On Comparing Mutation Testing Tools through Learning-based Mutant Selection

Milos Ojdanic, Ahmed Khanfir, Aayush Garg, Renzo Degiovanni,
Mike Papadakis, and Yves Le Traon
University of Luxembourg, Luxembourg, Luxembourg
milos.ojdanic@uni.lu, ahmed.khanfir@uni.lu, aayush.garg@uni.lu, renzo.degiovanni@uni.lu,
michail.papadakis@uni.lu, yves.letraon@uni.lu

***Abstract*—Recently many mutation testing tools have been proposed that rely on bug-fix patterns and natural language models trained on large code corpus. As these tools operate fundamentally differently from the grammar-based traditional approaches, a question arises of how these tools compare in terms of 1) fault detection and 2) cost-effectiveness. Simultaneously, mutation testing research proposes mutant selection approaches based on machine learning to mitigate its application cost. This raises another question: How do the existing mutation testing tools compare when guided by mutant selection approaches? To answer these questions, we compare four existing tools – μBERT (uses pre-trained language model for fault seeding), IBIR (relies on inverted fix-patterns), DeepMutation (generates mutants by employing Neural Machine Translation) and PIT (applies standard grammar-based rules) in terms of fault detection capability and cost-effectiveness, in conjunction with standard and deep learning based mutant selection strategies. Our results show that IBIR has the highest fault detection capability among the four tools; however, it is not the most cost-effective when considering different selection strategies. On the other hand, μBERT having a relatively lower fault detection capability, is the most cost-effective among the four tools. Our results also indicate that comparing mutation testing tools when using deep learning-based mutant selection strategies can lead to different conclusions than the standard mutant selection. For instance, our results demonstrate that combining μBERT with deep learning-based mutant selection yields 12% higher fault detection than the considered tools.**

I. Introduction

Mutation testing is considered one of the most powerful testing techniques [6]. It operates by posing the requirement to write tests to reveal artificially injected faults, hence also revealing the real faults coupled with artificial ones [46], [16].

Motivated by its fault-revealing capability and application in various domains such as testing [46], debugging [47], [36], maintenance and change-aware dependability analysis [39], [10], [17], researchers and practitioners have proposed several mutation testing approaches to automate fault injection and test suite assessment. Most approaches inject faults based on predefined syntactic transformation rules (aka mutation operators) [6], [18], such as replacing an instance of a relational operator with another operator, e.g., replacing $>$ with $>=$. Other approaches aim at injecting faults by either following fault patterns created or learned from recurrent fault instances [29], [51], [48] or by employing code pre-trained language models [19]. These approaches have been implemented and made openly available as tools, serving the main purposes of mutation testing – tests assessment and guidance criterion.

Interestingly, while several novel mutation testing approaches and their corresponding tools have recently emerged, their fault revelation potential has not been assessed and compared with the traditional grammar-based mutation testing tools. Since these tools rely on fundamentally different underlying techniques such as manually defined patterns [29], deep learning [51], grammar-based rules [18], and code pre-trained language models [19], it is particularly interesting to check for potential complementarities along with their strengths. Previous studies [32], [35] have limited their studies to only grammar-based mutation testing tools [18], [26], [37]. Hence, we ventured to investigate the effectiveness of the most recent mutation testing tools and contrast their performance with the traditional ones under a new and larger dataset by employing Defect4J *v2.0*.

A. Rationale behind the comparison

Powerful learning-based mutant selection strategies have been proposed recently [23], [15], [28], intending to reduce the application cost and noise of mutation testing, which has been for long considered as a primary cause that keeps the technique away from broad industrial service. These strategies aim at discarding redundant mutants and providing testers with mutants that will bring value to the testing process. Typically, these strategies employ user-defined features - code features learned using deep learning - independent of how mutants are introduced. Since these mutant selection strategies give different importance to mutants, this may affect the cost-effectiveness of mutation testing and raises the question of how the different tools compare in terms of fault detection under mutant selection strategy guidance.

To this end, we model the application cost they entail and perform a controlled cost-effectiveness comparison under two different cost models, which reflect the main efforts spent in mutation testing campaigns. These cost models are repeatedly used for work simulation and encompass the number of analysed mutants and the number of written tests required to reveal the injected faults [46], [28].

Precisely, we study the fault detection ability and the related cost-effectiveness of four fundamentally different mutation

testing tools that we deemed as representatives of different approaches when guided with and without a mutant selection strategy. In particular, we consider 1) IBIR [29] – a mutation testing tool that represents manually crafted fault patterns – 2) DeepMutation [51] – a deep learning-based tool that derives patterns from real bug fixes – 3) μBERT [19] – a mutation testing tool that uses a pre-trained NL-PL language model to replace tokens based on large code corpus learning – and 4) two sets of operators from PIT [18] – a popular grammar based mutation testing tool. As a learning-based selection strategy, we use *Cerebro* [23], a deep-learning-based mutant selection technique that has been proven efficient in reducing mutation testing campaign costs.

B. Contributions

In this study, we hypothesise that different mutation testing approaches - directed by learning-based selection strategies - lead to different conclusions on the fault revelation, raising a risk that their suitability can be misinterpreted. Our results show that IBIR reveals most of the Defects4J faults (approximately 90% of the considered real faults), followed by μBERT (approximately 74%) and PIT (approximately 73%). However, IBIR and PIT introduce significantly more mutants than μBERT; approximately, IBIR produces twice as many mutants as PIT, which produces 3.2 times as many as μBERT. This seems to introduce a size effect on the number of mutants, which influences fault detection. To account for this, we also control the number of mutants (or tests) and perform a cost-effectiveness comparison.

Thus, when cost-effectively comparing the tools, we find that except for DeepMutation, which is the least effective, all tools have similar fault-revealing abilities when controlling cost/effort and applying them out of the box – without any guidance. Perhaps surprisingly, when we combine them with mutant selection strategy, we see a much different picture with μBERT performing significantly better, approximately 12%, than the other tools. Additionally, we find that the other tools subsume DeepMutation by being able to identify more faults and doing it at a much lower cost.

Overall, our work aims to study the fault detection performance of different testing approaches when employing mutant selection strategies. Our key contributions can be summarized by the following points:

1) We perform the first study investigating the fault detection capability of fundamentally different fault seeding approaches (IBIR, DeepMutation, PIT, μBERT) in a newly released bug dataset, i.e., Defect4J *v2.0*.
2) We propose a new way to compare the mutation testing tools using learning-based strategy and show that leads to different conclusions on which is the most cost-effective tool than the ones that could be drawn when not considering it. We investigate the use of transformers, a state-of-the-art deep learning technique *Cerebro*.
3) We show that combining μBERT with learning-based mutant selection yields significantly higher fault detection, approximately 12% higher, than any other tool.

