Received: 8 October 2021 | Revised: 17 January 2022 | Accepted: 18 March 2022

DOI: 10.1002/smr.2450

SPECIAL ISSUE - EMPIRICAL PAPER

Journal of Software: Evolution and Process WILEY

Towards practical application of mutation testing in industry — Traditional versus extreme mutation testing

Maik Betka | Stefan Wagner

Institute of Software Engineering, University of Stuttgart, Stuttgart, Germany

Correspondence
Maik Betka, Universitaetsstr. 38, 70569 Stuttgart, Germany.
Email: maik.betka@iste.uni-stuttgart.de

Funding information
Advantest

Abstract

Mutation testing is a technique that changes code instructions to assess the quality of automated software tests. Industry has not broadly adopted the technique because execution and analysis times are too long and not considered worth the effort. To change this, a variation called "extreme mutation testing" emerged, which mutates whole methods instead of instructions. The extreme variant trades accuracy for speed gains and also provides pre-analyzed results. In this study, we aim to analyze both techniques on their granularity levels, look for benefits when combining them, and find motivations when a developer considers killing mutants. For that, we conducted a case study in a company from the semiconductor industry. We mutated a large Java software project which is tested by more than 11,000 unit tests, analyzed the results, manually inspected more than 1000 mutants, and conducted a focus group with five developers of the software. Among other results, we provide the distribution of traditional across extreme mutants as well as qualitative coding results of our mutant inspection and focus group transcript. We conclude that the traditional approach can be similarly strategically applied as the extreme one and that motivations of developers to target mutants are mostly not code related.

KEYWORDS
code coverage, java programming, mutation testing, software engineering, software quality

1 | INTRODUCTION

The state of practice to evaluate the effectiveness of a software test suite is code coverage. It can be used to improve an existing test suite by adding test cases that cover previously uncovered code. The main problem with this is that it only shows which parts of the code have been executed by test, not whether faults are missed.

Mutation testing is an approach that introduces small, controlled changes in the source code of the software under test to change its behavior. The types of changes are called *mutators*; the respective altered software versions are called *mutants*. Afterwards, the given test suite is rerun. If no test fails, this means that the test suite cannot detect the mutant. Thus, it is said that the mutant *survived*. Otherwise, it is said that the mutant is *killed*. It may happen that some introduced software changes produce syntactically equivalent mutants, that is, mutants that do not have a different behavior. These mutants are called *equivalent mutants*. The goal is to strengthen the test suite by enhancing or writing new tests to kill the surviving nonequivalent mutants. The ratio of killed mutants to all generated nonequivalent mutants is called the *mutation score*. Thus, the goal of mutation testing is often expressed by optimizing the mutation score towards one.[1,2] Hence, mutation testing is a precise method to improve

test suite effectiveness. Yet, despite being around for several decades and thoroughly studied in research, mutation testing is still not widely adopted in industry.[1,2]

Extreme mutation testing is a variation of mutation testing and was introduced by Niedermayr, Juergens and Wagner in 2016.[3] In contrast to traditional mutation testing, where mutations are usually performed on an instruction-level, extreme mutation testing performs these mutations on a method level. The goal is to find *pseudo-tested methods*, which are methods where their whole functionality can be removed and still no test will fail.[3] Pseudo-tested methods are surprisingly common. Niedermayr analyzed 19 open-source projects and found the median proportion of pseudo-tested methods to the total number of mutated methods to be 10.1%.[4]

To categorize a method as pseudo-tested, particular mutants have to survive. For methods with no return value (void methods), the mutator empties the body of the whole method. If the resulting mutant survives, the method is categorized as pseudo-tested. For methods with return values, often multiple mutants have to survive. For example, for a primitive data type like boolean, two mutants have to survive. One mutator replaces the method body with a single return-statement that returns true and one that only returns false. If both resulting mutants survive, the method is categorized as pseudo-tested. If only one survives, the method is categorized as *partially tested*. Analogously, other mutators that replace the whole method body with default return values for other data types like integers (e.g., 0 and 1) or strings (e.g., "" and "A") are chosen such that, if the mutants survive, the method can be categorized as pseudo-tested. Complex data types derived from classes, like objects, can be simply set to null. The choice of which default values are used depends on the mutation testing tool used.[3,4]

Compared with traditional mutation testing, extreme mutation testing has the advantage of generating far fewer mutants, as the number of methods is significantly lower than the number of instructions that can be mutated. This reduces both: runtime and time to analyze the mutants to eventually kill them.

1.1 | Problem statement

Vera-Pérez, Monperrus, and Baudry[5] implemented the mutation testing engine Descartes to incorporate extreme mutation testing in the popular open-source mutation testing tool PIT.[6] Vera-Pérez et al[7] then used this implementation in a large study to better understand pseudo-tested methods. They analyzed more than 28,000 methods of Java open-source code and found—as did Niedermayr, Juergens, and Wagner[3] previously—that there are pseudo-tested methods in all analyzed Java projects. In their manual analysis, they also found that pseudo-tested methods are actually significantly less tested and also that developers would not be willing to spend extra effort on improving the test suits for them. Therefore, at present, it is not clear whether extreme mutation testing and the detection of pseudo-tested methods are useful in practice.

1.2 | Research objective

Our overarching research objective is to find out how mutation testing must change to foster industry adoption. Its current implementation produces too many mutants and is too tedious to analyze. Extreme mutation testing is a promising candidate to solve these issues because it is faster, produces fewer mutants, and is easier to interpret. However, compared with its traditional counterpart, it is also more coarse grained and central questions remain unanswered:

- **RQ1**: Which issues can be detected by the traditional approach but not by the extreme approach?
- **RQ2**: How big are the time savings of execution and analysis time compared with the traditional approach, and where do they come from?
- **RQ3**: Which mutants are actually relevant for developers, that is, what motivates them to kill a mutant?

Thus, we want to analyze the time savings and the lost precision of the extreme compared with the traditional technique. We also want to find motivations of developers which mutants are relevant since this point in particular is vital for industry adoption. To answer the research questions, we conducted a case study using quantitative and qualitative research methods.

1.3 | Context

The case is a large Java software that has a test suite of more than 11,000 unit tests from a company in the semiconductor industry. We were able to use mutation testing tools on this software and conduct a focus group with developers of the system.

This article is an extension of a short paper[8] published at the 2021 IEEE/ACM International Conference on Automation of Software Test (AST) where we already compared the performance of traditional and extreme mutation testing and reported on preliminary interview results. In this paper, we add insights on where time savings come from, a detailed comparison of what issues are detected by which approach, and a comprehensive qualitative analysis of what is relevant for developers.

We follow the reporting guidelines by Runeson and Höst.[9]

2 | RELATED WORK

Traditional mutation testing was already introduced in the 1970s.[10–12] Since then, a lot of research in that field has been done and is still ongoing. Jia and Harman[1] summarized the research activities from 1977 to 2009 by analyzing more than 390 papers. More recently, Papadakis et al[2] analyzed 502 papers to summarize the work in that field from 2008 to 2017. Both surveys show that many research efforts focused on cost reduction techniques to make mutation testing more practical. In general, Offutt and Untch[13] divided these cost reduction techniques into three categories: "do fewer," "do faster," and "do smarter." "Do fewer" refers to reducing the total number of mutants by using, for example, fewer mutation operators[14] or random sampling[15] of mutants. "Do faster" refers to runtime optimizations like parallel test execution.[16] "Do smarter" refers to optimizing the mutant execution and validation procedure, like evaluating a mutant right after the mutated statements are executed instead of waiting until test execution finishes. This is commonly called "weak mutation".[17] Further advancements led to the creation of several mutation testing tools. The most frequently used tools for Java systems are muJava,[18] PIT,[6] Major,[19] and Javalanche.[20]

Nonetheless, major challenges in mutation testing remain unsolved. Equivalent and redundant mutants still account for a large proportion of generated mutants.[2] This causes issues when calculating the mutation score. Furthermore, despite the extensive work in that area, the number of mutants that have to be analyzed by developers still remains impractical.[8]

To address these issues, Niedermayr, Juergens, and Wagner introduced extreme mutation testing with the concept of pseudo-tested methods in 2016.[3] Niedermayr and Wagner[4,21] analyzed 19 open-source projects and found that pseudo-tested methods were present in all of them. The median of relative shares of pseudo-tested methods was 10%.

Vera-Pérez et al.[5] introduced the extreme mutation testing engine Descartes for PIT. In their tool demonstration, they compared execution time and mutation score between the default PIT engine, Gregor, and Descartes. The demonstration showed that extreme mutation testing executed noticeably faster than its traditional counterpart. Moreover, it showed that scores by the extreme approach tend to be higher compared with the traditional approach and that both mutation scores correlate.

Vera-Pérez et al.[7] also analyzed how many pseudo-tested methods developers consider relevant. Out of 101 pseudo-tested methods, less than 30% are considered relevant enough to be remediated by the developers. Furthermore, the same developers confirmed that pseudo-tested methods are easy to interpret. In contrast to these studies, we conducted the first study that thoroughly compares traditional with extreme mutation testing. We did provide not only the execution time but also a detailed distribution of which traditional mutants address the same methods as extreme mutation testing. Additionally, we also analyzed the limits of extreme mutation testing by inspecting mutants regarding their testing verdict. We also presented both approaches to developers to find what motivates them to kill mutants.

Similar to our study, studies by Petrović et al.[22,23] evaluated the industrial application of traditional mutation testing at Google. For that, mutation testing was integrated into the code review process. Developer feedback from each code review was used to filter out less important mutants and reduce the number of mutants for future reviews. Petrović et al. divided mutants in productive and unproductive mutants. Productive mutants are mutants that can be killed, and their analysis leads to writing an effective test, or they are equivalent mutants where their analysis leads to improving code quality. Unproductive mutants are all mutants that do not fulfill either criteria, which includes redundant or less important mutants. They conclude that only productive mutants are relevant enough to be killed, whereas the unproductive ones should be ignored. They also conclude that mutation score is too expensive and that it does not guide the developer to improve a test suite but to make mutation testing adequate. In fact, our study reaches very similar conclusions but differs by the applied research methods and results. We did not apply mutation testing in a code review process and collected developer feedback. Instead, we conducted a focus group where we asked the developers which mutants they consider important and why. In addition to that, we also provide the motivations when a developer considers a mutant important and not only that there are productive and unproductive mutants. Although it was feasible for us to calculate the mutation score, we agree that it is meaningless since some mutants, and many pseudo-tested methods, are ultimately not meant to be killed and many similar or duplicate mutants exist that inflate the score.

3 | CASE STUDY DESIGN

3.1 | Research questions

RQ1: Which issues can be detected by the traditional approach but not by the extreme approach?

Since it is hard to compare both techniques due to their different types of mutators, we would like to see which types of issues the extreme variant cannot find which the traditional variant can. For that, we look at the distribution of traditional mutants, inspect them, and analyze why extreme mutation testing cannot detect the issue.

RQ2: How big are the time savings of execution and analysis time compared with the traditional approach, and where do they come from?

Extreme mutation testing is faster but more coarse grained than its traditional counterpart because the technique removes or replaces whole method bodies. We are interested in how applying it before traditional mutation testing would reduce the number of traditional mutants that would not need to be analyzed any longer because of this "extreme" method substitution. For that, we analyze traditional mutants which are located inside pseudo-tested methods.

RQ3: Which mutants are actually relevant for developers, that is, what motivates them to kill a mutant?

Mutation testing creates many mutants, but not every mutant is equally relevant. The decision which mutants are ultimately more relevant than others is made by the developers that use the technique. Therefore, we conduct a focus group to ask many developers at once and find out what influences a developer's motivation to kill some mutants but others not. Developer feedback is vital to foster industry adoption, since developers should profit from using mutation testing and not see it as a burden. From our findings, we also draw conclusions about how mutation testing should be applied.

3.2 | Case and subjects selection

To be able to make a meaningful comparison, we aim for a case software system that is large enough that the execution times between the different mutation approaches will likely differ. Furthermore, as we want to understand the developer perspective on the mutation approaches, we need subjects with good testing experience and intimate knowledge of the system and its tests.

3.3 | Data collection procedure

As a first step, we ran traditional and extreme mutation testing tools against the case software system. For traditional mutation testing, we used PIT[*, 6] with its default mutation engine Gregor as a mutation testing tool (version 1.4.10). For extreme mutation testing, we used the same tool but with the Descartes engine[†, 5] (version 1.2.6). We configured to use the default mutators that are provided by both engines. In addition to that, we also measured code coverage with the Java code coverage library JaCoCo[‡] (version 0.8.6). Results from PIT were available in JSON and HTML form. JaCoCo generated HTML files to inspect code coverage.

To answer RQ1 and RQ2, we manually inspected traditional mutants to find what issues they reveal. We looked at killed mutants which, per definition of mutation testing, should be a proof of a strong test suite. Killed mutants were present in all, even pseudo-tested, methods which made us curious. We systematically analyzed more than 1100 mutants in small chunks with the support of a self-written Python script. For each chunk of mutants that we analyzed, we have chosen the extreme mutation testing verdict, the method's return category, the traditional mutator type (e.g., `NegateConditionalsMutator`), and its status (survived, killed).

We provided this selection as arguments to our Python script in addition with a maximum chunk size of 50, which means that we have at most analyzed 50 mutants per selection. We programmed the Python script to randomly choose a mutant and open the JaCoCo code coverage report in a web browser. At the terminal window, the script displayed the mutator information that precisely described what was mutated. In addition to that, the method name with its internal description and the line number where the mutant was placed were also displayed. We used the line number to jump directly with the search function of the browser to the line where the mutant was placed.

