Empir Software Eng
https://doi.org/10.1007/s10664-017-9582-5

How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults

Marinos Kintis[1] · Mike Papadakis[1] · Andreas Papadopoulos[2] · Evangelos Valvis[2] · Nicos Malevris[2] · Yves Le Traon[1]

Abstract Mutation analysis is a well-studied, 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 the ratio that is revealed by the candidate tests. Unfortunately, applying mutation to real-world programs requires automated tools due to the vast number of defects involved. In such a case, the effectiveness of the method strongly depends on the peculiarities of the employed tools. Thus, when using automated tools, their implementation inadequacies can lead to inaccurate results. To deal with this issue, we cross-evaluate four mutation testing tools for Java, namely PIT, 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

Communicated by: Michaela Greiler and Gabriele Bavota

✉ Marinos Kintis
marinos.kintis@uni.lu

Mike Papadakis
michail.papadakis@uni.lu

Andreas Papadopoulos
p3100148@aueb.gr

Evangelos Valvis
p3130019@aueb.gr

Nicos Malevris
ngm@aueb.gr

Yves Le Traon
yves.letraon@uni.lu

[1] Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg, Luxembourg

[2] Department of Informatics, Athens University of Economics and Business, Athens, Greece

Published online: 21 December 2017

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 the other tools by finding 6% more faults than the other tools combined.

Keywords Mutation testing · Fault detection · Tool comparison · Human study · Real faults

1 Introduction

Software testing forms the most popular practice for identifying software defects (Ammann and Offutt 2008). 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 (Chekam et al. 2017), and capable of subsuming, or probably subsuming, almost all the structural testing techniques (Ammann and Offutt 2008; Jia and Harman 2011). 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 behaviour of the original (non-mutated) and the defective (mutant) programs, they are also capable of distinguishing the expected behaviour from the faulty one (Chekam et al. 2017).

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 to 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 that it can lead experiments to incorrect conclusions (Papadakis et al. 2016). 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 employed).

To date, many mutation testing tools have been developed and used by researchers and practitioners (Papadakis et al. 2017). 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 four mutation testing tools on a set of real faults from the Defects4J benchmark (Just et al. 2014). These tools are: three widely-used tools for Java (Papadakis et al. 2017), namely MUJAVA (Ma et al. 2005), MAJOR (Just et al. 2011) and PIT (Coles 2010) and the research version of PIT, PIT_{RV} (Coles et al. 2016). We also included PIT_{RV} in the tools studied because it is an improved version of PIT and our initial results suggest that it is a more effective version of the tool (Laurent et al. 2017a). We complement our analysis using human analysis and comparison

of the tools to detect potential weaknesses and suggest improvements. Our results demonstrate that PIT_{RV} is more effective than the other tools, managing to reveal approximately 6% more real faults, with a statistically significant difference. 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 whilst 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 manual study shows that the mutants of MUJAVA (for the subjects studied) require 138 test cases in order to be killed, the ones of MAJOR 97, the ones of PIT_{RV} 105 and the ones of PIT 80. With respect to the number of equivalent mutants, MUJAVA, MAJOR, PIT_{RV} and PIT produced 203, 94, 382 and 43, respectively.

This paper forms an extended study of our previous one (Kintis et al. 2016), published in the 16th International Working Conference on Source Code Analysis and Manipulation, which investigated the effectiveness of MUJAVA, MAJOR and PIT based on manual analysis. We extend our 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 (Coles et al. 2016). The extended results demonstrate that PIT_{RV} forms the most prominent choice with respect to fault revelation as it outperforms the other tools both in terms of fault and mutant detection. Another new finding is that MUJAVA does not operate well on most of the subjects of the benchmark set we study and that PIT and MAJOR perform similarly. Overall, the contributions of the present paper can be summarised in the following points:

1. A controlled study investigating the fault revelation ability of four mutation testing tools for the Java programming language, including the three most popular ones.
2. An extensive, manual study of 6,493 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 presents the posed research questions and the adopted experimental procedure. Sections 4 and 5 discuss our results and potential threats to the validity. Finally, Section 6 details the previous research studies and Section 7 concludes the paper.

2 Background

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

2.1 Mutation Testing

Mutation testing is a well-studied technique that introduces artificial faults to the program under test, which is termed the *original* program in mutation's terminology. According to the findings of the most recent survey in the area (Papadakis et al. 2017), which summarises the advances in the field between 2008 and 2017, mutation enjoys increased research interest with a growing trend of high-quality publications per year.

Mutation induces simple syntactic changes to the program under test, creating many different versions of the original program, termed `mutants`. Mutants are produced using a set of syntactic rules called *mutation* or *mutant operators*. The process requires practitioners to design test cases that are able to distinguish the mutants' behaviour from that of the original program. In essence, these test cases should force the original program and its mutants to result in different outputs, i.e., they should *kill* the mutants. In this study, we investigate the *fault-revealing* ability of mutants (produced by the tools studied) based on real faults. We say that a mutant *reveals* a fault iff the test cases that kill the mutant also lead to the discovery of the underlying fault.

Unfortunately, not all mutants can be killed. Some mutants are semantically equivalent to the original program but syntactically different, thus, they result in the same output as the original program for all possible inputs. These mutants are called *equivalent mutants*. As mentioned earlier, test cases are evaluated based on their ability to kill mutants. This effectiveness measure is called *mutation score* and is the ratio of killed mutants to the killable ones.[1] A test suite that kills all killable mutants, i.e. achieves an 100% mutation score, is called *mutation adequate test suite*.

The problem of identifying and removing equivalent mutants is known as the *Equivalent Mutant Problem* (Kintis 2016; Kintis et al. 2017a; Papadakis et al. 2015). Regrettably, the Equivalent Mutant Problem has been shown to be undecidable in its general form (Budd and Angluin 1982), 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., Papadakis et al. (2015), Kintis et al. (2015), and Kintis and Malevris (2015).

Another problem of mutation testing is that it produces many mutants that are *redundant*, i.e., they are killed when other mutants are killed (Kintis et al. 2010). 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 (Papadakis et al. 2016). Therefore, when mutation testing is used as a comparison basis, there is a need to deflate the mutation score measurement by removing redundant mutants. This can be done by using a subset of the generated mutants which is representative of all the generated ones in the sense that when its mutants are killed, all killable mutants (of the generated set) are also killed.

A way to approximate this representative set was first introduced by Kintis et al. (2010) with the notion of the *disjoint mutants*. Disjoint mutants approximate the minimum "subset of mutants that need to be killed in order to reciprocally kill the original set". Ammann et al. (2014) presented another way to approximate this set based on the notion of *dynamic mutant subsumption*. Given a test set T, mutant A dynamically subsumes mutant B if the test cases that kill A also kill B and A is killable based on T. In our study, we use disjoint mutants, computed following the best practices presented in the recent survey in the area (Papadakis

[1] The killable mutant set is the set of all the generated mutants excluding the equivalent ones.

et al. 2017), to remove mutant redundancy and utilise the *disjoint mutation score* as a more accurate alternative of the mutation score (Papadakis et al. 2016). Similarly to the definition of the mutation score, the disjoint mutation score is the ratio of the killed disjoint mutants to the total disjoint ones.

Mutation's effectiveness depends largely on the used mutants (Ammann and Offutt 2008). 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 four mutation testing tools for the Java programming language, namely MUJAVA, MAJOR, PIT and PIT_{RV}, based on real faults and on the results of an extensive manual study. In particular, we evaluate the tools based on their *fault-revealing* ability, i.e., based on the number of faults their mutants can reveal and on their *mutant-killability*, i.e., on the ability of the produced mutants to kill the mutants of the other tools. Before presenting the conducted empirical study, the considered tools and their implementation details are introduced.

2.2 Java Mutation Testing Tools

Our recent survey, summarising 10 years of mutation testing research (2008–2017), concluded that mutation is a well-studied technique with a plethora of mutation testing tools available (Papadakis et al. 2017). In the present study, we choose to work on Java since it is widely used by practitioners and forms the subject of most of the recent research papers. Thus, we selected to evaluate the effectiveness, in term of real-fault detection and mutant-killability, of the most popular Java mutation testing tools. In our survey (Papadakis et al. 2017), we analysed more than 400 papers related to mutation research and found that the most frequently used Java mutation testing tools are MUJAVA (Ma et al. 2005), MAJOR (Just et al. 2011) and PIT (Coles 2010). Thus, in this study we focus on these tools and the research version of PIT, PIT_{RV} (Coles et al. 2016), whose initial evaluation suggested that it outperforms PIT (Laurent et al. 2017a). In the following, these tools are briefly introduced.