II. Mutation Testing and Mutant Selection

Mutation is a test adequacy criterion representing test requirements by the mean of artificially seeded faults called mutants. Mutants are usually obtained by performing slight syntactic modifications to the original program. For instance, an expression like `x > 0` can be mutated to `x < 0` by replacing the relational operator `>` with `<`. The standard workflow starts by introducing a developer with a mutant to design a test case to *kill* it, i.e., to distinguish the observable behaviour between the mutant and the original program. Some mutants cannot be killed as they are functionally *equivalent* to the original program; thus, they are discarded by the developer. Hence, the thoroughness of a test suite is measured in terms of its mutation score, computed as the ratio of killed mutants over the total number of generated ones.

Mutation testing is a promising, empirically validated software testing technique [46], which is often considered computationally expensive, mainly due to the large number of mutants it introduces, which requires analysis and execution with the related test suites. One may notice that the number of mutants is disproportionate to the number of test cases since one test case can kill several mutants simultaneously. Thus, the effort to analyze and execute mutants that do not help improve test suites is wasted. In order to scale mutation testing, it is of most significance the provision mechanisms that avoid analysing redundant cases. The redundant mutant will always be killed if other mutants are killed with the same test [5]. Plus, redundant mutants provide noise to the overall mutation adequacy score, which consequentially provides the noise into an overall observation about a test suite quality. The reason why redundant mutants contribute to the overall computation cost is that they are numerous, and in order to analyze them, the tests need to be executed [34]. Hence, it is desirable to employ mechanisms that avoid analyzing redundant cases, and analyze only the mutants that add value, i.e., the subsuming ones [25], [33], [6].

Intuitively, subsuming mutants are the minimum subset of all mutants that, when killed by any possible test suite, results in killing the entire set of mutants. For instance, given M_1, M_2 and T two mutants and a test suite, where $T_1 \subseteq T$ and $T_2 \subseteq T$ are non-empty subsets of tests from T that kill mutants M_1 and M_2, respectively. We say that mutant M_1 subsumes mutant M_2, if and only if, $T_1 \subseteq T_2$. For instance, by assuming that $T_1 = \{t_1, t_2\}$ and $T_2 = \{t_1, t_2, t_3\}$, we can notice that every time that we run a test to kill mutant M_1 (t_1 or t_2) we will also kill mutant M_2. Though, the vice versa does not hold. In case $T_1 = T_2$, we say that mutants M_1 and M_2 are indistinguishable. The set of mutants which are both killable and subsumed only by indistinguishable mutants are called *subsuming mutants*.

Suites targeting subsuming mutants lead to a high mutation score since killing all subsuming mutants ends up killing every killable mutant. Since it is impractical to know the subsumption relations between mutants in advance, novel machine learning-based approaches recently emerged that aim

to predict whether a mutant is likely to be subsuming or not, given the surrounding code in which the mutation occurs [23]. Hence, when mutants guide the testing process, it is possible to use these machine learning model predictions to prioritize the selection of (likely) subsuming mutants, among others that are more likely to be redundant.

III. Research Questions

This study aims to compare the cost-effectiveness of the recently proposed mutation testing tools. To do so, we start our analysis by investigating the fault detection ability of the studied tools in a scenario when a developer writes a test that distinguishes a mutant and identifies coupled fault. Thus we ask:

RQ1 *(Tool's effectiveness) What is the fault detection ability of IBIR, DeepMutation, PIT and μBERT mutation testing tools? How do the employed techniques compare in terms of cost-effectiveness?*

The answer to this question allows us to identify the most effective and cost-effective tools in standard comparison settings with random mutant selection, which have merits in deciding on their use and shedding light on their strengths.

Intelligent mutant selection strategies have recently been proposed to prioritize mutants and reduce the mutation testing effort. Hence, our other objective is to investigate whether the cost-efficiency of fault-seeding approaches would take advantage similarly by these strategies. We consider an advanced learning-based mutant selection *Cerebro* aiming to select subsuming mutants utilizing Neural Machine Translation proficiency. We, therefore, investigate the following research question:

RQ2: *(Learning-Based Selection) What is the cost-effectiveness of mutation testing tools when mutants are selected according to a learning-based mutant selection strategy?*

Answering these questions provides evidence of whether and how much the mutation testing tools/approaches benefit from the intelligent mutant selection and whether there is one that benefits more than the others.

Taken all together, by answering the above questions, we study the fault detection ability of fundamentally different mutation testing tools and estimate their cost-effectiveness when guided by mutant selection strategy.

IV. Mutation Tools and Selection Strategies

A. Mutation Testing Tools

PIT [18] is one of the state-of-the-art mutation testing tools that seeds faults using syntactic transformation rules (aka mutant operators) at the bytecode level. We selected PIT as a representative of tools for grammar-based transformation since it is considered a state of art tool with a vast community providing continuous support. Besides, recently we have witnessed many empirical proofs and studies distinguishing the tool of its competitors [35]. The tool implements 29 task-specific categories of mutation operators; for instance, the Conditionals Boundary category mutates relational expressions. When considering the 29 categories, PIT has over 120 mutation operators. However, PIT also provides different pre-defined configurations. Thus in this study, we consider two. We will denote by PIT to the setting in which all mutation operators from the 29 categories are considered and by *PIT_Default* to the set of mutants contained in the default configuration of the tool - consisting of 11 categories. The tool default configuration is often used in industry settings, while many existing studies employed all mutants for experimental purposes. We decided to take both configurations for our study to dismiss the threat of biasing the tool. We provide a code snippet demonstrating the mutation induced in the following box.

```
//PIT uses grammar transformations - e.g., relation > to <=
  public boolean contains(final Object object) {
        return indexOf(object) <= 0;
    }
```

μBERT [19] is a mutation testing tool that uses a pre-trained language model (CodeBERT) [21] to generate mutants by masking and replacing tokens. μBERT takes a Java class and extracts tokenized expressions, which mask for token replacement (mutation), e.g., it masks a variable name and invokes CodeBERT to complete the masked sequence (i.e., to predict the missing token). This approach has been proven efficient in increasing the fault detection of test suites [19] and improving the accuracy of learning-based bug-detectors [49]; therefore, we consider it as a representative of pre-trained language-model-based techniques. For instance, please consider the code snippet provided, in sequence `return indexOf(object) > 0;` μBERT mutates the method invocation expression `indexOf` by feeding CodeBERT with the masked sequence `return <mask>(object) > 0;`. CodeBERT predicts the 5 most likely tokens to replace the masked one, e.g., it predicts `contains`, `indexOf`, `lastIndexOf`, `count`, and `size` for the given masked sequence. μBERT takes these predictions and generates mutants by replacing the masked token with the predicted ones (per masked token creates five mutants). μBERT discards non-compilable mutants and those syntactically the same as the original program (cases in which CodeBERT predicts the original masked token).

```
//mBERT uses CodeBERT to alter tokens based on the context
  public boolean contains(final Object object) {
        return lastIndexOf(object) > 0;
    }
```