To address RQ3, we ran a focus group with the developers of the software. We wanted to discuss the intentions and opinions of developers when a mutant is considered to be worth to be fixed or not. The duration of the focus group was set to about 60 min. To select a subset of methods and mutants that can be discussed in that time frame, we first had one meeting in advance with the lead developer of the software. We asked which packages all developers should be familiar with so that they can talk about them in the focus group. In our last study,[8] we noticed that some code areas were developed by a different development group, which made it necessary to ask the lead developer which packages the team members that participate in the focus group actually developed. From 97 packages, the lead developer provided us with a list of 19 packages where we have chosen mutants from. We then selected 15 pseudo-tested methods from three different classes by choosing methods that have different verbs. For example, if we had to select two methods from methods that are named `getName`, `getId`, and `setTimeout`, we would have selected one that starts with the verb "get" and the one that starts with the verb "set." Thus, we aimed for a selection of methods that behave differently from each other. When we had to select between methods that contained the same verb, we have chosen one that has not too many

lines so that it would fit on a presentation slide. Other than that, we selected one randomly if there were still more methods to select from. For traditional mutation testing, we selected traditional mutants that are placed on tested methods. We selected tested methods by the same criteria that we used for pseudo-tested methods. In addition to that, we also looked that we covered each type of mutator at least once and as balanced as possible. In total, we have selected six tested methods from four classes, where 19 traditional mutants survived.

Five developers participated in the focus group, which was held virtually in a video conference. All of them were familiar with the code and the concept of mutation testing. We recorded the audio of the focus group with the consent of the developers to transcribe it and deleted it afterwards. Half of the time was used to discuss pseudo-tested methods and the other half to discuss traditional mutants. We displayed each method and mutant together with the coverage report on presentation slides. The following questions were used to guide the discussion for each method or mutant:

- What does this method do, and can you summarize it?
- Which issue does this mutant reveal?
- Would you fix the issue?
- Why would you fix it or not, and how would you do it?
- Can you describe the priority when you would fix it?

In total, we discussed nine pseudo-tested methods from three different classes and six tested methods from four different classes within the given time constraint of 60 min in the focus group. Additionally, we discussed 15 traditional mutants that were placed on six tested methods.

3.4 | Analysis procedure

After performing mutation tests with both engines and measuring the code coverage, we consolidated the results by writing several Python scripts. To answer RQ1, we have written two Python scripts. The first script generated a CSV file that aggregated the number of methods, by their extreme mutation testing verdict (pseudo-tested, partially tested, tested) and their return category (boolean, numeric, string, array, object, void). We fully derived the data by analyzing the JSON files that were generated by PIT. We assigned any return type that represents a number to the "numeric" category. This includes integer types like `long`, `integer`, `short`, and `byte` but also floating-point types like `double` and `float`. The boolean and numeric categories include the primitive types as well as the object version of these primitive types. Note that we categorized strings and also the `char` type into the "strings" category.

To answer RQ1 and RQ2, we merged both JSON files that are generated by both mutation engines into one common JSON file for further analysis. For that, we used a combination of the package name, the class name, the method name, and Java's internal method descriptors as a key in a JSON dictionary, to uniquely identify each method, and we used the respective data about the mutators of both techniques as values. The method descriptor consists of the ordered data types of its arguments and the return data type. It is defined in the Java Virtual Machine Specification.[§] With that structure, we matched the respective traditional mutants to the respective extreme mutants and were able to extract information on a method level. In addition to that, we observed and analyzed deviations in the addressed methods. This means, we observed methods that were targeted only by one of the mutation testing techniques. Hence, we did a manual inspection of a subset of these mutants, their tests, and the code coverage reports verifying correct execution of the mutation engines and to explain the deviations. We used the merged JSON file in another Python script to generate an intermediary CSV file that represents the distribution of traditional mutants in comparison with extreme mutants. This CSV file was then parsed with a spreadsheet program to manually inspect the results and to decide how to perform manual inspection of mutants.

To analyze traditional mutants, we looked at the code coverage report in the browser and used the mutant information that was displayed by the console terminal. We manually derived what issues the mutant reveals and took free-form notes in the terminal window. The Python script added the notes to a new JSON field of the respective traditional mutant so that each note can be tracked. Thus, this additional JSON field was an update of the existing merged JSON file. Analysis was completely manual and static, which means that we did not rerun tests or stepped through a debugger due to time constraints.

Inspecting mutants this way has the advantages that we were able to analyze many mutants because:

- The process was mostly automated with the Python script by showing the mutant location and keeping track of the free-form notes.
- We had no major analysis context switches between the mutants because each chunk of mutants shared the same attributes like extreme mutation testing verdict, return category, status, or mutator type. This enabled us to quickly spot similarities between these mutants.
- Manual static analysis without rerunning tests and recompiling classes saves a lot of time and was in most cases sufficient to derive conclusions.

However, this approach has the disadvantages that:

- We were limited to analyze only within the scope of the respective class because we did not dynamically step through a debugger, and thus, other classes.
- We did not verify our observed conclusions with runtime information.
- We were unable to derive conclusions for mutants where runtime information is necessary.

We discuss some of these disadvantages as threats to validity in Section 4.5.

While analyzing mutants and taking free-form notes, we also developed guidelines for some types of traditional mutators. These guidelines contain some rules which data we were also interested to collect. We applied them when analyzing further mutants. These guidelines affect the following mutators:

- `ConditionalsBoundaryMutator` and `NegateConditionalsMutator`
  - Keep track on which type of statement (if-statement, return-statement, ...) the conditional was mutated.
  - Keep track of the number of conditionals that the boolean expression has in total.
  - Keep track whether changing the conditional also changes the control flow such that previously uncovered code is executed. Uncovered code is code that was not executed by any test.
- `VoidMethodCallMutator`
  - Extract the verb of the method. For instance, a method named `verifyResult()` contains the verb *verify*. Do not interpret what these methods do in particular and skip taking free-form notes.

Eventually, we parsed the notes into a CSV file, loaded them into a spreadsheet, and consolidated them by applying qualitative coding on them. This means that we ended up with a list of categorized reasons that describe why a mutant survived or was killed, with several examples per reason. Furthermore, we counted the number of examples per reason to also have quantitative data that we additionally used when deriving conclusions.

To answer RQ3, we transcribed the audio of the focus group. We qualitatively analyzed the transcript to identify factors that influence why a developer would kill certain mutants and others not. We used Straussian's grounded theory coding[24] which is exemplarily shown in Table 14. We started with open coding to generate categories that describe the discussion topics and anything that was related to either killing or ignoring the survived mutant or pseudo-tested method. Afterwards, we looked through all open codes and continued with axial coding. Axial codes describe relationships and their implications between the categories that were obtained through open coding. For that, we many times used if-then sentences to describe these relations and also used simple sentences when we described facts or implications. In the last step, we applied selective coding to put all statements from axial coding into central categories. We focused to choose central categories that are factors which influence a developer's decision to kill a mutant. We labeled statements obtained through axial coding that could not be put into a particular central category with "Additional findings."

3.5 | Validity procedure

We did not know in advance how many mutants can be analyzed without runtime data. To ensure that our results are significant, we chose to inspect a large number of mutants. Furthermore, we verified that our numbers for each individual traditional mutator are correct by calculating the sum of some metrics and check whether they add up correctly. Methods that were mutated by the extreme and not by the traditional approach were analyzed to verify that both mutation testing engines work as expected. In addition to that, we reviewed our qualitative analysis results of the focus group by another researcher that did not participate in the focus group.

4 | RESULTS AND DISCUSSION

4.1 | Case and subjects description

To address any of the previously mentioned research questions, we ran traditional and extreme mutation testing for a software project that is used in the semiconductor testing industry. The software implements a parser of a custom domain-specific language (DSL) together with an API, which both are used by customers to write extensive tests for semiconductor chips. Further, the software controls the hardware that directly interacts with the chip to, for example, apply or measure voltages. That hardware is commonly referred to as automatic test equipment (ATE). The

software is tested by more than 11,000 unit tests that call methods about the size of 12,500 lines of code. In addition to unit tests, the software is also subject to component, system-level, and integration tests, which were out of scope of mutation testing. The software project is written in Java and consists of multiple smaller projects of different age, authors, and coding styles.

4.2 | RQ1: Detected issues

To analyze differences of both mutation testing techniques that are caused by their granularity, that is, by either mutating on instruction or on method level, we looked at the distribution of traditional mutants across extreme mutants. Tables 1–3 show the distributions of mutation testing results by return category and mutator type. Each of these tables shows the results for one extreme mutation testing verdict, respectively. To

TABLE 1 Distribution of mutation testing results of pseudo-tested methods by return category and traditional mutator type

				Traditional mutator					
Return category	Metric	Total extreme	Total traditional	Con.[a]	Neg.[b]	Inc.[c]	Math.[d]	Ret.[e]	Void.[f]
Boolean	Survived	24	56	2	41	0	0	13	0
	Killed	0	2	0	2	0	0	0	0
	Mutants	24	58	2	43	0	0	13	0
	Methods	12	12	1	11	0	0	12	0
	Tests	516	663	74	329	0	0	260	0
Numeric	Survived	12	8	0	0	0	0	6	2
	Killed	0	0	0	0	0	0	0	0
	Mutants	12	8	0	0	0	0	6	2
	Methods	6	6	0	0	0	0	6	2
	Tests	482	278	0	0	0	0	241	37
String	Survived	18	22	5	7	0	5	5	0
	Killed	0	7	1	4	1	0	1	0
	Mutants	18	29	6	11	1	5	6	0
	Methods	6	6	2	4	1	3	6	0
	Tests	525	779	164	346	1	145	123	0
Array	Survived	2	1	0	0	0	0	1	0
	Killed	0	0	0	0	0	0	0	0
	Mutants	2	1	0	0	0	0	1	0
	Methods	1	1	0	0	0	0	1	0
	Tests	34	17	0	0	0	0	17	0
Object	Survived	149	173	2	13	0	1	150	7
	Killed	0	8	2	4	1	0	1	0
	Mutants	149	181	4	17	1	1	151	7
	Methods	149	149	1	9	1	1	149	5
	Tests	506,873	507,421	78	337	10	29	506,875	92
Void	Survived	141	203	6	71	0	4	na	122
	Killed	0	93	11	74	3	5	na	0
	Mutants	141	296	17	145	3	9	na	122
	Methods	141	141	12	91	3	5	na	79
	Tests	3,108	4,882	208	1,784	32	566	na	2,292

[a]ConditionalsBoundaryMutator.
[b]NegateConditionalsMutator.
[c]IncrementsMutator.
[d]MathMutator.
[e]ReturnValuesMutator.
[f]VoidMethodCallMutator.

TABLE 2 Distribution of mutation testing results of partially tested methods by return category and traditional mutator type

				Traditional mutator					
Return category	Metric	Total extreme	Total traditional	Con.[a]	Neg.[b]	Inc.[c]	Math.[d]	Ret.[e]	Void.[f]
Boolean	Survived	34	35	6	14	1	1	11	2
	Killed	37	137	6	77	8	6	40	0
	Mutants	71	172	12	91	9	7	51	2
	Methods	34	34	4	27	2	2	34	2
	Tests	1,359	1,812	169	680	84	49	761	69
Numeric	Survived	4	2	0	0	0	0	1	1
	Killed	4	4	0	0	0	1	3	0
	Mutants	8	6	0	0	0	1	4	1
	Methods	4	4	0	0	0	1	4	1
	Tests	527	237	0	0	0	1	235	1
String	Survived	6	0	0	0	0	0	0	0
	Killed	3	3	0	0	0	0	3	0
	Mutants	9	3	0	0	0	0	3	0
	Methods	3	3	0	0	0	0	3	0
	Tests	44	6	0	0	0	0	6	0
Array	Survived	0	0	0	0	0	0	0	0
	Killed	0	0	0	0	0	0	0	0
	Mutants	0	0	0	0	0	0	0	0
	Methods	0	0	0	0	0	0	0	0
	Tests	0	0	0	0	0	0	0	0

[a]ConditionalsBoundaryMutator.
[b]NegateConditionalsMutator.
[c]IncrementsMutator.
[d]MathMutator.
[e]ReturnValuesMutator.
[f]VoidMethodCallMutator.

quickly compare both mutation testing techniques, first, the total numbers for both techniques are listed next to the return category and metric columns. For detailed analysis, the rest of the following columns represent the numbers for traditional mutator types. The metrics "survived" and "killed" denote the number of mutants by their respective status. The values for the metrics "survived," "killed," "mutants," and "tests" from the traditional mutator type columns can be added up and can be directly read from the "Total Traditional" column. The number of methods from the same columns cannot be added up because there can be multiple mutants on the same method.

Methods of the void return category have the particularity that it is not possible to place `ReturnValuesMutators` on them because void methods do not have any return values. This is denoted with "na" entries in Tables 1 and 3. Furthermore, since methods in the return categories "object" and "void" have only one extreme mutator, respectively, they cannot be partially tested and the lines for these return categories were removed in Table 2. Section 4.3 describes the class that is responsible for the majority of executed tests that we considered as an outlier in our data. The return category " object* " in Table 3 shows the results for mutation testing without that class. This means that only numbers for "tested" methods that have an object return type are affected by excluding that class. The rest of the results from Tables 1–3 remain the same, whether the class is excluded or not.

