Cerebro: Static Subsuming Mutant Selection

Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon

Abstract—Mutation testing research has indicated that a major part of its application cost is due to the large number of low utility mutants that it introduces. Although previous research has identified this issue, no previous study has proposed any effective solution to the problem. Thus, it remains unclear how to mutate and test a given piece of code in a best effort way, i.e., achieving a good trade-off between invested effort and test effectiveness. To achieve this, we propose *Cerebro*, a machine learning approach that *statically* selects subsuming mutants, i.e., the set of mutants that resides on the top of the subsumption hierarchy, based on the mutants' surrounding code context. We evaluate *Cerebro* using 48 and 10 programs written in C and Java, respectively, and demonstrate that it preserves the mutation testing benefits while limiting application cost, i.e., reduces all cost application factors such as equivalent mutants, mutant executions, and the mutants requiring analysis. We demonstrate that *Cerebro* has strong inter-project prediction ability, which is significantly higher than two baseline methods, i.e., supervised learning on features proposed by state-of-the-art, and random mutant selection. More importantly, our results show that *Cerebro*'s selected mutants lead to strong tests that are respectively capable of killing 2 times higher than the number of subsuming mutants killed by the baselines when selecting the same number of mutants. At the same time, *Cerebro* reduces the cost-related factors, as it selects, on average, 68% fewer equivalent mutants, while requiring 90% fewer test executions than the baselines.

Index Terms—Mutant, mutation, mutation testing, subsuming mutant, mutant prediction, static selection, static mutant selection, static subsuming mutant selection, static subsuming mutant prediction, encoder-decoder, machine translation, tf-seq2seq

1 Introduction

Research and practice with mutation testing has shown that it can effectively guide developers in improving their test suite strengths [3], [14], and can be used to reliably compare test techniques [5], [50]. A key issue though, is that it is expensive, as a large number of mutants are involved, the majority of which are of low utility, i.e., they do not contribute to the testing process [3], [27], [30]. This means that mutation testers should filter their mutant sets using manual analysis to identify equivalent mutants [9], and perform numerous test executions to discard mutants that do not provide testing value, i.e., mutants that are detected by the tests designed to detect other mutants [3], [27], [30].

Working with large real-world systems makes the problem almost intractable due to the vast numbers of mutants involved. Test execution overheads alone can limit the scalability of the technique. For instance, in our experiments, we needed around 48 hours to execute the mutants for a single component of the systems we examined. At the same time the manual effort required by testers is escalated with larger programs as the number of mutants grows proportionally to program size.

To reduce application cost, it is imperative to limit the number of mutants to those that are actually useful, prior to any manual mutant analysis or test execution. Thus, we need to identify which mutants are killable in order to limit the manual effort involved in their identification, and also to identify the mutants that are subsuming (disjoint)[1], in order to reduce unnecessary computations, and to provide accurate adequacy measurements [46].

This problem is known as the mutant selection problem [47] and has been studied in the form of selective mutation [43], [68], i.e., restricting the number of transformations to be used, with limited success [11], [34]. Though, the key issue with mutant selection is the simple syntactic-based nature of the selection process. The issue is that mutants are introduced everywhere with respect to simple language operators, e.g., by replacing an operator with another, that completely ignore the program and particular location semantics. This operator matching mutant selection has the unfortunate effect of introducing mutants independent of their context and program semantics.

We propose *Cerebro* [2], a machine learning technique that learns to identify interesting mutants given their context. In particular we learn the associations between mutants and their surrounding code. Our learning scope is a relatively small area around the mutation point that differentiates

- *Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon are with the University of Luxembourg, 4365 Esch-sur-Alzette, Luxembourg. E-mail: {aayush.garg, milos.ojdanic, renzo.degiovanni, michail.papadakis, yves.letraon}@uni.lu.*
- *Thierry Titcheu Chekam is with SES, 6815 Betzdorf, Luxembourg. E-mail: thierry.titcheu.chekam@ses.com.*

Manuscript received 26 April 2021; revised 29 December 2021; accepted 30 December 2021. Date of publication 11 January 2022; date of current version 9 January 2023.
This work was supported by Luxembourg National Research Funds (FNR) through the INTER Project under Grant INTER/ANR/18/12632675/SATOCROSS.
(Corresponding author: Aayush Garg.)
Recommended for acceptance by L. Mariani.
Digital Object Identifier no. 10.1109/TSE.2022.3140510

1. The term disjoint mutants refers to a minimal subset of mutants that need to be killed in order to reciprocally kill the original set [30], [45].

2. Cerebro is a fictional device appearing in Marvel comics used by the X-Men to detect human mutants. More details in https://en.wikipedia.org/wiki/Cerebro.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

locally, the mutants that are useful from those that are not. This allows mutating the program elements to fit best to their context, instead of mutating entire codebases with every possible transformation, enabling inter-project predictions.

Cerebro operates at lexical level, with a simple code preprocessing. In particular, a mutant and its surrounding code is represented as a vector of tokens where all literals and identifiers, i.e., user defined variables, types, and method calls, are replaced with predefined, hence predictable, identifier names. This allows restricting the related vocabulary and learning scope to a relatively small fixed size of tokens around the mutation points. Learning is performed using a powerful and language-agnostic machine translation technique [8] that we train on related code fragments and their labels.

We consider useful, the subset of mutants that resides on top of the subsumption hierarchy and subsumes the others [33], *aka subsuming mutants* [27], for the set of all possible mutant instances produced by a given set of mutation operators. Mutant M_1 subsumes mutant M_2 if every test case detecting M_1 also detects M_2. This implies that the tests detecting the subsuming mutant will also detect the subsumed ones thereby making subsumed mutants redundant.

We implemented *Cerebro* and evaluated its ability to predict (inter-project predictions) subsuming mutants on a large set of programs, composed of 48 C programs (CoreUtils) and 10 Java projects (Apache Commons, Joda-Time, and Jsoup). Our results demonstrate that *Cerebro* significantly outperforms both, random mutant selection and a supervised machine learning approach (used by previous research) on both, C and Java benchmarks.