IBIR [29] is a fault seeding tool that uses automatic program repair inverted fix-patterns to inject faults that are similar to real ones. It takes as input the git repository of the program to mutate and a bug report, written in natural language and seeds (introduces) multiple fault candidates (mutants) that emulate the fault described in the bug report.In particular, IBIR's mutation operators are inverted fix-patterns crafted from actual bug fixes, and their inverse would induce seeded faults that are similar to actual faults. IBIR, in one of its configurations, applies faulty patterns exhaustively over

the system-under-test to generate mutants without any bug-report IRFL guidance. We use this configuration in our study to exclude the advantage brought by the IRFL component to IBIR's performance and, thus, make a fair comparison between the considered approaches mutations. For instance, if you consider the code snippet provided, in sequence `return indexOf(object) > 0;` IBIR can mutate the condition by expanding the expression with an extra one `&& object == null`.

```
//IBIR uses inverted fix-patterns
  public boolean contains(final Object object) {
        return indexOf(object) > 0 && object == null;
    }
```

DeepMutation [51] generates mutants by employing Neural Machine Translation [53], aka NMT. It uses an NMT model trained on a large corpus (∼787k) of existing bug-fixing commits mined from GitHub repositories. It takes a Java method as input and outputs a mutant. Hence, it generates one mutant for every method in a Java class file. In particular, every method is abstracted, in which pre-defined identifiers replace the user-defined variable names and literals to obtain an abstracted code representation. These abstracted code representations are then given as input into the trained NMT model to produce abstracted mutants, which are converted back to source-code mutants by reversing the abstraction.

We use the publicly available trained model of DeepMutation [1] to generate the mutants and `src2abs` [4] tool to perform the abstraction process. This approach is one of its kind until this moment, and we followed its guidelines [51] to generate one mutant per method.

```
//DeepMutation uses Machine Translation for bug-fixing
  public int contains(final Object object) {
        return indexOf(object);
    }
```

B. Mutant Selection Strategies

The fault seeding techniques generate a very different number of mutants. Thus, to make a fair comparison, we aim to control the number of mutants in answering RQs 1 and 2. Since the order in which mutants are analyzed is relevant due to the existence of equivalent and trivial mutants and can affect the application cost of mutation testing, it can also alter the cost-effectiveness of the tools/techniques used. Thus, we consider two mutant selection strategies that are very different from each other.

Standard Mutant Selection consists of sampling uniformly from the entire set of mutants [34], [32], i.e., every mutant has the same probability of being selected since no prioritization heuristic is considered.

Cerebro [23] is a machine learning approach that has been shown effective in statically selecting *subsuming* mutants. *Subsuming* mutants - a minimal subset of mutants to identify such as to identify the original set reciprocally [30], [44] - are the set of mutants that resides on the top of the subsumption hierarchy and subsume all other mutants [33]. *Cerebro* learns to identify subsuming mutants given their context. In particular, it learns the associations between mutants and their surrounding code by using language-agnostic *Neural Machine Translation* [13], which is also used by many recent studies [53], [22], [52], [50]. *Cerebro*'s learning scope is a relatively small area around the mutation point that differentiates locally the mutants that are subsuming from those that are not. This procedure allows the selection of the mutants from program elements which fit best to their context rather than using entire codebases with every possible transformation. *Cerebro* demonstrated preserving the mutation testing benefits while limiting application cost, i.e., reducing all cost application factors such as equivalent mutants, mutant executions, and the mutants that require analysis. *Cerebro* outperformed other approaches that concern machine learning models that capture code properties through manually engineered code features [23].

V. Experimental Design And Analysis

A. Benchmarks and Ground Truth

We use Defects4J [27] v2.0.0, which contains the build infrastructure to reproduce (over 800) real faults for Java programs. Every bug in the dataset consists of the faulty and fixed versions of the code and a developer's test suite accompanying the project that includes at least one fault-triggering test that fails in the faulty version and passes in the fixed one.

The set of faults spans more than a decade of development history, making it challenging for us to synchronize the execution of faults over different fault-seeding tools, following obsolete dependencies not supported by relatively recent tools and old versions of frameworks or languages, i.e., some of the mutation tools require Java 1.8+. Thus, intending to be as fair as possible with the selected tools, we had not considered those faults that did not satisfy the building requirements — specifically, the 26 faults from the project Jfreechart and 174 from Closure-compiler. Additionally, when conducting this study, we found that 82 faults from the Jsoup project were not compilable due to technical reasons [2]. In total, we analyzed 509 faults from 15 different projects.

It is pertinent to note that when comparing and observing performance between different tools, we strictly use the intersection of faults, i.e., the faults we were able to study for all tools in question, and strictly those faults where every tool generated at least one killable mutant.

B. Generated Mutants

For each selected faulty project version from Defects4J, we start by identifying the modified classes between the faulty and fixed versions. Then we generate mutants for the fixed version of each modified class by employing the selected mutation testing tools. Table I records the number of faults analysed and the number of mutants generated by each mutation testing tool. DeepMutation delivers only one mutant per method and produced 5,559 mutants for the 348 analysed faults. μBERT was applied on 499 faults and produced 293,304 mutants. IBIR produced 1,113,113 mutants for the 393 analysed faults. As we previously introduced, we consider two configurations

in the case of the PIT mutation testing tool. PIT_Default uses the subset of the mutation operators as specified in the tool's production-ready setup. These categories are considered the most effective ones (11 out of 29) and generate 110,480 mutants for 508 faults analysed. For the sake of thoroughness of the study, as we already mentioned, we also use *all* available mutation operators of the tool, denoted by PIT, and generate 1,212,544 mutants across 29 mutants categories for the 509 faults analysed.

TABLE I: Number of Faults and mutants used in the study.

Mutation Testing Tool	# of Analysed Faults	# of Mutants
DeepMutation	348	5,559
PIT_Default	508	110,480
μBERT	499	293,304
IBIR	393	1,113,113
PIT	509	1,212,544

* *When comparing different tools, we strictly use the intersection of faults*

C. Experimental Analysis Procedure

We start by executing all the mutants generated by the different tools on the selected project subjects and recording the failing tests distinguishing those mutations. Next, we use *Cerebro*, the machine learning approach, to obtain the (subsuming) probability associated with each mutant needed for answering RQ2.

The procedure to answer RQ1 studies the cost-effectiveness of the fault seeding techniques when employing standard (random) mutant selection as the strategy of selecting mutants in a developer work simulation. We repeat the procedure for RQ2; however, this time, *Cerebro* guides the selection of mutants by prioritising mutants and assigning the highest probability of being useful to those likely to subsume others, considering their surrounding code context.

In particular, the standard developer workflow simulation emulates a testing scenario where the mutants guide the testing process and serve as test requirements. A tester selects mutants and designs tests to kill them until every (killable) mutant is killed (a standard simulation often reported in the literature [46]). Intuitively, the work simulation starts with an initial empty test set and the set of mutants to be covered. The next step is to select a mutant with high priority given by some strategy and either, select randomly a test (without replacement) that kills it, or judge it as equivalent. Each selected test is added to the test suite, and every mutant killed by that same test is discarded. The simulation is repeated until all mutants are treated.