In total, 2083 methods were mutated by the extreme variant that also contained at least one traditional mutant. We had to remove 10 methods from this set because these methods had extreme mutants that finished with the PIT status of `NON_VIABLE` or `TIMED_OUT`. The traditional variant additionally created mutants for 92 constructors which are, by design, not mutated by the extreme variant. The extreme variant additionally created mutants for 90 methods, where no traditional mutant was created. These were void methods that were mostly simple setter that set fields of classes with assignments or called other methods that may have return values, but these are ignored or assigned to a variable. The traditional variant does not create any mutants for these methods because none of the traditional mutators can be applied to them. The `ReturnValuesMutator` cannot be applied because these methods have no return-statement. The `VoidMethodCallMutator` is not applied because they do not call void methods. None of the remaining mutators can be applied because these methods also did not contain any

TABLE 3 Distribution of mutation testing results of tested methods by return category and traditional mutator type

				Traditional mutator					
Return category	Metric	Total extreme	Total traditional	Con.[a]	Neg.[b]	Inc.[c]	Math.[d]	Ret.[e]	Void.[f]
Boolean	Survived	0	58	18	25	0	2	5	8
	Killed	262	539	23	222	5	31	237	21
	Mutants	262	597	41	247	5	33	242	29
	Methods	125	125	33	94	5	18	125	11
	Tests	1144	18,783	16,799	866	5	148	846	119
Numeric	Survived	0	44	15	3	0	11	5	10
	Killed	218	280	10	69	2	71	121	7
	Mutants	218	324	25	72	2	82	126	17
	Methods	103	103	19	40	2	48	102	14
	Tests	568	1702	358	344	2	262	595	141
String	Survived	0	11	5	1	0	2	0	3
	Killed	138	127	3	49	2	8	57	8
	Mutants	138	138	8	50	2	10	57	11
	Methods	46	46	7	21	2	7	46	7
	Tests	734	498	122	126	4	19	135	92
Array	Survived	0	15	3	2	1	0	0	9
	Killed	232	339	39	123	11	29	123	14
	Mutants	232	354	42	125	12	29	123	23
	Methods	116	116	39	56	10	17	116	18
	Tests	492	1211	172	332	59	124	342	182
Object	Survived	0	1109	121	149	0	115	584	140
	Killed	869	2838	91	1402	25	151	957	212
	Mutants	869	3947	212	1551	25	266	1541	352
	Methods	869	869	103	345	22	112	847	213
	Tests	54,808	3,535,417	298,856	813,454	81	375,418	2,018,678	28,930
Object[g]	Survived	0	303	45	28	0	18	72	140
	Killed	828	2297	91	861	25	151	957	212
	Mutants	828	2600	136	889	25	169	1029	352
	Methods	828	828	84	304	22	93	806	213
	Tests	23,600	81,421	4964	8353	81	319	38,774	28,930
Void	Survived	0	333	26	39	1	11	na	256
	Killed	458	1184	44	554	16	57	na	513
	Mutants	458	1517	70	593	17	68	na	769
	Methods	458	458	53	248	17	30	na	367
	Tests	2700	16,272	1023	3781	48	494	na	10,926

[a]ConditionalsBoundaryMutator.
[b]NegateConditionalsMutator.
[c]IncrementsMutator.
[d]MathMutator.
[e]ReturnValuesMutator.
[f]VoidMethodCallMutator.
[g]Without the class that is responsible for the majority of executed tests.

mathematical expressions or conditionals. This means that Tables 1–3 show the results for 2073 methods. When excluding the class that is responsible for the majority of executed tests, the number of methods reduces to 2032.

We statically analyzed in total 1185 traditional mutants, where 737 mutants survived and 448 were killed. Tables 4, 6, 8, 9, and 10 show the summarized results of our free-form notes for each inspected mutant. These tables are split into two columns. The left column lists explanations

TABLE 4 Analysis why mutants of the ConditionalsBoundaryMutator survived or were killed

Explanation why mutants survived with the number of observations

Missing boundary check

(42) Changing the conditional boundary in an if-statement led to executing the same code and statements because the conditional boundary value was never used by any of the tests.

(9) Changing the conditional boundary in an if-statement. The if-statement checks whether a provided index is within array boundaries. There are no test cases that use an index that uses the array boundary values.

(6) Changing the conditional boundary in an if-statement. The if-statement checks whether a timeout value is a positive integer and sets it. Otherwise, a default value for the timeout is set. There is no test that validates that if a timeout is set to zero, the correct default timeout is used.

(6) Changing the conditional boundary in the header of a for-loop that checks for the upper iteration limit. The upper iteration limit of the loop is never reached because a return-statement inside it is executed before that. There is no test that fully iterates the loop.

(3) Changing the conditional boundary in the header of a for-loop led to an additional call of `StringBuilder.append(String s)` on a class field object.

(2) Changing the conditional boundary in an if-statement that checks whether the number of received bytes from a network connection is less or equal to zero to throw an exception if that is the case. There is no test that provides an empty response and verifies that behavior.

(2) Changing the conditional boundary in the header of a while-loop. The header checks whether the number of received bytes is greater or equal to the number of bytes read from a byte buffer. There is no test that returns a number of zero bytes read.

(1) Changing the conditional in an if-statement that checks whether a log-level is exceeded to log a message. There is no test that executes it with that exact threshold.

(1) Changing the conditional boundary in an if-statement that is located inside a loop. The loop iterates over each character of the string. The if-statement checks whether the string contains only bytes, that is, characters as numbers from 0 to 254. There is no test that checks a string that contains a character as the byte number 255.

(1) Changing the conditional boundary in an if-statement. The if-statement checks whether the byte array does not exceed a particular length. There is no test case where this is the case.

(1) Changing the conditional boundary in the header of a for-loop that checks for the upper iteration limit. There are more elements added to a list than before because the loop now iterates an additional time. There is no test case that validates the number of elements in the list.

(1) Changing the conditional boundary in an if-statement. The method counts recursively the number of nodes in an acyclic graph up to a maximum depth. The if-statement is used to check whether that depth is reached. There is no test that uses a graph that exceeds the maximum depth and does not validate the boundary.

Changed return value not detected

(36) Changing the conditional boundary in a return-statement or in the header of a for-loop. The mutators were placed on a class that is responsible for the majority of executed tests. The class implements various Java operators that can be used in an API when using a domain-specific language. The changes on the return-statements are not detected. The changes inside the header of the for-loops led to different return values which were mostly empty collections. These changed return values were also not detected.

Explanation why mutants survived with the number of observations

Uncaught IndexOutOfBoundsException

(35) Changed the conditional in the header of a for-loop that checks for the upper iteration limit. This leads to an `IndexOutOfBoundsException` when accessing an array within the for-loop with the count-variable (e.g., i).

(1) Changed conditional in an if-statement that implements a boundary check that verifies that a variable is within array-boundaries. Changing the conditional led to executing code that explicitly throws an `IndexOutOfBoundsException`.

Uncaught IllegalArgumentException

(1) Changed conditional in an if-statement that implements a boundary check that verifies that a variable is within array-boundaries. Changing the conditional led to executing code that explicitly throws an `IllegalArgumentException`.

Uncaught custom-implemented runtime exception

(7) Changed conditional in an if-statement led to throwing a custom-implemented runtime exception.

(3) Changed conditional in an if-statement that implements a boundary check that verifies that a variable is within array-boundaries. Changing the conditional led to executing code that explicitly throws a custom-implemented runtime exception.

Changed return value detected

(8) Changing the conditional boundary in an if-statement led to returning a different return value which is detected elsewhere, that is, outside the method.

Unknown why the mutant was killed

(14) Conclusions could not be derived because of missing runtime information.

(4) No source code was available to be analyzed. This happened because the Java reflection API was used and code from another software project was mutated that is used for performance tracing and instrumentation.

TABLE 4 (Continued)

Explanation why mutants survived with the number of observations	Explanation why mutants survived with the number of observations
Equivalent mutants	
(8) Changing the conditional boundary in a header of a for-loop that checks for the upper iteration limit. The method checks if two collections are the same. When increasing the loop limit, the collections will return each a null value which are then compared with each other. The method that checks for comparison can tolerate that. Thus, the change of the boundary will cause an additional comparison on null values but will return the same boolean, namely, only true if all values are the same.	
(6) Changing the conditional boundary in an if-statement. The if-statement checks if an integer is less or equal to one and returns a boolean if this is true. Subsequently, an array is iterated in a loop which uses the integer value to access elements of the array by subtracting one from it. Changing the conditional boundary in that if-statement does not change the returned value. The loop is either not entered because the integer is zero or returns the same result because of the called method inside the loop.	
(4) Changing the conditional boundary in an if-statement. The if-statement that compares two integers with `val1 < val2 ? val1 : val2` is equivalent to `val1 <= val2 ? val1 : val2` because the same value is returned when both integers are equal.	
(2) Changing the conditional boundary in an if-statement. The if-statement checks whether a floating-point number is less or equal to a second one. For that, it checks with a first condition whether the first number is less than the second one. With a second condition, the two floating-point numbers are checked whether they are equal with a method that accounts for imprecision when comparing floating-point numbers. Changing the less-than-operator to a less-equals-operator results in an equivalent mutant, as the same result is produced.	
(1) Changing the conditional boundary in an if-statement. The method sets the number of threads in a thread pool. The if-statement checks whether the provided variable is less or equal to a configured limit. The default size was always used. This produces a functional equivalent mutant because the number of threads does not influence the functionality but the performance of the software.	
(1) Changing the conditional boundary in an if-statement. The method obtains an integer value by executing `String.lastIndexOf('.')`. The value is then used to return a substring up to that last dot. The `lastIndexOf` method returns -1 if the string does not contain any dots. The if-statement checks whether the obtained value is less than zero and returns an empty string in that case. Changing the comparison to less-equal is returning also an empty string, as when the dot would be placed in the first position of the string.	
(1) Changing the conditional boundary in an if-statement. The if-statement executed `String.indexOf('/')` to check if it is less than zero. The method returns −1 if the string does not contain the character. Because of other checks that are executed before reaching that if-statement, the character at the zeroth position of the string can never contain the slash-character that the method is looking for.	
Unknown why the mutant survived	
(13) Conclusions could not be derived because of missing runtime information.	

why the mutants survived and the right one why they were killed. The bold headings describe the root cause why mutants survived or were killed. Below these headings, additional explanations and context information are listed where possible. Each list item is preceded with a number in parentheses. This number denotes the number of occurrences that we observed for a particular explanation.

Note that we merged the explanations regardless of the extreme mutation testing verdict because the explanations why a mutant was killed or survived did not fundamentally differ. However, it very often helped to analyze the mutants when knowing the extreme mutation testing verdict. For example, Table 10 shows the results for the `ReturnValuesMutator`. Although it is trivial to describe that the changed return value has not been detected by the test suite, it is possible to additionally conclude whether the method does something else like calling an important method or setting an important class field. Hence, we were able to conclude that in these situations, a method's return value may be optional, but the method cannot be removed because of its other effects. This is possible because extreme mutation testing removes the whole method body that includes all statements that cause these side-effects.

Table 11 shows the results for the `VoidMethodCallMutator`. This mutator only removes a call to a void-method, and this is either detected by the test suite or not. Thus, when we looked at mutants that are created by that mutator, we noticed that it would require knowledge about the rest of the code, including other methods, to derive meaningful context information. Hence, we did not take notes about the surrounding code and control flow but extracted the verbs of these removed void methods because verbs encode rich semantic information. The number of occurrences of the verbs of these methods is shown in Table 11 in a similar way as in the previously mentioned tables. These results are used to spot whether some void methods can be more often removed than others without being detected by the test suite.

Tables 5 and 7 additionally show in which context the mutated conditionals of the `ConditionalsBoundaryMutator` and `NegateConditionalsMutator` were observed. These tables list whether the mutant survived or were killed. Also, these tables show whether the mutated conditional was placed on an if-, for-, while-, or return-statement, or whether it was used in an assignment. The "Conditionals" column shows the number of conditionals of the mutated boolean expression. For clarity, logical AND- (&&) or OR- (||) operators are not counted as conditionals. Furthermore, PIT treats single boolean values as expressions with one conditional like `(a == true)` and mutates these expressions. For example, we would count a boolean expression like `(a=` b && c || c < d), as three conditionals as well as the expression `(a == b == c=` d). We have collapsed the numbers for each type of statement to be either exactly one or greater or equal than two. We did that because the majority of expressions contained either one or two conditionals and only in 10 cases more than two conditionals were observed. The right-most parts of Tables 5 and 7 additionally show whether mutating the conditional leads to uncovered code, that is, code that is not executed by the test suite.

