one. The approaches of techniques range widely from selecting test inputs that are already known to synthesizing entirely new executions that might not even be feasible with the faulty version of the program. We objectively examine their fitness for selecting peers by using these techniques to derive peers for known bugs in real world programs. We use the patched versions of the programs to generate the actual correct executions, the ideal peers, and compared the generated peers against them to discover which techniques created peers most similar to the correct executions. Furthermore, we examine the strengths and weaknesses of the techniques with respect to different properties of the bug under consideration, of the failure generated by the bug, and of the faulty program as a whole. The developer can then use this information to either automatically or interactively help choose the most effective peer selection technique for understanding a particular failure. In summary, the contributions of this paper are as follows:

1. We survey and implement the existing techniques that can select peers for execution comparison. We consider the applicability of each technique with respect to properties of the program and failure that the developer knows a priori.

2. We objectively examine the fitness of each technique for peer selection on 20 real bugs in 3 real world programs. They represent the full set of reported bugs for these programs during a roughly one year period. Using the corrected versions of the programs as extracted from their source repositories, we generate the expected correct execution and compare the peers from the techniques against it.

3. We examine the real world bugs to infer when the given techniques may or may not be applicable if more information about a bug is already known.

2. SELECTING EXECUTIONS

In this section, we review five existing techniques for selecting execution peers. Not all of the techniques were originally designed with peer selection in mind, but those that were not still generate peers either as a side effect or intermediate step toward their respective intended purposes.

2.1 Input Isolation

The same input that causes an execution to behave unexpectedly, henceforth called a *failing input*, can sometimes be reduced to produce a valid alternative input. This new and simpler input might yield an execution that does not fail. This insight was used by Zeller in [30] to produce a pair of peer executions, one passing and the other failing, whose internal states were then compared against each other. Given a failing input, Zeller used his previously developed delta-debugging technique [31] to simplify the input and isolate a single input element such that including this element caused a program to crash, and excluding the element caused the program to succeed. The two executions on these inputs, one with the inducing element and one without, were used as the execution peers in his technique. The underlying intuition is that the less the inputs for the two executions differ, the less the two executions should differ, as well.

The central idea of the technique is to use a modified binary search over the possible inputs of the program. A subset of the original input is extracted and tested on the program. If the subset of the original failing input also yields a failing execution, then that new input is used instead of the original, effectively reducing the size of the failing input. In contrast, if the new input yields a passing execution, then this acts as a lower bound on the search. Any later input that is tested must at least include this passing input. In this way, the algorithm terminates when the failing input and the passing input are minimally different[1].

Figure 2 presents a brief example showing how to use delta debugging to generate inputs for peer executions. Suppose that a program crashes whenever the characters '`b`' and '`g`' are present in the input. The full input `a..p` thus causes a program to crash, as seen on the first line. We also assume that an empty input yields a passing execution, as denoted on the last line. The approach then tries to narrow down exactly which portions of the input are responsible for the crash by selectively removing portions of the input and re-executing the program on the new input. In the first test, the algorithm selects `a..h`. When the program executes this

[1]The differences between inputs are locally minimal as noted in [31].

Figure 2: Delta debugging can produce two minimally different inputs, yielding two executions to use as peers.

input, it fails, so the failing input reduces to `a..h` on step 1. On step 2, `a..d` is selected. Because this doesn't include 'g', the execution passes, and the passing input increases to `a..d`. Steps 3 and 4 continue the process until the failing input contains `a..g` and the passing input is `a..f`. Because they differ by only 'g', the inputs are minimally different, and the technique selects the executions on these two inputs as peers.

2.2 Spectrum Techniques

Modern program development includes the creation of test suites that can help to both guide the development process and ensure the correctness of a program and the components from which it is built. One straightforward approach for finding a peer is to simply select an execution from this already existing test suite. This allows the developer to reuse the existing work of generating the tests, but the developer must still select the specific execution from the test suite to use. Execution profiles, or *program spectra*, are characteristic summaries of a program's behavior on a particular input [10] and researchers have long used them as a means of establishing test suite sufficiency and investigating program failures [7, 10, 16, 20, 21]. In particular, spectra have been used to select a passing execution from a test suite with the intent of comparing the execution to a different, faulty one [20]. Thus, spectra provide an existing means of selecting an execution peer.

We focus in particular on the spectrum based peer selection in [20]. This paper introduces the *nearest neighbor model* for selecting a passing peer. For every passing test in the suite, as well as the failing execution itself, we compute a frequency profile that records the number of times each individual basic block of the program executes during the test. Next, we evaluate a distance function with respect to the failing profile against each of the passing profiles. This allows us to determine which profile, and thus which test, is most similar to the failing execution.

One key observation in [20] is that the nature of the distance function is critical to the utility of the overall technique. A Euclidean distance, where each element in the frequency profile is a component in a vector, can lead to illogical results. For example, if one execution has long running loops and another does not, Euclidean distance may ignore the similarity between the parts that are the same in both executions. General inconsistancies with distance functions influenced by the concrete number of times a program executes a basic block led the authors to use the Ulam distance metric, or an edit distance between two sequences of the same unique elements. Each profile is first transformed into an ordered sequence of basic blocks of the program, sorted by their execution frequencies. Note that the *order* is the discriminating factor, the actual frequencies are not even contained in the profile. The distance function between profiles is then the edit distance [13], or the number of operations necessary to transform the sorted list of basic blocks from the test into that from the failing execution. The lower the distance, the fewer dissimilarities there were between the two profiles, so the algorithm selects the test with the lowest edit distance as the *nearest neighbor*, and peer.

