Causal Testing: Understanding Defects' Root Causes

ABSTRACT

Isolating and repairing buggy software behavior requires finding where the bug is happening and understanding the root cause of the buggy behavior. While there exist tools that can help developers identify where a bug is located, existing work has paid little, if any, attention to helping developers understand the root cause of buggy behavior. We present Causal Testing, a new method of root-cause analysis that relies on the theory of counterfactual causality to identify a set of executions that likely hold key causal information necessary to understand and repair buggy behavior. Evaluating Causal Testing on the Defects4J benchmark, we find that Causal Testing could be applied to 71% of real-world defects, and for 77% of those, it can help developers identify the root cause of the defect. A controlled experiment with 37 developers showed that Causal Testing improved participants' ability to identify the cause of the defect: Users with standard testing tools correctly identified the cause 88% of the time in comparison to 96% of the time with Causal Testing. Overall, participants agreed Causal Testing provided useful information they could not get using tools like JUnit alone.

1 INTRODUCTION

Debugging and understanding software behavior is an important part of building software systems. To help developers debug, many existing approaches, such as spectrum-based fault localization [13, 28], aim to automatically localize bugs to a specific location in the code. However, finding the relevant line is often not enough to help fix the bug [35]. Instead, developers need help identifying and understanding the root cause of buggy behavior. While techniques such as delta debugging can minimize a failing test input [47] and a set of test-breaking changes [46], they do not help explain *why* the code is faulty [27].

To address this shortcoming of modern debugging tools, this paper presents *Causal Testing*, a novel method for identifying root causes of failing executions based on the theory of counterfactual causality. While prior approaches have applied statistical causal inference to observational, non-experimental data to help locate bugs [5, 6], they have not aimed to help *explain* root causes of the behavior and have not used experimentally-driven counterfactual analysis. Causal Testing does just that by taking a manipulationist approach to causal inference [44], modifying and executing tests to observe causal relationships and derive causal claims about the defects' root causes.

Given one or more failing executions, Causal Testing conducts *causal experiments* by modifying the existing tests to produce a small set of executions that differ minimally from the failing ones but do not exhibit the faulty behavior. By observing a behavior and then purposefully changing the input to observe the behavioral changes, Causal Testing infers causal relationships [44]: The change in the input *causes* the behavioral change. Causal Testing looks for two kinds of minimally different executions, ones whose inputs are similar and ones whose execution paths are similar. When the differences between executions, either in the inputs or in the execution paths, are small, but exhibit different test behavior, these small, causal differences can help developers understand what is causing the faulty behavior.

Consider a developer working on a web-based geo-mapping service (such as Google Maps or MapQuest) receiving a bug report that the directions between "New York, NY, USA" and "900 René Lévesque Blvd. W Montreal, QC, Canada" are wrong. The developer replicates the faulty behavior and hypothesizes potential causes. Maybe the special characters in "René Lévesque" caused a problem. Maybe the first address being a city and the second a specific building caused a mismatch in internal data types. Maybe the route is too long and the service's precomputing of some routes is causing the problem. Maybe construction on the Tappan Zee Bridge along the route has created flawed route information in the database. There are many possible causes to consider. The developer decides to step through the faulty execution, but the shortest path algorithm coupled with precomputed-route caching and many optimizations is complex, and it is not clear how the wrong route is produced. The developer gets lost inside the many libraries and cache calls, and the stack trace quickly becomes unmanageable.

Suppose, instead, a tool had analyzed the bug report's test and presented the developer with the following information:

```
1  Failing: New York, NY, USA to
          900 René Lévesque Blvd. W Montreal, QC, Canada
2  Failing: Boston, MA, USA to
          900 René Lévesque Blvd. W Montreal, QC, Canada
3  Failing: New York, NY, USA to
          1 Harbour Square, Toronto, ON, Canada
4  Passing: New York, NY, USA to
          39 Dalton St, Boston, MA, USA
5  Passing: Toronto, ON, Canada to
          900 René Lévesque Blvd. W Montreal, QC, Canada
6  Passing: Vancouver, BC, Canada to
          900 René Lévesque Blvd. W Montreal, QC, Canada
```

Minimally different execution traces:

```
7   Failing:                          Passing:
8   [...]                             [...]
9   findSubEndPoints(sor6, tar7);     findSubEndPoints(sor6, tar7);
10  findSubEndPoints(sor7, tar8);     findSubEndPoints(sor7, tar8);
11  metricConvert(pathSoFar);
12  findSubEndPoints(sor8, tar9);     findSubEndPoints(sor8, tar9);
13  [...]                             [...]
```

The developer would quickly see that the special characters, the first address being a city, the length of the route, and the construction are not the root cause of the problem. Instead, all the failing test cases have one address in the United States and the other in Canada, whereas all the passing test cases have both the starting and ending addresses in the same country. Further, the tool found a passing and a failing input with minimal execution trace differences: the failing execution contains a call to the `metricConvert(pathSoFar)` method but the passing one does not.[1] Armed with this information, the developer is now better equipped to find and edit code to address the root cause of the bug.

[1] Note that prior work, such as spectrum-based fault localization [13, 28], can identify the differences in the traces of existing tests; the key contribution of the tool we describe here is generating the relevant executions with the goal of minimizing input and execution trace differences.

We evaluate Causal Testing in two ways. First, we use a proof-of-concept implementation of Causal Testing, Holmes, to evaluate Causal Testing in a controlled experiment. We asked 37 developers to identify the root causes of real-world defects, with and without access to Holmes. We found that when using Holmes, developers could identify the root cause 96% of the time, while those unable to use Holmes could only do so 88% of the time. While the difference is not statistically significant ($p = 0.17$, Fisher's exact test), this experiment provides promising evidence that Causal Testing, and tools like Holmes, can be helpful for developers.

Second, we evaluate Causal Testing's applicability to real-world defects by considering defects from real-world programs, collected in the Defects4J benchmark [32]. We found that Causal Testing could be applied to 71% of real-world defects, and for 77% of those, it could help developers identify the root cause.

The rest of this paper is structured as follows. Section 2 illustrates how Causal Testing can help developers on a real-world defect. Section 3 details Causal Testing and Section 4 describes Holmes, our prototype implementation. Section 5 evaluates how useful Holmes is in identifying root causes and Section 6 evaluates how applicable Causal Testing is to real-world defects. Finally, Section 7 identifies threats to the validity of our studies, Section 8 discusses our findings, Section 9 places our work in the context of related research, and Section 10 summarizes our contributions.

2 MOTIVATING EXAMPLE

Consider Amaya, a developer who regularly contributes to open source projects. Amaya codes primarily in Java and regularly uses the Eclipse IDE and JUnit. Amaya is working on addressing a bug report in the Apache Commons Lang project. The report comes with a failing test (see ① in Figure 1).

Figure 1 shows Amaya's IDE as she works on this bug. Amaya runs the test to reproduce the error and JUnit reports that an exception occurred while trying to create the number `0Xfade` (see ② in Figure 1). Amaya looks through the JUnit failure trace, looking for the place the code threw the exception (see ③ Figure 1). Amaya observes that the exception comes from within a `switch` statment, and that there is no case for the `e` at the end of `0Xfade`. To add such a case, Amaya examines the other `switch` cases and realizes that each case is making a different kind of number, e.g., the case for `l` creates a `long` or `BigInteger`. Since `0Xfade` is 64222, Amaya conjectures that since this number fits in an `int`, she creates a new method call to `createInteger()` inside of the case for `e`. Unfortunately, the test still fails.