TABLE 5 Inspected number of mutants by their status, statement type, and information whether they executed uncovered code for the ConditionalsBoundaryMutator

				Executed uncovered code		
Status	**Statement type**	**Conditionals**	**Total**	**Yes**	**No**	**NA**
Survived	If	≥ 1	46	3	37	6
		≥ 2	25	1	19	5
	For	≥ 1	15	0	13	2
		≥ 2	0	0	0	0
	While	≥ 1	5	0	4	1
		≥ 2	3	0	3	0
	Return	≥ 1	40	na	na	40
		≥ 2	0	na	na	0
	Assignment	≥ 1	1	0	1	0
		≥ 2	0	0	0	0
Killed	If	≥ 1	10	2	7	1
		≥ 2	13	6	3	4
	For	≥ 1	35	0	34	1
		≥ 2	0	0	0	0
	While	≥ 1	2	0	2	0
		≥ 2	2	0	2	0
	Return	≥ 1	4	na	na	4
		≥ 2	0	na	na	0
	Assignment	≥ 1	3	0	3	0
		≥ 2	0	0	0	0

TABLE 6 Analysis why mutants of the NegateConditionalsMutator survived or were killed

Explanation why mutants survived with the number of observations	Explanation why mutants were killed with the number of observations
Same return value is returned	**Changed return value detected**
(16) Negated the condition in an if-statement. The method compares whether two collections are the same. The collections are stored together with their properties as class fields. The if statement checks for some of these properties of the compared collections to avoid iterating each element in both collections, and checks them for equality. Negating the conditional led to avoiding this performance improvement and comparing each element, which still resulted in the same return value.	(39) Negated the condition in an if-statement which led to returning a different return value that is detected elsewhere, i.e., outside the method.
(14) Negating the condition of an if-statement changed the control flow, but the same return value was returned. Nothing else besides the return-value was changed.	**Incorrectly set class field**
Changed return value is not detected	(1) Negating the condition of an if-statement led to setting a class field with a different value which is detected elsewhere, i.e., outside the method.
(44) Negated the condition in an if-statement which resulted in returning a different return value. Nothing else besides the return-value was changed. For example, the changed conditionals led to additional or missing calls of methods of a String Builder. This impacted descriptions, messages and file-paths. Note that also other return types were affected.	(1) Negating the condition of an if-statement caused that a previously created object was not added to a list which is a class eld. This is detected elsewhere, i.e., outside the method.
(40) Negating the conditional in a return-statement or in the header of a for-loop. The mutators were placed on a class that is responsible for the majority of executed tests. The class implements various Java operators that can be used in an API when using a domain-specific language. The changes on the return statements are not detected. The changes inside the header of the for-loops led to different return values, which were mostly empty collections. These changed return values were also not detected.	**Necessary method is not called**
(5) Negated the condition in a header of a for-loop. The method converts Java arrays to collection types and vice versa, e.g., int [][] to List<List<int». Negating the for-loop results in not entering it and the method returns an empty array or collection type object instead containing the correct elements.	(7) Negating the condition of an if-statement caused that a necessary method call is missed which is detected elsewhere.
Equivalent Mutant	**Necessary exception is not thrown**
(4) Negated the condition in an assignment. The method checks with `LongBuffer.hasArray()` whether a buffer has an array that can be accessed directly. If so, values of the buffer are set with array access, e.g., array[42] = 1 instead of buffer.put(42, 1) because it is faster. The result remains the same despite being faster, and we thus categorized this as a functional equivalent mutant.	(1) Negating the condition of an if-statement caused that a necessary custom-implemented runtime exception is not raised which is detected elsewhere.
(2) Negated the condition in an if-statement. The negated conditional changes the used thread pool size. This produces a functional equivalent mutant because the number of threads does not influence the functionality but the performance of the software.	**Uncaught custom-implemented runtime exception**
(1) Negated the condition in an if-statement. The if-statement is similar to `myVal = map.isEmpty() ? null : map.get(key)`. The map serves as a cache for future calls to quickly return myVal but will otherwise load the value with another method call and put it into the map. Thus, the method returns the same return value but does not use the implemented caching.	(36) Negating the condition of an if-statement led to throwing a custom-implemented runtime exception.
Same control flow	(4) Negating the condition of an if-statement that checked whether the method does not support a certain input value led to throwing a custom-implemented runtime exception.
(9) Negated a condition in an if-statement that contained multiple conditions which were connected with a logical OR-operator (||). When an if-statement usually evaluates to true, but it contains multiple conditions that are connected with a logical OR-operator then negating one condition does not change the control flow if any of the other conditions is true. This led to executing the same statements as without the mutator that was applied to only one condition.	(1) Negating the condition of an if-statement that checked whether a string is empty led to throwing a custom-implemented runtime exception.
	Uncaught NullPointerException
	(16) Negating the condition of an if-statement that checks whether an object is `null` led to executing code that calls a method of an object that is `null` and raises a `NullPointerException`.
	Uncaught IndexOutOfBoundsException
	(1) Negating the condition of an if-statement that implements a boundary-check that varifies that a variable is within array-boundaries. Negating the conditional led to accessing an array with an index that is out of bounds.
	(1) Negating the condition of an if-statement led to calling String.charAt (int index) of a character position that is larger than the length of a string.
	Uncaught NoSuchElementException
	(1) Negating the conditional `iterator.hasNext()` in a loop header leads to calling iterator.next() when the iterator is empty.
	Unknown why the mutant was killed
	(20) Conclusions could not be derived because of missing runtime information.
	(1) No source code was available to be analyzed. This happened because the Java reaction API was used and code from another software project was mutated that is used for performance tracing and instrumentation.

(Continues)

TABLE 6 (Continued)

Explanation why mutants survived with the number of observations	Explanation why mutants were killed with the number of observations
(7) Negated a condition in an if-statement that contained multiple conditions which were connected with a logical AND-operator (&&). When an if-statement usually evaluates to false, but it contains multiple conditions that are connected with a logical AND-operator then negating one condition does not change the control flow if any of the other conditions is false. This led to executing the same statements as without the mutator that was applied to only one condition.	
Additional method call	
(14) The negated condition led to calling an additional method that would usually not be called. This was often related to enabling certain behaviour or settings like enabling performance instrumentation, invalidating caches, clean-ups, setting log-levels, etc.	
Missing method call	
(32) The negated condition led to missing calling a method that would usually be called. This is often related to methods that close a connection, unlock a resource lock, set fields of classes, add elements to a cache, or were used inside exception handling to perform clean-ups.	
Incorrectly set class field	
(23) Negated the condition in an if-statement led to setting a class field with a different value than before the mutation. (4) Negated the condition of an if-statement which led to not adding objects to a collection (e.g. list, map, etc.) or deleting them from it. These collections were class fields.	
Different re-thrown exception	
(3) Negated the condition in an if-statement. The if-statement was placed inside a catch-clause that analyzes a raised exception and re-throws another according to certain criteria. Negating the conditional led to re-throwing a different exception than without the mutation.	
Newly raised exception	
(2) Negated the condition in an if-statement. Negating the if-statement led to throwing a NullPointerException that was not raised without the mutation. Raising the exception did not kill the mutant.	
Unknown why the mutant survived	
(13) Conclusions could not be derived because of missing runtime information.	

Since we analyzed the mutants statically, there was sometimes not enough runtime information to conclude whether this was the case or not. Thus, we categorized mutants where runtime information would be required to categorize them in the "NA" column. Note that we analyzed code within method- or class-scope. Thus, we could not categorize whether the changed conditional executed uncovered code for return-statements because we were missing the runtime information which method called the analyzed method.

As a final note, we want to point out that all numbers in the tables of this section represent the number of observations and not actual distributions. We used them to spot potential tendencies or recurring patterns, but they are incomplete because we did not analyze all mutants, and they are also prone to human analysis errors, which is natural when analyzing code manually without runtime information.

The `ReturnValuesMutator` may detect that a particular return-statement is not well tested, which cannot be found by the extreme variant. For example, consider a method that contains two return-statements and that there are two unit tests, symbolically A and B, that each executes one return-statement of that method. Moreover, consider that test A does depend on the correct return value, whereas test B does not depend on the return value. When extreme mutation testing creates its mutants, it would potentially execute both tests, A and B because both unit tests call the method. Since test A does require a correct return value, the extreme mutant would be killed, the result of test B would be ignored, and the method would be categorized as tested. The traditional approach would create two mutants, one for each executed return-statement, and would execute for each mutant the respective test. Since only one test requires the correct return value, this mutant would be killed, whereas the other would not. Thus, the traditional approach may find particular untested return-statements with its fine-grained approach of mutating instructions instead of whole methods.

Note that Table 10 shows that we also found 17 methods where all return-statements were mutated and survived, but the methods themselves are not pseudo-tested. This means that, although traditional mutants survived, the extreme ones were killed. For these methods, the return value is not important, but the body of the methods is relevant. They call other methods, set class fields, or raise exceptions which are required for the tests to pass. This is detected because the extreme approach additionally empties the whole method body instead of only changing the return value. Therefore, similarly as combining extreme mutators to imply a testing verdict, traditional and extreme mutation testing can be combined to determine whether a method can be removed and, if not, whether the return value is relevant or not.

Another issue that is not detected by extreme mutation testing are missing boundary checks. Boundary checks are conditions that contain comparisons of numbers with the conditionals <, <=, >, or >=. Thus, if a test suite does not provide a test where the exact boundary values are used, then these mutators usually survive because the control flow remains the same. The mutator that most frequently detected these missing boundary checks is the `ConditionalsBoundaryMutator` which is shown in Table 4. Note that Table 5 shows that mostly these conditional boundaries are not tested for if-statements, and boundaries on for-loops seem to be less likely to survive. This happens because for-loops are often fully iterated and raise exceptions if arrays are accessed with a counter variable that is out of bounds. For if-statements, multiple tests must be written that execute both branches with boundary values, which is often not the case.

Extreme mutation testing can also not detect that some particular methods calls or class field assignments are not tested. For example, Table 6 shows that the `NegateConditionalsMutator` often changes the control flow in such a way that some methods are additionally called or not at all. This is similar for class field assignments. Table 7 shows that the majority of negated conditionals are placed on if-statements that contain a single conditional. Thus, negating the conditional leads to a changed control flow, which leads to executing different statements that often cause detection of lacking tests for method calls and class field assignments. These issues are often not detected because the return value stays the same despite changing the control flow.

In addition to that, extreme mutation testing can also not detect insufficiently tested decisions. We consider a decision to be the result of which branch is taken when evaluating a boolean expression. A decision may be constructed by multiple boolean conditions that are connected via AND- or OR-operators. In these cases, negating a single condition may result in executing the same branch. If any condition in the AND-decision is false, then the whole decision will always be false. If any condition in the OR-decision is true, then the whole decision will always be true. Thus, there are potentially missing test cases where all conditions have the same value for these decisions, that is, `true` for AND-decisions and `false` for OR-decisions.

The `NegateConditionalsMutator` can also reveal issues with handling thrown exceptions. Table 6 shows three cases where a changed conditional caused rethrowing a different exception which is not detected by the test suite. Similarly, in two cases, exceptions were raised that are normally not thrown, but the test suite does not detect any issue. Thus, these issues are indicators for overly generous exception handling. Extreme mutation testing cannot detect these issues because the currently implemented mutators do not address exception handling at all.

The rest of the mutants are either equivalent mutants or detect mostly similar, if not the same, issues like the rest of the mutants. For example, the `IncrementsMutator` produces a subset of the `MathMutator`. According to PIT,[¶] the `IncrementsMutator` differs by only mutating local variables, whereas the `MathMutator` mutates member variables, that is, class fields.[#] Table 8 shows that modified increments or decrements remain undetected when the changed integer is used for array access, but the array length is only one. In this case, the loop terminates after one iteration and changing the increment has no effect on code execution. No `IndexOutOfBoundsException` is thrown because the loop header checks for the lower boundary of the counter variable and exits the loop before the exception can be raised. We also found that the changed increments or decrements remain undetected if the return value does not change. In one case, the changed integer led to negating the condition in an if-block. This resulted in executing another return-statement from a different line of code. However, that return-statement returned the same boolean value as the one that was used prior mutation.

Table 9 shows that the `MathMutator` reveals similar issues like the `ConditionalsBoundaryMutator`, `NegateConditionalsMutator`, and `ReturnValuesMutator`. When the mutator changes a variable that is used in a boundary check, it reveals a missing boundary check. This could be found by the `ConditionalsBoundaryMutator`. When the mutator changes a variable that is used in a condition, this may have implications on decisions in the control flow. If the decision is changed and a different branch is executed, then additional, missing, or wrong method calls may not be detected. If a decision consists of multiple conditions, then the same control flow may be executed, depending on the logical operators that construct the decision. If the mutator changes the calculation of how a class field is set, then the wrongly set class field may not be detected. Many of these issues could be found by the `NegateConditionalsMutator`. When the `MathMutator` changes the value of a variable that is returned, a different return value may not be detected. Table 10 shows that this could be found by the `ReturnValuesMutator`.

Note that not only the `MathMutator` may reveal issues that are found by other mutators. For example, Table 7 shows that the `NegateConditionalsMutator` mutated 43 conditionals on return-statements that contained only a single condition. Thus, these mutants are the same ones that are created by the `ReturnValuesMutator` because they change the boolean return value. The `ConditionalsBoundaryMutator` and `NegateConditionalsMutator` can both change decisions and then cause that void methods are not called, which would be regularly called. This would also be found by the `VoidMethodCallMutator`. Thus, very often, mutants are duplicates or, depending on the test, would be killed together when killing one of the other mutants. In some cases, a return-statement which contains a condition that compares two calculated numbers may create up to five mutants.