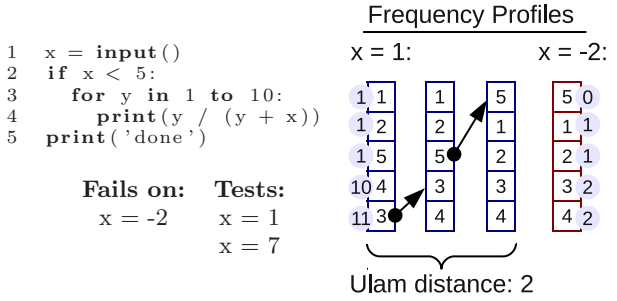


Figure 3: The nearest neighbor model's approach to finding peers. Basic blocks (represented by the numbers inside the boxes) are sorted by frequency (the tagged numbers), then the test profile with the lowest edit distance to that of the failure belongs to the peer.

Consider the example in Figure 3 . For simplicity, it uses frequency profiles over the statements instead of basic blocks. The program listed here has 5 statements which should all execute if the program behaves properly. Line 1 reads a number, `x`, from input. If `x` is less than 5, then enter a loop that computes a new value and prints it. Finally, the program prints 'done' when it is complete. Observe, however, that when `x=-2`, the program crashes on the second execution of statement 4 due to integer division by 0. This gives the failing execution the sorted frequency profile shown on the far right. Basic block IDs are shown in the cells, while their execution frequencies are shown in adjacent circles. Note that the failing execution omits statement 5 and misses the final check of the loop guard. Thus, statements 5 and 3 are out of place. A correct input of `x=1` instead executes the full loop and statement 5, giving the leftmost frequency profile. To compute the Ulam distance between the two, we count the minimum number of blocks that need to be moved. In this case, it is 2. Because the only other test `x=7`, has a distance of 3, the technique chooses input `x=1` as the input of the peer execution in this example.

2.3 Symbolic Execution

Both input isolation and spectra based approaches derive peers from existing input. It is not always possible for this to succeed. For instance, if there are no passing test cases then the spectrum based approach will fail. If there are no *similar* passing cases, it may simply yield poor results. One alternative is to instead generate new inputs for a program with the goal of creating inputs that will inherently yield executions similar to an original failing run. This goal is achievable through recent work in test generation using *symbolic execution* [2, 4, 5, 9, 22, 28].

When a program is symbolically executed, it does not merely execute any operation that might use or depend on

input data. Instead, it builds a formula that reflects what the result of the operation could be for *any* possible input value. When it executes a branching instruction like an if statement, the symbolic execution takes all possible paths, constraining the possible values along those paths appropriately. For example, in Figure 4 , the variable `x` receives an integer value from `symbolic_input()` on line 1. Because the input is symbolic, `x` represents all possible numeric inputs at that point. On line 2, execution encounters an `if` statement, and it must consider both branches. Thus, along the path where the condition is true, the execution adds the constraint `x>5` to the formula for `x`. Along the other path, it adds the constraint `x<=5`.

```
1  x = symbolic_input()
2  if x > 5:
3    print('greater')
4  elif x > 0:
5    if x % 2 == 0:
6      print('even')
7    else:
8      print('even')
```

Figure 4: Buggy code snippet. The last print statement should print "odd".

The sequence of branch constraints along a path, or *path condition*, determine what conditions on the input must hold for the path to be followed. By solving these constraints at a certain point in the symbolic execution, concrete inputs sufficient to reach that program point can be discovered. This is the common underlying approach to test generation using symbolic execution. Precise mechanisms may, however, differ from approach to approach. Note, for instance, that programs may have infinitely long or infinitely many different execution paths, so no approach can cover them all. Thus, the way in which a particular test generation approach chooses to explore program paths can alter. In the context of peer selection, it is desirable to generate executions that are similar to an already existing path. One example of such a strategy is presented in [2].

The test generation approach in [2] creates new tests that are similar to a failing execution in order to aid fault localization in PHP programs. It first uses the failing execution as a 'seed' to the process, collecting its symbolic path condition. It then generates additional paths by negating the conjuncts along the path and solving to generate inputs for those paths as well. The process then continues, always prioritizing the generation of tests along paths that measure as 'similar' to the original failing path condition.

A simplified version of the algorithm is reproduced in Figure 5 . In fact, this is a general test generation algorithm, but as we shall discuss later, we can customize the function $selectNextTest()$ to direct it toward executions similar to the failing one. Line 2 initializes a set of candidate tests to consider adding to the test suite. To start, this contains a test for every branch encountered in the failing execution such that the test takes the alternate path at that branch. E.g., if the failing execution took the true path, the test would take the false path for that branch. Lines 3-6 then iteratively select a candidate test to add to the test suite and generate new candidates until either a time limit expires or there are no more paths to explore. A more complete presentation of the algorithm also handles such optimizations as not following the same path more than once and avoiding

GENERATENEWTESTS(seed)
Input: $failing$ - a failing test
Output: a set of new tests
1: $tests \leftarrow \emptyset$
2: $toExplore \leftarrow getNeighboringTests(failing)$
3: **while** $toExplore \neq \emptyset$ **and** time hasn't run out **do**
4: $test \leftarrow selectNextTest(toExplore)$
5: $tests \leftarrow tests \cup test$
6: $toExplore \leftarrow toExplore \cup getNeighboringTests(test)$
7: **return** $tests$

GETNEIGHBORINGTESTS(seed)
Input: $seed$ - a previously selected test
Output: tests with paths branching from that of seed
1: $neighbors \leftarrow \emptyset$
2: $c_1 \wedge c_2 \wedge \ldots \wedge c_n = getPathCondition(seed)$
3: **for all** $i \in 1, 2, \ldots, n$ **do**
4: $path \leftarrow c_1 \wedge c_2 \wedge \ldots \wedge \neg c_i$
5: $neighbors \leftarrow neighbors \cup solve(path)$
6: **return** $neighbors$

Figure 5: A test generation algorithm directed by the objective function $selectNextTest()$.

tests that are infeasible. We refer the reader to the original papers for the details, which are not relevant to the discussion here.

