A Method Level Test Generation Framework for Debugging Big Data Applications

Huadong Feng
Department of Computer Science & Engineering
The University of Texas at Arlington
Arlington, USA
huadong.feng@mavs.uta.edu

Jaganmohan Chandrasekaran
Department of Computer Science & Engineering
The University of Texas at Arlington
Arlington, USA
jaganmohan.chandrasekaran@mavs.uta.edu

Yu Lei
Department of Computer Science & Engineering
The University of Texas at Arlington
Arlington, USA
ylei@cse.uta.edu

Raghu Kacker
Information Technology Lab
National Institute of Standards and Technology
Gaithersburg, USA
raghu.kacker@nist.gov

D. Richard Kuhn
Information Technology Lab
National Institute of Standards and Technology
Gaithersburg, USA
d.khun@nist.gov

***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 preserves certain code coverage, e.g., edge coverage. The inputs of these method-level tests are 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, which could be time-consuming. We implemented the framework and applied the framework to seven algorithms in the WEKA tool. The initial results show that 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;

I. INTRODUCTION

Big data applications are software programs that process massive amounts of data. Debugging big data applications can be more complicated and time-consuming. For programs with much smaller inputs, they often take a lot less time to execute comparing to big data applications. When a failure occurs while executing a program with their original dataset, programs with smaller dataset are often easier to debug by using the system-level inputs as tests; developers can debug the suspicious methods one execution at a time. However, it can be challenging for big data applications, because of the long execution time, a large number of method executions, and/or a large number of different variables to be inspected in the method. For example, a classification algorithm *DecisionTable* that is implemented in the WEKA tool [20], takes over two hours to execute the *Heterogeneity Activity Recognition Dataset* from the UC Irvine (UCI) Machine Learning Repository[21]. One of the *DecisionTable's* methods, *updateStatsForClassifier* with 66 lines of code (not including comments and spaces) were executed over half a billion times. Furthermore, each execution was passed with a unique combination of values of its input variables. If a failure occurred during real-world usage of such systems and developers need to investigate such methods, debugging the system with the failing system-level input can be very expensive to accomplish.

Some approaches have been proposed to reduce the effort required for testing and debugging big data applications at the system level [5, 8, 9, 10, 11]. For example, data mining and machine learning techniques are used to reduce the size of the real dataset or generate synthetic datasets [9, 10] for the testing purpose. However, when debugging on the system level with the real dataset, such methods are less likely to provide assistance for reducing the dataset. Debugging approaches such as delta debugging [14] can identify the minimum failure-inducing inputs at the system level, which can reduce the size of the inputs while preserving the failure that the real dataset triggered. However, delta debugging can be very expensive in practice. This is because it requires the input data be recursively split into smaller chunks, and individual and combinations of chunks need to be repeatedly executed at the system level until a minimal failure-inducing input is identified. For big data applications, due to their input size and execution time, delta debugging may have to execute almost exhaustively with a very large number of different combinations of inputs on the system level, which can be impractical to use. In such situations, developers would likely

have to randomly debug some method executions or just by guessing to investigate the suspicious methods. Providing developers a small set of simpler method executions that would likely be sufficient to trigger the fault they are looking for would give developers a big head start on the debugging, and could potentially save developers a lot of time and efforts.

Our approach consists of mainly two steps, generating method-level tests from a failed system-level execution while preserving a given code coverage, e.g., edge coverage [19], and reducing the size of the method-level tests using our binary reduction technique. The main idea is to evaluate each method execution based on specific coverage criteria, such as statement coverage, edge, node and different types of path coverage based on Control Flow Graph (CFG) [19], note that other coverage criteria, e.g., prime-path coverage[19], could also be used in our approach. We record the method executions as method-level tests when they cover any new coverage elements with respect to the chosen coverage criterion. So only the necessary method executions concerning the selected criterion will be executed later for debugging purposes instead of the entire system. Furthermore, we reduce the inputs of method-level tests with large collection typed variables using a binary reduction technique that is inspired by binary search, to select the minimal inputs that still preserve the coverage criterion covered by the originally recorded method-level tests. Based on the coverage criterion we choose, a much smaller set of method-level tests can effectively detect the fault that may have caused the failure observed at the system level.