TABLE 7 Inspected number of mutants by their status, statement type, and information whether they executed uncovered code for the NegateConditionalsMutator

				Executed uncovered code		
Status	**Statement type**	**Conditionals**	**Total**	**Yes**	**No**	**NA**
Survived	If	≥ 1	135	90	43	2
		≥ 2	43	1	36	6
	For	≥ 1	12	0	12	0
		≥ 2	0	0	0	0
	While	≥ 1	1	0	1	0
		≥ 2	4	0	4	0
	Return	≥ 1	43	na	na	43
		≥ 2	2	na	na	2
	Assignment	≥ 1	1	0	1	0
		≥ 2	6	0	6	0
Killed	If	≥ 1	92	53	39	0
		≥ 2	18	7	8	3
	For	≥ 1	9	0	9	0
		≥ 2	0	0	0	0
	While	≥ 1	1	0	1	0
		≥ 2	0	0	0	0
	Return	≥ 1	2	na	na	2
		≥ 2	1	na	na	1
	Assignment	≥ 1	0	0	0	0
		≥ 2	1	0	0	1

TABLE 8 Analysis why mutants of the IncrementsMutator survived or were killed

Explanation why mutants survived with the number of observations	Explanation why mutants were killed with the number of observations
Same return value is returned	**Uncaught IndexOutOfBoundsException**
(2) Changed an increment to a decrement of a counter variable (e.g., `i`) that is used to access an array (i.e., `array[i++] = val`) inside a loop. The change is not detected because the loop is iterated only once and the decrement happens after execution of the assignment.	(43) Changed the increment to a decrement of a counter variable (e.g., `i`) in the header of a for-loop. This leads to an `IndexOutOfBoundsException` when accessing an array within the for-loop because the array is accessed with a negative index.
(1) Changed the increment to a decrement of an integer variable that is used afterwards in an if-statement. There is a return-statement inside the not entered if-block. Skipping over it leads to another return-statement that returns the same boolean value.	(2) Changed the decrement to an increment of a counter variable (e.g., `i`) inside a loop that led to calling `String.charAt(int index)` of a character position that is larger than the length of a string.
	Unknown why the mutant was killed
	(6) Conclusions could not be derived because of missing runtime information.
	(1) No source code was available to be analyzed. This happened because the Java reflection API was used and code from another software project was mutated that is used for performance tracing and instrumentation.

Lastly, the `VoidMethodCallMutator` differs from the extreme mutation testing approach by that it is known which method call in particular can be removed. In general, this mutator finds the same issues as the extreme approach, despite being more granular. For example, Table 11 shows the verbs of the removed void method calls. The verbs "assert" and "check" are frequently listed in both columns. When manually inspecting the results, these verbs belonged to the same tested methods but occasionally, their removal was not detected and, if used by other methods, their removal was detected. We cannot conclude which granularity is more beneficial when comparing execution time against usefulness of both approaches. Thus, this could be subject to future research work.

TABLE 9 Analysis why mutants of the MathMutator survived or were killed

Explanation why mutants survived with the number of observations	Explanation why mutants were killed with the number of observations
Missing boundary check	**Uncaught IndexOutOfBoundsException**
(4) Changing the mathematical operation that calculates a boundary value that is used in an if-statement. Changing the boundary value led to executing the same code and statements because the conditional boundary value was never used by any of the tests.	(16) The mutator changed the increment to a decrement of a counter-variable (e.g., i) in the header of a for-loop. This leads to an `IndexOutOfBoundsException` when accessing an array within the for-loop because the array is accessed with a negative index.
(1) Changing the mathematical operation that calculates a boundary value. The method counts recursively the number of nodes in an acyclic graph up to a maximum depth. The if-statement is used to check whether that depth is reached. There is no test that uses a graph that exceeds the maximum depth and does not validate the changed boundary.	(6) Changed the decrement to an increment of a counter-variable (e.g., i) inside a loop that led to calling `String.substring(int beginIndex)` with a `negative beginIndex` which raised an `IndexOutOfBoundsException`.
Same control flow	(1) Changed mathematical operator that calculated at which position an object should be put into a `LongBuffer`. Accessing the buffer with an index that is out of bounds led to throwing an `IndexOutOfBoundsException`.
(8) Changing the mathematical operation that calculates a value that is used in an if-statement that contained multiple conditions which were connected with a logical OR-operator (||).When an if-statement usually evaluates to true, but it contains multiple conditions that are connected with a logical OR-operator then negating one condition does not change the control flow if any of the other conditions is true. This led to executing the same statements as without the mutator that was applied to only one condition.	(1) Changed mathematical operator that is used to calculate the capacity of a `StringBuilder`. The `StringBuilder` is then initialized with a negative capacity, which raised an `IndexOutOfBoundsException`.
(3) Changing the mathematical operation to access a two-dimensional array that checks for the length of the second dimension (e.g., array [i-1].length). Since all lengths of the second dimension are the same and no `IndexOutOfBounds` exception was raised because the array was large enough, the same statements as without the mutator were executed.	**Changed return value detected**
Same return value returned	(10) The changed mathematical operator directly changes the returned variable. This led to returning a different return value which is detected elsewhere, i.e., outside the method
(7) Changing the mathematical operation to access an array (e.g., array [i-1]). A loop checks whether all elements in an array are equal and returns false if any of the values is different, otherwise true. The mutator causes skipping one element from comparison but no case was tested were the return value is false, thus the mutant survives.	(3) The changed mathematical operator indirectly changes the returned variable. The returned variable is created by substring(`int beginIndex, int endIndex`) but the arguments, i.e., the indices, changed. This led to returning a different return value which is detected elsewhere, i.e., outside the method.
Changed return value is not detected	(9) The changed mathematical operator indirectly changes the returned variable. The returned variable is chosen from an array, but the mutator changes the index that selects the element from the array. This led to returning a different return value which is detected elsewhere, i.e., outside the method.
(44) Changing the mathematical expression in a return-statement. The mutators were placed on a class that is responsible for the majority of executed tests. The class implements various Java operators that can be used in an API when using a domain-specific language. The changes on the return statements are not detected.	**Necessary method is called with wrong arguments**
(4) Changing the mathematical expression changes how a string is constructed and returned. The changed string is not detected.	(3) The mutator changed the bit shift operator that alters the value of a long variable. This variable is then used by `OutputStream.write(int b)`.Writing the wrong value into the output stream is detected elsewhere, i.e., outside the method.
(1) Changing the mathematical expression in an if-statement. The if-statement calculates a chunk size for receiving from a stream. The mutator should actually lead to overflowing the buffer, but the received bytes from the stream are always less than the original chunk size and thus, the mutant survives.	(2) The mutator changed the bitwise AND-operator to a bitwise OR-operator which changed the value of a long variable. This variable is put into a `LongBuffer` which is detected elsewhere, i.e., outside the method.
Method is called with wrong arguments	**Uncaught custom-implemented runtime exception**
(1) Changing the mathematical expression in a variable assignment. The variable is used to print a progress bar which is changed by the mutant but was not detected.	(1) The mutator changed the modulo-operator in an if-statement to a multiply-operator. This changes the conditional which led to throwing a custom-implemented runtime exception.
(1) Changing the mathematical expression in a variable assignment. The variable is used to write performance metrics into an `OutputStream`. The different value that is used by the write method is not detected.	**Unknown why the mutant was killed**
Incorrectly set class field	(11) Conclusions could not be derived because of missing runtime information.
(3) Changing the mathematical expression in a variable assignment. The variable is a class field of a performance instrumentation class. The assignment is the elapsed time by calculating the difference between two `timestamps`. The mutant caused the `timestamps` to be added and survived.	(7) No source code was available to be analysed. This happened because the Java reflection API was used and code from another software project was mutated that is used for performance tracing and instrumentation.

(Continues)

TABLE 9 (Continued)

Explanation why mutants survived with the number of observations	Explanation why mutants were killed with the number of observations
Equivalent mutant	
(2) Changing the mathematical expression in a variable assignment. The variable is the calculated capacity that a `StringBuilder` should be initialized with. Since capacity is dynamically added when it is required by the `StringBuilder`, this mutant does not change the functionality of the code.	
(2) Changing the mathematical expression in a variable assignment. The variable is used to set the number of threads in a thread pool. This produces a functional equivalent mutant because the number of threads does not influence the functionality but the performance of the software.	
Unknown why the mutant was killed	
(3) Conclusions could not be derived because of missing runtime information.	
(2) No source code was available to be analysed. This happened because the Java reflection API was used and code from another software project was mutated that is used for performance tracing and instrumentation.	

We want to point out that, with all these findings, significant time savings and improvements are well possible. For example, application of the `ConditionalsBoundaryMutator` could be often avoided. Whether conditional boundary values were used during a test can be detected by extending JaCoCo's bytecode instrumentation. An additional check that verifies whether the compared numbers are equal and which comparison operator was used should be sufficient to detect whether the test was executed with boundary values. This alone would save 23,607 executed tests of nonpseudo-tested methods, which account for 19.3% of all executed tests of nonpseudo-tested methods. Similarly, the `NegateConditionalsMutators` could only be applied when all branches of a decision are covered. Table 7 shows that the mutator creates many mutants that execute uncovered code, that is, code that was not executed by the test suite. However, the additional findings that are obtained through the focus group (see Section 4.4) show that these mutants are considered to be less relevant for developers.

In conclusion, the traditional approach naturally finds more particular testing issues than the extreme variant. Traditional mutation testing mainly discovers issues that are related to changed method calls, return values, and class fields, as well as issues related to weak exception handling, missing boundary checks, or insufficiently tested branches. However, the technique suffers from creating many similar, duplicate, or equivalent mutants which bloats execution and analysis times. A more strategic approach that focuses on mutating these properties only once would reduce the number of mutants without losing precision. Further, we think that some mutants may be simply not necessary if the code coverage report from JaCoCo would contain more information. Additionally avoiding the creation of mutants that are irrelevant for developers, like the ones that execute previously uncovered code, seems also reasonable. Thus, we see a lot of potential time savings for traditional mutation testing.

TABLE 10 Analysis why mutants of the ReturnValuesMutator survived or were killed

Explanation why mutants survived with the number of observations	Explanation why mutants were killed with the number of observations
Changed return value was not detected	**Uncaught RuntimeException**
(56) The mutator changes the return value which is not detected.	(4) The mutator itself throws a `RuntimeException` because the code was written to return `null`.
(39) The mutator changes the return value, which is not detected. The mutators were placed on a class that is responsible for the majority of executed tests. The class implements various Java operators that can be used in an API when using a domain-specific language. The changes on the return statements are not detected.	**Changed return value detected**
(17) The mutator changes the return value, which is not detected. The return value is only an indicator whether something has changed or whether it was successful, but it is not actually verified. For example, these methods add objects to a list and return true if the object was created and otherwise return false.	(61) The mutator changes the return value which is detected elsewhere, i.e., outside the method.
	Unknown why the mutant was killed
	(2) No source code was available to be analysed. This happened because the Java reflection API was used and code from another software project was mutated that is used for performance tracing and instrumentation.

TABLE 11 Verbs of the void methods that the VoidMethodCallMutator mutated by status

Verbs of methods that can be removed and the resulting mutant survives with the number of observations			Verbs of methods that can be removed, and the resulting mutant were killed with the number of observations	
(38) set	(2) disconnect	(1) handle	(10) write	(1) log
(34) assert	(2) forEach	(1) indent	(7) assert	(1) marshal
(29) check	(2) initialize	(1) info	(7) check	(1) print
(8) add	(2) stop	(1) invalidate	(6) set	(1) read
(8) clear	(2) switch	(1) mark	(3) sync	(1) remove
(8) put	(1) apply	(1) remove	(2) add	(1) setup
(7) sync	(1) connect	(1) run	(2) copy	(1) sort
(6) close	(1) context	(1) send	(2) fill	(1) veri
(5) verify	(1) count	(1) store	(2) split	
(4) log	(1) determine	(1) update	(2) update	
(4) reset	(1) error	(1) validate	(1) clear	
(4) trace	(1) fill	(1) wait	(1) define	
(3) notify	(1) finish		(1) execute	
(3) register	(1) flush		(1) get	

4.3 | RQ2: Time savings

Table 12 shows the numbers of mutants and execution times of both mutation testing approaches. Note that these numbers are obtained directly from PIT to show actual execution time. Through our extensive manual inspection of mutants, we noticed that a single class in the software project accounted for 82% of the executed tests and 24% of the generated mutants of the traditional mutation testing approach. The class is used to parse boolean and mathematical expressions like `(a == b)` or `(a + b)` from strings and then returns an `Object` that encodes the value that the JVM would return when these expressions were compiled. The methods of the class contain more than 500 if-statements and more than 500 return-statements. This is the reason for the high number of mutants and executed tests. Since this class had such big influence on the results of traditional mutation testing and is so different from the rest of the code, we see it as an outlier in our data. Thus, we reran mutation testing without it and extended the entries in Table 12 with the new results. Additionally, we summarized the results from Tables 1–3 into Table 13 to quickly identify how time savings are distributed.

Note that the data obtained from PIT in Table 12 and the aggregated data in Table 13 slightly differ. This has mainly two reasons: First, the data obtained from PIT also include methods which may be mutated by only one of the two approaches, like constructors or simple methods. This is already described in the previous section and explains, for example, why the total number of mutants and tests is higher for the data obtained from PIT. Second, the number of tests is also higher because PIT takes tests of mutants into account that terminate with a status like `TIMED_OUT` or `NON_VIABLE`. However, these mutants are excluded in our comparison. Therefore, Table 12 shows which mutation testing results should be expected by PIT. Table 13 shows the intersection of both mutation testing approaches, which excludes particularities of each one.

