Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests

Chen Huo
Computer & Information Sciences Department
University of Delaware, DE, USA
huoc@udel.edu

James Clause
Computer & Information Sciences Department
University of Delaware, DE, USA
clause@udel.edu

ABSTRACT

Writing oracles is challenging. As a result, developers often create oracles that check too little, resulting in tests that are unable to detect failures, or check too much, resulting in tests that are brittle and difficult to maintain. In this paper we present a new technique for automatically analyzing test oracles. The technique is based on dynamic tainting and detects both brittle assertions—assertions that depend on values that are derived from uncontrolled inputs—and unused inputs—inputs provided by the test that are not checked by an assertion. We also presented OraclePolish, an implementation of the technique that can analyze tests that are written in Java and use the JUnit testing framework. Using OraclePolish, we conducted an empirical evaluation of more than 4000 real test cases. The results of the evaluation show that OraclePolish is effective; it detected 164 tests that contain brittle assertions and 1618 tests that have unused inputs. In addition, the results also demonstrate that the costs associated with using the technique are reasonable.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Measurement

Keywords

Unit testing, Improving oracles, Brittleness, Unused inputs, Dynamic tainting, Mutation

1. INTRODUCTION

Although software tests are conceptually simple—they are composed of two parts: *inputs* that are used to execute the program under test and an *oracle* that is used to verify that the execution induced by the inputs produces the expected results—they are often difficult to write in practice. This is especially true for modern software which is typically large and complex. Together, these characteristics produce a situation where test writers have an imperfect understanding of not only what inputs a program may receive but also how the program should behave and what outputs it should produce. In short, when writing tests, selecting neither inputs nor oracles is straightforward.

Fortunately, the software engineering research community has provided many techniques that can help developers write test cases (e.g., [6, 7, 9, 15, 17–19, 22, 26, 28]). In general, these techniques focus primarily on test data selection or generation (i.e., choosing inputs). While such techniques can be successful in helping developers write tests, they only address part of the overall problem. In order to provide more effective help, it is necessary to not only provide developers with help choosing inputs but also with help creating oracles.

In many testing frameworks, an oracle is encoded as a set of assertions that check whether a subset of a program's state (variables) have particular values. Considered in this way, choosing an oracle is analogous to choosing a point on the continuum from checking nothing to checking the entire state of the program. While neither extreme is appropriate—oracles that check nothing will never find bugs and oracles that check everything will likely be difficult to maintain and enormous—there is a point somewhere between that represents the ideal oracle. Unfortunately, identifying this point is challenging. In practice, the oracles written by testers often miss the mark by either: (1) checking too little by failing to include assertions for relevant variables—which can result in tests that are unable to reveal failures (i.e., missed warnings), or (2) checking too much by including assertions about irrelevant variables—which can lead to brittle tests that fail when they should not (i.e., false warnings). Existing work on assessing the quality of test oracles addresses only the first of these possibilites by detecting tests that are likely missing assertions (e.g., [21, 23, 25]).

In this paper, we present a novel dynamic analysis technique that addresses both possibilites. The technique is based on dynamic tainting and works by tracking the flow of controlled and uncontrolled inputs along data- and control-dependencies at runtime. Intuitively, controlled inputs are inputs explicitly provided by the test itself (e.g., constants that appear in the test method) and all other inputs are considered uncontrolled. When a test finishes execution, the technique uses the tracked information to generate reports that identify *brittle assertions*—assertions that check values that are derived from inputs that are not controlled by the test and *unused inputs*—inputs that are controlled by the test

FSE'14, November 16–21, 2014, Hong Kong, China
ACM 978-1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2635917

```
1 public class EmployeeTest extends TestCase {
2  String firstName, lastName, ssn;
3  double baseSalary, commissionRate = 0.5, grossSales;
4  Employee E = new Employee(firstName, lastName, ssn,
5                            baseSalary, commissionRate,
6                            grossSales);
7
8  public void testToString() {
9    E.setFirstName("John")
10   E.setGrossSales(200);
11   E.setBaseSalary(100);
12   String expected = "Employee: null\n" +
13                     "social security number: null\n" +
14                     "total salary: 200.00";
15   assertEquals(E.toString(), expected);
16 }
17
18 public void testAbbreviateLastName() {
19   E.setLastName("Moore-Towers");
20   String expected = "Moore";
21   assertEquals(E.abbreviatedLastName(), expected);
22 }
23 }
```

Figure 1: An example of a brittle test case.

but are not checked by an assertion. These reports are then filtered to remove false positives and presented to testers.

To evaluate our technique, we created a prototype implementation that analyzes Java applications and tests written using the JUnit testing framework.[1] We used the prototype tool to analyze over 4,000 tests from real, open source software projects and to answer several research questions about (1) the feasibility and effectiveness of the technique, and (2) the quality of existing test oracles.

This work makes the following contributions:

- The definition of a new technique that can automatically analyze tests to detect both brittle assertions and unused inputs.
- A prototype implementation of the technique that implements the technique for Java applications with test cases written using JUnit.
- An extensive empirical study that demonstrates our tool's feasibility, accuracy, and usefulness.

2. MOTIVATING EXAMPLE

In this section, we provide an example that will be used in the remainder of the paper to illustrate our technique. Figure 1 shows the example, which consists of several tests derived from the test suite for CommissionEmployee, a small application that is used to perform various computations necessary to calculate payroll information for sales employees. Consider `testToString` which is checking whether `Employee`'s `toString` method produces the expected output. While this test fulfills its goal, it has several problems.

First, `testToString` is brittle because it makes assertions about values derived from inputs that it does not control. More specifically, the test assumes that the values of the employee's last name, social security number, and commission rate are not changed between the time when the employee is created and the time when the result of `toString` is checked. Note that no such assumption is made about the employee's first name, gross sales, or base salary as these values are controlled by the test (i.e., they are explicitly set to "John", 200 and 100, respectively, during the execution of the test).

[1] `http://www.junit.org`

```
1 int x, y, z;
2 int a = input();
3 int b = input();

4 x = a - 2;
5 y = b * 4;
6 z = x + y;
```

(a)

```
1 int x, y;
2 int a = input();

3 if(a > 0)
4    x = 0;
5 else
6    x = 1;
7 y = 2;
```

(b)

Figure 2: Code examples that illustrate information flow through data and control dependencies.