Consider the *updateStatsForClassifier* method in the *DecisionTable* implementation. Recall that it consists of 66 lines of code. When the *DecisionTable* implementation was executed with the *Heterogeneity Activity Recognition Dataset* from the UC Irvine (UCI) Machine Learning Repository[21], the method was executed for 557,305,280 times. However, recording only three, five and nine method-level tests was sufficient to achieve the same edge, edge pair and edge set coverage as the original 557,305,280 method executions, with a mutant killing rate ranging from 86.07% to 86.89%. The edge, edge-pair, and edge-set coverage are defined in Section II-A. Mutation testing is defined in Section III-D. 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 will significantly reduce the number of method executions that developers have to manually inspect while maintaining a high probability that the recorded method-level tests can detect potential fault(s) in the suspicious methods.

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 the edge coverage, edge-pair coverage, and edge set coverage. 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 inputs, such as the previously mentioned *updateStatsForClassifier* method from the *DecisionTable* implementation. We further reduce the size of these large inputs using a binary reduction technique to select their minimal subset that preserves the same coverage elements the original recorded method-level test covers. The average input size for *updateStatsForClassifier* is reduce to 12.53 MB from 1269.76 GB. The efforts and time required from developers are further reduced using our binary reduction technique.

Moreover, the test effectiveness was evaluated using PITest (PIT) [24], a commonly used mutation testing tool. All 25 available mutant generators were enabled for mutant generation. When combining tests generated for the edge, edge-pair, and edge-set coverage, the mutant killing rate is ranging from 57.58% to 91.43%. The two methods *select_working_set* and *selectModel* with lowest mutant killing rate of 78.91% and 62.88% for their edge-set coverage tests are further investigated by comparing the mutation testing result differences between these two methods, and their original system-level execution.

We summarize the contribution of the paper as follows:

• We present a new framework to debug big data applications using method-level tests efficiently. When a failure occurs on the system level for big data applications, by utilizing different test effectiveness measurements, our framework can generate a small number of tests that are effective in detecting faults for debugging.

• We present a binary reduction technique that can effectively reduce the size of method-level tests and preserve the selected coverage elements, which further reduces the efforts and time required from developers for debugging.

• We implemented our framework and conducted experiments on seven methods from four different machine learning algorithms' implementations. The implementation currently requires manual code analysis and instrumentation of a few lines of code to record method-level tests. The code analysis and instrumentation can be later fully automated. Executing the recorded method-level tests have been fully automated.

The rest of the paper is organized as follows. Section II presents the details of our approach and implementation. The experimentation design and results are summarized in Section III. Section IV reviews the related works, and finally, Section V concludes this paper and our future work.

II. Approach

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

A. Record Test

Once a failure occurs, a developer typically identifies several suspicious locations based on his or her understanding about 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.

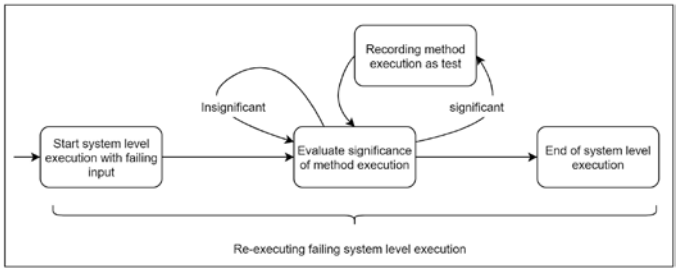
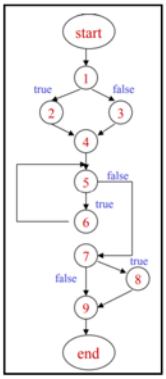


Fig. 1. Recording Process at Runtime

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 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 [19], edge-pair coverage [19], 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 representation, using graph notation, of all paths that might be traversed through a program during its execution, captures information about how the control is transferred in a program. 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[19]. Such as shown in Figure 2. A directed edge[19] represents the control flow from one node to another. And a path[19] is a sequence of nodes, where each pair of adjacent nodes is an edge. 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 [19], could also be used in our approach.

```
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

Fig. 2. Example of Control Flow Graph

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

1) Instrumentation

We first analyze the source code based on the selected coverage criteria using Control Flow Graph (CFG). Instead of manually drawing CFG for each suspicious method, we used a tool called Atlas [23], which is an Eclipse plugin tool developed by EnSoft Corp. Atlas can automatically generate CFGs based on the source code of a selected method. The CFGs generated by Atlas uses each line of code as a basic block. This is different from the classical definition [19] that a basic block consists of a sequence of consecutive statements with a single entry and a single exit point. As an example, a simple method and its CFG generated using Atlas is shown in Figure 3. 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 3 marks the lines of code combined to be a basic block as we previously defined.