2.2.1 MUJAVA – *Source Code Manipulation*

MUJAVA (Ma et al. 2005) 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 (Offutt et al. 1996), 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.

2.2.2 PIT – *Bytecode Manipulation*

PIT (Coles 2010) 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. Table 2 describes the corresponding operators. For example, the "Void Method Calls" operator deletes method calls of methods that do not return a value and the "Non Void Method Calls" operator removes method calls of methods that return a value, replacing the method call with a predefined value, e.g. `null`. By comparing this table with Table 1, it can be seen that PIT implements differently specific mutation operators of MUJAVA, for instance, the changes imposed by PIT's Conditionals Boundary (CB) operator are a subset of the ones of MUJAVA's Relational Operator Replacement (ROR). 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's changes are performed at the bytecode level, they cannot always be mapped onto source code ones.

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*	$\{((\text{a } op \text{ b}),\ \texttt{false}), ((\text{a } op \text{ b}),\ \texttt{true}), (op_1,\ op_2) \mid op_1, op_2 \in \{>, >=, <, <=, ==, !=\} \wedge op_1 \neq op_2\}$
COR: *Conditional Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\&\&, \mid\mid, \hat{}\} \wedge op_1 \neq op_2\}$
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*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\&, \mid, \hat{}\} \wedge op_1 \neq op_2\}$
LOI: *Logical Operator Insertion*	$\{(v, \sim v)\}$
LOD: *Logical Operator Deletion*	$\{(\sim v, v)\}$
ASRS: *Short-Cut Assignment Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{+=, -=, *=, /=, \%=, \&=, \mid=, \hat{}=, >>=, >>>=, <<=\} \wedge op_1 \neq op_2\}$

Table 2 Mutation operators of PIT

Mutation operator	Description
AP: *Argument Propagation*	$\{(nonVoidMethodCall(..., par), par)\}$
CB: *Conditionals Boundary*	$\{(op_1, op_2) \mid (op_1, op_2) \in \{(<, <=), (<=, <), (>, >=), (>=, >)\}\}$
CC: *Constructor Calls*	$\{(\texttt{new } AClass(), \texttt{null})\}$
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*	$\{(v, \texttt{-}v)\}$
M: *Math*	$\{(op_1, op_2) \mid (op_1, op_2) \in \{(+, \texttt{-}), (\texttt{-}, +), (*, /), (/, *), (\%, *), (\&, \mid), (\mid, \&), (\hat{}, \&), (<<, >>), (>>, <<), (>>>, <<)\}\}$
MV: *Member Variable*	$\{(\texttt{member\_var=...,member\_var=b}) \mid b \in \{\texttt{false}, 0, 0.0, \texttt{'\textbackslash u0000'}, \texttt{null}\}\}$
NC: *Negate Conditionals*	$\{(op_1, op_2) \mid (op_1, op_2) \in \{(==, \texttt{!=}), (\texttt{!=}, ==), (<=, >), (>=, <), (<, >=), (>, <=)\}\}$
NVMC: *Non Void Method Calls*	$\{(nonVoidMethodCall(),\ c) \mid c \in \{\texttt{false}, 0, 0.0, \texttt{'\textbackslash u0000'}, \texttt{null}\}\}$
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
RV: *Return Values*	$\{(\texttt{return } a, \texttt{return } b) \mid (a, b) \in \{(\texttt{true}, \texttt{false}), (\texttt{false}, \texttt{true}), (0, 1), ((\texttt{int})\ x, 0), ((\texttt{long})\ \texttt{x}, \texttt{x+1}), ((\texttt{float})\ x, \texttt{-(x+1.0)}), (\texttt{NAN}, 0), (\texttt{non-null}, \texttt{null}), (\texttt{null}, \texttt{throw } RuntimeException)\}\}$
S: *Switch*	Replaces the `switch`'s labels with the default one and vice versa (only for the first label that differs)
VMC: *Void Method Calls*	$\{(voidMethodCall(), \varnothing)\}$

Table 3 Mutation operators of MAJOR

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

2.2.3 MAJOR – AST MANIPULATION

MAJOR (Just et al. 2011) 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. For instance, the tool implements the "Logical Operator Replacement" which replaces logical operators with each other and the "Conditional Operator Replacement" which replaces the "&&" and "||" conditional operators with predefined sets of operators and constructs, e.g. the left-hand side (LHS) or the right-hand side (RHS) of the expression that contains the "&&" and "||" operators. 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.

2.2.4 PIT_{RV} – *the Research Version of PIT*

PIT_{RV} (Coles et al. 2016; Laurent et al. 2017b) is the research version of PIT (Coles 2010) which greatly extends PIT's supported mutation operators with the aim of improving the tool's effectiveness. For example, it implements the "Absolute Value Insertion" operator which inserts the negation of arithmetic variables and the "Constant Replacement" operator which replaces constants with predefined values or negates them. As can be seen from Table 4, which describes the tool's supported operators, PIT_{RV}'s operators form a superset of the ones supported by PIT. Additionally, the tool supports most of MUJAVA's operators and, similarly to PIT, it employs mutation operators that are not implemented in MUJAVA.

Table 4 Mutation operators of PIT_{RV}

Mutation operator	Description
ABS: *Absolute Value Insertion*	$\{(v, \texttt{-}v)\}$
AOD: *Arithmetic Operator Deletion*	$\{((\texttt{a}\ op\ \texttt{b}),\ \texttt{a}), (\texttt{a}\ op\ \texttt{b}),\ \texttt{b})) \mid \text{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}\}$
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*	$\{(nonVoidMethodCall(..., par),\ par)\}$
CRCR: *Constant Replacement*	{(const, −const), (const, 0), (const, 1), (const, const − 1), (const, 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*	$\{(\texttt{new}\ AClass(), \texttt{null})\}$
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*	$\{(op_1, op_2) \mid (op_1, op_2) \in \{(\texttt{+}, \texttt{-}), (\texttt{-}, \texttt{+}), (\texttt{*}, \texttt{/}), (\texttt{/}, \texttt{*}), (\texttt{\%}, \texttt{*}), (\texttt{\&}, \texttt{\|}), (\texttt{\|}, \texttt{\&}), (\texttt{\^}, \texttt{\&}), (\texttt{<<}, \texttt{>>}), (\texttt{>>}, \texttt{<<}), (\texttt{>>>}, \texttt{<<})\}\}$
MV: *Member Variable*	$\{(\texttt{member\_var=...,member\_var=b}) \mid b \in \{\texttt{false}, 0, 0.0, \texttt{'\textbackslash u0000'}, \texttt{null}\}\}$
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*	$\{(nonVoidMethodCall(),\ c) \mid c \in \{\texttt{false}, 0, 0.0, \texttt{'\textbackslash u0000'}, \texttt{null}\}\}$
OBBN: *Bitwise Operator Mutation*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\texttt{\&}, \texttt{\|}\} \wedge op_1 \neq op_2\}$
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. $\{((\texttt{a}\ op\ \texttt{b}),\ \texttt{true})\ or\ ((\texttt{LHS \&\& RHS}),\ \texttt{RHS})\}$
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
RV: *Return Values*	$\{(\texttt{return}\ a, \texttt{return}\ b) \mid (a, b) \in \{(\texttt{true}, \texttt{false}), (\texttt{false}, \texttt{true}), (0, 1), ((\texttt{int})\ x, 0), ((\texttt{long})\ \texttt{x}, \texttt{x+1}),\ ((\texttt{float})\ x, \texttt{-(x+1.0)}), (\texttt{NAN}, 0), (\texttt{non-null}, \texttt{null}), (\texttt{null}, \texttt{throw}\ RuntimeException)\}\}$
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*	$\{(voidMethodCall(), \varnothing)\}$

3 Design of the Experiment

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

3.1 Research Questions

Mutation testing is popular and widely-used in research (Papadakis et al. 2017). 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) existing research results. Therefore, our first question regards the effectiveness of the studied tools (measured in terms of real fault revealing ability). 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 differences in terms of fault detection, we demonstrate that the choice of mutation testing tools really matters. 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 when mutation adequate test suites are used. In other words, we seek to investigate how effective are the studied tools in killing the mutants of the other tools and we ask:

RQ2: *Does any mutation testing tool lead to tests that kill all the killable mutants produced by the other tools?*

This comparison enables checking 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 mutant-killability effectiveness. A negative answer to this question indicates that there are mutants not covered by the mutation adequate test suites of the tools. We view these missed mutants as weaknesses of the tools. The main difference from RQ1 is that we perform an objective comparison with mutation adequate test suites, which helps reducing potential threats to validity.