As previously mentioned, this is a general test generation algorithm. The way in which it selects tests to further explore and add to the test suite is controlled by the objective function $selectNextTest()$. This function guides the test selection toward whatever goals a developer may have in mind. In [2], the goal was similarity with a failing execution, so they additionally developed a heuristic objective function based on path conditions to select such tests. Specifically, given a sequence of branch conditions, or path condition, encountered in a program, their similarity score is the number of the same conditions that evaluated to the same values in both executions.

For example, consider again the program in Figure 4 . The program has a bug such that when line 1 assigns the value 1 to `x`, the program executes the faulty print statement on line 8. The path condition for this execution is $\langle 2_F \wedge 4_T \wedge 5_F \rangle$, meaning that the condition on line 1 evaluated to false, the condition on line 4 evaluated to true, and the last condition on line 5 evaluated to false. If the candidate tests in *toExplore* are the inputs -1, 2, and 10, then their respective path conditions are $\langle 2_F \wedge 4_F \rangle$, $\langle 2_F \wedge 4_T \wedge 5_T \rangle$, and $\langle 2_T \rangle$. Observe, input -1 yields the same value for condition 1, so its similarity is 1. Input 2 has similarity 2, and 3 has similarity 0 for the same reason. Because input 2 has the highest similarity, it is the next test added to the test suite and further explored.

We decide if the generated inputs lead to passing executions with an oracle. We further select from the pool the one that is most similar to the failing run using the same path condition based criterion.

2.4 Predicate Switching

Instead of generating input that will force a buggy program to yield a strictly correct, passing execution, some approaches for selecting peers relax the notion of correctness.

This allows peers that may represent slightly incorrect executions or normally infeasible executions. Instead of finding some new input for the program, these techniques can patch the program or even the execution itself at runtime to create a new execution that behaves similarly to a failing one. Because incorrectness is undesirable in a peer, these approaches must be careful in how they modify a program's behavior.

One such dynamic patching technique is *predicate switching* [32]. While originally designed for fault localization, it generates peers as a part of its core algorithm. The authors of this paper noted through experiments that less than 10% of an execution's instructions relate to actual computation of values produced as output. As a result, they inferred that they could correct most faults fixing only the control flow of an execution. That is, by forcing an execution to follow the desired control flow, it would behave, to an observer, exactly the same as a correct program. This insight was then extended to a technique for constructing peers when given a failing execution.

One simple way to alter the control flow of a program is to switch the result of evaluating a branch condition, or predicate. For example, if the tenth predicate of a program originally evaluated to true, predicate switching would dynamically patch the execution and flip the outcome to false, forcing the execution along the false path instead of the true path. Predicate switching as a technique combines this operation with various search strategies for such a dynamic patch, considering each individual predicate instance as a candidate. If the observable behavior of a patched execution matches the expected behavior of the failing execution, then the search found a patch, and this patched execution can be immediately returned as a peer.

```
1  x = [101, 102, 103]
2  for i in 0 to 3:
3    if x[i] % 2 == 0:
4      print(x[i])
5  print('done')
```

Figure 6: Buggy code snippet. Index 3 is invalid when accessing the array x.

The code snippet in Figure 6 presents a small example where predicate switching is of use. This program creates a list with 3 elements, but its loop body executes for indices 0 through 3 inclusive. When the execution accesses index 3 of the list on line 3, it throws an exception, terminating the program. The sequence of predicates encountered within the failing execution is $\langle 2^1_T, 3^2_F, 2^3_T, 3^4_T, 2^5_T, 3^6_F, 2^7_T \rangle$, with superscripts the timestamps, distinguishing the different instances of the same static predicate. The program crashes after it executes 2^7_T. Predicate switching attempts to switch each predicate in order starting with 2^1 until it finds a patch. Note that the second iteration of the loop, when $i = 1$ should print out '102', so the algorithm won't find a valid patch until after that iteration. When the technique switches the fifth predicate, 2^5_T, to false, the program prints 'done' and successfully ends. This is the expected behavior of the correct program, so the search ends with the dynamic patch of 2^5 providing the peer.

Note that the result does not perfectly match the control flow of the correct program. Namely, the correct program also expects a third iteration of the loop to complete, even though it doesn't do anything. Because the last iteration was not necessary for generating the expected observable behavior of the execution, predicate switching considers that loss an acceptable approximation for the peer.

2.5 Value Replacement

Value replacement [14, 15] is a technique related to predicate switching in that it also modifies the behavior of a program dynamically in order to create an approximately correct execution. Whereas predicate switching works over the boolean domain of branch conditions, value replacement instead replaces other values within a program as well, allowing it to dynamically patch value based errors in addition to control flow errors.

The underlying abstraction used by value replacement is the *value mapping* for a statement, or the set of values used by a particular dynamic instance of a statement during an execution. By changing the value mapping for a statement, value replacement can cause a failing execution to behave observably correctly. Indeed, predicate switching is a special case of value replacement, wherein it only considers the values at branching statements. The difficulty of the technique lies in finding appropriate value mappings to use at a given statement. Unlike predicate switching, which only needs to consider the values true and false, replacing all values at a statement gives intractably many different options for what the value mapping in a patch might be. It is not generally knowable *a priori* which of these options might correctly patch the execution. To make a solution feasible, value replacement only considers the other values observed in the test suite of a program, as these are at least known to be valid alternatives. The first step of the technique constructs a *value profile* that holds the value mappings for a statement observed in any of the tests of a test suite. For each dynamic statement in the failing execution, the previously observed value mappings are then used to generate dynamic patches.

The value profile for a statement may still have many different mappings. Instead of directly applying each of the value mappings, the technique only looks at a subset of the value mappings. For a statement, these mappings use the values from the profile that are: the minimum less than the original value, the maximum less than the original value, the minimum greater than the original value, and the maximum greater than the original value.