In practice, there are many ways that `testToString`'s assumption that the employee's last name, social security number and commission rate are not changed could be violated. For example, if `testAbbreviateLastName` was added to the test suite, `testToString` would fail intermittently, depending on the order in which the test cases are executed. If `testToString` is executed first, the assumption holds and both tests will pass. However, if `testAbbreviateLastName` is executed first, the assumption is violated and `testToString` will fail because the value of the employee's last name will no longer be null. To prevent the possibility of failures due to brittleness, a test should not check values derived from inputs that it does not control. For `testToString`, this can be accomplished by explicitly controlling the values of the employee's last name, social security number, and commission rate, as is done for first name, gross sales, and base salary. This can also be accomplished by creating a new instance of `Employee` with known values or by explicitly setting the employee's last name, social security number, and commission rate. Both of these options fix the test's brittleness by ensuring that the values checked by the its oracle are derived from controlled inputs.

The second problem with `testToString` is that one of the test's controlled inputs is unused. Although the test specifically sets the employee's first name to "John", none of its assertions check values derived from this input. Unused inputs suggest that the test's author is unsure about the behavior of the application under test, possibly because modifications made to the application have not been reflected in the test or simply because the tester was unfamiliar with the code when the test was written. In the worst case, an unused input indicates that a test is missing an assertion which could lead to missed warnings—situations where the test should fail but does not. Even if it they do not lead to missed warnings, unused inputs increase the costs of test maintenance by increasing the cognitive burden on the tester. To eliminate unused inputs, additional assertions could be added to the test (e.g., adding `assertEquals(E.getFirstName(), "John")` to `testToString`) or the unused inputs could be removed (e.g., deleting Line 9 in `testToString`).

3. BACKGROUND

In this section, we provide background information on dynamic tainting. Note that the material in this section is paraphrased from our previous work on dynamic tainting [4]. Intuitively, dynamic tainting consists of (1) marking some data values in a program with a piece of metadata called a taint mark, and (2) propagating taint marks according to how data flows in the program at runtime. In this way, dynamic tainting can track and check the flow of information through a program while it executes.

Information flows through a program in two ways: through data dependences and through control dependences. We illustrate these two kinds of flows using the code examples in Figure 2. First, consider the code in Figure 2a. Assume that variable a is tainted with taint mark t_a at Line 2 and variable b is tainted with taint mark t_b at Line 3. Given this assignment of taint marks, variables x, y, and z would be tainted, at the end of the execution, with sets of taint marks $\{t_a\}$, $\{t_b\}$, and $\{t_a, t_b\}$, respectively. Taint mark t_a would be associated with x because the value of a is used to calculate the value of x (`x = a + 2`), that is, x is data dependent on a. Analogously, y would be tainted with t_b because the value of b is used to calculate the value of y (`y = b * 4`). Finally, z would be tainted with both t_a and t_b because the values of both x and y are used to compute the value of z (`z = x + y`), that is, z is indirectly data dependent on both a and b.

Consider now the code in Figure 2b and assume that variable a is tainted with taint mark t_a at Line 2. Although a's value is not directly involved in the computation of x in this case, it nevertheless affects x's value: the outcome of the predicate at Line 3 decides whether Line 4 or Line 6 will be executed, that is, the statements at Lines 4 and 6 are control dependent on the statement at Line 3. Therefore, the value of x at the end of the execution would be associated with taint mark t_a.

In general, the propagation of taint marks along data dependences is called *data-flow propagation* and the propagation along control dependences is called *control-flow propagation*.

4. DETECTING BRITTLE ASSERTIONS AND UNUSED INPUTS

This section presents our technique for helping testers improve the quality of their test oracles. We first provide an intuitive description of the technique and then discuss its main characteristics in the following 4 steps.

4.1 Overview

The overall goal of the technique is to reduce the costs of testing by automatically identifying, and helping testers fix, both brittle assertions and unused inputs. The basic intuition behind the approach is that dynamic tainting, due to its ability to mark and track inputs at runtime, can be successfully used to accomplish this goal. In this spirit, our approach works by (1) assigning taint marks to two types of inputs: inputs that are controlled by the test and inputs that are not controlled by the test, (2) tracking both types of inputs by suitably propagating the taint marks as a test executes, and (3) identifying, when an assertion is executed, which taint marks are associated with the values checked by the assertion. The taint marks discovered in the third step allow for identifying situations where a test checks too much (i.e., is brittle) and situations where a test checks too little (i.e., has unused inputs).

Before presenting the details of the approach, we discuss how it works on `testToString` from Figure 1. This test is brittle because it contains assertions about values derived from uncontrolled inputs, and also has unused inputs because some controlled inputs, or values derived from them, are not checked by an assertion. Figure 3 provides an intuitive view of how our technique can detect both problems.

The top of Figure 1 shows `testToString`'s inputs. Note that the inputs are divided into two categories: controlled

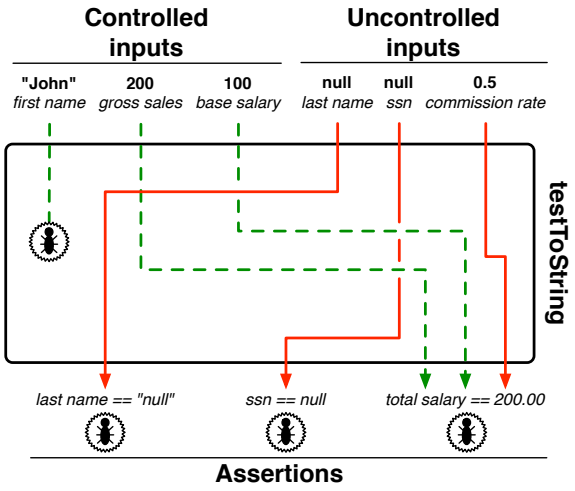

Figure 3: Intuitive view of the application of our technique to `testToString` from Figure 1.

inputs and uncontrolled inputs. Intuitively, controlled inputs are values that are provided as part of the test itself (in this case, "John", 200 and 100) and uncontrolled inputs are inputs that are not explicitly set during the execution of the test (in this case null, used to initialize `lastName` and `ssn`, and 0.5, used to initialize `commissionRate`). Section 4.2 provides a detailed discussion of how the technique identifies controlled and uncontrolled inputs. To make the example more clear, each input shows both the value (top) and a brief description of what the value represents (bottom). For example, the value "John" represents the first name of the employee, the value 200 represents the employee's gross sales value, etc.

The bottom of Figure 1 shows, conceptually, the test's oracle. The call to `assertEquals` at Line 15 implicitly checks whether the employee's first name is equal to "John", whether the employee's social security number is equal to null, and whether the employee's total salary is equal to 200.00.