Using the debugger to step through the test's execution, Amaya sees the `NumberFormatException` thrown on line 545 (see ③ in Figure 1). She sees that there are two other locations the input touches (see ④ and ⑤ in Figure 1) during execution that could be affecting the outcome. She now realizes that the code on lines 497–545, despite being where the exception was thrown, may not be the location of the defect's cause. She is feeling stuck.

But then, Amaya remembers a friend telling her about Holmes, a Causal Testing Eclipse plug-in that helps developers debug. Holmes tells her that the code fails on the input `0Xfade`, but passes on input `0xfade`. The key difference is the lower case `x`. Also, according to the execution trace provided by Holmes, these input differ in the execution of line 458 (see ④ in Figure 1).

The `if` statement fails to check for the `0x` prefix. Now, armed with the cause of the defect, Amaya turns to the Internet to find out the hexadecimal specification and learns that the test is right, `0x` and `0x` are both valid prefixes for hexadecimal numbers. She augments the `if` statement and the bug is resolved!

Holmes implements Causal Testing, a new technique for helping understand root causes of behavior. Holmes takes a failing test case (or test cases) and perturbs its inputs to generate a pool of possible inputs. For example, Holmes may perturb `0Xfade` to `0XFADE`, `0xfade`, `edafX`, `0Xfad`, `Xfade`, `fade`, and many more. Holmes then executes all these inputs to find those that pass the original test's oracle, and, next, selects from the passing test cases a small number such that either their inputs or their execution traces are the most similar to the original, failing test case. Those most-similar passing test cases help the developer understand the key input difference that makes the test pass. Sometimes, Holmes may find other failing test cases whose inputs are even more similar to the passing ones than the original input, and it would report those too. The idea is to show the smallest difference that causes the behavior to change.

Holmes presents both the static (test input) and dynamic (execution trace) information to the developer to compare the minimally-different passing and failing executions to better understand the root cause of the bug. For example, for this bug, Holmes shows the inputs, `0XFADE` and `0xFADE`, and the traces of the two executions, showing that the passing test enters a method from `createInteger` that the failing test cases do not, dictating to Amaya the expected code behavior, leading her to fix the bug.

3 CAUSAL TESTING

Amaya's debugging experience is based on what actual developers did while debugging real defects in a real-world version of Apache Commons Lang (taken from the Defects4J benchmark [32]). As the example illustrates, software is complex and identifying root causes of program failures is challenging. This section describes our Causal Testing approach to computing and presenting developers with information that can help identify failures' root causes.

Figure 2 describes the Causal Testing approach. Given a failing test, Causal Testing conducts a series of causal experiments starting with the original test suite. Experimental results are provided to developers in the form of minimally different passing and failing tests and traces of their executions.

3.1 Causal Experiments with Test Cases

Causal Testing conducts causal experiments using test cases and observes then reports behavior changes. A causal experiment in the context of Causal Testing involves perturbing test input, running the original failing test with the perturbed input to determine if the input yields faulty behavior, and determining if the input is "close" to the original. Once the experiments are complete, Causal Testing provides the developer with a list of minimally different passing and failing tests along with execution traces, all of which help explain the cause the failure.

3.1.1 Perturbing Test Inputs. To conduct causal experiments, we need ways of manipulating test inputs. Causal Testing uses existing tests in the test class and test case generation to find inputs on which the original failing test passes. Causal Testing then perturbs

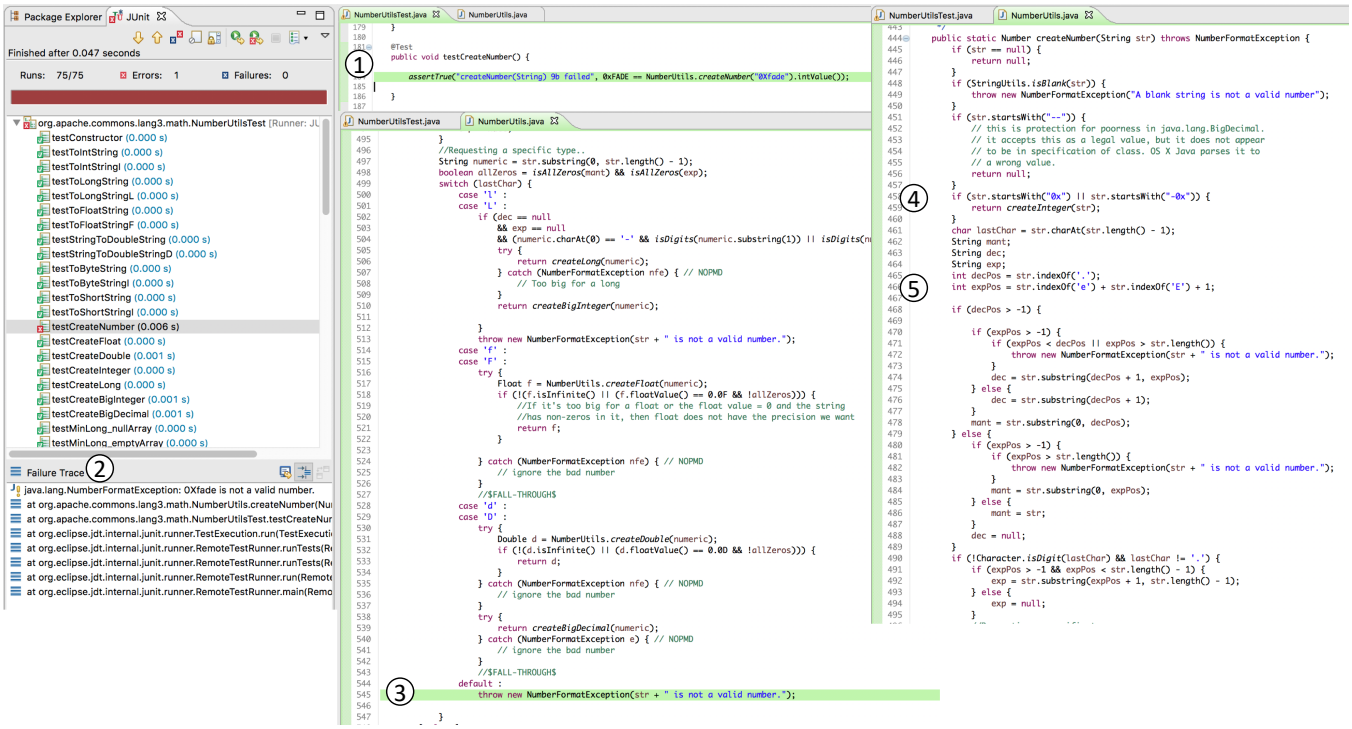

Figure 1: Amaya's Eclipse IDE, while she is debugging.