In particular, our results show that *Cerebro* significantly outperforms the baselines. In Java projects, *Cerebro* obtained 2.81 times higher MCC[3] values, an improvement of 82% in F-measure, 68.88% in Precision, and 85.71% in Recall over the state-of-the-art supervised machine learning. In C programs, *Cerebro* obtained 2.76 times higher MCC values, 3.72 times higher precision, and slightly increased Recall value (4% higher). The improvement measured in F-measure is approximately 65%.

To put the predictions into a context and understand its influence on mutation testing, we also validated *Cerebro* in a controlled simulation of the envisioned use case. In particular, we simulate a scenario where testers are guided by mutation testing, i.e., they design test cases based on mutants. Therefore, fewer mutants imply less effort, while stronger mutants imply stronger tests. Our analysis shows that *Cerebro* achieved more than twice the subsuming mutation scores[4] in both, C and Java programs that we use. At the same time *Cerebro* required significantly less effort in terms of both, analyzed equivalent mutants and test executions. In C programs, 3.70% of the mutants analyzed by *Cerebro* are equivalent, while 55.56% and 53.33% analyzed by random mutant selection and supervised learning, respectively are equivalent; *Cerebro* also required 91% fewer test executions than random selection and supervised learning, respectively. In Java programs, *Cerebro* required the analysis of 41% and 36% fewer equivalent mutants, and 92% and 87% fewer test executions than random mutant selection and supervised learning, respectively.

All-in-all our paper makes the following contributions:

1) We present *Cerebro*, a powerful static subsuming mutant selection technique.
2) We provide evidence suggesting that *Cerebro* successfully predicts subsuming mutants with 0.85 Precision, 0.33 Recall and 0.46 MCC.
3) We show that *Cerebro* significantly outperforms the current state-of-the-art, i.e., random mutant selection and previously proposed machine learning technique, by revealing 2 times the subsuming mutants, while analyzing 64% to 67% fewer equivalent mutants and requiring 89% to 92% fewer test executions.

The remainder of the paper is organized as follows. Section 2 introduces preliminary concepts necessary in subsequent sections. Section 3 describes the envisioned use case for *Cerebro* and elaborates on a particular motivating example. Section 4 describes the approach in detail. Section 5 introduces the research questions and Section 6 details the experimental setup. The results of our experimental evaluation are summarized in Section 7. We discuss threats to validity in Section 9. In Section 8 we also discuss the impact of the abstraction process and mutants' context size on *Cerebro*'s prediction performance. Finally, we discuss related work in Section 10, and present our conclusion and future work in Section 11.

2 BACKGROUND

2.1 Subsuming Mutants

Mutation is a test adequacy criterion in which test requirements are represented by mutants that are obtained by performing slight syntactic modifications to the original program. Then, the tester needs to design test cases in order to *kill* the mutants, i.e., to distinguish the observable behavior between the mutant and the original program. Some mutants cannot be killed as they are functionally *equivalent* to the original program. Hence, the quality of a test suite is measured by the mutation (adequacy) score, a percentage metric obtained by the ratio of killed mutants over the total number of (non-equivalent) generated mutants.

Mutation testing is a promising, empirically validated software testing technique that hasn't achieved its full potential yet [47]. It is often considered as computationally expensive, mainly due to the large number of mutants that it introduces, which require analysis and execution with the related test suites. One may notice that the number of mutants is disproportionate with the number of test cases to kill them, since one test case can kill several mutants at the same time. Thus, the effort put into analyzing and executing mutants that do not help to improve test suites is wasted. Hence, it is desirable to analyze only the mutants that add value, i.e., subsuming mutants [3], [27], [30], [33].

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 killable mutants. Given

3. The *Matthews Correlation Coefficient* (MCC) [40] is a reliable metric of the quality of prediction models [55], relevant when the classes are of very different sizes, e.g., in case of C programs, 10.2% subsuming mutants (positives) over 89.8% non-subsuming mutants (negatives).

4. *Subsuming mutation score* (MS*) is the ratio of the killed and the total number of subsuming mutants.

two mutants M_1 and M_2, it is said that M_1 subsumes M_2 if every test suite T killing M_1 also kills M_2. Unfortunately, identifying subsuming mutants is undecidable as it is not possible to know a mutant's behavior under every possible input. Thus, researchers typically approximate them through test suites [3], [27], [34], [45], [46].

More precisely, let M_1, M_2 and T be two mutants and a test suite, respectively, where $T_1 \subseteq T$ and $T_2 \subseteq T$ are the set of tests from T that kill mutants M_1 and M_2, respectively, and $T_1 \neq \emptyset$ and $T_2 \neq \emptyset$, indicating that both M_1 and M_2 are killable mutants. We will say that mutant M_1 subsumes mutant M_2, if and only if, $T_1 \subseteq T_2$. In case $T_1 = T_2$, we say that mutants M_1 and M_2 are indistinguishable for T. The set of mutants which are both killable, and subsumed only by indistinguishable mutants are called *subsuming mutants*.

For example, if we have a mutant set of 3 mutants (M_1, M_2, and M_3) and a test set $T = \{t_1, t_2, t_3\}$, where M_1 is killed by $T_1 = \{t_1\}$; M_2 is killed by $T_2 = \{t_1, t_2\}$; and M_3 is killed by $T_3 = \{t_3\}$. We can notice that every time that we run a test (t_1) to kill mutant M_1 we will also kill mutant M_2. However, the opposite does not hold. Thus, we have two subsuming mutants, i.e., M_1 and M_3.

