TOWARDS EFFICIENT TESTING AND DEBUGGING OF EMERGING SOFTWARE APPLICATIONS

by

HUADONG FENG

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

May 2021

THE UNIVERSITY OF TEXAS AT ARLINGTON

Acknowledgements

It has been 25 years since I first started school. It seemed like yesterday when I had my father yelling in my ears asking me what looked wrong when I wrote Ɛ for number three in my entire homework. This sure was a long, but incredible journey. Overall, I consider myself extremely lucky and blessed. I have had a series of great teachers that have inspired wonders and curiosities in me. I am forever grateful to these people that have brought me to where I am now, writing this dissertation.

Firstly, I would like to thank Dr. Lei, my supervising professor, for his patience, guidance, encouragement, and inspiration. This dissertation would not have been possible without his help and generous support. I am grateful for Dr. Lei, for making me a better independent thinker, problem solver, and decision maker. It was truly an honor working with Dr. Lei.

Secondly, I would like to thank my committee members, Prof. Levine, Dr. Csallner, and Dr. Ming, for generously sharing their time and ideas.

Finally, my deepest gratitude to my family for their continuous and unparalleled love, help and support. I am grateful to my wife for always being there for me as a friend. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am. They selflessly encouraged me to explore new directions in life and seek my own destiny. This journey would not have been possible if not for them, and I dedicate this milestone to them.

May 2021

Abstract

TOWARDS EFFICIENT TESTING AND DEBUGGING OF EMERGING SOFTWARE APPLICATIONS

Huadong Feng

The University of Texas at Arlington, 2021

Supervising Professor: Yu Lei

Big Data and Smart Contract are among the top emerging technologies tipped to revolutionize the way businesses and organizations are run. Testing and debugging are the most important tasks during the development of any software application. Big data and smart contract applications possess unique characteristics. There is an urgent need to develop efficient techniques for testing and debugging these applications.

The first part of the dissertation addresses the problem of how to debug big data applications. When a failure occurs in big data applications, debugging at the system-level can be expensive due to the large amount of data being processed. We introduce a test generation framework for effectively generating method-level tests to facilitate debugging of big data applications. This is achieved by running a big data application with the real dataset and by automatically recording input to a small number of method executions, which we refer to as method-level tests, while preserving certain code coverage, e.g., line coverage. When debugging, a developer could inspect the execution of these method-level tests, instead of the entire program execution with the real dataset, which could be time-consuming. We implemented the framework and applied the framework to seven data mining algorithms. The results show that only a very small number of method-level tests

need to be recorded to preserve code coverage. Furthermore, these tests could kill between 53.08% to 96.89% of the mutants generated using a third-party tool. This suggests that the framework could significantly reduce the effort required for debugging big data applications.

The second part of the dissertation addresses the problem of how to test smart contracts A smart contract is a program deployed on blockchain and is often used to handle financial transactions. Unlike traditional programs, contract code cannot be changed after it is deployed. Any security breach would be permanent and could be difficult to be remedied. In this dissertation, we present a fuzzing approach to testing smart contracts. While significant progress has been made, achieving high code coverage remains an important concern for fuzzing. Our fuzzing approach utilizes constraint solving, selective state exploration, and combinatorial testing to improve code coverage. Constraint solving is used to generate test inputs that meet preconditions in a smart contract. Selective state exploration allows different state-dependent behaviors to be exercised while alleviating the state explosion problem. Combinatorial testing is used together with fuzzing to make the testing process more efficient. We implemented our approach in a tool called MagicMirror and evaluated our approach using more than 2,000 contracts. The experimental results show that MagicMirror is effective for achieving high code coverage and detecting vulnerabilities. MagicMirror has been publicly released by National Institute of Science and Technology (NIST).

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

Big Data and Smart Contract are among the top emerging technologies tipped to revolutionize the way businesses and organizations are run. Testing and debugging are the most important tasks during the development of any software application. Big data and smart contract applications possess unique characteristics. There is an urgent need to develop efficient techniques for testing and debugging these applications.

1.1. Research Overview

This dissertation consists of two parts. In the first part, we present an approach to debugging big data applications. Big data applications process and analyze large volumes of data, often measured by gigabytes or more. The execution time of big data applications can range from hours to days. When failure occurs, it is impractical to debug big data applications at the system level. The major challenge is how to debug with less effort, less time, and still preserve the fault detection effectiveness. In this dissertation, we present a framework that can significantly reduce the number of method executions that developers have to manually inspect while maintaining a high probability that the failing method execution(s) is among the selected small number of method-level tests. On average, the execution time is reduced by over 99% when executing the method-level tests generated by our framework instead of the system-level execution.

In the second part of the dissertation, we present a fuzzing approach to test smart contracts. A smart contract is a program deployed on blockchain and is often used to handle financial transactions. Unlike traditional programs, contract code cannot be changed after it is deployed. Any security breach would be permanent and could be difficult to be remedied. Hence, it is important to thoroughly test smart contracts before they are deployed on the blockchain. While significant progress

has been made to fuzzing, achieving high code coverage remains an important concern for fuzzing. In this dissertation, we present a fuzzing tool called MagicMirror. In our experiment, MagicMirror outperforms current state-of-the-art smart contract fuzzing tools in both code coverage and vulnerability detection abilities.

1.2. SUMMARY OF PUBLICATIONS

This dissertation is presented in an article-based format and includes two research papers. In Chapter 2, we present the paper titled, "A Method-Level Test Generation Framework for Debugging Big Data Applications", which was published in the IEEE International Conference on Big Data (Big Data), in 2018. The paper reports our work for improving the efficiency of debugging big data applications. In our approach, we focus on identifying a small number of method executions from the failing system-level execution that can effectively induce method-level failures that propagated into the system-level failing output. The main idea is to evaluate each method execution based on certain testing effectiveness criteria, such as line coverage, edge, node and different types of path coverage based on Control Flow Graph (CFG), or any other types of static code analysis measurements. Then we record the method executions as method-level tests when they cover any new entities of the selected testing effectiveness criterion. So only the necessary method executions with respect to the selected criterion will be executed later for debugging purposes instead of the entire system. Based on the testing effectiveness criterion we choose, a much smaller set of method-level tests can achieve the exact same test effectiveness for a selected method as the original system-level execution. The framework we implemented will analyze and record necessary method executions at runtime with a relatively small overhead depending on the number of the method executions, and the size of the method-level inputs to be serialized. Our framework can significantly reduce the number of method

executions that developers have to manually inspect while maintaining a high probability that the failing method execution(s) is among the selected small number of method-level tests.

In Chapter 3, we present the paper titled, "MagicMirror: Towards High-Coverage Fuzzing of Smart Contracts", which was submitted to the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), in 2021. The paper reports our work for improving fuzzing testing for smart contracts.

Our approach is centered on how to increase code coverage by addressing three challenges: meeting preconditions, state-dependent behaviors, and combinatorial explosion. For meeting preconditions, in many functions, there are *require* statements written at the beginning of the function. These statements specify preconditions, i.e., conditions that must be satisfied before a function can be successfully executed. Test inputs that do not meet preconditions would cause the current transactions to be reverted. To address the meeting preconditions challenge, our approach identifies preconditions and uses a constraint solver to generate test inputs that satisfy the preconditions. For state-dependent behaviors, like parameters, state variables are also input to a function, and their values may affect the behavior of the function. Thus, a function needs to be tested at different states; otherwise, some state-dependent behaviors may not be exercised. However, unlike parameters, state variables cannot take arbitrary values due to the reachability concern. We could derive reachable states by exploring the state space of a smart contract; however, this would introduce the state explosion problem. To address the state-dependent behaviors challenge, our approach uses a selective state exploration framework to derive reachable states while alleviating the state explosion problem. For combinatorial explosion, when the number of parameters is large, the input space of a function can be huge due to the combinatorial explosion

problem. Many vulnerabilities are due to interaction between parameters. However, important combinations of values of parameters can be easily missed by pure random test generation. To address the combinatorial explosion challenge, we combine fuzzing and Combinatorial Testing (CT). The CT tests allow us to exercise parameter interactions in a systematic manner, while fuzzing is used to discover important parameter values, which further improves the quality of CT tests. We implemented our approach in a tool called MagicMirror. We conducted an experimental evaluation of our approach by comparing MagicMirror to two recently published state-of-the-are smart contract fuzzing tools, sFuzz and ILF. Our experiment results show that MagicMirror performs better than sFuzz and ILF on both code coverage and vulnerability detection abilities.

Chapter 2. A Method-Level Test Generation Framework for Debugging Big Data Applications

The chapter contains a paper published in the IEEE International Conference on Big Data (Big Data), in 2018.

A Method-Level Test Generation Framework for Debugging Big Data Applications*

Huadong Feng[1], Jaganmohan Chandrasekaran[1], Yu Lei[1], Raghu Kacker[2], D. Richard Kuhn[2]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019, USA

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA

Abstract – When a failure occurs in a big data application, debugging with the original dataset can be difficult due to the large amount of data being processed. This paper introduces a framework for effectively generating method-level tests to facilitate debugging of big data applications. This is achieved by running a big data application with the original dataset and by recording the inputs to a small number of method executions, which we refer to as method-level tests, that preserve certain code coverage, e.g., edge coverage. The size of each method-level test is further reduced if needed, while maintaining code coverage. When debugging, a developer could inspect the execution of these method-level tests, instead of the entire program execution with the original dataset. We applied the framework to seven algorithms in the WEKA tool. The initial results show that in many cases a small number of method-level tests are sufficient to preserve code coverage. Furthermore, these tests could kill between 57.58% to 91.43% of the mutants generated using a mutation testing tool. This suggests that the framework could significantly reduce

the efforts required for debugging big data applications.

Keywords – Testing; Unit Testing; Big Data Application Testing; Test Generation; Test Reduction; Debugging; Mutation Testing;

2.1. INTRODUCTION

Big data applications are software programs that process large amounts of data. Debugging big data applications can be complicated and time-consuming. This is due to the fact that inspecting the execution of a big data application often involves long execution time, a large number of method executions, and/or a large number of objects. For example, a classification algorithm, called *DecisionTable*, in the WEKA tool [12] takes more than two hours to execute the *Heterogeneity Activity Recognition Dataset* (HAR) from the UC Irvine (UCI) Machine Learning Repository [13]. During the execution, one of the *DecisionTable's* methods, named *updateStatsForClassifier*, is executed more than half a billion times. (This method has 66 lines of code, not including comments and spaces.) If there exists a fault in this method, it can be very difficult to locate this fault due to the large number of times this method is executed.

Some approaches have been proposed to reduce the effort required for testing and debugging big data applications at the system level [1, 2, 3, 4, 5]. For example, data mining and machine learning methods are used to reduce the size of the original dataset or generate synthetic datasets [3, 4] for the testing purpose. The reduced dataset using such methods are executed at the system level, which can still be time-consuming. Furthermore, these methods are not designed to reproduce the original failure. Debugging approaches such as delta debugging [8] can identify the minimum failure-inducing input at the system level, which can reduce the size of the input while preserving the failure triggered by the original dataset. However, delta debugging can be very expensive for big data applications. This is because it

requires the input data be recursively split into smaller chunks, each of which has to be executed at the system level. For big data applications, there can be a large number of chunks and system-level execution of each chunk can be time-consuming.

Our approach consists of two major steps. In the first step, we re-execute the failing system-level execution to record method-level tests for suspicious method(s). The main idea is to evaluate each method execution based on a chosen coverage criterion. In this paper, we used edge coverage, edge-pair coverage and edge-set coverage based on the Control Flow Graph (CFG) [11]. Note that other coverage criteria, e.g., prime-path coverage [11], could also be used in our approach. We record the input to a method execution as a method-level test when it covers any new coverage element with respect to the chosen coverage criterion. In the second step, we reduce method-level tests with large collection-typed variables using binary reduction. The reduced tests preserve the same coverage achieved by the originally recorded method-level tests. During debugging, a developer will first identify suspicious methods based on his or her understanding of the program. Then, the developer will only need to re-execute the reduced method-level tests recorded for these methods, instead of executing the entire application with the original dataset. Doing so could significantly speed up the debugging process.

We conducted an experimental evaluation of our approach. In our experiments, we selected seven methods from four machine learning algorithms that were implemented in WEKA using Java. The four machine learning algorithms from WEKA and two datasets from UCI dataset repository were selected based on the execution time and size of datasets. Method-level tests were recorded for these seven methods based on three coverage criteria, including edge coverage, edge-pair coverage, and edge-set coverage. (The three coverage criteria are defined in Section 2.2.1) On average, 4.4 tests were recorded for edge coverage, 5.9 tests for edge-pair

coverage, and 18.6 tests for edge-set coverage. While initially, the seven methods were executed from 191 to half a billion times. For some of the recorded method-level tests with large-size inputs, e.g., the previously mentioned *updateStatsForClassifier* method in the *DecisionTable* algorithm, we further reduced the size of the inputs using a binary reduction technique while preserving the same coverage achieved by the original method-level test. For example, the average input size for *updateStatsForClassifier* was reduced to 12.53 MB from 1269.76 GB.

Moreover, test effectiveness was evaluated using PITest (PIT) [16], a commonly used mutation testing tool. Mutation testing seeds faults in a systematic manner to simulate mistakes that developers may make during programming. All 25 available mutant generators were enabled for mutant generation. When combining each set of tests generated for the edge, edge-pair, and edge-set coverage for each method, the mutant killing rate ranges from 57.58% to 91.43%.

We summarize the contributions of our paper as follows:

• We present a new framework for debugging big data applications based on method-level tests. Compared to executing the original dataset at the system level, these method-level tests can be much faster to execute and inspect, which could significantly speed up the debugging process.

• We built a prototype that implements our framework and conducted an experimental evaluation of the framework. The evaluation results suggest that our framework could significantly reduce the time and effort required for debugging big data applications.

The rest of the paper is organized as follows. Section 2.2 presents the details of our approach and discusses several implementation challenges. Section 2.3

presents the experimental design and analysis of the experimental results. Section 2.4 provides an overview of existing work that is closely related to ours. Section 2.5 provides concluding remarks as well as several directions for our future work.

2.2. APPROACH

Our approach consists of two major steps, recording method-level tests and reducing the size of the recorded tests. In this section, Section 2.2.1 presents our approach to recording method-level tests based on a given coverage criterion. Section 2.2.2 presents our approach to reducing the size of a recorded test.

2.2.1. Record Test

In a typical scenario, once a failure occurs, a developer identifies several suspicious locations based on his or her understanding of the program. Next, the developer could set up breakpoints in these locations and then start the debugging process with the system-level inputs. The breakpoints allow the developer to inspect the program state during the debugging process. This approach may not be effective for big data applications. This is because when the dataset is large, a breakpoint may be executed for a large number of times before an incorrect program state is found, and each breakpoint has to be inspected manually.

In our approach, the developer first identifies suspicious methods, in a way that is similar to the identification of suspicious locations. Next, our approach runs the program with the original dataset and records, for each suspicious method, a small number of method executions, which we refer to as method-level tests, based on a specific coverage criterion. The method-level tests recorded for a given method achieve the same coverage criterion as the original dataset for the method. The developer can then debug each method with the recorded method executions, instead of a potentially large number of method executions. Since the same

coverage criterion is satisfied, there is a high probability that debugging these recorded method-level tests would allow us to detect the fault that may have caused the failure observed at the system level.

