An Investigation of Compression Techniques to Speed up Mutation Testing

Zhu, Qianqian; Panichella, Annibale; Zaidman, Andy

DOI
10.1109/ICST.2018.00035

Publication date
2018

Document Version
Final published version

Published in
Proceedings of the 11th International Conference on Software Testing, Verification, and Validation (ICST)

2018 IEEE 11th International Conference on Software Testing, Verification and Validation

An Investigation of Compression Techniques to Speed up Mutation Testing

Qianqian Zhu
Delft University of Technology
Email: qianqian.zhu@tudelft.nl

Annibale Panichella
University of Luxembourg
Email: annibale.panichella@uni.lu

Andy Zaidman
Delft University of Technology
Email: a.e.zaidman@tudelft.nl

***Abstract*—Mutation testing is widely considered as a high-end test coverage criterion due to the vast number of mutants it generates. Although many efforts have been made to reduce the computational cost of mutation testing, in practice, the scalability issue remains. In this paper, we explore whether we can *use compression techniques to improve the efficiency of strong mutation based on weak mutation information*. Our investigation is centred around six mutation compression strategies that we have devised. More specifically, we adopt *overlapped grouping* and *Formal Concept Analysis* (FCA) to cluster mutants and test cases based on the reachability (code coverage) and necessity (weak mutation) conditions. Moreover, we leverage mutation knowledge (mutation locations and mutation operator types) during compression. To evaluate our method, we conducted a study on 20 open source Java projects using manually written tests. We also compare our method with pure random sampling and weak mutation. The overall results show that mutant compression techniques are a better choice than random sampling and weak mutation in practice: they can effectively speed up strong mutation 6.3 to 94.3 times with an accuracy of $>$90%.**

I. Introduction

Mutation testing has been actively investigated as a technique to evaluate the quality of test suites [1]. The main idea is (i) to introduce small syntactic changes (*mutants*) into the production code using *mutation operators*, and (ii) to measure the ability of a given test suite in detecting them [2]. One of the benefits reported in literature is that mutation testing provides a better measure of the fault detection capability of test suites compared to other test coverage criteria [3]–[5]. Despite its well-known advantages, mutation testing remains an extremely expensive activity since it requires to re-run the test suites against each mutant, whose number increases exponentially with the size of the program under test [6].

To address this limitation, several methods have been proposed and these can be classified in three main categories [7]: (*do fewer*) selecting fewer mutants to evaluate [8], [9], (*do smarter*) using run-time information to avoid unnecessary test executions [10], [11], (*do faster*) reducing the execution time for each single mutant [12]. Techniques falling into the first category are the most investigated. Indeed, researchers have proposed various strategies to sample mutants, such as random sampling [13], mutation operator selection [9], clustering [14], static analysis [15], and machine learning based sampling [16].

Recently, Gopinath et al. [17] have challenged the effectiveness and efficiency of mutation reduction strategies: their empirical evaluation with eight common mutant reduction techniques showed that *none* of them *provide any practical advantage over pure random sampling*. Although some techniques showed small improvements in effectiveness, the gains do not compensate for the extra overhead. Therefore, there is a need for reduction techniques that are not only more effective, but also more efficient compared to random sampling.

This paper originates from the insights of Gopinath et al. [17] and focuses on the mutant reduction technique recently proposed by our previous work [18]. We originally tackle the problem of reducing the cost of mutation testing by combining *do fewer* and *do smarter* techniques through data compression methods. First, *weak mutation* (*do smarter*) is used to determine which mutants lead to an *infection state* through one single execution of the test suite against the original program. Then, *formal concept analysis* [19] (FCA) is applied to derive the *maximal groupings* [18], which are two-way clusters of mutants and tests. Each *maximal grouping* is composed of a set of mutants M and a set of tests T with the property that any mutant in M is weakly killed by any test in T. Finally, a *do fewer* strategy is applied by running one single test case (test selection) against one single mutant (mutant sampling) from each maximal grouping [18]. Our initial empirical study with five Java programs and automatically generated unit test suites showed that FCA reduces the execution time of mutation testing by up to 85%.

In this paper, we spot two important limitations of FCA that can affect its ability to correctly estimate strong mutation. First, FCA groups mutants and tests according to weak mutation only: mutants leading to an infection state when running the same test case t (or set of tests) are assumed to be *redundant*. This is why only one mutant in each maximal grouping is evaluated for strong mutation. However, mutants that are redundant in terms of weak mutation are not necessarily redundant in terms of strong mutation, because, for example, they are injected in different code locations (e.g., different methods) or are generated by different mutation operators. Second, we previously [18] focused only on *maximal groupings*, which may leave some tests and/or some mutants not assigned to any maximal grouping. As consequence, the estimated mutation score may be inaccurate.

To overcome these limitations, we enhance FCA with (i) *mutant location* and (ii) *mutation operator type* information when grouping mutants and tests according to weak mutation. This prevents mutants infecting different statements or gener-

0-7695-6388-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICST.2018.00035

ated by different operators to be inserted in the same grouping. We also investigate maximal and non-maximal groupings to prevent final clusters from missing test cases and/or mutants.

To evaluate the benefits of our enhancements, we conducted an empirical study with 20 open-source Java projects and using the test suites manually written by the original developers. Then, we compare the different variants of the FCA-based technique with and without our enhancements and against weak mutation and pure random sampling (with 10% as sampling percentage).

Our results show that FCA with our enhancements is more accurate in estimating the (strong) mutation score compared to (i) the original FCA-based technique by our previous work [18], (ii) random sampling, and (iii) weak mutation. In particular, we find that the compression strategy based on non-maximal groupings and enriched with mutant location information (referred as *overlap+mloc* in the remainder of the paper) estimates the strong mutation score with an average absolute error of 5% and an average accuracy of 93% while being five time faster than strong mutation, i.e., its average speed up is 5X. Instead, random sampling achieves a higher absolute error of 13% while requiring the same execution time of *overlap+mloc*, i.e., its average speed up is 5X as well. The other compression strategies lead to larger speed-up scores (up to 18X) but with the cost of having a larger absolute error, which ranges between 5% (i.e., the absolute error of *overlap+mloc*) and 13% (i.e., the absolute error of random sampling) on average. Therefore, our findings challenge prior results [17] as we find that mutation strategies based on compression methods (and FCA in particular) are more effective and/or more efficient than random sampling. Finally, weak mutation is the fastest technique but it also produces the largest absolute error of 23% on average.

As a final remark, we observe that all FCA-variants allow to estimate whether each individual mutant is strongly killed or not based on relatively few test executions by relying on the two-way clusters generated with FCA. Instead, random sampling does not take into account relationships between mutants and test cases, possibly also leading to underrepresented areas of production code in the estimation.

II. Background and Related Work

In this section, we begin with an overview of mutation reduction techniques in literature and the works that motivate our approach. Then, we introduce the crucial concepts and theories on which our approach is based.

A. Mutation Reduction Strategies

Techniques for reducing the high computational cost have been an active area of research. Offutt and Untch [7]'s literature review summarises these approaches into three categories: *do fewer*, *do smarter* and *do faster*. The most well-known techniques for reducing the computational cost of mutation testing are *random sampling* [8], *selective mutation* [9], *weak mutation* [10] and *mutant schema* [20]. The aforementioned methods are independent of the program under test which can be flexibly combined with our methodology.

More recently, researchers have aimed to make further gains by including run-time information; a widely-adopted strategy is to execute the test suite on the original program before mutation execution to avoid unnecessary executions. *Coverage-based optimisation* filters out test executions when a test case does not cover the mutated statement; this optimisation is in use in existing tools such as JAVALANCHE [21], Major [22] and PIT/PiTest [23]. *Infection-based optimisation* on the other hand only executes a test case on a mutant when the test infects the state of the mutant, filtering out weakly live mutants.

Just et al. [11] improved upon this by only executing a test on a mutant if the execution state of the mutated expression propagates to a top-level expression; they also partitioned mutants based on their intermediate results. Ma and Kim [24] applied a similar idea to cluster mutants for each test case by comparing the values of innermost expressions. Compared to Just et al. [11] and Ma and Kim [24]'s, we partition mutants for all test cases instead of targeting each test case.