Subsuming mutation score (MS*) is the ratio between killed subsuming mutants over the total number of subsuming mutants [46]. Subsuming mutation score has been proposed [3], [30], [46] as a reliable metric to evaluate the effectiveness of testing techniques as it does not consider the presence of subsumed mutants. Subsumed mutants can artificially inflate the mutation score of a testing technique and can mislead its apparent ability to detect faults. For instance, following our previous example, a test suite $\{t_1, t_2\}$ kills 66.7% of all the mutants (i.e., M_1 and M_2), but 50% of the subsuming ones (M_3 is not killed).

Interestingly, killing subsuming mutants leads to the killing of all killable mutants, thus, testers needs to focus mutation analysis on subsuming mutants. The problem though, is that one needs to know the subsumption relations between mutants in advance, before starting to analyze the mutants and designing tests. To deal with this issue, we introduce *Cerebro*, a *static* technique that predicts subsuming mutants without requiring any dynamic analysis, with the aim to help testers decide on which mutants to use when performing mutation-guided test generation [23], [48].

2.2 Machine Translation

Machine Translation can be considered as a transformation function $transform(X) = Y$, where the input $X = \{x_1, x_2, \ldots, x_n\}$ is a set of *entities* that represents a component to be transformed, to produce the output $Y = \{y_1, y_2, \ldots, y_n\}$, which is a set of entities that represent a transformed (desired) component. In the training phase, the transformation function learns on the example pairs (X, Y) available in the training dataset. In our context, X contains the source code with an annotation that indicates the location and type of the mutation operator applied, and Y contains the same information, plus a label that indicates whether the mutant is subsuming or not.

The transformation function is trained to append the label to a given mutant by training the function on the example pairs (Code+MutationAnnotation, Code+MutationAnnotation+Label), where Code+MutationAnnotation represents the source code with an annotation in the statement to indicate the mutation operator type applied. This learned transformation is used as our prediction model for predicting subsuming mutants. Among the several machine translation algorithms that have been suggested over the past years, we use the RNN Encoder-Decoder which is established and is used by many recent studies [58], [60], [61].

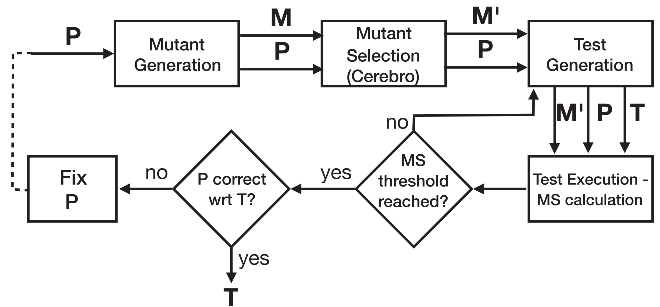


Fig. 1. *Cerebro* Mutation Testing process. Given a program P and a mutant set M, *Cerebro* selects from M a subset of mutants M' to be used for test generation. M' is then used to in Test generation, test execution and mutation score calculation steps.

2.3 RNN Encoder-Decoder Architecture

The RNN Encoder-Decoder machine translation is composed of two major components: an RNN Encoder to encode a sequence of terms x into a vector representation, and an RNN Decoder to decode the representation into another sequence of terms y. The model learns a conditional distribution over an (output) sequence conditioned on another (input) sequence of terms: $P(y_1; \ldots; y_m | x_1; \ldots; x_n)$, where n and m may differ. For example, given an input sequence $x = Sequence_{in} = (x_1; \ldots; x_n)$ and a target sequence $y = Sequence_{out} = (y_1; \ldots; y_m)$, the model is trained to learn the conditional distribution: $P(Sequence_{out} | Sequence_{in}) = P(y_1; \ldots; y_m | x_1; \ldots; x_n)$, where x_i and y_j are space-separated tokens. A bi-directional RNN Encoder [8] (formed by a backward RNN and a forward RNN) is considered the most efficient to create representations as it takes into account both past and future inputs while reading a sequence [6].

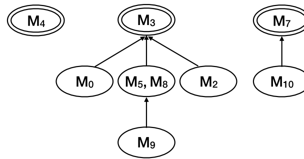
3 Use Case Scenario and Motivation

3.1 Use Case Scenario

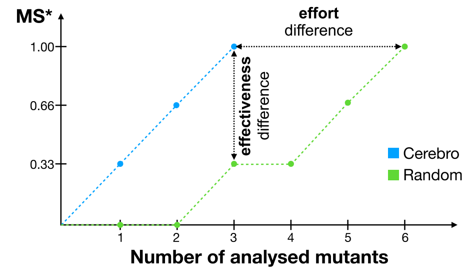
Fig. 1 shows an overview of how the testing process is performed when it is guided by mutation. We adapted this figure from the one published in [4, Figure 5.2]. Given a program P as input, the mutation testing process starts by creating a set M of mutants forming the test requirements. Test requirements are satisfied when tests kill the mutants. Since the number of mutants are excessive and form the key cost factor of mutation testing [47], testers select a subset M' of mutants from M to focus on their analysis. Then, testers pick a mutant $m \in M'$ and design a test t capable of killing m or judge it as equivalent and discard it. The process is repeated until the design of test is capable of killing a predefined ratio of mutants (threshold). Finally, the designed test suite T is used to check the correctness of program P (w.r.t. test suite T). If test suite T detects some bug in program P, then P has to be fixed and the same mutation testing procedure can again be employed.

```
1  int max(int a, int b, int c){
2    if (a >= b && a >= c) //M0: (a < b && a >= c)
                           //M1: (a >= b && a > c)
                           //M2: (a >= b || a >= c)
                           //M3: (true && a >= c)
3      return a; //M4: return b;
4    else if (b >= a && b >= c)//M5: (b < a && b >= c)
                           //M6: (b >= a && b > c)
                           //M7: (b >= a || b >= c)
                           //M8: (false && b >= c)
5      return b; //M9: return a;
6    else
7      return c; //M10: return 0;
8  }
```

(a) Code and mutants for function `max`.



(b) Subsuming Mutants Graph for function `max`.