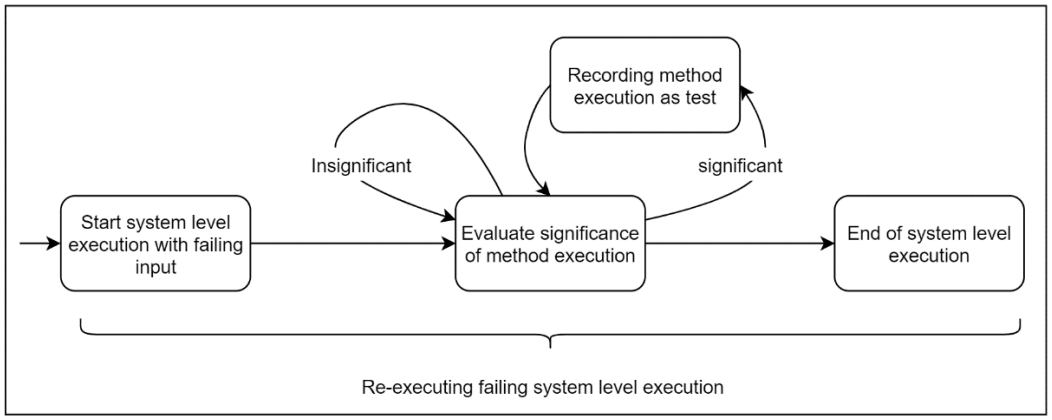


Figure 2-1. Recording Process at Runtime

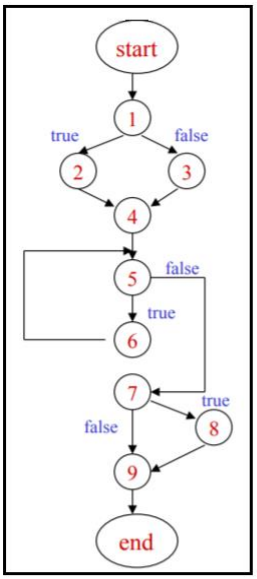
After the developer identifies a list of suspicious methods to be recorded, we instrument these methods to capture the coverage elements that need to be covered for the selected coverage criterion. After instrumentation, our recording process at runtime is shown in Figure 2-1. While re-executing the failing system-level execution, each method execution of the suspicious methods is evaluated to determine whether it is significant based on the selected coverage criterion. A method execution is considered to be significant if it covers at least one new coverage element. When a method execution is deemed to be significant, its corresponding input for reproducing the method execution is recorded as a method-level test. Otherwise, the execution will continue until it reaches the next significant method execution.

In this paper, we will use edge coverage [11], edge-pair coverage [11], and edge-set coverage, as the coverage criteria based on Control Flow Graph (CFG) to determine if a given method execution is significant. A CFG is a graphical

representation of all possible paths that might be traversed by a program at runtime. Thus it captures information about how the control is transferred in a program.

Figure 2-2 shows an example CFG. In a CFG, each node in the graph represents a basic block, i.e. a sequence of consecutive statements with a single entry and a single exit point[11]. A directed edge [11] represents that the control can flow from one node to another. And a path [11] is a sequence of nodes, where each pair of adjacent nodes is an edge.

```
1.  begin
2.     int x, y, power;
3.     float z;
4.     input (x, y);
5.     if (y < 0)
6.        power = -y;
7.     else
8.        power = y;
9.     z = 1;
10.    while (power != 0) {
11.       z = z * x;
12.       power = power – 1;
13.    }
14.    if (y < 0)
15.       z = 1/z;
16.    output (z);
17. end;
```



Block	Lines	Entry	Exit
1	2, 3, 4, 5	2	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

Figure 2-2. Example of Control Flow Graph

We record the method executions as method-level tests when they cover any new coverage elements with respect to the chosen coverage criterion. For edge coverage, each edge covered by a method execution is recorded for the method evaluation. For edge-pair coverage, each edge-pair (reachable path of length up to two) is recorded for the method evaluation. Note that when edge-set coverage is used, a method execution is considered significant if it covers a unique set of edges, i.e., no other method executions exactly cover the same set of edges. Also note that other coverage criteria, e.g., prime-path coverage [11], could also be used in our

approach.

To record method-level tests, three major tasks need to be accomplished, including instrumentation, method execution evaluation, and serialization. We further discuss these tasks in the following subsections.

2.2.1.1. Instrumentation

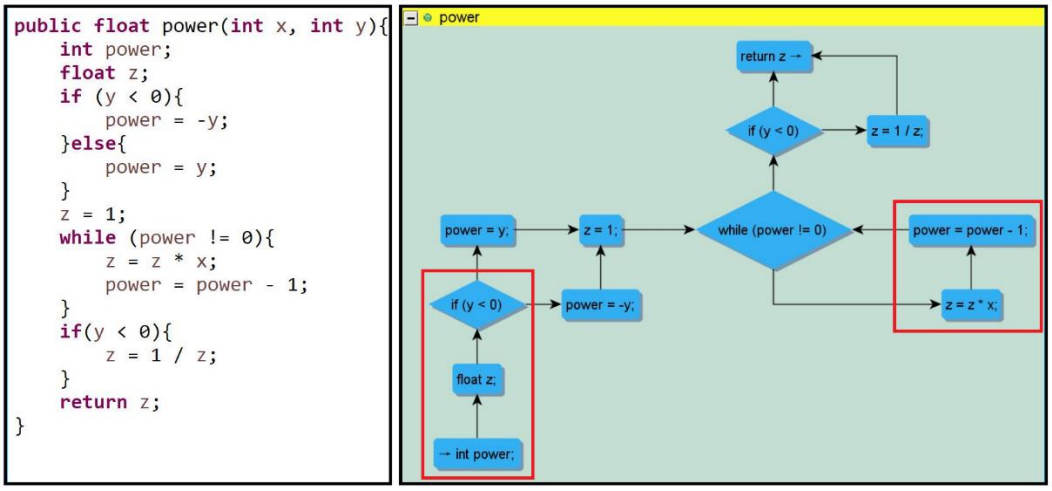


Figure 2-3. Example of Modifying Generated Control Flow Graph

We use a tool called Atlas [15], which is an Eclipse plugin developed by EnSoft Corp to automatically generate CFGs from the source code of a selected method. Atlas uses each line of code as a basic block. This is different from the classical definition [11] that a basic block consists of a sequence of consecutive statements with a single entry and a single exit point. Figure 2-2 shows a simple method and its CFG generated using Atlas. We modify the generated CFGs from Atlas by combining blocks that are in a consecutive sequence without inner branches. Doing so reduces the amount of instrumentation and thus the runtime overhead when executing the instrumented code. The red rectangle in Figure 2-3 marks the lines of code combined to be a basic block as we previously defined.

Once we have the CFG of a suspicious method, we instrument the method by adding a few lines of code that invokes our recording program. Figure 2-4 shows an example of how we instrument a sample method. The highlighted statements are extra code added by instrumentation. The code from line 3 to line 10 initializes the recording process. They are inserted at the beginning of a suspicious method. The ParaArray array contains the list of input parameters used for a method execution. The ParaTypeArray array contains the object types of the input parameters, which are needed to reload the recorded inputs using Java Reflection. When recording a method execution, we record not only the input parameters but also the current object on which the suspicious method was invoked, to store the instance variables accessed during the execution. They are loaded into our system using the "R.loadInputs(ParaArray, this);" statement. The statement "R.enterBlock(#number);" is added before each basic block to record the index of the basic block when it is executed. The block number #number is manually determined based on the previously discussed CFG. Moreover, the statement "R.endOfProcess();" is added before each return statement or at the end of a method to notify our program a method execution is completed, and start the method execution evaluation process.

Recording basic block indexes with multiple entrances at runtime requires more work than just adding the "*R.enterBlock(#number)"* statement in front of it. As shown in Figure 2-4, lines 25 to 26 and lines 30 to 31 are the extra codes added for recording the basic block contains line 24. To record the basic block indexes correctly for basic blocks with multiple entrances such as for *while* loop, *for* loop, *else if,* and *switch* statements, etc., we are inserting the "*R.enterBlock(#number);*" statement before its descendants' "*R.enterBlock(#number);*" statement based on the CFG to capture every execution of such blocks. For example, if we only add "*R.enterBlock(#number);*" statement right before the *while* statement shown in

Figure 2-4 at line 24, when the loop comes back to re-evaluate the loop condition at the *while* statement, the repeated execution of this block will not be captured.

```
1  public float power(int x, int y) {
2      //Insert @ Beginning Of Method
3      RecordMethodExecution R = new RecordMethodExecution();
4      Object[] ParaArray = {x, y};
5      Class[] ParaTypeArray =
6              new Object() {}.getClass().getEnclosingMethod().getParameterTypes();
7      String MethodName = this.getClass().getName() + "/" +
8              new Object() {}.getClass().getEnclosingMethod().getName();
9      R.initializeProcess(MethodName, ParaTypeArray);
10     R.loadInputs(ParaArray, this);
11     //Insert @ Beginning Of Each Block
12     R.enterBlock(0);
13     int power;
14     float z;
15     if (y < 0) {
16         R.enterBlock(1);
17         power = -y;
18     }else {
19         R.enterBlock(2);
20         power = y;
21     }
22     R.enterBlock(3);
23     z = 1;
24     while (power != 0) {
25         R.enterBlock(4);
26         R.enterBlock(5);
27         z = z * x;
28         power = power - 1;
29     }
30     R.enterBlock(4);
31     R.enterBlock(6);
32     if (y < 0) {
33         R.enterBlock(7);
34         z = 1 / z;
35     }
36     R.enterBlock(8);
37     //Insert @ Before Each Return or End of Method
38     R.endOfProcess();
39     return z;
40 }
```

Figure 2-4. Example of Instrumentation

2.2.1.2. Method Execution Evaluation

In our implemented framework, we temporarily store the covered edges, edge-pairs, and edge-set for each method execution. We consider a method

execution to be significant, and thus record the execution as a method-level test if it covers any edge, edge-pair or edge-set that has not been covered before. Note that we check for uncovered edges first for each method execution. This is because if a method execution covers any edge that has not been covered before, it must cover some new edge-pair(s) and a new edge-set. The time complexity for evaluating each method executions is $O(n^2)$ where n represents the number of coverage elements each method execution has to evaluate. For each method execution, each coverage element of the method execution will be compared to the list of the previously covered elements. If a method execution covers any new coverage element, the method execution will be recorded, and the newly covered elements will be added to the list.

2.2.1.3. Serialization

Once a method execution is determined to be significant, we record the inputs of the method execution using serialization. Serialization can be an expensive process, the built-in serialization support in Java is rather slow when serializing large objects. We used an alternative tool called FST [17] that can be ten times faster [17] to improve the performance of our test recording. In our experiments, FST was able to serialize and deserialize objects correctly. However, there are some reported cases [17] where FST was unable to correctly serialize and deserialize objects that the built-in Java serialization could. In comparison, FST provides better performance, but FST does not provide serialization ability that is as strong as the Java built-in serialization.

While our performance is improved using FST, there are still some situations where we experience significant overhead. To ensure an exact copy of the input objects is created, we perform deep copy on the objects by serializing and deserializing these objects. This is needed because the value of an input object

could potentially change during a method execution, especially for void methods that operate on instance variables.

However, most of the stored input objects will not be recorded if the method execution does not cover any new coverage element. Thus, much of the time spent to store the deep copies of objects is unnecessary. These unnecessary time can be huge when a method takes large inputs and/or is executed for a large number of times. The recording overhead can be as high as 7 to 30 times the original system-level execution time for some of the selected methods. In such cases, our solution is recording the method-level tests by executing the entire system twice. In the first execution, we do not store any inputs. Instead, we only record the IDs of significant method executions. In the second execution, we only serialize the selected method executions to store their inputs as method-level tests. Doing so can significantly reduce the runtime overhead in cases where a method takes large inputs or is being executed for a large number of times.

2.2.2. Test Reduction

While the recorded method-level tests can be used for debugging, these tests in some case consist of very large inputs. For example, one of the selected methods *cutPointsForSubset*, its recorded method-level tests have the average size of 1.62GB, executing these tests can take a lot of time. And breakpoints in loop statements can be executed for a large number of times. These inputs are large mostly due to the fact that they contain large collections of objects. For the three methods mentioned above, they all have *Instances* typed (Implements *Collection*) variables that contain instances from the original dataset for processing. Some of the recorded data could potentially be reduced while still reproducing the method execution and preserving the coverage elements. The reduction can further reduce the time for executing the tests, and the debugging efforts required from developers.

Our binary reduction technique is inspired by the commonly used binary search technique. For each recorded method-level test, we divide its collection typed input variables into halves. Next, we take each half and other non-collection typed inputs and re-execute them with the suspicious method. We then check whether a half can preserve the originally covered coverage elements. If one of the halves does preserve all the coverage elements, we will continue dividing it into halves and check for the coverage elements repeatedly, until the minimal subset of the collection variables that can preserve the coverage elements are identified. Note that when preserving the coverage during reduction, we are preserving the exact covered elements of edge coverage, edge-pair coverage, and edge-set coverage.

2.3. EXPERIMENTS

We implemented the initial working prototype of our framework in Java. Some Manual efforts are required from developers to instrument the source code of suspicious methods. After instrumentation, the recording process has been automated. The reduction approach requires developers to manually identify the large collection typed input variables. The re-execution of the recorded and reduced method-level tests has been automated for debugging. We also conducted mutation testing to evaluate the fault detection effectiveness of our recorded and reduced method-level test. The currently implemented coverage criteria are the edge, edge-pair, and edge-set coverage.

In the following, we discuss how we conducted our experiments and present the experiment results. In Section 2.3.1, we discuss how we selected datasets, applications, and methods to be used for our experiments. Section 2.3.2 presents the statistics of the recorded method-level tests. Section 2.3.3 presents the statistics of the reduced method-level tests. Section 2.3.4 presents how we conducted a mutation testing experiment and the results of our mutation testing for both the

recorded tests, and the reduced tests. And finally, Section 2.3.5 presents the performance analysis of our framework. All the source code, recorded method-level tests, reduced method-level tests and mutation reports are publicly available at https://www.dropbox.com/sh/3k4kjwqjpa9i2qv/AAAkeYYNaQOVfT9WGe4OUp_Pa?dl=0 for review. The machine we used for our experiment is a workstation with two Xeon E5-2630V3 8 core CPUs @ 2.40GHz, 64GB DDR4 2133 MT/s memory, and a Samsung 850 EVO 500GB SSD.

2.3.1. Subjects

We design our experiments to reflect real-world situations for evaluating the effectiveness of our framework. First, we randomly selected ten algorithms that are implemented in the WEKA tool. WEKA is one of the most widely used tools for data mining by practitioners. Next, we selected one collection of dataset with the largest number of instances (accessed on 08/18/2018) from the UCI Machine Learning Repository that consists of 440 real-world collected datasets as a start. The selected collection of datasets, *Heterogeneity Activity Recognition* (HAR), contains four datasets for four different types of devices with a total of 43,930,257 instances and 16 attributes. The HAR collection includes several data types, including multivariate, time-series and real numbers. The datasets can be used for both classification and clustering. Among the four datasets, the largest dataset, *Phones_gyroscope*, is used to execute the ten algorithms.