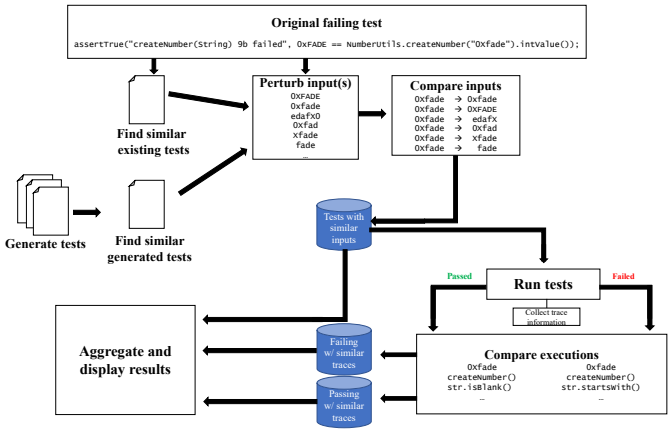

Figure 2: Causal Testing computes minimally different tests that, nevertheless, produce different behavior.

existing and generated test inputs using a form of fuzzing to find additional tests that exhibit expected and unexpected behavior.

Note that it is also possible to perturb a test oracle. For example, we might change the `assertTrue` in Figure 2 to `assertFalse`. However, perturbing test oracles is unlikely to produce meaningful information to guide the developer to the root cause of the bug because a passing test with a perturbed oracle would be passing because the expectation of the behavior changed, not because of a causal experiment showing a change in the input *causing* a change in the behavior. Instead, Causal Testing focuses on perturbing test inputs. For example, the input `0Xfade` can be perturbed as shown in Figure 2.

Perturbing test inputs can be done in three different ways. First, Causal Testing could simply rely on the tests already in the test suite, finding the most similar passing test case. Second, Causal Testing could use automated test generation to generate a large number of test inputs [1, 16, 34] and then filter them to select only those that are close to the original input. Third, we could use test fuzzing to change the original input in small ways to generate inputs only similar to the original input. Fuzz testing is an active research area [17, 18, 23, 29, 41] (although the term fuzz testing is also used to mean simply generating tests [1]). Fuzz testing has been applied in the security domain to stress-test an application and automatically discover vulnerabilities, e.g., [18, 23, 41].

Unsurprisingly, using existing tests alone is likely to often not produce a similar-enough test to articulate the root cause. Still, it is worthwhile to consider these tests first, before trying to generate more. Meanwhile, our research led us to the realization that existing fuzz testing techniques, for the most part, generate random inputs. As such, our solution to the challenge of generating similar inputs is to (1) use multiple fuzzers, (2) generate many tests, and (3) filter those tests to select those similar to the original test. As we observed with Holmes, our proof-of-concept Causal Testing tool (described in Section 4), using multiple input fuzzers provided a different set of perturbations, increasing the chances that Holmes found a set of minimally different inputs and that at least one of which would lead to a non-failing execution.

3.1.2 Input Similarity. Given two tests with different inputs (but the same oracle), Causal Testing needs to be able to determine how similar they are. Conceptually, to apply the theory of causal inference, the idea behind test input similarity is that two minimally-different test inputs should differ in only one factor. For example, imagine a software system that processes apartment rental applications. If two application inputs are identical in every way except one entry, and the software crashes on one but not on the other, this pair of inputs provides one piece of evidence that the differing entry *causes* the software to crash. (Other pairs that also only differ in that one entry would provide more such evidence.) If the inputs differed in multiple entries, it would be harder to know which entry is responsible. Thus, to help developers understand root causes, Causal Testing should be able to precisely measure input similarity. We propose two ways to evaluate input similarity: *syntactic differences* and *execution path differences*.

Syntactic Input Differences. Input syntactic similarity can be viewed at different scopes. First, inputs can consist of multiple arguments and thus can agree in some and differ in others of those arguments. Agreement across more arguments makes inputs more similar. Second, each argument whose values in the two tests differ can differ to varying degrees. A measure of that difference depends on the type of the argument. For arguments of type `String`, the Levenshtein distance (the minimum number of single-character edits required to change one `String` into the other) is a reasonable measure, though there are others as well, such as Hamming distance (difference between two values at the bit level). For numerical arguments, their numerical difference or ratio is often a reasonable measure.

The specific semantics of the similarity measure are domain specific. For example, in apartment rental applications, a difference in the address may play a much smaller role than a difference in salary or credit history. How the similarity of each argument is measured, and how the similarities of the different arguments are weighed are specific to the domain and may require fine tuning by the developer, especially for custom data types (e.g., project-specific `Object` types). However, we found that relatively simple measures suffice for general debugging, and likely work well in many domains. Using Levenshtein or Hamming distance for `String`s, differences for numerical values, sums of elements distances for `Array`s works reasonably well, in practice, in the 330 defects from four different real-world systems we examined from the Defects4J benchmark [32]. In the end, while our measures worked well for us, there is no single right answer for measuring input similarity, and further empirical research is needed to evaluate different measures.

Execution Path Differences. Along with syntactic differences, two inputs can differ based on their execution paths at runtime. One challenge when considering only syntactic input differences is that a syntactically similar input may not always yield an outcome that is relevant to the original execution. For example, it is possible that in some cases we will find an input that is within our threshold of similarity but that those changes lead to a completely different execution path. Therefore, Causal Testing also collects and compares input execution paths and uses this information when determining which inputs are closest to the original.

There are various ways that execution differences can be calculated, but primarily depends on the information collected during execution. The naive way to compare two executions based on their execution path would be to compare the number of statements that got executed with each input. Though this can provide some insights, two inputs can take two different paths that execute the same number of statements. To improve on this, we can compare exactly what statements or method calls are executed. This also strengthens the causal connection between the input change and the behavior change; if two inputs' execution, one passing and one failing, only differ by one statement, it is reasonable to assume that one statement plays some role in the behavior change. Another option similar to this would be to compare the return values of method calls. This method could provide insights in situations where the bug is not in the sequence of statements executed but in the usage of a method that returns an unexpected value (e.g., the wrong method call is being used for a given input or kind of input).

Both syntax and execution path can be useful in identifying relevant, useful tests. However, an even stronger argument for causality can be made when a given input is similar syntactically and in terms of execution paths. Therefore, when possible, Causal Testing prioritizes tests whose inputs that are syntactically and dynamically close to the original.

3.2 Presenting Experimental Results

After generating and executing test inputs, Causal Testing ranks them by similarity and present the tests that are most similar to the original. This process continues until either a fixed target number of passing tests is found (in our experience, 3 was a good target), or a time-out is reached. This approach enables Causal Testing to produce results for the developer as quickly as possible, while it performs more computation, looking for potentially more results.

With each test execution, Causal Testing collects the inputs used and the execution trace information. This trace includes methods that execute during the test's run, the arguments each method executed with, call sites for the methods, and methods' return values. Execution traces can get large, making parsing them difficult. However, the trace differences between the two similar tests should be small, and thus Causal Testing displays a minimized trace, focused on differences in the traces.

4 HOLMES: A CAUSAL TESTING IMPLEMENTATION

We have implemented Holmes, an open source Eclipse plug-in Causal Testing prototype.[2] Holmes consists of *input and test case generators*, *edit distance calculators & comparers*, *test executor & comparator*, and *output view*.