To further compare the mutation testing tools and identify their weaknesses, we need to assess the quality of their mutation adequate test suites compared to 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 that attempts to approximate this "ground truth". This set is composed of the disjoint mutants of all mutants generated by the tools combined. Therefore we ask:

RQ3: *How do the studied tools perform compared to a reference mutant set?*

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 this analysis, we can identify the most effective (in terms or mutant killability) mutation testing tool and quantify the differences between the tools.

This is important when choosing a tool, 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:

RQ4: *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. Whilst 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:

RQ5: *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.

3.2 Mutation Testing Tools

To answer the stated RQs, we applied mutation testing by independently using each one of the selected tools. More precisely, we used version 3 of MUJAVA, version 1.1.8 of MAJOR, version 1.1.10 of PIT (Coles 2010) and the version of PIT_{RV} presented by Coles et al. (2016) and Laurent et al. (2017b), and applied all the mutation operators provided. In the case of MUJAVA, only the method-level operators were employed, since the other tools do not provide object-oriented operators.

3.3 Test Subjects

We study two sets of subjects; one set of large, real-world programs and one set of small methods (units). The first set is used to assess the real-fault revelation of the tools, whilst the second one to get results based on manual analysis. Unfortunately, manually analysing large real-world programs is time consuming and requires extensive manpower. Therefore, to complete the experiment with reasonable resources we used small units that we selected from our prior study (Kintis et al. 2016).

The set of large real-world programs used belong to the Defects4J database (Just et al. 2014) (version 1.1.0). This is a benchmark set of reproducible real faults mined from source code repositories. The benchmark contains real faults carefully located and isolated using the version control and bug tracking systems of several 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 5 presents the name of the projects utilised in this study (first column), a small description of their application domain (second column), the source code lines as reported by the `cloc` tool[2] (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 (Just et al. 2014) and on its GitHub page.[3]

The second set (with the small units) is composed of 12 methods; 10 of them were randomly picked from 4 real-world projects (Commons-Math, Commons-Lang, Pamvotis and

[2] https://github.com/AlDanial/cloc

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

Table 5 Fault benchmark set details

Test Subject	Description	LoC	#Real Faults	#Gen. Tests	#Faults Found
JFreeChart	A chart library	79,949	26	3758	17
Closure	Closure compiler	91,168	133	3552	12
Commons Lang	Java utilities library	45,639	65	6408	30
Commons Math	Mathematics library	22,746	106	8034	53
Joda-Time	A date and time library	79,227	27	1667	13
Total	–	318,729	357	23,419	125

XStream) and another 2 (Triangle and Bisect) from the mutation testing literature (Ammann and Offutt 2008). Details regarding the selected subjects are presented in Table 6. The table presents the name of the test subjects, the names of the studied methods, their source code lines and number of branches as reported by the JaCoCo library,[4] the number of generated and disjoint mutants per tool and the number of the resulting mutants of the reference mutant set.

3.4 Test Suites

To perform our analysis we need test suites. Unfortunately, the two sets of subject we have contain only some developer tests from the projects' repositories. This is potentially problematic in our case since developer tests have small overlap between them (as each test case is testing a different scenario) and are not produced by any systematic procedure. As a result, very few of these tests reveal the faults. Furthermore, these tests have not been generated using any controlled or known procedure and, thus, they can introduce several threats to the validity of our results, underestimating our measurements. To circumvent this problem, we used automated tools, to generate test suites (for the large programs, recorded in Table 5) and manually analysis to generate test suites (for the set of the small programs, recorded in Table 6).

3.4.1 Automatically Generated Test Suites

To simulate the mutation-based test process, we used multiple test suites that were generated by two state-of-the-art test generation tools, namely EvoSuite (Fraser and Arcuri 2011) and Randoop (Pacheco and Ernst 2007). 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.

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. It should be noted that in the case of EvoSuite, the test cases were generated based on the branch coverage criterion. 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.[5]

[4] http://www.jacoco.org/

[5] The reason we chose to generate only one test suite with Randoop is that it generates far more tests than EvoSuite.

Table 6 Test subject details: generated mutants, disjoint mutants and reference mutant set

			# Generated mutants				# Disjoint mutants				# Mutants
Test Subject	Method	LoC (# br.)	PIT	MAJOR	PIT_{RV}	MUJAVA	PIT	MAJOR	PIT_{RV}	MUJAVA	Reference mutant set
Commons-Math	gcd	23 (22)	79	133	392	237	9	7	12	9	10
	orthogonal	11 (10)	65	120	392	155	11	11	11	11	11
Commons-Lang	toMap	20 (10)	50	23	115	32	6	6	6	5	10
	subarray	13 (8)	27	25	95	64	6	6	6	3	6
	lastIndexOf	15 (16)	43	29	139	81	8	11	18	13	17
	capitalize	18 (14)	42	37	132	69	4	5	4	4	5
	wrap	34 (16)	70	71	328	198	7	12	12	16	13
Pamvotis	addNode	35 (10)	53	89	447	318	16	16	23	29	37
	removeNode	11 (6)	29	18	109	55	6	7	6	6	6
Triangle	classify	22 (34)	94	139	486	354	16	31	26	31	55
XStream	decodeName	30 (24)	81	73	315	156	7	8	9	8	10
Bisect	sqrt	21 (6)	29	51	219	135	6	7	6	7	6
Total	–	253 (176)	662	808	3169	1854	102	127	139	142	186

2. We systematically removed flaky test cases from the generated test suites, i.e. 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.

Based on the above procedure, we generated the test suites that were used for the purposes of our large-scale experiment (Step 1) and removed flaky tests (Step 2). It is known that flaky tests exist in practice (Luo et al. 2014) and that automated test generation tools can create such tests (Shamshiri et al. 2015). In our case, flaky tests were mostly generated for `Joda-Time` and `Closure`; in the first case, tests were susceptible to the date that they were created on and, in the second, to the project version (buggy or fixed) that they were created for. Finally, we discovered which faults the generated test suites managed to detect (Step 3), from the 357 developer faults of the Defects4J database. In order for a fault to be detected by a generated test suite, the test suite must contain at least one test case that fails when executed against the project version that contains the fault, i.e. the buggy version. We term this test suite a *triggering* one. In total, our test suites managed to detect 125 faults; for the remaining faults no failure was triggered, thus, the tests did not manage to detect them.

Table 5 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. Based on these results, our fault set consists of 125 real faults from 5 open source projects and our triggering test suites are composed of 23,419 test cases.

3.4.2 Manually Generated Test Suites

To complement our analysis and identify potential weaknesses of the tools, 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 4 times, one time per studied tool. Each analysis was not dependent on the others as it was performed by a different person. Thus, in total, we manually analysed 48 methods and 6,493 mutants which constitutes one of the largest studies in the literature of mutation testing to the best of our knowledge, e.g., Yao et al. (2014) consider 4,181 mutants, Baker and Habli (2013) 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 6).

The primary objective of this experiment is to cross-evaluate the mutant-killability of the mutation adequate test suites of the tools (for the programs recorded in Table 6) and identify potential weaknesses in their mutation testing practice. To this end, we performed a complete manual analysis of the mutants produced by the tools by designing tests that kill all killable mutants, i.e., by creating mutation adequate test suites, and manually identifying the equivalent mutants generated. To perform this task, we asked different persons to apply mutation testing on our subjects for each tool studied. 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. Specifically, students that had grades greater than 7 out of 10 in programming and software engineering courses were favoured. In total, 8 teams consisted of 1 or 2 students were selected. These teams were responsible to apply mutation on specific methods that were given to them. It should be noted that the teams did not have any time limitation for the application of mutation; when they finished the analysis of one method, another one was given to them for analysis until the end of the respective semester. Before moving to the mutation analysis of another method, the results of their previous analysis were checked for correctness by one author of this paper. All students 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 by one of the authors of this paper, with regular team meetings throughout the semester.

– 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, i.e. test suites that covered all feasible branches of the corresponding methods, and then to randomly pick a live mutant and attempt to kill it incrementally by fulfilling the conditions of the RIP Model[6] (Ammann and Offutt 2008), until all live mutants were killed or identified as equivalent. Next, we minimised these mutation adequate test suites by checking, for each contained test case, whether its removal would result in a decreased mutation score (Ammann and Offutt 2008). More precisely, we adopted the following process: for each test case, we removed it from the corresponding test suite and examined if the mutation score drops; if it does, it means that the test is required; in a different case, the test is redundant and should be removed. The manual experiment of our study is concerned with evaluating the effectiveness of the test suites that are explicitly designed to kill the mutants of one tool in killing the mutants of the other tools. Since redundant tests are not needed to satisfy any of the criterion requirements (i.e. kill any additional mutant), they can artificially result in overestimating the strengths of the test suites and bias our results (Ammann and Offutt 2008). Consider the case where for a mutation tool, say X, we have generated 10 test cases for its mutation adequate test suite but only 4 out of these 10 are needed to kill all its mutants. If we now compare these 10 tests against the mutants of another mutation testing tool, say Y, we bias our results towards X because these 6 redundant tests may kill additional mutants of Y that the 4 mutation adequate tests would have failed to, thus, overestimating the mutant-killability of the mutation adequate test suites of X. To avoid this bias, we minimised the generated mutation adequate test suites.