```
public float power(int x, int y){
    int power;
    float z;
    if (y < 0){
        power = -y;
    }else{
        power = y;
    }
    z = 1;
    while (power != 0){
        z = z * x;
        power = power - 1;
    }
    if(y < 0){
        z = 1 / z;
    }
    return z;
}
```

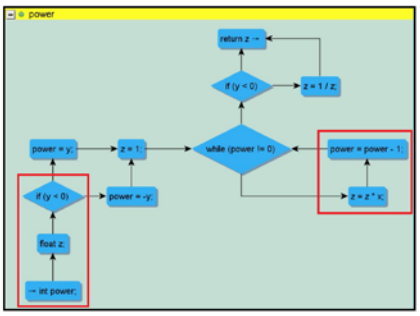


Fig. 3. Example of Modifying Generated Control Flow Graph

Once we have the CFGs that represents the suspicious methods, we will instrument these methods by adding a few lines of code that invokes our recording program. Figure 4 shows an example of how our implementation instruments 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 the suspicious methods. The *ParaArray* contains the list of input parameters used for a method execution. The *ParaTypeArray* 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 object where the suspicious method was invoked from, to store the instance variables accessed during the execution as well. 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. 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.

```
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 }
```

Fig. 4. Example of Instrumentation

Recording basic block indexes with multiple entrances at runtime requires different instrumentation as we previously described. As the example shown in Figure 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 exercise of such blocks.

2) Method Execution Evaluation

In our program, 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 edges, edge-pairs or edge-set that have 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 edges that have 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.

3) Serialization

Once a method execution is determined to be significant, we will record the inputs of the method execution using serialization. Serialization is generally an expensive process, especially for the built-in serialization method that Java provides. There are other tools [27] developed by third-parties for serializing Java objects at a much faster speed such as serializing Java objects into JSON, XML files. However, many of these alternative serialization methods have some limitations. For example, some serialization methods [27] do not work when objects contain circular references or there is no constructor. Java default serialization does guarantee that if a class implements Serializable, its objects can be correctly serialized and deserialized. We used a tool called FST [25] that can be ten times faster [25]. In our experiments, FST performed significantly faster than Java built-in serialization. Furthermore, FST is the only alternative tool that can correctly serialize and deserialize all the objects in the subject programs used in our experiments.

After using FST to improve the performance of our test recording, there is still a situation where we experience significant overhead. To ensure an exact copy of the inputs is created, we perform deep copy on objects by serializing and deserializing the object. Most of the stored inputs will not end up become recorded after the evaluation of method executions. Much of the time spent to store deep copied objects are unnecessary. And these unnecessary time can be huge when a method takes large inputs, and/or been executed for a large number of times. We store and perform deep copy on inputs is because the value of an input object could potentially change during a method execution, especially for void methods that operate on instance variables, we need to store the inputs of the method execution before the execution starts. For *buildClusterer* and *EM_Init*, this recording approach took less than 18% overhead to the original system-level execution, and the recording can be completed by executing the entire program once. However, the overhead can be as high as 7 to 30 times the original system-level execution time for recording tests of the other five selected methods. In such cases, our solution is recording the method-level tests by executing the entire system twice. In the first execution, we will not store any inputs. Instead, we only record the ID of a significant method

execution. 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. In our experiments, the overhead was reduced to near in average 2.5 times the original system-level execution from 7 to 30 times for five of our selected methods.

B. 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, for three (*selectModel*, *updateStatsForClassifier*, *cutPointsForSubset*) of the selected methods during our experiments, we recorded a total of 35 method-level tests. Their inputs have the average size of 1.45GB, and total over 50GB. 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 collection typed variables. Such as for the above mentioned three methods, 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 preserve the coverage elements. The reduction can further reduce the waiting time for loading 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.

We implemented the initial working prototype of our framework in Java with around 800 lines of code to work with applications implemented in Java. Minor efforts are required from developers to instrument the suspicious method's source code. After instrumentation, the recording process has been automated. The reduction approach requires developers to identify the large collection typed input variables. In the end, the reloading of the recorded and reduced method-level tests has been automated for debugging. The currently implemented coverage criteria are the edge, edge-pair, and edge-set coverage. The experiments we conducted to evaluate our framework will be shown in the next section.