The lines that traverse `testToString` illustrate, intuitively, how our technique assigns a unique taint mark to each input. In the figure, each input is the source of a unique line. In addition, the color and style of the lines indicate the type of input: green, dashed lines indicate taint marks assigned to controlled inputs and solid, red lines indicate taint marks assigned to uncontrolled inputs. In the remainder of the paper, we refer to taint marks assigned to controlled inputs as c-marks and taint marks assigned to uncontrolled inputs as u-marks. The lines in the figure also illustrate how our technique tracks inputs at runtime, by propagating the taint marks as the test executes. For example, the line connecting the input "null" to the assertion `last name == null` indicates that the value checked by the assertion (the employee's last name) is derived from the value null. Similarly, the lines that connect the inputs 200, 100, and 0.5 to the assertion `total salary == 200.00` indicate that the value checked by the assertion (the employee's total salary), is derived from the values 200, 100, and 0.5. The technique uses this information to detect brittle assertions and unused inputs. In Figure 3, both types of errors are shown using a bug icon.

Brittle assertions are detected by (1) identifying, when an assertion occurs, the set of taint marks associated with the values checked by the assertion, and (2) checking whether the set of taint marks contains a u-mark. If the set does contain a

u mark, the assertion is considered to be brittle. For example, as Figure 3 shows, there are three taint marks associated with the value checked by the assertion `total salary == 200`: the taint marks for 200, 100, and 0.5. Because the taint mark for 0.5 is a u-mark, the technique identifies this assertion as brittle. Similarly, the other two assertions are also identified as brittle because the values that they check are tainted with a u-mark.

Unused inputs are detected by (1) computing the union of all taint marks associated with every value checked by an assertion, and (2) checking whether the union contains every c-mark assigned during the execution of the test. If a c-mark is not present in the union, its corresponding input is unused. For example, in Figure 3 the union of all taint marks associated with values checked by the assertions does not contain the c-marks for "John". Consequently, "John" is identified as an unused input.

In addition to detecting brittle assertions and unused inputs, the information provided by propagating taint marks is used to give testers additional data about the identified errors. Intuitively, the technique tracks backwards to identify the origins of the problems and outputs the source of each input (i.e., the locations where the controlled and uncontrolled inputs were assigned a taint mark). These locations can serve as a starting point to help testers fix the identified problems with their assertions.

4.2 Input Tainting

Input tainting is responsible for associating taint marks with a test's inputs. The technique intercepts the execution of the test at specific points and assigns either an c-mark, for controlled inputs, or a u-mark, for uncontrolled inputs.

4.2.1 Tainting Controlled Inputs

Currently, our technique considers two types of values to be controlled inputs. First, the technique considers values (constants) that are (1) used during the execution of the test method itself, and (2) not passed to an assertion method to be controlled inputs. For example, in Figure 1 the constant "John" used at Line 9 is a controlled input, but the constant "Moore" at Line 20 is not because it is used as an argument to `assertEquals` on Line 21. The decision to only include constants in the test method itself is based on our experience, domain knowledge, and intuition about how testers write tests. Initially, the technique also considered constant values from the test's setup code to be controlled inputs. However, we found that many tests use the same setup code to construct a complex state. While, when considered individually, the tests appear to have unused inputs (i.e., parts of the common state that are not checked), when they are considered collectively, they check the entire state. In future work, we plan to extend the technique to consider entire test suites rather than individual tests when identifying missing assertions (see Section 7).

Second, the technique considers the return values of no-argument methods called in the test method itself and also implemented in the test class to be controlled inputs. We found that no-argument methods are frequently used to separate long sequences of initialization code from a test. Moving such initialization code to a separate method decreases the size of the test, which can improve its readability and understandability. Because such methods are conceptually part of the test, we consider their results to be controlled inputs.

To taint constants that are used as part of the test method, the technique simply intercepts the loading of the constants and applies a unique c-mark to the loaded constant. Similarly, to taint the return value of no-argument methods implemented in test suite, the technique intercepts the test's execution immediately after the method returns and applies a unique c-mark to the return value. Note that if either type of controlled input is used repeatedly, as would be the case inside of a loop, the technique reuses the same c-mark for each iteration. Based on the results of our experiments (see Section 5), we found that this approach produces the most understandable reports. From the point of view of a tester, regardless of the number of times an input is used, it is still, conceptually, the same input.

As a concrete example of how our technique assigns c-marks to controlled inputs, consider `testToString` in Figure 1. The technique identifies three controlled inputs in this test: the literal value "John" used at Line 9, the literal value 200 used at Line 10, and the literal value 100 used at Line 11. When each of these constants is loaded, the technique assigns a unique c-mark to each value (e.g., c_1 is assigned to "John", c_2 is assigned to 200, and c_3 is assigned to 100).

4.2.2 Tainting Uncontrolled Inputs

Currently, our technique considers two types of values to be uncontrolled inputs. First, the values of global variables (static, mutable fields) are considered to be uncontrolled inputs. The intuition behind this choice is that reading the value of a global variable is the most likely way for a test to unintentionally depend on a value that it does not control. Because globals variables maintain their state and can be written to at any time, a test has no way of knowing what values they contain. Similarly, tests can also unintentionally depend on the contents of the files, databases, network connections, etc. We chose not to consider values read from such sources as uncontrolled inputs because unit tests typically use mock objects instead of the real resources.

Second, the values of all non-final fields in the test's containing class are considered as uncontrolled inputs. We choose to consider such values because the fields of the test's class's are essentially global variables within the context of the test class. Their values can be changed by any test method inside the class.

To assign u-marks to global variables, the technique iterates over all of the classes that are loaded at the start of the test. For each loaded class, the technique assigns a unique u-mark to each non-final, static field.

To assign u-marks to the values of the test's class's fields, it is necessary to understand how fields are initialized by the Java Virtual Machine (JVM). When an object is initialized, each of its fields is assigned the initial value for its type.[2] Then, if the field has an variable initializer (e.g., `public int i = 5;`), code added by the compiler to the object's constructors evaluates the variable initializer and assigns the result to the field. This means that, in order to assign taint marks to the test's class's fields, the technique must intercept object initialization as well as the execution of variable initializers. If only object initialization was intercepted, the taint marks associated with the fields would be overwritten when the result of the variable initializer was assigned to the field (see Section 4.3 for additional information on how taint marks

[2] `http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.12.5`

are propagated). If only the execution of variable initializers was intercepted, fields without a variable initializer would not be assigned a taint mark.