Hence, to answer RQ2, we only refer to the data in Tables 1–3 and 13. We first looked at mutants that result from mutating instructions on pseudo-tested methods. Since these methods are pseudo-tested, there is no need to additionally mutate the contained instructions because the testers would know that the tests never verify these methods.

The extreme variant generated 346 mutants and executed 511,538 tests. The traditional approach generated 573 mutants and executed 514,040 tests for the same methods. We expected that the extreme approach of mutating whole methods would generate fewer mutants than the traditional approach. We also expected that many traditional mutants can be safely ignored, and testers need less analysis time.

When comparing both approaches by taking only pseudo-tested methods into account, this is true. The traditional approach generated 227 additional mutants on pseudo-tested methods compared with the extreme approach. This means that traditional mutation testing created 65.6% additional mutants. However, the traditional approach executed only 2502 more tests than the extreme one. This means that there are only 0.5% more executed tests.

Moreover, when comparing both approaches by taking all mutants and executed tests into account, the differences shrink to almost negligible amounts. The 227 additionally created mutants only account for 3.6% of all traditional mutants together.[‖] The number of additionally executed tests shrinks then to 0.4%. Thus, when looking at all mutants and tests of the traditional approach, the savings from mutants on pseudo-tested methods are small. Hence, although extreme mutation testing generates noticeably fewer mutants in general, the majority of savings, in terms of execution time, come from ignoring traditional mutants on nonpseudo-tested, that is, tested, methods.

TABLE 12 Mutation testing results obtained from PIT

Type	Survived	Killed	Total	Mutators	Executed tests	Time
Extreme	409	2297	2706	19	597,877	13 min
Traditional	2176	5813	7989	7	4,228,169	37 min
Extreme[a]	409	2256	2665	19	566,669	9 min
Traditional[a]	1370	5272	6642	7	774,173	17 min

[a]Without the class that is responsible for the majority of executed tests.

TABLE 13 Number of mutants and executed tests from methods that are mutated by both approaches and without the class that is responsible for the majority of executed tests

Testing verdict	Type	Survived	Killed	Total	Executed tests
Pseudo-tested	Extreme	346	0	346	511,538
	Traditional	463	110	573	514,040
Partially tested	Extreme	44	44	88	1930
	Traditional	37	144	181	2055
Tested	Extreme	0	2136	2136	29,238
	Traditional	764	4766	5530	119,887
Any	Extreme	390	2180	2570	542,706
	Traditional	1264	5020	6284	635,982

There are multiple reasons why the difference is not that large for pseudo-tested methods. First, the extreme mutator for void methods and the `VoidMethodCallMutator` of the traditional approach create mostly the same mutants (see Table 1). The difference in the amount of created mutants originates from the mutation testing implementations. When running PIT, the user configures which packages and classes should be mutated. The extreme approach will only mutate void methods that belong to these packages and classes. The traditional approach may remove calls to void methods that are outside that configuration because it mutates any calls to methods that belong to the configured scope.

For example, consider PIT would be configured to mutate classes that belong to the packages `com.example.mypackage.*`. Also consider that a single unit test calls a void method named `com.example.mypackage.MyClass.myMethod()`. That `myMethod()` itself makes a call to `java.util.ArrayList.clear()`. The extreme technique would only mutate `myMethod()` because its class belongs to the configured packages that should be mutated. The traditional technique would only mutate the call to the `clear()` method because it is called by `myMethod()` and the class of it belongs to the configured packages that should be mutated. It would not mutate the call to `myMethod()` itself because it is directly called by the unit test and the traditional approach does not mutate instructions on unit tests. Thus, the traditional approach can mutate calls to different packages, whereas the extreme one does not. However, if the method would be called by another method of a class that belongs to `com.example.mypackage.*` then the call to `myMethod()` would also be mutated.

Second, looking at Table 2 at nonvoid methods and the respective `ReturnValuesMutators`, the traditional approach generated either closely equal or less than half the number of mutants compared with the extreme one. For methods that return an object, the traditional mutator changes the returned value to `null` if the object is not `null`, otherwise the mutator raises a `RuntimeException`. Raising the exception and not catching it is the reason that some mutants are killed even though these methods are pseudo-tested. The number of traditional mutants depends on the number of executed return-statements a method has. Thus, for methods that return an object, in total, only two additional traditional mutants are created because those methods had two executed return-statements. For nonvoid methods besides the ones that return an object, extreme mutation testing must create at least two mutants to imply the testing verdict. For example, if a method returns a boolean, the extreme variant will empty the method body and return once `true` for all tests and once only `false`. If the return value is the object version of the primitive type, that is, `Boolean`, then it will additionally mutate it to return `null`. Thus, the extreme mutator creates about two to three times as many mutants than the traditional approach given that the traditional approach has to mutate only one return-statement per method, which was mostly the case.

Third, the number of mutants that are created by the `IncrementsMutator` and the `MathMutator` is small because the analyzed pseudo-tested methods simply do not contain many increments, decrements, or mathematical expressions. Their number of mutants and executed tests are almost negligible when comparing them to the number of mutants and tests that originate from the traditional approach on pseudo-tested methods.

Fourth, the mutators that change conditionals also increase the total number of traditional mutants on pseudo-tested methods. However, the amount is only a fraction of mutants that the `ReturnValuesMutator` creates. Noteworthy, both mutators also create mutants that were killed despite being generated on pseudo-tested methods. The `ConditionalsBoundaryMutator` often creates mutants that raise `IndexOutOfBoundExceptions` when iterating in a loop through arrays with a counter variable (see Table 4). The `NegateConditionalsMutator` often creates mutants that raise either `NullPointerExceptions` because a check whether an object is `null` is negated or raise other custom-implemented runtime exceptions (see Table 6). These custom-implemented runtime exceptions are often checks to verify that the software is not in an invalid state and are usually not executed by any test.

Lastly, there are multiple reasons why time savings are noticeably larger on tested methods. Extreme mutation testing stops executing further tests once the extreme mutant is killed. This saves a significant number of mutants and tests when mutating, for example, void methods. Whereas the traditional approach will mutate each occurrence of a void method, the extreme approach stops as soon as it finds a test that requires that method. In other words, the extreme approach focuses on a method in general and not on every method call.

Another reason is that pseudo-tested methods are, in general, less complex and contain fewer instructions to mutate than tested methods. This fact was already found by Niedermayr[4] but can be well observed when comparing Tables 1 and 3. In Table 1, the number of mutants for the `ReturnValuesMutator` and the number of methods are very close to each other. In Table 3, the number of mutants is always noticeably larger than the number of methods. Thus, tested methods often have multiple return-statements in our case software, and each one is individually mutated by the traditional approach. Since the most common return category is `object` in our case software, the effect is amplified because only one extreme mutant is required for that return category. Similarly, this can be observed by looking at the number of mutants that are created by the remaining mutators. There are, in general, more mutants created by the remaining mutators because there are more statements that can be mutated.

In summary, the number of mutants that are generated for pseudo-tested methods and substituted traditional mutants are roughly the same when comparing them to the total number of traditional mutants. When looking at execution time by using the number of executed tests as an approximate measure, it is also roughly the same. However, although there are no savings in execution time, there are savings in analysis time. The extreme mutation testing verdict is valuable information when analyzing mutants or methods. The traditional approach does not derive that information and would produce numerous mutants that would be otherwise manually inspected by developers. Thus, when applying extreme mutation testing first, about 9% of all traditional mutants do not need to be created because these would be superfluous. This comes, however, at the expense of mutating all methods and thus increasing the total execution time. Nonetheless, we consider it worth executing extreme mutation testing first because the increase in execution time is rather small. About 94% of all tests that the extreme variant executes originate from pseudo-tested methods, which makes the overhead for mutating the rest of the methods negligible.

4.4 | RQ3: Relevance of mutants

The subsequent paragraphs show the summarized results of this qualitative coding approach. The bold headings for the paragraphs are the central categories that were obtained by selective coding. The text below the headings are the statements from axial coding that were merged into a continuous text. Table 14 shows an example coding from the transcript.

4.4.1 | Focus on customer needs

One of the most recurring reasons why a mutant would be killed was related to the needs of customers. If a mutant resides on code that affects the API that can be used by the customer, then the team considers that these mutants should be fixed with high priority because all API functionality should be well tested. Any mutants that reveal testing issues of features that has been discussed with the customer or are known to be

TABLE 14 Example of the applied Straussian grounded theory coding on the transcript of the focus group

Transcript	Open coding	Axial coding	Selective coding
"I would not fix it because I think we should have system-level or integration tests to detect this, and maybe that's the reason it's pseudo-tested. So, it's not an untested feature."	System-level tests Integration tests Won't fix	If a mutant resides on code that is tested by higher level tests then it may indicate that it does not need to be killed because it is already tested.	Existing higher level tests

important for the customer should be killed as well. This also includes mutants that reveal testing issues of internal performance optimizations of which the customer is unaware because performance is a customer requirement. The frequency of how often a customer uses a feature influences whether it is well tested in general.

4.4.2 | Existing higher level tests

Many mutants that we have presented to the developers were tested by higher level tests like integration or system-level tests. Developers often considered mutants that reside on code that is tested by higher level tests do not need to be killed because they are already tested. However, it is not always known whether code is tested by higher level tests. Large and complex classes are mostly tested with higher level tests because they are hard to test with unit tests. The developers agreed that these large classes should be split up and refactored to test them with unit tests.

4.4.3 | Code reviews and API features

Established testing practices like code reviews influence how the code is tested. The developers were sure that one of the presented pseudo-tested methods would not exist with the existing code review process which they have in place nowadays. In general, we observed that discussing mutation testing results in a code review fosters clarity of how to test methods and how an API should behave and be implemented. Thus, these mutants would be killed when a consensus between developers is reached how the code should look like after discussing the mutation testing results. Surprisingly, developers may also consider equivalent mutants useful when they reveal that the API should behave differently. Then they would rewrite the code.

4.4.4 | Point in time when mutation testing is performed

The developers categorized one pseudo-tested method as technical debt. In general, technical debts with low importance are not actively fixed by the developers. However, whereas low-priority mutants would not be killed when spotted during a code review, they would be killed when observed while programming.

4.4.5 | Code readability and coding style

Coding style generally influences which mutants are generated. For example, `String.indexOf(int ch)` returns the index of the first occurrence of a provided character in a string if the character is present. Otherwise, the method returns -1.[**] Thus, developers can check whether such a method returns -1 or check for any negative return value, because the index can only be greater or equals to zero if a solution exists. Checking for the particular return value -1 is done with the equals operator $==$ whereas checking for any negative return value could be done with the less-than operator $<$. Depending on the case, both coding styles may be applicable, but different mutants would be created. Developers would not kill equivalent mutants when the coding style contributes to readability.

4.4.6 | Team structure

Multiple teams contribute to the development of the software project. Developers would not kill a mutant if the team is not responsible for a particular part of the software project. If a mutant resides on code that is tested by higher level tests and another development team is considered to quickly notice an issue, then the priority to kill the mutant additionally shrinks.

4.4.7 | Code areas that process sensitive data

Some code areas process sensitive data like intellectual property or confidential data. If a mutant resides on code that processes sensitive data, then the developers state that these should be killed with high priority.

4.4.8 | Legacy code

The software project also contains legacy code. Legacy code is usually not revised and thus not target of mutation testing. In addition to that, developers stated that they would not kill a mutant by rewriting code when it breaks backwards compatibility but rather write additional tests if required.

4.4.9 | Number of existing tests for a class or method

The developers stated that they are guided firstly by code coverage to see *what* is potentially tested and secondly by mutants to check *if and how well* the code is tested. Thus, developers argued that they would first increase code coverage of a method when they intend to test it and only then apply mutation testing in the second step. This implies that sparsely tested classes or methods are generally of less importance for mutation testing because the developers would be mainly guided by increasing code coverage to create new tests and not by killing mutants.

4.4.10 | Additional findings

Despite looking for factors that influence a developer's decision to kill a mutant, we also obtained results on how developers write unit tests and use code coverage with mutation testing. In general, developers noted that private methods are not directly tested but are not less important to test. Static methods are perceived to be more difficult to test than nonstatic ones with unit tests. Moreover, even if a method has a return value, it does not mean that it is the most important property to focus on when testing it. Despite having many higher level tests, the developers agreed that unit tests should be the main source of code coverage because they are cheap to execute and can quickly detect issues. However, they also agreed that it is not always clear where code coverage comes from when running unit tests. Sometimes, few unit tests may result in a lot of coverage without really testing the code. This issue is identified by mutation testing. Mutation testing itself is perceived as a technique to measure the quality of the tests and reveals missing test oracles. Developers consider mutants more meaningful that execute statements that were already executed by the regular test suite, that is, mutants that execute previously uncovered code are not perceived as meaningful. Lastly, some mutants were already fixed in the most recent software version when presenting them to the developers in the focus group. This indicates that mutation testing is well capable to spot relevant issues when testing code.

We expected that we would find what exactly developers motivate to kill a mutant or not. We also thought that some mutant types may be more relevant to developers than others. Surprisingly, we found that most of the intentions to kill a mutant or not are not related to any code- or mutant-specific reasons. Motivations to kill mutants or not mostly originate from priorities which features should be well tested and the organization of the development team as well as its established development practices. The development team focuses on developing a high-quality software for customers. Thus, features of the API that are used by customers are agreed to be well tested. Furthermore, areas that process sensitive data have a high importance to be tested because they may affect customers or the company itself. Every time a particular feature shall be well tested, mutation testing should verify the quality of the tests. If mutation testing, either the extreme or the traditional variant, finds issues with the quality of these tests, then developers are willing to kill these mutants to improve the test suite.