For example, consider the statement `z = x + y` where `x` and `y` have the values 1 and 3 in the failing execution, respectively. If the value profile contains `x = 0,1,2,3` and `y = 0,1,2,3`, then the potential value mappings of the statement will consist of {`x=0,y=0`}, {`x=0,y=2`}, {`x=2,y=0`}, {`x=2,y=2`}, {`x=3,y=0`}, and {`x=3,y=2`}.

3. ANALYTICAL COMPARISON

In this section, we classify the techniques and discuss their applicability. We empirically compare the techniques in Section 4 . Table 1 summarizes these techniques. Column `Class` classifies the techniques based on how they select peers. Column `Oracle` lists the power of the testing oracle that must be available for applying the technique. Column `Test Suite` notes whether or not a given technique requires a test suite.

Classification. The first three techniques either generate new inputs or use existing inputs to select peer executions. We call them input based techniques. These technique can only generate executions that exercise feasible paths in the

Technique	Class	Oracle	Test Suite
Input Isolation	input synthesis	complete	no
Spectra Based	input selection	test suite	yes
Symbolic Execution	input synthesis	complete	no
Predicate Switching	execution synthesis	1 test	no
Value Replacement	execution synthesis	1 test	yes

Table 1: Common features among peer selection approaches.

faulty program (recall that a feasible path is an execution path driven by a valid input).

The remaining two techniques synthesize executions from the failing execution. They may generate executions that may not be feasible under the original faulty program.

Oracle. Both input isolation and symbolic execution require complete oracles in theory. These techniques check whether or not arbitrary executions pass or fail, so the oracle must theoretically work on any execution. In fact, input isolation also needs to determine whether or not an execution fails in the same way as the original, e.g. crashing at the same point from the same bug. In practice, implementations of techniques use approximate oracles instead. For example, simple oracles can be composed for certain types of failures (e.g. segfaults). Spectra based selection examines the passing cases from an existing test suite, so an oracle must be able to determine whether or not each individual test passes or fails. Such an oracle is weaker than a perfect oracle, because it is usually a part of the test suite. The execution synthesis techniques only check that the patched executions are able to observably mimic the expected behavior. This means that an oracle must be able to check for the expected behavior of a single execution. This is the weakest oracle.

Test Suite. Both the spectra based and value replacement approaches require a test suite to function. As noted earlier, test suites are commonly used during development, but the efficacy of these particular approaches will depend on how the tests in the test suite relate to the passing and failing executions. For example, if the test executions do not follow control flows similar to those in the failing execution, the spectra based approach will yield a poor peer. On the other hand, if statements within tests never use the values that the failing execution needs to patch and correct its own behavior, then value replacement will fail entirely.

4. EXPERIMENT

We implemented all the techniques using the LLVM 2.8 infrastructure and an additional set of custom libraries, requiring about 11,000 lines of C and C++ and 1500 lines of python in addition to using the KLEE infrastructure for symbolic execution [4]. We ran all tests on a 64-bit machine with 6 GB RAM running Ubuntu 10.10.

For input isolation, we performed delta debugging both on the command line arguments passed to the faulty program and on the files that the program uses during the buggy execution. For our spectrum based selector, we computed the frequency profiles over the test suite for each program that existed at the time the bug was not yet fixed. Each of our analyzed programs contains a test directory in its repository that contains regression tests for bugs that developers previously fixed. These suites range in size from 150 to 1000 tests.

We built our symbolic execution based selector using a modified version of the Klee engine. Instead of generating tests with a breadth first branch coverage policy, we modified the search heuristics to initially follow the path of the failing execution as much as possible, then explore paths that branch off from the original execution. If the constraints along one path are too complex, the system gives up on solving input for that path and continues test generation on other paths. Once the system finishes (or the time limit expires), it selects the passing test with the a path condition most similar to that of the failing run as the peer.

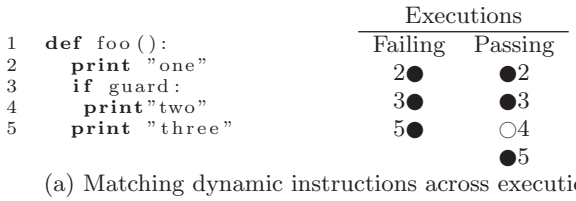
We performed an empirical study of how these techniques behave for executions of a set of real bugs. The study examines (1) the overhead incurred by using each technique in terms of time required, (2) how accurately the peers generated by each technique approximate the correct execution, and (3) how the approximations from different techniques compare to each other and in which cases one may be more appropriate than another.

Evaluating the second objective is challenging. Given the selected peer, an execution of the existing buggy program, we must objectively measure how similar it is to the intended correct execution. The more similar, the better the peer approximates the behavior of the correct execution. This necessitates both knowing what the correct execution should be and having a way to compare it with the chosen peer. In order to know what the correct execution should be, we limit our study to bugs that maintainers have already patched in each of the above programs. To reduce the possible bias caused by bug selection, we extracted the bugs for each program by searching through a one-year period of the source repository logs for each program and extracting the correct versions where the logs mentioned corrections and fixes for bugs. We also extracted the immediately preceding version of the program to capture the faulty program in each case. Excluding those that were not reproducible or objectively detectable, we have 10 versions for `tar`, 5 for `make`, and 5 for `grep`. By running the failing version of the program on the input that expresses each bug, we generate the execution for which we desire a peer. By running the corrected version, we generate the perfect peer against which we must compare the selected peers. The precise mechanism for comparing the executions is discussed in Section 4.1 .

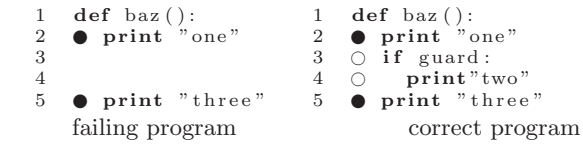
4.1 Measuring Execution Similarity

We need a way of objectively comparing the similarity of these executions. We do this by measuring along two different axes. First, we measure the *path similarity* of the two executions through execution indexing [29]. Indexing builds the hierchical structure of an execution at the level of each executed instruction. Executions can be very precisely compared by comparing their structures. It has been used in a number of applications that demand comparing executions [17, 24]. For example, in Figure 7 a, the instructions 2, 3, and 5 correspond across the two executions as denoted by the filled circles, but instruction 4 in the correct execution does not correspond to anything in the failing execution, as it only appears in one of the executions.

