How Effective Mutation Testing Tools Are? An Empirical Analysis of Java Mutation Testing Tools with Manual Analysis and Real Faults

Marinos Kintis*, Mike Papadakis*, Andreas Papadopoulos†, Evangelos Valvis†, Nicos Malevris† and Yves Le Traon*

Abstract

Mutation analysis is a popular fault-based testing technique. It requires testers to design tests based on a set of artificial defects. The defects help in performing testing activities by measuring their ratio that is revealed by the candidate tests. Unfortunately, applying mutation to real world programs requires automated tools due to the vast number of the involved defects. In such a case, the strengths of the method strongly depend on the peculiarities of the employed tools. Thus, when employing automated tools, their implementation inadequacies can lead to inaccurate results. To deal with this issue, we cross-evaluate three popular mutation testing tools for Java, namely MUJAVA, MAJOR and the research version of PIT, PIT_{RV}, with respect to their fault detection capabilities. We investigate the strengths of the tools based on: a) a set of real faults and b) manual analysis of the mutants they introduce. We find that there are large differences between the tools' effectiveness and demonstrate that no tool is able to subsume the others. We also provide results indicating the application cost of the method. Overall, we find that PIT_{RV} achieves the best results. In particular, PIT_{RV} outperforms both MUJAVA and MAJOR by finding 6% more faults than both of the other two tools together.

1 Introduction

Software testing forms the most popular practice for identifying software defects [1]. It is performed by exercising the software under test with test cases that check whether its behaviour is as expected. To analyse test thoroughness, several criteria, which specify the requirements of testing, i.e., what constitutes a good test suite, have been proposed. When the criteria requirements have been fulfilled they provide confidence on the function of the tested systems.

Empirical studies have demonstrated that mutation testing is effective in revealing faults [7], and capable of subsuming, or probably subsuming, almost all the structural testing techniques [1, 19]. Mutation testing requires test cases that reveal the artificially injected defects. This practice is particularly powerful as it has been shown that when test cases are capable of distinguishing the behaviours of the original (non-mutated) and

*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg, ({marinos.kintis, michail.papadakis, yves.letraon}@uni.lu)

†Department of Informatics, Athens University of Economics and Business, Greece, ({p3100148, p3130019, ngm}@aueb.gr)

the defective (mutant) programs, they are also capable of distinguishing the expected behaviours from the faulty ones [7].

The defective program versions are called *mutants* and they are typically introduced using syntactic transformations. Clearly, the effectiveness of the technique depends on the mutants that are employed. For instance if the mutants are trivial, i.e., they are found by almost every test that exercises them, they do not contribute the testing process. Therefore, testers performing mutation testing should be cautious about the mutants they use. Recent research has demonstrated that the method is so sensitive to the employed mutants so that it can lead experiments to incorrect conclusions [41]. Therefore, particular care has to be taken when selecting mutants in order to avoid potential threats to validity. Similarly, the use of mutation testing tools can lead to additional threats to validity or incompetent results (due to the peculiarities of the mutation testing tools).

To date, many mutation testing tools have been developed and used by researchers and practitioners [41]. However, a key open question is how effective these tools are and how reliable are the research results based on them. Thus, in this paper, we seek to investigate the fault revelation ability of popular mutation testing tools with the goal of identifying their differences, weaknesses and strengths. In short, our aim is threefold: a) to inform practitioners about the effectiveness and relative cost of the studied mutation testing tools, b) to provide constructive feedback to tool developers on how to improve their tools, and c) to make researchers aware of the tools' inadequacies.

To investigate these issues, we compare the fault revelation ability of three widely-used mutation testing tools for Java, namely MUJAVA, MAJOR and PIT_{RV} on a set of real faults. We complement our analysis using human analysis and comparison of the tools. Our results demonstrate that one tool, the research version of PIT [8], named PIT_{RV} [18], is significantly more effective than the others, managing to reveal approximately 6% more real faults than the other two tools together. However, due to some known limitations of PIT_{RV}, it cannot fully subsume the other tools.

Regarding a reference effectiveness measure (control comparison at 100% coverage level), we found that PIT_{RV} scores best with 91%, followed by MUJAVA with 85% and MAJOR with 80%. These results suggest that existing tools have a much lower effectiveness than what they should or what researchers believe they ought to. Therefore, our findings emphasise the need to build a reference mutation testing tool that will be strong enough and capable of at least subsuming the existing mutation testing tools.

Another concern, when using mutation, is its application cost. This is mainly due to the manual effort involved in constructing test cases and due to the effort needed for deciding when to stop testing. The former point regards the need for generating test cases while the latter pertains to the identification of the so-called *equivalent mutants*, i.e., mutants that are functionally equivalent to the original program. Both these tasks are labour-intensive and should be performed manually. Our study shows that MUJAVA leads to 138 tests, MAJOR to 97 and PIT_{RV} to 105. With respect to the number of equivalent mutants, MUJAVA, MAJOR and PIT_{RV} produced 203, 94 and 382, respectively.

This paper forms an extended study of our previous one [26], published in the International Working Conference on Source Code Analysis and Manipulation, which investigated the effectiveness of the tools based on manual analysis. We extend this previous study by investigating the actual fault revelation ability of the tools, based on a benchmark set of real faults and by considering the research version of the PIT tool, which was realised after the previous study [18]. The extended results demonstrate that PIT_{RV} forms the most prominent choice as it significantly outperforms the other tools

both in terms of fault revelation and mutant revelation. Overall, the contributions of the present paper can be summarised in the following points:

1. A controlled study investigating the fault revelation ability of three, widely-used mutation testing tools for the Java programming language.
2. An extensive, manual study of 5,831 mutants investigating the strengths and weaknesses of the Java mutation testing tools considered.
3. Insights on the relative cost of the tools' application in terms of the number of equivalent mutants that have to be manually analysed and the number of test cases that have to be generated.
4. Recommendations on specific mutation operators that need to be implemented in these tools in order to improve their effectiveness.

The rest of the paper is organised as follows: Section 2 presents the necessary background information and Section 3 outlines our study's motivation. In Section 4, we present the posed research questions and the adopted experimental procedure and, in Section 5, we describe the obtained results. In Section 6, we discuss potential threats to the validity of this study, along with mitigating actions and in Section 7, previous research studies. Finally, Section 8 concludes this paper, summarising the key findings.

2 Background

This section details mutation testing and presents the studied mutation testing tools.

2.1 Mutation Testing

Applying mutation testing requires the generation and execution of a set of mutants with the candidate test cases. Mutants are produced using a set of syntactic rules called *mutation operators*. The process requires practitioners to design test cases that are able to distinguish the mutants' behaviour from that of the program under test, termed *original program* in mutation's terminology. In essence, these test cases should force the original program and its mutants to result in different outputs, i.e., they should *kill* its mutants.

Normally, the ratio of the killed mutants to the generated ones is an effectiveness measure that should quantify the ability of the test cases to reveal the system's defects. Unfortunately, among the killable mutants, there are some that cannot be killed, termed *equivalent mutants*. Equivalent mutants are syntactically different versions of the program under test, but semantically equivalent [40, 22]. These mutants must be discarded in order to have an accurate effectiveness measure, which is called *Mutation score*, i.e., the ratio of killed mutants to the number of killable mutants, and to decide when to stop testing. The problem of identifying and removing equivalent mutants is known as the *Equivalent Mutant Problem* [40, 22].

Regrettably, the Equivalent Mutant Problem has been shown to be undecidable in its general form [6], thus, no complete, fully automated solution can be devised to tackle it. This problem is largely considered an open issue in mutation's literature, but recent advances provide promising results towards practical, automated solutions, albeit partial, e.g., [40, 25, 23].

Another problem of mutation testing is that it produces many mutants that are redundant, i.e., they are killed when other mutants are killed. These mutants can inflate the mutation score making it skew. Thus, previous research has shown that these mutants can have harmful effects on the mutation score measurement with the effect of leading experiments to incorrect conclusions [41]. Therefore, when mutation testing is used as a comparison basis, there is a need to deflate the mutation score measurement. This can be done by using the subset of subsuming mutants [41, 2] or disjoint mutants [24]. Disjoint mutants approximate the minimum "subset of mutants that need to be killed in order to reciprocally kill the original set" [24]. We utilise the term *disjoint mutation score* for the ratio of the disjoint mutants that are killed by the test cases under assessment (which in our case are those that were designed to kill the studied tools' mutants).

Mutation's effectiveness depends largely on the mutants that are used [1]. Thus, the actual implementation of mutation testing tools can impact the effectiveness of the technique. Indeed, many different mutation testing tools exist that are based on different architectural designs and implementations. As a consequence, it is not possible for researchers, and practitioners alike, to make an informed decision on which tool to use and on the strengths and weaknesses of the tools.

This paper addresses the aforementioned issue by analysing the effectiveness of three widely-used mutation testing tools for the Java programming language, namely MUJAVA, MAJOR and PIT_{RV}, based on the results of an extensive manual study. Before presenting the conducted empirical study, the considered tools and their implementation details are introduced.

2.2 Selected tools

Mutation is popular [41] and, thus, many mutation testing tools exist. In this study we choose to work in Java since it is widely used by practitioners and forms the subject of most of the recent research papers. To select our subject tools, we performed a mini literature survey on the papers published during 2014 and 2015 in the three leading Software Engineering conferences (ISSTA, (ESEC)FSE and ICSE) and identified the mutation testing tools that were used. The analysis resulted in three tools, MUJAVA [31], MAJOR [20] and PIT [8].

2.2.1 MUJAVA – Source Code Manipulation

MUJAVA [31] is one of the oldest Java mutation testing tools and has been used in many mutation testing studies. It works by directly manipulating the source code of the program under test and supports both method-level and class-level mutation operators. The former handle primitive features of programming languages, such as arithmetic operators, whereas the latter handle object-oriented features, such as inheritance. Note that MUJAVA adopts the selective mutation approach [33], i.e., it implements a set of 5 operators whose mutants subsume the mutants generated by other mutation operators not included in this set. Table 1 presents the method-level operators of the tool, along with a succinct description of the performed changes. For instance, AORB replaces binary arithmetic operators with each other and AODS deletes the ++ and -- arithmetic operators.