Mutant clustering's aim is to reduce the number of mutants based on the similarity of mutants instead of random sampling. Hussain [14] applied clustering algorithms (e.g. K-means), however, the approach requires the execution of all mutants against all the test cases, which cannot reduce the overhead during the mutation execution. Later, Ji et al. [13] measured the similarity of the mutants using domain analysis. They divide mutants based on static control flow analysis. But they only manually analysed the clustering accuracy without indicating the runtime overhead caused by the domain analysis. Different from these works, our approach groups mutants based on their reachability and necessity conditions against the tests.

An approach that eliminates redundant mutants is *mutant subsumption* (e.g. [25]–[27]). However, mutant subsumption requires full knowledge of the mutation kill matrix, which requires the execution of every mutant against every test. Computationally this process is more costly than traditional strong mutation, thus cannot be used to speed-up mutation execution. Furthermore, *test prioritization and reduction* are also used to speed up mutation testing, e.g. Zhang et al. [28].

Moreover, Zhang et al. have recently proposed *Predictive Mutation Testing* (PMT) to predict mutation testing results without execution [29]. They extracted 12 features from the programs and constructed a classification model to predict whether a mutant is killed or surviving. Their experiment showed that PMT could improve the efficiency of mutation testing by up to 151.4 times with a small loss in accuracy. Despite high efficiency, their approach needs to collect a series of program features, which requires different tools to fulfil; this is a substantial burden for the common programmer. Unlike Zhang et al. [29], we do not require any additional program features; the weak mutation information needed for our mutant clustering and data compression can be collected by our tool during the initial execution against the original program.

Our approach is an extension of our previous work [18]. Despite the encouraging results, we have identified two impor-

tant limitations of our initial methods as mentioned before (in Section I), i.e., (1) weak mutation information is not enough; and (2) FCA could lead to missing mutants and/or tests. To address these limitations, we propose another compressing strategy, i.e., *overlapped grouping*, which is the simplest and strictest clustering method. Moreover, we take full advantage of mutation location and operator type knowledge when compressing.

B. Mutant Compression

We now describe the core concepts behind our approach: *weak mutation* and *FCA-based compression technique [18]*.

Weak Mutation. For a test case t to kill a mutant m which mutates the statement s of a program P, there are three conditions [30]: (i) *reachability*: the execution of t must cover s; (ii) *necessity*: the execution state of m is different from the execution state of s; (iii) *sufficiency*: the incorrect state of m must propagate to the output causing a failure in t.

Weak mutation uses the necessity condition, i.e., a mutant is *killed* if its execution leads to a state change. For example, the expression `c=a*b` and its mutated version `c=a/b` have different outcomes (i.e., the mutant is weakly killed) if `a≠1`, `a≠0` and `b≠1`. Differently from *strong* mutation, *weak* mutation scores can be computed with one single execution of each test by instrumenting the mutated locations [31].

FCA-based compression technique [18]. *Formal Concept Analysis* (FCA) was originally a data analysis method and has shown to be a powerful mathematical technique to convey and summarize large amounts of information [19]. It takes as input the *formal context* which is a structure $C = (O, A, I)$ where O is the set of *objects*, A is the set of *attributes* while $I \subseteq O \times A$ is a binary relation between O and A. Then, FCA produces the *concept lattice*, which is a collection of *formal concepts* in the data ordered by *sub-concept* relations, i.e., from super-concepts to sub-concepts. Each *formal concept* is composed of (i) a group of objects sharing the same attributes, and (ii) all attributes that apply to the objects in the concept [19].

In mutation testing context, the objects in O are the mutants, the attributes in A are the test cases, and I is the mutant-by-test infection matrix. Then, FCA derives formal concepts that represent groups of mutants that are weakly killed by the same subset of tests. In other words, the output of FCA can be viewed as two-way clustering since mutants and tests are grouped in concepts such that all mutants in the same concept c are weakly killed by all tests in c. Among these concepts, *FCA-based compression technique* only considers the maximal concepts that are directly connected to the exit point in the lattice hierarchy that are referred to as *maximal groupings*.

After obtaining the maximal groupings from the concept lattice, this approach first compresses the mutant-by-test infection matrix by condensing the rows, i.e., select one mutant from each maximal grouping. Then to further perform the compression on the columns, there are three approaches for *test case selection*: (i) *random*: randomly select one test from each maximal grouping; (ii) *Set cover based*: find a sufficient subset of test cases that weakly kill all possible mutants. i.e., at each stage, choose the test that weakly kills the largest number of *uncovered* maximal groupings; (iii) *Sorting-based*: select the test cases with the largest number of maximal groupings at each stage until all possible mutants are covered.

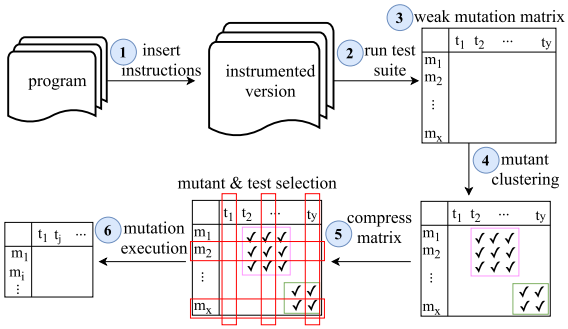


Fig. 1. Overall Methodology of Mutation Compression Strategies

However, as our previous study [18] show, applying test case selection on FCA groupings leads to a relatively small (5.89%) reduction in execution time. Therefore, in this paper, we do not use test case selection. Instead, we apply dynamic coverage-based optimisation [21]–[23] and infection-based optimisation [11], [24] (see Sec. II) to filter out unnecessary executions.

III. Approach

A. Overall Methodology

Our 6-step compression strategy is illustrated in Figure 1:

(1) Instrumentation. We instrument the original program to keep track of the mutation locations: at every mutation point we insert all the mutants (mutated codes) right after the original one and assign a unique id to each mutant for later activation (we applied the technique of the *mutant schemata* [12]). To perform weak mutation, we also insert the comparison instructions at each mutation point to compare the intermediate states of the original program and mutated part (we compare the state after the first execution of the innermost expression that surrounds the mutant). We insert additional instructions to record information of each mutant including its location, operator type and mutation details (e.g., m_1 on Line 12 applies replace constant operator: $0 \rightarrow 1$).

(2) Test execution. Once instrumented, the test suite is executed once on the original program. During this stage, we record the mutants that are touched by the tests, as well as the ones which are weakly killed by the tests. Only the instructions related to weak mutation and mutant information collection are executed at this stage. No mutants are activated.

(3) Reachability and necessity analysis. The results of the previous stage are stored in the *mutant-by-test reachability* and *mutant-by-test necessity* matrices. Let P be the program under analysis and let T be the test suite; let $\mathbb{M}$ be the set of mutants for the program P generated by preselected mutation operators. A *mutant-by-test reachability* matrix is a $m \times n$ matrix where m is the number of mutants, n is the number of test cases in T, and an entry $x_{i,j}$ is a binary value indicating whether the statement containing the i-th mutant is executed

($x_{i,j} = 1$) or not ($x_{i,j} = 0$) by the j-th test $\in T$. The *mutant-by-test necessity* matrix has the same size as the *mutant-by-test reachability* matrix, but the binary entry $x_{i,j}$ represents the outcome of weak mutation ($x_{i,j} = 1$ indicates weakly killed).

(4) Mutant clustering. Using the two aforementioned matrices, we apply clustering to group similar mutants together. We consider two clustering methods: (1) the *overlapped* grouping, and (2) *FCA* grouping from our previous study [18] (See Section II-B).

(5) Data compression. The mutant clusters are then used to compress the *mutant-by-test* matrix. The resulting matrix is likely to have lower dimensionality: the rows denote groups of mutants belonging to the same clusters; similarly, tests are grouped into clusters to form the columns. The compressed matrix is then used to apply mutants and execute tests for the strong mutation analysis. We take *mutation knowledge* into consideration during compression to achieve higher accuracy.

(6) Mutant Execution. The compressed matrix from the previous step is then used for the strong mutation analysis. Here, we load each mutant by its id and run the actual mutation execution against the tests.

The details of *overlapped grouping* and how we use *mutation knowledge* are described in the next sub-sections.

B. overlapped grouping

As mentioned earlier, the FCA-based compression technique could result in missing mutants and/or tests. Therefore, we propose the *overlapped* grouping to overcome this. The *overlapped* method is the simplest and strictest clustering method, i.e., elements are only grouped together if they are identical. Specifically, we first identify distinct mutants with regard to their reachability and necessity conditions against all the test cases. Subsequently, we group mutants having the same reachability and necessity conditions into one cluster.