Phones_gyroscope dataset has the size of 1.37GB, it is too large for two of our selected algorithms *EM* and *LibSVM* to finish their execution within a day. The execution time is too long for our experimentation purpose due to our limited time and resources. For these two algorithms, we reduced the size of the *Phones_gyroscope* dataset by dividing the dataset in half and continue to divide in

half until the execution time for *EM* and *LibSVM* are reduced to be near an hour. The reduced *Phones_gyroscope* dataset for *EM* and *LibSVM* now has the size of 3.3 MB. *EM* will now take 5352 seconds (1.49 Hours) to execute and 4491 seconds (1.25 Hours) for *LibSVM*.

Table 2-1. Selected Method Information

Method	**Algorithm**	**# of Covered Lines of Code**	**# of Total Lines of Code**	**# of Execution Count**
buildClusterer	EM	115	165	1,910
cutPointsForSubset	DecisionTable	62	64	29,564
EM_Init	EM	47	53	191
handleNumericAttribute	J48	51	53	28,314
select_working_set	LibSVM	50	52	417,989
selectModel	J48	50	58	12,391
updateStatsForClassifier	DecisionTable	46	66	557,305,280

After two datasets (original *Phone_gyroscope* dataset and the reduced dataset) and ten algorithms' implementations (*Apriori, DecisionTable, EM, HierarchicalClusterer, J48, LibSVM, LinearRegression, MakeDensityBasedClusterer, RandomTree, SimpleKMeans*) have been selected. We select methods with a larger number of executed statements, and a larger number of executions for our experiments. This is because longer methods and methods that have been executed for a larger number of times often require more effort to debug. A total of seven methods are selected. The selected methods and their information are shown in Table 2-1. These methods are then instrumented as previously described in Section 2.2.

Table 2-2. Recorded Method Execution Information

Method	Edge Coverage		Edge-Pair Coverage		Edge-Set Coverage		Total # of Recorded Tests	# of Original Execution Count	Statement Coverage
	# of Tests	# of Covered Edges	# of Tests	# of Covered Edge-Pairs	# of Tests	# of Covered Edge-Sets			
buildClusterer	3	83	4	181	3	3	4	1,910	69.70%
cutPointsForSubset	8	30	9	64	17	17	18	29,564	96.88%
EM_Init	1	24	3	55	1	1	3	191	88.68%
handleNumericAttribute	4	32	5	70	33	33	33	28,314	96.23%
select_working_set	7	41	11	111	61	61	63	417,989	96.15%
selectModel	5	35	5	74	6	6	6	12,391	86.21%
updateStatsForClassifier	3	26	5	59	9	9	11	557,305,280	69.70%

2.3.2. Recorded Method-Level Tests

For our experiments, we have recorded method-level tests for all of the seven selected methods for preserving edge coverage, edge-pair coverage, and edge-set coverage of the original system-level execution. Some important information about the recorded method-level tests is shown in Table 2-2. Note that the statement coverage column in Table 2-2 is for all three types of recorded tests, as well as the original failing system-level execution. This is because edge coverage subsumes statement coverage, once all edges are preserved, all the statement coverage will be preserved as well, and edge-pair coverage and edge-set coverage both subsume edge coverage.

Based on the results shown in Table 2-2, we can see that only a small number of method-level tests are sufficient for preserving coverage for a suspicious method. Empirical studies show that there exists a high correlation between code coverage and fault detection effectiveness. The actual fault detection ability of our recorded method-level tests will be further evaluated using mutation testing in Section 2.3.4. Thus, when failures occur on a system level, it is likely that executing the method-level tests for the suspicious methods would trigger the failure observed during the execution with the original dataset. Thus, the use of method-level tests could potentially save developers a lot of time and efforts.

2.3.3. Reduced Method-Level Tests

As shown in Table 2-3, while some of the tests have a reasonable size, three methods, *cutPointsForSubset*, *selectModel* and *updateStatsForClassifier* have significantly large inputs for their recorded method-level tests. While debugging with these tests is easier than debugging with the original dataset at the system level, loading and debugging these tests could still take a lot of time. We further reduce the size of these tests using our binary reduction approach as discussed in Section

Table 2-3. Test Reduction Results

Method	Total # of Tests	# of Large Collection Variables	Average Input Size (MB)		Total Input Size (MB)		Maximum Input Size (MB)		Minimum Input Size (MB)		Test Execution Time (Seconds)	
			Recorded	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*
buildClusterer	4	1	3.18	3.18	12.75	12.75	3.23	3.23	3.16	3.16	5 s	5 s
cutPointsForSubset	18	2	1628.16	9.77	29306.81	176.25	1628.16	37.22	1628.16	0.001	1836 s	5 s
EM_Init	3	1	4.71	4.71	14.12	14.12	4.34	4.34	5.18	5.18	5 s	5 s
handleNumericAttribute	33	1	54.00	0.74	1781.76	24.42	1300.48	1.75	0.0012	0.001	155 s	2 s
select_working_set	63	2	39.50	0.08	2488.32	5.04	41.9	0.29	1.83	0.002	176 s	1 s
selectModel	6	2	1392.64	0.02	8357.04	0.11	1628.16	0.01	1320.96	0.001	682 s	1 s
updateStatsForClassifier	11	1	1269.76	12.53	13967.34	138.06	1269.76	44.86	1269.76	0.51	875 s	3 s

2.2. In Table 2-3, we compare the differences between the recorded method-level tests before and after they were reduced.

For size reduction, our binary reduction technique was able to reduce the input size of tests for five out of seven methods. Our result shows that the reduction amount is often above 95%. Most of the method-level tests can be reduced significantly while still preserving our selected coverage elements. The coverage element refers to the edges, edge-pairs, and edge-set covered by each recorded method-level test. While one of the tests for *selectModel* can be reduced to 1.7 KB from 1.63 GB, some tests still have a fair amount of input data remaining, such as the reduction from 1.63GB to 37.22 MB for one of the tests of *cutPointsForSubset*. Furthermore, we were unable to reduce any test inputs for two methods, *buildClusterer* and *EM_Init*. We further investigated this by looking into how the variables of collection type are accessed and used. We noticed mainly three different scenarios that may have contributed to our results.

The first scenario is when a collection variable is partially used as inputs. When the partially accessed instances are in a consecutive sequence in the collection variable, or when only one instance is accessed, our binary reduction technique will reduce such collection variable to its minimal subset. However, if the accessed instances are spread across the collection variable, our binary reduction will not be able to identify only the accessed instances. Hence, the reduction may not be minimal, many unnecessary data based on the coverage elements may remain.

The second scenario is when the collection variable is accessed in branching statements, e.g. for the tests recorded for *buildClusterer* and *EM_Init*. The collection variables identified for these two methods were used at a few branching statements and passed to other methods that return value to the execution as well. In this situation, maintaining the exact coverage elements can be difficult to achieve

for our binary reduction technique. As an example, part of the code of *buildClusterer* is shown in Figure 2-5. The *instances* variable was used at an *if* statement and in the conditions of a *for* loop. Reducing the *instance* variable using our binary reduction approach will compromise the originally covered coverage elements (edges, edge-pairs, edge-set) of the method-level tests recorded for the *buildClusterer* method.

```
744       if (m_executionSlots <= 1
745         || instances.numInstances() < 2 * m_executionSlots) {
746         for (i = 0; i < instances.numInstances(); i++) {
747           Instance toCluster = instances.instance(i);
748           int newC =
749             clusterProcessedInstance(
750               toCluster,
751               false,
752               true,
753               m_speedUpDistanceCompWithCanopies ? m_dataPointCanopyAssignments
754                 .get(i) : null);
755           if (newC != clusterAssignments[i]) {
756             converged = false;
757           }
758           clusterAssignments[i] = newC;
759         }
760       } else {
761         converged = launchAssignToClusters(instances, clusterAssignments);
762       }
```

Figure 2-5. Collection Variable Used at Branching Condition

The third scenario is when the collection variable is not accessed at all. In our implementation, to reduce manual efforts required for instrumentation and reproduce method executions precisely, we automatically record both the parameters passed to the method and the object where the method was invoked from, ensuring all possible inputs are recorded. However, not all recorded information is used as inputs, such as for some instance variables of the object where the method was invoked from. In this situation, our binary reduction technique may be able to reduce unnecessary collection variables to empty, while still preserving the coverage elements.

The first and second scenario can potentially use delta debugging [8] or

preserving superset of the coverage elements to further the reduction. However, delta debugging could significantly increase the reduction overhead, and preserving superset of the coverage elements may lose or introduce some coverage elements that could potentially have a large impact on the reduced method-level test. For the third scenario, we can implement systematic static analysis in the future to help our framework identify and record only the necessary inputs for reproducing method executions.

For execution time reduction, many of the recorded set of method-level tests are now taking seconds instead of minutes after the binary reduction. When debugging with these reduced tests, not only the tests will be short and easier to debug, the execution time is also easy to manage.

2.3.4. Mutation Testing

For mutation testing, we used PITest (PIT) [16], a mutation testing tool for Java, to evaluate the fault detection effectiveness of our recorded method-level tests. In PIT, different types of faults (or mutants) are automatically seeded into the source code. Each mutation (a mutated version of source code) simulates a single fault and is executed against the unit tests that developers provide.

Mutation testing requires the provided unit tests to be passing tests. This is because only when the mutant's output differs from the expected output, a mutant is said to be killed. In our experiments, when a method-level test is executed, we record the outputs as the expected output for mutation testing purpose. The output for each test contains not only the returned object if there is one, but also the object where the method was invoked from and the input parameters of the method. This is because the values of these parameters and the object where the method was invoked from could change and should be considered as part of the output.

Table 2-4. Mutation Testing Reduction Results

Method	# of Mutants Generated for Covered Code	Statement Coverage	Edge Coverage Mutant Killing Rate		Edge-Pair Coverage Mutant Killing Rate		Edge-Set Coverage Mutant Killing Rate		Combined Recorded Tests Mutant Killing Rate	
			Recorded	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*
buildClusterer	269	69.70%	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%
cutPointsForSubset	164	96.88%	81.71%	81.1%	81.71%	82.32%	85.98%	85.98%	85.98%	85.98%
EM_Init	102	88.68%	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%
handleNumericAttribute	140	96.23%	89.29%	88.57%	90.71%	90.71%	91.43%	91.43%	91.43%	91.43%
select_working_set	128	96.15%	71.88%	75%	73.44%	75%	75%	78.91%	75%	78.91%
selectModel	132	86.21%	57.58%	57.58%	57.58%	57.58%	62.88%	62.88%	62.88%	62.88%
updateStatsForClassifier	122	69.70%	86.07%	81.15%	86.07%	84.43%	86.89%	86.07%	86.89%	86.89%
Average			79.00%	78.55%	79.42%	79.49%	81.23%	81.67%	81.23%	81.79%

PIT provides a total of 25 different mutators to mutate different type of code. When conducting mutation testing, we have enabled all 25 mutators in PIT for generating mutants in our selected methods. PIT also provides an option to set a timeout factor for executing each test against each mutant. The default is 1.25 times the original test execution time. We increased the timeout factor to 10 times the original execution time, as an effort to avoid false positives killing of mutants. This is because a timed-out mutant is also considered as a killed mutant. We have also increased the Java heap size to 60GB and stack size to 128MB using JVM configuration in PIT, to avoid false positive killing of memory error mutants.

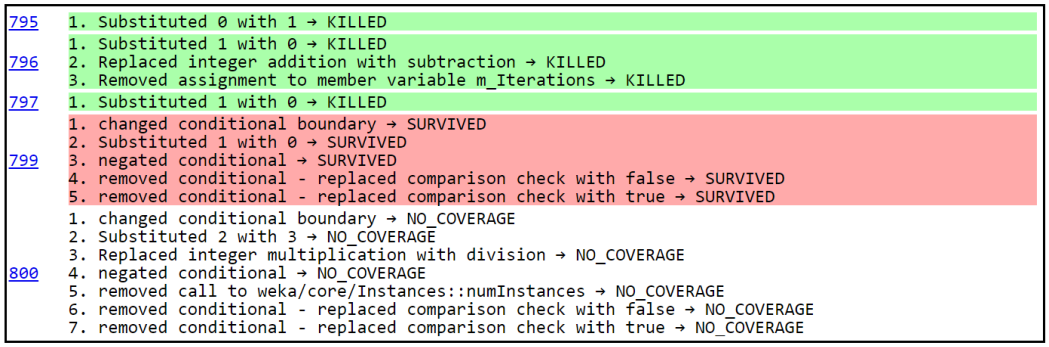


Figure 2-6. Sample Mutation Testing Report

Table 2-4 shows the mutation testing result of our recorded and reduced method-level tests. Note that PIT currently does not support the mutant generation of only covered statements. Because the mutation generation of PIT is done statically, it will generate mutants for all the statements of a selected method, instead of only the reachable ones. In other words, if a mutant is located at a statement that was not covered by any of the tests, the mutant will not be exercised, and thus is impossible to be killed. Such mutants will not be considered in our experiments. This is because if a mutant is not exercised by our recorded tests, it is not exercised by the original system-level execution. The total number of mutants generated for each selected method in Table 2-4 are calculated manually which

consist of only exercised mutants by our tests. This is done by removing mutants that are labeled as *NO_COVERAGE* in the mutation testing report generated using PIT, such as shown in Figure 2-6.

For recorded method-level tests without reduction shown in Table 2-4, we can see that most of the recorded tests for different methods and coverage criteria have a high mutant killing rate. Even without comparing to the original system-level execution, a small number of tests show high effectiveness in detecting potential faults that could occur in the selected methods. For four out of seven selected methods, recorded tests achieve over 80% of mutant killing rate for all the selected coverage criteria. The average mutant killing rate across seven methods are around 80% for all four different sets of tests that achieve edge coverage, edge-pair coverage, edge-set coverage, and these three combined. By only using edge coverage, the recorded method-level tests can achieve reasonably high mutant killing rate. With edge-pair and edge-set coverage, the mutant killing rate is further improved slightly in some cases. This indicates the method-level tests generated using our framework can effectively help developers to debug and find faults they are looking for, while significantly reducing the time and efforts required from developers for debugging.

For reduced method-level tests, their mutant killing rates are nearly the same as their original recorded tests. With differences no larger than 5% of their original killing rate. We even see some cases with increased mutant killing rate, such as for the edge-pair coverage of method "*cutPointsForSubset*". While coverage elements of our specifically selected coverage criteria are maintained, other elements from other coverage criteria could become lost, or may be newly introduced after our binary reduction, such as combinations of the different branches being executed. The mutation testing results of the reduced tests show that even after the input sizes are significantly reduced, the coverage elements and also

the fault detection effectiveness are still preserved. Our binary reduction technique on method-level tests can further help developers to reduce efforts for debugging while maintaining the debugging effectiveness of the method-level tests.