(c) Our motivating example shows that mutants selected by *Cerebro* lead to stronger test suites than those designed to kill randomly selected mutants, when equal number of mutants is analyzed.

Fig. 2. The example shows that by analyzing only the three subsuming mutants M_3, M_4 and M_7 is enough for covering all 9 killable mutants. Particularly, mutants M_1 and M_6 are equivalents.

It is worth mentioning that there are two major cost factors in mutation testing, these are the equivalent and subsumed mutants. This is because they introduce overheads both during test generation and test execution, leading to minor test effectiveness improvements. Therefore, to reduce mutation testing effort while preserving its effectiveness, it is essential to focus on subsuming mutants.

Hence, we develop *Cerebro*, a machine learning technique that learns from mutants' surrounding context to predict which mutants are subsuming. Given the input program, P and the set M of mutants, *Cerebro* selects a subset M' of mutants that is probably subsuming (predicted as subsuming by *Cerebro*), to be used for mutation testing (to guide testers and evaluate test effectiveness). Based on M', testers and/or automatic test generation techniques can focus on the few strong mutants and design effective test cases.

3.2 Motivating Example

Let us consider the code snippet of function `max` of Fig. 2a, which takes three integers as input and returns the maximum number among them. Also, consider (for simplicity) that we have the 11 mutants shown in the figure. For instance, mutant M_0 mutates sub-expression `a >= b` of line 2 into `a < b`. Similar mutations on relational operations were applied to produce mutants M_1, M_3, M_5, M_6 and M_8. Mutants M_2 and M_7 replace the conjunction (`&&`) by the disjunction (`||`). While mutants M_4, M_9 and M_{10} replace the returned variable name by other variable name or constant (M_{10} replaces variable name `c` by constant `0`).

For the sake of the thorough demonstration, we observed scenarios under the following testing conditions: A test case invoking `max(1,2,0)` and expecting `2` as a result, kills mutant M_3, as well as, mutants M_0, M_2, M_5, M_8, and M_9. But tests invoking `max(2,0,1)`, `max(1,0,2)`, and `max (0,2,1)` will kill mutants M_0, M_2, M_5, M_8, and M_9, except M_3. Fig. 2b shows a graph representation of the subsumption relation between the 9 killable mutants. Moreover, Fig. 2b shows that M_3 subsumes M_0, M_5, M_8 and M_2. Particularly notice that mutants M_5 and M_8 are indistinguishable, since they are killed by the same tests, and subsume mutant M_9. Although, mutants M_1 and M_6 are equivalent.

In summary, mutants M_3, M_4 and M_7 are subsuming, indicating that in order to kill every killable mutant it is sufficient to kill only these 3 subsuming mutants.

Cerebro will take as input the program `max` and the set of mutants, and it will point to those that are most likely subsuming. In an ideal scenario, *Cerebro* would point only to M_3, M_4 and M_7, but it is possible, as in every machine learning based technique, that it does some mistakes, i.e., incorrect predictions of subsuming mutants, pointing to some non-subsuming (subsumed or equivalent mutants) as subsuming.

For instance, consider the case in which *Cerebro* predicts M_3 and M_4 and M_{10} as subsuming mutants. Therefore, a tester will incrementally design test cases to kill all the predicted mutants. Assume that the tester starts by analyzing mutant M_3 and designs a test to kill it, e.g., by invoking `max(1,2,0)`. This test does not kill the rest of the selected mutants. The tester then proceeds to analyze the surviving mutant M_4, for which he/she designs a test that invokes `max(2,0,1)` to kill it. Finally, the tester designs a test by invoking `max(0,1,2)`, which kills mutant M_{10} and also (non selected) subsuming mutant M_7. Notice that this test suite designed to kill all mutants selected by *Cerebro* progressively increments the MS*: first test kills subsuming mutant M_3 leading to a MS* of 33.33%; second test kills subsuming mutant M_4, obtaining 66.66% of MS*; and finally, third test kills collaterally subsuming mutant M_7 leading to a MS* of 100%.

Consider a scenario in which mutants are selected *randomly*. For instance, assume that M_9 is the first one to be selected for analysis for which a test case invoking `max(0,2,1)` is designed to kill it. This test collaterally kills mutants M_5 and M_8, but it does not kill any subsuming mutant. Then, assume that equivalent mutant M_1 is randomly selected, adding no value to the testing process, but requiring analysis anyway. Afterwards mutant M_0 is randomly selected for which a test case invoking `max(2,0,1)` is designed to kill it, that fortunately also kills subsuming mutant M_4. Then, mutant M_2 is randomly selected for which the tester designs a test to kill it by invoking `max(1,0,2)`. This test also kills mutant M_{10}, but no subsuming mutant is killed. After that, tester randomly selects mutant M_3 for analysis and designs a test by invoking `max(1,2,0)` to kill it. This test kills subsuming mutant M_3 and also mutant M_2. Finally, mutant M_4 is randomly selected for which the tester designs a test to kill it, by invoking `max (2,0,2)`. Hence, all subsuming mutants are killed.

In this particular scenario we can observe that MS* remains at 0% after analyzing the first 2 mutants randomly selected, and reaches a MS* of 33.33% after analyzing the