Table 1: Mutation operators of MUJAVA

Mutation Operator	Description
AORB: *Arithmetic Operator Replacement Binary*	$\{(op_1, op_2) \mid op_1, op_2 \in \{+, -, *, /, \%\} \wedge op_1 \neq op_2\}$
AORS: *Arithmetic Operator Replacement Short-Cut*	$\{(op_1, op_2) \mid op_1, op_2 \in \{++, --\} \wedge op_1 \neq op_2\}$
AOIU: *Arithmetic Operator Insertion Unary*	$\{(v, -v)\}$
AOIS: *Arithmetic Operator Insertion Short-cut*	$\{(v, --v), (v, v--), (v, ++v), (v, v++)\}$
AODU: *Arithmetic Operator Deletion Unary*	$\{(+v, v), (-v, v)\}$
AODS: *Arithmetic Operator Deletion Short-cut*	$\{(--v, v), (v--, v), (++v, v), (v++, v)\}$
ROR: *Relational Operator Replacement*	$\{((\mathtt{a}\ op\ \mathtt{b}), \mathtt{false}), ((\mathtt{a}\ op\ \mathtt{b}), \mathtt{true}), (op_1, op_2) \mid op_1, op_2 \in \{>, >=, <, <=, ==, !=\} \wedge op_1 \neq op_2\}$
COR: *Conditional Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\&\&, \vert\vert, \hat{}\} \wedge op_1 \neq op_2\}$
COD: *Conditional Operator Deletion*	$\{(!cond, cond)\}$
COI: *Conditional Operator Insertion*	$\{(cond, !cond)\}$
SOR: *Shift Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{>>, >>>, <<\} \wedge op_1 \neq op_2\}$
LOR: *Logical Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\&, \vert, \hat{}\} \wedge op_1 \neq op_2\}$
LOI: *Logical Operator Insertion*	$\{(v, \sim v)\}$
LOD: *Logical Operator Deletion*	$\{(\sim v, v)\}$
ASRS: *Short-Cut Assignment Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{+=, -=, *=, /=, \%=, \&=, \vert=, \hat{}=, >>=, >>>=, <<=\} \wedge op_1 \neq op_2\}$

2.2.2 PIT_{RV} – Bytecode Manipulation

PIT_{RV} [18] is the research version of PIT [8] which is a mutation testing framework that targets primarily the industry but has also been used in many research studies. PIT works by manipulating the resulting bytecode of the program under test and employs mutation operators that affect primitive programming language features, similarly to the method-level operators of MUJAVA. PIT_{RV} greatly extends PIT's supported mutation operators with the aim of improving the tool's effectiveness. It is noted that in the conference version of this paper we used the original version of PIT and found that it was significantly less effective than both MUJAVA and MAJOR [26]. Therefore, here we investigate how does the PIT_{RV} compares with the other tools.

Table 2 describes the corresponding operators. By comparing this table with Table 1, it can be seen that PIT_{RV} implements most of MUJAVA's mutation operators, while implementing some in a different way. For instance, the changes imposed by PIT_{RV}'s

Bitwise Operator Mutation (OBBN) are a subset of the ones of MUJAVA's Logical Operator Replacement (LOR). Additionally, it employs mutation operators that are not implemented in MUJAVA, e.g. the Void Method Calls (VMC) and Constructor Calls (CC) operators. Finally, it should be mentioned that since PIT_{RV}'s changes are performed at the bytecode level, they cannot always be mapped onto source code ones.

Table 2: Mutation operators of PIT_{RV}

Mutation Operator	Description
ABS: *Absolute Value Insertion*	$\{(v,\texttt{-}v)\}$
AOD: *Arithmetic Operator Deletion*	$\{((\text{a}\ op\ \text{b}),\ \texttt{a}),(\text{a}\ op\ \text{b}),\ \texttt{b}))\mid \text{op}\in\{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\}\}$
AOR: *Arithmetic Operator Replacement*	$\{(op_1,op_2)\mid op_1,op_2\in\{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\}\wedge op_1\neq op_2\}$
AP: *Argument Propagation*	$\{(\mathit{nonVoidMethodCall}(...,\ par),\ par)\}$
CRCR: *Constant Replacement*	$\{(\text{const},-\text{const}),(\text{const},0),(\text{const},1),(\text{const},\text{const}-1),(\text{const},\text{const}+1)\}$
CB: *Conditionals Boundary*	$\{(op_1,op_2)\mid(op_1,op_2)\in\{(\texttt{<},\texttt{<=}),(\texttt{<=},\texttt{<}),(\texttt{>},\texttt{>=}),(\texttt{>=},\texttt{>})\}\}$
CC: *Constructor Calls*	$\{(\texttt{new}\ \mathit{AClass}(),\texttt{null})\}$
I: *Increments*	$\{(op_1,op_2)\mid op_1,op_2\in\{\texttt{++},\texttt{--}\}\wedge op_1\neq op_2\}$
IC: *Inline Constant*	$\{(c_1,c_2)\mid(c_1,c_2)\in\{(1,0),((\texttt{int})\ x,\ \texttt{x+1}),(1.0,0.0),(2.0,0.0),((\texttt{float})\ x,\ 1.0),(\texttt{true},\ \texttt{false}),(\texttt{false},\texttt{true})\}\}$
IN: *Invert Negatives*	$\{(\texttt{-}v,v)\}$
M: *Math*	$\{(op_1,op_2)\mid(op_1,op_2)\in\{(\texttt{+},\texttt{-}),(\texttt{-},\texttt{+}),(\texttt{*},\texttt{/}),(\texttt{/},\texttt{*}),(\texttt{\%},\texttt{*}),(\texttt{\&},\vert),(\vert,\texttt{\&}),(\texttt{\^{}},\texttt{\&}),(\texttt{<<},\texttt{>>}),(\texttt{>>},\texttt{<<}),(\texttt{>>>},\texttt{<<})\}\}$
MV: *Member Variable*	$\{(\texttt{member_var=...},\texttt{member_var=b})\mid b\in\{\texttt{false},0,0.0,\ \texttt{'\textbackslash u0000'},\texttt{null}\}\}$
NC: *Negate Conditionals*	$\{(op_1,op_2)\mid(op_1,op_2)\in\{(\texttt{==},\texttt{!=}),(\texttt{!=},\texttt{==}),(\texttt{<=},\texttt{>}),(\texttt{>=},\texttt{<}),(\texttt{<},\texttt{>=}),(\texttt{>},\texttt{<=})\}\}$
NVMC: *Non Void Method Calls*	$\{(\mathit{nonVoidMethodCall}(),\ c)\mid c\in\{\texttt{false},0,0.0,\texttt{'\textbackslash u0000'},\texttt{null}\}\}$
OBBN: *Bitwise Operator Mutation*	$\{(op_1,op_2)\mid op_1,op_2\in\{\texttt{\&},\vert\}\wedge op_1\neq op_2\}$
ROR: *Relational Operator Replacement*	$\{(op_1,\ op_2)\mid op_1,op_2\in\{\texttt{>},\texttt{>=},\texttt{<},\texttt{<=},\ \texttt{==},\texttt{!=}\}\wedge op_1\neq op_2\}$
RC: *Remove Conditionals*	Removes or negates a conditional statement to force or prevent the execution of the guarded statements, e.g. $\{((\text{a}\ op\ \text{b}),\ \texttt{true})$ *or* $((\texttt{LHS \&\& RHS}),\ \texttt{RHS})\}$
RI: *Remove Increments*	$\{(\texttt{--}v,v),(v\texttt{--},v),(\texttt{++}v,v),(v\texttt{++},v)\}$
RS: *Remove Switch*	Changes all labels of the `switch` to the default one

Table 2: Mutation operators of PIT_{RV}

Mutation Operator	Description
RV: *Return Values*	$\{(\texttt{return}\ a, \texttt{return}\ b) \mid (a,b) \in \{(\texttt{true},\texttt{false}),(\texttt{false},\texttt{true}),(0,1), ((\texttt{int})\ x,0),((\texttt{long})\ \texttt{x},\texttt{x+1}), ((\texttt{float})\ x, \texttt{-(x+1.0)}),(\texttt{NAN},0),(\texttt{non-null},\texttt{null}), (\texttt{null},\texttt{throw}\ \textit{RuntimeException})\}\}$
S: *Switch*	Replaces the `switch`'s labels with the default one and vice versa (only for the first label that differs)
UOI: *Unary Arithmetic Operator Insertion*	$\{(v,\texttt{--}v),(v,v\texttt{--}),(v,\texttt{++}v),(v,v\texttt{++})\}$
VMC: *Void Method Calls*	$\{(\textit{voidMethodCall}(),\varnothing)\}$

2.2.3 MAJOR – AST MANIPULATION

MAJOR [20] is a mutation testing framework whose architectural design places it between the aforementioned ones: it manipulates the abstract syntax tree (AST) of the program under test. MAJOR employs mutation operators that have similar scope to the previously-described ones. The implemented mutation operators of the tool are based on selective mutation, similarly to MUJAVA. Table 3 summarises MAJOR's operators and their imposed changes. Compared to MUJAVA's operators, it is evident that the two tools share many mutation operators, but implement them differently. Compared to PIT_{RV}, most operators of MAJOR impose a superset of changes with respect to the corresponding ones of PIT_{RV} and there are operators of PIT_{RV} that are completely absent from MAJOR.

3 Motivation

Mutation testing is important since it is considered as one of the most effective testing techniques. Its fundamental premise, as coined by Geist et al. [15], is that:

> *"If the software contains a fault, it is likely that there is a mutant that can only be killed by a test case that also reveals the fault."*

This premise has been empirically investigated by many research studies which have shown that mutation adequate test suites, i.e., test suites that kill all killable mutants, are more effective than the ones generated to cover various control and data flow coverage criteria [19, 7]. Therefore, researchers use mutation testing as a way to either compare other test techniques or as a target to automate.

Overall, a recent study by Papadakis et al. [41] shows that mutation testing is popular and widely-used in research (probably due to its remarkable effectiveness). In view of this, it is mandatory to ensure that mutation testing tools are powerful and do not bias (due to implementation inadequacies or missing mutation operators) the existing research.