4.1 Input & Test Case Generation

The first thing Holmes does is attempt to find and generate additional similar tests from the existing test suite. Before generating any test, Holmes searches all tests in the current test suite for tests that are similar to the original failing test. Currently, Holmes uses string matching to determine if two tests match. More specifically, the test file is converted to a string and parsed line by line for tests that test the same functionality on similar inputs. One plan for

[2]The link to Holmes has been removed for blind review.

future work to improve Holmes is to increase accuracy and utility of the similar tests found by basing the comparison on execution paths rather than just input/test similarity.

If similar passing tests are found in the developer provided test suite, Holmes stores both the test for presentation, along with the input to the test. If after searching existing tests no similar passing tests are found, Holmes proceeds to generate new tests. Holmes gets new inputs for generating new tests in two ways:

- **Test case generation.** One way to get new tests and test inputs is to use existing test case generation tools. Holmes currently uses Evosuite [16] for this step. We chose Evosuite because it a state-of-the-art open source tool that works with Java and JUnit. Holmes determines the target class to generate tests from based on the name of the test class. For example, if the test class is called `NumberUtilsTest`, Holmes tells Evosuite to generate tests for `NumberUtils`. To determine if a test is related to the original failure, Holmes searches the generate tests for test cases that call the same method as the original test. From this process, Holmes will get at least one valid input to use during fuzzing, which we describe next.
- **Input fuzzing.** To generate additional inputs for new tests, Holmes fuzzes existing test inputs that have been gathered for additional valid test inputs. Currently, Holmes uses two off-the-shelf, open source fuzzers: Peach[3] and Fuzzer[4]. To increase the chances that fuzzed inputs will produce passing tests, Holmes prioritizes (when available) inputs from passing tests. Holmes also fuzzes the original input and all valid inputs from generated test cases, again to heighten the chance that passing tests will be found.

Once Holmes runs test generation and fuzzes the valid inputs, the next step is to determine which of the generated inputs are closest to the original. This is done by calculating the edit distance between inputs pairs, which we describe next.

4.2 Test Execution & Edit Distance Calculation

Causal Testing uses syntactic and execution differences should be considered when conducting causal experiments. The current version of Holmes automatically finds and determines similar passing and failing test based on syntactic similarity. Before attempting to integrate and modify tooling for tracing executions, we needed a better understanding of how execution information could be collected and used most effectively. However, we wanted to evaluate our technique in its entirety. Therefore, we semi-automated the execution trace portion of Holmes for the user study.

Holmes first considers differences in the inputs themselves (syntactic difference). To evaluate syntactic differences, Holmes first determines the data type of each parameter in the method-under-test; this determines how Holmes will calculate edit distance. For parameters with numerical values, Holmes calculates the edit distance by calculating the absolute value of the numerical difference between the original and generated test input. For example, inputs 1.0 and 4.0 have an edit distance of three. For string or character inputs, Holmes uses two different metrics. First, Holmes determines the hamming distance between the original and generated string. We use hamming distance, which works at the bit level rather than the character level, first to increase accuracy in the similarity of randomly generated inputs from test generation and fuzzing. Once Holmes has as set of truly close inputs, Holmes uses Levenshtein distance to determine which of these close inputs requires the fewest character changes to get back to the original. Currently, the edit distance threshold in Holmes is set to three; this means that any tests that Holmes reports is within an edit distance of three from the original test value(s).

Once a set of similar inputs have been found, Holmes executes each test to determine which inputs exhibit expected behavior. For each original parameter, Holmes iteratively replaces the original value with new input values and executes the test while observing the outcome. Although Holmes may also find additional failing tests, Holmes iterates on this process to try and provide developers with at least three similar passing tests to compare to the failing ones (or times out trying). Once Holmes is finished, developers can view Holmes' causal execution information and similar tests found as a result of the causal experiments.

For each test executed by Holmes, we manually executed the test using InTrace [24] to collect method calls and return values. We then manually minimized the traces to help determine the output provided by Holmes in our user study. We also hypothesize that execution information can be useful to the developer when debugging, therefore we included the minimized traces in the output provided to participants. In Section 5, we discuss if and how developers used trace information and whether they found value in this additional piece of information provided by Causal Testing.

4.3 Reporting Results

An important consideration when building a tool is how it will communicate with the developer [26]. Therefore we wanted to make sure we put thought into how the tool presents information to its user. Once Holmes is done with all calculations and has a set of passing (and probably failing) tests, Holmes organizes the information for presentation. First, Holmes goes through each generated test and bolds the input(s) that were generated during execution. Tests are organized by passing and failing, with the original test case at the top of the output window for the ability to easily compare without going back and forth between the code to find it. Second, under each test, Holmes presents the minimized test execution trace. So to not overwhelm the developer with information, Holmes' output includes the option to toggle showing and hiding trace information.

5 IDENTIFYING ROOT CAUSES WITH CAUSAL TESTING

We designed a controlled experiment with 37 developers to answer the following research questions:

- RQ1 Does Causal Testing improve the developers' ability to identify the root causes of defects?
- RQ2 Does Causal Testing improve the developers' ability to repair defects?
- RQ3 Do developers find Causal Testing useful, and, if so, what aspect of Causal Testing is most useful?

[3] https://github.com/MozillaSecurity/peach
[4] https://github.com/mapbox/fuzzer

5.1 User Study Design

The goal of our research is to help developers determine the cause of a test failure, thereby helping developers better understand and eliminate defects from their code. We designed our user study, and prototype version of Holmes, to provide evidence of a need for Causal Testing while also providing a foundation regarding what information is important and useful for Causal Testing.

We randomly selected seven defects from Defects4J, from the Apache Commons Lang project. We chose Apache Commons Lang because it (1) is the most widely known project in Defects4J, (2) had defects that required only limited domain knowledge, and (3) can be developed in Eclipse.

Our user study consisted of a training task and six experimental tasks. Each task mapped to one of the seven defects selected. To reduce the effects of ordering bias, we randomized defect order across participants. Each participant completed three tasks in the control group and three tasks in the experimental group.

For the training task, we provided an Eclipse project with a defective code version and single failing test. We explained how to execute the test suite via JUnit, and how to invoke Holmes. We allowed participants to explore the code and ask questions, telling them that the goal is for all to tests pass. For each task that followed were similar to the training task; control tasks did not have access to Holmes, experimental group tasks did.

We recorded audio and the screen for later analysis. We asked participants to complete a causality questionnaire after each task consisting of two questions: "What caused Test XX to fail?" and "What changes did you make to fix it?"

At then end, the participants completed a post-evaluation survey where we asked them open-ended questions, such as *What information did you find most helpful when determining what caused tests to fail?*, and 4-point Likert scale questions, such as *How useful did you find X?*. For Likert-scale questions, we gave participants the options "Very useful", "Somewhat useful", "Not useful", and "Misleading or harmful". We also gave participants an opportunity to provide any additional feedback they saw fit.