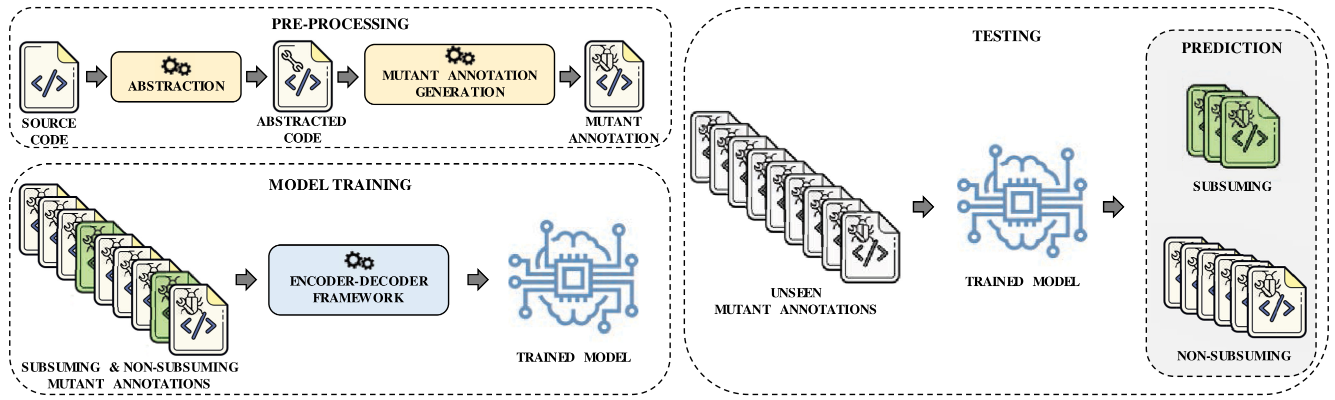


Fig. 3. Implementation: Source code is abstracted and attached with mutation annotation to produce mutant annotations. Model is trained on mutant annotations to further append the label (subsuming/non-subsuming). Trained model is provided with an unseen mutant annotation to append the label. The appended label acts as the prediction for the unseen mutant annotation.

third randomly selected mutant. The analysis of the fourth selected mutant (non-subsuming) did not add value (MS* remains the same). Finally, fifth and sixth analyzed mutants were subsuming, leading to a test suite that obtains MS* of 100% after analyzing 6 mutants.

Fig. 2c depicts the progress of MS* obtained by the test suites when guided by *Cerebro* and random mutant selection in the previously described scenarios. Through this example we demonstrate a case where two approaches analyze the same number of mutants (same effort) with *Cerebro* having higher effectiveness (MS*) than the random mutant selection baseline. At the same time, in order to reach the same MS* as *Cerebro*, random mutant selection needs more effort, i.e., it will require the analysis of many more mutants than *Cerebro* (in the example random baseline analyzed two times more mutants than *Cerebro*).

There are several points we want to highlight about the particular scenarios just described. First, it is essential to notice that mutants selected by *Cerebro* will be as close as possible to subsuming in the subsumption relation. Killing these (almost subsuming) mutants can help in killing subsuming mutants predicted as non-subsuming by *Cerebro*, for instance, the test that kills subsumed mutant M_{10}, also kills subsuming mutant M_7 that was incorrectly predicted as non-subsuming by *Cerebro*. Second, it is also important to notice that *Cerebro* selects the least possible number of equivalent mutants, saving the time of analysis to the tester (in the example, *Cerebro* did not predict any equivalent mutant as subsuming). Third, notice that the prediction performance obtained by *Cerebro* does not necessarily reflect its effectiveness in practice, since mutant kills are not independent of one another. While *Cerebro* reached 66.66% of Precision and 66.66% of Recall in the example, in practice, the test suite designed to kill all selected mutants obtains 100% of subsuming mutation score (MS*). And fourth, it is worth to study the trade-off between the effectiveness and effort of the different mutant selection techniques. We consider all these points in our empirical evaluation to assess the prediction performance, effectiveness, and effort required by *Cerebro* and the related mutant selection techniques.

4 Approach

The main objective of *Cerebro* is to automatically learn the silent features/patterns of the context surrounding subsuming mutants without requiring any features definition and/or selection by human intervention, that we can use later to predict if mutants on an unseen source code are likely to be subsuming or not. Thus, we train a machine translator (viz. an encoder-decoder model) to identify subsuming mutants, by feeding it with source code where the statement (to mutate) is annotated with the mutant type and its label (subsuming or not). Machine translators have been successfully used to translate text from one language to another, as they automatically recognize (i) the features of the language (to be translated) and (ii) the required translation (to the desired language). In our case, it is used to automatically identify the features of subsuming mutants without any investment of time and/or resources to define features.

After training, one can input to the translator, an unseen mutant (source code where the statement to mutate is annotated with the mutation annotation). The translator will append the label to the mutant given as input, to predict whether it is subsuming or not.

Fig. 3 shows an overview of the implementation. For training, *Cerebro* takes a set of mutants and their corresponding label. In each mutant source code, the statement (to mutate) is annotated with the mutation annotation, and the model learns the label to be appended to this annotation, that indicates whether the mutant is subsuming or non-subsuming. We can summarize *Cerebro*'s pre-processing, training and testing steps as follows:

1) *Abstraction:* Producing abstracted code of the actual source code by removing irrelevant information (e.g., comments) and replacing user-defined identifiers and literals (e.g., variable names) by predictable tokens;
2) *Pairs Generation:* Generating the pairs (input-expected output) to be used for training, by adding the corresponding label into the mutation annotations;
3) *Training:* Training the machine translator to learn which label is to be appended to the mutation annotations;
4) *Testing:* Utilizing the trained translator to predict and append labels to the mutation annotations present in unseen mutant source code.

In the remainder of this section we describe each of the aforementioned phases of our approach, in detail.