Note that execution indexing first requires that we know



(a) Matching dynamic instructions across executions



(b) Matching static instructions across versions

Figure 7: Matching of instructions across executions and program versions. Matches are denoted by ●, while mismatches are denoted by ○.

which static instructions correspond between the two executions. Because the executions come from two different versions of a program, some functions may have changed, with instructions being either added or removed between the two versions. In order to determine which instructions correspond, we extracted the static control dependence regions into a tree and computed a recursive tree edit distance [3] to find the largest possible correspondence between the two programs. For example, Figure 7 b shows a program where a developer inserted an `if` statement guarding a new instruction. An edit distance algorithm will uncover that the fewest possible changes between the two functions require that lines 2 and 5 correspond across the program versions, and the `if` statement on line 3 along with the nested control dependence region on line 4 are new additions to the correct version of the program.

We score path similarity using the formula

$$100 \times \frac{2I_{both}}{I_f + I_c} \quad (1)$$

where I_{both} is the number of dynamic instructions that match across the executions, I_f is the total number of instructions in the failing execution, and I_c is the number of instructions in the correct execution. This scores the path similarity of two executions from 0 to 100, where 0 means that the paths are entirely different, and 100 means that the paths execute exactly the same instructions.

Second, we measure the *data similarity* of the executions. We compare the memory read and write instructions that correspond across the two executions, examining both the target location of the memory access and the value that it reads or writes. If these are equivalent across the executions, then the accesses match, otherwise they represent a data difference between the programs. The similarity of the accesses is then scored, again using formula 1, but only considering the instructions that access memory. One difficulty performing this comparison, as noted by others [19], is that comparing pointers across two executions is difficult because equivalent allocation instructions may allocate memory at different positions in the heap. This makes it difficult to determine both (1) when the targets of memory accesses match, and (2) when values of read or written pointers match. To enable this, we use *memory indexing* [24], a technique for identifying corresponding locations across different executions, to provide canonical IDs for memory locations. We also use shadow memory and compiler level instrumentation to mark which words in memory hold pointer values. Thus, when comparing the targets of accesses, we compare their memory indices. When comparing the values of accesses, we use either memory index of that value if the value is a pointer; otherwise we use the actual value in memory.

These two approaches allow us to objectively determine both how similar the paths taken by the peers and correct executions are as well as how similar the values they used and created were along those matching paths.

4.2 Results

Bug ID	Program	Patch Date	Fault Type	Failure Type
1	tar	23 Jun 2009	missing guard	behavior
2	tar	30 Jul 2009	missing function call	value
3	tar	30 Jul 2009	weak guard	behavior
4	tar	5 Aug 2009	missing function call	behavior
5	tar	4 Oct 2009	wrong formula	value
6	tar	7 Oct 2009	design error	behavior
7	tar	17 Mar 2010	design error	behavior
8	tar	27 Mar 2010	incorrect guard	loop
9	tar	28 Jun 2010	call at wrong place	behavior
10	tar	23 Aug 2010	incorrect guards	behavior
11	make	3 Jul 2010	design error	behavior
12	make	6 Oct 2009	design error	behavior
13	make	30 Sep 2009	design error	behavior
14	make	23 Sep 2009	design error	behavior
15	make	1 Aug 2009	wrong function called	value
16	grep	14 Dec 2009	design error	behavior
17	grep	4 Mar 2010	corner case	behavior
18	grep	4 Mar 2010	corner case	behavior
19	grep	14 Mar 2010	corner case	behavior
20	grep	25 Mar 2010	incorrect goto	value

Table 2: Bugs used in the study along with their failures and the faults that cause them.

Using these representatives of the peer selection techniques, we considered the collected bugs. Table 2 summarizes some details for the bugs. Column `BugID` uniquely identifies each bug across our results. `Program` lists the program to which each bug belongs, along with the date that a developer committed and documented a fix for the bug in the source repository in column `Patch Date`. The table classifies the fault in the program that induces a failure in `Fault Type`. For instance, in bug 15 on `make`, the program calls `strcpy()` instead of `memmove()`, resulting in a corrupted value, so the fault has the description 'wrong function called'. Finally, column `Failure Type` classifies what the user of a program is able to directly observe about a failure. These observations primarily fall into two categories: the program either computed an unexpected value (denoted value), or it performed an unexpected action (behavior). For example, the incorrect value in bug 5 is an integer representing Unix permissions that the program computes incorrectly, while the incorrect behavior of bug 14 is that `make` doesn't print out errors that it should. Bug 8 caused `tar` to execute an infinite loop, which is a specific type of incorrect behavior.

4.2.1 Overhead

First, let us consider the overhead of the different techniques, as presented in Table 3. For each examined bug in column `BugID`, we ran each of the five peer selection tech-