Precisely, given a list `M` of mutants sorted by a particular mutant selection strategy (i.e., Standard or *Cerebro*) and their predefined test pool `P` (provided within the dataset Defects4J), we incrementally construct and measure the number of tests in test suite `T` required to distinguish every (killable) mutant from `M`, likewise measuring the number of analyzed mutants (killed or judged equivalent) during the process. The simulation starts by picking the top mutant `m`, according to the selection strategy used, among survived mutants (initially considering all mutants from `M`). Next, we check if there exists some test in the test pool `P` that kills `m` (this process simulates a tester picking, analyzing, and designing a test to kill a mutant). If no test kills a mutant `m`, we judge it as equivalent and remove it from `M`. Otherwise, we randomly pick one test `t` from the pool that kills `m`, add `t` to the suite `T`, and remove from `M` every mutant that is killed by `t`. This process continues by taking the next surviving mutant from `M`, finding a test `t` to kill it, and repeating until every mutant in `M` is killed (or judged as equivalent).

In order to perform a more complete and fair comparison between the tools, we measure the *cost* of a mutation testing tool in two ways: The *number of tests* designed/written to kill all (killable) mutants [34], and the *number of analyzed mutants* during the process [32].

Furthermore, it is necessary to note that we consider the *effectiveness* of a mutation testing tool as the ability to devise a test suite `T` to *detect the real fault*. That is, we measure whether, by running forged test suite `T` on the faulty version of the program, we could detect the real fault.

To answer RQ1, we run previously described simulation by *randomly* sorting the list of mutants from the different mutation tools and comparing their effectiveness when applying the same effort, i.e., how many faults we can find when writing the same number of tests or analyzing the same number of mutants. To answer RQ2, we run the same simulation and comparison, but with the mutants prioritized according to *Cerebro*'s importance prediction.

Since our simulation process includes some random effects (e.g., which test `t` is selected to kill a mutant `m`), we repeat this process 100 times for all approaches to reduce the threat of randomness [9].

Overall, this experimental setup promises to investigate the performance and usability of the studied mutation testing tools when applied together with mutant selection strategies.

D. Cerebro Mutant Selection Prediction Performance

To use the machine-learning approach *Cerebro* [23] and select (subsuming) mutants when addressing RQ2, we need to train it on our data set. We follow the guidelines of Garg et al. [23] to implement *Cerebro*'s approach and perform training. Garg et al. employ a 5-fold cross-validation to evenly split the benchmark into five parts, providing five models to obtain probabilities. To evaluate the performance of *Cerebro* on our dataset, we repetitively use one-fold of our benchmark for testing and 4 for training. Table II reports the average prediction performance of our implementation of *Cerebro*, which is comparable with the results of Garg et al. [23] when trained on PIT mutants. When we train it on μBERT mutants, we observe better prediction performance indicators (10% in Precision, 17% in Recall, and 13% in MCC) than trained on PIT mutants. When training *Cerebro* on IBIR mutants, we obtain slightly worse prediction performance than when trained on other tools (3% and 16% lower MCC w.r.t to PIT and μBERT) (Note that since DeepMutation produces only one mutant per method, no mutant prioritization is required).

TABLE II: Prediction Performance of *Cerebro*.

***Cerebro* trained on:**	**MCC**	**Precision**	**Recall**
μBERT	0.56	0.81	0.52
IBIR	0.40	0.84	0.25
PIT	0.43	0.71	0.35

* Cerebro *maintains similar performance as reported by Garg et al. [23]*

For the sake of clarity, it is pertinent to note that column *Precision* describes the ratio of mutants truly subsuming among all the mutants predicted as subsuming, while column *Recall* is the ratio of mutants correctly predicted as subsuming among all the subsuming mutants. The column MCC (*referring to Matthews Correlation Coefficient*) [38] denotes the coefficient between 1 and -1. An MCC value of 1 indicates a perfect prediction, whereas a value of -1 indicates a perfect inverse prediction, i.e., a total disagreement between prediction and reality. An MCC value equal to 0 indicates that the prediction performance is equivalent to random guessing.

E. Statistical Analysis

To evaluate whether fault detection under the same invested effort is significantly different between techniques, we use the non-parametric effect size measure Vargha and Delaney A_{12} [54]. Intuitively, A_{12} measure will tell us how frequently one tool obtains better indicators than the others. It returns values between 0 and 1, where A_{12} = 0.5, showing that the two measures are completely equivalent; otherwise, they have some differences.

VI. Empirical Evaluation

*A. **RQ1: Tool's effectiveness under Standard Selection***

We start our analysis by examining the effectiveness of the mutation testing tools/techniques under the standard mutant selection strategy.

TABLE III: **Cost-effectiveness comparison under different mutant selection strategies (RQ1 and RQ2).**
Each cell represents the absolute difference in fault-detection between the **Observed Tool** and the **Baseline** (-/+ for lower/higher fault detection) when the same effort is invested ($\#M$ stands for the same number of mutants analysed, and $\#T$ stands for the same number of tests written). For instance, using Standard Selection (RQ1) IBIR detects, on average, 14.50% more faults than DeepMutation, when analysing the same number of mutants.

RQ1: Standard Selection

Observed Tools	**Comparison Baseline Tools**							
	DeepMutation		PIT_Default		μBERT		PIT	
	$\#M$	$\#T$	$\#M$	$\#T$	$\#M$	$\#T$	$\#M$	$\#T$
PIT_Default	15.52	1.95	—	—	—	—	—	—
μBERT	16.68	2.57	-0.64	0.72	—	—	—	—
PIT	12.30	2.28	-7.42	-0.65	-7.59	-2.84	—	—
IBIR	14.50	3.79	-3.66	2.12	-0.50	1.15	11.66	3.06

RQ2: Learning-Based Selection (Cerebro)

Observed Tools	**Comparison Baseline Tools**					
	PIT_Default		μBERT		PIT	
	$\#M$	$\#T$	$\#M$	$\#T$	$\#M$	$\#T$
μBERT	12.14%	3.30%	—	—	—	—
PIT	-2.09%	-2.45%	-10.42%	-3.64%	—	—
IBIR	-1.78%	-2.39%	-7.06%	-0.73%	5.77%	-0.70%

** Columns correspond to columns in the grids of Figures 1 and 2*

Figure 1 visualizes fault detection concerning the number of analyzed mutants and the number of written tests. It is pertinent to note that the selection number is controlled since different observed tools generate different numbers of mutants. Hence, when studying the detection of each fault, the maximum cost is directed by the tool that produces the least number of mutants, more precisely, which requires the least effort to analyse all of its mutants. Table III summarises the differences in fault detection of the tools involving the same effort. Let us consider from Sub-table (RQ1) in Table III, the first column – DeepMutation. This column summarises the fault detection difference between the tools observed in the two left Sub-figures of 1, in which DeepMutation is considered the reference tool that limits the maximum cost of the simulation (as its mutants require the least effort to be all analysed, in the majority of the cases). The sub-columns $\#M$ and $\#T$ values report the fault detection advantage (or disadvantage) of using a tool from the first column instead of using DeepMutation, when spending the same effort in terms of respective mutants analysed and tests written. For instance, the first row indicates that PIT_Default (29.54%) can detect 15.52% more faults than DeepMutation (14.02%) when analyzing the same number of mutants while detecting near 2% more faults when writing the same number of tests. Hence, we use different baselines to present our results in Table III and Figure 1, sorted in ascending order according to the number of mutants generated by each tool: DeepMutation, PIT_Default, μBERT and PIT.