To reliably compare the selected tools, it is mandatory to account for mutant subsumption [41] when performing a complete testing process, i.e., using mutation-adequate tests. Accounting for mutant subsumption is necessary in order to avoid bias from subsumed mutants [41], while complete testing ensures the accurate estimation of the tools' effectiveness. An inaccurate estimation may happen when failing to kill some

Table 3: Mutation operators of MAJOR

Mutation Operator	Description
AOR: *Arithmetic Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\} \wedge op_1 \neq op_2\}$
LOR: *Logical Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{\texttt{\&},\vert,\texttt{\^{}}\} \wedge op_1 \neq op_2\}$
COR: *Conditional Operator Replacement*	$\{(\texttt{\&\&},op_1), (\vert\vert,op_2) \mid op_1 \in \{\texttt{==},\texttt{LHS},\texttt{RHS},\texttt{false}\}, op_2 \in \{\texttt{!=},\texttt{LHS},\texttt{RHS},\texttt{true}\}\}$
ROR: *Relational Operator Replacement*	$\{(\texttt{>},op_1),(\texttt{<},op_2),(\texttt{>=},op_3),(\texttt{<=},op_4),(\texttt{==},op_5),(\texttt{!=},op_6) \mid op_1 \in \{\texttt{>=},\texttt{!=},\texttt{false}\}, op_2 \in \{\texttt{<=},\texttt{!=},\texttt{false}\}, op_3 \in \{\texttt{>},\texttt{==},\texttt{true}\}, op_4 \in \{\texttt{<},\texttt{==},\texttt{true}\}, op_5 \in \{\texttt{<=},\texttt{>=},\texttt{false},\texttt{LHS},\texttt{RHS}\}, op_6 \in \{\texttt{<},\texttt{>},\texttt{true},\texttt{LHS},\texttt{RHS}\}\}$
SOR: *Shift Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{\texttt{>>},\texttt{>>>},\texttt{<<}\} \wedge op_1 \neq op_2\}$
ORU: *Operator Replacement Unary*	$\{(op_1,op_2) \mid op_1,op_2 \in \{\texttt{+},\texttt{-},\sim\} \wedge op_1 \neq op_2\}$
STD: *Statement Deletion Operator*	$\{(\texttt{--}v,v),(v\texttt{--},v),(\texttt{++}v,v),(v\texttt{++},v),(aMethodCall(),\ \varnothing),\ (a\ op_1\ b,\ \varnothing) \mid op_1 \in \{\texttt{+=},\texttt{-=},\texttt{*=},\texttt{/=},\texttt{\%=},\texttt{\&=},\vert\texttt{=},\texttt{\^{}=},\texttt{>>=},\texttt{>>>=},\texttt{<<=}\}\}$
LVR: *Literal Value Replacement*	$\{(c_1,c_2) \mid (c_1,c_2) \in \{(0,1),(0,-1),(c_1,-c_1),(c_1,0),(\texttt{true},\texttt{false}),(\texttt{false},\texttt{true})\}$

killable mutants, which consequently results in failing to design tests (to kill these mutants) and, thus, underestimate effectiveness. Even worse, the use of non-adequate test suites ignores hard to kill mutants which are important [3, 43] and among those that (probably) contribute to the test process. Since we know that very few mutants contribute to the test process [41], the use of non-adequate test suites can result in major degradation of the measured effectiveness.

Unfortunately, generating test suites that kill all killable mutants is practically infeasible because of the inherent undecidability of the problem [40]. Therefore, from a practical point of view, examining the partial relationship for the case of non-adequate test suites is important. Thus, we need to consider both scenarios in order to adequately compare the tools we study. For all these reasons, we use both mutation adequate test suites specially designed for each tool that we study (using a small sample of programs) and non-adequate test suites (using large real world programs).

4 Empirical Study

This section presents the settings of our study, by detailing the research questions, the followed procedure and the design of our experiments.

4.1 Research Questions

Mutation testing's aim is to help testers design high quality test suites. Therefore, the first question to ask is whether there is a tool that is more effective or at least as effective as the other tools in real world cases. In other words, we want to measure how effective are the studied tools in finding real faults. Since we investigate real faults, we are forced to study the partial relationship between the tools under the "practical" scenario. Hence we ask:

RQ1: *How do the studied tools perform in terms of real fault detection?*

This comparison enables checking whether mutation testing tools have different fault revelation capabilities when applied to large real world projects. In case we find that the tools have significant differences in terms of fault detection, we demonstrate that the choice of mutation testing tools really matters. Given that we find significant differences between the tools, a natural question to ask is:

RQ2: *Does any mutation testing tool lead to tests that subsume the others in terms of real fault detection? If not, which is the relatively most effective tool to use?*

This comparison enables the ranking of the tools with respect to their fault revelation capabilities (with respect to the benchmark set we use) and identifying the most effective mutation testing tool. It also quantifies the effectiveness differences between the tools in real world settings. Given that the effectiveness ranking offered by the above comparison is bounded to the reference fault set and the automatically generated test suites used, an emerging question is how the tools compare with each other under complete testing, i.e., using adequate test suites. In other words, we seek to investigate how effective are the studied tools in killing the mutants of the other tools. Hence we ask:

RQ3: *Does any mutation testing tool lead to tests that kill all the killable mutants produced by the other tools?*

This comparison enables checking whether there is a tool that is capable of subsuming the others, i.e., whether the mutation adequate tests of one tool can kill all the killable mutants of the others. A positive answer to the above question indicates that a single tool is superior to the others, in terms of effectiveness. A negative answer to this question indicates that the tools are generally incomparable, meaning that there are mutants not covered by the tools. We view these missed mutants as weaknesses of the tools. The main differences from the RQ1 and RQ2 are that we perform an objective comparison under complete testing, which helps reducing potential threats to validity.

To further compare mutation testing tools and identify their weaknesses, we need to assess the quality of the test suites that they lead to. This requires either an independent, to the used mutants, effectiveness measure or a form of "ground truth", i.e., a golden set of mutants. Since both are not known, we constructed a reference mutant set, the set of disjoint mutants, from the superset of mutants produced by all the studied tools together and all generated test cases. We use the disjoint set of mutants to avoid inflating the reference set from the duplicated, i.e., mutants equivalent to each other but not to the original program [40], and redundant mutants, i.e., mutants subsumed by other mutants of the merged set of mutants [41]. Both duplicated and redundant mutants inflate the mutation score measurement with the unfortunate result of committing Type I errors [41]. Since in our case these types of mutants are expected to be numerous, as the tools

support many common types of mutants, the use of disjoint mutants was imperative. Therefore we ask:

RQ4: *How do the studied tools perform compared to a reference mutant set? Which is the relatively most effective tool to use?*

This comparison enables the ranking of the tools with respect to their effectiveness. The use of the reference mutant set also helps aggregate all the data and quantify the relative strengths and weaknesses of the studied tools in one measure (the disjoint mutation score). Given the effectiveness ranking offered by this comparison, we can identify the most effective mutation testing tool and quantify the effectiveness differences between the tools.

This is important when choosing a tool to use but does not provide any constructive information on the weaknesses of the tools. Furthermore, this information fails to provide researchers and tool developers constructive feedback on how to build future tools or strengthen the existing ones. Therefore, we seek to analyse the observed weaknesses and ask:

RQ5: *Are there any actionable findings on how to improve the effectiveness of the studied tools?*

Our intentions thus far have been concentrated on the relative effectiveness of the tools. While this is important when using mutation, another major concern is the cost of its application. Mutation testing is considered to be expensive due to the manual effort involved in identifying equivalent mutants and designing test cases. Since we manually assess and apply the mutation testing practice of the studied tools we ask:

RQ6: *What is the relative cost, measured by the number of tests and number of equivalent mutants, of applying mutation testing with the studied tools?*

An answer to this question can provide useful information to both testers and researchers regarding the trade-offs between cost and effectiveness. Also, this analysis will better reflect the differences of the tools from the cost perspective.

4.2 Assessing Fault Revelation Ability

In order to assess the fault revelation capabilities of the studied tools, we used Defects4J [21] (version 1.1.0), which is a benchmark set of reproducible real faults mined from source code repositories. The benchmark is composed of 395 faults carefully located and isolated using the version control and bug tracking systems of 6 open source projects. For each one of the faults, the benchmark contains a buggy and a fixed program version and at least one test case that reproduces the faulty behaviour. Table 4 presents the name of the projects (first column), a small description of their application domain (second column), the source code lines as reported by the `cloc` tool (third column) and the number of faults available per project (fourth column). Additional details about the benchmark set and its construction can be found in the demo paper of the Defects4J [21] and on its GitHub page[1].

Unfortunately, the benchmark contains only some developer tests that were recorded on the repository. This is potentially problematic in our case as developer tests are generally weak (they typically achieve low coverage) and as a consequence very few of

[1] https://github.com/rjust/defects4j

Table 4: Fault Benchmark Set Details

Test Subject	Description	LoC	#Real Faults	#Gen. Tests	#Faults Found
JFreeChart	A chart library	79,949	26	3,758	17
Closure	Closure compiler	91,168	133	3,552	12
Commons Lang	Java utilities library	45,639	65	6,408	30
Commons Math	Mathematics library	22,746	106	8,034	53
Mockito	A mocking framework	5,506	38	-	-
Joda-Time	A date and time library	79,227	27	1,667	13
Total	-	**324,235**	**395**	**23,419**	**125**

these tests reveal the faults. Furthermore, these have not been generated using any controlled or known procedure and thus, they can introduce several threats to the validity of our results as they are few and may only kill trivial mutants, underestimating our measurements. To circumvent this problem, we simulated the mutation-based test process using multiple test suites generated by two state-of-the-art test generation tools, namely EvoSuite [13] and Randoop [35]. Although this practice may introduce the same threats to validity as the developer test suites, it has several benefits as the tests are generated with a specific procedure, they are multiple and they represent our current ability of generating automated test suites.

4.2.1 Automated Test Suite Generation

For the generation of the test suites, we used version 1.0.3 of EvoSuite and 3.0.8 of Randoop. To configure and execute the tools, we used the scripts accompanying Defects4J. Overall, we proceed with the following procedure:

1. For each fault, we run EvoSuite and Randoop on the fixed version of the project to generate 3 test suites, two with EvoSuite and one with Randoop, for the classes that were modified in order to fix the corresponding buggy version.

2. We systematically removed problematic test cases from the generated test suites, e.g. test cases that produced inconsistent results when run multiple times, using the available scripts of Defects4J.

3. Run the generated test suites against the buggy versions of the projects to identify the faults they reveal.

The aforementioned procedure identified the faults (from all the faults of Defects4J) that could be discovered by our automatically generated test suites. In the remainder of the paper, we call as "triggering test suite" a test suite that contains at least one test case capable of revealing the fault that it is referring to. Table 4 presents more details about the results of the previously-described procedure. More precisely, column "#Gen. Tests" presents the number of test cases in the triggering test suites per project and column "#Faults Found", the number of the corresponding discovered faults. Note that we did not include any results for the `Mockito` project because most of the resulting test cases were problematic. In total, our real fault set consists of 125 real faults from 5 open source projects and our triggering test suites are composed of 23,419 test cases.