The main difference between *overlapped* and *FCA* grouping is that *overlapped* grouping is stricter than *FCA* grouping. The *overlapped* grouping does not lose any information in the matrix, i.e., the clustering contains all the mutants. While *FCA* grouping loses some information since the main idea of *FCA* is to find the maximal sub-matrixes (or formal concepts), and if a mutant or a test does not belong to a sub-matrix, *FCA* removes this mutant or test.

In our example in Figure 2, there are three maximal groupings, which are $\{m5, m6|t1, t2\}$, $\{m4|t3\}$, and $\{m3|t4\}$. The other concepts in the lattice (e.g., $\{m2, m5, m6|t1\}$ in Figure 2) are already included in the maximal grouping by the sub-concept relation which is graphically represented by the hierarchy in the lattice. As for the *overlapped* grouping, it generates five clusters, i.e. $\{m1|t2\}$, $\{m2|t1\}$,$\{m3|t4\}$, $\{m4|t3\}$ and $\{m5, m6|t1, t2\}$. The example shows that for the clusters generated by the *overlapped* grouping no mutants are discarded, while the *FCA* grouping discards $m1$ and $m2$.

C. Mutation Knowledge

Once we have the mutant clusters, we compress the mutant-by-test infection matrix by condensing the rows, i.e., we select

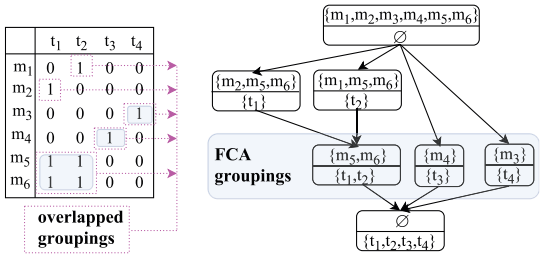


Fig. 2. A toy program and its mutant clusters

one/several mutants from each cluster to represent the whole group. To select the representative mutant(s), the FCA-based technique [18] adopted a random strategy which selects one mutant from each cluster at random, which we believe causes another limitation: based on weak mutation information alone, FCA could mis-cluster mutants at different code locations. Thus, we enhance FCA by adding *mutation knowledge*: (i) mutant location and (ii) mutation operator type.

More specifically, we investigate three mutant selection strategies: (1) random strategy; (2) random strategy with knowledge of the mutation operator type[1]; (3) random strategy with knowledge of the mutation location. The first one randomly chooses one mutant from each grouping/cluster as the representative mutant. The second strategy first divides each cluster into partitions by the type of mutation operator and then randomly selects one mutant from each partition. The third strategy partitions the cluster by the locations of the mutants (the line number) and then applies random selection; this guarantees that at least one mutant is selected for every potential mutation point. Notice that the second and third strategies might select more than one mutant from each grouping/cluster, which could lead to less speed-up in strong mutation.

To sum up, we devise one mutant clustering algorithm in addition to *FCA* as well as three mutant selection strategies, therefore, resulting in six compression strategies in total:

overlap	the combination of *overlapped* grouping and random strategy in mutant selection.
overlap+mop	the combination of *overlapped* grouping and random strategy with the knowledge of the mutation operator type in mutant selection.
overlap+mloc	the combination of *overlapped* grouping and random strategy with the knowledge of the mutation location in mutant selection.
fca	the combination of *FCA* grouping and random strategy in mutant selection.
fca+mop	the combination of *FCA* grouping and random strategy with the knowledge of the mutation operator type in mutant selection.
fca+mloc	the combination of *FCA* grouping and random strategy with the knowledge of the mutation location in mutant selection.

IV. Experimental Study

We conducted an empirical study to evaluate the effectiveness of the different compression strategies presented in the previous section. The *goal* of the study is to answer the following research questions:

[1]Regarding the mutation operator type knowledge, we consider the operator type at a high level, e.g., arithmetic replacement operators.

- **RQ1:** *How accurate are different compression techniques?* We assess the ability of the six compression strategies to estimate the strong mutation scores. We also asses their performance in comparison with random sampling and weak mutation.
- **RQ2:** *How do compression techniques perform in terms of speed-up?* We investigate the speed-up in terms of execution time that can be obtained when using each compression strategy over strong mutation. We also consider random sampling and weak mutation as baselines.
- **RQ3:** *What is the trade-off between accuracy and speed-up for the compression techniques?* We evaluate to what extent the compression strategies can reduce execution time while maintaining an accurate estimation of the strong mutation scores.

A. Experimental setup

To answer our research questions, we evaluated the six compression strategies using 20 open source projects publicly available on GitHub. Table I summarises the main characteristics of the selected projects. These projects have been randomly selected among the top 3000 GitHub repositories which (1) have most stars on 04/04/2017, (2) can be built using Maven, and (3) contain JUnit 4 test suites. In our study, we focus on the *manually-written* test suites available in the original project repositories.

As mentioned in Section III, we first need one test execution against the original program to collect statement coverage (i.e., the *mutant-by-test reachability*) and the weak mutation information (i.e., the *mutant-by-test necessity* matrix). To collect weak and strong mutation information, we implemented our own prototype tool[2]. The instrumentation framework to generate mutants and detect the reachability and necessity condition is extracted from EvoSuite. Then, we integrated this instrumentation framework into our mutation testing runner. We record information about test cases (#id, method name, execution results, #touched mutants and #weakly killed mutants) and mutants (#id, mutation operator type, location and detailed information) for further analysis. After that, we run each test case against each mutant of the class under test (strong mutation) to establish the *mutant-by-test sufficiency* matrix which is used to evaluate our methods.

The mutation operators we adopted in this experiment are six method-level operators: `replace arithmetic`, `replace bitwise`, `replace comparison`, `replace variable`, `replace constant`, and `insert unary`. Further details about these mutation operators can be found in the paper by Fraser and Arcuri [31]. We opted for the mutation engine available in EvoSuite [31] because it instruments the production code at bytecode level and allows to directly measure the infection state for each mutant (weak mutation). To the best of our knowledge, no publicly-available mutation tool provides utilities for computing the weak mutation scores.

[2]All the tools, scripts and metadata for this experimental study are available in our GitHub repository [32].

TABLE I
SUBJECT PROGRAMS

PID	Project	LOC	#Classes	#Tests	COV[3]	#Mutants			
						#Total	#Covered	#Weakly Killed	#Strongly Killed
1	assertj	24978	830	8545	0.90	59955	21677	19533	9178
2	checkstyle	31441	524	631	0.74	79464	18777	17738	9448
3	commons-lang	26578	264	2936	0.93	45630	43597	40453	33158
4	crawler4j	3745	57	11	0.27	3046	1017	853	505
5	dex-translator	4981	32	3	0.61	5812	1015	882	493
6	distributedlog	27976	697	339	0.43	21520	593	535	395
7	dynjs	34579	672	887	0.51	29148	14688	12960	8307
8	geotools	75236	991	670	0.38	62963	19852	17024	9135
9	graphhopper	26175	384	874	0.74	50319	11176	10168	7299
10	apns	1618	39	85	0.66	905	537	487	366
11	jctools	6262	133	43	0.82	7058	425	401	302
12	jfreechart	98334	657	2256	0.54	129417	14776	13057	5669
13	jpacman	1890	61	41	0.82	1606	1355	1159	580
14	junit-quickcheck	3038	67	354	0.98	1000	995	893	786
15	pac4j	5281	146	424	0.63	2703	1934	1689	929
16	pf4j	3021	67	24	0.28	1176	380	294	205
17	stream-lib	4767	77	121	0.83	11907	9920	9445	6924
18	telegrambots	1480	21	31	0.20	772	221	196	52
19	vraptor	12021	407	385	0.83	3490	2057	1652	1072
20	zt-zip	4255	84	3	0.71	2440	1049	906	422
Overall		397656	6210	2470	0.64	520331	166041	150325	95225

Note: Column 4 is the total number of *passed* test cases under our ComMT tool. We marked the value with underline when the total number of passed test cases is less than the entire test suite size[4].