We observe that DeepMutation is the least cost-effective technique - other tools detect between 12% and 17% more faults when the same number of mutants is analyzed. Moreover, we notice that the rest of the tools require the analysis of fewer mutants than DeepMutation (around four times less) to reach the same fault detection. The differences are statistically significant. We also compared them with the Vargha-Delaney A measure *($\hat{A}_{12}$) [54], showing that other tools achieve better fault detection on average in 99.6% cases.*

We can also observe that the effectiveness of PIT_Default (58%), μBERT (57%) and IBIR (54%) is similar, outperforming PIT (50%) when analyzing as many mutants as PIT_Default. When writing the same number of tests, we observe that μBERT reaches similar effectiveness (54%) as PIT_Default, PIT 53% and IBIR 56%.

When we focus on μBERT, we can observe that both μBERT (73.2%) and IBIR (72.7%) are more effective than PIT (65.7%) under the same effort. These differences also have statistically significant p-value, and $\hat{A}_{12}$ when compared to their cost-effectiveness, evidencing that μBERT and IBIR in $\approx$ 99% cases can detect more faults. Moreover, to reach the same effectiveness as PIT, μBERT and IBIR need to analyze 44% and 38% fewer mutants than PIT. When we compare the number of tests, we observe that IBIR (65.2%) and μBERT (64.01%) can detect near 4% more faults than PIT (61.16%) under the same effort.

Finally, we can observe that IBIR (near 90%) is more effective than PIT (74%) when the same number of mutants are analyzed. IBIR needs to analyze 80% fewer mutants (and 25% fewer tests) than PIT to reach the same effectiveness. This difference is also statistically significant *p-value<0.01*.

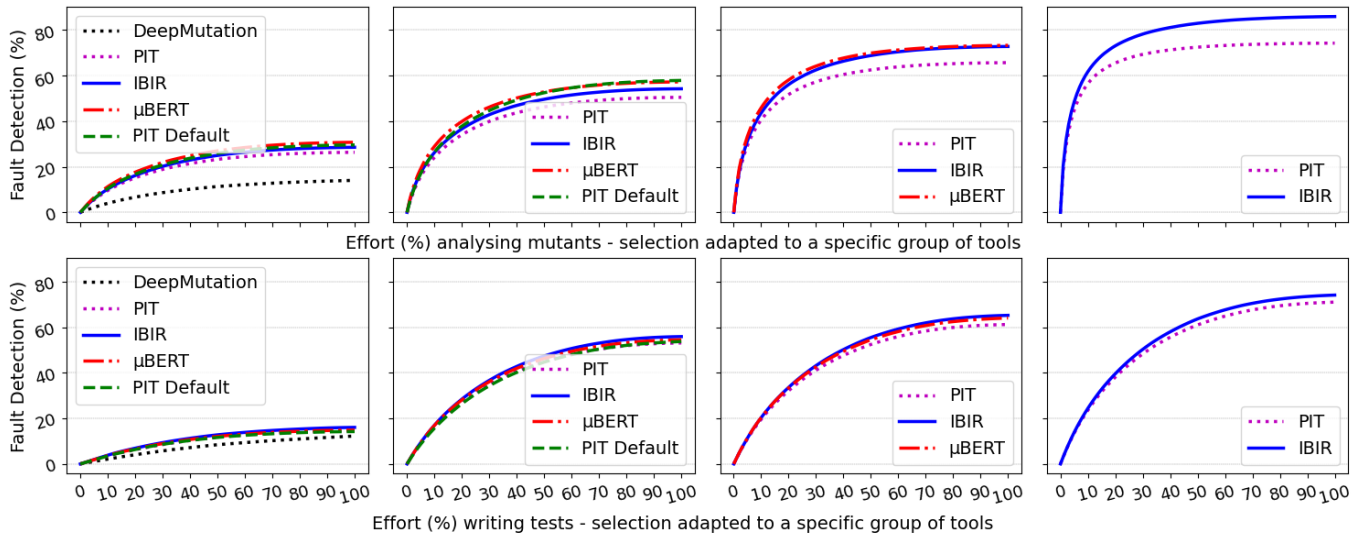

Fig. 1: **RQ1 Standard Selection:** tools' **cost-effectiveness** in fault detection over **different effort models** – analysing mutants / writing tests. **Different groups** of tools control the number for selection to address differences in the scope of mutant generation.

IBIR is the most effective tool, identifying on average ≈ 90% of real faults. It requires the analysis of 80% fewer mutants (and 25% fewer tests) than PIT to reach the same effectiveness. In terms of cost-effectiveness µBERT, IBIR, PIT_Default perform similarly, with PIT performing slightly worse. All other tools subsume DeepMutation concerning fault detection through standard selection.

B. *RQ2: Tool's effectiveness under* Cerebro

Figure 2 visualize cost-effectiveness simulation of the fault seeding techniques when we use a machine learning-based approach, i.e., *Cerebro* for mutant selection. It is important to note that due to findings in the previous research question, we don't find it suitable to consider DeepMutation in this research question since it is subsumed even when using all its mutants.

As can be seen in Figure 2, learning-based mutant selection improves the remaining fault-seeding techniques' cost-effectiveness. Table III in RQ2 cells presents the absolute differences in fault detection between the tools.

Interestingly, when using *Cerebro* to select mutants, μBERT achieves ≈12%, ≈14% and ≈13% higher fault detection rate than PIT_Default, PIT and IBIR, respectively, when analyzing the same number of mutants and ≈4%, ≈6% and ≈6% when analysing the same number of tests. The differences are statistically significant, according to the computed p-value ($<$ 0.01). We have also validated these findings by computing the ($\hat{A}_{12}$) measure, showing that it achieves higher fault detection on average in 96.2% cases. Surprisingly, μBERT analyses 50%, 82%, and 78% fewer mutants than PIT_Default, PIT and IBIR, respectively, to reach the same effectiveness. Concerning the number of tests, the difference is also noticeable when compared with PIT, since μBERT obtains ≈5% higher fault detection under the same number of tests written.

Interestingly, in the presence of the learning-based mutant selection strategy, IBIR keeps its significantly high difference in fault detection when analyzing the same number of mutants as PIT, ≈6%. This difference is also statistically significant with *p-value<0.01*.

Overall, our results indicate that using learning-based mutant selection significantly impacts the cost-effectiveness of the fault-seeding techniques. This observation promises to impact how we use fault-seeding techniques and how we compare new fault-seeding techniques' effectiveness.