To assign taint marks to the initial values of the test's class's fields, the technique assigns a unique u-mark to each field at the end of object initialization. To assign taint marks to the results of variable initializers, the technique identifies the code added by the compiler by checking the debug information. Compiler-added code will have a source line location that is outside the bounds of the constructor. After identifying the compiler-added code, the technique intercepts the execution immediately after the variable initializer is evaluated and applies a unique u-mark to the result. Note that taint marks assigned to the results of variable initializers will overwrite the taint marks assigned during object initialization. While this does not impact the performance of the technique—the only consequence is that unnecessary taint marks are created—a simple static-analysis could be used to identify fields with variable initializers. Such fields could then be skipped when taint marks are assigned during the initialization process.

As a concrete example of how our technique assigns u-marks to uncontrolled inputs, consider `testToString` in Figure 1. `EmployeeTest` has six fields, only one of which has a variable initializer (`commissionRate`). When an instance of `EmployeeTest` is initialized, all six of its fields are assigned a unique u-mark (e.g., `firstName` is assigned u_1, `lastName` is assigned u_2, `ssn` is assigned u_3, `baseSalary` is assigned u_4, `commissionRate` is assigned u_5, and `grossSales` is assigned u_6). During the execution of `EmployeeTest`'s constructor, when `commissionRate` is assigned the value 0.5, the taint mark assigned during object initialization is overwritten with a fresh taint mark (e.g., u_5 is overwritten by u_7). Later when `testToString` is executed, the technique would assign a unique u-mark to each static, mutable field of every currently loaded class.

4.2.3 Recording Supplemental Information

Regardless of the type of taint mark, our technique performs an additional action when assigning a taint mark t to an input i. To help testers debug the problems identified by the technique, additional information is recorded. More specifically, the technique logs a tuple $\langle t, loc, value \rangle$, where *loc* is the location in the execution where the taint mark was associated with the input and *value* is the initial value of the input. The location is expressed differently depending on the type of taint mark. For c-marks, the location is the file name and line number corresponding to where the constant or return value was used. For u-marks, the location is the fully qualified name (field name and declaring class name) of the field that contains the input. This information is used by our technique when generating reports, as described in Section 4.4.

4.3 Taint Mark Propagation

A taint propagation policy specifies how taint marks are propagated during execution. Typically it is defined along two dimensions: how to combine taint marks and which types of dependences to consider.

Our technique's policy for combining taint marks is fairly intuitive. In general, the technique taints all values written by a statement with the union of all taint marks associated with the values read by that statement. For instance, after the execution of statement `x = y + z`, where `y` and `z` are tainted with taint marks t_1 and t_2, respectively, `x` would be associated with the set of taint marks $\{t_1, t_2\}$. The only type of statement where the techniques deviates from this general policy is the execution of native methods. Because native methods are executed by the JVM, it's is often unclear which values are read by the native method. Rather than require a precise model of every native method, the technique conservatively assigns the union of all taint marks associated with the native method's arguments to its return value.

When choosing which dependences to consider, our technique considers both data-flow and control-flow dependencies. Identifying data-flow dependencies is trivial as they are encoded as the semantics of the language. Identifying control-flow dependencies is more challenging.

A control-flow dependence arises when a conditional branch b decides whether a statement s is executed. In this case, the values that affect b's outcome indirectly affect the values of any data written by s. Therefore, to be conservative, the taint marks associated with the values read by b must be combined and associated with the values written by s. To achieve this, our technique uses an approach that we proposed in prior work [5]. In brief, the technique keeps track of relevant taint marks at runtime by leveraging statically-computed post-dominance information. When an execution reaches a conditional branch b, the technique (1) computes T, the union of the taint marks associated with the values read by b, and (2) adds a pair $\langle b, T \rangle$ to CF, the set of active control flow marks. When execution reaches the immediate post-dominator of a conditional branch b, it removes from CF all pairs $\langle x, y \rangle$ such that x is equal to b. Note that CF will contain multiple pairs with x equal to b when b is executed as part of a loop. Each iteration will add a new pair $\langle b, T \rangle$ to CF, all of which must be removed when the immediate post-dominator is executed. When a statement s is executed and CF is not empty, the technique computes the union of all active control flow marks (i.e., the union of y, for each pair $\langle x, y \rangle$ in CF) and adds this set to the set of taint marks associated with the values written by s.

4.4 Checking Taint Marks

This third part of our technique, checking, is responsible for two tasks: (1) identifying brittle assertions and unused inputs, and (2) generating the error reports that will be presented to testers.

To identify brittle assertions, the technique intercepts the execution of comparison operations (e.g., greater than, less than, equals, etc.) that occur inside the execution of an assertion method (e.g., `assertEquals`). Intercepting the execution of comparison operations, rather than simply examining the actual argument of the assertion method, allows for a more precise identification of brittle assertions. For example if the actual parameter of the assertion method is an object, examining the taint marks associated with all of the object's fields is likely to be incorrect as not all of the fields are necessarily involved in checking whether the actual and expected arguments are equal. Rather than attempting to identify, a priori, which values are used to check for equality, the technique can simply monitor the comparison operations to achieve the same effect. In addition, testers often inadvertently swap the order of the actual and expected arguments which means that the actual parameter may not in the correct location which would result in incorrect reports.

```
testToString appears to be brittle.  The assertions at the
following lines check values that are derived from
uncontrolled inputs:

assertEquals at Line 25 depends on:
EmployeeTest.lastName being null (object initialization)
EmployeeTest.ssn being null (object initialization)
EmployeeTest.commissionRate being 0.5 (object initialization)
```

(a) **Report for brittle assertions.**

```
testToString appears to be missing one more assertions. The
following values are provided as input but are not checked by
an assertion:

"John" (EmployeeTest.java, Line 9)
```

(b) **Report for unused inputs.**

Figure 4: Example reports output by our technique when run on `testToString` from Figure 1.

For each comparison operation inside of an assertion method, the technique identifies the taint marks associated with the values involved in the comparison and checks whether the set of identified taint marks contains a u-mark. If a u-mark is found, the technique detects a brittle assertion.

To identify unused inputs, the technique calculates the union of all c-marks that were encountered when checking for brittle assertions. The technique then subtracts this set from the set of all c-marks that were assigned to controlled inputs. If the resulting set is not empty, the inputs initially assigned with the remaining c-marks are marked as unused.