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 (Ammann and Offutt 2008) and the sub-categories of equivalent mutants described by Yao et al. (2014). Also, all detected equivalent mutants were independently verified by at least

[6]The RIP model states that mutants are killed by test cases that reach (execute) the mutated statement and manage to cause an infection on the program state (the mutant and the original program are in a different state at the mutated point) and manifest the infection to the program output (the different program states result in different outputs).

one of the authors. It should be noted that since PIT_{RV}'s mutants form a superset of PIT mutants, to manually analyse them one of the authors of this paper extended the manual analysis of the PIT mutants 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 (Kintis et al. 2017b, c).

3.5 Analysis Procedure

To reliably compare the selected tools, it is mandatory to account for redundant mutants (Papadakis et al. 2016, 2017). Accounting for redundant mutants is necessary in order to avoid bias from their inflating the mutation score (Papadakis et al. 2016), whilst the use of mutation adequate tests ensures the accurate estimation of the tools' effectiveness (Papadakis et al. 2017). An inaccurate estimation may happen when failing to kill some 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 for fault revelation (Andrews et al. 2006; Visser 2016). Additionally, the majority of the mutants that are actually helping to improve test suites are the hard to kill mutants. Since we know that very few mutants contribute to the test process (Papadakis et al. 2016), 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 (Papadakis et al. 2015; Kintis et al. 2017a). Therefore, we are forced to use non-adequate test suites for large programs. To account for the above reasons, our empirical study involves two parts: the fault-revelation experiment, performed on open source projects with real faults, whose results answer RQ1; and, the mutant-killability experiment, performed based on manual analysis of mutants of sampled functions, whose results answer RQs 2 to 5. RQ1 relies on non-adequate test suites generated for the large real world programs of Table 5, whilst RQs 2 to 5 rely on mutation-adequate test suites specially designed for each tool that we study for the programs of Table 6. For details about the test generation process that we followed, please refer to Section 3.4.

In RQ1, we are interested in the fault revelation ability of mutation-based test suites based on the 5 Defects4J subjects (Table 5). Thus, we want to see whether mutation-driven test cases reveal the faults studied. We measure the fault revelation ability of the studied mutation testing tools by evaluating the fault-revealing ability of the mutants they produce. 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. This means that testers aiming at killing these mutants will generate test cases that reveal the studied faults. Of course, we approximate the true fault revealing cases based on our generated test suites. Thus, if a mutant is killed only by test cases that reveal a studied fault, we consider that the mutant *reveals* the fault, i.e., leads to test cases that reveal the fault.

It is noted that this approach is in accordance with the traditional fundamental premises of mutation-based test selection process (Geist et al. 1992). According to Geist et al. (1992) and Ammann and Offutt (2008):

> "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."

Evidence supporting this premise has been provided by many researchers, e.g., Chekam et al. (2017). Therefore, to answer RQ1, we compare the number of revealed faults per tool and project and rank them accordingly.

To answer RQ2, we used the selected methods and the manually generated mutation adequate test suites (subjects of Table 6). 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, none of them subsumes the others. For calculating the disjoint mutation score, the procedure outlined in the recent survey of Papadakis et al. (2017) was followed. To compute this score, 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.

To answer RQ3, we used the mutation adequate test suites of each one of the studied tools (from the manually analysed subjects, i.e. Table 6) and measured the disjoint mutation score they achieve when evaluated against the reference mutant set. The reference mutant set was constructed by identifying the disjoint mutants of the mutant set composed of all mutants of all the studied tools for the manually analysed subjects. This score provides the common ground to compare the tools and rank them with respect to their mutant-killability effectiveness.

The use of disjoint mutants sets in answering RQ2 and RQ3 is necessary to avoid inflating the mutation scores from redundant mutants, i.e., mutants subsumed by other mutants of the merged set of mutants (Papadakis et al. 2016). Redundant mutants inflate the mutation score measurement with the unfortunate result of committing Type I errors (Papadakis et al. 2016). 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.

To answer RQ4, for each tool we analysed the mutants that remained alive after the execution of the mutation adequate test suites 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 RQ5, we measured and report the number of test cases required to construct mutation adequate test suites for the mutants of the tools and the number of equivalent mutants that we found.

4 Empirical Findings

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

4.1 RQ1: Tools' Real Fault Revelation Ability

This question investigates the performance of the studied tools with respect to fault revelation (studied using the subjects of Table 5). Figure 1 records our results. The figure depicts how many of the 125 faults the mutants of the tools managed or failed to reveal. Recall that

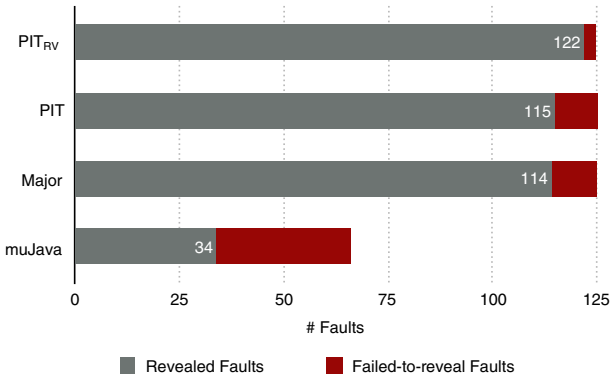

Fig. 1 Comparison of the tools' fault-revelation ability

a mutant reveals a fault iff the test cases that kill the mutant also lead to the discovery of the underlying fault (see also Section 2). It should be noted that the exact IDs of the faults detected by the tools are listed in Table 10 in Appendix A at the end of this paper.

As can be seen from the figure, PIT$_{RV}$ was successfully applied to all fixed program versions of the 125 faults and it managed to reveal 122 of the faults, i.e., for the 122 faults there was at least one generated mutant of PIT$_{RV}$ which is killed only by test cases that lead to the discovery of the respective fault. MAJOR and PIT also successfully run on all 125 fixed program versions but they managed to reveal only 114 and 115 faults, respectively. Finally, MUJAVA was unable to run on all program versions, it was able to handle only 66 of them and revealed 34 faults. Although, we report results for the 66 faults that the tool was able to run, in most of these cases the tool could not produce many mutants or properly operate. Hence, it detected only 34 out of the 66 faults. Since MUJAVA could not work properly on most of the test subjects, we excluded it from any further analysis we make on the Defects4J benchmark.

Based on the figure's results, it becomes evident that PIT$_{RV}$ managed to detect more faults than MAJOR and PIT and probably subsumes the other tools. However, it should also be noted that none of the tools subsumes the others; there are faults that are only revealed by 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 1 unique fault (`Time-27`) compared to PIT$_{RV}$ and 7 unique faults compared to PIT (`Chart-17`, `Time-13`, `Time-27`, `Math-25`, `Closure-7`, `Closure-42`, `Closure-103`). 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 7 unique faults compared to PIT (`Chart-17`, `Time-13` , `Math-25`, `Math-27`, `Closure-7`, `Closure-42`, `Closure-103`). Finally, PIT does not reveal any unique fault compared to PIT$_{RV}$ and reveals 8 unique faults compared to MAJOR (`Lang-56`, `Math-6`, `Math-22`, `Math-89`, `Math-105`, `Closure-27` , `Closure-49` , `Closure-52`). One interesting finding is that 2 faults (`Math-75`, `Math-90`) are not revealed by any tool. Overall, PIT$_{RV}$ managed to reveal 122 real faults out of 125 for which it run successfully, MAJOR revealed 114 out of 125, PIT 115 out of 125 and MUJAVA 34 out of 66.