µBERT has significantly improved its performance under the machine-learning-based selection strategy, i.e., Cerebro, *becoming the most cost-effective fault seeding technique. µBERT needs to analyse 50%, 82%, and 78% fewer mutants than PIT_Default, PIT and IBIR, respectively, to reach the same effectiveness when selecting mutants according to* Cerebro. *IBIR, PIT_Default and PIT all experience improved performance but overall, they all perform similarly.*

VII. Discussion

A. *Mutant Selection or not: How does the mutant selection impact the mutation testing tools?*

Regardless of the fundamental differences between the considered approaches, our results show they are all efficient in guiding testing towards higher fault detection capabilities. In fact, with relatively low efforts, they score comparable fault detection rates. Their difference becomes noticeable only by spending extra efforts or leveraging a mutants selection strategy to spare the efforts lost in analyzing irrelevant mutants - as can be seen from our results in RQ2. While in RQ1, under the standard mutant selection strategy, we do not report any statistically significant difference ($p<0.05$) between the tools, except in the cases where DeepMutation is subsumed and μBERT and IBIR deviate from others. Although we observe that IBIR is the most effective when considering extra effort in analyzing mutants - with statistically significant differences. The reasoning is that a) IBIR provides mutants with high fault detection capabilities but b) produces numerous equivalent and irrelevant ones, which increase the cost to the target.

The impact is revealed when we applied a learning-based mutant selection approach in RQ2, which deflates mutant

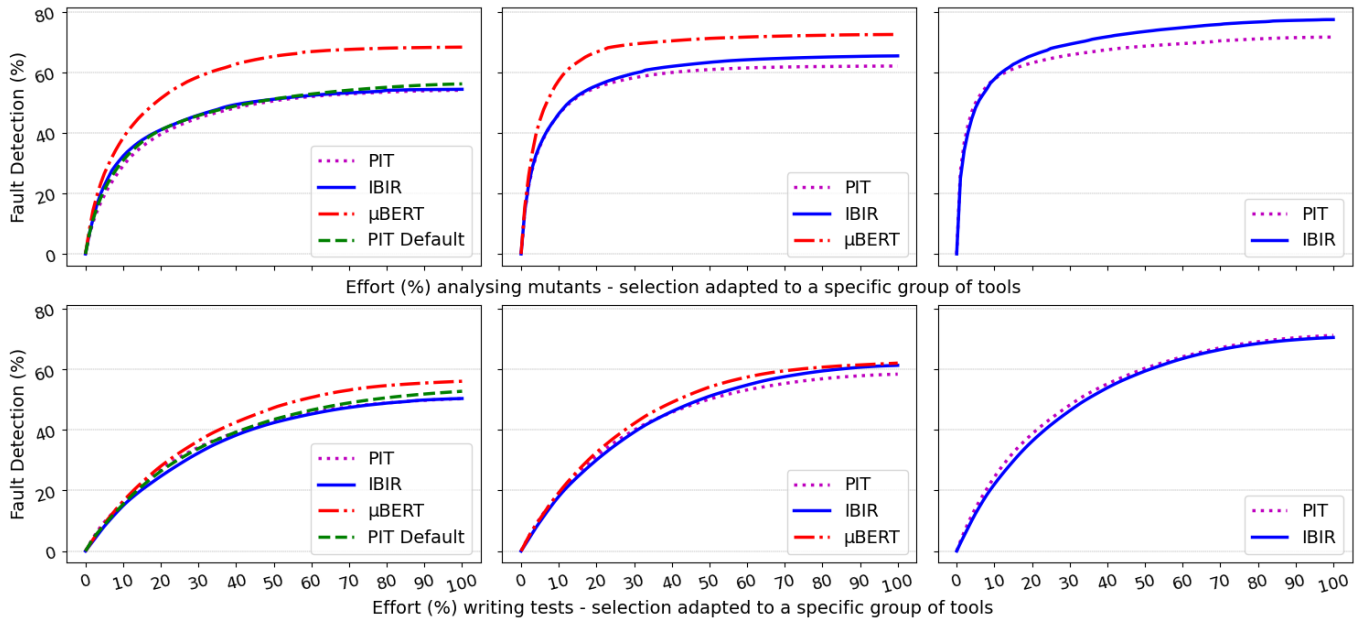

Fig. 2: **RQ2 Learning-Based Selection:** tools' **cost-effectiveness** in fault detection over **different effort models** – analysing mutants / writing tests. **Different groups** of tools control the number for selection to address differences in the scope of mutant generation.

redundancy. Learning-based selection *Cerebro* makes μBERT the most cost-effective, significantly outperforming the other approaches, which differs from standard selection conclusions. We argue that *Cerebro* boosting μBERT, particularly, unlike the others, lies in the similarity and homogeneity among its mutants, which all replace one token with another considering the code context. In contrast, others may remove or alter multiple tokens or statements, making the learning task harder.

To further investigate this variance in cost-efficiency between the studied techniques, we should check whether the other approaches could also get boosted by classification techniques like μBERT. This question is particularly interesting in the case of IBIR, which introduces significantly more mutants, thus, more subsuming and subsumed mutants than μBERT, which challenges any classifier. To check this hypothesis, we plan in future work to construct and study more classifiers (learning-based or not) together with a perfect classifier (an artificial model that perfectly predicts whether a mutant is subsuming or not) and thus obtain more insights about the cost-effectiveness of the available fault-seeding techniques.

Altogether, we conclude that when comparing mutation testing techniques, the standard mutant selection strategy can lead to incomplete conclusions as the most cost-effective tool/technique is not necessarily the same under learning strategies which we encourage researchers and practitioners to utilize and explore further.

So far, the investigation also implies that some operators of different approaches provide beneficial ingredients that should be considered and further explored as complements to mutation testing tools. In the following, we scratch the surface and pave the way for future researchers towards this direction.

B. *Complementarity of the approaches*

So far, we have elaborated on the cost-effectiveness of fundamentally different approaches when guided – or not – by intelligent selection. However, we haven't investigated and given insights into "how?" and "what?" an approach can and cannot reveal, and consequently, 1) what could be the added value of spending more effort using each approach and 2) whether the approaches could complement each other.

As a first step in this direction – as encouragement for future studies since a thorough breakdown may undermine the intent and scope of this work – we amend our quantitative study with a qualitative one. We investigate the bugs that each approach can find – at least once among our simulation repetitions – and distinguish the ones that could not be found by all approaches. Then, we examine these bugs with their revealing mutants and discuss the particularities and shortages, i.e. the fault injection patterns that make any difference between all considered approaches.

The Venn diagram in Figure 3 depicts the distribution of the bugs that are revealed by each tool. Same as per previous results, IBIR outperforms all approaches in terms of fault detection capability, finding 99,57% of the target bugs followed by μBERT (92,7%), PIT (87,55%), PIT_Default (85,83%) and finally DeepMutation (41,20%). In fact, IBIR can discover all the target bugs except one (Mockito 5), which no approach can find, indicating the pseudo-completeness of its mutation operators. Indeed, 2 bugs are only found by IBIR – Math 12 and Mockito 33 – mainly thanks to patterns' power, such as removing or inserting new statements, replacing method invocations, and adding extra conditions.

