Open Source Mutation Testing Additions and Research of Possible Future Additions*

Arjun Sethi
The University of Texas at Dallas
800 West Campbell Road
Richardson, Texas 75080
arjun.sethi@utdallas.edu

Github Link: https://github.com/arjun-sethi/extendedPIT

ABSTRACT

Mutation testing is a widely known concept in the computer science and software engineering industry to test how well test suites for newly developed programs are able to catch bugs and faults. Currently, many tools exist, especially for Java, to aid programmers create mutated versions of their code, run tests on them, and output a mutation score that signifies how well the test suite is. Utilizing the PIT testing tool for Java and JVM programs, additions to the mutation options available are added to the project, and future functions and options for programmers to use are researched. Testing the reliability of the added functionalities is tested with real-world projects located on GitHub with JUNIT test suites available in the open source packages.

CCS CONCEPTS

• **Software Testing** → Mutation testing, reachability, necessity, sufficiency, **Real World Projects** → JAVA, JVM, JUNIT, PIT, JavaAgent on-the-fly, ASM bytecode framework, Maven build system

KEYWORDS

ASM byte code manipulation, Java mutation testing

ACM Reference format:

A. Sethi. 2018. PIT Additions Paper in word Format. In *University of Texas at Dallas, Richardson, Texas USA, February 2018 (Sethi'18),*

1 INTRODUCTION

Software testing and verification is always a crucial part of designing software. Over the past few decades, many researchers and companies have created tools to aid in software testing, and one of the most effective method proven to date is mutation testing. Mutation testing is used to test the test suites that developers create to check the validity of their program codes. The principle behind mutation testing is to invoke mutations, or changes, in certain statements of the program code, running the test suites on the mutated code, checking to see if the output of the mutated code matches the output of the original code. The more tests in the test suite that fail matching the output of the original code, the more mutants or bugs are caught, and therefore the validity of the test suite increases. If a test in the test suite does have the same output on both codes, the test is considered a weak test because it was not able to cover bugs in the code. The effectiveness of test suites can be measured using an equation:

$$MS(T) = \frac{\#KilledMutants}{\#AllMutants\text{-}\#EquivalentMutants}$$

Where an equivalent mutant is one who's output of program P is equivalent to the output of the mutated program M when both are run in test t, P(t)=M(t) [1].

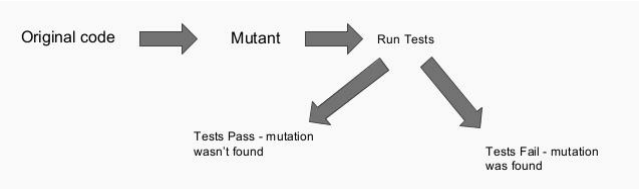

Figure 1: How Mutation Testing works.

2 PROBLEM

Traditional test coverage such as line coverage or branch coverage provide users with measures of how well tests will cover code, but they do not detect whether or not the tests include faults or bugs. This is why mutation testing tools aid programmers to automatically test their program codes and how

well the tests they create for their program code will be reliable or not. However, one problem computer scientists constantly face is tools that are both fast and offer a wide variety of mutation operators. Most tools offer speed over reliability, reliability over speed, or a low to medium mix of both. This projects goal is to investigate this issue by utilizing current open source mutation testing tools, and making changes and additions to the software code to create a better user experience, and faster and more reliable mutation testing. One of the most widely utilized systems in the Java world for mutation testing that is fast and offers a high mutation testing score is PIT.

PIT offers range of mutations, but does not currently include traditional mutations and functions that would make testing test suites even more simple. These include, but are not limited to: arithmetic operation deletion, relational operator replacement, and arithmetic operator replacement, unary operator replacement, scalar variable replacement, and absolute value insertion.

3 TECHNIQUES

Existing technique of PIT allows it to be very efficient by offering a high mutation score while running over program code in seconds as opposed to lower mutation scores and minutes with other programs. PIT chooses which test to run in test suites by priorities three factors: line coverage, test execution speed, and test naming convention [2]. PIT also has default mutations set up for testing, additional optional mutations that users must set, and options for on and off of both the default and optional features. It has been proved that as long as the same number of mutations are tested in pre-selection and random selection of mutations, random selection will result in more efficient bug finding [1]. One of the techniques this project will attempt to include is an option for random selection of the mutations in PIT, allowing PIT to find bugs faster.