As a concrete example of how the checking part of the technique operates, consider again `testToString` from Figure 1. For this test, the technique would intercept the comparison operations that occur inside the call to `assertEquals`. Because the actual value is a string, the `String` class's `equals` method is used to perform the check. The `equals` method uses a series of equality comparisons (i.e., `==`) to check whether the same characters make up each string. Each time this equality is executed, the technique determines whether either character is tainted with a u-mark. Because the actual value is derived from three uncontrolled inputs, three u-marks are found and a brittle assertion is detected.

At the end of the test's execution, the technique calculates the union of all encountered c-marks. Because only values tainted with only two of three total c-marks were checked by the assertion, an unused input report for the remaining mark (the one initially assigned to "John") is created.

Figure 4 shows the error reports generated by the technique when run on `testToString`. As the figure shows, both the brittleness report (top) and the unused input report (bottom) include all of the information necessary to help testers fix the identified issues. The brittleness report includes the name and location of the brittle assertion (assertEquals on Line 25), the uncontrolled values that were used to compute the values checked by the assertion (null, null, and 0.5), the names of the fields where the uncontrolled values were stored (EmployeeTest.firstName, EmployeeTest.ssn, and EmployeeTest.commissionRate), and the locations where the uncontrolled values were stored into the fields (during object initialization). Note that to collect the location information, the technique traverses the test's call stack to find the name of the outermost enclosing assertion method invocation and the location where the assertion method was invoked. The unused input report includes the controlled inputs that were not checked by an assertion and the line number where the value was loaded.

4.5 Removing False Positives

The purpose of the fourth part of the technique is to filter false positive error reports. Taint mark propagation is known to be imprecise, especially in the case of native methods. As a result, error reports generated by the third part of the technique may be false positives. More specifically, the technique may generate false positive reports if it under-propagates c-marks or over-propagates u-marks. As a result, controlled inputs may appear to be unused when in fact they are used and uncontrolled inputs may appear to be checked by an assertion when in fact they are not.

To eliminate such false positives, the technique uses an approach inspired by mutation testing [7, 10]. Essentially, the technique preemptively makes changes that may happen in the future and checks to see whether such changes alter the outcome of the test. More specifically, for each input that is identified as the cause of a brittle assertion or as an unused input, the technique re-executes the test. As the test is being re-executed, at the point when the taint mark was assigned to the input in the original execution, the technique mutates the value of the input to a randomly chosen value of the same type. Note that data- and control- flow analysis is not needed to accomplish this. The technique then compares the outcomes of the re-executed and original executions.

In the case of uncontrolled inputs that cause brittle assertions, we would expect that changing the input's value would alter the outcome of the test. If a test checks values derived from an input, changing the value of the input should change the outcome of the test. If the outcome of the test does change, the report is a true positive (i.e., if the change were to be made, the test would fail) and is presented to the user. Conversely, if the outcome of the test does not change, the report is a false positive (i.e., the u-mark should have been over written but was not) and is discarded.

In the case of unused inputs, we would expect that changing the value of the input would not alter the outcome of the test. If an input is really unused, its value doesn't matter. If the outcome of the test does not change, the unused input report is a true positive and is presented to the user. Conversely, if the outcome of the test does change, the error report is a false positive (i.e., the c-mark should have propagated to an assertion but did not) and is discarded.

Note that this filtering strategy is precise—reports that are identified as true positives have an associated witness (a concrete change that will cause the test to fail)—but not safe—reports that are identified as false positives may actually be true positives. True positives can be identified as false positives from the point-of-view of the test when the randomly chosen value is indistinguishable from the original value. For example, consider an assertion that checks whether a value is positive. If the original value checked by the assertion is 1 and the randomly chosen replacement is 5, the outcome of the assertion will be the same for both values. To reduce the possibility of this occurring, multiple re-executions, each with a unique value, can be run or additional analysis could be performed to identify values that are more likely to cause the outcome of the test to change.

In the case of `testToString`, both of the error reports generated by the third part of the technique are true positives. Changing the value of `commissionRate` causes the test to fail and changing the value of the employee's first name does not cause the test to fail.

5. EVALUATION

To evaluate our technique, we created a prototype implementation called OraclePolish and analyzed over 4,000 tests for real Java applications. Using the output of the tool, we investigated the following research questions:

RQ 1—Effectiveness. Can the technique detect both brittle assertions and unused inputs in real test suites?

RQ 2—Cost. What are the costs associated with using the technique and are they reasonable?

Note that RQ1 provides a quantitative assessment of the technique; it does not make any assumptions about whether the reported errors are likely to cause problems in the future. Conversely, RQ2 is a qualitative assessment that does take into account the users perspective.

The remainder of this section describes (1) OraclePolish, the prototype implementation of our technique, (2) the experimental subjects we chose, (3) the experimental protocol we used and the data we generated, (4) the results of evaluation, and (5) threats to the validity of our results.

The prototype implementation of our technique, as well as the subjects we chose and the experimental data we generated, are available from: `http://bitbucket.org/udse`.

5.1 Prototype Tool

OraclePolish is a prototype implementation of our technique for applications written in the Java language using the JUnit testing framework. It consists of three separate components: the *analyzer*, the *runtime system*, and the *mutator*.

The primary task of the analyzer is to statically compute the information needed by the runtime system. More specifically, the analyzer computes the post-dominance information needed to perform control-flow propagation. The current implementation of the analyzer uses the T.J. Watson Libraries for Analysis (WALA) to perform the necessary analyses. We choose WALA because it (1) analyzes Java bytecode, which means that we do not need to obtain the source code for all parts of the application, (2) provides built-in dominator analyses, and (3) is extensible enough to allow us to easily implement the other necessary analyses.

The runtime system implements the input tainting, taint mark propagation, and checking parts of the technique described in Section 4.2, Section 4.3, Section 4.4, respectively. The current implementation of the runtime system is an extension to Java PathFinder (JPF), an explicit state software model checker for Java software.[3] To assign taint marks to inputs, OraclePolish uses JPF's listener callbacks to intercept class and object initialization and to intercept the execution of instructions that load constants. To implement taint mark propagation, OraclePolish uses JPF's bytecode overloading facilities to replace each Java bytecode with a modified version that replicates the instruction's original semantics while also propagating taint marks. Finally, to implement checking, OraclePolish again uses JPF's listener callbacks to intercept the execution of comparisons instructions that occur inside of assertion methods.