To answer the three **RQ**s, we selected another two mutation reduction techniques (see Section II) for comparison: mutation sampling and weak mutation. We selected random sampling (*do fewer* strategy) as baseline because Gopinath et al. [17], [33] showed that none of the most common reduction strategies provide any practical advantage over *pure random sampling*. Moreover, we selected weak mutation (*do smarter* strategy) because it is one of the key components of all mutant compression techniques. Therefore, we considered it as an additional baseline to verify whether the other components of the compression strategies (e.g., computing the maximal groupings) are indeed needed. For random sampling, we set the sampling rate to 10% as suggested by Budd [6] and Acree [8]. They showed that 10% sampling could already estimate the mutation score with 99% of accuracy. It also corresponds to the sampling rate used by Gopinath et al. [17], [33]. Since random sampling and the mutant compression strategies involve random processing (i.e., in mutant selection), we carry out the corresponding random process 100 times for each project to address their randomised nature. In total, we compared eight mutation strategies: six compression strategies, random sampling (10%) and weak mutation.

B. Evaluation Metrics

To answer **RQ1**, we selected two well-known performance metrics: the *absolute error* and the *accuracy*. Let M be a given mutation strategy (e.g., random sampling); let $strong_M(C,T)$ be the percentage of mutants for a class C that are strongly killed by the test suite T; let $estimated_M(C,T)$ be the

[4]The fifth column "COV" means the line coverage of the test suite which is measured by IntelliJ IDEA coverage runner.

[5]The failures of the test cases in our tool is because the dependencies of these test cases need to be configured by Maven plugin; this cannot be solved in the current version of ComMT.

estimated percentage of mutants that are killed according to the strategy M; the *absolute error* is defined as follows:

$$AE(C,T) = \mid strong_M(C,T) - estimated_M(C,T) \mid \quad (1)$$

While the compression techniques select only a subset of the mutants for strong mutation, they can also estimate whether the non-selected mutants are killable by leveraging the groupings generated by FCA. Therefore, we use the accuracy as further performance metric, which is defined as follows:

$$accuracy(C,T) = (TP + TN)/total \quad (2)$$

where TP denotes the number of mutants that are strongly killed by T and that are also correctly identified by a given method M (*true positives*); TN is the number of mutants that are not strongly killed by T and that are correctly identified by M (*true negatives*); *total* denotes the total number of mutants for the class C. To ease the comparison, we use the *mean* and *standard deviation* of *absolute error* and *accuracy* scores obtained for all classes in a given Java project.

For **RQ2** we consider the *speed-up* metric. When establishing the speed-up, we should first consider the overhead induced by an approach. For random sampling, we consider the overhead to be zero, as mutation sampling does not require any prerequisite knowledge. For weak mutation, the overhead consists of one single execution of the test suite against an instrumented version of the original program. For the compression strategies, the overhead is composed of the overhead incurred by both weak mutation and the compression procedure (mutant clustering and mutant selection):

$$overhead = exec\_time(weak\_mutation + compression) \quad (3)$$

The *speed-up* metric itself is computed using strong mutation with coverage-based optimisation as the baseline. We explicitly chose this optimisation as it is already integrated into several existing mutation testing tools (e.g., JAVALANCHE [21], Major [22] and PIT/PiTest [23]). The results of random sampling and compression strategies are the average values over 100 runs; for weak mutation, the execution time is zero. Then, the speed-up is defined as follows:

$$speed\text{-}up = \frac{exec\_time(strong\_mutation)}{exec\_time(M)} \quad (4)$$

where the denominator is the execution time of a method M computed as the sum of its overhead and the execution time needed to run the tests against the selected mutants.

For **RQ3**, we first provide a graphical comparison among the different mutation strategies by using the *speedup-error graphs*. In such a graph, the X-axis denotes the speed-up scores and the Y-axis shows the mean absolute error achieved by each strategy and for each project in our study; an "ideal" score would have a high X-value and a low Y-value. We also use the *speedup-accuracy graphs*, which plot the speed-up (mean) on the X-axis and the corresponding accuracy (mean) on the Y-axis; an "ideal" score would have a high X-value and a high Y-value. Although we may see some trends via graphical analysis, we would like to know which strategy achieves the best speed-up when accepting a given absolute error rate. Therefore, we consider the following absolute error thresholds: $\sigma_{e_1} = 5\%$, $\sigma_{e_2} = 10\%$, and $\sigma_{e_3} = 15\%$. Then, for each threshold σ_{e_i} and for each mutation strategy M, we count the number of projects for which M achieves the highest speed-up compared to the other strategies while yielding an absolute error score lower than σ_i. Similarly, we also consider three accuracy thresholds, i.e., $\sigma_{a_1} = 95\%$, $\sigma_{a_2} = 90\%$, and $\sigma_{a_3} = 85\%$. Different from the absolute error, we count the number of projects in which M achieves an accuracy higher than the threshold σ_{a_i}.

Statistical Analysis. To assess whether the differences among the various mutation strategies are statistically significant or not, we adopt Friedman's test [34] with $\alpha = 0.05$. It is a non-parametric test for comparing multiple treatments (mutation strategies) in the context of a multiple-problem analysis (i.e., multiple projects) [34]; it does not require data to be normally distributed and it is widely applied to compare randomised algorithms [35], [36] (e.g., random sampling). While Friedman's test reveals whether data distributions differ statistically, tests for pairwise comparison are needed to determine which treatment outperforms the others. For this, we use Conover's post-hoc procedure [37] and we further adjusted the obtained p-values using Holm-Bonferroni [38].

V. Results

A. RQ1: accuracy

Table II reports the mean accuracy and absolute error scores for each project in our study as well as the corresponding standard deviation scores.

Absolute error. Focusing on the absolute error we observe that weak mutation performs worst with an error rate of 23% and a standard deviation of 22% on average. This result is due to this strategy using all mutants that are weakly covered to approximate strong mutation coverage, yet, not all infected states propagate to changes observable in assertions.

Moreover, *overlap+mloc* produces the lowest mean absolute error in 19 out of 20 projects, followed by *fca+mloc* and *overlap+mop*. Overall, the strategies using *overlapped grouping* perform slightly better than FCA-based maximal groupings. To ease the comparison between the compression techniques and random sampling, in Table II we underline the mean and std values of a compression technique if they are smaller (better) than the values of random sampling in the same project. We can see that the six compression strategies outperform random *sampling* in terms of absolute error scores for most projects. The best (lowest) scores are obtained using the *overlap+mop*, *overlap+mloc*, *fca+mop* and *fca+mloc* approaches.

Looking at the six compression strategies, we can see that the mean absolute error and standard deviation scores are reduced when incorporating knowledge of mutation operators (*mop*) and location (*mloc*). For example, *overlap* achieves an error rate of 15% for the `jpacman` project, while when adding mutation operator and location knowledge it achieves an error rate of 10% and 7%, respectively. In general, we