Table 2-5. System-Level Mutation Testing

Method	**Algorithm**	**# of Mutants Killed by System-Level Execution**	**# of Propagatable Mutants Killed by Combined Method-Level Tests**
select_working_set	LibSVM	58	51
selectModel	J48	61	56

We also investigated the two methods *select_working_set* and *selectModel* with the lowest mutant killing rate by comparing their results to the mutation testing results of their system-level execution. We have planned on comparing all recorded method-level tests' mutation testing results with their corresponding system-level execution. However, while mutation testing is a very effective method to evaluate the quality of tests, mutation testing is a rather expensive method to use. In this paper, we only have two system-level mutation testing results for *select_working_set* and *selectModel*. Moreover, their system-level mutation tests both took over one week to complete. Note that some mutants that can be killed with method-level tests are not propagatable on the system level, i.e., a mutant may cause a method execution producing incorrect output, but such incorrect output on the method level did not cause an incorrect system-level output. We considered the option of recording all method executions of a method during its system-level execution. However, it is impractical, because of our selected methods have been executed with a large number of times, and many of them have large inputs as well. For comparing mutation testing results between method-level tests and system-level execution, we will only be considering the propagatable mutants for the

method-level tests.

The system-level mutation testing results for *select_working_set* and *selectModel* are shown in Table 2-5. For *LibSVM*, the system-level execution was able to kill 58 mutants, the combined method-level test of *select_working_set* was able to kill 51 out 58 propagatable mutants with a propagatable mutant killing rate of 87.93%. For *J48*, the system-level execution was able to kill 61 mutants, the combined method-level tests of *selectModel* were able to kill 56 out of 61 propagatable mutants with a propagatable mutant killing rate of 91.80%. The further investigation shows the reason why method-level tests recorded for *select_working_set* and *selectModel* have a lower mutant killing rate. It is likely because their original system-level execution has a lower mutant killing rate.

After investigating the un-killed propagatable mutants in the recorded method-level tests, we discovered three un-killed propagatable mutants from *select_working_set* and one from *selectModel* were mutations related to modifying boundary conditions. This means by adding more coverage criteria related to boundary conditions, a higher mutant killing rate can be achieved for the method-level test. With a few basic coverage criteria implemented for our framework, method-level tests produced by our framework can be very effective in detecting faults during debugging.

2.3.5. Performance Evaluation

We evaluate the performance of our implementation by investigating the original system-level execution time, the time taken to evaluate and record the method-level tests, time taken to reduce tests, and the time taken to execute the recorded method-level tests. The results are shown in Table 2-6. Recall that in the experiments for mutation testing, both inputs and outputs of the selected method executions are recorded. However, the results shown in Table 2-6 are only for

recording the inputs and executing the recorded method-level tests with only inputs without comparing their outputs. This is because, in real-world use of our framework, outputs of the method executions do not need to be recorded.

Table 2-6. Performance Evaluation Results

Method	**Original Execution Time**	**Total Test Recording Time**	**Total Test Execution Time**		**Total Test Reduction Time**
			Recorded	***Reduced***	
buildClusterer	5352 s	6303 s	5 s	5 s	27 s
cutPointsForSubset	9559 s	*21615 s	1836 s	5 s	7558 s
EM_Init	5356 s	5361 s	5 s	5 s	22 s
handleNumericAttribute	6357 s	*14624 s	155 s	2 s	1965 s
select_working_set	4491 s	*11531 s	176 s	1 s	2763 s
selectModel	6357 s	*14212 s	682 s	1 s	3122 s
updateStatsForClassifier	9559 s	*30513 s	875 s	3 s	4088 s

As previously mentioned in Section 2.2, we have two solutions for recording selected method executions. One approach is to serialize and temporarily store the inputs for each method execution and record the inputs locally when a method execution is determined to be significant. This method requires executing the entire system only once. However, in cases where a method has large inputs or is executed for a large number of times, this approach may have a significant performance issue due to all the unnecessary serialization. The other approach is to execute the entire system twice. In the first execution, we evaluate each method execution and store the execution IDs of the method executions. An execution ID is the index of a method execution based on the order of each method executions that happened during the system-level execution. In the second system-level execution, we only serialize and record the inputs of the selected method executions based on their execution IDs. The numbers marked with "*" indicates that the method-level tests were recorded using the second recording approach as shown in Table 2-6. The execution time is computed by subtracting the execution end time

by the execution start time that was created using the Java *System.CurrentTimeMillis()* function.

In Table 2-6, we see that recording method-level tests using our framework can take up to three times of the initial system execution. Additional test reduction time could take as much as two hours based on the size of the inputs (Our binary reduction utilizes serialization for deep copy as well). The reduced tests can be executed for many times during the debugging, the reduction time is a one-time investment, we believe the time is manageable for developers. Moreover, our approach is automated, allowing developers to work on other tasks while running our approach. For executing the recorded method-level tests, we see that it usually takes much less time than executing the entire system, especially for the reduced tests, the execution time can range from as little as one second to five seconds. Overall, we believe that recording and reducing method-level tests using our framework will help developers save a lot of time and efforts in debugging big data applications.

2.4. RELATED WORK

We first review previous work related to generating tests for big data applications. Csallner et al. proposed an approach that uses dynamic symbolic execution to automatically generate tests for general *MapReduce* programs [1]. Morán et al. proposed *MRFlow,* a testing technique tailored to test *MapReduce* programs [5]. *MRFlow* uses data flow test criteria and oriented to transformations analysis between the input and the output in order to detect defects in *MapReduce* programs. Morán et al. also proposed a technique to generate different infrastructure configurations for a given *MapReduce* program that can be used to reveal functional faults [4]. They also proposed an automatic test framework that can detect functional faults automatically [3]. Chandrasekaran et al. proposed an

approach to generate test input data using combinatorial testing for testing big data applications [6]. Previous work reported in [1, 2, 3, 4, 5] focuses on generating tests that help to identify functional faults, i.e., faults that will cause the program to generate unexpected outputs. In contrast, our work focuses on reducing debugging efforts for big data applications. Our tests are recorded in an effort to reproduce failures using a small number of method-level tests.

Second, some work has been reported on debugging big data applications. Gulzar et al. developed a tool, BigDebug, that simulates breakpoints to enable a developer to inspect a program without actually pausing the entire computation [7]. To help a user inspect millions of records passing through a data-parallel pipeline, BigDebug provides *guarded watchpoints*, which dynamically retrieve only those records that match a user-defined guard predicate. Chandrasekaran et al. proposed a technique that uses different annotators to debug the tracking data independently and their debugging results were collected for joint correction propagation for later analysis [9]. Our work is similar to Gulzar [7] and Li [9] in terms of only focusing on a subcomponent of the system. However, our work focuses on recording significant method-level executions to be replayed for debugging suspicious methods. Gulzar [7] and Li [9] focuses on tracking the changes made to certain objects using data flow analysis approach.

Third, our work is also related to existing work that records program information and uses the information to generate unit tests. Pasternak et al. proposed a technique that records interactions that take place during the execution of Java programs and uses these interactions to construct unit tests automatically using GenUTest [10]. Orso et al. proposed a technique and conducted a feasibility study using SCARPE, a prototype tool, for selective capture and replay of program executions [6]. Similar to our work presented in this paper, Orso's technique [6] can be used to automatically generate unit tests based on the recorded information

for testing purpose. Our work is similar to Pasternak [10] and Orso [6] in terms of recording method-level tests based on the system-level execution. However, our work focuses on recording unit tests for debugging one or more failures that have been observed instead of generating tests for triggering failures that have not been observed yet. Furthermore, our work also does not require complex instrumentation techniques on the target's bytecode [6]. Instead, we only employ simple instrumentation that keeps track of code coverage.

Finally, we review work related to reducing input size for the debugging purpose. Zeller et al. proposed Delta Debugging [8] technique to isolate failure-inducing inputs on the system level to reduce work required for debugging. Clause [14] et al. presented a technique based on dynamic tainting for automatically identifying subsets of a program's inputs that are relevant to a failure. These techniques reduce the debugging effort at the system level, in terms that the reduced datasets need to be executed at the system level. This is in contrast with our work that reduces the debugging effort at the method level.

2.5. Conclusion & Future Work

In this paper, we presented a framework to provide developers with method-level tests that were recorded from a failed system-level execution with the original dataset. These method-level tests preserve a given coverage criterion, e.g. edge, edge-pair, and edge-set coverage, and thus are likely to reproduce the failure observed at the system level. The binary reduction is used to further reduce method-level tests with large input. The set of method-level tests that are provided by our approach could help developers to effectively debug suspicious methods against properties of the original input dataset, and significantly reduce time and effort required for debugging big data applications.

There are two major directions for future work. First, we plan to conduct

more experimental evaluation of our approach using more big data applications, datasets, and coverage criteria. Second, we plan to further automate our approach. In particular, we will develop techniques that can fully automate the instrumentation process. Our current approach still needs manual effort in modifying CFG generated by Atlas, inserting code for instrumentation, and identifying collection typed variables for reduction. It is our plan to make the tool publicly available.

2.6. ACKNOWLEDGMENT

This work is supported by a research grant (70NANB15H199) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

2.7. REFERENCES

1. Csallner, C., Fegaras, L., & Li, C. (2011, September). New ideas track: testing mapreduce-style programs. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (pp. 504-507). ACM.
2. Jaganmohan Chandrasekaran, Huadong Feng, Yu Lei, Richard Kuhn, Raghu Kacker, "Applying combinatorial testing to data mining algorithms", Software Testing Verification and Validation Workshops (ICSTW) 2017 IEEE Fourth International Conference on-6th International Workshop on Combinatorial Testing (IWCT), 2017.
3. Morán, J., Bertolino, A., de la Riva, C., & Tuya, J. (2017, July). Towards

Ex Vivo Testing of MapReduce Applications. In Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on (pp. 73-80). IEEE.

4. Morán, J., Rivas, B., De La Riva, C., Tuya, J., Caballero, I., & Serrano, M. (2016, August). Infrastructure-aware functional testing of mapreduce programs. In Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on (pp. 171-176). IEEE.
5. Morán, J., Riva, C. D. L., & Tuya, J. (2015, August). Testing data transformations in MapReduce programs. In Proceedings of the 6th International Workshop on Automating Test Design, Selection and Evaluation (pp. 20-25). ACM.
6. Orso, A., & Kennedy, B. (2005, May). Selective capture and replay of program executions. In ACM SIGSOFT Software Engineering Notes (Vol. 30, No. 4, pp. 1-7). ACM.
7. Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, Miryung Kim. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. Proceeding ICSE '16 Proceedings of the 38th International Conference on Software Engineering, Pages 784-795
8. A. Zeller and R. Hildebrandt, “Simplifying and Isolating Failure-Inducing Input”, IEEE Transactions on Software Engineering28(2), February 2002, pp. 183-200.
9. Mingzhong Li, Zhaozheng Yin. Debugging Object Tracking by a Recommender System with Correction Propagation. In IEEE Transactions on Big Data (Volume: 3, Issue: 4, Dec. 1 2017)
10. Pasternak, B., Tyszberowicz, S., & Yehudai, A. (2009). GenUTest: a unit test and mock aspect generation tool. International journal on software tools for technology transfer, 11(4), 273.

11. N. Li, U. Praphamontripong, and J. Offutt, “An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage,” in Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings, 2009, pp. 220–229.
12. Eibe Frank, Mark A. Hall, and Ian H. Witten (2016). The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition, 2016.
13. Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
14. J. Clause and A. Orso. Penumbra: Automatically identifying failure relevant inputs using dynamic tainting. In ISSTA, pages 249–260, 2009.
15. “Atlas Platform, EnSoft Corp.” http://www.ensoftcorp.com.
16. “PITest.” http://pitest.org/.
17. “FST, fast-serialization.” https://github.com/RuedigerMoeller/fast-serialization.

Chapter 3. MagicMirror: Towards High-Coverage Fuzzing of Smart Contracts

The chapter contains a paper submitted to the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), in 2021.

MagicMirror: Towards High-Coverage Fuzzing of Smart Contracts †

Huadong Feng[1], Xiaolei Ren[1], Qiping Wei[1], Yu Lei[1], Raghu Kacker[2], D. Richard Kuhn[2], Dimitris E. Simos[3]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019, USA

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA

[3]SBA Research, Vienna, Austria

Abstract – A smart contract is a program deployed on blockchain that is often used to handle financial transactions. Unlike traditional programs, contract code cannot be changed after it is deployed. This significantly increases the impact of potential defects in the contract code. Thus, it is important to test smart contracts thoroughly before deployment. In this paper, we present a fuzzing approach to testing smart contracts. Our approach utilizes constraint solving, selective state exploration, and combinatorial testing to improve code coverage. Constraint solving is used to generate test inputs that meet preconditions in a smart contract. Selective state exploration allows different state-dependent behaviors to be exercised in a way that alleviates the state explosion problem. Combinatorial testing is used to break an impasse that may be reached during the fuzzing process. We implemented our approach in a tool called MagicMirror and evaluated our approach using more than 2,000 contracts. The experimental results show that MagicMirror is effective for achieving high code coverage and detecting vulnerabilities.

Keywords – Blockchain, Ethereum, smart contracts, fuzzing, constraint solving, combinatorial testing, security analysis, vulnerability detection.

3.1. INTRODUCTION

A smart contract is a program deployed on blockchain and is often used to handle financial transactions. Unlike traditional programs, contract code cannot be changed after it is deployed. Any security breach would be permanent and could be difficult to be remedied. For example, in April 2016, the reentrancy attack [1] on the DAO smart contract stole more than 3.6 million Ether (equivalent to about $45 million at the time). While the DAO attack was remedied via an expensive and controversial hard fork due to its severe public impact, many other attacks [2, 3, 4] have been observed and have never been resolved.

In recent years many fuzzing approaches have been reported for testing smart contracts. Examples of these approaches include AFL based fuzzing [8], where inputs are generated using strategies such as bit/byte flip and guided with code coverage; grammar-based fuzzing [9], where valid inputs are produced syntactically following certain grammars; input approximation-based fuzzing [8, 11], where the inputs are approximated based on the boolean expressions of certain branching statements; and machine learning-assisted fuzzing [12], where inputs are generated using a machine learning model that was trained by learning from the inputs generated from symbolic execution of a large number of contracts. While significant progress has been made, achieving high code coverage remains an important concern for fuzzing [5, 7, 25].

In this paper, we present a fuzzing approach to test smart contracts. Our approach is centered on how to increase code coverage by addressing the following three challenges:

3.1.1. Meeting preconditions

In many functions, there are require statements written at the beginning of the function. These statements specify preconditions, i.e., conditions that must be satisfied before a function can be successfully executed. Test inputs that do not meet preconditions would cause the current transactions to be reverted.

3.1.2. State-dependent behaviors

Like parameters, state variables are also input to a function, and their values may affect the behavior of the function. Thus, a function needs to be tested at different states; otherwise, some state-dependent behaviors may not be exercised. However, unlike parameters, state variables cannot take arbitrary values due to the reachability concern. We could derive reachable states by exploring the state space of a smart contract; however, this would introduce the state explosion problem.

3.1.3. Combinatorial explosion

When the number of parameters is large, the input space of a function can be huge due to the combinatorial explosion problem. Many vulnerabilities are due to interaction between parameters. However, important combinations of values of parameters can be easily missed by pure random test generation.