III. Experiments

In this section, we discuss how we conducted our experiments and present the experiment results. In Section III-A, we discuss how we selected datasets, applications, and methods to be used for our experiments. Section III-B presents the statistics of the recorded method-level tests. Section III-C presents the statistics of the reduced method-level tests. Section III-D presents how we used a mutation testing tool and the results of our mutation testing for both the recorded tests, and the reduced tests. And finally, Section III-E presents the performance analysis of our framework. All the source code, generated graphs, recorded method-level tests, reduced method-level tests and mutation reports are openly 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.

A. Subjects

To evaluate our framework in everyday real-world challenging situations for debugging big data applications, we have the following criteria to follow when selecting our subject datasets, applications, and methods.

- To produce experiment results that reflect real-world situations, our evaluation should be done with real-world datasets and applications.
- To reflect situations where big data applications take a long time to execute, the execution time of the candidate datasets with corresponding candidate applications should be more than one hour.
- To reflect situations where the significant methods are extensively executed, candidate methods should be executed for at least 100 times.
- To reflect the higher complexity of big data applications' and have a larger code sample to conduct coverage experiments on, candidate methods should have at least 30 lines of code covered by the original system-level execution.
- Total of at least five methods should be used for the experiments.

TABLE I. Algorithm Execution Time

Algorithm	Execution time for Phones_gyroscope Dataset
Apriori	N/A (Cannot handle Numeric Attributes)
DecisionTable	9,559 seconds (2.66 Hours)
EM	Unable to finish within 24 hours
HierarchicalClusterer	OutOfMemoryError
J48	6,357 seconds (1.77 Hours)
LibSVM	Unable to finish within 24 hours
LinearRegression	N/A (Cannot handle multi-valued nominal class)
MakeDensityBasedClusterer	191 seconds
RandomTree	577 seconds
SimpleKMeans	301 seconds

Based on the above criteria, we first randomly selected ten algorithms' implementations from the WEKA tool, a collection of machine learning algorithms for data mining tasks, which is used in real-world by many data analysts. Next, we selected one collection of dataset with the largest number of instances(Collected 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 dataset *Heterogeneity Activity Recognition* contains four CSV files for four different types of devices with a total of 43,930,257 instances and 16 attributes. The Heterogeneity Activity *Recognition* collection has the data type of multivariate and time-series and has the attribute type of real numbers. The dataset is applicable for both classification and clustering in default. Among the four CSV files, the largest CSV file *Phones_gyroscope* dataset of 1.38GB is used to be executed against the ten randomly selected algorithms to see how long the execution time is. The randomly selected ten algorithms and their execution time are shown in Table I. Note for *HierarchicalClusterer* implementation, we have tried to increase the Java heap space to 60GB using JVM argument "*-Xmx60g.*" However, it still fails with "*OutOfMemoryError*" message.

TABLE II. 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

As shown in Table I, both the *DecisionTable* and *J48* implementation executing the selected dataset meets the execution time requirement, and we were able to select a total of five methods within the implementation of these two algorithms using the selection criteria we previously mentioned. However, the input parameter type for one of the selected method *getInstance* from *DecisionTable* is of type "*java.io.streamtokenizer*," which does not implement Java Serializable. Such types of variables will prevent us from recording its input without making further instrumentation that could potentially change the functionality of the method, which makes this method inapplicable to our experiments. For *EM* and *LibSVM* implementations, they do in fact meet the execution time requirement, but they are taking too long to execute the *Phones_gyroscope* dataset for experimentation purpose due to our limited resources and time. 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*.

After two datasets (original *Phone_gyroscope* dataset and the reduced dataset) and four algorithms' implementations (*DecisionTable*, *EM*, *J48*, *LibSVM*) have been selected, we used EclEmma [26], a Java code coverage tool for Eclipse, to identify the methods with more than 30 lines of code(without comments and spaces) covered by the original system-level execution. A total of seven methods are selected based on the selection criteria we discussed previously. The selected methods and their information are shown in Table II. These methods are then instrumented as previously described in Section II.