But not only the importance of well-tested features is a motivator to kill mutants, also efficiency and team organization influence which mutants are selected to be killed. For example, if it is known that a development team is responsible to test a particular part of the software, then this is not tested again by another development team. The same is often true if higher level tests, like integration tests, exist for particular unit tests. Moreover, some agreements exist about what shall be tested or not. In our case, legacy code was not required to be tested and thus no mutants that change that code would be killed.

Furthermore, it matters whether mutation testing is applied during code development or discussed during a code review. The results from our conducted focus group confirm that observation again. Mutation testing sometimes reveals low-priority mutants that cannot be spotted with code coverage alone. These mutants would be killed by the developers when mutation testing would be applied during code development. However, these mutants would not be killed when they would be spotted during a code review because developers would not consider them to be important enough to be addressed at that phase of development. In contrast, some mutants would be ignored during code development but would be killed after discussing them in a code review. Discussions about these mutants reveal that a method or a feature should behave differently or should be written in a different way. These discussions are missing during code development, which is the reason these issues remain undetected during that development phase.

Lastly, few reasons why mutants are killed or not are related to the code itself that is mutated and its readability. Some coding styles favors creation of certain mutants, whereas a different coding style might not create these mutants. If a mutant is created that is related to the coding style, usually developers prefer the coding style that they consider to be more readable. These mutants have a rather low priority and would not be killed by writing additional tests. Instead, the existing code would be rewritten to not create these mutants. Otherwise, these mutants would be just ignored.

4.5 | Evaluation of validity

4.5.1 | Construct validity

Construct validity refers to the degree of how appropriate our measures represent the studied constructs.[25] Constructs are concepts that are not directly observable, e.g., software quality. Measures are conceptualizations of constructs, for example, the number of mutants. For RQ1, we studied the coarse granularity of the extreme technique compared with the traditional approach. As measures, we used the results obtained through qualitative coding from manual inspection of mutants. We think that these measures precisely show the limits of extreme mutation testing because we directly observe why a traditional mutant survives on an apparently tested methods. To increase construct validity, we did not only analyze traditional mutants on tested but also on partially and pseudo-tested methods. Pseudo-tested methods contained killed traditional mutants, which we did not expect. This was related to the finer granularity of the traditional approach, so we extended our set of analyzed mutants to not bias our results only towards traditional mutants on tested methods. It is a threat to construct validity that we only did a static analysis of the mutants and had no other method of analysis to triangulate our results. For example, we could have used runtime information for analysis, looked additionally at the tests that were executed, or even kill mutants. These methods may have revealed different reasons or confirmed our observations. However, we had to decide for one method of analysis and considered that many statically analyzed mutants would be most valuable.

For RQ2, we studied the potential savings when applying extreme prior traditional mutation testing, which is our construct in this case. As a measure, we used the number of traditional mutants and executed tests that result from mutating instructions on pseudo-tested methods. We compared these numbers to the total number of extreme mutants and executed tests of pseudo-tested methods. Additionally, we compared these numbers to the total number of mutants and executed tests of all traditional mutants. We think that these comparisons are appropriate measures to define savings because we calculate fractions of populations and do not use absolute numbers that are hardly comparable to other software projects. We increased construct validity by using not only the number of mutants but also the number of executed tests when we compared execution times. However, we used the number of executed tests only as an approximate measure. It is not possible to calculate the exact execution time from the number of tests because each test may have a different execution time and its duration is not recorded by PIT. Table 12 shows that when we removed the class that executed the majority of tests, the number of executed tests decreased by 82%, whereas execution time decreased only by 54%. Because it is not possible to calculate the exact execution time, it is also not possible to calculate the real savings made. Thus, using executed tests as an approximation is a threat to construct validity.

Lastly, for RQ3, we studied the motivations and intentions of developers to kill mutants. As measures, we used our qualitative coding results that we obtained by conducting a focus group. We think these measures were suited to conclude motivations and intentions because observing mutants and discussing why a developer would kill these or not should reveal them. To prevent our codes to be biased because a single researcher derived them, we have double-checked them by another researcher that did not participate in the focus group and, thus, increased construct validity. However, we only conducted a single focus group with a single set of preselected mutants. Conducting multiple focus groups and perhaps additional interviews would have saturated our results and thus increased construct validity.

4.5.2 | Internal validity

Internal validity refers to which extent our measures are really caused by our independent variables and not by some other influences.[25] Independent variables are, for example, the traditional and extreme mutation testing techniques. The mutation testing results in Table 12 show a nonproportional decrease in execution time in comparison with the number of executed tests. It is a threat to internal validity that the execution time is not only caused by survived or killed mutants. Mutants may have, besides the `KILLED` and `SURVIVED` status, the `TIMED_OUT` and `NON_VIABLE` status when using PIT. Timeouts may be caused by mutants that create infinite loops and mutants may be nonviable if the mutated bytecode cannot be loaded by the JVM. However, there is no absolute time limit that can be configured for time-outs. Therefore, we cannot subtract it from the execution time. Furthermore, it is unknown how large the overhead for nonviable mutants is. Just for comparison, the number of timeouts was 44 with the class that executed the majority of tests and 32 without. The number of nonviable mutants was three for both executions. To account only for the time that the analyzed mutants caused in our comparison, we used the number of executed tests as an approximate measure. This, however, comes with its own set of threats to construct validity that we discussed in the previous section.

As already described in our previous study,[8] we rely on the correctness of the mutation testing tool PIT and of both mutation engines. However, nondeterminism influence the mutation testing results. For example, some tests of the software project are designed to test parallel execution features. Other tests depend on each other and rely on static classes and their fields. PIT splits the mutation testing workload in chunks and independently executes these. To avoid effects that are caused by nondeterminism, we set the number of execution threads to one and removed parallel tests where possible. We further increased internal validity by performing mutation testing with each engine three times and verifying that we obtained the same number of mutants and tests for each run.

We also rely on the correctness of our Python scripts that extracted data from the JSON files that PIT created. Programming errors may influence the obtained results and are thus a threat to internal validity. We verified the correctness of our scripts by creating checksums wherever possible. For instance, from the output log that PIT produced, we knew how many mutants were generated in total and how many survived or were killed. When we parsed the mutants with our Python scripts, we counted them and verified manually that these numbers match. When we categorized mutants by their return category, we summed up the number of mutants from all categories and programmatically verified that the numbers correctly add up. We did this similarly for the number of methods and the number of executed tests. We also gained high confidence that these numbers are correct through our broad manual inspection of mutants. Since we analyzed mutants from each return category and mutator combined, we would have quickly spotted any programming errors.

It is a threat to internal validity that only one researcher did the analysis and coding of the manual inspection of mutants. Other researchers may have described or categorized the analyzed mutants differently. Therefore, the results are biased towards the interpretation of that one researcher. Furthermore, human errors naturally creep in when analyzing more than 1000 mutants in 3 weeks. Therefore, we explicitly mentioned that the number of observations is not actual distributions and that it is only provided to spot tendencies. We increased internal validity by analyzing mutants with similar properties that were randomly selected in chunks. We did this to easier spot common or unique reasons and categories and minimize human error by avoiding context switches. In addition to that, our chunk size of 50 mutants was large enough to not bias our results to particular issues and to capture a broad range of reasons why a mutant survived or was killed.

Lastly, the experience of the developers may have influenced the answers provided in the focus group. All developers were already familiar with mutation testing and had multiple years of experience in Java programming. Inexperienced developers that are not familiar with mutation testing could have answered differently, but we could not choose different developers. Moreover, the selection of the presented mutants was not random and therefore may have had an influence on the obtained results. We justify our selection process through the fact that the software project is large, and we wanted to obtain opinions from as many developers as possible. To achieve that, we had to select code that all developers are familiar with. However, our selection may have unintentionally limited which answers could be given. Moreover, the researcher that was the moderator of the focus group could have influenced what the developers said. To increase internal validity, we did not participate in the discussions and only asked the questions presented in Section 3.3. Occasionally, we rephrased statements of the developers and asked them to confirm them during the focus group. We did that to avoid misinterpretation of the answers later, which, however, should have had no influence on the given answers. Eventually, hierarchical team structure could have also influenced the discussion. The lead developer participated in the focus group. Other developers may have refrained to give answers that are contrary to the opinion of the lead developer. We tried to avoid that by particularly asking other participants first so that they were free to judge about mutants without an anchoring effect. With the same strategy, we also tried to keep the share of talk as equal as possible among the developers.

4.5.3 | Conclusion validity

Conclusion validity refers to which extent the drawn conclusions are correct.[25] For RQ1, we conclude that traditional mutation testing is more fine granular because it targets particular method calls, return values, weak exception handling, missing boundary checks, and insufficiently tested branches. We consider the number of observations large enough to reach that conclusion. However, we derived our conclusions from observing only the method with its coverage information. For many mutants, however, we cannot derive how the mutant influenced program execution because we had no runtime data. This is an inherent limitation of our research method of manually and statically inspecting mutants and, thus, a threat to conclusion validity.

For RQ2, we drew the conclusion that the number of created mutants and executed tests are roughly the same for pseudo-tested methods. We did this although the traditional approach created 65.6% more mutants than the extreme one for pseudo-tested methods. We considered the effect to be insignificant because the additional mutants only account for 3.6% of all traditional mutants. Further, we also considered the number of executed tests for our comparison where the difference was even smaller and is, thus, in favor of our conclusion. However, it is a threat to conclusion validity that the comparison excludes 90 methods that have no traditional mutants and 92 constructors that were only mutated by the traditional approach. Their impact on the conclusion is considered to be small because these numbers are only fractions of the total number of compared methods and mutants. However, the impact may be larger than assumed. Lastly, excluding the class that was responsible for the majority of executed tests threatens our conclusion that the majority of tests are executed for pseudo-tested methods. We argue that the exclusion was justified because the class was not written with the same coding style as the rest of the classes. By that, we mean that it had disproportional high amounts of if- or return-statements compared with the rest of the analyzed classes. Furthermore, it was responsible for about a quarter of mutants and more than 80% of executed tests, which clearly indicates that this class is an outlier.

For RQ3, we concluded that developers are mainly guided by their priorities which features should be tested well and less by code- or mutant-related reasons when deciding to kill a mutant. We drew the conclusion from our qualitative coding results of the focus group transcript. Our conclusions are affected by the reliability that the given answers are the same as in a realistic development situation. Developers may derive reasons why to kill mutants in a focus group differently than during development. For example, developers have additionally the unit tests

available and write their own code. The focus group where developers need to express why they want to kill a mutant by discussing a preselected set of mutants may not be a similar situation and thus leads to different answers. However, because the answers could be different, our conclusions could be different as well. Thus, it is a threat to conclusion validity that is given by the research method.

4.5.4 | External validity

External validity refers to which extent our findings can be generalized.[25] Since this is a case study, we could test both mutation testing techniques in a very realistic setup. However, results from case studies cannot be easily generalized because those are constrained by the particular case. The Java software project that we used for mutation testing and analysis belongs to the semiconductor industry and is not representative for all Java software projects. For example, we got a comparatively low number of mutants on pseudo-tested methods that result from the `MathMutator`. When mutating a math library, a hypothesis might be that this number could be higher because there are more mathematical operations that can be mutated. Therefore, the expected savings by extreme mutation testing might be larger. An alternative hypothesis might be that the number stays the same because mutants created by the `MathMutator` are more likely to be killed. There is no way to reach either conclusion with our results because we only mutate a single software project. Therefore, we also cannot generalize that our savings by applying extreme mutation testing first are the same for every Java software. For the same reason, we cannot generalize that access modifier and return type are associated with extreme mutation testing verdict in every software project.

The developers that participated in the focus group do not represent the population of all developers. We had to choose the developers of that particular software project and could not select a different, more representative, group. Thus, we can only show their personal motivations why they would kill some mutants and others not. Other developers or teams may have different opinions about similar mutants.

Lastly, the mutation testing tool PIT with its engines Gregor and Descartes does not represent all traditional and extreme mutation testing approaches. We only used the default settings of both engines and have not used any other tool to compare both techniques. We have chosen the default mutators because these are the ones developers would usually choose when they start using mutation testing. We have chosen PIT because it made it easier to compare both approaches. There are other traditional mutators which can be configured to be used. There are other mutation testing tools which work differently compared with PIT. Using different configurations and tools would noticeably change our mutation testing results. Thus, we cannot imply that our results and conclusion are valid for both mutation testing techniques in general.

5 | CONCLUSIONS AND FUTURE WORK

5.1 | Summary of conclusions

We showed that although traditional mutation testing is more fine-grained, it is also mostly inefficient. There exist many similar or redundant mutants. Focusing mutation on particular properties like return values or class fields could effectively reduce the number of similar or redundant mutants and enable easier interpretation of the results. In our case study, the time savings by using extreme mutation testing mainly come from omitting mutants on tested methods and not on pseudo-tested ones. However, knowing the extreme mutation testing verdict leads to ignoring 9% of all traditional mutants. The overhead for applying the extreme before the traditional approach is small. In general, developers prefer not to kill all mutants. The mutants that developers consider relevant are the ones that reveal test issues of important features. Customer focus, personal priorities, team structure, and the established development processes influence the decision which mutants should be killed.