The mutator implements the part of the technique that filters false positives (see Section 4.5). It is also implemented as a plugin to JPF. Similarly to the runtime system, the mutator uses JPF's listener callbacks to intercept class and object initialization and to intercept the execution of instructions that load constants. However, instead of assigning taint marks, the mutator randomly changes the values of the inputs. Currently, the mutator re-executes the test three times. As we demonstrate in Section 5.4, this number is sufficient to eliminate many false positives.

[3] `http://javapathfinder.sourceforge.net/`

5.2 Subjects

The goal of the technique is to improve oracle quality by detecting brittle assertions and unused inputs. To suitably evaluate the technique with respect to this goal, we selected the test suites of 20 applications as our subjects. Table 1 describes the applications. In the table, the first column, *Subject* shows the name and version of each application, if available. The first eight applications (CommissionEmployee through Sudoku) are taken from the Proteja Test Suite Executor and Coverage Monitor repository.[4] The remaining subjects were obtained from various repositories including: (1) the Software-artifact Infrastructure Repository (SIR),[5] which provides a variety of open-source projects for empirical software engineering, (2) SourceForge,[6] a popular repository for open-source projects, and (3) Apache Commons,[7] a collection of reusable components. The second column, *LoC*, shows the number of non-blank, non-comment, lines of code that comprise the application and the third column, *# Tests* shows the number of tests in each application's test suite.

We chose the test suites of these applications as subjects for several reasons. First, the applications cover a variety of subject domains. For example, Commons CLI is a library for processing command-line options, Commons IO is a library for performing various input/output operations, Joda-Time is a library for handling dates and times, etc. Second, the applications vary in size. For example, Commons-math has over 70,000 lines of code, while Sudoku only has 376 lines of code. Finally, the test suites also vary in size. The test suites for some of the application contain more than 3,000 tests while others contain fewer than 20. Selecting test suites and applications of various sizes and subject domains improves the generalizability of our results.

After selecting our subjects, we performed an initial sanity check and removed any tests that can not be run using JPF. The number of remaining tests is shown in the fourth column, *# Executable*. For example, although Commons-beanutils-1.8.3's test suite contains 810 tests, 73 of which are executable using JPF. After filtering, we were left with 4,718 tests.

5.3 Experimental Protocol and Data

To generate the experimental data necessary for answering our research questions, we ran OraclePolish on each of our 4,718 tests and recorded its output. The experiments were all conducted on the same computer: a machine running Ubuntu 12.04 LTS 64-bit edition with a 3.40 GHz Intel Core i7-2600 processor and 8 GB of memory. Java version 1.7.0_03 was used and was configured with 2 GB of heap space (default).

Table 1 shows the experimental data that we generated. The last four columns in the table show the number of reports generated by the technique, *# Reports*, and the number of reports that are true positives, *# TP*, for both *Brittle Assertions* and *Unused Inputs*. The number of reports is the

[4] `https://code.google.com/p/proteja/`

[5] `http://sir.unl.edu`

[6] `https://sourceforge.net`

[7] `http://commons.apache.org`

Table 1: Experimental subjects and data.

Subject	LoC	Test Suite		Brittle Assertions		Unused Inputs	
		# Tests	# Executable	# Reports	# TP	# Reports	# TP
CommissionEmployee	100	15	15	2	1	13	13
DataStructures	429	106	99	0	0	55	47
Employee	183	15	15	3	3	11	11
LoopFinder	49	13	13	0	0	13	5
Point	69	13	11	0	0	10	8
ReductionAndPriority	3,245	52	38	0	0	37	37
Sudoku	376	25	18	0	0	7	0
commons-beanutils-1.8.3	11,375	810	73	12	4	59	44
commons-cli-1.2	1,978	164	130	22	5	119	24
commons-collections-3.2.1	26,414	886	672	20	12	426	247
commons-io-2.4	26,614	824	236	1	0	154	78
commons-lang-3.1	23,070	2,024	1,547	128	114	1,133	549
commons-math-3.0	70,006	1,150	72	0	0	39	12
JDepend-2.9.1	2,531	39	0	0	0	0	0
Jfreechart-1.0.15	92,252	2,234	862	1	0	538	285
joda-convert-1.2	2,675	105	0	0	0	0	0
Joda-time-2.2	86,797	3,962	506	181	25	226	144
Jtopas-0.8	4,373	53	27	0	0	21	11
PMD-5.0.4	100,300	770	346	2	0	184	93
total		13,609	4,718	405	164	3,060	1,618

total number of reports generated by the checking part of the technique (see Section 4.4) and the number of true positives is the number of reports that remain after being filtered by the fourth part of the technique (see Section 4.5).

5.4 RQ1: Effectiveness

The purpose of our first research question is to determine the effectiveness of the technique at detecting brittle assertions and unused inputs in real tests. To answer this question, we first judged effectiveness quantitatively, by examining the number of true positive reports generated by OraclePolish.

As Table 1 shows, for the subjects we considered, OraclePolish was able to detect both brittle assertions and unused inputs. In total, it detected 164 tests that contain at least one brittle assertion and 1,618 tests that contain unused inputs. These results are encouraging and also a bit surprising. Because most of the tests that we considered are from the test suites of mature applications, we expected them to contain few errors.

It is interesting to note that OraclePolish detects far more unused inputs than brittle assertions. Intuitively, this makes sense as unused inputs are unlikely to cause any observable problems. While missing assertions may cause a test to pass when it should fail, there is no way to detect this occurrence. Similarly, there is not an easy way to measure the amount of additional effort needed to comprehend and maintain tests with unused inputs. As a result, unused inputs are more likely go undetected and unfixed than brittle assertions.

The second way we judged effectiveness was by qualitatively assessing the reports generated by OraclePolish. In the remainder of the section, we provide a more detailed discussion of two randomly chosen brittle tests and two randomly chosen tests with unused inputs.