TABLE II
SUMMARY OF THE RESULTS FOR RQ1

PID	Absolute Error Summary (Mean / St. Dev.)								Accuracy Summary (Mean / St. Dev.)							
	Overlap			FCA			Random	Weak	Overlap			FCA			Random	Weak
	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Sampling	Mutation	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Sampling	Mutation
1	0.09/0.17	0.08/0.16	**0.08/0.15**	0.13/0.17	0.11/0.16	0.11/0.16	0.13/0.17	0.20/0.27	0.90/0.17	0.91/0.17	**0.91/0.16**	0.83/0.20	0.86/0.18	0.90/0.17	-	0.76/0.29
2	0.06/0.11	0.06/0.10	**0.03/0.06**	0.08/0.12	0.07/0.10	0.03/0.07	0.10/0.13	0.13/0.17	0.93/0.12	0.93/0.11	**0.95/0.08**	0.91/0.14	0.92/0.12	0.95/0.08	-	0.86/0.17
3	0.07/0.11	0.04/0.07	**0.04/0.06**	0.13/0.13	0.08/0.08	0.06/0.07	0.14/0.16	0.18/0.19	0.88/0.13	0.91/0.10	**0.92/0.08**	0.74/0.19	0.80/0.16	0.90/0.09	-	0.81/0.18
4	0.10/0.12	0.06/0.07	**0.02/0.03**	0.11/0.12	0.07/0.08	0.03/0.03	0.07/0.09	0.28/0.18	0.89/0.13	0.91/0.10	**0.96/0.05**	0.87/0.13	0.89/0.11	0.96/0.05	-	0.71/0.17
5	0.13/0.16	0.09/0.16	**0.07**/0.16	0.14/0.17	0.10/0.16	0.07/0.16	0.10/0.16	0.15/**0.13**	0.86/0.16	0.87/0.16	0.90/0.16	0.85/0.17	0.87/0.16	**0.90**/0.16	-	0.85/**0.13**
6	0.03/0.10	0.03/0.08	**0.02/0.05**	0.06/0.12	0.05/0.09	0.04/0.08	0.14/0.21	0.11/0.17	0.97/0.10	0.97/0.09	**0.98/0.07**	0.92/0.14	0.95/0.11	0.96/0.09	-	0.82/0.21
7	0.12/0.15	0.09/0.13	**0.06/0.09**	0.16/0.16	0.11/0.13	0.07/0.09	0.19/0.19	0.33/0.27	0.86/0.16	0.88/0.14	**0.92/0.10**	0.80/0.18	0.84/0.15	0.91/0.11	-	0.67/0.27
8	0.05/0.09	0.03/0.06	**0.02/0.05**	0.07/0.11	0.05/0.08	0.03/0.05	0.12/0.16	0.24/0.26	0.94/0.10	0.95/0.08	**0.97/0.06**	0.91/0.13	0.93/0.10	0.96/0.06	-	0.75/0.26
9	0.11/0.13	0.07/0.09	**0.04/0.07**	0.15/0.14	0.09/0.10	0.05/0.08	0.12/0.13	0.19/0.18	0.85/0.15	0.88/0.12	**0.92/0.09**	0.76/0.16	0.81/0.14	0.90/0.09	-	0.80/0.17
10	0.06/0.10	0.04/0.05	**0.02**/0.05	0.09/0.12	0.06/0.07	0.03/**0.05**	0.09/0.12	0.25/0.28	0.92/0.12	0.95/0.07	0.97/0.05	0.89/0.13	0.92/0.09	**0.97/0.05**	-	0.75/0.27
11	0.23/0.23	0.22/0.22	**0.20**/0.22	0.23/0.23	0.22/0.22	0.20/0.21	0.29/**0.21**	0.38/0.33	0.77/0.23	0.77/0.22	**0.78**/0.22	0.77/0.23	0.77/0.22	0.78/**0.21**	-	0.62/0.33
12	0.05/0.07	0.04/0.05	**0.01/0.02**	0.09/0.09	0.05/0.06	0.02/0.02	0.03/0.03	0.24/0.23	0.91/0.09	0.92/0.07	**0.95**/0.05	0.87/0.11	0.89/0.10	0.95/**0.05**	-	0.76/0.23
13	0.15/0.16	0.10/0.11	**0.07/0.09**	0.16/0.17	0.11/0.12	0.07/0.09	0.11/0.12	0.39/0.25	0.83/0.17	0.85/0.14	0.90/**0.11**	0.81/0.18	0.84/0.15	**0.90**/0.11	-	0.61/0.25
14	0.04/0.07	0.03/0.06	**0.03/0.05**	0.10/0.11	0.07/0.08	0.07/0.08	0.16/0.18	0.09/0.12	0.96/0.08	0.96/0.07	**0.96/0.06**	0.78/0.18	0.82/0.15	0.95/0.06	-	0.87/0.13
15	0.10/0.15	0.07/0.13	**0.05/0.10**	0.17/0.15	0.12/0.13	0.11/0.11	0.19/0.18	0.24/0.26	0.88/0.15	0.91/0.13	**0.93/0.10**	0.71/0.19	0.80/0.17	0.89/0.12	-	0.74/0.25
16	0.08/0.13	0.06/0.09	**0.04/0.07**	0.11/0.14	0.09/0.11	0.06/0.07	0.21/0.19	0.21/0.24	0.90/0.14	0.93/0.09	**0.96/0.07**	0.86/0.15	0.89/0.12	0.95/0.08	-	0.77/0.23
17	0.10/0.13	0.07/0.10	**0.05/0.09**	0.15/0.14	0.09/0.10	0.05/0.09	0.12/0.14	0.22/0.20	0.84/0.15	0.87/0.13	**0.89/0.11**	0.76/0.18	0.79/0.15	0.88/0.11	-	0.76/0.22
18	0.10/0.12	0.07/0.10	0.03/0.04	0.11/0.14	0.08/0.11	**0.03/0.04**	0.12/0.14	0.30/0.21	0.89/0.12	0.91/0.11	0.94/0.09	0.87/0.15	0.89/0.13	**0.94/0.09**	-	0.70/0.21
19	0.06/0.11	0.04/0.08	**0.03/0.07**	0.11/0.15	0.07/0.11	0.06/0.10	0.17/0.20	0.20/0.25	0.94/0.11	0.95/0.09	**0.97/0.07**	0.88/0.17	0.91/0.13	0.96/0.08	-	0.77/0.24
20	0.14/0.17	0.09/0.12	**0.02**/0.03	0.15/0.17	0.09/0.12	0.02/**0.03**	0.07/0.10	0.31/0.29	0.85/0.17	0.86/0.15	**0.94**/0.08	0.85/0.17	0.86/0.15	0.94/**0.08**	-	0.66/0.28
Mean	0.09/0.13	0.07/0.10	**0.05/0.08**	0.13/0.14	0.09/0.11	0.06/0.08	0.13/0.15	0.23/0.22	0.89/0.14	0.90/0.12	**0.93/0.09**	0.83/0.16	0.86/0.14	0.92/0.10	-	0.75/0.22

TABLE III
RANKING PRODUCED BY THE FRIEDMAN'S (LARGER RANK INDICATES SMALLER ERROR) AND STATISTICAL SIGNIFICANCE BY THE CONOVER'S POST-HOC PROCEDURE.

ID	Algorithms	Rank	Significantly better than
(1)	overlap + Mloc	8.00	(2), (3), (4), (5), (6), (7), (8)
(2)	FCA + Mloc	6.55	(4), (5), (6), (7), (8)
(3)	overlap + Mop	6.30	(4), (5), (6), (7), (8)
(4)	FCA + Mop	4.55	(6), (7), (8)
(5)	Overlap	4.40	(6), (7), (8)
(6)	Random Samp.	2.70	(8)
(7)	FCA	2.50	(8)
(8)	Weak Mut.	1.10	-

find that mutant location is more important than mutation operator, as the absolute error decreases by 4% on average, with a minimum improvement of 1% for `assertj` and a maximum improvement of 13% for `zt-zip`. This is an important novel contribution distinct from most related work, which consider mutation operators as a key element to detect redundant mutants or subsuming mutants (e.g. [25]–[27]). Hence, we found that *mutation location trumps mutation operator information when selecting mutants to evaluate for strong mutation.*

According to results of Friedman's test, the different mutation strategies achieve significantly different absolute error values (p-value = 10^{-16}). To better understand which strategies perform best, Table V reports the final ranking produced by Friedman's test as well as the results of the pairwise comparison from Conover's procedure. As we can observe, *overlap* with mutant location knowledge (*mloc*) is ranked first and performs significantly better than all other strategies in the comparison. FCA based on maximal groupings with mutant location knowledge is ranked second and statistically outperforms all other strategies with lower ranks. Finally, random sampling is statistically worse than all *overlap* strategies and FCA enhanced with *mloc* and *mop*. Instead, the original FCA approach proposed by our previous study [18] is statistically equivalent to *random sampling* in terms of absolute error.

Accuracy. The results for the accuracy are also reported in Table II. Since random sampling selects 10% mutants to evaluate in strong mutation, it cannot be used to estimate whether the other non-selected mutants are killable or not. Conversely, the six compression strategies can estimate whether each mutant is killable even if only a few mutants are actually evaluated for strong mutation. This is possible thanks to the two-way clusters generated by FCA: if a mutant is strongly killed, then we assume that all other mutants within its own cluster are killable as well. In weak mutation, we consider as strongly killable all mutants that lead to a state infection (i.e., the weakly killed ones).

Similar to results of the absolute error, the top three strategies are *overlap+mloc*, *fca+mloc* and *overlap+mop* in terms of accuracy. Again, weak mutation produces the worst accuracy in terms of both mean and standard deviation values. The three compression methods based on the overlapped grouping are slightly better than those in FCA. These differences are due to the fact that FCA considers only the maximal groupings as clusters from which selecting tests and mutants to run. However, as explained in Section III, maximal groupings can miss some mutants, which therefore are not assigned to any cluster. Hence, we cannot accurately estimate whether the missed mutants are likely to be strongly killed or not based on the results of other selected mutants. We also notice that compression with additional mutant information can enhance the predication accuracy. Finally, the finding that mutation location trumps mutation operator information still holds: the improvements range between 1% (for `commons-lang`) and 13% (for `junit-quickcheck`) in terms of accuracy.