We conducted a pilot of our initial user study design with 23 students from a graduate software engineering course. Our pilot study consisted of 5 tasks and a mock-up version of Holmes. We used lessons learned and challenged encountered to finalize the design of our study. All final study materials are available online.[5]

5.2 Participants

We recruited a total of 39 participants from industry and academia: 15 undergraduate students, 12 PhD students, 9 Masters students, 2 industry developers, and 1 research scientist. Participants' programming experience ranged from 1 to 30 years and experience with Java ranged from a few months to 15 years. All participants reported having prior experience with Eclipse and/or JUnit. We analyzed data from 37 participants; 2 undergraduate participants did not follow instructions so we removed them from our dataset.

[5]Link to study materials removed for blind review.

Defect	Group	Correct	Incorrect	Total
1	Control	17 (89%)	2 (11%)	19
	Holmes	17 (95%)	1 (5%)	18
2	Control	12 (60%)	8 (40%)	20
	Holmes	9 (53%)	8 (47%)	17
3	Control	19 (95%)	1 (5%)	20
	Holmes	16 (94%)	1 (6%)	17
4	Control	15 (83%)	3 (17%)	18
	Holmes	18 (95%)	1 (5%)	19
5	Control	13 (87%)	2 (13%)	15
	Holmes	21 (95%)	1 (5%)	22
6	Control	12 (67%)	6 (33%)	18
	Holmes	15 (79%)	4 (21%)	19
Total	Control	**88 (80%)**	**22 (20%)**	110
	Holmes	**96 (86%)**	**16 (14%)**	112

Figure 3: Distribution of correct and incorrect cause descriptions, per defect.

5.3 User Study Findings

RQ1: Root Cause Identification

The primary goal of Causal Testing is to help developers identify the root cause of test failures. To answer RQ1, we analyzed the responses participants gave to the question "What caused Test XX to fail?" We marked responses as either correct (captured full and true cause) or incorrect (missing part of true cause).

Figure 3 shows the root cause identification correctness results, with the total for each category in bold at the bottom. We analyzed data for all but two participants (P2 and P3); these two participants did not use Holmes at any point in time during their session, therefore we felt it best to exclude them from the analysis. When using Holmes, developers correctly identified the cause 86% of the time (96 out of 112), but the control group only identified the cause 80% of the time (88 out of 110, $p = 0.17$, Fisher's exact test).

For four of the six defects, (defects 1, 4, 5, and 6), developers using Holmes were more accurate when identifying root causes than the control group. For defects 1, 4, and 5, participants only incorrectly identified the cause approximately 5% of the time when using Holmes, compared to 11–17% of the time without Holmes. For defect 6, participants identified the correct cause 79% of the time; without Holmes they could only identify the correct cause 67% of the time. Our findings suggest that **Causal Testing supports and improves developer ability to understand root causes.**

RQ2: Defect Repair

The secondary goal of Holmes is to improve the developers' ability to repair defects (though we note that Holmes provides information on the root cause of the defect, which, while likely useful for repairing the defect, is a step removed from that task). To answer RQ2, we analyzed participants' responses to the question "What changes did you make to fix the code?" We used the same evaluation criteria and labeling as for RQ1. To determine if causal execution information improves developers' ability to debug and

	Average Resolution Time (in minutes)					
Defect:	**1**	**2**	**3**	**4**	**5**	**6**
Control	16.5	10.6	6.8	12.9	3.7	10.0
Holmes	17.0	12.7	6.4	17.7	4.9	10.1

Figure 4: The average time developers took to resolve the defects, in minutes.

Defect	Group	Correct	Incorrect	Total
1	Control	16 (100%)	2 (11%)	16
	Holmes	12 (86%)	2 (14%)	14
2	Control	12 (100%)	0 (0%)	12
	Holmes	7 (100%)	0 (0%)	7
3	Control	19 (100%)	0 (0%)	19
	Holmes	16 (100%)	0 (0%)	16
4	Control	15 (100%)	0 (0%)	185
	Holmes	19 (100%)	0 (0%)	19
5	Control	12 (86%)	2 (14%)	14
	Holmes	21 (95%)	1 (5%)	22
6	Control	6 (75%)	2 (25%)	8
	Holmes	5 (100%)	0 (50%)	5
Total	Control	**80 (93%)**	**6 (7%)**	86
	Holmes	**80 (96%)**	**3 (4%)**	83

Figure 5: Distribution of correct and incorrect repairs implemented by participants, per defect.

repair defects, we observed the time it took participants to complete each task and the correctness of their repairs.

Figure 4 shows the average time it took developers to repair each defect. We omitted times for flawed repair attempts that do not address the defect. On average, participants took more time with Holmes on all but one defect (Defect 3). One explanation for this observation is that while Holmes helps developers understand the root cause, this does not lead to faster repair. Repairing the defect depends on factors other than understanding the root cause, such as experience in the necessary domain(s) [26], which can introduce noise in the time measurements.

Figure 5 shows repair correctness results (omitting several repairs that were either incomplete or made the tests pass without fixing the underlying defect). When using Holmes, developers correctly repaired the defect 96% of the time (80 out of 83) while the control group repaired the defect 95% of the time (80 out of 84).

For two of the six defects, Defects 5 and 6, developers using Holmes repaired the defect correctly more often (Defect 5: 95% vs. 85%; Defect 6: 100% vs. 75%). For Defects 2, 3, and 4, developers repaired the defect correctly 100% of the time both with and without Holmes. For one defect (Defect 1), developers were only able to repair the defect correctly 85% (12 out of 14) of the time while developers without Holmes correctly fixed defects 100% of the time.

Over all defect repairs, only defects repaired in the control group, aside from Defects 1 and 5, resulted in an incorrect fix. Still, there was no meaningful trend. Again, as repairing the defect depends on factors other than understanding the root cause, Holmes did not demonstrate an observable advantage when repairing defects. Our findings suggest that **Causal Testing sometimes helped developers repair defects, but neither consistently nor statistically significantly.**

RQ3: Usefulness of Causal Testing

To answer RQ3, we analyzed post-evaluation survey responses on information most useful when understanding and debugging the defects. We extracted and aggregated quantitative and qualitative results regarding information most helpful when determining the cause of and fixing defects. We also analyzed the Likert-scale ratings regarding the usefulness of JUnit and the various components of causal execution information.

Quantitative Results. Overall, participants found the information provided by Holmes more useful than other information available when understanding and debugging the defects. Out of 37 participants, 17 (46%) found the addition of at least one aspect of Holmes more useful than output provided by JUnit alone. Fifteen (41%) participants found the addition of Holmes at least as useful as JUnit alone. The remaining five (13%) found the addition of Holmes not as useful as JUnit alone. Though majority of participants found Holmes' output more useful, JUnit and interactive debuggers are an important part of debugging. Therefore, our expectations would be that Causal Testing would augment those tools, not replace them.

Of the information provided by Holmes, our findings suggest participants found the minimally different passing tests the most useful; 20 out of 37 participants (54%) rated this piece of information as "Very Useful". The passing and failing test inputs that Holmes provided received "Very Useful" or "Useful" rankings more often than the test execution traces. Of the different pieces of information provided by Holmes, participants found the execution traces the least useful; 18 marked either the passing or failing execution trace as "Not Useful". One participant felt the passing test traces were "Misleading or Harmful"; during their session, they noted that they felt in some cases the execution paths were not as similar as others which made interpreting the output more confusing.