To address the first challenge, our approach identifies preconditions and uses a constraint solver to generate test inputs that satisfy the preconditions. To address the second challenge, our approach uses a selective state exploration framework to derive reachable states while alleviating the state explosion problem. To address the third challenge, we combine fuzzing and Combinatorial Testing (CT) [28]. The CT tests allow us to exercise parameter interactions in a systematic manner, while fuzzing is used to discover important parameter values, which further improves the quality of CT tests.

We implemented our approach in a tool called MagicMirror. We conducted an experimental evaluation of our approach by comparing MagicMirror to two other recently published state-of-the-are smart contract fuzzing tools, sFuzz [8] and ILF [12]. Our evaluation used 2,397 real-world smart contracts [24]. The results show that MagicMirror significantly outperforms sFuzz in both code coverage and vulnerability detection. In particular, on average, MagicMirror achieves 21% higher branch coverage than sFuzz and detects significantly more vulnerable contracts. Compared to ILF, MagicMirror achieves slightly better code coverage than ILF, while detecting significantly more vulnerabilities. We note that all inputs, results, and scripts for running the experiments are saved for reproducibility and are available at [33].

The remainder of the paper is organized as follows. Section 3.2 demonstrates three major challenges of fuzzing smart contracts using real-world examples. Section 3.3 presents our approach, focusing on how to address the three challenges. Section 3.4 discusses some implementation considerations. Section 3.5 presents an experimental evaluation of our approach. Second 3.6 reviews related work. Section 3.7 concludes the paper and discusses future work.

3.2. MOTIVATION

In this section, we present motivating examples to demonstrate three major challenges in fuzzing smart contracts. Our approach is developed to address these challenges.

3.2.1. Meeting preconditions

Figure 3-1 shows a code snippet from the BecToken[‡] contract. BecToken is deployed on the Ethereum Blockchain and is implemented following the ERC-20

[‡] https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d

Token Standard [26]. Tokens are digital assets issued on the Ethereum network and could be used as currencies, like Bitcoin and Ether. *transferFrom()* allows a third account with adequate allowance to transfer tokens from one account to another.

```
contract BecToken{
  ...
  function transferFrom(address _from, address _to, uint256 _value)
  public returns (bool) {
    require(_to != address(0));
    require(_value > 0 && _value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);
    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    allowed[_from][msg.sender] =
      allowed[_from][msg.sender].sub(_value);
    Transfer(_from, _to, _value);
    return true;
  }
  ...
}
```

Figure 3-1. Code Snippet of the BecToken Contract

```
contract CryptoMinerToken {
  ...
  function buy(address _referredBy) public payable returns (uint256) {
    purchaseTokens(msg.value, _referredBy);
  }
  ...
  function purchaseTokens(uint256 _incomingEthereum, address _referredBy)
  internal returns (uint256) {
    address _customerAddress = msg.sender;
    ...
    if (
      _referredBy != 0x0000000000000000000000000000000000000000 &&
      _referredBy != _customerAddress &&
      tokenBalanceLedger_[_referredBy] >= 50e18
    ) {
      ...
    } else {...}
    if (tokenSupply_ > 0) {
      ...
    } else {...}
    ...
  }
  ...
  function sell(uint256 _amountOfTokens) onlyBagholders public {
    ...
    tokenSupply_ = SafeMath.sub(tokenSupply_, _tokens);
    tokenBalanceLedger_[_customerAddress] = SafeMath.sub(tokenBalanceLedger_[_customerAddress], _tokens);
    ...
  }
  ...
}
```

Figure 3-2. Code Snippet of the CryptoMinerToken Contract

require statements specify preconditions that must be satisfied by the inputs of a function before the function can be executed. When executing a function, if any *require* statement is not satisfied, the current transaction is reverted without exercising the actual business logic implemented in the function. To effectively test a function, it is important to generate test inputs that satisfy the *require* statements. Consider the *transferFrom()* function in Figure 3-1. Line 5 requires that the destination address, *_to*, cannot be the default address value *0x0…0*. Line 6 requires that the amount, *_value*, to be transferred from address *_from* be less than or equal to the balance of *_from*. Line 7 requires that *msg.sender* has adequate allowance to make the transfer. These preconditions could be difficult to satisfy with randomly generated inputs.

3.2.2. State-dependent behavior

Figure 3-2 shows a code snippet from the CryptoMinerToken [§]contract. CryptoMinerToken is used to help cryptocurrency miners to secure their mining assets by providing safe token transfers and exchanges. Function *purchaseTokens()* is a helper function to function *buy()*. It allows users to buy CryptoMiner tokens using Ether.

In addition to parameters, state variables are also input to a function. Some code may never be executed if the function is not executed at a particular state. Thus, a function should be fuzzed at different states to exercise the different behaviors a function could execute. One could explore the state space to derive all reachable states, which would however introduce the state explosion problem.

Consider the *purchaseTokens()* function in Figure 3-2. Since lines 14 and 18 check the values of state variables *tokenBalanceLedger_* and *tokenSupply_*,

[§] https://etherscan.io/address/0x0a97094c19295e320d5121d72139a150021a2702

reaching different branches of these *if* statements would require different values of these state variables. One could try to assign arbitrary values to these state variables to reach different branches. For example, if *tokenBalanceLedger_[referredBy]* is set to *50e18* and *tokenSupply* to *0*, a transaction would execute the *true* branch of the first *if* statement at line 11 and the *false* branch of the second *if* statement at line 18. However, if we consider the entire contract, the sum of balances in *tokenBalanceLedger_* should always be equal to *tokenSupply*, because they are always updated together in the implementation to maintain this constraint, such as the *sell()* function shown in Figure 3-2. There exists no reachable state where the balances in *tokenBalanceLedger_*are not zero while *tokenSupply_* being *0*. Thus, it is impossible to exercise the statements in both line 16 and 20 in a single transaction.

```
1  contract BTC20Exchange {
2    ...
3    modifier onlyAdmin {
4      if (msg.sender != owner && !admins[msg.sender]) throw;
5        _;
6    }
7    ...
8    function adminWithdraw(address token, uint256 amount, address user, uint256 nonce,
9    uint8 v, bytes32 r, bytes32 s, uint256 feeWithdrawal) onlyAdmin returns (bool success) {
10     ...
11     if (feeWithdrawal > 50 finney) feeWithdrawal = 50 finney;
12     if (tokens[token][user] < amount) throw;
13     ...
14     if (token == address(0)) {
15         if (!user.send(amount)) throw;
16     } else {
17         ...
18     }
19     ...
20   }
21 }
```

msg.sender:	*account1, account2, attacker1, attacker2*
token:	*account1, account2, attacker1, attacker2, 0x0*
amount:	*0, 1, 10,* 2^{256}*-2,* 2^{256}*-1*
user:	*account1, account2, attacker1, attacker2, 0x0*
nonce:	*0, 1, 10,* 2^{256}*-2,* 2^{256}*-1*
v:	*0, 1, 10, 254, 255*
r:	*0x0, 0xff, 0xffffffff, 0xff..*(32 f's)
s:	*0x0, 0xff, 0xffffffff, 0xff..*(32 f's)
feeWithdrawal:	*0, 1, 10,* 2^{256}*-2,* 2^{256}*-1,* $5x10^{10}$*,* $5x10^{10}$*+1*

Figure 3-3. Code Snippet of the BTC20Exchange Contract and Representative

Values of Parameters

3.2.3. Combinatorial explosion

Figure 3-3 shows a code snippet from the BTC20Exchange [**] contract. BTC20Exchange is a crowdfunding contract that works by issuing tokens that are purchased by contributors to finance some projects. Function *adminWithDraw()* allows contract administrators to withdraw Ethers from the contract into other accounts. There are nine parameters to *adminWithDraw()*. The execution of some branches would require multiple input variables to take some particular values simultaneously. For example, to execute the *false* branch of the *if* statement at line 12, the values taken by the input variables must satisfy the condition, i.e., certain *token, user* has more balance than the *amount* needs to be withdrawn. Assume we use the input model shown at the bottom of Figure 3-3. The identification of these values is discussed in Section 3.3.4. Enumerating all combinations of parameters and their values would yield 1.4 million tests. Due to the combinatorial explosion problem, it can be difficult for random inputs to exercise input combinations that are required to cover the different branches.

3.3. APPROACH

3.3.1. Overview

Our fuzzing framework contains four major components, as shown in Figure 3-4. Static Analysis is performed on the contract's Solidity source code before fuzzing starts to collect information that is used later in the fuzzing process. Selective State Exploration drives the fuzzing process. Combinatorial Fuzzer generates CT tests and mutates them to fuzz functions on different contract states. Constraint Solver is used by both Selective State Exploration and Combinatorial

[**] https://etherscan.io/address/0xdc468a1504fcbdf09705ee298bbec9b16ee263d0

Fuzzer to evaluate contract states and generate test inputs that satisfy preconditions. Vulnerability Analysis includes several detectors that take a transaction debugging trace as input and detects vulnerabilities that may exist in the trace.

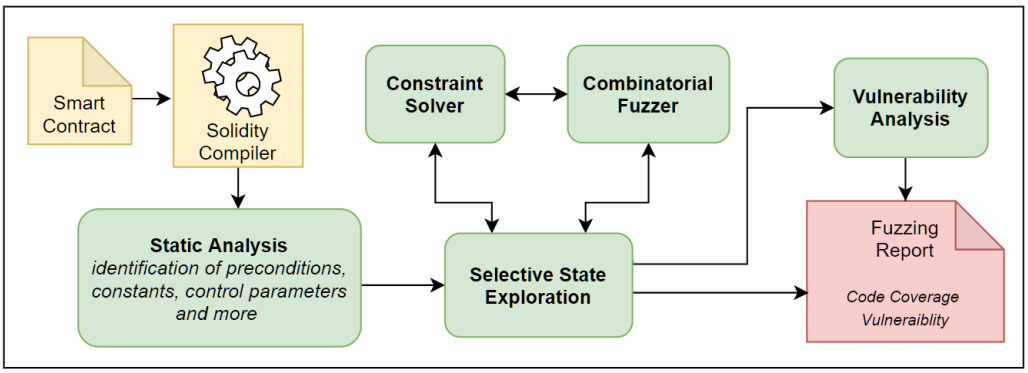


Figure 3-4. Overview of the MagicMirror Fuzzing Framework

Our approach is centered on how to address the three challenges mentioned in Section 3.2.

Meeting preconditions. To generate tests meeting preconditions, MagicMirror identifies the preconditions via lightweight static analysis. The preconditions are then parsed and encoded into constraints in a format that a constraint solver could accept. These constraints are solved to generate tests that can satisfy these preconditions.

State-dependent behavior. To exercise state-dependent behaviors, MagicMirror generates reachable states using a selective state exploration process. Starting from the initial states, i.e., the state right after a contract is deployed, we execute functions that can be executed at the states to derive successor states. This process is repeated at a subset of the successor states until the maximum exploration depth is reached. MagicMirror then restarts at the initial states and repeats the selective state exploration. The selection of a successor state for further exploration is performed to maximize the chance of increasing code coverage. This selective

state exploration process, i.e., exploring a subset of successor states instead of every successor state, helps to alleviate the state explosion problem.

Combinatorial explosion. To handle the combinatorial explosion problem, CT is used to select a subset of input combinations that achieves a combinatorial coverage criterion. Representative values for each input are predefined and identified via lightweight static analysis, e.g., constant values that appear in a branching statement, and/or discovered during fuzzing, e.g., a value that triggers new code coverage. *t-way* test generation is applied to control parameters, i.e., parameters that influence control flow decisions. For non-control parameters, we cover every one of its representative values once. Note that, we refer to both parameters from the function signature, *msg.sender*, and *msg.value* as parameters to a function. Using this method on the example shown in Figure 3-3, with *3-way* test generation for control parameters, we can generate 211 tests instead of 1.4 million tests. At the same time, still cover every combination of parameters and their values involved in the branching conditions. Lastly, we combine CT and fuzzing by using the CT tests as seed to generate additional tests, where fuzzing helps CT to extend its test set by discovering new representative values, and CT helps fuzzing to cover important input combinations.

3.3.2. Meeting Precondition

In many functions, there are *require* statements written at the beginning of the function. These *require* statements are used to check preconditions that must be satisfied for the execution of a function to be successful. Due to preconditions, many randomly generated test inputs could be rejected because the contract states do not have the necessary state variable values for a function to be successfully executed. If preconditions are not properly accounted for, it can be difficult for fuzzing to achieve high code coverage. MagicMirror addresses the precondition

issue by using constraint solving to evaluate contract states and generate valid inputs during fuzzing.

To evaluate contract states and generate precondition-satisfying tests, there are two major technical problems to handle, including (1) identifying, parsing, and encoding preconditions into constraints that a constraint solver can solve and (2) randomizing solutions produced by a constraint solver, which typically gives the same solution during multiple calls of the same constraint.

3.3.2.1. Identifying, Parsing, and Encoding Preconditions

To create precondition constraints for a function, we generate a control flow graph based on its source code. Each node in the control flow graph corresponds to a line of statement and is represented as an Abstract Syntax Tree (AST). The AST allows *MagicMirror* to parse the boolean expression inside each require statement and encode it into constraints. Note that require statements can be directly written at the beginning of a function or included in the function's modifiers.

There are more complex scenarios that make the precondition parsing more difficult. Such as when parsing a precondition with a variable other than parameters and state variables. For example, assigning the reference of a state variable *sv[0]* to a local variable *lv*, then use *lv* for evaluation in a precondition *lv.count > 0*. Such indirect accesses in preconditions are identified using static taint analysis by analyzing the Static Single Assignment (SSA) of the source code so that the constraint can be transformed into expressions containing only state variables and parameters. *lv.count > 0* would then become *sv[0].count > 0* after the transformation.

Once we identified all the preconditions, they are encoded into constraints that a constraint solver can use. To evaluate whether the precondition of a function can be satisfied on a given contract state, we encode the values of state variables

into the existing constraints. We then check whether the constraint solver can generate at least one test that satisfies the constraints.

To generate precondition-satisfying tests using the constraint solver, we will need the following information, a contract state, a CT test, and a selected parameter to be randomized. For state variables, their values are encoded into the existing constraints. For not selected parameters, their value in the CT test is also encoded into the constraints. The selected parameter's value is then solved based on the actual values of state variables and the values of the unselected parameters. Consider a constraint, $s<p_1$ && $p_1<p_2$, that involves parameter p_1, p_2 and state variable *s*. *s* is *10* in the selected state, the CT test is *(p1=0, p2=15)*, and *p1* is to be randomized. We encode *s* and p_2 in the constraint by adding the expressions $s==10$ and $p_2==15$ into the constraint. The constraint solver will then provide a solution of p_1 that is greater than ten and less than *15*.

3.3.2.2. Randomizing Solutions

Constraint solvers typically give the same solution during multiple calls (for the same constraint). We need to help the constraint solver randomize solutions better. For numerical or dynamically sized array parameters, we first use the constraint solver to identify the minimum and maximum value or array length to satisfy the constraints. We then divide the range from the minimum value to the maximum value into 100 regions and force the constraint solver to find a solution in a randomly selected region. Note that we are assuming the solution space is continuous for these parameters, i.e., all values between the maximum and minimum values are possible solutions. In some instances, this may not be true. For example, constraint a <= 10 || a >= 100 where a has the data type of uint8. The minimum and maximum value satisfying the constraint is 0 and 255. However, values between 10 and 100 would not satisfy the constraint. We handle this type of

scenario by randomly selecting different regions until a solution can be found. For other discrete typed parameters, we let the constraint solver enumerate all solutions among their representative values, we then randomly select one from the solutions.