Overlap with *mutation location* knowledge outperforms all other mutation strategies in terms of both **absolute error** and **prediction accuracy**. Random sampling is statistically worse than all mutant compression techniques.

B. RQ2: speed-up

Speed-up performance. Table IV summarises the overall speed-up for the eight approaches in our comparison. For each project, we highlight the two strategies achieving the

TABLE IV
SUMMARY OF THE RESULTS FOR RQ2

PID	Overhead summary (compression overhead 10^{-4}%/ overall overhead%)						Speed-up summary (selected mutant% / speed-up)							
	Overlap			FCA			Overlap			FCA			Random	Weak
	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Sampling	Mutation
1	1.04/0.06	**1.18/0.06**	1.04/0.06	1.05/0.06	0.21/0.06	0.78/0.06	8.4/52.1	9.9/27.3	12.5/22.4	3.1/**94.3**	4.9/43.3	9.0/25.8	10.0/22.7	0/**1747.8**
2	0.28/0.08	**0.30/0.08**	0.28/0.08	0.30/0.08	0.09/0.08	0.01/0.08	1.2/32.8	2.2/27.8	12.0/7.6	0.6/**42.1**	1.0/34.1	3.9/7.6	10.0/7.6	0/**1257.0**
3	2.92/0.27	3.19/0.27	3.01/0.27	**3.35/0.27**	0.17/0.27	0.02/0.27	15.0/8.8	22.1/5.4	30.7/4.8	3.2/**30.8**	8.5/14.2	25.6/5.4	10.1/4.9	0/**369.2**
4	0.01/0.30	0.02/0.30	**0.03/0.30**	0.02/0.30	0.01/0.30	0.02/0.30	2.0/27.3	5.1/13.0	17.4/3.8	1.3/**34.2**	3.0/14.5	11.3/3.8	10.0/3.8	0/**334.6**
5	0.01/1.10	0.02/1.10	**0.07/1.10**	0.03/1.10	0.01/1.10	0.03/1.10	0.6/37.6	1.6/21.0	16.0/3.9	0.4/**42.5**	1.1/22.0	5.8/3.9	10.0/4.0	0/**91.2**
6	0.02/9.23	0.02/9.23	0.03/9.23	**0.03/9.23**	0.02/9.23	0.01/9.23	0.7/5.5	1.1/4.9	2.0/4.6	0.4/6.3	0.7/5.4	1.1/4.9	10.0/**8.0**	0/**10.8**
7	0.03/0.03	0.04/0.03	0.04/0.03	**0.05/0.03**	0.02/0.03	0.01/0.03	9.7/11.2	14.5/8.6	25.2/3.1	3.5/**16.1**	6.9/11.5	18.8/3.2	10.0/3.1	0/**3280.9**
8	1.56/0.23	**2.09/0.23**	1.97/0.23	1.93/0.23	0.41/0.23	0.18/0.23	5.2/7.0	8.2/5.2	21.5/3.0	1.7/**13.2**	3.4/8.3	11.4/3.1	10.0/3.1	0/**432.9**
9	0.11/0.18	**0.15/0.18**	0.13/0.18	0.13/0.18	0.02/0.18	0.01/0.18	2.7/7.8	4.5/5.7	8.6/4.2	0.7/**36.9**	1.9/14.4	6.2/4.9	10.0/4.3	0/**561.7**
10	0.01/1.08	0.02/1.08	0.02/1.08	**0.05/1.08**	0.02/1.08	0.01/1.08	4.5/10.1	8.2/6.4	14.2/3.9	2.0/**15.0**	4.3/8.2	9.4/4.1	10.0/4.1	0/**92.9**
11	0.03/0.32	0.08/0.32	0.08/0.32	**0.12/0.32**	0.08/0.32	0.07/0.32	0.6/19.8	1.5/7.6	3.5/4.8	0.5/**19.8**	1.0/7.6	1.8/4.8	10.0/4.9	0/**310.9**
12	3.95/0.21	**4.43/0.21**	3.86/0.21	3.78/0.21	0.16/0.21	0.03/0.21	1.0/12.0	1.6/7.4	7.4/3.9	0.2/**64.9**	0.5/27.9	4.1/4.0	10.0/4.0	0/**481.9**
13	0.01/0.16	0.02/0.16	0.02/0.16	**0.03/0.16**	0.02/0.16	0.01/0.16	6.5/17.8	12.8/8.3	25.4/4.6	3.9/**21.9**	8.4/9.9	20.6/4.7	10.0/4.7	0/**630.9**
14	0.03/0.91	0.07/0.91	0.07/0.91	**0.11/0.91**	0.03/0.91	0.03/0.91	41.1/5.6	48.2/4.6	52.9/4.2	13.3/**11.1**	24.5/7.2	41.5/4.4	11.4/4.4	0/**110.2**
15	0.15/1.29	0.37/1.29	0.37/1.29	**0.45/1.29**	0.18/1.29	0.12/1.29	21.0/4.8	25.5/3.9	35.4/3.0	6.2/**9.1**	11.1/5.9	26.3/3.1	10.0/3.2	0/**77.5**
16	0.39/3.01	0.83/3.01	1.00/3.01	**1.58/3.01**	1.22/3.01	0.84/3.01	9.9/4.6	14.5/3.5	31.3/2.1	4.9/**6.8**	8.2/4.1	17.7/2.2	10.0/2.3	0/**33.3**
17	0.00/0.16	0.01/0.16	0.01/0.16	**0.01/0.16**	0.00/0.16	0.00/0.16	4.9/23.9	9.8/10.7	18.7/6.3	1.3/**94.6**	4.1/31.0	15.2/6.4	10.0/6.3	0/**626.7**
18	0.00/1.56	0.01/1.56	0.01/1.56	**0.01/1.56**	0.00/1.56	0.00/1.56	2.5/15.8	5.2/9.1	13.6/3.9	1.3/**18.1**	2.7/10.4	8.5/3.9	10.0/4.2	0/**63.9**
19	2.55/2.48	6.57/2.48	6.38/2.48	**8.59/2.48**	4.66/2.48	3.19/2.48	20.7/5.1	29.5/4.0	44.7/3.1	9.4/**8.7**	14.6/5.6	29.5/3.1	10.8/3.3	0/**40.3**
20	0.01/0.09	0.04/0.09	0.07/0.09	**0.09/0.09**	0.04/0.09	0.02/0.09	1.1/68.5	3.4/24.4	18.6/5.4	0.9/**69.0**	2.1/25.7	10.2/5.4	10.0/5.5	0/**1157.8**
Mean	0.66/1.14	0.97/1.14	0.92/1.14	1.09/1.14	0.37/1.14	0.27/1.14	7.97/18.91	11.47/10.44	20.58/5.13	2.94/**32.77**	5.65/**15.56**	13.90/5.44	10.12/5.42	0/**585.62**

best speed-up scores in **bold**. Notice that speed-up measures the overall execution time of strong mutation divided by the overall execution time of a mutation strategy. Hence, higher values denote a larger improvement in execution time.

We observe that weak mutation shows the highest speed-up scores since it requires only one test suite execution against the original program. Except for weak mutation, FCA achieves the highest speed-up scores in 19 out of 20 cases. *FCA* is also faster than *random sampling*, which selects 10% mutants for strong mutation. Indeed, the former is 6.6 times faster than the latter on average, with a minimum speed-up of 2.6X (in `vraptor`) and a maximum one of 15.4X (in `stream-lib`). This is because *FCA* suggests on average less than 10% of mutants (with a minimum of 0.4% of mutants) to evaluate in strong mutation analysis. Instead, the sampling strategy constantly (and randomly) selects 10% of mutants to execute.

The only exception to the previous finding is represented by `distributedlog` for which *random sampling* is faster than *FCA*. In this case, the total percentage of mutants that are injected into statements covered by the test suite (*reachability condition*) is fairly low, being 2.8%. Thus, random sampling can achieve a considerable speed-up if we leverage the coverage-based optimisation, i.e., if we skip uncovered mutants (i.e., mutants of uncovered statements). Instead, *FCA* selects almost twice as many mutants for these projects.