4.3 Manual Assessment

To complement our analysis we manually applied the tools to parts of several real-world projects. Since manual analysis requires considerable resources, analysing a complete project is infeasible. Thus, we picked and analysed 12 methods from 6 Java test subjects for 3 independent times, once per studied tool. Thus, in total, we manually analysed 36 methods and 5,831 mutants which constitutes one of the largest studies in the literature of mutation testing, e.g., Yao et al. [44] consider 4,181 mutants, Baker and Habli [4] consider 2,555. Further, the present study is the only one in the literature to consider manually analysed mutants when comparing the effectiveness of different mutation testing tools (see also Section 7). The rest of this section discusses the test subjects, tool configuration and the manual procedure we followed in order to perform mutation testing.

We selected 12 methods to perform our experiment; 10 of them were randomly picked from 4 real-world projects (Commons-Math, Commons-Lang, Pamvotis and XStream) and another 2 (Triangle and Bisect) from the mutation testing literature [1]. Details regarding the selected subjects are presented in Table 5. The table presents the name of the test subjects, their source code lines as reported by the `cloc` tool, the names of the studied methods, the number of generated and disjoint mutants per tool and the number of the resulting mutants of the reference mutant set.

4.3.1 Selection of Mutation Testing Tools

We used the three mutation testing tools for Java that were mentioned in the papers published during 2014 and 2015 in the ISSTA, (ESEC)FSE and ICSE conferences. Thus, we used version 3 of MUJAVA, version 1.1.8 of MAJOR and the research version of PIT, PIT_{RV} [18], and applied all the provided mutation operators. In the case of MUJAVA, only the method-level operators were employed, since the other tools do not provide object-oriented operators.

4.3.2 Manual Analysis Procedure

The primary objective of this experiment is to accurately measure the effectiveness of the studied tools. Thus, we performed complete manual analysis (by designing tests that kill all the killable mutants and manually identified the equivalent mutants) of the mutants produced by the selected tools. Performing this task is labour-intensive and error-prone. Thus, to avoid bias from the use of different tools we asked different users to perform mutation testing on our subjects. To find this number of qualified human subjects we turned to third- and fourth-year Computer Science students of the Department of Informatics at the Athens University of Economics and Business and adopted a two-phase manual analysis process:

- The selected methods were given to students attending the "Software Validation, Verification and Maintenance" course (Spring 2015 and Fall 2015), taught by Prof. Malevris, in order to analyse the mutants of the studied tools, as part of their coursework. The participating students were selected based on their overall performance and their grades at the programming courses. Additionally, they all attended an introductory lecture on mutation testing and appropriate tutorials before the beginning of their coursework. To facilitate the smooth completion of their projects and the correct application of mutation, the students were closely supervised, with regular team meetings throughout the semester.

Table 5: Test Subject Details: Generated Mutants, Disjoint Mutants and Reference Mutant Set

			# Generated Mutants			# Disjoint Mutants			# Mutants
Test Subject	LoC	Method	MAJOR	PIT_{RV}	MUJAVA	MAJOR	PIT_{RV}	MUJAVA	Reference Mutant Set
Commons-Math	16,489	gcd	133	392	237	7	12	9	10
		orthogonal	120	392	155	11	11	11	11
Commons-Lang	17,294	toMap	23	115	32	6	6	5	10
		subarray	25	95	64	6	6	3	6
		lastIndexOf	29	139	81	11	18	13	17
		capitalize	37	132	69	5	4	4	5
		wrap	71	328	198	12	12	16	13
Pamvotis	5,505	addNode	89	447	318	16	23	29	37
		removeNode	18	109	55	7	6	6	6
Triangle	47	classify	139	486	354	31	26	31	55
XStream	15,048	decodeName	73	315	156	8	9	8	10
Bisect	37	sqrt	51	219	135	7	6	7	6
Total	**54,420**	-	**808**	**3,169**	**1,854**	**127**	**139**	**142**	**186**

- The designed test cases and detected equivalent mutants were manually analysed and carefully verified by at least one of the authors.

To generate the mutation adequate test suites, the students were first instructed to generate branch adequate test suites and then to randomly pick a live mutant and attempt to kill it based on the RIP Model [1]. Although the detection of killable mutants is an objective process, i.e., the produced test case either kills the corresponding mutant or not, the detection of equivalent ones is a subjective one. To deal with this issue, all students were familiarised with the RIP Model [1] and the sub-categories of equivalent mutants described by Yao et al. [44]. Also, all detected equivalent mutants were independently verified.

It should be noted that for the PIT_{RV} mutants, one of the authors of this paper extended our previous manual analysis of the PIT mutants [26] by designing new test cases that kill (or identify as equivalent) the PIT_{RV} mutants that remained alive after the application of PIT's mutation adequate test suites. To support replication and wider scrutiny of our manual analysis, we made all its results publicly available [27].

4.4 Methodology

To answer the stated RQs, we applied mutation testing by independently using each one of the selected tools. As the empirical study involves two parts, an experiment on open source projects with real faults and a manual analysis on sampled functions, we simulate the mutation testing process in two ways.

For the large-scale experiment (using open source projects with real faults), we constructed a test pool composed of multiple test suites that were generated by EvoSuite [13] and Randoop [35]. We then identified the triggering test suites, i.e., the test suites that contain at least one test case that reveals the studied faults and discarded all the Defects4J faults for which no triggering test suite was generated.

Related to the sampled functions, we manually generated three mutation adequate test sets (one for every analysed tool) per studied subject. We then minimised these test sets by checking, for each contained test case, whether its removal would result in a decreased mutation score [1]. In case the removal of a test does not result in any decrease of mutation score, it is redundant (according to the used mutation testing tool) and has to be removed. As redundant tests do not satisfy any of the criterion requirements, they can artificially result in overestimating the strengths of the test suites [1]. We used the resulting tests and computed the set of disjoint mutants produced by each one of the tools. We then constructed the reference mutant set by identifying the disjoint mutants (using all the produced tests) of the mutant set composed of all mutants of all the studied tools. To compute the disjoint mutant set we need a matrix that records all test cases that kill a mutant. The construction of such a matrix is available in the case of PIT_{RV}. For MUJAVA, we extended the corresponding script to handle certain cases that it failed to work, e.g., a case where a class that belonged to a package was given as input. Finally, in the case of MAJOR, we utilised the scripts accompanying Defects4J to produce this matrix. The disjoint set of mutants was computed using the "Subsuming Mutants Identification" process that is described in the study of Papadakis et al. [41]. Here, we use the term "disjoint" mutants, instead of "subsuming" ones since this was the original term that was used by the first study that introduced them and suggested their use as a metric that can accurately measure test effectiveness [24].

In RQ1 we are interested in the fault revelation ability of mutation-based test suites. Thus, we want to see whether mutation-driven test cases can reveal our faults. We

measure the fault revelation ability of the studied mutation testing tools by evaluating the fault revelation of the test cases that kill their mutants. More precisely, we consider a fault as revealed when there is at least one mutant which is killed only by triggering test cases for that particular fault. Thus, based on our generated test suites, if this mutant is killed, a test case that reveals the respective fault is bound to be generated. To answer the research question, we compare the number of revealed faults per tool and project and rank them accordingly, thus, answering RQ2.

To answer RQ3, we used the selected methods and the manually generated test suites. For each selected tool we used its mutation adequate test suite and calculated the mutation score and disjoint mutation score that it achieves when it is evaluated with the mutants produced by the other tools. This process can be viewed as an objective comparison between the tools, i.e., a comparison that evaluates how the tests designed for one tool perform when evaluated in terms of the other tool. In case the tests of one tool can kill all the mutants produced by the other tool, then this tool subsumes the other. Otherwise, the two tools are incomparable.

To answer RQ4, we used the tests that were specifically designed for each one of the studied tools (from the manually analysed subjects) and measured the score they achieve when evaluated against the reference mutant set. This score provides the common ground to compare the tools and rank them with respect to their effectiveness and identify the most effective tool.

To answer RQ5, for each tool we manually analysed the mutants that were not killed by the tests of the other tools with the intention of identifying inadequacies in the tools' mutant sets. We then gathered all these instances and identified how we could complement each one of the tools in order to improve its effectiveness and reach the level of the reference mutant set. Finally, to answer RQ6, we measured and report the number of tests and equivalent mutants that we found.

5 Empirical Findings

This section presents the empirical findings of our study per posed research question.

5.1 RQ1: Tools' Real Fault Revelation Ability

This question investigates whether one of the studied tools subsumes the others in terms of fault revelation ability. Table 6 records our results. The first column of the table presents the ids of the 125 real faults of our benchmark set (per project); the remainder of the table is divided into three parts (columns "MAJOR", "PIT_{RV}", "MUJAVA") and each of these parts is divided into two sub-columns. The first sub-column presents whether the corresponding tool can run on the corresponding fixed program version (sub-column "Runs?") and whether it manages to reveal the corresponding fault (sub-column "Reveals?"), i.e., whether there is at least one generated mutant which is killed only by test cases that also reveal the respective fault.

By examining the table, it can be seen that PIT_{RV} and MAJOR were run successfully on all studied program versions whereas MUJAVA only on 66 of them. This fact indicates that PIT_{RV} and MAJOR have higher chances of being used in practice than MUJAVA due to their higher applicability rate. Since MUJAVA does not work properly on all the projects, we compare the tools in a pairwise manner by considering only the projects that both compared tools operate. Based on the table's results, it becomes

evident that none of the tools subsumes the others; there are faults that are only revealed by killing test cases of mutants generated by only one tool and not the others and, as a consequence, none of the tools alone can reveal all the faults studied. Specifically, Major reveals 2 unique faults (`Time-27` and `Math-55`) compared to PIT_{RV} and 29 faults compared to muJava. muJava reveals one unique fault (`Math-27`) compared to Major; when compared to PIT_{RV}, all muJava-revealed faults are also revealed by PIT_{RV}. Finally, PIT_{RV} reveals 9 unique faults (`Lang-56`, `Math-6`, `Math-22`, `Math-27`, `Math-89`, `Math-105`, `Closure-27`, `Closure-49`, `Closure-52`) compared to Major and 31 unique faults compared to muJava. One interesting finding is that 2 faults (`Math-75`, `Math-90`) are not revealed by any tool. Overall, PIT_{RV} managed to reveal 121 real faults out of 125 for which it run successfully, Major revealed 114 out of 125 and muJava 34 out of 66.