B. 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 Table III. Note that the code coverage column in Table III is for all three types of recorded tests, as well as the original failing system-level execution. This is because once all edges are preserved, all the code 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 III, we can see that only a small number of method-level tests is sufficient for preserving coverage for a suspicious method. When failures occur on a system level, recording method-level tests for the suspicious methods using our framework could potentially save developers a lot of time and efforts. The actual fault detection ability of our recorded method-level tests will be further evaluated using mutation testing in Section III-D.

C. Reduced Method-Level Tests

As shown in Table IV, while some of the tests have a reasonable size, three of the 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 dataset on a system level, loading and debugging these tests could still take some time. We will further reduce the size of these tests using our binary reduction approach that was discussed in Section II. In Table V, 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. While one of the tests for *selectModel* can be reduced to 1.7 KB from1.63 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 wereunable to reduce any test inputs for two methods,

TABLE III. RECORDED METHOD EXECUTION INFORMATION

Method	# of Tests for Edge Coverage	# of Covered Edges	# of Tests for Edge-Pair Coverage	# of Covered Edge-Pairs	# of Tests for Edge-Set Coverage	# of Combined Recorded Tests	# of Original Execution Count	Code Coverage	Total # of Lines of Code
buildClusterer	3	83	4	181	3	4	1,910	69.70%	165
cutPointsForSubset	8	30	9	64	17	18	29,564	96.88%	64
EM_Init	1	24	3	55	1	3	191	88.68%	53
handleNumericAttribute	4	32	5	70	33	33	28,314	96.23%	53
select_working_set	7	41	11	111	61	63	417,989	96.15%	52
selectModel	5	35	5	74	6	6	12,391	86.21%	58
updateStatsForClassifier	3	26	5	59	9	11	557,305,280	69.70%	66

TABLE IV. TEST REDUCTION RESULT

Method	Combined # of Tests	# of Reducible 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

buildClusterer and *EM_Init*. We further investigated this by looking into how the collection typed input variables 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.

The third scenario is when the collection variable is not accessed at all. In our implementation, to reproduce method executions precisely, we 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 un-accessed collection variables to empty, while still preserving the coverage elements.

The first and second scenario can potentially use delta debugging [14] 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 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 time reduction, many of the recorded set of method-level tests are now taking seconds instead of minutes after the binary reduction. The loading time of these tests is significantly reduced. When debugging with these reduced tests, not only the tests will be short and easier to debug, the waiting time is also easy to manage.

TABLE V. MUTATION TESTING RESULT

Method	Edge Coverage Mutant Killing Rate		Edge-pair Coverage Mutant Killing Rate		Edge-set Coverage Mutant Killing Rate		Combined Recorded Tests Mutant Killing Rate		# of Mutants Generated for Covered Code	Code Coverage	Total # of Lines of Code
	Recorded	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*			
buildClusterer	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%	269	69.70%	165
cutPointsForSubset	81.71%	81.1%	81.71%	82.32%	85.98%	85.98%	85.98%	85.98%	164	96.88%	64
EM_Init	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%	102	88.68%	53
handleNumericAttribute	89.29%	88.57%	90.71%	90.71%	91.43%	91.43%	91.43%	91.43%	140	96.23%	53
select_working_set	71.88%	75%	73.44%	75%	75%	78.91%	75%	78.91%	128	96.15%	52
selectModel	57.58%	57.58%	57.58%	57.58%	62.88%	62.88%	62.88%	62.88%	132	86.21%	58
updateStatsForClassifier	86.07%	81.15%	86.07%	84.43%	86.89%	86.07%	86.89%	86.89%	122	69.70%	66
Average	79.00%	78.55%	79.42%	79.49%	81.23%	81.67%	81.23%	81.79%			

D. Mutation Testing

For mutation testing, we used PITest (PIT) [24], a mutation testing system 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) contains only one fault and is executed against the unit tests that developers provide.

Mutation testing requires the provided unit tests to be passing tests. This 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 as well and should be considered as part of the output. 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 provide 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, 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.

Table V shows the mutation testing result of our recorded and reduced method-level tests. Please note that PIT currently does not support the mutant generation of only covered statements. If a mutant is located at a statement that was not covered by the test, the mutant will not be exercised, hence will not 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 also not exercised by the original system-level execution. The total number of mutants generated for each selected method in Table V are calculated manually which consist of only exercised mutants by our tests.