To further ease the comparison, in Table IV we underline the compression strategies that achieve better speed-up scores than random sampling in each project. We observe that *overlap*, *FCA+mop* and *overlap+mop* outperform random sampling in terms of speed-up for 19 projects out of 20. On average, they are respectively 3.7X, 3.0X, and 2.1X faster than random sampling. It is worth noticing that the number of selected mutants does not directly determine the overall speed-up. For example, for the project `pac4j`, *overlap* selected 21.0% of mutants, which is larger than the percentage of mutants selected by mutation sampling (i.e., 10%). However, *overlap* achieves a larger speed-up of 4.8X against 3.1X of mutation sampling. The reason is that compression strategies uses weak mutation information to further filter out the unnecessary test executions, while mutation sampling does not.

TABLE V
RANKING PRODUCED BY THE FRIEDMAN'S (SMALLER RANK INDICATES BETTER SPEED-UP) AND STATISTICAL SIGNIFICANCE BY THE CONOVER'S POST-HOC PROCEDURE.

ID	Algorithms	Rank	Significantly better than
(1)	Weak Mut.	1.00	(2), (3), (4), (5), (6), (7), (8)
(2)	FCA	2.10	(3), (4), (5), (6), (7), (8)
(3)	Overlap	3.50	(5), (6), (7), (8)
(4)	FCA + Mop	3.55	(6), (7), (8)
(5)	overlap + Mop	5.05	(6), (7), (8)
(6)	Random Samp.	6.25	(8)
(7)	FCA + Mloc	6.65	(8)
(8)	overlap + Mloc	7.90	-

From the comparison of the six compression strategies, we observe that including mutant location leads to selecting more mutants for strong mutation, thus, reducing the overall speed-up. For example, *overlap+mloc* achieves lower speed-up scores than *overlap* in all 20 projects. Moreover, by comparing the two strategies based on mutation location knowledge (i.e., *overlap+mloc*, and *fca+mloc*) with random sampling, we observe that the differences in terms of speed-up are small. Indeed, the average speed-up scores of *overlap+mloc*, *fca+mloc* and *random sampling* are 5.13, 5.44 and 5.42, respectively. Instead, selecting mutants according to mutation operator generates a lower number of mutants to evaluate in strong mutation compared to mutation location.

Our findings are confirmed by Friedman's test: the mutation strategies are statistically different in terms of speed-up scores (p-value = 10^{-16}). According to Conover's procedure, *weak mutation* and FCA statistically outperform all other mutation strategies. Moreover, random sampling is ranked sixth and is statistically more efficient than *overlap+mloc* only, although the difference is marginal as suggested by the average scores reported in Table IV. Instead, FCA, *overlap*, *FCA+mop* and *overlap+mop* are statistically superior to random sampling.

Overhead In the previous paragraphs, we observed that

most of mutation compression strategies are more efficient than random sampling. Here, we investigate the overhead that is due to the different steps that such strategies implement. In the following, we consider as *compression overhead* the execution time needed to compute the maximal and/or overlapped groupings; while the *overall overhead* is the sum of the *compression overhead*, the time for running all tests once for weak mutation, and the time to select the mutants.

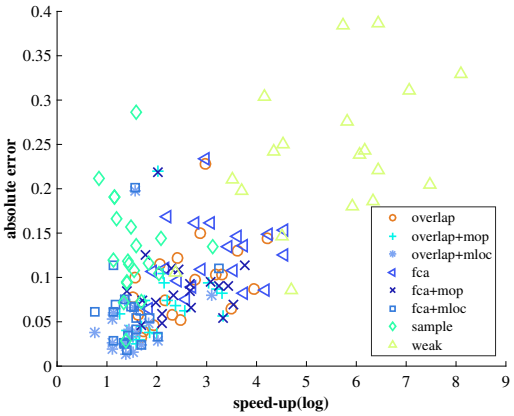
Table IV reports the *compression overhead* and *overall overhead* as a percentage (ratio) of the full execution time of each strategy, which also includes the time needed to run the selected tests and mutants for strong mutation. The highest values for each project are highlighted in **bold** face. From Table IV, we can observe that the compression overhead takes up less than 0.001% of the total execution time; thus, it is negligible with respect to the execution time of evaluating the selected mutants for strong mutation. The overall overhead accounts for up to 9.30% of the total execution time and weak mutation represents the larger portion of this overhead. Among the 20 projects, the overall overhead of the strategy *FCA* is likely higher than the other compression strategies. However, the differences among them are lower than 0.1%.

Weak mutation scores best among the eight techniques in terms of speed-up. Without considering weak mutation, four mutant compression strategies are statistically more efficient (have better speed-up scores) then random sampling.

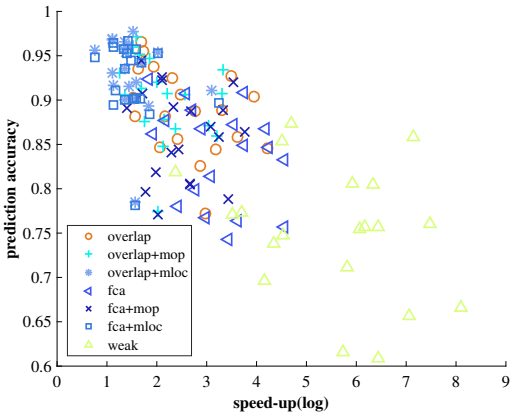
C. RQ3: trade-offs

From the results of **RQ1** and **RQ2**, it is clear that the mutation strategies that perform best in terms of accuracy are also the more expensive to perform. Weak mutation and *FCA* grouping strategies perform best in terms of speed-up, while the *overlapped* grouping strategies, and *overlap+mloc* in particular, better approximate the strong mutation score. Therefore, in this section, we analyse the trade-offs between speed-up, absolute error and accuracy.

From Figure 3, we observe that weak mutation achieves the best speed-up, while its absolute error and accuracy typically score worst when compared to other techniques. FCA comes second in terms of the overall speed-up, but its absolute error as well as accuracy are better than weak mutation for most of the projects. We notice that *overlap* is slightly slower than FCA, but shows a small improvement in both absolute error and accuracy when compared to FCA. *FCA+mop* and *overlap+mop* have quite similar trade-offs considering speed-up and absolute error as their data points are very close to each other. However, in terms of speed-up and accuracy, *fca+mop* is slightly faster than *overlap+mop*, while *overlap+mop* is more accurate than *fca+mop*. Moreover, *overlap+mloc*, *fca+mloc* and *random sampling* have the same speed-up score; however, the absolute error of *random sampling* is higher than for the other two strategies. *Overlap+mloc* and *fca+mloc* are the most accurate strategies in terms of both absolute error and accuracy, but their speed-up performance is the least good.



(a) Speed-up vs. Absolute Error



(b) Speed-up vs. Accuracy

Fig. 3. Graphical comparison of the eight mutation strategies in terms of speed-up, absolute error and accuracy.

TABLE VI
NUMBER OF PROJECTS FOR WHICH EACH STRATEGY M_j PROVIDES THE BEST SPEED-UP SCORE AT DIFFERENT THRESHOLDS.

Absolute Error	Overlap			FCA			Random Sampling	Weak Mut.
	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc		
≤5%	2	4	**10**	0	1	3	0	0
≤10%	4	1	1	5	**7**	1	0	1
≤15%	1	0	1	**11**	3	0	0	4
overall	7	5	12	**16**	12	4	0	5

(a) error rate

Accuracy	Overlap			FCA			Random Sampling	Weak Mut.
	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc		
≥95%	2	2	**9**	0	0	7	-	0
≥90%	**5**	4	2	3	1	**5**	-	0
≥85%	4	2	1	**8**	1	1	-	3
overall	11	8	12	11	2	**13**	-	3

(b) accuracy

Table VI shows the number of projects for which each strategy M provide the best speed-up score at different thresholds of absolute error (σ_{e_i}) or accuracy (σ_{a_i}). As we can observe, *overlap+mloc* has the highest speed-up score when considering an absolute error ≤5% for 10 projects out of 20. This indicates that using mutation location leads to more accurate estimations of the actual strong mutation. When considering a 10% error rate threshold, *FCA+mop* and *FCA*

show the largest speed-up for the majority of projects. Instead, when the goal is to reach an absolute error $\leq$15%, FCA has the best speed-up scores for 11 out of 20 projects. Moreover, we notice that random sampling performs worst as it has speed-up scores that are always lower then the other mutation strategies at the same (or higher) level of absolute error.