4 IMPLEMENTATION

4.1 Planning

The first step of implementation will be to learn ASM bytecode engineering framework, JavaAgent on-the-fly code instrumentation, and Maven build system. ASM bytecode engineering framework will be learned because PIT operates mutations on byte code, giving it the extra edge on speed over other programs. Introducing new mutation operations that are not currently supported by PIT will be added at bytecode level. The JavaAgent on-the-fly code instrumentation will be used to edit byte code as it is running. Finally, the Maven build system will be learned because this is what PIT runs on.

The next step will be to include some of the missing mutations in PIT that are useful in mutation testing: arithmetic operation deletion, relational operator replacement, and arithmetic operator replacement, unary operator replacement, and absolute value insertion. The process to complete these should be clearer after understanding the first three parts of the system explained above. Relational operator replacement exists in PIT; however, PIT can be taken a step forward and replace a relational operator with each of the other ones, not just one specific one. For example, < should be replaced by !=, ≤, ≥, etc. not just ≥.

The third step will be to implement additional features to provide a higher mutation score. The first mutation will be to add a conditional checker for each field dereference only when o!=null. The second mutation will be method invocation replacement, which replaces an invoked method with another overloading method of the same name but different arguments, followed by replacing a method invocation with a call to another method with same return type and argument types but different named method. Lastly, in each expression a variable replacement with another variable of the same type, also known as scalar variable replacement.

The last part will be to implement ideas that are above and beyond what has been researched, such as editing byte code level to include meta mutants. Currently meta mutants can only be included via source code, and doing it via byte code has been proven to be a difficult process. Other options worth at least researching into and giving commentary as to why or why not they can be included into PIT is PIT supporting object-oriented mutation operators along with the currently supported traditional operators, mutation operators for Java generics, mutations for linked list or array list operations (perhaps instead of inserting at end change to insert at beginning), floor/ceiling function mutations, and variable to value replacement. Variable to value replacement maybe quite simple as it is similar to scalar variable replacement. These are only a few different types of mutations that would be extremely helpful in PIT but are not yet supported. If time persists, research should be done in some or all of these areas

4.2 Creating New Mutators

In order to extend Pitest and introduce custom mutation operators, the same steps need to be followed regardless of the mutator being introduced. Step one consists of creating a new instance of MethodMutatorFactory, creating a new instance of MethodVisitor to perform operations of creating the mutant, and finally update the Mutator.java file so that pitest knows about the new operator.

A MethodMutatorFactory is a factory creating method mutating method visitors. Those method visitors will serve two purposes: finding new mutation points and applying those mutations to the byte code. A MethodMutatorFactory instance will have a globally unique id and must provide a human readable name via the getName() method. This name will be used in the reports created to document and describe the mutation(s) applied, and inside the mutator.java file to notify pitest when this name is referenced in pom.xml files then refer back to this specific factory instance.

When creating the instances of MethodMutatorFactory and MethodVisitior, these should be saved in a java file under “pitest/src/main/java/org/pitest/mutationtest/engine/gregor/mutators”. This is where pitest looks for the instructions of what to

perform when performing a mutation operation. The new java file should be a part of the package "org.pitest.mutationtest.engine.gregor.mutators".

The mutator.java file that needs to be edited to notify pitest of the new mutation is located at "pitest/src/main/java/org/pitest/mutationtest/engine/gregor/config/Mutators.java. This file should have an import of the java file that includes the instances of MethodMutatorFactory and MethodVisitor. Pitest will only know of the mutation names that are listed in mutator.java, and use the imported java files that correlate to the mutation operator names to find out what instructions to perform to implement the mutation.

4.3 Arithmetic Operation Deletion

A key to understanding the arithmetic operation deletion (AOD) is that this mutation must be performed at the byte code level. There are only two types of mutators in this mutation, namely replaced a [operator] b with a or with b.