In order to determine whether the observed difference between the fault revelation ability of the tools is statistically significant, we performed a statistical test. Since we want to

Table 7 Fault revelation: statistical analysis

Comparison	p-value	$\hat{\Delta}$	95% confidence interval (CI)
PIT_{RV} VS MAJOR	**0.011**	6.4%	1.8–12.5%
PIT_{RV} VS PIT	**0.008**	5.6%	2.5–11.1%
MAJOR VS PIT	0.067	–	−7.4–5.8%

compare the proportions of two different dichotomous measurements (e.g., the proportions of faults revealed by PIT_{RV} and MAJOR) when applied to the same sample (125 faults), we used the McNemar test. The McNemar test is a non-parametric, statistical test for the analysis of paired binomial proportions. Table 7 presents the results of the statistical analysis: (a) the (asymptotic) *p-value* obtained; (b) as an effect measure we adopt the difference Δ between the marginal proportions (Fagerland et al. 2014), estimated by the maximum likelihood estimate $\hat{\Delta}$ which denotes the estimated difference between the proportions studied; and, (c) the corresponding confidence intervals. For the application of the test, we followed the guidelines of Fagerland et al. (2014). In all cases, the hypotheses examined are the following:

- H_0 : there is no difference between the proportions of faults revealed by the two tools
- H_1: there is a difference between the proportions of faults detected by the two tools

As can be seen from the table, there is a statistically significant difference (*p-value* = 0.011) between the fault revelation ability of PIT_{RV} and MAJOR; the $\hat{\Delta}$ effect measure indicates that the fault revelation ability of PIT_{RV} is estimated to be 6.4% better than the one of MAJOR with a 95% confidence interval (CI) from 1.8% to 12.5%. Analogous, statistically significant results are obtained when comparing PIT_{RV} with PIT. Finally, we can see that the null hypothesis H_0 cannot be rejected when comparing the fault revelation ability of MAJOR and PIT (*p-value* = 0.067), thus, we do not have enough evidence to conclude that one of these two tools is better.

Our previous work (Kintis et al. 2016) that we currently extend has found that MUJAVA is the most effective tool, followed by MAJOR and PIT. The difference between that ranking and the one described earlier is that our previous work focused on mutant-killability whereas the aforementioned results on fault revelation based on real faults. Thus, any difference between these studies can be attributed to the different characteristics of mutants and real faults. Indeed, as presented later in this paper, our mutant-killability and efficiency results are in accordance to the ones presented in our previous work (Kintis et al. 2016). By also including real faults, this study investigates the effectiveness of the tools in a more thorough and pragmatic way.

To summarise this subsection's findings, PIT_{RV} was found to be the most effective tool at revealing faults with statistically significant differences compared to PIT and MAJOR. PIT_{RV} managed to reveal 122 faults out of the 125 studied with an estimated difference of approximately 6% compared to PIT and MAJOR. Regarding the comparison of MAJOR and PIT, our analysis concluded that there are not enough data to indicate that one of the tools is better than the other.

4.2 RQ2: Tools' Cross-evaluation

This question investigates the effectiveness of the tools in terms of mutant killability (studied using the subjects of Table 6). Table 8 presents the respective results per test subject.

Table 8 Tools' cross-evaluation results

	Major						PIT_{RV}					
	PIT-TS		PIT_{RV}-TS		muJava-TS		Major-TS		muJava-TS		PIT-TS	
Method	All	Dis.	All.	Dis.	All	Dis.	All.	Dis.	All	Dis.	All	Dis.
gcd	97.4%	71.4%	97.4%	87.5%	97.4%	71.4%	99.4%	58.3%	98.8%	58.3%	98.8%	78%
orthogonal	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	99.5%	90.9%	100.0%	100%
toMap	88.9%	66.7%	100.0%	100.0%	77.8%	83.3%	97.9%	66.6%	92.8%	83.3%	96.9%	50%
subarray	90.0%	66.7%	90.0%	66.6%	85.0%	50.0%	100.0%	100.0%	97.6%	83.3%	96.4%	83%
lastIndexOf	100.0%	100.0%	100.0%	100.0%	92.6%	91.0%	100.0%	100.0%	97.7%	83.3%	96.1%	76%
capitalize	93.5%	60.0%	93.5%	60.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	67%
wrap	98.4%	91.7%	100.0%	100.0%	100.0%	100.0%	99.0%	75.0%	99.3%	83.3%	97.6%	75%
addNode	98.7%	93.8%	100.0%	100.0%	100.0%	100.0%	81.8%	21.7%	99.5%	95.7%	83.8%	94%
removeNode	93.8%	85.7%	100.0%	100.0%	100.0%	100.0%	93.7%	33.3%	100.0%	100.0%	93.7%	71%
classify	90.2%	61.3%	97.0%	87.0%	100.0%	100.0%	97.1%	88.5%	100.0%	100.0%	95.7%	62%
decodeName	98.0%	87.5%	95.9%	80.0%	100.0%	100.0%	95.0%	55.5%	98.4%	55.5%	97.7%	89%
sqrt	93.6%	71.4%	100.0%	100.0%	100.0%	100.0%	99.0%	66.7%	99.5%	50.0%	97.5%	80%
Average	95.2%	79.7%	97.8%	90.1%	96.1%	91.3%	96.9%	72.1%	98.6%	82.0%	95.7%	77.0%
	PIT						muJava					
	PIT_{RV}-TS		Major-TS		muJava-TS		Major-TS		PIT-TS		PIT_{RV}-TS	
Method	All	Dis.	All.	Dis.	All	Dis.	All.	Dis.	All	Dis.	All	Dis.
gcd	100.0%	100.0%	100.0%	100.0%	97.1%	81.8%	99.5%	88.9%	100.0%	100.0%	100.0%	100.0%
orthogonal	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
toMap	100.0%	100.0%	100.0%	100.0%	91.7%	83.3%	100.0%	100.0%	96.0%	80.0%	100.0%	100.0%

Table 8 (continued)

	PIT						muJava					
	PIT_{RV}-TS		Major-TS		muJava-TS		Major-TS		PIT-TS		PIT_{RV}-TS	
Method	All	Dis.	All.	Dis.	All	Dis.	All.	Dis.	All	Dis.	All	Dis.
subarray	100.0%	100.0%	100.0%	100.0%	95.8%	83.3%	100.0%	100.0%	91.1%	66.7%	100.0%	100.0%
lastIndexOf	100.0%	100.0%	100.0%	100.0%	97.6%	87.5%	100.0%	100.0%	97.4%	85.0%	100.0%	100.0%
capitalize	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.2%	75.0%	98.2%	75.0%
wrap	100.0%	100.0%	98.5%	85.7%	98.5%	85.7%	98.9%	93.8%	96.1%	81.2%	100.0%	100.0%
addNode	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	76.5%	37.9%	78.2%	34.5%	100.0%	100.0%
removeNode	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	91.7%	50.0%	93.8%	66.7%	100.0%	100.0%
classify	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.1%	90.3%	94.9%	58.1%	99.1%	90.3%
decodeName	100.0%	100.0%	96.9%	71.4%	100.0%	100.0%	95.3%	62.5%	99.2%	87.5%	99.2%	87.5%
sqrt	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	97.5%	71.4%	100.0%	100.0%
Average	100.0%	100.0%	99.6%	96.4%	98.4%	93.5%	96.7%	85.3%	95.2%	75.5%	99.7%	96.1%

The table is divided into 2 parts, the upper part and the lower one, each of which is further divided into 2 large columns (apart from the one describing the test subjects): the "Major" column of the upper part of the table presents, in the 3 sub-columns contained, the percentage of mutants of the MAJOR tool that the mutation adequate test suites of PIT, PIT_{RV} and MUJAVA manage to kill. Each of these 3 columns contains 2 sub-columns (columns "All" and "Dis.") that depict the mutation scores achieved by the corresponding test suites when all generated mutants of MAJOR are considered (column "All") and when only the disjoint ones are used (column "Dis."); respectively, the "PIT_RV" column of this part of the table presents the same information but for the mutants of the PIT_{RV} tool when executed against the mutation adequate test suites of MAJOR, MUJAVA and PIT. The lower part of the table presents the respective data for the mutants of PIT and MUJAVA. To exemplify, the "gcd, Major/PIT-TS/All" cell of the upper part of the table indicates that for the `gcd` method the mutation adequate test suite of PIT managed to kill 97.4% of the killable mutants of MAJOR, i.e., achieved a 97.4% mutation score for the mutants of MAJOR. The "gcd, Major/PIT-TS/Dis." cell presents the same information but in this case the mutation adequate test suite of PIT is evaluated against the disjoint mutants of MAJOR.

By examining Table 8, 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 (*"PIT_{RV}-TS"* columns) perform the best, with an effectiveness of approximately 100% with respect to MUJAVA, 98% with respect to MAJOR and 100% with respect to PIT when all mutants are considered; for the disjoint ones, this score is 100% for PIT and drops to approximately 90% for MAJOR and 96% for MUJAVA. The next better performing tool is MUJAVA, whose mutation adequate test suites (*"muJava-TS"* columns) achieve an effectiveness of approximately 96%, 99% and 98% for MAJOR, PIT_{RV} and PIT, when all mutants are considered and approximately 91%, 82% and 93% for the disjoint ones, respectively. The mutation adequate test suites of MAJOR (*"Major-TS"* columns) achieve a mutation score of approximately 97% for PIT_{RV} and MUJAVA and 100% for PIT, for all generated mutants; for the disjoint ones, the effectiveness (mutant-killability) drops to 72%, 85% and 96%, respectively. Finally, PIT's mutation adequate test suites (*"PIT-TS"* columns) perform the worst, with an effectiveness of approximately 95% with respect to MUJAVA, MAJOR and PIT_{RV} when all mutants are considered; for the disjoint ones, this score drops to approximately 80% for MAJOR, 75% for MUJAVA and 77% for PIT_{RV}.