Qualitative Results. To gain a better understanding of what parts of causal execution information are most useful, and why, we also analyzed participants' qualitative responses to the questions asked in our post-evaluation questionnaire.

What information did you find most helpful when determining what caused tests to fail?

Twenty-one participants explicitly mentioned some aspect of Holmes as being most helpful. For six of these participants, all the information provided by Holmes was most helpful for cause identification. Another eight participants noted that specifically the similar passing and failing tests were most helpful of the information provided by Holmes. For example, P36's stated these similar tests when presented "*side by side*" made it "*easy to catch a bug.*"

The other six participants stated the execution traces were most helpful. One participant's response said that the parts of Holmes output that were most helpful was the output *"showing method calls, parameters, and return values."* This was particularly true when there were multiple method calls in an execution according to P26 as "*it was useful to see what was being passed to them and what they*

were returning."

What information did you find most helpful when deciding changes to make to the code?
Fourteen participants mentioned some aspect of Holmes as being most helpful. Of the 14 participants that mentioned Holmes, five explicitly stated that the similar passing tests specifically were most helpful of the information provided by Holmes. P7, who often manually modified failing tests to better understand expected behavior noted "*it helped to see what tests were passing*," which helped him "*see what was actually expected and valid*."

For the other four, the execution trace was most helpful for resolving the defect. One participant in this group specifically mentioned that the return values in the execution traces for passing and failing inputs were most helpful because then he could tell "*which parts are wrong*."

Would you like to add any additional feedback to supplement your responses?
Many participants used this question as an opportunity to share why they thought Holmes was useful. Many reported comments such as "Holmes is great!" and "really helpful." For many of these participants, Holmes was most useful because it provided concrete, working examples of expected and non-expected behavior that help with "*pinpointing the cause of the bug*".

One participant noted that without Holmes, they felt like it was *"a bit slower to find the reason why the test failed."* Another participant noted that the trace provided by Holmes was "somewhat more useful" than the trace provided by JUnit.

In free-form, non-prompted comments through-out the study, participants often mentioned that the passing and failing tests and traces were useful for their tasks; several participants explicitly mentioned during their session that having the additional passing and failing tests were "super useful" and saved them time and effort in understanding and debugging the defect.

Despite all the positive comments and feedback, as we mentioned, we do not intend for Causal Testing tools to replace tools like JUnit but for them to work together to help developers understand and fix bugs. Three participants explicitly mentioned that Holmes is most useful in conjunction with JUnit and other tools available in the IDE. The complementary nature of these tools were highlighted in the comments from some participants. For example, in his response P26 explained that though Holmes was "*very useful when debugging the code*," it is most useful with other debugging tools as "*it does not provide all information*."

Participants also had suggestions for how we could improve Holmes. One participant mentioned in their post-evaluation that Holmes should add the ability to click the output and jump to the related lines of code in the IDE. One suggestion for improvement was to make the differences between the passing and failing tests more explicit. During their sessions, three participants explicitly suggested, rather than bolding the entire fuzzed input, only bolding the character(s) that are different from the original. Our findings suggest that **Causal Testing is useful for both cause identification and defect resolution.**

6 CAUSAL TESTING APPLICABILITY TO REAL-WORLD DEFECTS

To evaluate the usefulness and applicability of Causal Testing to real-world defects, we conducted an evaluation on the Defects4J benchmark [32]. Defects4J is a collection of reproducible defects found in real-world, open-source Java software projects: Apache Commons Lang, Apache Commons Math, Closure compiler, JFreeChart, and Joda-Time. For each defect, Defects4J provides a defective and a repaired version of the source code, along with the developer-written test suites that exhibit the defect.

We manually examined 330 defects in four of the five projects in the Defects4J benchmark and categorized them based on whether Causal Testing would work and whether it would be useful in identifying the root cause of the defect. We excluded Joda-Time from our analysis because of difficulty reproducing the defects.[6]

6.1 Evaluation Process

To determine applicability of Causal Testing to defects in the Defects4J benchmark, we first imported the buggy and fixed project versions into Eclipse. Next, we executed the defective project's test suite to identify the failing test(s) and methods it tested. Once found, we determined the input(s) to the test(s). We then perturbed test inputs to produce reasonably similar passing tests that likely capture the defect's root cause. When possible, we used Holmes to produce tests; otherwise we manually perturbed test inputs to find similar passing tests. We examined the developer-written patch for the failing test to help understand the root cause and determine whether Causal Testing would help guide a developer to fix it. From this process, five defect categories emerged in regards to the applicability of Causal Testing.

6.2 Defect Applicability Categories

We categorized each defect into the following categories:

I. **Works, useful, and fast.** For these defects, Causal Testing can produce at least one minimally different passing test that captures its root cause. We reason Causal Testing would be helpful to developers. In our estimate, the difference between the failing and minimally different passing tests is reasonably small that it can be found on a reasonable personal computer, reasonably fast.

II. **Works, useful, but slow.** For these defects, Causal Testing again can produce at least one minimally different passing test that captures its root cause, and this would be helpful to developers. However, the difference between the tests is large, and, in our estimation, Causal Testing would need additional computation resources, such as running overnight or access to cloud computing.

III. **Produces minimally different passing test, but is not useful.** For these defects, Causal Testing again can produce at least one minimally different passing test, but in our estimation, this test would not be useful to understanding the root cause of the defect.

[6] Some difficulties have been documented in the Joda-Time issue tracker: https://github.com/dlew/joda-time-android/issues/37.

Project	Applicability Category I	II	III	IV	V	Total
Math	14	15	11	20	46	106
Lang	11	6	3	14	31	65
Chart	2	4	1	1	18	26
Closure	2	22	8	5	96	133
Total	29	47	23	40	191	330

Figure 6: Distribution of defects across five applicability categories described in Section 6.2.

IV. **Will not work.** For these defects, Causal Testing would not be able to perturb the tests, and would tell the developer it cannot help right away.
V. **We could not make a determination.** Understanding the root cause of these defects required project domain knowledge that we lacked, so we opted not to make an estimation of whether Causal Testing would work.

One important aspect of both our user study and applicability evaluation is the notion that Causal Testing could be useful for understanding a defects' root cause. In most cases, we were able to make a determination on potential usefulness of Causal Testing for a given defect. Because defects in our study are from real-world projects, many defects required project-specific domain knowledge to understand. As we are not the original projects' developers, for some defects, the lack of domain-specific knowledge prevented us from understanding what information would help developers debug, and we elected not to speculate.

6.3 Results

Figure 6 shows our defect classification results. Of the 330 defects, we could make a determination for 139 of them. Of these, Causal Testing could promise the developer to try to help for 99 (71%). For the remaining 40 (29%), Causal Testing would simply say it cannot help and would not waste the developer's time. Of these 99 defects, for 76 (77%), Causal Testing can produce information helpful in identifying the root cause. For 29 (29%), a simple local IDE-based tool would work, and for 47 (47%), a tool would need more substantial resources, such as running overnight or on the cloud. The remaining 23 (23%) would not benefit from Causal Testing. Our findings suggest that **Causal Testing produce results for 71% of real-world defects, and for 77% of those, it can help developers identify the root cause of the defect.**

7 THREATS TO VALIDITY

External Validity. Our studies used Defects4J defects, which may not generalize. We mitigated this threat by using a well-known and widely-used benchmark of real-world defects. We selected defects for the user study randomly from those that worked with our current implementation of Holmes and that required little or no prior project or domain knowledge, with varying levels of difficulty. The applicability evaluation considered defects across four projects.