5.2 RQ2: Better Performing Tool

Overall, we found that PIT_{RV} is the most effective tool at revealing faults. PIT_{RV} achieves a higher fault revelation (w.r.t. the studied fault set) than Major in 9 cases, equal in 106 and lower in 2 cases. In summary, these results indicate that PIT_{RV} has higher fault revelation ability than Major by 6%. Therefore, we conclude that according to our fault sample PIT_{RV} is the most effective tool.

Table 6: Major's, PIT_{RV} and muJava fault revelation on real faults studied

Project-BugID	Major		PIT_{RV}		muJava	
	Runs?	Reveals?	Runs?	Reveals?	Runs?	Reveals?
Chart-1	✓	✓	✓	✓		
Chart-2	✓	✓	✓	✓	✓	✓
Chart-4	✓	✓	✓	✓		
Chart-5	✓	✓	✓	✓	✓	✓
Chart-6	✓	✓	✓	✓	✓	✓
Chart-8	✓	✓	✓	✓	✓	
Chart-11	✓	✓	✓	✓	✓	
Chart-14	✓	✓	✓	✓		
Chart-15	✓	✓	✓	✓		
Chart-16	✓	✓	✓	✓	✓	
Chart-17	✓	✓	✓	✓		
Chart-18	✓	✓	✓	✓	✓	✓
Chart-19	✓	✓	✓	✓		
Chart-22	✓	✓	✓	✓	✓	
Chart-23	✓	✓	✓	✓		
Chart-24	✓	✓	✓	✓	✓	✓
Chart-26	✓	✓	✓	✓		
Time-1	✓	✓	✓	✓	✓	
Time-2	✓	✓	✓	✓	✓	
Time-4	✓	✓	✓	✓		
Time-5	✓	✓	✓	✓	✓	
Time-6	✓	✓	✓	✓		
Time-8	✓	✓	✓	✓	✓	
Time-9	✓	✓	✓	✓	✓	✓

Table 6: MAJOR's, PIT_{RV} and MUJAVA fault revelation on real faults studied

Project-BugID	MAJOR		PIT_{RV}		MUJAVA	
	Runs?	Reveals?	Runs?	Reveals?	Runs?	Reveals?
Time-11	✓	✓	✓	✓		
Time-12	✓	✓	✓	✓	✓	
Time-13	✓	✓	✓	✓		
Time-15	✓	✓	✓	✓	✓	
Time-17	✓	✓	✓	✓	✓	
Time-27	✓	✓	✓			
Lang-5	✓	✓	✓	✓	✓	✓
Lang-7	✓	✓	✓	✓	✓	
Lang-9	✓	✓	✓	✓		
Lang-10	✓	✓	✓	✓		
Lang-11	✓	✓	✓	✓	✓	✓
Lang-12	✓	✓	✓	✓	✓	✓
Lang-16	✓	✓	✓	✓	✓	
Lang-19	✓	✓	✓	✓	✓	✓
Lang-23	✓	✓	✓	✓		
Lang-24	✓	✓	✓	✓	✓	
Lang-27	✓	✓	✓	✓	✓	✓
Lang-33	✓	✓	✓	✓		
Lang-35	✓	✓	✓	✓		
Lang-36	✓	✓	✓	✓	✓	✓
Lang-37	✓	✓	✓	✓		
Lang-39	✓	✓	✓	✓		
Lang-41	✓	✓	✓	✓		
Lang-43	✓	✓	✓	✓		
Lang-44	✓	✓	✓	✓	✓	
Lang-45	✓	✓	✓	✓	✓	✓
Lang-46	✓	✓	✓	✓	✓	
Lang-47	✓	✓	✓	✓	✓	✓
Lang-49	✓	✓	✓	✓	✓	✓
Lang-52	✓	✓	✓	✓	✓	
Lang-54	✓	✓	✓	✓	✓	✓
Lang-56	✓		✓	✓		
Lang-58	✓	✓	✓	✓	✓	
Lang-59	✓	✓	✓	✓	✓	✓
Lang-60	✓	✓	✓	✓	✓	✓
Lang-61	✓	✓	✓	✓	✓	✓
Math-1	✓	✓	✓	✓	✓	✓
Math-3	✓	✓	✓	✓		
Math-4	✓	✓	✓	✓	✓	
Math-5	✓	✓	✓	✓		
Math-6	✓		✓	✓	✓	
Math-8	✓	✓	✓	✓		
Math-10	✓	✓	✓	✓		
Math-11	✓	✓	✓	✓	✓	✓
Math-14	✓	✓	✓	✓	✓	✓

Table 6: MAJOR's, PIT_{RV} and MUJAVA fault revelation on real faults studied

Project-BugID	MAJOR		PIT_{RV}		MUJAVA	
	Runs?	Reveals?	Runs?	Reveals?	Runs?	Reveals?
Math-22	✓		✓	✓	✓	
Math-23	✓	✓	✓	✓	✓	✓
Math-24	✓	✓	✓	✓	✓	✓
Math-25	✓	✓	✓	✓	✓	
Math-27	✓		✓	✓	✓	✓
Math-29	✓	✓	✓	✓		
Math-31	✓	✓	✓	✓	✓	
Math-32	✓	✓	✓	✓		
Math-35	✓	✓	✓	✓	✓	✓
Math-36	✓	✓	✓	✓		
Math-37	✓	✓	✓	✓		
Math-42	✓	✓	✓	✓		
Math-45	✓	✓	✓	✓		
Math-46	✓	✓	✓	✓		
Math-47	✓	✓	✓	✓		
Math-49	✓	✓	✓	✓		
Math-51	✓	✓	✓	✓		
Math-55	✓	✓	✓		✓	
Math-56	✓	✓	✓	✓	✓	✓
Math-59	✓	✓	✓	✓		
Math-60	✓	✓	✓	✓	✓	✓
Math-61	✓	✓	✓	✓	✓	
Math-63	✓	✓	✓	✓		
Math-66	✓	✓	✓	✓	✓	✓
Math-70	✓	✓	✓	✓		
Math-73	✓	✓	✓	✓		
Math-75	✓		✓			
Math-77	✓	✓	✓	✓	✓	
Math-85	✓	✓	✓	✓	✓	✓
Math-86	✓	✓	✓	✓	✓	
Math-87	✓	✓	✓	✓		
Math-89	✓		✓	✓		
Math-90	✓		✓			
Math-92	✓	✓	✓	✓		
Math-93	✓	✓	✓	✓		
Math-95	✓	✓	✓	✓	✓	✓
Math-97	✓	✓	✓	✓		
Math-98	✓	✓	✓	✓	✓	✓
Math-99	✓	✓	✓	✓		
Math-101	✓	✓	✓	✓	✓	✓
Math-102	✓	✓	✓	✓	✓	
Math-103	✓	✓	✓	✓	✓	
Math-104	✓	✓	✓	✓	✓	
Math-105	✓		✓	✓	✓	
Closure-7	✓	✓	✓	✓		

Table 6: MAJOR's, PIT_{RV} and MUJAVA fault revelation on real faults studied

Project-BugID	MAJOR		PIT_{RV}		MUJAVA	
	Runs?	Reveals?	Runs?	Reveals?	Runs?	Reveals?
Closure-27	✓		✓	✓		
Closure-33	✓	✓	✓	✓		
Closure-42	✓	✓	✓	✓		
Closure-49	✓		✓	✓		
Closure-52	✓		✓	✓		
Closure-54	✓	✓	✓	✓		
Closure-56	✓	✓	✓	✓	✓	✓
Closure-73	✓	✓	✓	✓		
Closure-82	✓	✓	✓	✓		
Closure-103	✓	✓	✓	✓		
Closure-106	✓	✓	✓	✓	✓	
Total	**125**	**114**	**125**	**121**	**66**	**34**

5.3 RQ3: Tool's Cross-evaluation

This question investigates whether one of the studied tools subsumes the others in terms of mutant killability. Table 7 presents the respective results. The table is divided into three parts (columns "MAJOR", "PIT_{RV}", "MUJAVA") and each of these parts is divided into two columns that correspond to the mutation adequate test sets of the remaining tools. Finally, each of these columns is split into two sub-columns that depict the mutation scores achieved by the corresponding test suites when considering all generated mutants (column "All") and the disjoint ones (column "Dis.").

By examining Table 7, it becomes evident that none of the tools subsumes the others; all generated test suites face effectiveness losses when evaluated against the mutants of the other tools. Specifically, PIT_{RV}'s mutation adequate test suites perform the best, with an effectiveness of approximately 100% with respect to MUJAVA and 98% with respect to MAJOR when all mutants are considered; for the disjoint ones, this score drops to approximately 90% for MAJOR and 96% for MUJAVA. The next better performing tool is MUJAVA, whose mutation adequate test suites achieve an effectiveness of 96% and approximately 99% for MAJOR and PIT_{RV}, when all mutants are considered and 91% and approximately 82% for the disjoint ones, respectively. Finally, MAJOR comes last, with an effectiveness of approximately 97% for PIT_{RV} and MUJAVA for all generated mutants; for the disjoint ones, its effectiveness drops to 72% and 85%, respectively.