Note that there can be scenarios where mutating only one parameter of a CT test cannot yield a precondition-satisfying test. In such scenarios, we will randomly include additional parameters to be randomized by the constraint solver until the CT test can be mutated into a test that meets the preconditions. Consider a constraint $s_1 > p_1$ && $s_2 > p_2$, where s_1 and s_2 are state variables, p_1 and p_2 are parameters. s_1 is *10*, s_2 is *20* and the CT test is (p_1*=11*, p_2*=21)*. The parameter to be mutated is p_1. However, only mutating p_1 cannot produce a test that satisfies the precondition with the given state and CT test. Hence, we also randomize p_2, which allows the constraint solver to find a solution, e.g., (p_1*=9*, p_2*=19)*.

When randomizing values for multiple parameters, instead of letting the constraint solver provide the solution of all the parameters at once, we solve for parameter values one by one, so we can randomize each parameter separately. This is because randomly selecting a value for one parameter could change the solution space for another parameter.

3.3.3. State-Dependent Behavior

Like parameters, state variables are also input to functions. Thus, a function needs to be tested at different states. Furthermore, for certain functions with more complex *require* statements, their precondition-satisfying states can be more challenging to explore because such states can often require several functions to be executed in particular orders. MagicMirror addresses the state-dependent behavior issue with a selective state exploration fuzzing strategy.

MagicMirror employs a Breadth-First Search (BFS) exploration strategy. At first, the constructor of the target contract is fuzzed. Initial contract states are

created from different contract deployments. We execute functions that can be executed at the initial states to derive successor states. This process is repeated at a subset of the successor states until the maximum exploration depth is reached. To limit state explosion, we limit both depth and width of the BFS exploration. We also limit the functions that can be fuzzed on a contract state.

When the maximum exploration depth is reached, MagicMirror will restart by fuzzing the constructor again and repeat the BFS exploration process until the timeout is reached. This is because some transactions may introduce permanent changes to contract states that would stop certain functions from meeting their preconditions. For example, many ERC-20 Token contracts can have a state variable *mintingFinished* and a function *finishMinting()*. When contracts are deployed, *mintingFinished* is set to *false*. Once *finishMinting()* is executed, *mintingFinished* is permanently set to *true*. Any functions with preconditions requiring *mintingFinished* being *false* cannot be entered again. Hence, MagicMirror can create more precondition-satisfying states for these functions by restarting the fuzzing process.

3.3.3.1. Random State Selection

During the BFS exploration, not all successor states are explored. We perform state analysis for each successor state to compute a score for weighted random selection. The score is calculated based on the code coverage of the transactions executed to produce the state and precondition-satisfiability of contract functions on the state. The precondition-satisfiability is determined using constraint solving, which was discussed in Section 3.3.2. The score is then used to select a number (equal to width) of successor states using a general weighted random selection with replacement technique.

When computing the score of a successor state, each of the following criteria of

the state will count as one point:

- Every state-modifying transaction that triggered new code coverage in the sequence of transactions that has produced the state.
- Every precondition-satisfiable function, i.e., a function having their preconditions satisfied at the given state.
- Every precondition-satisfiable function that is yet to achieve 100% opcode coverage at the given state.
- Every precondition-satisfiable function that is yet to be successfully executed at the state.

The score is computed only once when the contract state is first created. In principle, we favor contract states produced by transactions that have triggered new coverage. This is because such transactions are more likely to have modified the contract state different. We also favor states that allow MagicMirror to enter more functions that are yet to be fully covered. Doing so can increase code coverage faster.

3.3.3.2. Random Function Selection

Once a random contract state is selected, we will select one using weighted random selection among its precondition-satisfiable functions. The weight is calculated by inverting the relative opcode coverage of a function. Similar to Random State Selection, we favor functions with lower opcode coverage as well. We also limit the number of times a function is fuzzed to ensure every function can generate a relatively fair number of states and fuzzed for a fair number of times. The details of how a function is fuzzed are discussed next.

3.3.4. Combinatorial Explosion

Branching conditions often depend on a specific combination of parameter

values, which can be difficult to be covered by pure random testing. As shown in Figure 3-3, not all inputs interact with each other. Thus, testing all possible combinations of all parameters is often not necessary. The challenge is how to select a subset of the combinations that are effective for testing.

MagicMirror addresses this challenge by generating tests using CT [28]. The CT tests are then fuzzed to discover new representative values to extend the CT test set. In CT, a system is specified by a set of parameters and their representative values. A test set is a *t-way* test set if it satisfies the following property: Given any *t* parameters of a system, every combination of (representative) values of these *t* parameters is covered in at least one test in the test set. In our case, we apply CT to functions in a smart contract, where the parameters are function parameters (including *msg.sender* and *msg.value*). We note that *t-way* test generation is only applied to control parameters, i.e., parameters influencing the result of branching conditions. For non-control parameters, we cover every representative value only once.

Once a contract state and a function are randomly provided by the BFS exploration, there are four major steps to fuzz the function, including (1) identifying representative values of each parameter, (2) identifying control parameters, (3) generating combinatorial tests, and (4) mutating CT test to discover more representative values.

3.3.4.1. Identifying Parameter Representative Values

By default, we include the boundary values, near-boundary values, and common values based on the parameter's data type. The following values are included as pre-defined representative values based on parameter type:

- *intn*: min, min + 1, 0, 1, 10, max – 1, max. For example, for *int8*, its representative values are -128, -127, 0, 1, 10, 126, 127.

- *uintn*: 0, 1, 10, max – 1, max.
- *byte*: “0x0”, “0xff”.
- *bytes*: “0x0”, “0xff”, “0xffffffff”
- *bytesn*: “0x0”, “0xff”, “0xffffffff” if *n* >= 4, and “0xff” * *n*.
- *string*: “”, “Hello”.
- *bool:* true, false.
- *address:* normal account addresses, attacker contract addresses, and invalid address “0x0”.
- *arrays (fix sized)*: A sample array randomly populated with the base data type representative values. For example, for *int8[2]*, we could have [0, 126].
- *arrays (dynamically sized)*: Three sample arrays of length *zero*, *one*, and a random length, randomly populated with the base data type representative values.

In addition, we also identify constant values that are used to compare with parameters in the source code. These constant values are identified by analyzing the AST of the function source code. These constant values are added to the parameter(s) in which they were compared. There can also be an additional near-boundary value when a constant value is compared to a parameter using <, >, <= and >=. For example, in *param>10*, both *10* and the near-boundary value *11* are added to *param* as additional representative values.

3.3.4.2. Identifying Control Parameters

To identify control parameters, we analyze the function’s AST to determine the parameters used in branching conditions. We also use static taint analysis by analyzing the SSA of the function to identify parameters that indirectly influenced branching conditions.

3.3.4.3. Generating Combinatorial Test Set

```
1  contract BTC20Exchange {
2    ...
3    modifier onlyAdmin {
4      if (msg.sender != owner && !admins[msg.sender]) throw;
5        _;
6    }
7    ...
8    function adminWithdraw(address token, uint256 amount, address user, uint256 nonce,
9    uint8 v, bytes32 r, bytes32 s, uint256 feeWithdrawal) onlyAdmin returns (bool success) {
10     ...
11     if (feeWithdrawal > 50 finney) feeWithdrawal = 50 finney;
12     if (tokens[token][user] < amount) throw;
13     ...
14     if (token == address(0)) {
15         if (!user.send(amount)) throw;
16     } else {
17         ...
18     }
19     ...
20   }
21 }
```

Combinatorial Test Set

msg.sender	token	amount	user	feeWithdrawal	nonce	v	r	s
0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1
1	0	1	0	1	*	*	*	*
1	1	0	1	0	*	*	*	*
*	0	0	1	1	*	*	*	*
*	1	1	0	0	*	*	*	*

t-way test of control parameters

populated non-control parameter values

* represents don't care values that can take any rep value of the parameter.

Figure 3-5. Combinatorial Test Set Generation for Function adminWithdraw()

To achieve higher code coverage, combinations of control parameters and their values are more important than non-control parameters. t-way test generation is only applied to control parameters. In Figure 3-5, we illustrate how to generate a t-way combinatorial test set for function adminWithdraw(). For simplicity, we will only include two representative values (0 and 1) for each parameter and use 2-way

test generation. First, we generate the 2-way test set for the control parameters msg.sender, token, amount, user, and feeWithdrawal. Nrxt, we add the representative values of the non-control parameters into the 2-way test set to complete the CT test set for function adminWithdraw().

3.3.4.4. Mutating CT Test

Not every CT test will execute unique scenarios. Some CT tests could share similar behaviors. Instead of mutating every CT test, we want to identify tests that covered new branches and/or reached deeper and hard-to-reach branches. After executing the CT tests from step (3), we analyze the branch coverage achieved by the CT tests. Next, we sort the tests by the number of covered branches of each test, from large to small, because tests covering more branches likely indicate deeper branches were reached. Lastly, from the first test in the sorted list, we select tests covering branches that were not covered by any previous tests in the list. As a result, we have a small subset of CT tests that can reach the same depth of branches, and have the same branch coverage as the origianl CT test set.

In addition to predefined representative values based on parameter data type and representative values identified using static analysis, we use fuzzing to discover new representative values by mutating the CT tests. For each selected CT test, we mutate one control parameter at a time. We maintain the values of other parameters, so only a small number, or ideally one, branching condition involving the control parameter, is flipped. This strategy helps MagicMirror increase the likelihood of exploring new branches and discovering new representative values. We do not mutate non-control parameters since they are unlikely to trigger new code coverage.

To mutate the value of a selected parameter, we have two ways, using constraint solver or random generation. When a parameter is used in preconditions, its value is randomized using a constraint solver to ensure the mutated tests will not

be rejected by the preconditions. The details of how a constraint solver randomizes parameter values were discussed in Section 3.3.2. For parameters that do not appear in preconditions, their values are either randomly selected from their existing representative values (i.e., *bool*, *address*, *string*, *byte*/*bytes, bytesn* and *fixed sized array*), or randomly generated within the valid range of their data type (*uintn*, *intn*, and *dynamically sized array*). The number of times a control parameter in a CT test is mutated is configurable.

If any new representative values are discovered after the mutation of CT tests, the initial CT test set will be extended to include the new representative values. We will take these extended CT tests and repeat the process at step (1) until no additional representative values are discovered.

3.3.5. Vulnerability Detection

Our fuzzing approach is independent from vulnerability detection. Thus, in principle, we could incorporate third-party vulnerability detectors. As discussed in Section 3.4.6, we adopted 14 vulnerability detectors from sFuzz [8] and ILF [12].

3.4. IMPLEMENTATION

In this section, we discuss some major decisions in the implementation of MagicMirror, including static analysis using Slither, constraint solving using Z3 [35], CT test generation using ACTS [13], code coverage computation, test execution using a custom Geth EVM [34], and vulnerability detectors. The source code and a ready-to-use Docker [32] image of MagicMirror is available at [33].

3.4.1. Static Analysis with Slither

We use Slither, a static analyzer for Solidity programs, to analyze the source code of a contract. In particular, we utilize the control flow graph, AST of the source

code, and SSA from Slither to identify the information we need as discussed below:

3.4.1.1. Identify Preconditions

Preconditions need to be written in terms of state variables and function parameters. However, a *require* statement may involve internal variables, which need to be rewritten. To avoid the need for symbolic execution, MagicMirror only handles *require* statements at the beginning of a function. A *require* statement is considered to be at the beginning of a function if it appears before any statement that changes a state or local variable or any branching statement that is not a *require* statement.

3.4.1.2. Identify Implicit Parameters

Since implicit parameters, *msg.sender* and *msg.value* do not appear in a function's signature, we gather this information by analyzing the AST of the function. Furthermore, we also identify their indirect usage in inner function calls to other functions.

3.4.1.3. Identify Control Parameters

By analyzing the control flow graph. AST and SSA of a function, we can identify parameters used in branching conditions as control parameters.

3.4.1.4. Identify Constants

In addition to handling the direct comparison of a parameter to a constant value, constant values can also appear in type conversions. If not handled correctly, the value may be missed or incorrectly assigned. We detect this type of operation using the AST of the function and parse the constant value to its destined data type.

3.4.2. Constraint Solving with Z3

We use Z3 [35] to perform constraint solving. As discussed in Section 3.3.2, constraint solving is used to identify states and input parameter values that satisfy preconditions. The preconditions identified using Slither are parsed and encoded as Z3 constraints. During the creation of Z3 constraints, we create Z3 variables to represent different state variables and input parameters. Z3 provides data types that can handle Solidity primitive types such as int, bool, bytes, etc. However, some data types in Solidity, e.g., mapping, user-defined constructs, require creation of customized data types in Z3. We utilize static typing information provided by Slither to automatically create custom Z3 data types to handle non-primitive type variables.

When loading state variables' values into constraints, we call the state variable getters to obtain the values. For state variables that are arrays or mappings, obtaining their values requires additional effort. Solidity compilers do not generate getters that can return the full content of an array or mapping structure. Instead, an index must be provided to get the element at a specific position of an array or mapping structure. For an array, we repeatedly call the getter function with the index starting from 0, then add the value returned by the getter function into the Z3 constraint. We increase the index until the getter function fails.

For a mapping, obtaining all the values in the mapping can be more challenging because mappings do not store their keys. Instead, only the value is stored at the memory address calculated by the SHA3 hash of the key. If no value is ever written into a position, accessing the position would return the default value. On the other hand, a transaction could intentionally write the default value into a position. This means we cannot distinguish the two cases when the default value is returned. To address this problem, we keep track of the keys of a mapping structure whose associated values were previously modified. If a key is non-numeric, this

problem does not exist as non-numeric parameters are only fuzzed with their predefined values. We keep track of all possible values that could be assigned to the key during state exploration for a numeric key.

3.4.3. Combinatorial Test Generation using ACTS

We use Automated Combinatorial Testing for Software (ACTS) [13], a test generation tool for constructing *t-way* CT test sets. ACTS is implemented using Java. To interact with ACTS, we implemented a custom wrapper using Py4j to launch ACTS as a service, so MagicMirror can interact with ACTS as a client. The input to ACTS contains control parameters for the function, *t-way* strength, and representative values of control parameters. When generating the CT test set, the default strength of control parameters is set to 2.

3.4.4. Test Execution with Custom Geth EVM

Our transaction execution backend is implemented on top of Go Ethereum (Geth EVM). This idea is inspired by the backend implementation of ILF [12]. In the wrapper of the Geth EVM, we implemented contract state management, i.e., taking/restoring snapshots of contract states. With the wrapped Geth EVM built into a shared library, MagicMirror can execute transactions natively via inter-process communication (IPC) without communication delays instead of performing RPC calls.