Bug ID	Input Isolation		Spectrum		Symbolic Execution	Predicate Switching		Value Replacement		
	Tests	Time	Profile Time	Selection Time	Time	Tests	Time	Tests	Profile Time	Selection Time
1	51	0.7	144	1233	N/A	22412	178	637381	148	5883
2	44	0.2	144	1236	N/A	-	65	-	143	2934
3	213	23.2	145	1257	N/A	25023	204	-	143	>4 hours
4	9	0.2	147	1299	N/A	26880	302	-	143	7032
5	14	10.1	226	1516	N/A	-	74	-	212	3279
6	40	0.6	228	1517	N/A	-	239	-	210	8592
7	208	3.1	262	1890	N/A	27252	228	836191	240	8054
8	27	50.4	251	1877	N/A	24132	12122	-	251	>4 hours
9	55	50.2	255	1866	N/A	31878	143	-	247	2691
10	6	0.1	315	2695	N/A	-	230	-	306	3381
11	5	0.1	35	841	N/A	12581	33	-	1112	>4 hours
12	6	0.1	36	811	N/A	21228	66	819332	1021	13753
13	6	0.1	36	758	N/A	16504	48	674999	1092	11167
14	7	0.1	38	757	N/A	-	230	-	1143	>4 hours
15	9	0.2	35	761	N/A	15599	50	-	648	>4 hours
16	20	0.1	1.1	248	N/A	-	7.1	-	18.7	65
17	5	0.1	4.5	188	N/A	26	0.1	4232	35.2	16.4
18	7	0.1	4.5	197	N/A	252	3.4	8446	25	28.1
19	5	0.1	2.6	218	N/A	196	3.4	232	31.8	6
20	13	40	3.4	627	N/A	39	2.1	59	41.8	1.2
Average	38	9	116	1090	N/A	16000	711*	432202	361	3386
StdDev	61	17	108	676	N/A	11545	2688*	675610	401	4244

Table 3: Peer selection work for each bug. For each technique, `Tests` holds the number of reexecutions of the program required by a technique before finding a peer. `Time` is the total time required to find a peer in seconds. *Average and StdDev for Predicate Switching without the infinite loop outlier are 119 and 101.

niques and noted both the `Time` in seconds required to select a peer and the number of `Tests` or reexeutions of the program that the technique used. Note that symbolic execution based approaches don't necessarily terminate on their own, as they explore as many of the infinite program paths as possible within a given time bound. As a result, this metric doesn't apply to symbolic execution; we always stopped it after 4 hours. Similarly, we stopped any other approaches after 4 hours and considered that attempt to find a peer unsuccessful. Furthermore, two of the approaches, the spectrum based approach and value replacement both have the profiling cost.

The input isolation approach is consistently the fastest, always finishing in under a minute, which is highly desirable. Next, we note that the average time for predicate switching includes an outlier. Bug 8 is an infinite loop in `tar`, and we use a five second timeout to detect such failures. As a result, every test costs one to two orders of magnitude more in that test case. Discarding that outlier, predicate switching is the next fastest with an average of 2 minutes to find a peer or determine that it cannot find one. The next fastest is the spectrum based approach, using over 18 minutes to find a peer on average. Finally, value replacement is the slowest, taking about an hour on average to find a peer.

These results are more intuitive when we consider what bounds each approach. Input isolation uses delta debugging over the execution input, so it has a running time polynomial in the size of the input in the worst case and logarithmic in normal cases. The spectrum based approache computes an edit distance over metadata for every statement in a program, so it is polynomially bounded by the program size. In contrast, predicate switching and value replacement are bounded by the length of the execution in either branch instructions or all instructions respectively, so they can take longer in theory. Looking at the `Tests` columns, both approaches execute the buggy program several thousand times when finding a peer, whereas input isolation executes it only 38 times on average.

4.2.2 Fitness

While the speed of finding a peer is important for providing convenience to the developer and scalability to client analyses, we saw in Figure 1 , that a peer is not useful for debugging unless it forms a good approximation of the correct execution. To that end, we collected the peers generated by each of the techniques. We then traced the control flow and data accesses of the executions and compared these traces using the similarity metric outlined in Section 4.1 . Table 4 summarizes the results. For each technique, we present both the path similarity (`path`) and the data similarity (`data`), along with the average and standard deviation for each technique. We further discuss our observations, which will be highlighted and tagged with symbol O_i.

Predicate Switching.

We first note that (O_1)**: predicate switching performs best in terms of both path and data similarity**, on average scoring in the 60's while having the lowest variance in both similarity metrics.

Since predicate switching often generates infeasible paths under the original faulty program, we observe in many cases, (O_2)**: correct executions resemble infeasible paths under the original faulty program**. Consequently, input based techniques are less effective because they only generate feasible paths for the faulty program. For example, consider bug 8 of `tar`, presented in Figure 8 . This function tries to open a file at a given path location. If the call to `open` fails, the execution enters the loop to attempt fixing

Bug ID	Input Isolation		Spectrum		Symbolic Execution		Predicate Switching		Value Replacement	
	Path	Data	Path	Data	Path	Data	Path	Data	Path	Data
1	0.2	67.8	98.6	59.3	0.2	69	99.6	67.5	99.7	65.4
2	99.0	56.1	84.0	56.2	0.2	67	-	-	-	-
3	0.2	69.7	87.6	59.4	0.2	68	87.6	56.3	-	-
4	97.4	69.6	95.9	59.0	0.2	69	97.7	66.7	-	-
5	99.3	60.0	98.7	59.7	0.5	69	-	-	-	-
6	0.1	9.1	0.1	9.1	0.1	9.1	-	-	-	-
7	0.1	37.5	0.1	37.5	0.1	9.1	0.1	37.5	0.1	37.5
8	0.2	66.8	95.9	54.4	0.2	67	99.5	67.2	-	-
9	0.2	67.2	75.4	58.9	0.2	67	74.8	58.3	-	-
10	0.1	21.9	0.1	21.9	0.1	22	-	-	-	-
11	74.7	56.2	66.2	52.4	93	50	97.8	61.9	-	-
12	67.1	52.4	0.1	13.9	15	55	66.9	59.2	66.8	59.2
13	78.4	63.3	43.2	63.3	75	65	38.6	61.2	78	61
14	0.1	13.8	0.1	13.9	0.1	14	-	-	-	-
15	52.3	61.4	66.2	52.4	82	76	97.8	61.7	-	-
16	0.1	0	0.1	0	0.1	0	-	-	-	-
17	13.6	46.5	0.7	70	0.6	66	1.1	81	74	63
18	14.4	49.6	93	67	0.6	66	55.2	63	71	63
19	75.8	64.5	93	64	0.6	67	97	64	3.1	69
20	30.4	60.9	0.2	70	0.2	67	2.6	59	31	82
Average	33	49	50	47	13	83	65	62	49	63
StdDev	44	21	42	20	30	143	33	8.7	51	15