For recorded method-level tests without reduction shown in Table V, 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, with a small number of tests, the tests show high effectiveness in detecting potential faults that could occur in the selected methods. Five out of seven selected methods have tests with over 80% of mutant killing rate. The average mutant killing rate across seven methods are around 80% for all four different sets of tests that achieves edge coverage, edge-pair coverage, edge-set coverage, and these three combined. 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*". This is likely due to some coverage elements other than our selected coverage criteria were introduced to the reduction of the method-level tests. The mutation testing results of the reduce tests show that even after a significant amount of inputs are reduced, if the coverage elements are preserved, the fault detecting effectiveness of the original method-level tests can still be 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 VI. 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. 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 VI. 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 of their original system-level execution has a lower mutant killing rate.

Furthermore, after investigating the un-killed propagatable mutants in the recorded method-level tests, three un-killed propagatable mutants from *select_working_set* and one from *selectModel* were mutations related to modifying boundary conditions. Which means by adding more method-level tests that cover boundary conditions, the 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.

E. 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 VII. 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 VII 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.

As previously mentioned in Section II, we have two solutions for recording selected method executions. One method 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 will only require executing the entire system once. However, in cases where a method has large inputs or is executed for a large number of times, this method may have a significant performance issue due to all the unnecessary serialization. The second solution was to execute the entire system twice. In the first execution, we evaluate each method execution and store the execution IDs of the to be recorded method executions. In the second 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 method. In Table VII.

TABLE VII. PERFORMANCE EVALUATION

Method	Original Execution Time	Combined Test Recording Time	Combined Test Execution Time		Combined 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

In Table VII, 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 sterilization for deep copy as well). We believe the time is manageable for developers. This is because without our framework, developers could spend more time to debug their big data application. 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.

IV. 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 to test general *MapReduce* programs [5]. Morán et al. proposed *MRFlow*, a testing technique tailored to test *MapReduce* programs [11]. Morán et al. also proposed a technique to generate different infrastructure configurations for a given *MapReduce* application that can be used to reveal functional faults [10]. They also proposed an automatic test framework that can detect functional faults automatically [9]. Li et al. proposed a tool to generate test input data using input doing model information for testing *BigData* applications [8]. Previous work reported in [5, 9, 10, 11] focuses on generating tests that help to identify functional faults, i.e., faults that will cause the program to generate expected output for some configurations and invalid output for other configurations. In contrast, when a failure occurs for big data application, our work focuses recording existing method executions that preserves coverage as tests to facilitate the debugging of certain suspicious methods searching for the fault(s) that may have caused the failure observed at the system level.

Next, we review work related to debugging big data applications. Gulzar et al. proposed a tool BigDebug that simulates breakpoints to enable a developer to inspect a program without actually pausing the entire computation [13]. 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. Li 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 [15]. Our work is similar to Gulzar [13] and Li [15] regarding the debugging of big data applications without re-executing the potentially time-consuming original system-level execution. In contrast, our work focuses on recording significant method-level executions to be replayed for debugging instead of recording part of the system-level inputs [13] and log information [15].