3.4.5. Code Coverage Computation

Due to the existing tools [37, 38] not being compatible with our implementation, we generate code coverage reports on our own based on the contract bytecode. The code coverage report contains both edge coverage and opcode coverage for the entire contract and individual functions. We identify edges in the bytecode using the bytecode control flow graph generated by Vandal [21], a

static program analysis framework for Ethereum smart contract bytecode. However, Vandal may not detect some edges because some destinations of JUMP or JUMPI instructions in the bytecode could be computed dynamically at runtime, and are thus not known at compile time. If any edge that Vandal does not detect is executed, the edge is added to the set of all possible edges. For edge and opcode coverage on individual functions, we utilize the source mapping produced from the Solidity compiler, and control flow graph produced from Vandal, to map bytecode segments to a specific function.

3.4.6. Vulnerability Detectors

MagicMirror, sFuzz and ILF all analyze debug trace of transactions to detect vulnerabilities. We adopted nine detectors from sFuzz, they are strictly translated from C++ to Python. We also adopted five detectors from ILF, we were able to use them directly without translation because they were also implemented in Python.

For triggering vulnerabilities requiring interactions between contracts, MagicMirror deploys attacker contracts to interact with target contracts. Any test can be sent by a normal wallet account or via an attacker contract. The attacker contract contains three functions, *AgentCallWithoutValue()*, *AgentCallWithValue()* and the *fallback* function. For the two agent call functions, besides calling the target contract with the provided call data, they also save the call data in a state variable for reentrancy attack. When sending transactions to target contract using agent calls, if the target contract tries to send ether to the attacker contract, the attacker contract's *fallback* function will resend the previously received call data to create a reentrancy scenario.

3.5. EXPERIMENTS

Our experiments are designed to evaluate the effectiveness of MagicMirror in terms of code coverage and vulnerability detection abilities. In particular, we compare MagicMirror to two recently published fuzzing tools, sFuzz [8] and ILF [12].

3.5.1. Research Questions

Our experiments are designed to ask the following two research questions:

RQ1: How does MagicMirror perform in terms of code coverage?

RQ2: How does MagicMirror perform in terms of vulnerability detection?

To answer **RQ1**, we compare the code coverage achieved by MagicMirror to sFuzz and ILF within the same amount of time. To answer **RQ2**, we compare the number of vulnerable contracts detected by MagicMirror to sFuzz and ILF within the same amount of time. MagicMirror implements all vulnerability detectors from sFuzz, and five out of seven vulnerability detectors from ILF.

3.5.2. Subjects

In the experiments, we used 2,397 smart contracts as our subjects. These contracts are selected from 1,838 smart contract source files [24]. These contract files were randomly collected from Etherscan [30] to evaluate another smart contract analysis tool. The 1,838 source files require Solidity compiler versions ranging from 0.4.0 to 0.5.10. Many of these files contain multiple contracts. Our selection excludes the following types of contracts: library contracts, interface contracts, abstract contracts, and contracts inherited by other contracts.

We compare MagicMirror to two state-of-the-art fuzzing tools that are publicly available, i.e., sFuzz [8] and ILF [12], using their latest release [31, 39] on

GitHub. sFuzz uses an AFL [23] like fuzzing strategy to fuzz smart contracts. ILF uses machine learning (ML) to fuzz smart contracts. The ML model is trained by a symbolic execution expert executing on a training dataset.

Note that none of the three tools could execute all subject contracts due to different kinds of exceptions. Out of the 2,397 subject contracts, MagicMirror reported results for 2,276 contracts, ILF reported results for 2,005 contracts, sFuzz reported results for 1,264 contracts. When comparing between tools, we only compare results on contracts where both tools reported results. The exceptions we encountered from ILF and sFuzz have been reported to the authors of sFuzz and ILF.

3.5.3. Metrics

3.5.3.1. Code Coverage

When answering **RQ1**, *MagicMirror* reports both edge and opcode (instruction) coverage based on the contract bytecode (both deploy-time and runtime bytecode). However, sFuzz and ILF report their results differently.

sFuzz code coverage: sFuzz reports branch coverage based on the contract bytecode. However, based on communication with the first author, sFuzz only analyzes non-constant functions and only recognizes branches with *JUMPI* instructions that can be mapped to an *if/while/require/assert* statement. To fairly compare with sFuzz, MagicMirror uses identical contract bytecode that sFuzz used. MagicMirror's edge coverage result is also filtered to contain only branches included in the sFuzz's result. To identify branches recognized by sFuzz, we added a few lines of code [33] to sFuzz for logging. The first author of sFuzz has confirmed that the changes made to sFuzz will not impact its performance and ability to detect vulnerabilities.

ILF code coverage: ILF reports opcode and basic block coverage based on runtime bytecode only. Additionally, ILF removes the metadata appended to the end of the runtime bytecode by the compiler. Hence, after disassembling the runtime bytecode, ILF would report fewer total opcodes in the coverage. However, after confirming with the first author, this would not affect the number of covered opcodes. To fairly compare with ILF, we will use the total number of opcodes disassembled from MagicMirror as the denominator for computing ILF's opcode percentage coverage.

3.5.3.2. Vulnerabilities

Since sFuzz analyzes non-constant functions only, when comparing with sFuzz, we exclude vulnerabilities detected in constant functions from *MagicMirror*. For ILF, constructors are not analyzed by ILF. When comparing with ILF, we exclude vulnerabilities detected in constructor functions from *MagicMirror*.

Note that MagicMirror, sFuzz, and ILF all analyze transaction debug trace for vulnerability detection. MagicMirror implements all sFuzz detectors in the same logic. For ILF, we could copy and paste five out of seven ILF detectors into MagicMirror without translation because ILF detectors were also written in Python. The two excluded detectors, *Leaking* and *Suicidal*, require a fuzzing strategy that is unique to ILF, and are thus excluded in MagicMirror. For false positives, since MagicMirror has identical detectors implemented in sFuzz and ILF, we do not investigate the false positive vulnerabilities detected between these tools.

3.5.4. Procedure

Due to limited resources and the large number of subject contracts, we execute MagicMirror, sFuzz, and ILF with a 15-minutes timeout on each subject contract. We run the experiment three times and report the average as the result.

For MagicMirror, the contract source code is provided as input, with other user-configurable options left as default. In particular, the default value of *t-way* test strength is two.

For sFuzz, we provide a JSON file consists of compiled contract information and the source file. In the JSON file, the compiled contract information is identical for both sFuzz and MagicMirror. The contract source file is required by sFuzz to identify branches with source mapping.

For ILF, its input consists of a trained ML model, the contract source code, and a contract deployment configuration. For the ML model, ILF uses the default model provided in the GitHub repository [31]. ILF also provides a script to automate the compilation and deployment transaction generation process. To ensure ILF and MagicMirror receive identical compilation results, instead of using the script, we generate the compilation information and provide it to ILF. For contract deployment, ILF does not fuzz constructor. A contract deployment must be provided to ILF. To fairly compare with ILF, we randomly generate a contract deployment for ILF, and we modified MagicMirror to only fuzz the contract based on the identical contract deployment instead of fuzzing the constructor.

Table 3-1. Statistical Results for ILF Executed on Multiple Contract Deployments

stdev < 1%	**1% ≤ stdev < 5%**	**stdev ≥ 5%**
36	11	3
Min	**Avg**	**Max**
0%	1.62%	21.90%

Due to limited resources, we are unable to execute ILF multiple times on different deployments. We acknowledge that different deployments may cause ILF and MagicMirror to produce different results. To study how this factor may impact

ILF's result, we randomly selected 50 contracts from the subject contracts for ILF to execute, ILF is provided with five random contract deployments for each contract, and each deployment is executed 15 minutes. Table 3-1 shows the statistics for the opcode coverage standard deviation among five deployments for the 50 contracts. We see that ILF would achieve similar opcode coverage results in most contracts when different deployments are provided. On average, the opcode coverage standard deviation of different deployments fluctuates around 1.62%.

Finally, all experiments are carried out on Docker containers with three cores and 25GB RAM. The Docker containers are hosted on a Windows 10 workstation with two Intel Xeon Platinum 8180 processors with 56 2.5GHz cores; and 512GB memory. We note that all inputs, results, and scripts for running the experiments are saved for reproducibility and are available at [33].

3.5.5. Results for RQ1

In Table 3-2 and Figure 3-6, we compare the branch coverage of MagicMirror to sFuzz on 1,225 contracts where both tools reported coverage. As previously mentioned, to objectively compare with sFuzz, MagicMirror's result has been filtered to contain only branches that were identified by sFuzz. On average, MagicMirror can achieve 21% more branch coverage than sFuzz, as shown in Table 3-2. Figure 3-6 shows the differences in the number of covered branches between MagicMirror and sFuzz. The vertical axis represents $MagicMirror - sFuzz$, i.e., the number of branches covered by MagicMirror minus the number of branches by sFuzz. The horizontal axis represents individual contracts sorted by $MagicMirror - sFuzz$. Note that each data point on the horizontal axis is a contract that has a particular value of $MagicMirror - sFuzz$. Among the 1,225 contracts, MagicMirror achieved higher branch coverage than sFuzz in 930 contracts, the same branch coverage in 166 contracts, and lower branch coverage in 129 contracts.

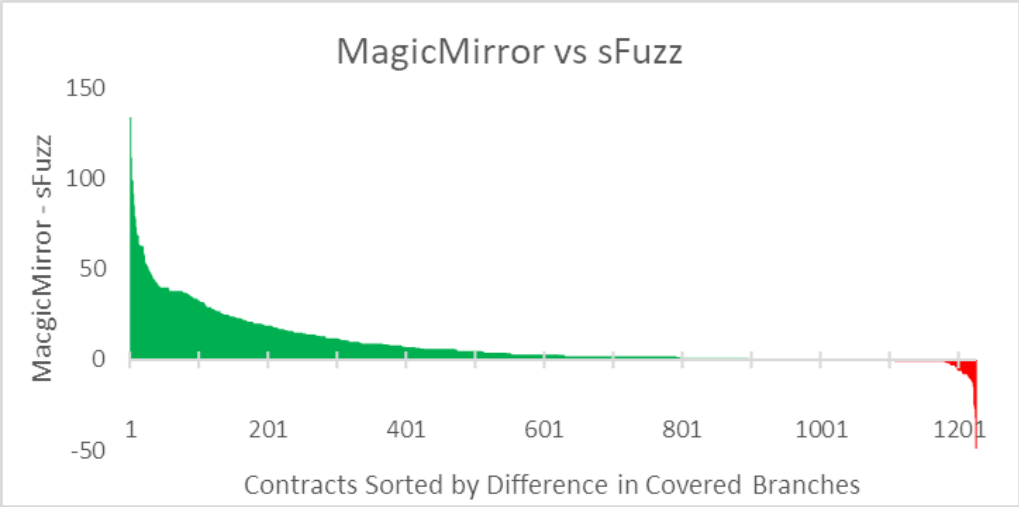


Figure 3-6. MagicMirror and sFuzz Covered Branches Comparison

Table 3-2. Branch Coverage for MagicMirror and sFuzz

	MagicMirror	***sFuzz***
Min	1.35%	0.76%
Median	90.00%	66.67%
Mean	82.75%	61.67%
Max	100.00%	100%

MagicMirror significantly outperforms sFuzz in terms of branch coverage. We intended to conduct further investigation to explain the results. However, sFuzz only reports branch coverage for the entire contract, without detailed information about the coverage, e.g., which function has lower coverage, or which branches/opcodes were covered or uncovered.

In Table 3-3 and Figure 3-7, we compare the opcode coverage of MagicMirror to ILF on 1,986 contracts where both tools reported coverage. In our experiments, MagicMirror achieves slightly better code coverage than ILF. On average, MagicMirror can achieve about 1.7% higher opcode coverage than ILF, as shown in Table 3-3. The difference in opcode coverage achieved for contracts between MagicMirror and ILF is shown in Figure 3-7. Among the 1,986 contracts, MagicMirror achieved higher branch coverage than ILF in 1209 contracts, same

branch coverage in 161 contracts, and lower branch coverage in 616 contracts, as shown in Figure 3-7.

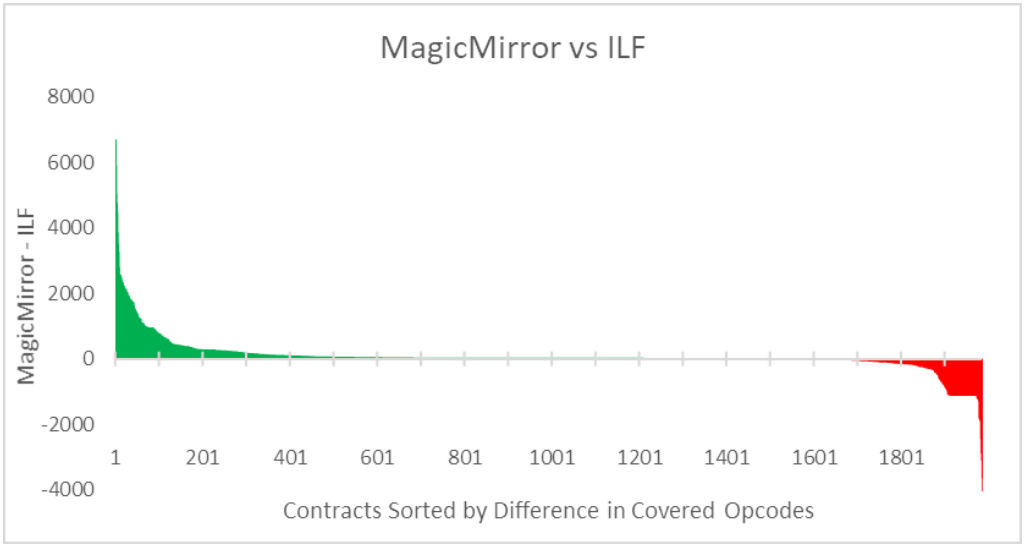


Figure 3-7. MagicMirror and ILF Covered Branches Comparison

Table 3-3. Opcode Coverage for MagicMirror and ILF

	MagicMirror	***ILF***
Min	2.80%	0.95%
Median	92.59%	91.53%
Mean	82.81%	80.10%
Max	99.84%	99.78%

3.5.6. Results for RQ2

In Table 3-4, we present the number of contracts flagged with vulnerabilities by MagicMirror and sFuzz. Note that the result of MagicMirror has been filtered to contain only vulnerabilities detected in non-constant functions, same as sFuzz. As previously shown in Table 3-2, MagicMirror outperforms sFuzz significantly in terms of branch coverage. For vulnerability detection, MagicMirror is also detecting significantly more vulnerabilities than sFuzz. Among the nine vulnerability detectors implemented in MagicMirror and sFuzz, MagicMirror

detects more vulnerable contracts in seven out of nine vulnerability detectors. When compared to sFuzz, MagicMirror can generally detect more vulnerabilities.