Table 4: Peer selection similarity scores for each bug. For each selection technique, `Path` denotes the path similarity score for each bug. `Data` denotes the data similarity score for each bug.

```
1  int create_placeholder_file(char *file, int *made) {
2    ...
3    while ((fd = open (file)) < 0)
4      if (! maybe_recoverable (file, made))
5        break;
6    ...
7  }
```

`tar`, extract.c: create_placeholder_file

Figure 8: Bug 8 requires unsound techniques to get a strong peer.

```
1  void mb_icase_keys (char **keys, size_t *len) {
2    ...
3  incorrectpatch:
4    for (i = j = 0; i < li ;) {
5      /* Convert each keys[i] to multibyte lowercase */
6    }
7    ...
8  }
```

`grep`, grep.c: mb_icase_keys

Figure 9: Execution synthesis on bug 17 yields an erroneous patch.

the error with a call to `maybe_recoverable`. If this call returns the constant `RECOVER_OK`, then the error was fixed and the function calls `open` again. The failing execution enters an infinite loop because the condition on line 4 should be compared against the constant `RECOVER_OK` instead of being negated. In this buggy form, `maybe_recoverable` returns a nonzero errorcode every time, so the loop never terminates. Techniques that rely on sound behavior cannot modify the execution to escape the loop, so they must select executions that never enter it to begin with. In contrast, because predicate switching can select peers from among unsound or otherwise infeasible executions, it is able to iterate through the loop the correct number of times and then break out of it. Thus, it matches the correct path very closely with a score of 99.5.

(O_3) **Predicate switching may fail to patch a failing run even though it often leads to peers with good quality if it manages to find one**. In comparison, since input based techniques are not constrained to patching the failing run. They will eventually select a closest peer (even with bad quality).

(O_4) **Predicate switching may coincidently patch a failing run, returning a peer with poor quality.** For example, in bug 17 for `grep`, predicate switching yields a peer with a path similarity of only 1.1, even though it produced the correct output. This is because the path of the peer execution actually deviates from the correct path early in the execution but the deviated path nonetheless produces the expected output. Figure 9 presents the location of the incorrectly patched predicate. When `grep` starts up, it converts user requested search patterns into a multibyte format when ignoring character case, but predicate switching toggles the condition on line 4, preventing the pattern from being converted. That `grep` still produced the correct output is merely coincidental. The unconverted search patterns happen to lead to a successful match when the converted ones did not, even though the buggy portion of the program is in an unrelated component of grep (see Figure 10). As a consequence, the comparison of the failing run with the generated peer does not isolate the faulty state but rather some random state differences. Fortunately, this is an unlikely enough coincidence that predicate switching still scores best on average.

We also note that (O_5) **predicate switching is less likely to provide a good peer when being used on bugs that exhibit value based failures**. In particular, for three of the bugs that exhibit value based failures, where the fault leads to the incorrect value through computation instead of control flow, predicate switching either yields a poor peer (bug 20) or no peer at all (bugs 2 and 5). Developers can use this insight to use a more general approach like the spectrum techniques when dealing with value failures.

```
1  void parse_bracket_exp_mb (void) {
2  bug19:
3    wt = get_character_class(str, case_fold);
4    ...
5  bug17:
6    if (case_fold)
7      remap_pattern_range(pattern);
8    ...
9  bug18:
10   if (case_fold)
11     canonicalize_case(buffer);
12 }
```

grep, dfa.c: parse_bracket_exp_mb

Figure 10: Bugs 17, 18, and 19 are related.

Value Replacement.

Value replacement shares a lot of common characteristics with predicate switching. Hence, (O_6) **value replacement scores high when it finds a peer, but it is less likely to find a peer**. This shortcoming is twofold: (1) it takes so much longer that we had to cut its search short, and (2), it can only mutate the values of an execution as guided by the values observed in the test suite. Thus, if the desired value is never seen in the test suite, value replacement cannot generate a peer. Sometimes, this leads to peers with very poor quality (e.g. bug 19) or even no peers (e.g. bugs 9 and 10).

Spectrum Techniques.

The next strongest technique is the spectrum based approach, which matched paths as well as value replacement, but was not able to match data accesses as well. (O_7) **The spectrum based approach works better with a larger test suite, which on the other hand implies the longer time spent on searching for the peer.** This is evidenced by its strongest scores for `tar`, which had the largest test suite of the programs we tested and the high computation cost. In other cases, as in `make` and `tar`, finding a desirable peer may be difficult or impossible because no similar test exists.

Of particular interest, however, are the similarity scores in the 90's for bugs 18 and 19 of `grep`. Both of these bugs are related to the component that was patched to fix bug 17. The peer selected for both of those bugs was actually the test added to make sure bug 17 did not reappear as a regression. As shown in Figure 10 , the three bugs are within the same function, dealing with the same set of variables. As a result, the regression test closely resembles the correct executions regarding bugs 18 and 19, making itself a good peer. Thus, (O_8) **the spectrum based approach is able to adapt**. Once a test for a buggy component is added to the test suite related bugs may have a stronger peer for comparison. This is particularly useful in light of the fact that bugs tend to appear clustered in related portions of code [18].

Input Isolation.