Next, we review work related to recording program information and using them in unit test generation. Pasternak et al. proposed a technique that records interactions occurring during the execution of Java programs and used that information to construct unit tests automatically using GenUTest [16]. Orso et al. proposed a technique and conducted a feasibility study using SCARPE, a prototype tool, for selective capture and replay of program executions [12]. Similar to work presented in this paper, Orso's technique [12] can be used to automatically generate unit tests based on the recorded information. Our work is similar to Pasternak [16] and Orso [12] regarding recording the method-level tests based on the system-level execution. In contrast, our work does not require complex instrumentation techniques on the target's bytecode [12], the instrumentation used in our implementation is simple and relies on certain code coverage criterion. And our implementation will work on any method, as long as their input variables implement Java Serializable, where GenUTest partially works on some inner classes and anonymous classes.

Next, we review work related to reducing input size for debugging purpose. Zeller et al. proposed a technique to isolate failure-inducing inputs on the system level to reduce work required for debugging using Delta Debugging [14]. Clause [22] et al. presented a technique based on dynamic tainting for automatically identifying subsets of a program's inputs that are relevant to a failure. Our work is similar to Zeller [14] and Clause [22] regarding reducing inputs based on certain aspects of the execution for debugging purpose. In contrast, we preserve coverage elements instead of failure, and using binary reduction for the dividing, which can have better performance than Delta Debugging and dynamic tainting.

V. Conclusion & Future Work

In this paper, we presented an approach to provide developers with concrete method-level tests that were recorded from the system-level input dataset and selected using edge, edge-pair, and edge-set coverage. Binary reduction is available for reducing method-level tests with large input. The set of method-level tests that are provided by our approach will help developers to effectively debug suspicious methods against properties of the original input dataset, and significantly reduce time required for debugging big data applications by avoiding the executions of other non-important methods while maintaining a high probability that the recorded tests will trigger failure(s) caused by the fault(s).

For future work, our approach will involve developing techniques for automatic code static analysis that provides information for instrumenting the source code. Moreover, automatic code instrumentation for inserting code to record related coverage elements. We will also focus on designing more detailed experiments with more real-world big data applications, datasets, and coverage criteria to more precisely analyze the effectiveness of our approach.

VI. 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.

REFERENCES

[1] Blue, D., Segall, I., Tzoref-Brill, R., & Zlotnick, A. (2013, May). Interaction-based test-suite minimization. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 182-191). IEEE Press.

[2] Bryce, R. C., Sampath, S., Pedersen, J. B., & Manchester, S. (2011). Test suite prioritization by cost-based combinatorial interaction coverage. *International Journal of System Assurance Engineering and Management*, *2*(2), 126-134.

[3] Chandrasekaran, J., Feng, H., Lei, Y., Kuhn, D. R., & Kacker, R. (2017, March). Applying Combinatorial Testing to Data Mining Algorithms. In *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on* (pp. 253-261). IEEE.

[4] Chen, T. Y., & Lau, M. F. (1996). Dividing strategies for the optimization of a test suite. *Information Processing Letters*, *60*(3), 135-141.

[5] 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.

[6] Harrold, M. J., Gupta, R., & Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *2*(3), 270-285.

[7] Jones, J. A., & Harrold, M. J. (2003). Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering*, *29*(3), 195-209.

[8] Li, N., Lei, Y., Khan, H. R., Liu, J., & Guo, Y. (2016, August). Applying combinatorial test data generation to big data applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (pp. 637-647). ACM.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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

[14] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input", IEEE Transactions on Software Engineering28(2), February 2002, pp. 183-200.

[15] 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)

[16] 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.

[17] Pan, J., & Center, L. T. (1995). Procedures for reducing the size of coverage-based test sets. In *Proceedings of International Conference on Testing Computer Software*.

[18] Sampath, S., Bryce, R. C., Jain, S., & Manchester, S. (2011, September). A tool for combination-based prioritization and reduction of developer-session-based test suites. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on* (pp. 574-577). IEEE.

[19] 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.

[20] 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.

[21] 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.

[22] J. Clause and A. Orso. Penumbra: Automatically identifying failure relevant inputs using dynamic tainting. In ISSTA, pages 249–260, 2009.

[23] "Atlas Platform, EnSoft Corp." http://www.ensoftcorp.com.

[24] "PITest." http://pitest.org/.

[25] "FST, fast-serialization." https://github.com/RuedigerMoeller/fast-serialization.

[26] "EclEmma." https://www.eclemma.org/.

[27] "Gson." https://futurestud.io/tutorials/gson-getting-started-with-java-json-serialization-deserialization.