Figure 6 shows an excerpt of `test13666` that is part of the test suite for Commons-cli. OraclePolish detects that the assertion at Line 252 is brittle because it depends on several

```
    public class BugsTest extends TestCase {
      public void test13666() throws Exception {
229     Options options = new Options();
230     Option dir = OptionBuilder.withDescription( "dir" )
                                  .hasArg()
                                  .create( 'd' );
233     options.addOption(dir);

236     final PrintStream oldSystemOut = System.out;
237     try {
239       OutputStream bytes = new ByteArrayOutputStream();

247       System.setOut(new PrintStream(bytes));

249         HelpFormatter formatter = new HelpFormatter();
250         formatter.printHelp( "dir", options );

252         assertEquals("usage: dir"+eol+" -d <arg>   dir"
                         + eol,
                         bytes.toString());
        }
        finally {
256       System.setOut(oldSystemOut);
        }
      }
    }
```

Figure 5: Brittle assertions in `test13666`

of `OptionBuilder`'s static fields. Because `OptionBuilder` is a singleton, it is possible for other users of the class to leave it in an indeterminate state by starting to build an option but never calling `create`. Internally, create resets the state of the `OptionBuilder` so that it is safe to reuse. To prevent the possibility that `OptionBuilder` has already been partially configured, the test should call `OptionBuilder`'s `reset` method before using starting to build an option at Line 230.

Figure 6 shows an excerpt of `testPut` that is part of test suite for Commons-beanutils. OraclePolish detects that the assertion at Line 246 is brittle because it checks the value of `stringVal`. As the code shows, `stringVal` is a static field of the `DynaBeanMapDecoratorTestCase`. Because `testPut` does not control the value of `stringVal`, it is assuming that

```
   public class DynaBeanMapDecoratorTestCase extends TestCase {
43   private static final DynaProperty[] properties =
       new DynaProperty[] { ... };

47   private static String stringVal = "somevalue";

52   private Object[] values = new Object[] {stringVal, ...};

54   private BasicDynaBean dynaBean;

     public void setUp() throws Exception {
96     dynaBean = new BasicDynaBean(dynaClass);
97     for (int i = 0; i < properties.length; i++) {
98         dynaBean.set(properties[i].getName(), values[i]);
99     }

103    modifiableMap = new DynaBeanMapDecorator(dynaBean,
                                                false);
     }

     public void testPut() {
235    String newValue = "ABC";

246    assertEquals(stringVal,
                    modifiableMap.put(stringProp.getName(),
                                      newValue);
247    assertEquals(newValue,
                    dynaBean.get(stringProp.getName()));
248    assertEquals(newValue,
                    modifiableMap.get(stringProp.getName()));
     }
   }
```

Figure 6: Brittle assertions in `testPut`

```
   public class DefaultKeyedValues2DTests extends TestCase {
     public void testGetRowKey() {
257    DefaultKeyedValues2D d = new DefaultKeyedValues2D();

266    d.addValue(new Double(1.0), "R1", "C1");
267    d.addValue(new Double(1.0), "R2", "C1");
268    assertEquals("R1", d.getRowKey(0));
269    assertEquals("R2", d.getRowKey(1));
     }
   }
```

Figure 7: Unused inputs in `testGetRowKey`

`stringVal` will not be modified between the time when the field is initialized and the time when the assertion is executed. To fix this error, the reference to the static field could be replaced with the expected constant. Alternatively, if the field were to be made final it would be guaranteed to have the expected value. Because the majority of `DynaBeanMapDecoratorTestCase`'s other fields are final, this later option is likely to be the correct fix.

Figure 7 shows an excerpt of `testGetRowKey` from JFreeChart's test suite. OraclePolish detected two unused inputs in this test: the value "C1" at Line 266 and the value "C1" at Line 267. Although these values are used as arguments to the calls to `addValue` at Line 266 and Line 267, they are not checked by an assertion. Adding additional assertions (i.e., `assertEquals("C1", d.getColumnKey(0))` and `assertEquals("C1", d.getColumnKey(1))`) would ensure that not only are the correct row keys returned, but also that the column keys are not modified.

Figure 8 shows an excerpt of `test13` from Employee's test suite. OraclePolish detected three unused inputs in this test: "FN" at Line 160, "SN" at Line 161, and "ssn" at Line 162. Although these values are used to construct `s2`, a new instance of `SalariedEmployee`, they are never checked by an assertion. Note that `s1` is used to construct the actual value passed to `assertEquals` at Line 166, not `s2`. In this case, it is not clear how to best fix the test. The unused inputs could be deleted or the actual value could be constructed using `s2` instead of `s1`.

```
   public class EmployeeTest extends TestCase {
 8   private String fn, ln, ssn;
 9   private double s;

20   SalariedEmployee s1 = new SalariedEmployee(fn,ln,
                                                ssn,s);

158  public void test13() {
159    s1.setWeeklySalary(10);
160    fn = "FN";
161    ln = "SN";
162    ssn = "ssn";
163    SalariedEmployee s2 = new SalariedEmployee(fn,ln,
                                                  ssn,s);
164    String actual = s1.toString();
165    String expected = "salaried employee: null null\n"
                       + "ssn: null\n"
                       + "weekly salary: $10.00";
166    assertEquals(actual, expected);
167  }
   }
```

Figure 8: Unused inputs in `test13`

5.5 RQ2: Cost

The purpose of our second research question is to investigate the costs of using OraclePolish and to determine if such costs are reasonable. Because our technique is fully automated, the primary cost is its runtime overhead.

To investigate the runtime overhead that OraclePolish imposes, we executed our subject tests twice, once using the JVM and once using OraclePolish (with the preceding static analysis), and compared the execution times of these runs. Based on these measurements, we found that running the tests using OraclePolish takes between 5 and 30 times longer than running the tests using the JVM. Although this cost is significant, we believe that it is reasonable. In our experience, developers will accept high overheads for tools that produce accurate results. This is especially true when, as is the case for OraclePolish, the tools do not require any developer interaction and can be run overnight, possibly as part of an automated build system whose results are inspected the next day. In addition, OraclePolish is an unoptimized prototype. We chose to implement it as a JPF plugin because JPF is a general platform that already implements many of the capabilities we needed (e.g., the ability to associate metadata with runtime values). However, JPF's generality comes at a cost. Based on our experience with taint-based techniques, we believe that a custom implementation of OraclePolish could reduce its overhead to less than 20 %, levels that are comparable to other recent tainting-based approaches (e.g., [1, 16]), by taking advantage of several optimizations (e.g., [2, 20]).

5.6 Threats to Validity

There are several threats to the validity of our evaluation. First, we considered a limited number of tests, all of which were written in Java and used the JUnit testing framework. In addition, we filtered out tests that could not be run using JPF. Consequently, our results may not generalize beyond the considered domains. However, the tests that we considered represent a wide range of application domains, sizes, and maturity levels. Therefore, we believe that our results are promising and motivate further research. Second, we qualitatively assessed the usefulness of the error reports generated by the technique ourselves, which may introduce bias. While we are planning to conduct a human study with developers to eliminate this threat in the future, we did not believe that such a study was justified at this stage of the research.