Concerning μBERT mutations, even if they do not remove or insert new code but only change one token by another, they can reveal 17 bugs which only IBIR could find. Additionally,

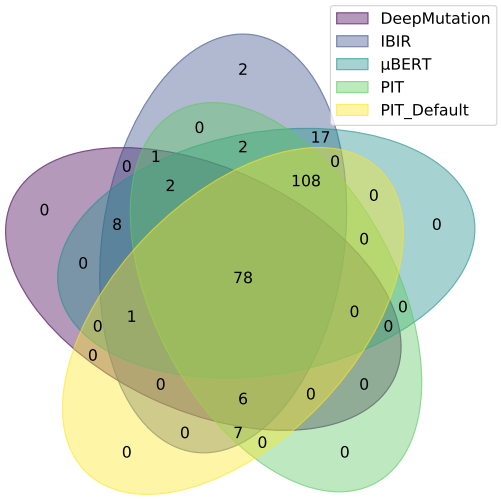

Fig. 3: Which bugs are revealed by every approach? IBiR is capable to find almost all (99,57%) bugs followed by μBERT (92,7%), Pit (87,55%), Pit-Default (85,83%) and finally DeepMutation (41,20%).

μBERT finds respectively 26 and 29 bugs that mature and sophisticated operators of PIT_Default and PIT missed. We explain this by the fact that μBERT's pre-trained model CodeBERT and its knowledge of code (the information it retrieves from the code) to mutate perms it to propose real-like code replacements, thus, real-like mistakes and bugs, i.e. changing method calls, access to objects' fields and arrays etc.

PIT and PIT_Default yield comparable results, showing they can amend μBERT capabilities in finding respectively 13 and 14 bugs more, thanks to patterns that involve multiple tokens changing, i.e. the removing mutation operators.

Overall, we believe that future research should investigate the appropriate joint use of IBIR's inverted fix-patterns, well-crafted and mature PIT_Default grammar transformations, and μBERT's code and context-knowledge based mutations.

C. *Implications for practice*

Over the last decade, mutation testing (a.k.a. fault seeding, fault injection) has been used exhaustively for testing, debugging, maintenance, change-aware dependability analysis, test assessment, etc. In the context of mutation testing, recent industrial applications less often include the generation of all mutations or coverage-adequate test sets. Cheaper trends concern the effort required, and the risk of unrealistic test requirements is seen through the objective of equivalent mutants. Instead, industrial applications are interested in obtaining a curbed sample of mutants that reliably mimic real faults. Thus, satisfying the famous mutation testing proverb: "Do fewer, do smarter, do faster" [41].

At the same time, with the wave of open possibilities brought by machine learning models, many tools emerged. Their diversity provokes a topic of interest in the development and research circles – together with empirical comparisons – about which approach/tool is more efficient in emulating and revealing actual bugs. However, as we showed in the paper, solely comparing tools based on their mutant generation degree does not lead to solid conclusions. Each technique brings additional value, with an inevitable cost in noise. Thus, with this study, we aspire to spread the message to practitioners to consider the actual cost of every technique expressed through some form of intelligent selection and effort analysis, thus discarding the noise. Furthermore, we shed light on different degrees of redundancy that different approaches carry, transformations that make them distinguishable and complimentary, making them less costly. Altogether, the central insight of our study to future researchers and tool developers is to consider appropriate selection strategies when comparing and developing fault-seeding techniques. Moreover, we encourage researchers to explore the combination of models to identify promising locations for mutant generation and joint transformation rules.

> *When comparing mutation testing techniques or tools, it is imperative to account for a mutant selection technique suitable for this purpose. The use of standard mutant selection entails a risk of drawing incorrect conclusions.*

VIII. Threats to Validity

External Validity: To reduce threats that may relate to the subjects we used, we selected 509 faults from 15 mature open-source real-world projects that are well maintained and tested from Defects4J *v2.0*. As we already discussed, while conducting our experiments, we could not compile or run the tests of all the versions available in Defects4J *v2.0*. Although our evaluation expands to many faults and Java projects of different sizes, the results may not generalize to other projects or programming languages.

Another external threat lies in the tools' specificity and running configurations we consider. To reduce this threat, we employ fundamentally different modern mutation tools and run them exhaustively using their corresponding default configurations, generating as many mutants as the tool can generate for each subject.

Another threat can be related to the mutant selection strategies used in the study. To reduce this threat, we consider two fundamentally different approaches. Nevertheless, we do not remove the threat that results can change when another mutant selection technique is employed or considering another language (e.g. *Cerebro* [23] was also evaluated on C programs, which was not explored in this paper) - yet we plan to follow this line of work in the future.

Internal Validity: Threats to internal validity may arise in how we train the machine learning-based approach *Cerebro* for RQ2. To address this threat, we strictly follow the guidelines reported by Garg et al. [23] and explain the steps in Section V-D. In contrast to Garg et al. [23] that evaluated *Cerebro* only with mutants generated with PIT, in this study, we also train *Cerebro* on mutants generated with μBERT and IBIR. Other threats may relate to how we label mutants as subsuming. To counter this threat, we rely on the developer suites provided by the Defect4J benchmark, as it is regarded as the most detailed dataset of faults from projects with thorough test sets in Java language. Any weakness in the suites may lead to

incorrect labelling for mutants, introducing some noise that can affect *Cerebro*'s prediction abilities. Unfortunately, we could not compile and run the master-branch [3] of DeepMutation. Thus, we had to operate the tool from the resources and pre-trained artefacts provided in the repository.

Another threat to internal validity may be that we generate mutants only for the class fixed in the bug-fix pairs provided by Defects4J. Thus, we do not reduce the potential threat that the results do not apply to mutants from other classes interacting with the mutants used in this study.

Construct Validity: Our assessment metrics, number of analyzed mutants, number of written tests and fault detection may not entirely and exhaustively reflect the actual testing cost / effectiveness values. These metrics have been suggested by literature [46], [7], [34] and are intuitive, i.e., the number of analyzed mutants and the number of tests essentially simulate the manual effort involved by testers when mutants guide the testing process [6], [45]. These two cost models illustrate objective comparison as the engineering effort is assumed to be fixed, which would not be the case in a human study, fluctuating based on a participant's experience. While measuring real execution time (computational effort) would be impacted by the environment, e.g., the number of machines, machines' performance, scheduling algorithms and maturity of the tools in a sense if they have built-in support for multi-threading/parallelism.

At the same time, fault detection is the effectiveness metric of interest in this study that can be impacted by randomization. To address this threat, we run and repeat 100 times a simulation scenario where a tester selects mutants and designs tests to kill them. Overall, we mitigate these threats by following suggestions from mutation testing literature [46], [7], [34], using state-of-the-art tools and performing several simulations. We also find consistent and stable results across our subjects.

IX. Related Work

Mutation testing is widely used in experimental studies to compare and assess testing techniques [46], motivated by studies showing that mutant killing ratios have similar trends to real fault detection ratios [8]. While traditionally mutants are seeded by performing simple syntactic changes in the programs, real faults are in their majority more complex [24], [12]. These studies introduce mutants that are complex and look like being similar to real faults. In particular, it was proposed to form mutation operators based on fault-fixing commits [14], [51] or recurrent real fault instances [11], [29].