4.1 Abstracting the Irrelevant Information

A major challenge in dealing with raw source code is the huge vocabulary created by the abundance of identifiers

```
1   ...
2   public String getOptionValue (
3    final Option option ) {
4    if ( option == null ) {
5     return null ;
6    }
7    final String [] values =
8     getOptionValues ( option ) ;
9    return ( values == null ) ?
10    null : values [ 0 ] ;
11  }
12  ...
```

(a) Actual Source Code

```
1   ...
2   public String fn_3 (
3    final tp_1 vr_3 ) {
4    if ( vr_3 == null ) {
5     return null ;
6    }
7    final String [] vr_5 =
8     fn_4 ( vr_3 ) ;
9    return ( vr_5 == null ) ?
10    null : vr_5 [ 0 ] ;
11  }
12  ...
```

(b) Abstracted Code

```
1   ...
2   public String fn_3 (
3    final tp_1 vr_3 ) {
4    if ( vr_3 == null ) {
5     return null ; MST[ReturnValsMutator]MSP[ ]
6    }
7    final String [] vr_5 =
8     fn_4 ( vr_3 ) ;
9    return ( vr_5 == null ) ?
10    null : vr_5 [ 0 ] ;
11  }
12  ...
```


(c) Mutant Annotation

Fig. 4. Abstraction: Actual Source Code (4a) is abstracted by replacing user-defined entities (Function names, Type names, Variable names) with tokens (fn_num, tp_num, vr_num) to achieve the Abstracted Code (4b). Mutant annotation (4c) is generated by adding the Mutation annotation with its corresponding label, i.e., Subsuming (S) or Non-Subsuming (N). The trained model is used for prediction of unseen mutant annotations.

and literals used in the code. On such a large scale, vocabulary may hinder the goal of learning features surrounding the subsuming mutants. Thus, to reduce vocabulary size, we abstract source code by replacing user-defined entities with re-usable identifiers.

Fig. 4 shows an actual code snippet (Fig. 4a) converted into its abstract representation (Fig. 4b). The purpose of this abstraction is to replace any reference to user-defined entities (function names, types, goto labels, variable names and string literals) by identifiers that can be reused across source code file, hence reducing the vocabulary size. Thus, our abstraction approach first detects user-defined entities before replacing them with unique identifiers (new IDs).

New IDs follow the regular expression (fn|tp|lb|vr|lr)_(num)$^+$, where num stands for numbers $1, 2, 3, \ldots$ assigned in a sequential and positional fashion based on the occurrence of that entity. All the user-defined *Function* names, *Type* names, *Variable* names, *Labels*, and *String Literals* are replaced with fn_num, tp_num, lb_num, vr_num, and lr_num, respectively. Thus, the first function name found receives the ID fn_1, the second receives the ID fn_2, and so on. If any of these entities appear multiple times in a source code file, it is replaced with the same ID.

Additionally, we remove code comments and add mutation annotations to encode the mutation operator and the corresponding label (to be learned by the translator). Our mutation annotations have the general shape "MST[+MutationOperator+]MSP[]", where MST and MSP denote mutation annotation start and stop, respectively, and MutationOperator indicates the applied mutation operation (in green in Fig. 4c). Between the last brackets [], our trained model adds one of the labels S or N, indicating that the mutant obtained by applying the mutation operation, is predicted as subsuming or non-subsuming, respectively.

4.2 Pairs Generation

The mutation operation (ReturnValsMutator[5]) shown in Fig. 4c represents a mutant in which the sentence return null is replaced by throw new java.lang.RuntimeException() exception. Notice that this mutant is labeled as subsuming in our dataset, since there is only one test that can kill it, when the input option is null. Hence for training we consider S as the label to be learned by the translator to predict this mutant as subsuming.

To do so, we train in pairs (MutantAnnotation, MutantAnnotation+Label), where the first component is the annotated code shown in Fig. 4c, and the second component is the same code with the predicted label, i.e., MST [ReturnValsMutator]MSP[S] in our case, to indicate that the mutant is subsuming. The resulting text is arranged in a single sentence to represent a sequence of space-separated entities (the representation supported by the machine translator). The only difference between the input sequence given to the translator and the expected output sequence produced by it, is the predicted label S or N. Using these sequences, we intend to capture as much code as possible around the mutant without incurring the exponential increase in training time.

5. https://pitest.org/quickstart/mutators/#RETURN_VALS

4.3 Building the Machine Translator

To build our machine translator, we train an encoder-decoder model that can transform an input sequence to a desired output sequence. In our representation, a sequence consists of tokens separated by spaces that ends with a newline character. Thus, we train the encoder-decoder by feeding it with pairs of sequences, produced in the previous step. The translator learns to replicate the abstracted source code with the mutation annotation and to append the label (*S*/*N*) that will be used as a prediction for the mutant.

We found that training the translator on sequences of maximum *100* tokens in length is computationally feasible, but expensive (740 training hours required on a Tesla V100 GPU). Hence, we also experiment with sequences of *50* tokens in length and demonstrate that the computation cost of training the translator can be further contained (360 training hours required). We name *Cerebro* trained on sequences of 100 tokens in length as Cerebro-*100*. Following our naming convention, we name *Cerebro* trained on sequences of 50 tokens in length as Cerebro-*50*.

4.4 Predicting From Appended Labels

To predict whether or not a certain mutation at a particular position in an unseen code is subsuming, we abstract the unseen code followed by sequence generation which results in abstracted code sequence attached with mutation annotation as depicted in Figure 3. We feed this sequence into the trained machine translator to yield an output sequence with an appended label. The appended label acts as a prediction (subsuming/non-subsuming) for this specific mutation. If the translator produces an output sequence with a change other than appending the predicted label, the input sequence is predicted as non-subsuming, by default. In our experiments reported in

Section 7, this happened in 4.2% and 0.1% of the sequences for C and Java programs, respectively.

5 Research Questions

We start by checking the prediction ability of *Cerebro* and ask:

RQ1 *Prediction Performance:* How effective is *Cerebro* in predicting subsuming mutants?

We leverage two datasets, made of C and Java programs, for which extensive mutation analysis has been performed to identify subsuming mutants. We reimplemented 2 techniques that we use as baselines in our analysis. The first baseline is a *Random* mutant sampling, while the second is a supervised machine learning method based on manually designed features that were used by previous work [11] (e.g., data flow, control flow, etc.). These features are used to train a binary classifier in order to predict whether a mutant is subsuming or not. Further details about the baselines can be found in Section 6.3.