Original	AOD1	AOD2
A+B	A	B
A-B	A	B
A*B	A	B
A/B	A	B
A%B	A	B

Figure 2: Arithmetic Operator Deletion of all arithmetic operators. AOD1 is the first mutator that replaced the original statement with A, and AOD2 replaces the statement with B.

When a format of A [operator] B is converted into bytecode, A is placed on the stack, and B is placed on top of the A. A pop method will discard the top word on the stack. Thus, it is quite simple to replace the A [operator] B with A. Simply inside of the MethodVisitor instance created for the mutation, utilize the visitInsn(int opcode) method, which is a method that performs an operation when a zero operand instruction is visited, and for the "opcode" parameter pass a pop stack instruction. This will pop the B off the top of the stack and discard it, leaving A as the top of the stack. Thus, since A is at the top of the stack, A [operator] B will now be replaced by A since we have discarded the B.

Stack	
Before	After
B	A
A	…
….	…

Figure 3: The stack in memory before and after Opcodes.pop of the top of the stack.

This method will only work for integers and floats because the variable types consist of a single word item on stack, and doing a single pop instruction will pop the entirety of the variable off of the stack. For longs and doubles, the bit length is double that of integer and float respectively. For this reason, if the opcode detected is of type long or double, the vistInsn method should be passed a Pop2 instruction, which removes two single word items from the stack instead of just one. Doing a pop would only pop off half of the long or double. Thus, after pop2 the double or long B will be popped off entirely from the stack, discarded, and A will be left at the top.

Stack	
Before	After
Item1 B	Item1 A
Item2 B	Item2 A
Item1 A	…
Item2 A	…

Figure 4: The stack in memory before and after Opcodes.pop2 of the top of the stack.

A similar method should be used when performing an AOD on A [operator] B to be mutated to B. There exists a method known as dup_x1 that will duplicate the word at the top of the stack and insert the duplication into the stack beneath the second word on the stack (i.e. set it as the second from the top of the stack). Now, popping the top of the stack until the duplication value is reached allows us to have a method similar to the previous method when mutating to A. In other words, after a dup_x1 of B, perform a pop2 to pop the top two words, B and A, from the stack leaving the top of the stack as the duplicated B.

Stack		
Before	After dup_x1	After pop2
B	B	B
A	B	…
…		…
…	…	…

Figure 5: The stack in memory before and after dup_x1 and pop2, leaving a mutation of A [operator] B to only B.

Again, integer and float versus long and double word lengths in the stack have to be taken into account. In order to duplicate the top of the stack when a long or double is involved, a dup2_x2 instead of dup2_x1 will duplicate the top two words of the stack and insert it into the stack after the first four words on the top of the stack. After this is complete, performing a pop2 twice will leave only the double or long version of B on the stack.

Stack			
Before	After dup_x1	After pop2	After pop2
Item1 B	Item1 B	Item1 A	Item1 B
Item2 B	Item2 B	Item2 A	Item2 B
Item1 A	Item1 A	Item1 B	…
Item2 A	Item2 A	Item2 B	…
…	Item1 B	…	…
…	Item2 B	…	…

Figure 6: The stack in memory before and after dup2_x2 and pop2, and a second pop2.

4.4 Relational Operation Replacement & Arithmetic Operation Replacement

Relational operation replacement (ROR) is simpler than arithmetic operational deletion. Currently, Pitest has a negate conditionals mutator as one of the default mutations to invoke when testing test suites, however, it only does one replacement per conditional operator. In ROR, every conditional needs to be replaced by every other conditional. For a total of six different types of conditionals and one set of mutations built into Pitest, there needs to be five more cases of mutations to achieve this goal.

Original	NEGATE_COND	ROR 1	ROR 2	ROR 3	ROR 4
==	!=	>	<	>=	<=
!=	==	>	<	>=	<=
<	>=	>	!=	==	<=
<=	>	!=	<	>=	==
>=	<	>	!=	==	<=
>	<=	!=	<	>=	==

Figure 7: The set of mutations implemented into Pitest to achieve relational operation replacement.