Traditional mutation testing approaches generate mutants by using simple syntactic changes since previous studies have shown the existence of the so-called *coupling effect* that states that simple faults can subsume almost all the complex ones [20], [40]. This brings into question the pattern-based approaches that aim at mimicking the various fault instances syntactically since it is likely that mutants that are syntactically dissimilar to real faults couple with them. This is a motivation for conducting our work.

The relation between mutants generated by different grammar-based mutation testing tools has also been studied [31], [46], and showed that PIT, which we use in our study, is the most effective tool today [32]. Though previous studies compare the effectiveness of mutation testing tools and draw conclusions under the assumption that every mutant has the same probability of being selected, that is, under a standard mutant selection policy. However, recent advances have resulted in powerful techniques for cost-effectively selecting mutants, i.e., by avoiding the analysis of redundant mutants (basically, equivalent and subsumed ones) [23]. In particular, such approaches utilise the knowledge of mutants' surrounding context, embedded into the vector space, to judge the usefulness of a mutant for specific tasks, which usually reduces the mutation testing effort. Besides, a recent study also explored mutant selection using manually engineered features to capture their test completeness probability - being subsuming. However, as the study on *Cerebro* [23] showed that the deep learning approach outperforms such models, we decided to use it in this study over the other approach. A recent line of work has also been formed to study the existence of commit-relevant mutants in capturing regression faults, and change-aware test assessment in the context of evolving systems [42], [43]. To this end, in our study we investigate selection and fault detection such as to differentiate the cost-effective performance of the tools, resulting in different conclusions on comparing the tools. This makes using various mutant selection strategies particularly important when comparing and assessing mutation testing tools and techniques.

X. Conclusion

We studied the fault detection performance of recently proposed mutation testing tools (DeepMutation, PIT, IBIR and μBERT) on a new and large fault dataset.We also employed two different mutant selection strategies a) standard, b) state-of-the-art deep learning driven one, then we performed a cost-effectiveness comparison using two typically adopted cost models; one associated with the number of mutants requiring analysis and a second one with the number of tests to reveal the injected faults. Our results showed that IBIR has the highest fault detection capability ($\approx$90% on average) but is not the most cost-efficient. In contrast, μBERT, even less effective, has significantly higher cost-effectiveness when using learning-based selection strategies, approximately 12% higher, than all the other tools. More notably, we found that mutation testing tools perform differently when guided by mutant selection strategies, indicating the need for considering intelligent selection when comparing mutation testing tools. In other words, mutant selection strategies must be considered when comparing the cost-effectiveness of mutation testing tools to avoid the risk of making wrong conclusions.

Acknowledgment

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C19/IS/13646587/RASoRS.

REFERENCES

[1] Deepmutation. https://github.com/micheletufano/DeepMutation.

[2] Defects4j issue- 353. https://github.com/rjust/defects4j/issues/353.

[3] Master branch deepmutation. https://github.com/micheletufano/DeepMutation/commit/a20882d8fbd107762e2d40f5742d838242dbf1e5.

[4] src2abs. https://github.com/micheletufano/src2abs.

[5] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *2014 IEEE seventh international conference on software testing, verification and validation*, pages 21–30. IEEE, 2014.

[6] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[7] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[8] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 402–411. ACM, 2005.

[9] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Trans. Computers*, 42(8):913–923, 1993.

[11] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry–a study at facebook, 2021.

[12] Marcel Böhme and Abhik Roychoudhury. Corebench: studying complexity of regression errors. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 105–115. ACM, 2014.

[13] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1442–1451, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[14] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 511–522. Association for Computing Machinery, 2017.

[15] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empir. Softw. Eng.*, 25(1):434–487, 2020.

[16] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 597–608. IEEE / ACM, 2017.

[17] Jörgen Christmansson and Ram Chillarege. Generation of error set that emulates software faults based on field data. In *Digest of Papers: FTCS-26, The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing, 1996*, pages 304–313. IEEE Computer Society, 1996.

[18] Henryy Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for java (demo). In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 449–452. ACM, 2016.

[19] Renzo Degiovanni and Mike Papadakis. μBERT: Mutation testing using pre-trained language models. In *Mutation Workshop at ICST*. IEEE, 2022.

[20] Richard Demillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34 – 41, 05 1978.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[22] Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Learning from what we know: How to perform vulnerability prediction using noisy historical data. *Empir. Softw. Eng.*, 27(7):169, 2022.

[23] Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Cerebro: Static subsuming mutant selection. *IEEE Transactions on Software Engineering*, pages 1–1, 2022.

[24] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ISSRE '14, page 189–200, USA, 2014. IEEE Computer Society.

[25] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009. Source Code Analysis and Manipulation, SCAM 2008.

[26] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 433–436, 2014.

[27] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, 2014.

[28] Samuel J Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. Prioritizing mutants to guide mutation testing. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1743–1754, 2022.

[29] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawende F. Bissyandé, Jacques Klein, and Yves Le Traon. Ibir: Bug report driven fault injection. *ACM Trans. Softw. Eng. Methodol.*, may 2022.

[30] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *2010 Asia Pacific Software Engineering Conference*, pages 300–309, 2010.

[31] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–156. IEEE, 2016.

[32] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empir. Softw. Eng.*, 23(4):2426–2463, 2018.

[33] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng. Mutant subsumption graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 176–185, 2014.

[34] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 571–582, 2016.

[35] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque. Assessing and improving the mutation testing practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435, March 2017.

[36] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 75–87. ACM, 2020.

[37] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Softw. Test. Verification Reliab.*, 15(2):97–133, 2005.

[38] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442 – 451, 1975.

[39] Roberto Natella, Domenico Cotroneo, João Durães, and Henrique Madeira. On fault representativeness of software fault injection. *IEEE Trans. Software Eng.*, 39(1):80–96, 2013.

[40] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, January 1992.

[41] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century*, pages 34–44, 2001.

[42] Miloš Ojdanić, Wei Ma, Thomas Laurent, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. On the use of commit-relevant mutants. *Empirical Software Engineering*, 27(5):1–31, 2022.

[43] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Mutation testing in evolving systems: Studying the relevance of mutants to code evolution. *ACM Transactions on Software Engineering and Methodology*, 2022.

[44] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. Mutant quality indicators. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 32–39. IEEE Computer Society, 2018.

[45] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 354–365, 2016.

[46] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378, 2019.

[47] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[48] Jibesh Patra and Michael Pradel. Semantic bug seeding: A learning-based approach for creating realistic bugs. ESEC/FSE 2021, page 906–918, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Cedric Richter and Heike Wehrheim. Learning realistic mutations: Bug creation for neural bug detectors. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 162–173, 2022.

[50] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[51] Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. Deepmutation: A neural mutation tool. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, page 29–32, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

[53] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes, 2019.

[54] András Vargha and Harold D. Delaney. A critique and improvement of the "cl"d common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.