5.2 | Relation to existing evidence

Niedermayr[4,21] showed that pseudo-tested methods were present in 19 open-source projects. We also observed many pseudo-tested methods in our industrial closed-source project. Vera-Pérez et al.[5] showed that extreme mutation testing is faster than its traditional counterpart. We can confirm that and found that it is mainly faster because it saves many mutants on tested methods and not on pseudo-tested ones. Petrović et al.[22,23] already showed that mutation score is not applicable in industrial context and that developers do not want to kill all mutants. We can confirm that result and extend insights why this is the case.

5.3 | Impact/implications

Before applying traditional mutation testing, extreme mutation testing should be performed. Both techniques seem to produce roughly the same number of mutants and executed tests for pseudo-tested methods. We showed that the overhead in terms of execution time appears to be

negligible for the additional extreme mutants for tested methods. However, knowing the extreme mutation testing verdict is helpful because knowing that a method is pseudo-tested is easier to comprehend than analyzing multiple traditional mutants resulting from that method.

It turns out that, similar to the extreme approach, traditional mutation testing can also be strategically performed to derive actionable items that are easy to understand. For example, if all return-statements of a method are mutated and all mutants survive, then the return value of that method is not tested because it does not impact the test outcome. The actionable item is then to test the return value of a method. In fact, this does not have to be limited to mutating only methods. Fields of a class could be similarly mutated. For example, a mutator that keeps the value of a class field constant could show whether a particular field impacts the test outcome. If all tests still pass, then mutation testing shows that the mutated class field does not influence the test outcome. The actionable item would be then to test that the particular class field is correctly used. Since class fields are often used to implement state changes, this mutation strategy should reveal whether state changes are correctly verified.

By manually inspecting mutants, we observed several examples where a method's testing verdict is "tested" but all return-statements were untested. This leads to the follow-up question: If the return value is not responsible for the testing verdict, then what else causes this categorization? The answer is that the extreme approach additionally removes method calls, class field assignments, and raised exceptions which impacts the test outcome. Particularly, exceptions are suboptimal in combination with any extreme mutator. If any test expects that an exception is raised, then applying extreme mutators to that method is almost pointless because it will always kill it. Technically, it is correct that the method is tested because the exception cannot be removed. However, the rest of the emptied method body and return-statements may not be tested. Although strategic application of traditional mutation testing could close this gap, the current `ReturnValuesMutator` comes with another flaw. It raises a `RuntimeException` when a `null` value is returned. Thus, vice versa, if no test expects an exception, then this mutant is always killed, but the conclusion is that the return value is tested. This, however, might not be true because no return value is actually returned.

Therefore, we propose that a method should be mutated with two operation modes in mind: normal and escalation mode. The normal mode is the mode where no exception is raised, whereas the escalation mode raises one. In normal mode, the regular extreme mutators can be applied. These mutators would verify that the method body and the return values are not pseudo-tested. In escalation mode, the exception that is raised should be mutated. We propose that a base class of the exception should be raised to verify that exception handling is correct. If exception handling is too generous, then this change would not be detected. Potentially, it could make sense to categorize the method then as partially tested. Alternatively, both modes could be independently considered, and the mutation testing report could highlight that and even have an additional mutator that removes the exception entirely. For the traditional approach, we do not propose to mutate return-statements that return `null`. Instead, we propose an enhancement to the code coverage report. If a return-statement can return an instantiated object but does only return `null` in all tests, then this should be highlighted in the report. The resulting actionable item is to write a test that returns an object that is not `null`. However, no mutation would be required to highlight that.

In fact, there are other examples where a strategy of recording test execution and analyzing the control flow results in avoidance to create mutants because they are not necessary. One example is the `ConditionalsBoundaryMutator`. To illustrate it, consider an example where the condition `(a <= 42)` should be mutated to `(a < 42)`. If there is only one test case that evaluates the condition where `a = 3`, then there is no point in mutating it. It is evident that the mutant will survive because the control flow does not change. The control flow would only change for `a = 42` because 42 is a boundary value. Thus, if a boundary value is not used, it would make sense to express this in a code coverage report to inform the developer. However, applying mutation testing is only necessary if the boundary values were used, otherwise application of it wastes resources. Similarly, the `NegateConditionalsMutator` only changes the control flow when the whole decision is changed. If `(a || b)` is mutated and only one test exists, where `a = b = true`, negating neither `a` nor `b` changes the control flow. Therefore, we suggest changing the mutator so that it changes the whole decision instead of single conditionals. This means, instead of generating `a || b)` and `(a ||b)`, the mutator should only create `(a || b)`. Otherwise, the control flow may stay the same and the mutant will survive. In addition to that, the proposed mutator should not be applied if the new branch was not executed before. The focus group revealed that, if a branch is not covered by the test suite, the mutant should not execute it.

Although the mutation score is commonly calculated in academia, it is almost meaningless in industry. Each of our results shows these deficiencies. For example, 110 mutants were killed although generated on pseudo-tested methods. When only taking pseudo-tested methods into account, a mutation score of 19% would be the result. However, intuitively, the score should be close to 0% for methods that can be completely removed. The high score is caused by mutants that are easily killed. Prominent examples are mutants that change access to arrays or negate conditions. Changing array access often leads to `IndexOutOfBoundExceptions`. Negating conditions often leads to calling methods of null-objects, that is, `NullPointerExceptions`, and also to raising exceptions which are never intended to be raised. However, having a low mutation score does also not mean that there are a lot of different testing issues. Manual inspection of mutants showed that many mutants are actually similar or duplicates. Many mutants on the same line can be killed with the same test. Most importantly, from conducting the focus group, we know that developers do not intend to kill all mutants but only the ones that are relevant for them.

Developers focus mostly on API features because these affect the customer, that is, the users of their software. Thus, mutants that reveal issues of these features are particularly important. These methods are more thoroughly tested than other methods. Potentially, scarcely covered methods are not even worth to be mutated because they were never really tested in the first place which is already known by the developers.

SUMMARY OF IMPLICATIONS

1. Extreme mutation testing should be applied before traditional mutation testing to filter out pseudo-tested methods and avoid inspection of traditional mutants.
2. Traditional mutation testing can also be strategically performed to derive actionable items that are easy to understand.
3. Method calls that raise exceptions should be differently handled than calls that do not.
4. Control flow analysis and enhancements to code coverage reports can improve application of mutation testing.
5. The mutation score is flawed and should be ignored.
6. Developers prefer not to mutate the whole code but only the important parts.

5.4 | Limitations

Only five developers participated in the focus group which limits our study. Other developers may have different priorities and procedures in place and so the answers might differ. Further, we are limited by the particular software that we tested which is quite large and is only used in the semiconductor industry. Thus, we cannot entirely generalize our results. Other software may produce different results and potentially different insights.

For instance, smaller software projects, that are, for example, often used in microservices architectures, may have drastically different execution times and distributions of traditional mutants across extreme ones. When roughly estimating such a software to be a 10th of the size of this case software, then this would probably reduce the execution time difference from the order of minutes to seconds. This would probably make execution time saving irrelevant but, of course, depends on various factors like the number and duration of tests. Furthermore, software that does a lot of computation may also generate far more mutants that alter mathematical operators. This changes the overall distribution of mutants. We do not know the exact implications that this would have on the time savings and it would be subject to further studies.

However, other findings seem to be very well generalizable and just require further empirical evidence in our opinion. For example, applying extreme mutation testing first to find pseudo-tested methods seems to be viable in general. This spots first testing issues and reduces analysis time of traditional mutants. Enhancing code coverage with further information to avoid mutants to be created in the first place is also not tied to our analyzed case software. Similarly, handling method executions that throw exceptions differently than executions that do not throw them are also more general advises but require further evidence.

5.5 | Future work

Future work should use our findings and combine extreme and traditional mutation testing to one joined mutation testing technique. The new technique should strategically apply the mutants as discussed in this study and validate its usefulness. Additionally, further studies need to increase our external validity. In particular, more studies that show the motivations of developers when to kill a mutant should be conducted. These results could be used to provide guidelines for developers. These could be shipped with the particular mutation testing implementation which should foster industry adoption.

ACKNOWLEDGMENTS

This research was supported by Advantest as part of the Graduate School "Intelligent Methods for Test and Reliability" (GS-IMTR) at the University of Stuttgart. Open access funding enabled and organized by Projekt DEAL.

DATA AVAILABILITY STATEMENT

The authors confirm that the data supporting the findings of this study are partially available within the article. Further data are subject to third-party restrictions and cannot be shared.

ORCID

Maik Betka https://orcid.org/0000-0002-2936-1024
Stefan Wagner https://orcid.org/0000-0002-5256-8429

ENDNOTES

* https://pitest.org/

† https://github.com/STAMP-project/pitest-descartes

‡ https://www.jacoco.org/jacoco/

§ https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.3.3

¶ https://pitest.org/quickstart/mutators/#INCREMENTS

https://docs.oracle.com/javase/tutorial/java/javaOO/variables.html

‖ We ignore the class that was responsible for the majority of executed test cases when we write "all traditional mutants".

** https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

REFERENCES

1. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Trans Softw Eng*. 2011;37(5):649-678. https://doi.org/10.1109/TSE.2010.62
2. Papadakis M, Kintis M, Zhang J, Jia Y, Le Traon Y, Harman M. Mutation testing advances: An analysis and survey. *Adv Comput*. 2019;112:275-378.
3. Niedermayr R, Juergens E, Wagner S. Will my tests tell me if I break this code? In: 2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED) IEEE; 2016:23-29.
4. Niedermayr R. Evaluation and improvement of automated software test suites. *Ph.D. Thesis*: Fakultät 5 - Informatik, Elektrotechnik und Informationstechnik; Pfaffenwaldring 47, 70569 Stuttgart; 2019.
5. Vera-Pérez OL, Monperrus M, Baudry B. Descartes: A pitest engine to detect pseudo-tested methods: Tool demonstration. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE) IEEE; 2018:908-911.
6. Coles H, Laurent T, Henard C, Papadakis M, Ventresque A. PIT: A practical mutation testing tool for java (demo). *ISSTA 2016*. New York, NY, USA: Association for Computing Machinery; 2016:449-452.
7. Vera-Pérez OL, Danglot B, Monperrus M, Baudry B. A comprehensive study of pseudo-tested methods. *J Empir Softw Eng*. 2019;24(3):1195-1225.
8. Betka M, Wagner S. Extreme mutation testing in practice: An industrial case study. In: 2021 IEEE/ACM International Conference on Automation of Software Test (AST) IEEE; 2021:113-116.
9. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empir Softw Eng*. 2009;14:131.
10. Hamlet RG. Testing programs with the aid of a compiler. *IEEE Trans Softw Eng*. 1977;SE-3(4):279-290.
11. DeMillo RA, Lipton RJ, Sayward FG. Program mutation: A new approach to program testing. *Infotech State Art Report, Softw Test*. 1979;2(1979):107-126.
12. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *Computer*. 1978;11(4):34-41. https://doi.org/10.1109/CM.1978.218136
13. Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. *Mutat Test New Century*. 2001:34-44.
14. Siami Namin A, Andrews J, Murdoch D. Sufficient mutation operators for measuring test effectiveness. In: 2008 ACM/IEEE 30th International Conference on Software Engineering ACM/IEEE; 2008:351-360.
15. Papadakis M, Malevris N. An empirical evaluation of the first and second order mutation testing strategies. In: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops; 2010:90-99.
16. Fleyshgakker VN, Weiss SN. Efficient mutation analysis: A new approach ACM; 1994:185-195.
17. Howden WE. Weak mutation testing and completeness of test sets. *IEEE Trans Softw Eng*. 1982;SE-8(4):371-379. https://doi.org/10.1109/TSE.1982.235571
18. Ma Y-S, Offutt J, Kwon YR. Mujava: An automated class mutation system. *Softw Test Verification Reliab*. 2005;15(2):97-133.
19. Just R. The major mutation framework: Efficient and scalable mutation analysis for java. *ISSTA 2014. ACM. Association for Computing Machinery*. New York, NY, USA; 2014:433-436.
20. Schuler D, Zeller A. Javalanche: Efficient mutation testing for java. In: ESEC/FSE, 09. ACM. Association for Computing Machinery; 2009; New York, NY, USA:297-298.
21. Niedermayr R, Wagner S. Is the stack distance between test case and method correlated with test effectiveness? *Proceedings of the Evaluation and Assessment on Software Engineering (EASE '19)*. New York, NY, USA: Association for Computing Machinery; 2019:189-198.
22. Petrovic G, Ivankovic M, Kurtz B, Ammann P, Just R. An industrial application of mutation testing: Lessons, challenges, and research directions. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) IEEE; 2018:47-53.
23. Petrović G, Ivanković M. State of mutation testing at google. In: ICSE-SEIP' 18. ACM. Association for Computing Machinery ACM; 2018; New York, NY, USA:163-171.
24. Corbin J, Strauss A. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*: SAGE Publications, Inc; 2014. ISBN: 978-1-4129-9746-1.
25. Shull F, Singer J, Sjøberg DI. *Guide to Advanced Empirical Software Engineering*: Springer; 2008. ISBN: 978-1-84800-043-8.

How to cite this article: Betka M, Wagner S. Towards practical application of mutation testing in industry — Traditional versus extreme mutation testing. *J Softw Evol Proc*. 2022;e2450. doi:10.1002/smr.2450