5.4 RQ4: Comparison with Reference Mutation Tool

This question investigates how the tools' mutation adequate test suites fare against a reference mutation testing tool, simulated by the disjoint mutants of the union of all mutants of the studied tools. Figure 1 depicts the obtained findings. The figure presents the percentage of the mutants that can be killed by the corresponding mutation adequate test suites per method, along with the average score for all methods. Although, the performance of the tools varies depending on the considered method, it can be seen that, on average, PIT_{RV} realises an 91% effectiveness score, followed by MUJAVA and MAJOR with 85% and 80%, respectively. An interesting observation from these results

Table 7: Tools' Cross-Evaluation Results

	MAJOR				PIT_{RV}				MUJAVA			
	PIT_{RV}-TS		MUJAVA-TS		MAJOR-TS		MUJAVA-TS		MAJOR-TS		PIT_{RV}-TS	
Method	All	Dis.	All.	Dis.	All	Dis.	All.	Dis.	All	Dis.	All	Dis.
gcd	97.4%	87.5%	97.4%	71.4%	99.4%	58.3%	98.8%	58.3%	99.5%	88.9%	100.0%	100.0%
orthogonal	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	99.5%	90.9%	100.0%	100.0%	100.0%	100.0%
toMap	100.0%	100.0%	77.8%	83.3%	97.9%	66.6%	92.8%	83.3%	100.0%	100.0%	100.0%	100.0%
subarray	90.0%	66.6%	85.0%	50.0%	100.0%	100.0%	97.6%	83.3%	100.0%	100.0%	100.0%	100.0%
lastIndexOf	100.0%	100.0%	92.6%	91.0%	100.0%	100.0%	97.7%	83.3%	100.0%	100.0%	100.0%	100.0%
capitalize	93.5%	60.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.2%	75.0%
wrap	100.0%	100.0%	100.0%	100.0%	99.0%	75.0%	99.3%	83.3%	98.9%	93.8%	100.0%	100.0%
addNode	100.0%	100.0%	100.0%	100.0%	81.8%	21.7%	99.5%	95.7%	76.5%	37.9%	100.0%	100.0%
removeNode	100.0%	100.0%	100.0%	100.0%	93.7%	33.3%	100.0%	100.0%	91.7%	50.0%	100.0%	100.0%
classify	97.0%	87.0%	100.0%	100.0%	97.1%	88.5%	100.0%	100.0%	98.1%	90.3%	99.1%	90.3%
decodeName	95.9%	80.0%	100.0%	100.0%	95.0%	55.5%	98.4%	55.5%	95.3%	62.5%	99.2%	87.5%
sqrt	100.0%	100.0%	100.0%	100.0%	99.0%	66.7%	99.5%	50.0%	100.0%	100.0%	100.0%	100.0%
Average	**97.8%**	**90.1%**	**96.1%**	**91.3%**	**96.9%**	**72.1%**	**98.6%**	**82.0%**	**96.7%**	**85.3%**	**99.7%**	**96.1%**

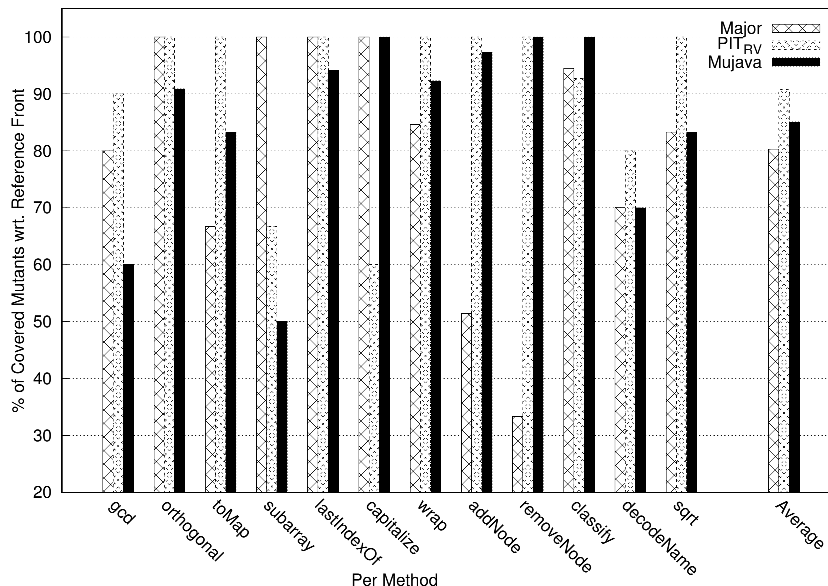

Figure 1: Comparison of Mutation Adequate Test Suites against Reference Mutation Set

is that all tools have important inadequacies that range from 0-66%. On average, the differences are 20%, 15% and 9% for MAJOR, MUJAVA and PIT_{RV}.

Overall we found that PIT_{RV} is the top ranked tool, followed by MUJAVA and MAJOR. PIT_{RV} achieves a higher mutation score (w.r.t. the reference mutant set) than MUJAVA in 9 cases, equal in 1 and lower in 2. Compared to MAJOR, PIT_{RV} performs better in 7 cases, equal in 2 and lower in 3. Therefore, we conclude that according to our sample PIT_{RV} is the most effective tool.

5.5 RQ5: Tools' Weaknesses and Recommendations

This question is concerned with ways of improving the mutation testing practice of the studied tools. To this end, Figure 2 presents the mutants per tool (divided into mutation operators) that remained alive after the application of the mutation adequate test suites of the other tools. The figure is divided into six parts, each one illustrating the live mutants of a corresponding tool with respect to the mutation adequate test suite of another tool.

5.5.1 Recommendations: PIT_{RV}

As can be seen from Figure 2, PIT_{RV}'s mutation adequate test suites fail to kill mutants generated by the COR operators of MUJAVA and MAJOR. This operator, although implemented differently in the two tools, affects compound conditional expressions. Unfortunately PIT_{RV} lacks support for such an operator, primarily because the tool manipulates the bytecode and such expressions are not present in bytecode. Thus, the mutation practice of PIT_{RV} can be improved by finding a way to simulate COR's

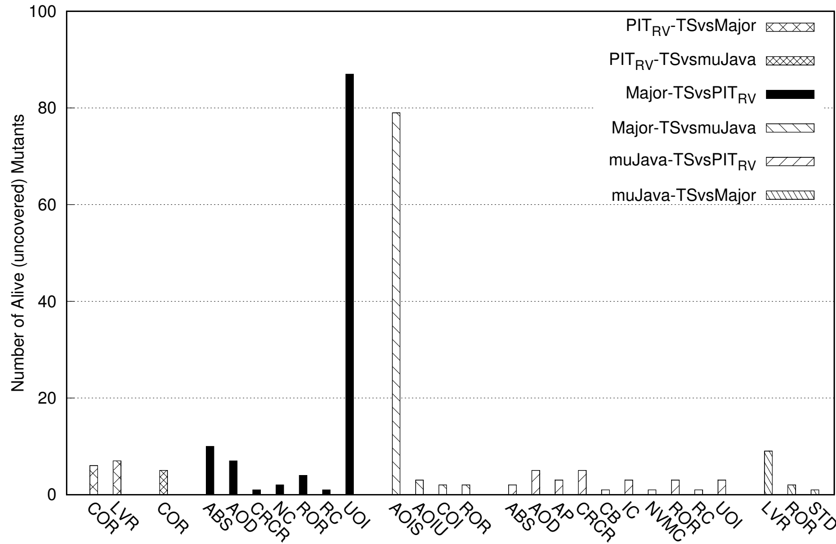

Figure 2: Cross-Evaluation Experiment: Number of Alive Mutants per Mutation Operator, Test Suite and Tool

changes in the bytecode. Additionally, PIT_{RV}'s CRCR operator can be enhanced because it misses certain cases that Major's LVR is applied to due to the aforementioned problem. For instance, at line 410 of the `gcd` method of the `Commons-Math` test subject Major mutates the statement `if (u > 0)` to `if (u > 1)`. This change is not made by PIT_{RV}'s CRCR operator because in the bytecode the zero constant is never pushed into the stack. In order to make PIT_{RV} more effective such cases should be handled accordingly.

5.5.2 Recommendations: Major

By examining Figure 2, it can be seen that Major's tests fail to cover muJava's AOIS mutants and the analogous mutants of PIT_{RV} (generated by the UOI operator). Thus, the tool's practice can be enhanced by implementing the changes imposed by these operators. Additionally, Major will benefit by adding an operator that negates arithmetic variables, analogous to muJava's AOIU and PIT_{RV}'s ABS and implementing PIT_{RV}'s AOD operator which manipulates arithmetic expressions by deleting the corresponding operands one at a time. Finally, we observed that the tests of Major failed to kill some mutants generated by muJava's and PIT_{RV}'s ROR operators. Recall that Major implements a specialised version of ROR that induces only a subset of its changes. The live mutants indicate a weakness in this specialised set that can lead to test effectiveness loss. An example of such a weakness manifested at line 161 of the `decodeName` method where muJava's ROR changed the sub-expression `c == escapeReplacementFirstChar` to `c > escapeReplacementFirstChar`. It is noted that the issue with the ROR mutants is an implementation choice of Major as detailed in Table 3 and discussed in the study of Lindström and Márki [30].

Table 8: Tools' Application Cost: Number of Equivalent Mutants and Required Tests

	MAJOR		PIT_{RV}		MUJAVA	
Method	#Eq.	#Tests	#Eq.	#Tests	#Eq.	#Tests
gcd	17	6	70	7	23	7
orthogonal	3	8	22	8	5	9
toMap	5	7	18	6	7	5
subarray	5	6	12	5	8	6
lastIndexOf	2	8	10	8	4	12
capitalize	6	5	22	4	14	9
wrap	8	10	36	7	19	7
addNode	11	8	57	22	33	34
removeNode	2	5	14	5	7	6
classify	7	25	42	22	38	27
decodeName	24	5	57	7	28	10
sqrt	4	4	22	4	17	6
Total	**94**	**97**	**382**	**105**	**203**	**138**

5.5.3 Recommendations: MUJAVA

As can be seen from Figure 2, MUJAVA's weaknesses centre around mutation operators that affect literal values, namely MAJOR's LVR and PIT_{RV}'s IC and CRCR operators. Thus, MUJAVA will benefit by implementing such operators. Furthermore, we found MUJAVA's implementation of the ROR mutation operator inconsistent; for example, at line 25 of the `wrap` method, the tool did not replace the original statement, `if (newLineStr == null)`, with `if (true)`, as it was supposed to, leading to inadequacies in the resulting test suites. Similar examples are present at line 248 of `toMap` and 1282 of `lastIndexOf`. These implementation defects lower the test effectiveness of the resulting mutation adequate test suites and addressing them will improve the tool's test quality. Finally, the tool will benefit from implementing PIT_{RV}'s AOD operator, as it was the case with MAJOR.

5.6 RQ6: Tools' Application Cost

The answer to this question provides insights on the relative cost of the studied tools' application in terms of the number of equivalent mutants that have to be manually analysed and the number of test cases required. Table 8 presents the corresponding findings. The table is divided into three parts, each one for a studied tool, and presents the examined cost metrics in the sub-columns of these parts ("#Eq." and "#Tests").