4.3 RQ3: 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 (studied on the subjects of Table 6). Figure 2 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 a 92% (mutant-killability) effectiveness score, followed by MUJAVA, MAJOR and PIT with 85%, 80% and 63%, respectively. An interesting observation from these results is that all tools fail to cover a percentage of the mutants of the reference mutant set which ranges from 0 to 57%. On average, PIT, MAJOR, MUJAVA and PIT_{RV} cannot kill 37%, 20%, 15% and 8% of the reference mutants, respectively.

Overall we found that PIT_{RV} is the top ranked tool, followed by MUJAVA, MAJOR and PIT. PIT_{RV} achieves a higher mutation score (w.r.t. the reference mutant set) than MUJAVA

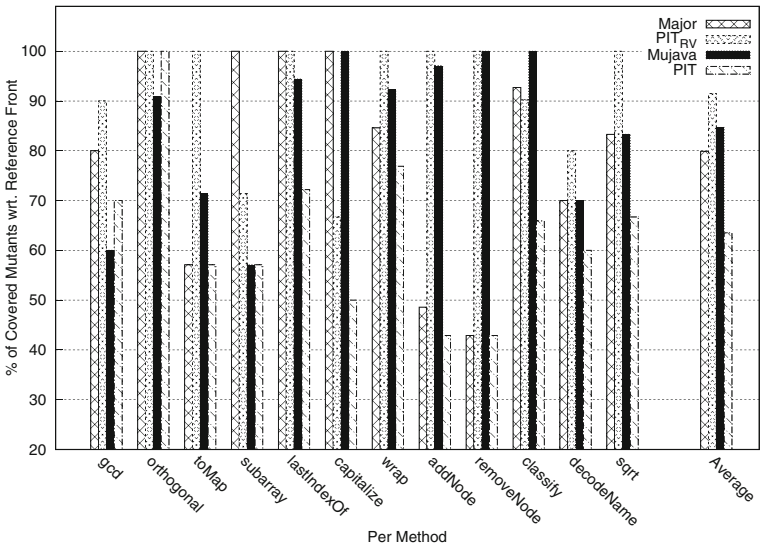

Fig. 2 Comparison of mutation adequate test suites against reference mutation set

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. Compared to PIT, PIT_{RV} performs better in 11 cases and equally in 1.

4.4 RQ4: Tools' Weaknesses and Recommendations

This question is concerned with ways of improving the mutation testing practice of the studied tools. To this end, Fig. 3 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. Note that PIT is not included in the analysis of this section since the previously-presented results suggest that PIT_{RV} is already a more effective version of PIT, thus, we only provide recommendations for the improvement of PIT_{RV}.

4.4.1 Recommendations: PIT_{RV}

As can be seen from Fig. 3, 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 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

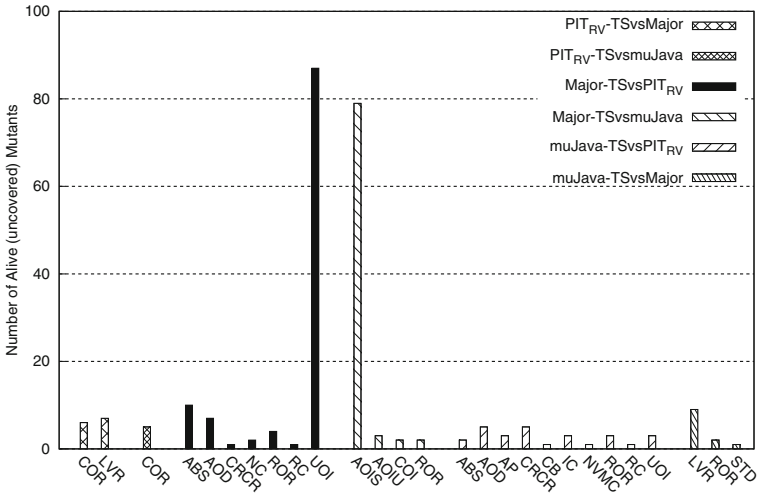

Fig. 3 Tools' weaknesses: number of alive mutants per mutation operator, test suite and tool

bytecode the zero constant is never pushed onto the stack. In order to make PIT$_{RV}$ more effective such cases should be handled accordingly.

4.4.2 Recommendations: MAJOR

By examining Fig. 3, 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 mutation testing practice can be enhanced by implementing the changes imposed by these operators. Additionally, MAJOR will benefit from 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 (2016).

4.4.3 Recommendations: MUJAVA

As can be seen from Fig. 3, 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 from 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

Table 9 Tools' application cost: number of equivalent mutants and required tests

	MAJOR		PIT_{RV}		MUJAVA		PIT	
Method	#Eq.	#Tests	#Eq.	#Tests	#Eq.	#Tests	#Eq.	#Tests
gcd	17	6	70	7	23	7	9	7
orthogonal	3	8	22	8	5	9	0	8
toMap	5	7	18	6	7	5	2	5
subarray	5	6	12	5	8	6	3	4
lastIndexOf	2	8	10	8	4	12	1	7
capitalize	6	5	22	4	14	9	1	6
wrap	8	10	36	7	19	7	4	6
addNode	11	8	57	22	33	34	3	8
removeNode	2	5	14	5	7	6	0	3
classify	7	25	42	22	38	27	1	16
decodeName	24	5	57	7	28	10	16	6
sqrt	4	4	22	4	17	6	3	4
Total	94	97	382	105	203	138	43	80

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.

4.5 RQ5: Tools' Application Cost

The answer to this question provides 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 required to kill all the corresponding killable mutants (studied using the subjects of Table 6). Table 9 presents the corresponding findings. The table is divided into 4 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 and 6% of PIT's ones. PIT was the tool that generated the least number of equivalent mutants, followed by MAJOR and MUJAVA, with PIT_{RV} generating the greatest number of equivalent mutants. Thus, PIT requires the least amount of human effort in identifying the equivalent mutants generated, whereas PIT_{RV} the greatest. Regarding the number of killing test cases the tools require, PIT requires 80 test cases, MAJOR 97, PIT_{RV} 105 and MUJAVA 138. Thus, PIT and MAJOR require the least amount of effort in generating mutation adequate test suites and MUJAVA the greatest, with PIT_{RV} placed in the middle.

It is interesting to notice that although PIT_{RV} generates a high number of mutants, it requires a considerably lower number of mutation adequate test cases indicating that the current version of the tool faces high mutant redundancy. As discussed in the Background section (Section 2), mutant redundancy is an important problem in mutation testing and all mutation tools suffer from it to a certain degree. Of course, redundancy depends on the number of generated mutants: more mutants increase the redundancy amongst them. PIT_{RV} generates a considerable number of mutants, thus, redundancy cannot be avoided which in

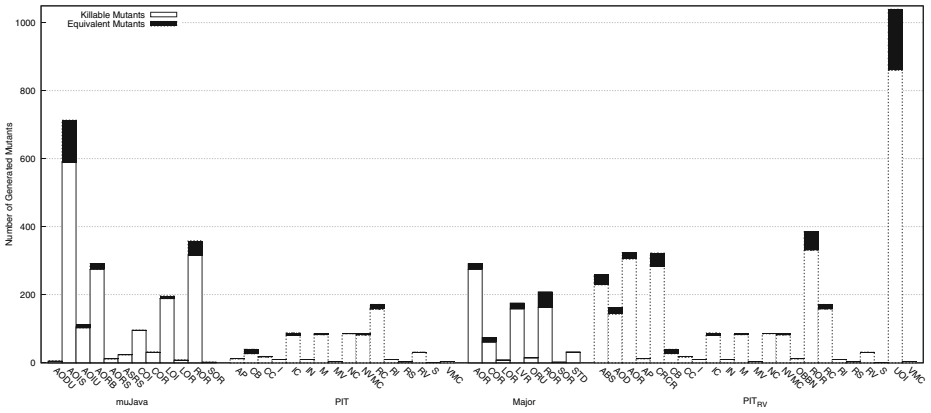

Fig. 4 Contribution of mutation operators to generated and equivalent mutants per tool

turn suggests that the efficiency of the tool can be improved. Additionally, the results on the number of mutation adequate test cases required by PIT_{RV} (presented earlier) support this statement. Future versions of PIT_{RV} should take this redundancy into account and attempt to reduce it. It should be mentioned that to avoid the problems caused by redundant mutants, we use disjoint mutants and the disjoint mutation score in our study (Kintis et al. 2010; Papadakis et al. 2017) (see also Section 2.1).