The user study used 37 participants, which is within range of higher data confidence and is above average for similar user studies [7, 15, 33, 37]. We mitigated this threat by finding participants with different backgrounds and experience.

Internal Validity. Our user study participants were volunteers. This leads to the potential for self-selection bias. We were able to recruit a diverse set of participants, somewhat mitigating this threat.
Construct Validity. We performed a manual analysis of whether Causal Testing would apply and be useful. This leads to the potential for researcher bias. We minimized the potential for this threat by developing and following concrete, reproducible methodology and criteria for usefulness.

The user study asked participants to understand and debug code they had not written. We mitigate this threat by selecting defects for the study that required the least project and domain knowledge. Additionally, we did not disclose the true purpose of the user study to the subjects until the end of each participant's session.

8 DISCUSSION

Our findings suggest that Causal Testing can be useful for understanding root causes and debugging defects. Below, we discuss some takeaway points based on our studies and results.

Encapsulating causality in generated tests. Our user study found that having passing and failing tests that are both similar to the original test that exposed a given defect and exemplify expected and unexpected behavior are especially useful for understanding, and even debugging, software defects. Participants especially found the passing tests that provided examples of expected behavior useful for understanding why a test failed. This suggests that our Causal Testing technique can be used to generate tests that encapsulate *causality* of a given defect and that an important aspect of debugging is being able to identify expected behavior when software is behaving unexpectedly.

Execution information for defect understanding & repair. Execution traces can be useful for finding the location of a defect [12], and our study has shown that such traces can also be useful for understanding and repairing defects. Our study also suggests that the kind of execution trace provided by a Causal Testing tool like Holmes can provide information specific to the root cause of the defect. Participants in our study found comparing execution traces useful for understanding why the test was failing but most importantly how the test should behave differently for a fix. For some participants, the execution trace information was the most useful of all. These results also support the use of execution traces when conducting causal experiments.

Causal Testing as a complementary testing technique. Our findings support Causal Testing as a complement to existing debugging tools, such as JUnit. Understandably, participants sometimes found themselves needing information that Holmes does not provide, especially once they understood the root cause and needed to repair the defect. Our findings suggest that Causal Testing is most useful for root cause identification. Still, majority of the participants in our study found Holmes useful for both cause identification and defect repair, despite sometimes taking longer to resolve defects with Holmes. We speculate that increased familiarity with tools like Holmes would improve developers ability to use the right tool at the right time, improving debugging efficiency, as supported by prior studies [26].

Supporting developers with useful tools. Often when creating or using a tool, the goal is to decrease developer effort (e.g., cost, time, etc.) such that developers will want to use that tool in practice. Our findings suggest that although developers do find value in tools that can decrease their effort, developers are able to find value in tools that are shown to be useful in practice even when effort is not necessarily lessened. When looking at the results from our study, participants often took more time to finish when using Holmes than when not using Holmes. However, despite this and other challenges developers encountered (see previous paragraph), participants still generally found our tool useful for both understanding and debugging defects. This also means that an important part of evaluating a tool intended for developer use if whether the tool provides useful information in comparison to, or in our case along with, existing tools available for the same problem.

9 RELATED WORK

Like Causal Testing, Delta debugging [46, 47] aims to help developers understand the cause of a set of failing tests. Given a failing test, the underlying *ddmin* algorithm minimizes that test's input such that removing any other piece of the test makes the test pass [22]. Delta debugging can also be applied to a set of test-breaking code changes to minimize that set, although in that scenario, multiple subsets that cannot be reduced further are possible because of interactions between code changes [39, 47]. By contrast, Causal Testing does not minimize an input or a set of changes, but rather produces *other* inputs (not necessarily smaller) that differ minimally but cause relevant behavioral changes. The two techniques are likely complementary in helping developers debug.

When applied to code changes, delta debugging requires a correct code version and a set of changes that make it fail. Iterative delta debugging lifts the need for the correct version, using the version history and traditional delta debugging to produce a correct version [3]. Again, Causal Testing is complementary, though future work could extend Causal Testing to consider the development history to guide fuzzing.

Fault localization (also known as automated debugging) is concerned with locating the line or lines of code responsible for a failing test [2, 28, 43]. Spectral fault localization uses the frequency with which each code line executes on failing and passing tests cases to identify the suspicious lines [13, 28]. Accounting for redundancy in test suites can improve fault localization precision [20, 21]. MIMIC can also improve fault localization precision by synthesizing additional passing and failing executions [48], and Apollo can do so by generating tests to maximize path constraint similarity [4]. Statistical causal inference uses observational data to improve fault localization precision [5, 6]. Unfortunately, research has shown that giving developers the ground truth fault location (even from state-of-the-art fault localization techniques) does not improve the developers' ability to repair defects [35], likely because understanding defect causes requires understanding more code than just the lines that need to be edited. By contrast, Causal Testing attempts to illustrate the changes to software inputs that *cause* the behavioral differences, and a controlled experiment has shown promise that Causal Testing positively affects the developers' ability to understand defect causes.

Mutation testing targets a different problem than Causal Testing, and the approaches differ significantly. Mutation testing mutates the source code to evaluate the quality of a test suite [30, 31]. Causal Testing does not mutate source code (it perturbs test inputs) and helps developers identify root causes of defects, rather than improving test suites (although it does generate new tests.) In a special case of Causal Testing, when the defect being analyzed is in software whose input is a program (e.g., a compiler), Causal Testing may rely on code mutation operators to perturb the inputs.

Reproducing field failures [25] is an important part of debugging complementary to most of the above-described techniques, including Causal Testing, as these techniques typically start with a failing test case. Field failures often tell more about software behavior than in-house testing [42].

Fuzz testing is the process of changing existing tests to generate more tests [17, 18] (though, in industry, fuzz testing is often synonymous with automated test input generation). Fuzz testing has been used most often to identify security vulnerabilities [18, 41]. Fuzzing can be white-box, relying on the source code [18] or black-box [29, 41], relying only on the specification or input schema. Causal testing uses fuzz testing and improvements to fuzz testing research can directly benefit Causal Testing by helping it to find similar test inputs that lead to different behavior. Fuzzing can be used on complex inputs, such as programs [23], which is necessary to apply Causal Testing to software with such inputs (as is the case for, for example, Closure, one of the subject programs we have studied). Fuzz testing by itself does not provide the developer with information to help understand defects' root causes, though the failing test cases it generates can certainly serve as a starting point.