We can observe that 12% of MAJOR's and PIT_{RV}'s mutants are equivalent and 11% of MUJAVA's ones. Although the tools generate the same percentage of equivalent mutants with respect to the total generated ones, MAJOR generated the least number of equivalent mutants, PIT_{RV} the greatest and MUJAVA was placed in the middle. Thus, MAJOR requires the least amount of human effort in identifying the generated equivalent mutants, whereas PIT_{RV} the greatest. Regarding the number of killing test cases the tools require, MAJOR requires 97 test cases, PIT_{RV} 105 and MUJAVA 138. Thus, MAJOR and PIT_{RV} require the least amount of effort in generating mutation adequate

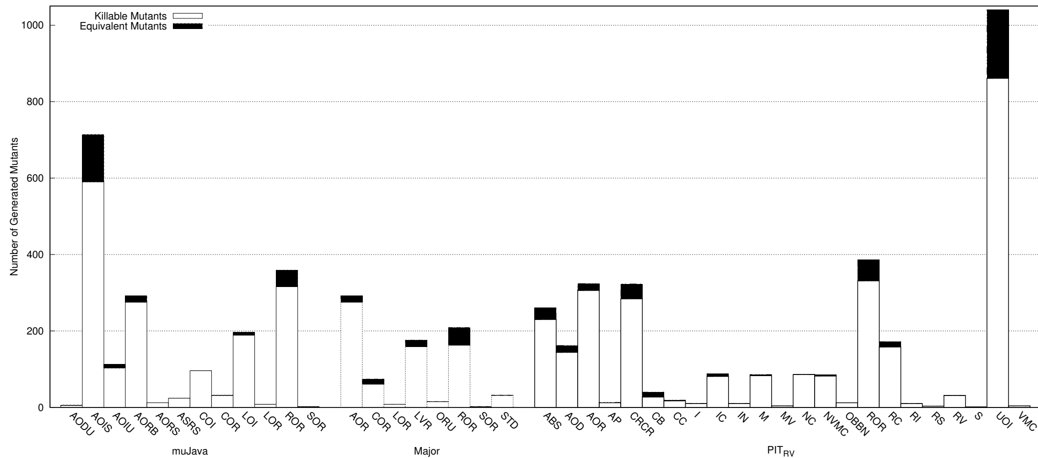

Figure 3: Contribution of Mutation Operators to Generated and Equivalent Mutants per Tool

test suites and MUJAVA the greatest. It is interesting to notice that althought PIT_{RV} generates a high number of mutants, it requires a considerably low number of mutation adequate test cases indicating that the current version of the tool faces high mutant redundancy, which in turn suggests that the efficiency of the tool can be greatly improved. Future version of PIT_{RV} should take this finding into account.

The previously-described results indicate that MAJOR is the most efficient tool and PIT_{RV} the most expensive one, with MUJAVA standing in the middle. Considering that PIT_{RV} was found the most effective tool both in terms of fault revelation and mutant killability, it is no surprise that it is the least efficient one. Analogously, MAJOR requires less effort, a fact justified by its lower performance.

To better understand the nature of the generated equivalent mutants, Figure 3 illustrates the contribution of each mutation operator to the generated killable and equivalent mutants per tool. In the case of MUJAVA, AOIS and ROR generate most of the tool's equivalent mutants. For MAJOR, ROR generates most of the equivalent mutants, followed by LVR, AOR and COR. In the case of PIT_{RV}, UOI generates the most equivalent mutants, followed by ROR, CRCR and ABS.

6 Threats to Validity

As every empirical study, this one faces several threats to its validity. Here we discuss these threats along with the actions we took to mitigate them.

External Validity. External validity refers to the ability of a study's results to generalise. Such threats are likely due to the programs, faults or test suites that we use, as they might not be representative of actual cases. To mitigate the underlying threats, we utilised a publicly available benchmark (Defects4J), which was built independently from our work and consists of 6 real world open source projects and real faults. We also used 6 additional test subjects and manually analysed 12 methods whose application domain varies. Although we cannot claim that our results are generalisable, our findings indicate specific inadequacies in the mutants produced by the studied tools.

These involve incorrect implementation or not supported mutation operators, evidence that is unlikely to be case-specific.

Internal Validity. Internal validity includes potential threats to the conclusions we draw. Our conclusions are based on a benchmark fault set, automatically generated test suites and manual analysis, i.e., on the identified equivalent mutants and mutation adequate test suites. Thus, the use of the tools might have introduced errors in our measurements. For instance, it could be the case where test oracles generated by the tools are weak and cannot capture mutants or the studied faults. We also performed our experiments on the clean (fixed) program versions, which may differ from that of the buggy version [7], because the existing Java tools only operate on passing test suites. Moreover this is common practice in this type of experiments. To mitigate these threats, we carefully checked our scripts, verified some of our results, performed sanity checks and generated multiple test suites using two different tools. However, we consider these threats of no substantial importance since our results are consistent in both the manual and automated scenarios we analyse.

Other threats are due to the manual analysis we performed. To control this fact, we ensured that this analysis was performed by different persons to avoid any bias in the results and that all results produced by students were independently checked for correctness by at least one of the authors. Another potential threat is due to the fact that we did not control the test suite size. However, our study focuses on investigating the effectiveness of the studied tools when used as a means to generate strong tests [36]. To cater for wider scrutiny, we made publicly available all the data of this study [27].

Construct Validity. Construct validity pertains to the appropriateness of the measures utilised in our experiments. For the effectiveness comparison, we used fault detection (using real faults), mutation score and disjoint mutation score measurements. These are well-established measures in mutation testing literature [41]. Another threat originates from evaluating the tools' effectiveness based on the reference fault and mutant set that are revealed by the manually generated or automatically generated test suites. We deemed this particular measure appropriate because it constitutes a metric that combines the overall performance of the tools and enables their ranking. Finally, the number of equivalent mutants and generated tests might not reflect the actual cost of applying mutation. We adopted these metrics because they involve manual analysis which is a dominant cost factor when testing.

7 Related Work

Mutation testing is a well-studied technique with a rich history of publications, as recorded in the surveys of Offutt [34] and Yia and Harman [19].

The original suggestion of mutation was a method to help programmers generate effective test cases [10]. Since then, researchers has used it to support various other software engineering tasks [34]. In particular, mutation analysis has been employed in: test generation [36], test oracle selection and assessment [14], debugging [38], test assessment [41] and in regression testing [46]. It has also been applied to artefacts other than source code, such as models [12] and software product lines configurations [17].

The main problems of mutation testing are the large number of mutants and the so-called equivalent mutant problem [40, 22]. To tackle these problems several mutant selection strategies were suggested. Mutant sampling is perhaps the simplest and most effective way of doing so. Depending on the sampling ratio it provides several trade-

offs between reduced number of mutants and effectiveness loss (fault detection) [37], e.g., sampling ratios of 10% to 60% have a loss on fault detection from 26% to 6%. Selective mutation [33] is another form of mutant reduction that only applies specific types of mutants. However, recent research has shown that there are not significant differences between selective mutation and random sampling [45, 28]. To deal with the equivalent mutant problem researchers has adopted compiler optimisations [40], constraint based techniques [32] and verification techniques [5]. However, despite the efforts this problem remains open especially for the case of Java. This is the main reason why we manually identified and report on the equivalent mutants produced by the tools.

Another problem related to mutation testing regards the generation of redundant mutants. These mutants do not contribute to the testing process, while at the same time they introduce noise to the mutation score measurement. Papadakis et al. [41] experimented and demonstrated that there is a good chance of drawing wrong conclusions (approximately 60%) for arbitrary experiments when measuring test thoroughness using all mutants rather than with only the disjoint/subsuming ones. Unfortunately, the above-mentioned result suggests that it is likely to conclude that one testing method is superior to another one but in fact it is not. The problem of redundant mutants has been initially identified by Kintis et al. [24] with the notion of disjoint mutants. Later Ammann et al. [2] formalised the concept based on the notion of dynamic subsumption. Unfortunately, these techniques focus on the undesirable effects of redundant mutants and not their identification. Perhaps the only available technique that is capable of identifying such mutants is "Trivial Compiler Equivalence" (TCE) [40]. TCE is based on compiler optimisations and identifies duplicated mutants (mutants that are mutually equivalent but differ from the original program). According to the study of Papadakis et al. [40], 21% of the mutants are duplicated and can be easily removed based on compiler optimisations. All these studies identified the problems caused by trivial/redundant mutants but none of them studied the particular weaknesses of modern mutation testing tools as we do here. Additionally, to deal with trivial/redundant mutants we used: (1) the mutation score, (2) the disjoint mutation score, and (3) the fault detection as effectiveness measures.

Manual analysis has been used extensively in the mutation testing literature. Yao et al. [44] analysed 4,181 mutants to provide insights into the nature of equivalent and stubborn mutants. Nan et al. [29] manually analysed 2,919 mutants to compare test cases generated for mutation testing with the ones generated for various control and data flow coverage criteria. Deng et al. [11] analysed 5,807 mutants generated by MUJAVA to investigate the effectiveness of the SDL mutation operator. Papadakis et al. [39] used manual analysis to study mutant classification strategies and found that such techniques are helpful only to partially improve test suites (of low quality). Older studies on mutant selection involved manual analysis to identify equivalent mutants and generate adequate test suites [33].

Previous work on the differences of mutation testing frameworks for Java is due to Delahaye and Du Bousquet [9]. Delahaye and Du Bousquet compare several tools based on various criteria, such as the supported mutation operators, implementation differences and ease of usage. The study concluded that different mutation testing tools are appropriate to different scenarios. A similar study was performed by Rani et al. [42]. This study compared several Java mutation testing tools based on a set of manually generated test cases. The authors concluded that PIT generated the smallest number of mutants, most of which were killed by the employed test suite (only 2% survived), whereas, MUJAVA generated the largest number of mutants, 30% of which

survived.

Gopinath et al. [16] investigated the effectiveness of mutation testing tools by using various metrics, e.g., comparing the mutation score (obtained by the test subjects' accompanying test suites) and number of disjoint/minimal mutants that they produce. They found that the examined tools exhibit considerable variation of their performance and that no single tool is consistently better than the others.

The main differences between our study and the aforementioned ones are that we compare the tools based on their real fault revelation ability and cross-evaluated their effectiveness based on the results of complete manual analysis. The manual analysis constitutes a mandatory requirement (see Section 3) for performing a reliable effectiveness comparison between the tools. This twofold comparison is one of the strengths of the present paper as it is the first one in the literature to compare mutation testing tools in such a way. Further, we identified specific limitations of the tools and provided actionable recommendations on how each of the tools can be improved. Lastly, we analysed and reported the number and characteristics of equivalent mutants produced by each tool.

8 Conclusions

Mutation testing tools are widely used as a means to support research. This practice intensifies the need for reliable, effective and robust mutation testing tools. Today most of the tools are mature and robust, hence the emerging question regards their effectiveness, which is currently unknown.