Simply, utilize the "MUTATIONS.put" method to replace a detected conditional statement with a new substitution. This needs to be done after the section 4.2.1 has been written, and inside the methodVisitor.

Arithmetic operation replacement (AOR) and relational operator replacement were implemented in similar ways. Pitest offers a Math mutator, which includes mutating + to - , - to + , * to / , / to *, and % to *. This works well, however, an even better method would be to replace each operator with every other arithmetic operator, hence creating the AOR mutational method.

Original	MATH	AOR1	AOR2	AOR3
+	-	*	/	%
-	+	*	/	%
*	/	+	-	%
/	*	+	-	&
%	*	+	-	/

Figure 8: Arithmetic Operator Replacement of all arithmetic operators.

4.5 Inversion

The Inversion mutator will change the sign of a numeric variable in pitest. If the variable is a positive number, the mutator will change it to negative. If the variable is a negative number, the mutator will change it to positive. Currently Pitest only changes positive to negative for floating point numbers and integers. The addition of this mutator will provide a mutation for double, float, long, and int, as well as provide positive to negative and negative to positive for all.

This mutator has a simple implementation. Push a constant value of "-1" onto the stack. Based on the detected instance of an ILOAD, FLOAD, DLOAD, or LLOAD, the type of "-1" to push to the stack will correspond with the type of load that is detected. Then, a multiplication of the top two values on the stack will result in the original variable with its opposite sign to be pushed on the stack. Finally, perform an ISTORE, LSTORE, FSTORE, or DSTORE to the table index of the original variable. This mutator will utilize the visitVarInsn(opcode, index) to detect which opcode and which variable index is being mutated.

Note that if the variable is positive, multiplying by a -1 will result in negative. If the variable is negative, multiplying by a -1 will result in a positive. Figure 8 is a stack implementation of this mutator.

Before	LLoad	new Long("-1")	LMUL	LSTORE
…	Item	-1	-1 * Item	…
…				…
…	…	Item	…	…
…	…		…	…

Figure 8: Stack implementation of inverting a long variable

4.6 Unary Operator Insertion

Unary Operator Insertion (UOI) simulates ++x and --x for variable of type float, double, long, and int. This mutator has two cases, increment and decrement, although they are very similar in implementation. When a load instruction is seen in the bytecode and x is being mutated to ++x, this mutator will place the variable value onto the stack, place a constant value "1" onto the stack, add the top two values of the stack (the variable and constant "1") and push the result of this to the stack, and finally store the result from the top of the stack into the original variable. Similarly when x is being mutated to --x, the only difference in the process will be to do a subtraction instead of an addition on the stack.

For example, this mutator will mutate x % y to (--x) % y or x % (--y). A stack representation is seen in figure 9.

Before	ILOAD	ICONST_1	IADD	ISTORE
…	Item	-1	Item-1	…
…	…	Item	…	…
…	…	…	…	…

Figure 9: Stack example of mutating x to --x

4.7 Scalar Variable Replacement

This mutator will mutate an arithmetic expression to replace the variables in the expression. For example, if an expression e is e= x+y, the mutation will be e=x+x or e=y+y. The mutator required two enums, one for performing x+y to x+x and one for performing x+y to y+y.

To perform x+y to x+x, the stack has to be popped of its top value, and duplicated with the result. This will remove y from the top, and leave two instances of x on the stack. To perform x+y to y+y, a dup_x1 and pop2 (for floats and ints) or a dup2_x2 and pop2 twice (for longs and doubles) is needed before performing a pop and dup as before. The purpose of dup_x1 or dup_x2 is to duplicate the top value of the stack, y, and place it underneath x. The pop2 will then remove the instance of y and x from the top of the stack, leaving only y. Duplicating the stack now will leave two instances of y that can be utilized in the arithmetic expression.

Before	DUP_X1	POP2	DUP
Y	Y	Y	Y
X	X	…	Y
…	Y	…	…

Figure 10: stack implementation of scalar variable replacement mutator for x+y to y+y.