To better understand the nature of the generated equivalent mutants, Fig. 4 illustrates the contribution of each mutation operator to the generated killable and equivalent mutants per tool. In the case of PIT, RC and CB generate the most equivalent mutants and in the case of MUJAVA, AOIS and ROR. 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.

To summarise our findings, our results suggest that PIT_{RV} is the most effective tool in terms of fault revelation, generating mutants that manage to reveal 122 faults out of the 125 studied ones, followed by MAJOR and PIT, with 114 and 115 faults revealed, respectively (see also Section 4.1). PIT_{RV} also scores better when the tools are evaluated based on their mutant-killability with a 92% effectiveness score, followed by MUJAVA, MAJOR and PIT with 85%, 80% and 63%, respectively (see also Section 4.3). Finally, regarding the efficiency of the tools, our results indicate that PIT_{RV} is the most expensive tool, followed by MUJAVA, MAJOR and PIT (see also Section 4.5). 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, PIT requires less effort, a fact justified by its lower performance.

5 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 several real-world, open-source projects and real faults. We also used 6 additional test subjects and manually analysed 12 methods whose application domain varies. More research is needed to adequately answer questions related to the effectiveness of the tools on different domains and fault types. Moreover, as we used automatically generated test suites (to study the fault-revelation question), our results might not generalise to developer test suites. However, 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 two subject sets, one with large programs and real faults, and one with small methods that we manually analyse, i.e., to identify equivalent mutants and compose mutation adequate test suites. Thus, one may come to different conclusions in case it applies manual analysis on the programs with real faults. However, our results are consistent between the two subject sets and thus, we believe that this threat is not of actual importance. We also used automatically generated test suites in combination to the mutation testing tools. 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 (Chekam et al. 2017), 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 a sample 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 and used the automated generation tools, EvoSuite twice, and Randoop only once. However, our study focuses on investigating the effectiveness of the studied tools when used as a means to generate strong tests (Papadakis and Malevris 2010a), which are the results of both test generation tools combined. To cater for wider scrutiny, we made publicly available all the data of this study (Kintis et al. 2017b, c).

Construct Validity Construct validity pertains to the appropriateness of the measures utilised in our experiments. For the effectiveness comparison, we used the fault revelation (using real faults), mutation score and disjoint mutation score measurements. These are well-established measures in mutation testing literature (Papadakis et al. 2017). All of our results (both the ones of fault revelation and mutant killability) are under-approximated by the employed test suites, therefore, they may differ under different and potentially stronger test suites. 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.

6 Related Work

Mutation testing is a well-studied technique with a rich history of publications, as recorded in the recent survey of Papadakis et al. (2017) which summarises the advances in the area from 2008 to 2017, extending the previous surveys of Offutt (2011) and Yia and Harman (Jia and Harman 2011).

The original suggestion of mutation was a method to help programmers generate effective test cases (DeMillo et al. 1978). Since then, researchers have used it to support various other software engineering tasks (Offutt 2011). In particular, mutation analysis has been employed in: test generation (Papadakis and Malevris 2010a), test oracle selection and assessment (Fraser and Zeller 2012), debugging (Papadakis and Traon 2015), test assessment (Papadakis et al. 2016) and in regression testing (Zhang et al. 2012). It has also been applied to artefacts other than source code, such as models (Devroey et al. 2016) and software product lines configurations (Henard et al. 2014).

The main problems of mutation testing are the large number of mutants and the so-called equivalent mutant problem (Papadakis et al. 2015; Kintis 2016). 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) (Papadakis and Malevris 2010b), e.g., sampling ratios of 10 to 60% have a loss on fault detection from 26 to 6%. Selective mutation (Offutt et al. 1996) is another form of mutant reduction that only applies specific types of mutants. However, recent research has shown that there are no significant differences between selective mutation and random sampling (Zhang et al. 2010; Kurtz et al. 2016). To deal with the equivalent mutant problem researchers have adopted compiler optimisations (Papadakis et al. 2015; Kintis et al. 2017a), constraint based techniques (Offutt and Pan 1997) and verification techniques (Bardin et al. 2015). However, despite the efforts this problem remains open especially in 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 help the testing process, whilst at the same time they introduce noise to the mutation score measurement. Papadakis et al. (2016) 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. (2010) with the notion of disjoint mutants. Later Ammann et al. (2014) 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) (Papadakis et al. 2015; Kintis et al. 2017a). TCE is based on compiler optimisations and identifies duplicated mutants (mutants that are mutually equivalent but differ from the original program). According to the studies of Papadakis et al. (2015) and Kintis et al. (2017a)

a considerable number of mutants are duplicated and can be easily removed based on compiler optimisations. All these studies identified the problems caused by redundant mutants but none of them studied the particular weaknesses of modern mutation testing tools as we do here. Additionally, to deal with 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. (2014) analysed 4,181 mutants to provide insights into the nature of equivalent and stubborn mutants. Nan et al. (Li et al. 2009) 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. (2013) analysed 5,807 mutants generated by MUJAVA to investigate the effectiveness of the SDL mutation operator. Papadakis et al. (2014) 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 (Offutt et al. 1996).

Previous work on the differences of mutation testing frameworks for Java is due to Delahaye and Du Bousquet (2015). 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. (2015). This study compared several Java mutation testing tools when executed against a common test suite generated based on boundary value analysis and equivalence partitioning. 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. This result indicates that the mutants of PIT are easier-to-kill than the ones of MUJAVA which is in accordance to the results presented in our study.

Gopinath et al. (2016) 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 mutant-killability effectiveness based on the results of complete manual analysis. 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.

7 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 considerably better than the other studied tools, namely MUJAVA, MAJOR and PIT. At the same time our results showed that none of the tools always subsumes the others (PIT_{RV} reveals 5.6% unique faults, whilst MAJOR reveals 1.6%). Based on this finding, we identified weaknesses of the tools and made actionable recommendations on how to strengthen and improve their mutation testing 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.

Acknowledgements Marinos Kintis and Nicos Malevris are partly supported by the Research Centre of Athens University of Economics and Business (RC AUEB).

Appendix A

Table 10 MAJOR's, PIT_{RV}, PIT and MUJAVA fault revelation on real faults studied

Project-BugID	MAJOR		PIT_{RV}		PIT		MUJAVA	
	Runs?	Reveals?	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 10 (continued)

Project-BugID	MAJOR		PIT_{RV}		PIT		MUJAVA	
	Runs?	Reveals?	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	✓	✓	✓	✓	✓	✓		

Table 10 (continued)

Project-BugID	MAJOR		PIT_{RV}		PIT		MUJAVA	
	Runs?	Reveals?	Runs?	Reveals?	Runs?	Reveals?	Runs?	Reveals?
Math-6	✓		✓	✓	✓	✓	✓	
Math-8	✓	✓	✓	✓	✓	✓		
Math-10	✓	✓	✓	✓	✓	✓		
Math-11	✓	✓	✓	✓	✓	✓	✓	✓
Math-14	✓	✓	✓	✓	✓	✓	✓	✓
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	✓	✓	✓	✓	✓	✓		

Table 10 (continued)

Project-BugID	MAJOR		PIT_{RV}		PIT		MUJAVA	
	Runs?	Reveals?	Runs?	Reveals?	Runs?	Reveals?	Runs?	Reveals?
Math-95	✓	✓	✓	✓	✓	✓	✓	✓
Math-97	✓	✓	✓	✓	✓	✓		
Math-98	✓	✓	✓	✓	✓	✓	✓	✓
Math-99	✓	✓	✓	✓	✓	✓		
Math-101	✓	✓	✓	✓	✓	✓	✓	✓
Math-102	✓	✓	✓	✓	✓	✓	✓	
Math-103	✓	✓	✓	✓	✓	✓	✓	
Math-104	✓	✓	✓	✓	✓	✓	✓	
Math-105	✓		✓	✓	✓	✓	✓	
Closure-7	✓	✓	✓	✓	✓			
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	122	125	115	66	34

References

Ammann P, Offutt J (2008) Introduction to software testing, 1st edn. Cambridge University Press, New York

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. https://doi.org/10.1109/ICST.2014.13

Andrews J, Briand L, Labiche Y, Namin A (2006) Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans Softw Eng 32(8):608–624. https://doi.org/10.1109/TSE.2006.83