In this paper, we reported results from a controlled study that involved manual analysis (on a sample of program functions selected from open source programs) and simulation experiments on open source projects with real faults. Our results showed that one tool, PIT_{RV}, the research version of PIT, performs significantly better than the other studied tools, namely MUJAVA and MAJOR. At the same time our results showed that the studied tools are generally incomparable as none of them always subsumes the others. Nevertheless, we identified the few deficiencies of PIT_{RV} and made actionable recommendations on how to strengthen it and improve its practice.

Overall, our results demonstrate that PIT_{RV} is the most prominent choice of mutation testing tool for Java, as it successfully revealed 97% of the real faults we studied and performed best in our manual analysis experiment.

References

[1] Ammann P, Offutt J (2008) Introduction to Software Testing, 1st edn. Cambridge University Press, New York, NY, USA

[2] Ammann P, Delamaro ME, Offutt J (2014) Establishing theoretical minimal sets of mutants. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA, pp 21–30, DOI 10.1109/ICST.2014.13

[3] Andrews J, Briand L, Labiche Y, Namin A (2006) Using mutation analysis for assessing and comparing testing coverage criteria. Software Engineering, IEEE Transactions on 32(8):608–624, DOI 10.1109/TSE.2006.83

[4] Baker R, Habli I (2013) An empirical evaluation of mutation testing for improving the test quality of safety-critical software. Software Engineering, IEEE Transactions on 39(6):787–805, DOI 10.1109/TSE.2012.56

[5] Bardin S, Delahaye M, David R, Kosmatov N, Papadakis M, Traon YL, Marion J (2015) Sound and quasi-complete detection of infeasible test requirements. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015, pp 1–10, DOI 10.1109/ICST.2015.7102607, URL http://dx.doi.org/10.1109/ICST.2015.7102607

[6] Budd TA, Angluin D (1982) Two notions of correctness and their relation to testing. Acta Informatica 18(1):31–45, DOI 10.1007/BF00625279

[7] Chekam TT, Papadakis M, Traon YL, Harman M (2017) Empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: ICSE

[8] Coles H (2010) ”the PIT mutation testing tool”. URL "http://pitest.org/", “Last Accessed June 2016”

[9] Delahaye M, Du Bousquet L (2015) Selecting a software engineering tool: lessons learnt from mutation analysis. Software: Practice and Experience 45(7):875–891, DOI 10.1002/spe.2312

[10] DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: Help for the practicing programmer. IEEE Computer 11(4):34–41, DOI 10.1109/C-M.1978.218136, URL http://dx.doi.org/10.1109/C-M.1978.218136

[11] Deng L, Offutt J, Li N (2013) Empirical evaluation of the statement deletion mutation operator. In: Software Testing, Verification and Validation, IEEE Sixth International Conference on, pp 84–93, DOI 10.1109/ICST.2013.20

[12] Devroey X, Perrouin G, Papadakis M, Legay A, Schobbens P, Heymans P (2016) Featured model-based mutation analysis. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, pp 655–666, DOI 10.1145/2884781.2884821, URL http://doi.acm.org/10.1145/2884781.2884821

[13] Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011, pp 416–419, DOI 10.1145/2025113.2025179, URL http://doi.acm.org/10.1145/2025113.2025179

[14] Fraser G, Zeller A (2012) Mutation-driven generation of unit tests and oracles. IEEE Trans Software Eng 38(2):278–292, DOI 10.1109/TSE.2011.93, URL http://dx.doi.org/10.1109/TSE.2011.93

[15] Geist R, Offutt A, Harris J FC (1992) Estimation and enhancement of real-time software reliability through mutation analysis. Computers, IEEE Transactions on 41(5):550–558, DOI 10.1109/12.142681

[16] Gopinath R, Ahmed I, Alipour MA, Jensen C, Groce A (2016) Does choice of mutation tool matter? Software Quality Journal pp 1–50, DOI 10.1007/s11219-016-9317-7

[17] Henard C, Papadakis M, Traon YL (2014) Mutation-based generation of software product line test configurations. In: Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings, pp 92–106, DOI 10.1007/978-3-319-09940-8_7, URL http://dx.doi.org/10.1007/978-3-319-09940-8_7

[18] Henry Coles and Thomas Laurent and Christopher Henard and Mike Papadakis and Anthony Ventresque (2016) PIT: a practical mutation testing tool for java (demo). In: ISSTA, pp 449–452, DOI 10.1145/2931037.2948707

[19] Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on 37(5):649–678, DOI 10.1109/TSE.2010.62

[20] Just R, Schweiggert F, Kapfhammer GM (2011) MAJOR: an efficient and extensible tool for mutation analysis in a java compiler. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011, pp 612–615, DOI 10.1109/ASE.2011.6100138, URL http://dx.doi.org/10.1109/ASE.2011.6100138

[21] Just R, Jalali D, Ernst MD (2014) Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014, pp 437–440, DOI 10.1145/2610384.2628055, URL http://doi.acm.org/10.1145/2610384.2628055

[22] Kintis M (2016) Effective methods to tackle the equivalent mutant problem when testing software with mutation. PhD thesis, Department of Informatics, Athens University of Economics and Business

[23] Kintis M, Malevris N (2015) MEDIC: A static analysis framework for equivalent mutant identification. Information and Software Technology 68:1 – 17, DOI 10.1016/j.infsof.2015.07.009

[24] Kintis M, Papadakis M, Malevris N (2010) Evaluating mutation testing alternatives: A collateral experiment. In: Proceedings of the 17th Asia-Pacific Software Engineering Conference, pp 300–309, DOI 10.1109/APSEC.2010.42

[25] Kintis M, Papadakis M, Malevris N (2015) Employing second-order mutation for isolating first-order equivalent mutants. Software Testing, Verification and Reliability (STVR) 25(5-7):508–535, DOI 10.1002/stvr.1529

[26] Kintis M, Papadakis M, Papadopoulos A, Valvis E, Malevris N (2016) Analysing and comparing the effectiveness of mutation testing tools: A manual study. In: International Working Conference on Source Code Analysis and Manipulation, pp 147–156

[27] Kintis M, Papadakis M, Papadopoulos A, Valvis E, Malevris N, Traon YL (2017) Supporting site for the paper: How effective mutation testing tools are? an empirical analysis of java mutation testing tools with manual analysis and real faultss. URL http://pages.cs.aueb.gr/~kintism/papers/mttoolscomp

[28] Kurtz B, Ammann P, Offutt J, Delamaro ME, Kurtz M, Gökçe N (2016) Analyzing the validity of selective mutation with dominator mutants. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, pp 571–582, DOI 10.1145/2950290.2950322, URL `http://doi.acm.org/10.1145/2950290.2950322`

[29] Li N, Praphamontripong U, Offutt J (2009) An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: Software Testing, Verification and Validation Workshops, International Conference on, pp 220–229, DOI 10.1109/ICSTW.2009.30

[30] Lindström B, Márki A (2016) On strong mutation and subsuming mutants. In: Procs. 11th Inter. Workshop on Mutation Analysis

[31] Ma YS, Offutt J, Kwon YR (2005) MuJava: an automated class mutation system. Software Testing, Verification and Reliability 15(2):97–133, DOI 10.1002/stvr.308

[32] Offutt AJ, Pan J (1997) Automatically detecting equivalent mutants and infeasible paths. Softw Test, Verif Reliab 7(3):165–192, DOI 10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U, URL `http://dx.doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U`

[33] Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C (1996) An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology 5(2):99–118

[34] Offutt J (2011) A mutation carol: Past, present and future. Information & Software Technology 53(10):1098–1107, DOI 10.1016/j.infsof.2011.03.007

[35] Pacheco C, Ernst MD (2007) Randoop: feedback-directed random testing for java. In: Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, pp 815–816, DOI 10.1145/1297846.1297902, URL `http://doi.acm.org/10.1145/1297846.1297902`

[36] Papadakis M, Malevris N (2010) Automatic mutation test case generation via dynamic symbolic execution. In: Software Reliability Engineering, 21st International Symposium on, pp 121–130, DOI 10.1109/ISSRE.2010.38

[37] Papadakis M, Malevris N (2010) An empirical evaluation of the first and second order mutation testing strategies. In: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings, pp 90–99, DOI 10.1109/ICSTW.2010.50, URL `http://dx.doi.org/10.1109/ICSTW.2010.50`

[38] Papadakis M, Traon YL (2015) Metallaxis-fl: mutation-based fault localization. Softw Test, Verif Reliab 25(5-7):605–628, DOI 10.1002/stvr.1509, URL `http://dx.doi.org/10.1002/stvr.1509`

[39] Papadakis M, Delamaro ME, Traon YL (2014) Mitigating the effects of equivalent mutants with mutant classification strategies. Sci Comput Program 95:298–319, DOI 10.1016/j.scico.2014.05.012, URL http://dx.doi.org/10.1016/j.scico.2014.05.012

[40] Papadakis M, Jia Y, Harman M, Traon YL (2015) Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: 37th International Conference on Software Engineering, vol 1, pp 936–946, DOI 10.1109/ICSE.2015.103

[41] Papadakis M, Henard C, Harman M, Jia Y, Traon YL (2016) Threats to the validity of mutation-based test assessment. In: Proceedings of the 2016 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2016

[42] Rani S, Suri B, Khatri SK (2015) Experimental comparison of automated mutation testing tools for Java. In: Reliability, Infocom Technologies and Optimization, 4th Inter. Conference on, pp 1–6, DOI 10.1109/ICRITO.2015.7359265

[43] Visser W (2016) What makes killing a mutant hard. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, pp 39–44, DOI 10.1145/2970276.2970345, URL http://doi.acm.org/10.1145/2970276.2970345

[44] Yao X, Harman M, Jia Y (2014) A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Procs. of the 36th Inter. Conf. on Software Engineering, pp 919–930, DOI 10.1145/2568225.2568265

[45] Zhang L, Hou S, Hu J, Xie T, Mei H (2010) Is operator-based mutant selection superior to random mutant selection? In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, pp 435–444, DOI 10.1145/1806799.1806863, URL http://doi.acm.org/10.1145/1806799.1806863

[46] Zhang L, Marinov D, Zhang L, Khurshid S (2012) Regression mutation testing. In: International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012, pp 331–341, DOI 10.1145/2338965.2336793, URL http://doi.acm.org/10.1145/2338965.2336793