The central goal of automated test generation (e.g., EvoSuite [16], and Randoop [34]) and test fuzzing is finding new failing test cases. For example, combining fuzz testing, delta debugging, and traditional testing can identify new defects, e.g., in SMT solvers [10]. Automated test generation and fuzzing typically generate test inputs, which can serve as regression tests [16] or require humans to write test oracles. Without such oracles, one cannot know if the tests pass or fail. Recent work on automatically extracting test oracles from code comments can help [8, 19, 40]. Differential testing can also produce oracles by comparing the executions of the same inputs on multiple implementations of the same specification [9, 11, 14, 36, 38, 45]. Identifying defects by producing failing tests is the precursor to Causal Testing, which uses a failing test to help developers understand the defects' root cause.

10 CONTRIBUTIONS

We have presented Causal Testing, a novel method for identifying root causes of software defects that supplements existing testing and debugging tools. Causal Testing is applicable to 71% of real-world defects in the Defects4J benchmark, and for 77% of those, it can help developers identify the root cause of the defect. Developers using Holmes, a proof-of-concept implementation of Causal Testing, were 8% (96% vs. 88%) more likely to correctly identify root causes than without Holmes. Majority of developers that used Holmes found it most useful when attempting to understand why a test failed and in some cases how to repair the defect. Overall, Causal Testing shows promise for improving the debugging process.

REFERENCES

[1] afl 2018. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[2] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering (ISSRE)*. 143–151.

[3] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246.

[4] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed test generation for effective fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*. 49–60.

[5] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal inference for statistical fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*. Trento, Italy, 73–84.

[6] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2011. Mitigating the confounding effects of program dependences for effective fault localization. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. Szeged, Hungary, 146–156.

[7] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. 2016. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, USA, 211–221.

[8] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*. Amsterdam, Netherlands, 242–253. https://doi.org/10.1145/3213846.3213872

[9] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *IEEE Symposium on Security and Privacy (S&P)*. 114–129. https://doi.org/10.1109/SP.2014.15

[10] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *International Workshop on Satisfiability Modulo Theories (SMT)*. 1–5.

[11] Yuting Chen and Zhendong Su. 2015. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 793–804. https://doi.org/10.1145/2786805.2786835

[12] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight defect localization for Java. In *European Conference on Object Oriented Programming (ECOOP)*. Glasgow, UK, 528–550.

[13] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. 2016. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. *CoRR* abs/1607.04347 (2016). http://arxiv.org/abs/1607.04347

[14] Robert B. Evans and Alberto Savoia. 2007. Differential Testing: A New Approach to Change Detection. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) Poster track*. Dubrovnik, Croatia, 549–552. https://doi.org/10.1145/1295014.1295038

[15] Laura Faulkner. 2003. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers* 35, 3 (2003), 379–383.

[16] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering (TSE)* 39, 2 (February 2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[17] Patrice Godefroid. 2007. Random testing for security: Blackbox vs. whitebox fuzzing. In *International Workshop on Random testing (RT)*. 1.

[18] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*. 151–166.

[19] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)*. Saarbrücken, Genmany, 213–224. https://doi.org/10.1145/2931037.2931061

[20] Dan Hao, Ying Pan, Lu Zhang, Wei Zhao, Hong Mei, and Jiasu Sun. 2005. A Similarity-aware Approach to Testing Based Fault Localization. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Long Beach, CA, USA, 291–294.

[21] Dan Hao, Lu Zhang, Hao Zhong, Hong Mei, and Jiasu Sun. 2005. Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study. In *IEEE International Conference on Software Maintenance (ICSM)*. 683–686.

[22] Ralf Hildebrandt and Andreas Zeller. 2000. Simplifying failure-inducing input. In *International Symposium on Software Testing and Analysis (ISSTA)*. 135–145.

[23] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security Symposium*. Bellevue, WA, USA, 445–458.

[24] InTrace 2018. InTrace. https://mchr3k.github.io/org.intrace.

[25] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *ACM/IEEE International Conference on Software Engineering (ICSE) (ICSE '12)*. Zurich, Switzerland, 474–484.

[26] Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder, Emerson Murphy-Hill, Sarah Heckman, and Caitlin Sadowski. 2016. A cross-tool communication study on program analysis tool notifications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Seattle, WA, USA, 73–84. https://doi.org/10.1145/2950290.2950304

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. San Fransisco, CA, USA, 672–681.

[28] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *International Conference on Software Engineering (ICSE)*. Orlando, FL, USA, 467–477. https://doi.org/10.1145/581339.581397

[29] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *ACM Conference on Computer and Communications Security (CCS)*. Alexandria, VA, USA, 279–288. https://doi.org/10.1145/1455770.1455806

[30] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, 433–436.

[31] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, 315–326.

[32] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, 437–440.

[33] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. 2015. Reducing feedback delay of software development tools via continuous analyses. *IEEE Transactions on Software Engineering (TSE)* 41, 8 (August 2015), 745–763. https://doi.org/10.1109/TSE.2015.2417161

[34] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Conference on Object-oriented Programming Systems and Applications (OOPSLA)*. Montreal, QC, Canada, 815–816. https://doi.org/10.1145/1297846.1297902

[35] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *International Symposium on Software Testing and Analysis (ISSTA)*. Toronto, ON, Canada, 199–209. https://doi.org/10.1145/2001420.2001445

[36] Vipin Samar and Sangeeta Patni. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. *CoRR* abs/1706.09357 (2017). http://arxiv.org/abs/1706.09357

[37] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 248–259.

[38] Varun Srivastava, Michael D. Bond, Kathryn S. McKinley, and Vitaly Shmatikov. 2011. A Security Policy Oracle: Detecting Security Holes Using Multiple API Implementations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, USA, 343–354. https://doi.org/10.1145/1993498.1993539

[39] Roykrong Sukkerd, Ivan Beschastnikh, Jochen Wuttke, Sai Zhang, and Yuriy Brun. 2013. Understanding Regression Failures through Test-Passing and Test-Failing Code Changes. In *International Conference on Software Engineering New Ideas and Emerging Results Track (ICSE NIER)* (22–24). San Francisco, CA, USA, 1177–1180. https://doi.org/10.1109/ICSE.2013.6606672

[40] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *International Conference on Software Testing, Verification, and Validation (ICST)*. Montreal, QC, Canada, 260–269. https://doi.org/10.1109/ICST.2012.106

[41] Robert J. Walls, Yuriy Brun, Marc Liberatore, and Brian Neil Levine. 2015. Discovering Specification Violations in Networked Software Systems. In *International Symposium on Software Reliability Engineering (ISSRE)* (2–5). Gaithersburg, MD, USA, 496–506. https://doi.org/10.1109/ISSRE.2015.7381842

[42] Qianqian Wang, Yuriy Brun, and Alessandro Orso. 2017. Behavioral Execution Comparison: Are Tests Representative of Field Behavior?. In *International Conference on Software Testing, Verification, and Validation (ICST)* (13–18). Tokyo, Japan, 321–332. https://doi.org/10.1109/ICST.2017.36

[43] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. 2010. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software (JSS)* 83, 2 (2010), 188–208.

[44] James Woodward. 2005. *Making things happen: A theory of causal explanation*. Oxford University Press.

[45] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[46] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Toulouse, France, 253–267. https://doi.org/10.1145/318773.318946

[47] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (February 2002), 183–200. https://doi.org/10.1109/32.988498

[48] Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. 2014. MIMIC: Locating and understanding bugs by analyzing mimicked executions. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 815–826.