4.8 M1: Field Deference Null Check

The goal of this mutator is for each field dereference, such as o.f. where o is a field, add a conditional checker to perform the field dereference only when o!=null. This mutator was a bit harder than the ones implemented so far. It begins by waiting for a GETFIELD opcode to occur. If a GETFIELD occurs, there will be an object reference on the top of the stack that will be duplicated using dup. Then, an ifnonnull will pop an objectref from the top of the stack and check if it is equal to null. If the object is null, then a pop of the top of the stack occurs to remove the object reference. Then, depending on the original type of the object, a default value of either 0 for int, double, float, and long or a null for anything else (i.e. array) will be pushed on the top of the stack. Then, a visitlabel instruction will take the code to obtain the next instruction. If the object is not null, then a visitlabel to skip the above-mentioned steps occurs and the operation of the code continues as it was supposed to.

4.9 M2: Replace Overloading Method Arguments

The goal of this mutator is to replace a method invocation with a call to another method invocation that has the same name and return type but a different number of arguments. If a new overloading method has more arguments, a inclusion of null or default values for each extra argument from the original should occur.

The key to this mutator is to create a scanner that will initially scan the java class and keep a list of the methods and their descriptions in memory. Then, during execution of the code utilize the list to randomly replace a method invocation with compatible ones (i.e. same return type and name but different number of arguments).

The scanner will have three lists, methodDescriptorList, accessTypeList, and staticTypeList. This will store each method's parameters and return type, access type (public, protected, or private), and if it is a static type or not. The scanner will visit each method in the class, compare its parameters, return type, access type, and static type to the method being replaced. If it is compatible, it will be stored in the three lists. If not, it is ignored. After scanning all the methods and building a list, the mutating class will select one method at random from the list of acceptable methods that can replace the current one. If the arguments of the new method are more in number than the arguments of the old method, the stack will be manipulated. For each argument missing, a push of a default value will be placed onto the stack. When the method is invoked, all parameters necessary will be available on the stack.

All classes are parsed into a byte array. This byte array is what is used in the scanner to shift through the whole code to find compatible methods. This byte array is passed in from the methodmutatingfactory enum instance.

4.10 M3: Replace Overloading Method Names

This mutator is similar to the M2 mutator, except each method invoked that is replaced with another method must have the same arguments and return type. Hence, the method's name is replaced.

A scanner method similar to M2 is required. Anytime a method is invoked in the java class, the class bytecode will be sent to the scanner to utilize to find a suitable method to replace it with. The method's descriptor should be the same, however, the name is the only change that needs to occur. If a method being scanned is compatible, it will be stored in a list. At the end of scanning the entire class, the mutating class will obtain a random method to mutate the current method with. The stack does not need to be changed because the arguments will be the exact same. Simply, when calling the super method pass in the new method instance instead as so: super.visitMethodInsn(opcode, owner, newName, desc, itf); , where newName corresponds to the name of the new method that will is to be invoked in place of the current method.

4.11 M4: Replace Local Variable

This mutator's purpose is to replace an instance of a local variable with another local variable of the same data type. Whenever a ISTORE, LSTORE, FSTORE, DSTORE, or ASTORE occurs in a java class, store the variable index and type into a list. Then, whenever a ILOAD, LLOAD, FLOAD, DLOAD, or ALOAD occurs in a java class, utilize the list to find a random variable with the same type to replace it with. If during a load instance a variable index is not in the list of indices to replace with, store it in the list for future variable to use it as a replacement. The purpose of this is because the parameters of a method are do not have store instance associated with them since they are passed variables, but they do have loads associated with them. Finally, to apply the mutation simply perform the original operation of the opcode (one of the stores) but utilize the index of the replacement variable.

4.12 Let Pitest Know New Mutations Exist

After new mutations have been implemented as mentioned above, Pitest needs to learn of the new mutations. To do this, mutattors.java needs to include an import of each new java file created. After, each mutator within the java files needs to be added by giving it a name that can be referenced from the pom.xml files of projects using Pitest to be mutated, as well with the name of the mutator named in the java files.