Baker R, Habli I (2013) An empirical evaluation of mutation testing for improving the test quality of safety-critical software. IEEE Trans Softw Eng 39(6):787–805. https://doi.org/10.1109/TSE.2012.56

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, https://doi.org/10.1109/ICST.2015.7102607

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

Chekam TT, Papadakis M, Traon YL, Harman M (2017) An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proceedings of the 39th international conference on software engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017, pp 597–608. https://doi.org/10.1109/ICSE.2017.61

Coles H (2010) The PIT mutation testing tool. http://pitest.org/, Last accessed October 2017

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

DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: help for the practicing programmer. IEEE Computer 11(4):34–41. https://doi.org/10.1109/C-M.1978.218136

Deng L, Offutt J, Li N (2013) Empirical evaluation of the statement deletion mutation operator. In: IEEE sixth international conference on software testing, verification and validation, pp 84–93. https://doi.org/10.1109/ICST.2013.20

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. https://doi.org/10.1145/2884781.2884821

Fagerland MW, Lydersen S, Laake P (2014) Recommended tests and confidence intervals for paired binomial proportions. Stat Med 33(16):2850–2875. https://doi.org/10.1002/sim.6148

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. https://doi.org/10.1145/2025113.2025179

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

Geist R, Offutt A, Harris JFC (1992) Estimation and enhancement of real-time software reliability through mutation analysis. IEEE Trans Comput 41(5):550–558. https://doi.org/10.1109/12.142681

Gopinath R, Ahmed I, Alipour MA, Jensen C, Groce A (2016) Does choice of mutation tool matter? Softw Qual J 1–50. https://doi.org/10.1007/s11219-016-9317-7

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. https://doi.org/10.1007/978-3-319-09940-8_7

Coles H, Laurent T, Henard C, Papadakis M, Ventresque A (2016) PIT: a practical mutation testing tool for Java (demo). In: ISSTA, pp 449–452. https://doi.org/10.1145/2931037.2948707

Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng 37(5):649–678. https://doi.org/10.1109/TSE.2010.62

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. https://doi.org/10.1109/ASE.2011.6100138

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. https://doi.org/10.1145/2610384.2628055

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

Kintis M, Malevris N (2015) MEDIC: a static analysis framework for equivalent mutant identification. Inf Softw Technol 68:1–17. https://doi.org/10.1016/j.infsof.2015.07.009

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. https://doi.org/10.1109/APSEC.2010.42

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. https://doi.org/10.1002/stvr.1529

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

Kintis M, Papadakis M, Jia Y, Malevris N, Traon YL, Harman M (2017a) Detecting trivial mutant equivalences via compiler optimisations. IEEE Trans Softw Eng PP(99):1–1. https://doi.org/10.1109/TSE.2017.2684805

Kintis M, Papadakis M, Papadopoulos A, Valvis E, Malevris N, Traon YL (2017b) Accompanying data for the paper: how effective mutation testing tools are? An empirical analysis of Java mutation testing tools with manual analysis and real faults. https://doi.org/10.6084/m9.figshare.5558587.v1

Kintis M, Papadakis M, Papadopoulos A, Valvis E, Malevris N, Traon YL (2017c) 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. http://pages.cs.aueb.gr/~kintism/papers/emse2017/

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. https://doi.org/10.1145/2950290.2950322

Laurent T, Papadakis M, Kintis M, Henard C, Traon YL, Ventresque A (2017a) Assessing and improving the mutation testing practice of pit. In: IEEE international conference on software testing, verification and validation, ICST

Laurent T, Ventresque A, Papadakis M, Henard C, Traon Y (2017b) PIT_{RV}: the extended version of PIT. https://github.com/laurenttho3/extendedpitest, Last Accessed October 2017

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: International conference on software testing, verification and validation workshops, pp 220–229. https://doi.org/10.1109/ICSTW.2009.30

Lindström B, Márki A (2016) On strong mutation and subsuming mutants. In: Proceedings of the 11th international workshop on mutation analysis

Luo Q, Hariri F, Eloussi L, Marinov D (2014) An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, ACM, New York, NY, USA, FSE 2014, pp 643–653. https://doi.org/10.1145/2635868.2635920

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

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

Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C (1996) An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Methodol 5(2):99–118

Offutt J (2011) A mutation carol: past, present and future. Inf Softw Technol 53(10):1098–1107. https://doi.org/10.1016/j.infsof.2011.03.007

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. https://doi.org/10.1145/1297846.1297902

Papadakis M, Malevris N (2010a) Automatic mutation test case generation via dynamic symbolic execution. In: 21st international symposium on software reliability engineering, pp 121–130. https://doi.org/10.1109/ISSRE.2010.38

Papadakis M, Malevris N (2010b) 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. https://doi.org/10.1109/ICSTW.2010.50

Papadakis M, Traon YL (2015) Metallaxis-fl: mutation-based fault localization. Softw Test Verif Reliab 25(5-7):605–628. https://doi.org/10.1002/stvr.1509

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

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. https://doi.org/10.1109/ICSE.2015.103

Papadakis M, Henard C, Harman M, Jia Y, Le Traon Y (2016) Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th international symposium on software testing and analysis, ACM, New York, NY, USA, ISSTA 2016, pp 354–365. https://doi.org/10.1145/2931037.2931040

Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M (2017) Mutation testing advances: an analysis and survey. Advances in Computers

Rani S, Suri B, Khatri SK (2015) Experimental comparison of automated mutation testing tools for Java. In: 4th international conference on reliability, infocom technologies and optimization, pp 1–6. https://doi.org/10.1109/ICRITO.2015.7359265

Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A (2015) Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (t). In: 2015 30th IEEE/ACM international conference on automated software engineering (ASE), pp 201–211. https://doi.org/10.1109/ASE.2015.86

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. https://doi.org/10.1145/2970276.2970345

Yao X, Harman M, Jia Y (2014) A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proceedings of the 36th international conference on software engineering, pp 919–930. https://doi.org/10.1145/2568225.2568265

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. https://doi.org/10.1145/1806799.1806863

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. https://doi.org/10.1145/2338965.2336793

Marinos Kintis is a research associate at the Interdisciplinary Centre for Security, Reliability and Trust at the University of Luxembourg. He received the PhD degree from the Department of Informatics of the Athens University of Economics and Business in 2016. The main topic of his dissertation was the introduction of effective techniques to ameliorate the adverse effects of the Equivalent Mutant Problem when testing software with Mutation. His main research interests include software and security testing and program analysis. He was awarded a Best Paper Award at the 16th International Working Conference on Source Code Analysis and Manipulation (SCAM 2016) and is one of the organisers of the 13th International Workshop on Mutation Analysis (MUTATION 2018).

Mike Papadakis is a research scientist at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received a Ph.D. diploma in Computer Science from the Athens University of Economics and Business. His research interests include software testing, static analysis, prediction modelling, mutation analysis and search-based software engineering.

Andreas Papadopoulos is a last-year undergraduate student in the Department of Informatics at the Athens University of Economics and Business (AUEB). His interests include Software Testing and Program Analysis, a subject in which he has participated in two published papers. He is currently working as a software developer in the financial sector.

Evangelos Valvis is an undergraduate computer science student, currently in his last year of studies at the Department of Informatics of the Athens University of Economics and Business. He has been a co-author in three publications so far, one of them received a Best Paper Award at the 16th International Working Conference on Source Code Analysis and Manipulation (SCAM 2016). His research interests include but are not limited to: software testing and security.

Nicos Malevris is a Professor in the Department of Informatics at the Athens University of Economics and Business (AUEB). He obtained his PhD from the University of Liverpool, UK, where he also served as a member of staff. He holds a MSc degree in Operational Research from the University of Southampton, UK and a BSc degree in Mathematics from the University of Athens, Greece. He has been with the Department of Informatics at AUEB, since 1991. His research interests include software quality assurance and in particular software testing and software reliability. He has gained experience in that area for more than 25 years, having been involved in research projects and having published a significant number of papers in journals and conferences. He serves at the Editorial Board of high quality International journals and has also been on the program committees of international conferences.

Yves Le Traon is professor at University of Luxembourg where he leads the SERVAL (SEcurity, Reasoning and VALidation) research team. His research interests within the group include (1) innovative testing and debugging techniques, (2) Android apps security and reliability using static code analysis, machine learning techniques and, (3) model-driven engineering with a focus on IoT and CPS. His reputation in the domain of software testing is acknowledged by the community. He has been General Chair of major conferences in the domain, such as the 2013 IEEE International Conference on Software Testing, Verification and Validation (ICST), and Program Chair of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). He serves at the editorial boards of several, internationally-known journals (STVR, SoSym, IEEE Transactions on Reliability) and is author of more than 140 publications in international peer-reviewed conferences and journals.