Table 3-4. Vulnerability Detection for MagicMirror and sFuzz

	MagicMirror	***sFuzz***
Gasless Send	241	187
Dangerous Delegate Call	24	19
Exception Disorder	29	17
Freezing Ether	22	17
Reentrancy	16	14
Block Number Dependency	8	17
Time Dependency	35	37
Integer Overflow	440	212
Integer Underflow	172	151

Table 3-5. Vulnerability Detection for MagicMirror and ILF

	MagicMirror	**ILF**
Locking	7	9
Block Dependency	42	40
Unhandled Exception	29	10
Controlled Delegatecall	19	7
Reentrancy	58	5

In Table 3-5, we present the number of flagged contracts with vulnerabilities by MagicMirror and ILF. As shown in Table 3-5, MagicMirror and ILF have a similar result for *Locking* and *Block Dependency*. For the other vulnerabilities, MagicMirror performs significantly better than ILF, which we investigated further.

For *Controlled Delegatecall*, we investigated into the 12 contracts MagicMirror flagged that ILF did not. Because ILF does not provide additional coverage information, e.g., exactly which opcode is covered or not, we could not conduct a thorough investigation. However, on average, MagicMirror achieved

78.83% higher opcode coverage in these 12 contracts than ILF. The reason is likely because ILF was unable to execute the opcodes that would have triggered the vulnerabilities.

For *Unhandled Exception* and *Reentrancy* vulnerabilities, they both require interactions between contracts to trigger them. MagicMirror was able to flag many more contracts because MagicMirror deploys attacker contracts to interact with target contracts. In contrast, ILF only sends transactions via normal wallet accounts, and thus cannot effectively detect *Unhandled Exception* and *Reentrancy* vulnerabilities.

3.6. Related Work

In this section, we briefly overview existing work on testing smart contracts, including fuzzing, symbolic execution, and other static analysis-based approaches.

Fuzzing. ContractFuzzer [10] is an unguided fuzzer. ContractFuzzer predefines a set of default values for each data type based on different lengths in their byte form, e.g., *0x0* and *0xff* as for *uint* with length *8*. When fuzzing a function, for each parameter, it randomly selects a valid length based on data type, e.g., *16* for *uint256* with predefined value 0x0 and 0xffff. It then takes the predefined values of each parameter, enumerates all the combinations of the parameters' values, and randomly selects some tests to execute. In contrast, MagicMirror is guided by its selective state exploration process. MagicMirror identifies control parameters and uses CT to execute only a subset of combinations. In addition, MagicMirror uses fuzzing to discover important parameter values.

sFuzz [8] is a coverage-guided fuzzer built on top of AFL [23], and mutates transaction input values using bit/byte flip, simple arithmetic, and other operations. Furthermore, sFuzz uses an adaptive approach to measure the distance of the

current input value to the value that would flip a branching condition to explore hard-to-reach statements. In contrast, MagicMirror uses a selective state exploration framework, where constraint solving is used to generate tests that satisfy preconditions and CT is used together with fuzzing to make the testing process more efficient. MagicMirror achieved higher code coverage than sFuzz in our experiments as discussed in Section 3.5.5.

ILF [12] combines Machine Learning and fuzzing to fuzz contracts based on a specific contract deployment. ILF utilizes imitation learning by training a neural network using test sequences produced by symbolic execution of a large number of contracts. The neural network is then used to generate test sequences to fuzz new contracts. The effectiveness of ILF depends on the quality of training contracts, especially on whether they are a good representation of real-life contracts. In contrast, MagicMirror does not require the user to provide contract deployment configurations. MagicMirror does not have the notion of model training.

Echidna [9] uses grammar-based fuzzing based on contract ABI to falsify user-defined properties, i.e., unit tests that check certain properties of user interest. After investigating online documentation [41], and multiple articles [9, 29, 42, 43] of Echidna. It is unclear how exactly Echidna generates random inputs. In contrast, MagicMirror automatically checks for vulnerabilities without requiring the user to implement property checkers.

Harvey [11] uses the Secant method [40], similar to sFuzz, to predict inputs by measuring the linear distance of existing inputs on how far they are from negating a branching condition. To handle the state-dependent behavior, Harvey fuzzes transaction sequences in a targeted and demand-driven way, assisted by an aggressive mode that directly fuzzes the persistent state of a smart contract. In contrast, MagicMirror uses constraint solving to generate precondition-satisfying tests, which can also easily handle non-linear relationships defined in preconditions.

To handle state-dependent behaviors, MagicMirror employs the selective state exploration to generate diverse contract states without introducing the state explosion problem.

Symbolic Execution. Oyente [14] analyzes a smart contract by symbolically executing individual functions. However, Oyente does not deal with the state reachability issue. That is, the states it uses to execute a function may not be reachable, which causes false positives. MAIAN [22] and Osiris [19] improve Oyente to reduce the number of false positives. MAIAN uses inter-procedural symbolic analysis combined with concrete validation to address the state reachability issue. Osiris focuses exclusively on detecting integer bugs and uses taint analysis to reduce false positives. Mythril [15] uses symbolic execution and its concolic models to check for a variety of vulnerabilities. Mythril concretize symbolic variables on demand to verify reachability and solve for path constraints requiring concrete values. In general, symbolic execution suffers from the path explosion problem. Also, path conditions collected during symbolic execution could be difficult to solve. MagicMirror also uses constraint solving, but only for preconditions that are typically much simpler than path conditions.

Other Static Analysis Approaches. MadMax [18] uses the bytecode level control flow graph, IR, and rules created using the Datalog language to identify gas-related vulnerabilities. Securify [17] extracts semantic information based on a contract's dependency graph and checks compliance and violation patterns defined in its domain-specific language. When compared to these approaches, MagicMirror is a dynamic approach in that it executes smart contract functions with concrete tests to detect vulnerabilities.

3.7. CONCLUSION

In this work, we present a novel approach that combines the power of

constraint solving, selective state exploration, CT, and fuzzing to test smart contracts effectively. MagicMirror requires access to the source code of smart contracts. Many real-world contracts have their source code publicly available on Etherscan. This is especially true for contracts that interact with public users. Public access to the source code allows involved parties to inspect the contract, increasing their confidence about contract execution. We note that all the contracts used in our experiments are real-world contracts, and their source code is publicly available. In our experiment, we compared MagicMirror to two state-of-the-art smart contract fuzzing tools, sFuzz and ILF. Our experiment results show that MagicMirror performs better than sFuzz and ILF on both code coverage and vulnerability detection abilities. We note that all inputs, results, and scripts for running the experiments are saved for reproducibility and are available at [33].

In the future, we plan to add more features and optimize MagicMirror. Currently, MagicMirror cannot access non-public state variables' values, because MagicMirror relies on the getter functions automatically created for public state variables to retrieve state variable values. We plan to access their values directly from contract storage instead of relying on getters. This would make it possible to access the values of non-public state variables. We also plan to add oracles to detect more types of vulnerabilities, e.g., Suicidal contracts allowing anyone to destruct, arbitrary send vulnerability where external contracts can call the target contract to send Ethers to an arbitrary address. Finally, we plan to perform static analysis of bytecode instead of source code to obtain information needed by MagicMirror, e.g., identifying preconditions. This would allow MagicMirror to be used when source code is not available.

3.8. REFERENCES

1. P. Daian, "Analysis of the DAO exploit," Hacking, Distributed, 18-Jun-

2016. [Online]. Available: https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. [Accessed: 01-Oct-2020].

2. PeckShield, “New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299),” blog.peckshield.com, 22-Apr-2018. [Online]. Available: https://blog.peckshield.com/2018/04/22/batchOverflow/. [Accessed: 01-Oct-2020].
3. PeckShield, “New ceoAnyone Bug Identified in Multiple Crypto Game Smart Contracts (CVE-2018–11329),” Medium, 21-May-2018. [Online]. Available: https://medium.com/@peckshield/new-ceoanyone-bug-identified-in-multiple-crypto-game-smart-contracts-cve-2018-11329-898cdceac7e0. [Accessed: 01-Oct-2020].
4. PeckShield, “New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376),” blog.peckshield.com, 25-Apr-2018. [Online]. Available: https://blog.peckshield.com/2018/04/25/proxyOverflow/. [Accessed: 01-Oct-2020].
5. López Vivar, A. T. Castedo, A. L. Sandoval Orozco, and L. J. García Villalba, “An Analysis of Smart Contracts Security Threats Alongside Existing Solutions,” Entropy, vol. 22, no. 2, p. 203, Feb. 2020.
6. P. Anderson, “The use and limitations of static-analysis tools to improve software quality,” CrossTalk: The Journal of Defense Software Engineering, vol. 21, no. 2, p. 18–21, 2008.
7. N. Stephens et al., “Driller: Augmenting fuzzing through selective symbolic execution,” Proc. Symp. Netw. Distrib. Syst. Secur. (NDSS), pp. 1-16, 2016.
8. T. Nguyen, L. Pham, J. Sun, Y. Lin and M. Tran, “sFuzz: An efficient adaptive Fuzzer for solidity smart contracts,” Proc. 42nd Int. Conf. Softw. Eng. (ICSE), Jul. 2020.
9. G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, “Echidna: effective,

usable, and fast fuzzing for smart contracts," Proc, 29th Int. Symp. Software Testing and Analysis (ISSTA), Jul. 2020

10. Jiang, Y. Liu and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE), pp. 259-269, Sep. 2018.
11. V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts, " arXiv preprint, 2019, [online] Available: https://arxiv.org/abs/1905.06944.
12. J. He, M. Balunovic, N. Ambroladze, P. Tsankov and M. T. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), p. 531-548, Nov. 2019.
13. L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Acts: A combinatorial test generation tool," in 6th International Conference on Software Testing, Verification and Validation (ICST), p. 370-375, 2013.
14. L. Luu, D.-H. Chu, H. Olickel, P. Saxena and A. Hobor, "Making smart contracts smarter," Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), p. 254-269, Oct. f2016.
15. Mythril GitHub Repository, [Online]. Available: https://github.com/ConsenSys/mythril. [Accessed: 01-Oct-2020].
16. Jiang, A. Wang, Z. Zheng, W. K. Chan, and N. Li, "Artemis: An improved smart contract verification tool for vulnerability detection", CCF China Blockchain Conf. (CCF CBCC), p. 1–17, 2019.
17. P. Tsankov, A. Dan, D. Cohen, A. Gervais, F. Buenzli and M. Vechev, "Securify: Practical security analysis of smart contracts," Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), p. 67-82, Jan. 2018.
18. N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts",

Proc. ACM Program. Lang, vol. 2, no. OOPSLA, p. 116:1-116:27, Oct. 2018,.

19. F. Torres, J. Schütte and R. State, “Osiris: Hunting for integer bugs in ethereum smart contracts,” Proc. 34th Annu. Comput. Secur. Appl. Conf., p. 19-34, Dec. 2018.
20. J. Feist, G. Grieco and A. Groce, “Slither: a static analysis framework for smart contracts,” Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), p. 8-15, 2019.
21. L. Brent et al., “Vandal: A scalable security analysis framework for smart contracts,” arXiv preprint, 2018, [online] Available: https://arxiv.org/abs/1809.03981.
22. Nikolić, A. Kolluri, I. Sergey, P. Saxena and A. Hobor,” “Finding the greedy prodigal and suicidal contracts at scale,” Proceedings of the 34th Annual Computer Security Applications Conference, p. 653-663, 2018
23. M. Zalewski, American Fuzzy Loop, [online] Available: http://lcamtuf.coredump.cx/afl/
24. C. Peng, S. Akca and A. Rajan, “SIF: A Framework for Solidity Contract Instrumentation and Analysis,” 26th Asia-Pacific Software Engineering Conference (APSEC), p. 466-473, 2019
25. G. Klees, A. Ruef, B. Cooper, S. Wei and M. Hicks, “Evaluating fuzz testing,” Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), p. 2123-2138, Jan. 2018.
26. Ethereum Improvement Proposals, “EIP-20: ERC-20 Token Standard,” eips.ethereum.org, 19-Nov-2018. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20. [Accessed: 01-Oct-2020].
27. P. Hegedus, “Towards Analyzing the Complexity Landscape of Solidity Based Ethereum Smart Contracts,” 2018 IEEE/ACM 1st International

Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Gothenburg, Sweden, 2018, pp. 35-39.

28. Y. Lei, R. Kacker, D. R. Kuhn, V. Okun and J. Lawrence, "IPOG: A General Strategy for T-Way Software Testing," 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), Tucson, AZ, 2007, pp. 549-556
29. Echidna, a smart fuzzer for Ethereum, [Online]. Available: https://blog.trailofbits.com/2018/03/09/echidna-a-smart-fuzzer-for-ethereum/.
30. Etherscan, [Online]. Available: https://etherscan.io/.
31. ILF GitHub Repository, [Online]. Available: https://github.com/eth-sri/ilf/tree/9e8e3015a48783634658c8e748f113d2da2628c7.
32. Docker, [Online]. Available: https://www.docker.com/.
33. MagicMirror, [Online]. Available: https://smart-explorer.gitbook.io/smart/.
34. Geth GitHub Repository, [Online]. Available: https://github.com/ethereum/go-ethereum.
35. L. Moura and N. Bjørner, "Z3: An Efficient SMT Solver", TACAS 2008. Lecture Notes in Computer Science, vol 4963. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78800-3_24
36. Py4j, [Online]. Available: https://www.py4j.org/
37. Solidity-coverage, [Online]. Available: https://github.com/sc-forks/solidity-coverage
38. Truffle development suite, [Online]. Available: https://www.trufflesuite.com/
39. sFuzz GitHub Repository, [Online]. Available: https://github.com/duytai/sFuzz/tree/eb690d4287af4c7dc0ecfce7447e4b4462775d55.
40. M. Avriel, "Nonlinear Programming: Analysis and Methods," Prentice Hall.

pp. 220–221.

41. Echidna GitHub Repository, [Online]. Available: https://github.com/crytic/echidna/tree/31865b1942733ad285f8d305db3c5ff3e3a193a1.
42. Using Echidna to test a smart contract library, [Online]. Available: https://securityboulevard.com/2020/08/using-echidna-to-test-a-smart-contract-library/.
43. Smart Contract Fuzzing, how to find edge cases with echidna, [Online]. Available: https://medium.com/coinmonks/smart-contract-fuzzing-d9b88e0b0a05.

Chapter 4. Conclusion

In this dissertation, we present two novel approaches for improving the testing and debugging of emerging software applications, big data applications, and smart contracts.

For big data application, we presented a framework to provide developers with method-level tests that were recorded from a failed system-level execution with the original dataset. These method-level tests preserve a given coverage criterion, e.g. edge, edge-pair, and edge-set coverage, and thus are likely to reproduce the failure observed at the system level. The binary reduction is used to further reduce method-level tests with large input. The set of method-level tests that are provided by our approach could help developers to effectively debug suspicious methods against properties of the original input dataset, and significantly reduce time and effort required for debugging big data applications.

For smart contracts, we present a novel approach that combines the power of constraint solving, selective state exploration, CT, and fuzzing to test smart contracts effectively. MagicMirror requires access to the source code of smart contracts. Many real-world contracts have their source code publicly available on Etherscan. This is especially true for contracts that interact with public users. Public access to the source code allows involved parties to inspect the contract, increasing their confidence about contract execution. We note that all the contracts used in our experiments are real-world contracts, and their source code is publicly available. In our experiment, we compared MagicMirror to two state-of-the-art smart contract fuzzing tools, sFuzz and ILF. Our experiment results show that MagicMirror performs better than sFuzz and ILF on both code coverage and vulnerability detection abilities.