5 EXPERIMENATION

5.1 Design and Test Subjects

After each step each step of the implementation, PIT tool and code added to the tool should be tested for accuracy. The same test suites will be utilized after each step. First, the output of each test should be noted, followed by output of mutated code by default mutations of the PIT test tool. These will be the "constant" in the experiment to see if after implementation between the second and last steps have improved the mutation score and speed of execution. Furthermore, random selection of mutations will be compared to default selection of mutations as well if time permits.

It is recommended to use at least five real-world Java projects from GitHub, however more tests are always better not just to help fix bugs in the PIT program but also be able to collect more data to see if the new implementations have worked or not. The projects will be selected at random for a list, and the projects already have JUNIT tests readily available. The JUNIT tests will be checked to see if they support reachability, necessity, and sufficiency in the code base.

Furthermore, to check for M1, M2, M3, M4 mutators, 4 projects are utilized to test that they work. These projects are listed in figure 11 below.

Program	Directory	Suitable for Testing
Apache Commons Codec	commons-codec	M1
Joda Time	joda-time	M2
Apache Commons Lang	commons-lang	M3
JFree Chart	jfreechart	M4

Figure 11: Programs corresponding to which mutator can be detected from M1 – M4

5.2 Evaluation and Results Phase 1

The following results were compared between the original Pitest and the Pitest with the additional mutators implemented (AOD, ROR, AOR).

Github Name	Pitest Mutation Killed	Pitest extended Killed	Extension better?
Jsfreechart-fse	32%	20%	None
Jumbler	53%	50%	yes
Mbassador	6%	3%	None
calculon	7%	6%	yes
caesar	87%	78%	yes

Figure 12: Comparisons of 5 github projects with over 1000 lines of code and at least 5 test between original pitest mutation coverage and pitest extension coverage.

Clearly seen within figure 9, the extended pitest with AOD , AOR, and ROR with defaults covers more mutations than does the normal Pitest results utilizing only default mutators. The percentage of mutations inserted and killed decreased with the addition of more mutators, as seen in figure 12, meaning the projects tested on contain more bugs that are not able to survive and should be fixed.

5.3 Evaluation and Results M1

M1 was tested against Apache Commons Codec project's org.jfree.chart.renderer class package. It was found that after placing a field dereference null check, the mutator did in deed fix bugs in the code. After testing with all mutations this code:

```
if (myContext.eof) {
    return;
}
```

resulted in:

```
1. Placed null check before field dereference → SURVIVED
2. negated conditional → KILLED
3. Replacments: == to > , != to > , < to > , <= to != , >= to > , > to != → KILLED
4. Replacments: == to < , != to < , < to != , <= to < , >= to != , > to < → KILLED
5. Replacments: == to >= , != to >= , < to == , <= to >= , >= to == , > to >= → KILLED
6. Replacments: == to <= , != to <= , < to <= , <= to == , >= to <= , > to == → KILLED
7. Replaced Object with Object → NON_VIABLE
8. removed conditional - replaced equality check with false → KILLED
9. removed conditional - replaced equality check with true → KILLED
```

The results clearly show that when the null check before field dereference was placed in the code, the mutation skipped placing the "return" statement and hence the mutation survived and indicated that there is a bug in the code at this point. None of the other mutators allowed the mutation to survive, hence they never showed the bug in the code.

5.2 Evaluation and Results M4

M4 had similar excellent results in fixing a bug in a code. This mutator was tested on JFree Chart's org.jfree.chart.renderer class as this intentionally has a bug placed in it. Consider this piece of code:

```
public Paint getPaint(double value) {
    double v = Math.max(value, this.lowerBound);
    v = Math.min(v, this.upperBound);
    int g = (int) ((value - this.lowerBound) / (this.upperBound
            - this.lowerBound) * 255.0);
    // FIXME:  it probably makes sense to allocate an array of 256
    // and lazily populate this array...
    return new Color(g, g, g, this.alpha);
}
```