6. RELATED WORK

To the best of our knowledge, our technique is the first technique that is able to detect both brittle assertions and unused inputs.

Schuler and Zeller propose checked coverage as an approach for assessing oracle quality [21]. The checked coverage of a test or test suite is the ratio of executed statements that compute values that are checked by the test to the total number of executed statements. A low checked coverage score suggests that a test is likely to be missing assertions. Unlike our technique, which uses dynamic tainting, checked coverage uses backward dynamic slicing to compute the set of statements that contribute to values checked by the test. While dynamic tainting and dynamic slicing are similar, dynamic tainting, due its focus on values rather than statements, provides several benefits. For example, our technique precisely identifies unused inputs while checked coverage only identifies sets of statements. Fixing the identified issues starting from sets of statements rather of inputs increases the amount of manual work that testers must perform. In addition, checked coverage shares the common limitation of all coverage based techniques: deciding how much coverage is sufficient. Obviously a checked coverage score of 0 % is bad, but what about a score of 60 %?

State coverage, originally proposed by Koster and Kao [11, 12] and extended by Vanoverberghe et al. [25], is similar to checked coverage. The primary difference is that state coverage is the ratio of executed output defining statements (ODS)—statements that define a variable that can be checked by the test suite—to the total number of ODSs. Unfortunately, there have only been small case studies on the technique's effectiveness so it is not clear how it compares to checked coverage. However, because state coverage is also a coverage metric it shares the same limitations as state coverage as compared to our technique.

The brittle assertion problem is closely related to the test dependency problem. A recent study [29] proposed an approach for detecting test dependencies in existing test suites. The authors decribed a *k-bounded dependence aware algorithm* which trims the search space for re-ordering the test methods which otherwise requires a full permutation over the test methods. The remaining sequences will be executed and checked to see if this certain ordering will alter the outcomes of some tests in the sequence. However, the search space is still so large that the authors had to limit the length of the sequence up to 2. Moreover, the detection on dependencies are limited to the ones which will unveil their presence by altering the test outcome in a certain order. Our technique presents a more precise data flow analysis that will further narrow down the search space and also be aware of the dependencies which not only cause changes of outcomes in a certain order but also lie deep in the test suite and application code such that failing the tests in the future.

In addition to techniques that attempt to improve the quality of existing oracles, there are also several techniques that attempt to automatically create oracles. Some of these techniques use mutation testing to discover how successful an oracle is at detecting mutants. For example, Staats et al. use mutation testing to support the creation or oracles by identifying the program variables are most successful at detecting mutants and therefore should be checked by an assertion [23]. Conversely, Fraser and Zeller use mutation testing to generate complete test cases, including oracles [8]. Another recent work by Loyola et al. presents an approach that supports the generation of test oracles [13]. Their technique firstly assumes that the test inputs have already been determined by the testers and then ranks variables based on the interactions and dependencies observed between them during program execution. The authors conducted an empirical study by comparing the effectiveness, defined by how many mutants killed, of the oracle data sets selected by their technique and the original oracle data sets by the testers, which shows improvement in finding faults. Other techniques create oracles from observed invariants (e.g., [17–19, 24]) or generate oracles for specific domains (e.g., GUIs [27], web pages [3, 14]).

Finally, the large number of existing approaches for generating test inputs are also related to our work (e.g., [6, 7, 9, 15, 17–19, 22, 26, 28]). However, such approaches are not alternatives to our technique. Instead they are complimentary. As we discuss in Section 7, we plan on investigating how our technique can be combined with these approaches to improve the tests that they generate.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented a new technique for automatically analyzing test oracles. The technique is based on dynamic tainting and can detect both brittle assertions—assertions that depends on values that are derived from uncontrolled inputs—and unused inputs—inputs provided by the test that are not checked by an assertion. We also presented OraclePolish, an implementation of the technique that can analyze tests that are written in Java and use the JUnit testing framework. Using OraclePolish, we conducted an empirical evaluation of the tool's performance on more than 4,000 tests from real applications. The results of the evaluation demonstrate that OraclePolish is able to detect both brittle assertions and unused inputs in real tests at a reasonable cost.

In future work, we will implement the automated generation of recomendations to fix the reported oracle problems. The possible fixes follow very regular and specific patterns so that templates will be provided for the developers. We will investigate the possibility of extending the technique to analyze entire test suites rather than individual tests. This will allow the technique to more precisely handle certain situations, such as when logically connected assertions are split among multiple test cases (e.g., the one assertion per test style). We are also planning on conducting additional evaluations of the technique. In particular, we are interested in conducting human studies with testers to qualitatively assess the technique more fully, such as the importance that developers would give to such reported issues, and increasing the number and type of subjects that we consider. Finally, we will investigate how our technique could be integrated with existing test generation approaches to improve the quality of the generated tests.

8. REFERENCES

[1] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, pages 1–20, 2011.

[2] M. Chabbi. Efficient taint analysis using multicore machines. Master's thesis, University of Arizona, 2007.

[3] S. Choudhary, H. Versee, and A. Orso. WEBDIFF: Automated identification of cross-browser issues in web applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[4] J. Clause and A. Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 249–260, 2009.

[5] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[6] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, June 2004.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[8] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 147–158, 2010.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[10] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.

[11] K. Koster. A state coverage tool for JUnit. In *Companion of the 30th International Conference on Software Engineering*, pages 965–966, 2008.

[12] K. Koster and D. C. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 541–544, 2007.

[13] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel. Dodona: automated oracle data set selection. In *Proceedings of the 14th International Symposium on Software Testing and Analysis*, pages 193–203, 2014.

[14] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 561–570, 2011.

[15] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[16] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri. SIFT: A low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, pages 37:1–37:11, 2011.

[17] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 504–527, 2005.

[18] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

[19] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 23–32, 2011.

[20] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO '08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 74–83, 2008.

[21] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 90–99, 2011.

[22] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, 2005.

[23] M. Staats, G. Gay, and M. P. E. Heimdahl. Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 870–880, 2012.

[24] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410, 2008.

[25] D. Vanoverberghe, J. de Halleux, N. Tillmann, and F. Piessens. State coverage: Software validation metrics beyond code coverage. In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science*, pages 542–553, 2012.

[26] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[27] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1), February 2007.

[28] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, 2005.

[29] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 14th International Symposium on Software Testing and Analysis*, pages 385–396, 2014.