Similar results can be observed when considering different thresholds for accuracy. As reported in Table VI, *overlap+mloc* and *fca+mloc* show the best speed-up for accuracy $\geq$95% for the relative majority of the projects. When considering an accuracy $\geq$90%, *overlap* and *fca+mloc* show the best speed-up for 5 out of 20 projects each. Instead, if we focus on 85% of accuracy, we observe that FCA is the best approach to choose as it shows a speed-up ranging from 6.3X up to 94.3X compared to strong mutation. Finally, if we focus on 85% accuracy, we observe that FCA is the best approach to choose.

Overlap+mloc provides the best speed-up scores when the goal is to achieve an accuracy $>$95% or an absolute error $<$0.05. Other mutation compression strategies provide larger speed-up, but with a corresponding decrease in accuracy. Random sampling is less accurate and/or slower than all compression strategies.

D. Discussion

Looking at all the results, we can observe that random sampling with 10% sampling ratio is able to speed up (strong) mutation testing from 2.0 to 22.7X with an absolute error within 15% for 80% of the projects. Mutation sampling is also easy to apply in mutation tools as it does not require any prerequisite knowledge of the program context and mutation operators. However, *mutation strategies based on compression techniques achieve better speed-up (i.e., are more efficient) and/or lower absolute error than random sampling*. For example, *overlap+mloc* yields an absolute error which is always lower than 9% with a speed-up ranging between 2.0 to 53X. This represents an important finding if we consider the recent study by Gopinath et al. [17], which showed that other mutation reduction techniques (including *e-selective*) provide small or negligible improvements in effectiveness and are more expensive compared to random sampling.

Another disadvantage of random sampling is that it can estimate the overall mutation score, but cannot estimate whether each mutant is strongly killable or not (it only does for the sampled mutants). The overall mutation score is of course very important when assessing the test suite quality at a high level; however, Coles [39, slide 57] observed that programmers prefer to obtain specific insights into which mutants their test suite is able to kill. From this perspective, mutation compression strategies select a subset of "representative" mutants for the programmers to investigate. In addition, *overlap+mloc* and *FCA+mloc* guarantee that for every possible statement that can be mutated, at least one mutant will be selected. Although this may negatively affect the speed-up compared to random sampling, programmers can benefit from the killable mutant results at every possible mutant location.

VI. Threats to Validity

Threats to external validity: Our results are based on mutants generated by the operators implemented in EvoSuite; these results might be different when using other mutation tools [40]. With regard to the subject selection, we chose 18 out of the 20 projects from GitHub's top starred 3000 repositories; the selected projects differ in size, number of test cases and application domain.

Threats to internal validity: The main threat for our study is the implementation of the compression strategies. For FCA, we use its implementation available in MATLAB [41], which is a well-known scientific software. For the instrumentation and the mutation operators, we relied on their implementation available in EvoSuite [42]. Moreover, we carefully reviewed and tested all code for our study to eliminate potential faults in our implementation.

Threats to construct validity: The main threat is the measurement we used to evaluate our methods. We minimise this risk by adopting evaluation metrics that are widely used in research, as well as proper statistical analysis to assess the significance.

VII. Conclusions

In this paper, we have conducted a detailed investigation of different compression techniques to speed up mutation testing based on the work by our previous work [18]. We have enhanced our original *FCA-based compression strategy* in two distinct ways: (1) by proposing a novel mutant clustering algorithm, *overlapped grouping*, in addition to *FCA*; (2) by incorporating mutation location and mutation operator information in the compression procedure. Thereby, we have introduced and investigated six compression strategies based on two clustering algorithms and three mutant selection strategies.

The results of an empirical study with 20 open-source projects show that mutant compression techniques can effectively speed up strong mutation testing up to 94.3 times with an accuracy $>$ 90%. FCA is the fastest strategy while $overlap + mloc$ is the most accurate. In comparison, *weak mutation* attains a higher absolute error (23%) and lower accuracy (75%). *Random sampling* with 10% as sampling percentage is statistically less accurate than all mutant compression strategies, and worse in terms of speed-up than four compression strategies (excluding the two with knowledge of mutation locations).

Another important finding is that *mutation location trumps mutation operator information when selecting mutants to evaluate for strong mutation.* Hence, researchers should take into account the mutation location in addition to the mutation operators when detecting redundant or subsuming mutants (e.g. [25]–[27]). This is a clear invitation for future work.

Since our results are encouraging, we envision the following future work: (i) combining mutant location and mutation operator information; (ii) investigating other compression methods,

such as Principal Component Analysis [43]; (iii) applying compression techniques in mutation-based test case generation [44], [45].

REFERENCES

[1] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[2] J. Offutt, "A mutation carol: Past, present and future," *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[3] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[4] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[5] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *ICST workshops*. IEEE, 2009, pp. 220–229.

[6] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, New Haven, CT, USA, 1980.

[7] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[8] A. T. Acree Jr, "On mutation," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

[9] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996.

[10] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Eng.*, no. 4, pp. 371–379, 1982.

[11] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Int'l Symp. on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 315–326.

[12] R. Untch, A. J. Offutt, and M. J. Harrold, "Mutation testing using mutant schemata," in *Int'l Symp. Software Testing and Analysis (ISSTA)*, 1993, pp. 139–148.

[13] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A Novel Method of Mutation Clustering Based on Domain Analysis," in *SEKE*, 2009, pp. 422–425.

[14] S. Hussain, "Mutation clustering," *Ms. Th., Kings College London, Strand, London*, 2008.

[15] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2015, pp. 1–10.

[16] J. Strug and B. Strug, "Machine learning approach in mutation testing," in *IFIP International Conference on Testing Software and Systems*. Springer, 2012, pp. 200–214.

[17] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 854–874, Sept 2017.

[18] Q. Zhu, A. Panichella, and A. Zaidman, "Speeding-up mutation testing via data compression and state infection," in *2017 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, March 2017, pp. 103–109.

[19] R. Wille, "Formal concept analysis as mathematical theory of concepts and concept hierarchies," in *Formal concept analysis*. Springer, 2005, pp. 1–33.

[20] R. H. Untch, "Mutation-based software testing using program schemata," in *Proc. annual Southeast regional conf.* ACM, 1992, pp. 285–291.

[21] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," in *Proc. ESEC/FSE*. ACM, 2009, pp. 297–298.

[22] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Int'l Conf. Automated Softw. Engineering (ASE)*. IEEE, 2011, pp. 612–615.

[23] "Available mutation operations (PIT)," http://pitest.org/quickstart/mutators/, [Online; accessed 10-August-2016].

[24] Y.-S. Ma and S.-W. Kim, "Mutation testing cost reduction by clustering overlapped mutants," *J. Systems and Software*, vol. 115, pp. 18–30, 2016.

[25] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, pp. 11–20.

[26] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Software testing, verification and validation workshops (ICSTW), 2014 IEEE seventh international conference on*. IEEE, 2014, pp. 176–185.

[27] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 21–30.

[28] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proc. Int'l. Symp. on Software Testing and Analysis (ISSTA)*. ACM, 2013, pp. 235–245.

[29] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 342–353.

[30] A. J. Offutt, "Automatic test data generation," Ph.D. dissertation, Georgia Institute of Technology, 1988.

[31] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.

[32] Q. Zhu, "Mutation testing Tools," https://zenodo.org/badge/latestdoi/122769075, [Online; accessed 24-February-2018].

[33] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 511–522.

[34] A. Richardson, "Nonparametric statistics for non-statisticians: A step-by-step approach by gregory w. corder, dale i. foreman," *International Statistical Review*, vol. 78, no. 3, pp. 451–452, 2010.

[35] S. García, D. Molina, M. Lozano, and F. Herrera, "A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization," *Journal of Heuristics*, vol. 15, no. 6, p. 617, 2008.

[36] A. Panichella and U. R. Molina, "Java unit testing tool competition - fifth round," in *10th IEEE/ACM International Workshop on Search-Based Software Testing (SBST)*, 2017, pp. 32–38.

[37] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[38] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[39] H. Coles, "Mutation testing - a practitioners perspective," https://github.com/hcoles/slides/blob/master/slides.pdf, accessed: 2017-05-09.

[40] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 571–582.

[41] MATLAB, *version 9.3.0 (R2017b)*. Natick, Massachusetts: The MathWorks Inc., 2017.

[42] G. Fraser and A. Arcuri, "A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, p. 8, 2014.

[43] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[44] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2014.

[45] A. Panichella, F. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.