After analyzing the predictions, we turn our attention to the envisioned application scenario; measuring test effectiveness of the predicted mutants. It is important to check the application case because a) predictions may select weak mutants [11] (weak subsuming mutants result in lower test effectiveness than the strong ones), b) selected mutants may not be diverse as they may include mutually subsuming mutants [33], and c) tester benefits are unclear. Thus, we ask:

RQ2 *Effectiveness Evaluation:* How does *Cerebro* compare with the baselines in terms of subsuming mutation score?

We perform a simulation of a mutation testing scenario where a tester analyzes the selected mutants in order to generate tests [5], [11], [34]. For test effectiveness, we measure the subsuming mutation score (MS*) achieved by the tests that kill the selected mutants. In essence, we evaluate the guidance offered by the mutants when testers design tests to kill the selected mutants. It is worth noticing that in this part of the experiment we control the number of mutants, i.e., all techniques analyze the same number of mutants. Such simulation is typical in mutation testing literature [5], [11], [34] and aims at quantifying the benefit of an approach over the other.

Complementary to the previous question, we compare the effort required by each technique to obtain the same level of test effectiveness. Hence, we first investigate the human effort measured in terms of the number of mutants analyzed by the tester, to reach the same subsuming mutation score using *Cerebro* and the baselines. Hence, we ask:

RQ3 *Manual Effort:* How many mutants require manual analysis in order to reach a given level of subsuming mutation score?

We perform a similar simulation of a testing scenario in which we measure how many mutants the tester needs to analyze (generate a test case to kill or judge equivalence), until he/she obtains a determined subsuming mutation score. This allows us to quantify the human effort required by each approach to obtain the same benefit.

Related to the previous question, we also investigate the number of test executions necessary to reach the same subsuming mutation score, by following the incremental process of mutation analysis, i.e., a tester picking a mutant and analyzing it. If the picked mutant is killable, he/she generates a test case that kills it, and then checks if the remaining alive (not analyzed and not killed) mutants are collaterally killed by the same test (by executing the generated test on alive mutants). The killed mutants are removed from the set of alive mutants. Then, we ask:

RQ4 *Computational Effort:* How many test executions are required in order to reach a given level of subsuming mutation score?

We perform a simulation as before, but in this case, every time that a test is generated, we count the number of test executions and measure the attained subsuming mutation score, until we reach a given subsuming mutation score.

6 Experimental Setup

6.1 Benchmarks and Ground Truth

In order to show that our approach is language agnostic, we make our evaluation on a set of C and Java programs.

C-Benchmark: To perform our study that requires strong test suites, we used an independently built dataset from related work [12]. It includes C programs from the GNU Coreutils,[6] that consist of file, text and shell utility programs widely used in Unix systems. The data-set is composed of 48 GNU Coreutils (v8.22) programs *aka* subjects (mentioned in Table 1), each packaged with an accompanying system test suite, generated by developers. The size of these programs ranges from 1,000 to 14,000 lines of code (LOC), with a median size of 3,500 LOC. For each subject, the data-set includes a mutant-test killing matrix that records, for each mutant, a set of tests that kill it.

The mutant-test killing matrices were obtained by generating mutants using the *Mart* mutant generation tool [13] and executing them against large test pools. The test pools were built by considering developer tests and adding automatically generated tests using a 24 hours run of KLEE [10]. Additionally, mutation-based test suites were automatically generated using 128 different configurations of *SEMu* [12], each running for 2 hours, and an additional 'seeded' test generation of KLEE. To reduce the total execution cost, for each program, the 3 functions that were covered by the largest number of developer tests were selected for mutation analysis, i.e., mutants were generated only for these functions.

We use these mutant-test killing matrices to compute the mutant subsumption, following the definition given in Section 2.1, and label each mutant as either subsuming or non-subsuming. To make the problem as balanced as possible (to assist in machine learning), we mark as subsuming all mutants in the top of the hierarchies, including mutually subsumed mutants.

Needless to say, it is possible to have some noise in our labeling process in the sense that mutants labeled as subsuming may be non-subsuming. The data-set reduced this noise by augmenting the test suites with multiple large and diverse

6. https://www.gnu.org/software/coreutils/

TABLE 1
Benchmark

Project	Web URL	Version/ Commit
C		
base64, basename, chcon, chgrp, chmod, chown, chroot, cksum, comm, date, df, dirname, echo, expr, factor, false, groups, join, link, logname, ls, md5sum, mkdir, mkfifo, mknod, mktemp, nproc, numfmt, pathchk, printf, pwd, realpath, rmdir, sha256sum, sha512sum, sleep, stdbuf, sum, sync, tee, touch, truncate, tty, uname, uptime, users, wc, whoami [12]	https://github.com/coreutils/coreutils.git	v8.22
Java		
commons-cli	https://github.com/apache/commons-cli.git	6490067
commons-collections	https://github.com/apache/commons-collections.git	d6eeceb
commons-text	https://github.com/apache/commons-text.git	26a308f
commons-csv	https://github.com/apache/commons-csv.git	865872e
commons-lang	https://github.com/apache/commons-lang.git	2c0429a
commons-io	https://github.com/apache/commons-io.git	c126bdd
commons-net	https://github.com/apache/commons-net.git	33df028
commons-codec	https://github.com/apache/commons-codec.git	475910a
jsoup	https://github.com/jhy/jsoup.git	528ba55
joda-time	https://github.com/JodaOrg/joda-time.git	767c94e

test suites generated by different state-of-the-art tools. Please refer to the threat in Section 9 for a related discussion.

Java-Benchmark: For Java we select a set of well-tested open source projects from GitHub. We select projects from the

TABLE 2
Test Subjects

Language	#Programs	#Mutants	#Killed	#Subsuming	#Testcases
C [12]	48	71,850	49,530 (68.9%)	7,358 (10.2%)	136,412
Java	10	153,823	124,064 (80.6%)	41,219 (26.8%)	21,878

Apache Commons Proper[7] repository of reusable Java components, Joda-Time[8] - a date and time library, and Jsoup[9] - an HTML manipulation library. The set counts 10 projects: `commons-cli`, `commons-codec`, `commons-collections`, `commons-csv`, `commons-io`, `commons-lang`, `commons-net`, `commons-text`, `jsoup`, `joda-time`. These projects contain up to 284 classes. Table 1 reports the version/commit of each project we used for our study. Following a similar procedure done for C in [12], we also build test pools by using developer tests and adding automatically generated tests by running EvoSuite[23] for each project with the default running time, but with multiple coverage metrics[10]. The mutant-test killing matrices were obtained using Pitest [17]. For each project, we run the mutants on the test pools for 48 hours. To reduce execution time, we select the classes processed during that time lapse.