Input isolation is by far the fastest, but there does not appear to be an indicator for when or if it will return a useful peer. Its tendency to yield paths of low similarity to the correct executions makes it generally less desirable for debugging purposes. This happens because (O_9) **input similarity and behavioral similarity don't necessarily correspond, so making small changes to input will likely create large changes in the behavior of a program**. For example, consider bug 8 for `tar`. When extracting an archive with a symbolic link, using the `-k` option causes `tar` to enter an infinite loop. This option, however, significantly alters the behavior of the program, preventing it from extracting some files. Input isolation selects a peer by generating new input where the `k` has just simply been removed. This avoids the error, but it also radically changes the behavior of the program such that the peer is highly dissimilar to the correct execution. This effect is undesirable because the differences between the failing run and the peer passing run may not be relevant to the fault, but rather just semantic difference resulted from the input difference.

In spite of this, its ability to exploit *coincidental similarity* between passing and failing inputs allows input isolation to create better peers for some bugs than any of the other approaches. For example, consider bug 5 of `tar`. Input isolation produced the peer with the highest path and value similarity for this bug. The original command line to produce the failure included the arguments `-H oldgnu`, telling the program to create an archive in the 'oldgnu' format. Input Isolation determined that the most similar passing run should instead use the arguments `-H gnu`. Coincidentally, 'gnu' is another valid file format, and creating an archive in that format follows almost the same path as the oldgnu format. Such behavior is difficult to predict, but interesting nonetheless.

Symbolic Execution.

We observe (O_{10}) **the symbolic execution approach performs poorly, generating only very short executions that divert substantially from the failing runs.** The main reason is that the approach tries to model path conditions of the entire failing runs in order to search for their neighbors. However, the original failing runs are too complex for Klee to handle (averaging 300k bitvector constraints). According to the algorithm 5, the technique degenerates to producing inputs for short executions, which it can manage. Note that good performance was reported for the same algorithm for web applications in [2]. But web applications are usually smaller and have very short executions. We note that good code coverage can be achieved for our subject programs if simply using Klee as a test generation engine. But we observe the generated executions are nonetheless very short, attributing to the path exploration stragety of the test generation algorithm. In contrast, peer selection requires modeling relatively much longer executions.

5. RELATED WORK

We have examined methods for selecting or synthesizing executions for use in execution comparison. Many of the techniques that are related to peer selection stem from existing areas of software engineering.

Execution comparison itself has a long history in software engineering, from earlier approaches that enabled manually debugging executions in parallel to more automated techniques that locate or explain faults in programs [6, 27, 30]. These systems require that a failing execution is compared with an execution of the same program. This paper augments these approaches by suggesting that the desired execution should approximate the correct one. This approximation aids the existing techniques in explaining failures.

Many of the peer selection techniques come from automated debugging analyses. Delta debugging, used for in-

put isolation, is well known for reducing the sizes of failing test cases and a vast number of other uses [31]. Zeller's work on execution comparison also utilized delta debugging to locate faulty program state [6], using either input isolation or spectrum based selection techniques to select a peer execution. Spectrum based approaches for fault localization [7,10,16,20,21] have long exploited existing test suites in order to identify program statements that are likely buggy. Symbolic execution and constraint solving are popular approaches for automatically generating test suites [4,9,22,28]. By building formulae representing the input along paths in an execution, they are able to solve for the input required to take alternative paths. Some systems also use search heuristics to guide the tests toward a particular path for observation [2, 28], appropriately generating a peer as we did in our experiments. Execution synthesis techniques like predicate switching and value replacement mutate an existing failing execution in unsound ways to see if small changes can make the executions behave correctly [15,32]. The information about how and where this dynamic patch occurred can be used for fault localization. Some effort has been put into finding corrections to executions that require multiple predicate switches [26]. More recently, the observation that unsound approximations of executions provide useful information about real program behaviors has even been explored for test generation [25].

Our system for comparing execution traces uses execution and memory indexing to identify corresponding instructions and variables across the different executions [24, 29]. Other trace comparisons exist, but they do not allow strictly aligning both data and control flow as necessary for examining suitability for execution comparison [11, 19, 33]. Existing work also looks at the static alignment and differencing of static program, but these approaches emphasize efficiency over precision or high level alignment over low level alignment in comparison to the edit distance approach used in this paper [1,12]. Edit distances have been used to align the abstract syntax trees of multiple programs [8], but not the control flow, which is necessary for aligning traces.

6. CONCLUSIONS

This paper introduces peer selection as a subproblem of using execution comparison for debugging. Given a failing execution, that failing execution must be compared against a peer that approximates the correct, intended execution in order to yield useful results. We have surveyed and implemented the existing five peer selection techniques, comparing how well the peers that they generate approximate the correct execution and how quickly they can find theses peers. The results provide insights into the advantages and disadvantages of the different techniques.

7. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Autom. Softw. Eng.*, 14(1):3–36, 2007.

[2] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, 2010.

[3] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337, 2005.

[4] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *TISSEC*, 12(2), 2008.

[6] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[7] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *FSE*, 2001.

[8] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33:725–743, 2007.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, 2005.

[10] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *PASTE*, 1998.

[11] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI*, 2009.

[12] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI*, 1990.

[13] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[14] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA*, 2008.

[15] D. Jeffrey, N. Gupta, and R. Gupta. Effective and efficient localization of multiple faults using value replacement. In *ICSM*, 2009.

[16] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[17] P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.

[18] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *FSE*, 2004.

[19] M. K. Ramanathan, A. Grama, and S. Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *ASE*, 2006.

[20] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[21] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *FSE*, 1997.

[22] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *FSE*, 2005.

[23] W. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In *FASE*, 2009.

[24] W. Sumner and X. Zhang. Memory indexing: canonicalizing addresses across executions. In *FSE*, 2010.

[25] P. Tsankov, W. Jin, A. Orso, and S. Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *ICST*, 2011.

[26] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE*, 2005.

[27] D. Weeratunge, X. Zhang, W. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *ISSTA*, 2010.

[28] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, 2009.

[29] B. Xin, W. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI*, 2008.

[30] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.

[31] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *TSE*, 28(2):183–200, 2002.

[32] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, 2006.

[33] X. Zhang and R. Gupta. Matching execution histories of program versions. In *FSE*, 2005.