Resulted in this after placing all mutations into the test:

```
1. Replaced DOUBLE A - B with A → KILLED
2. Replaced DOUBLE A - B with A → KILLED
3. Replaced DOUBLE A / B with A → KILLED
4. Replaced DOUBLE A * B with A → KILLED
5. Replaced DOUBLE A - B with B → KILLED
6. Replaced DOUBLE A - B with B → KILLED
7. Replaced DOUBLE A / B with B → KILLED
8. Replaced DOUBLE A * B with B → KILLED
9. Replaced double subtraction with multiplication → KILLED
10. Replaced double subtraction with multiplication → KILLED
11. Replaced double division with addition → KILLED
12. Replaced double multiplication with addition → KILLED
13. Replaced integer subtraction with division → KILLED
14. Replaced integer subtraction with division → KILLED
15. Replaced integer division with subtraction → KILLED
16. Replaced integer multiplication with subtraction → KILLED
17. Replaced double subtraction with modulus → KILLED
18. Replaced double subtraction with modulus → KILLED
19. Replaced double division with modulus → KILLED
20. Replaced double multiplication with modulus → KILLED
21. Placed null check before field dereference → KILLED
22. Placed null check before field dereference → KILLED
23. Placed null check before field dereference → KILLED
24. Substituted 255.0 with 1.0 → KILLED
25. Inverted sign of Double Variable → KILLED
26. Replaced double subtraction with addition → KILLED
27. Replaced double subtraction with addition → KILLED
28. Replaced double division with multiplication → KILLED
29. Replaced double multiplication with division → KILLED
30. Replaced Double with Double → SURVIVED
31. Double TEMP=A+B mutated to TEMP=A+A → KILLED
32. Double TEMP=A+B mutated to TEMP=A+A → KILLED
33. Double TEMP=A+B mutated to TEMP=A+A → KILLED
34. Double TEMP=A+B mutated to TEMP=A+A → KILLED
35. Double TEMP=A+B mutated to TEMP=B+B → KILLED
36. Double TEMP=A+B mutated to TEMP=B+B → KILLED
37. Double TEMP=A+B mutated to TEMP=B+B → KILLED
38. Double TEMP=A+B mutated to TEMP=B+B → KILLED
39. Added double decrement → KILLED
40. Double add 1 (++x) → KILLED
```

Clearly, M4 was able to find a bug in the program. After replacing a double variable with another double variable in this line:

```
int g = (int) ((value - this.lowerBound) / (this.upperBound
        - this.lowerBound) * 255.0);
```

The M4 mutator was the only one to place a mutation in the code that survived, indicating that there is a bug in the code. All other mutators were killed, and hence they did not indicate a problem in the code here.

A APPENDIX

A.1 Introduction

A.2 Problem

A.3 Techniques

A.3.1 Magnetization Curves and MFM Characterization

A.3.2 Field Dependent BLS Measurements and DMM Calculations

A.3.3 Analysis of the Dynamic Coupling as a Function of the Gap Size

A.4 Implementation

A.4.1 Planning

A.4.2 Creating New Mutators

A.4.3 Arithmetic Operation Deletion

A.4.4 Relational Operation Replacement & Arithmetic Operation Replacement

A.4.5 Inversion

A.4.6 Unary Operator Insertion

A.4.7 Scalar Variable Replacement

A.4.8 M1: Field Dereference Null Check

A.4.9 M2: Replace Overloading Method Arguments

A.4.10 M3: Replace Overloading Method Name

A.4.11 M4: Replace Local Variable

A.4.11 Let Pitest Know New Mutations Exist

A.5 Experimentation

A.5.1 Design and Test Subjects

A.5.2 Evaluation and Results Phase 1

A.5.3 Evaluation and Results M1

A.5.4 Evaluation and Results M4

A.5 References

ACKNOWLEDGMENTS

This work was partially learned from Dr. Lingming Zhang and Mr. Ali Ghanbari at the University of Texas at Dallas in Richardson, Texas through a Software Testing, Validation, and Verification course in Spring of 2018.

REFERENCES

[1] Dr. Lingming Zhang Notes

[2] Real world mutation testing. (n.d.). Retrieved February 07, 2018, from http://pitest.org/

Github link: https://github.com/arjun-sethi/extendedPIT