Table 2 records the total number of mutants, number (and percentage) of killable and subsuming mutants, and number of test cases conforming to the mutant-test killing matrices. Please note that the difference on the ratio of subsuming mutants with previous research [3], [33], [46] is due to the inclusion of all mutually subsuming mutants. As already explained, we include all subsuming mutants to avoid misleading our learner.

6.2 Equivalent Mutants

Early research on mutation testing has demonstrated that deciding whether a mutant is equivalent is an undecidable problem [9]. Mutation testing may produce a mutant that is syntactically different from the original, yet semantically identical, *aka* equivalent mutant [31]. Undecidability of equivalences means that it is impossible to automatically discard them all. As a result, the tester may never know whether he or she has failed to find a killing test case because the mutant is particularly hard to kill, yet remains killable (a 'stubborn' mutant [65]), or whether failure to find a killing test case derives from the fact that the mutant is equivalent. The best options we have are effective algorithms that can remove most equivalent mutants, e.g., in C data-set [12] authors applied TCE (Trivial Compiler Equivalence) [25], [31] to filter out equivalent and duplicated mutants. Interestingly, early research on mutation testing [2] has shown that humans also make many mistakes (approximately 20%) when judging mutants (as being equivalent or not). This means that it is unrealistic to expect that automated tools (or testers, in case of manual test case design) kill all killable mutants.

To make a fair approximation of killable mutants we used state-of-the-art test generation tools (KLEE[10], SEMu [12],

7. https://commons.apache.org
8. https://github.com/JodaOrg/joda-time/
9. https://github.com/jhy/jsoup
10. LINE:BRANCH:MUTATION:OUTPUT:METHOD:CBRANCH

and EvoSuite [23]), together with mature developer test suites to identify killable mutants. For the remaining live mutants (i.e., mutants that are killed neither by developers written nor automatically generated test suites) we assumed that live mutants are equivalent. Although, this assumption may have some impact on our results (refer to Section 8.4 for an analysis of the impact of this assumption), it allows quantifying the effort involved by testers in analyzing low utility mutants when using the current state-of-the-art advances. Moreover, since *Cerebro* performs machine learning, it learns from the employed data. This means that the availability of clean data, with a clear signal to learn, will allow *Cerebro* make better predictions, thereby potentially improving its performance.

6.3 Baselines

We consider 2 baselines. The first one is the *Random* mutant sampling that samples uniformly from the entire set of mutants. The second baseline is a Decision Tree classification based on the features proposed by related work [11], [29].

Previous works showed a strong connection between mutant utility and surrounding code (utility captured through CFG, data flows, AST, etc. features). Thus, we use the mutant features to predict subsuming mutants in both C and Java. Features belong to 4 categories: Mutant Type related features, Control-Flow graph related features, Control and Data dependency related features, and AST related features. In total we used the 28 features, used by the related work [11], for the C programs, and implemented 16 of those features for Java[11]. We excluded features such as AstChildHasIdentifier and AstChildHasLiteral that we found unfeasible to implement in the employed tools, i.e., Pitest works at byte-code level making it difficult to identify the original source code expression. Nevertheless, the excluded features were approximated by mutant type.

After extracting the features, following the related work [11], we trained a stochastic gradient boosted Decision Tree model by using the same configuration as the related work [11]. We followed the same validation setup for *Cerebro*.

6.4 Implementation and Model Configuration

We rely on the *srcML* tool [18] to convert source code into an XML format to tag literals, keywords, identifiers, comments, and our mutation annotations. This helps in separating user-defined identifiers and string literals (the largest part of the vocabulary) from language keywords as srcML supports C, Java and other languages. Then, we implement the ID replacement to generate the abstracted code.

We follow the sequence pair generation procedure mentioned in Section 4.2 to generate sequences from the abstracted code. These sequences serve as training input for our encoder-decoder model, which we build using *tf-seq2seq* [1], a general-purpose encoder-decoder framework. Following previous works [60], [61], we configure our model with bidirectional encoder. We use a Gated Recurrent Units (GRU) network [16] to act as the Recurrent Neural Network (RNN) cell, which was shown to perform better than possible alternatives (simple RNNs or gated recurrent units) in related prediction tasks [56]. To achieve good performance with acceptable model training time, we utilize AttentionLayerBahdanau [7] as our attention class, configured with 2 layered AttentionDecoder and 1 layered BidirectionalRNNEncoder, both with 256 units.

To determine an appropriate number of training epochs, we conducted a preliminary study involving a validation set, independent of both, training and test sets that we use in our evaluation. Here we incrementally train the model, with checks after every epoch to monitor model training accuracy. We pursue training the model till the training performance on the validation set does not improve anymore. We found 15 epochs to be a good default for our validation sets. Once model training is complete, we follow the procedure explained in Section 4.4 to predict whether an unseen mutant annotation sequence is subsuming or not.

The codebase of C and Java programs with mutant information, abstracted code, and mutant annotation sequences that the encoder-decoder model trains on and predict, with mapping to the original code, are publicly available at https://github.com/garghub/Cerebro. In addition to our dataset, we have made available our source code and trained models as well.

6.5 Experimental Procedure

In the first experimental part, we evaluate the prediction ability of our approach, answering RQ1, while in the second part, we evaluate cost-effectiveness of *Cerebro*, answering RQs2-4.

6.5.1 First Experimental Part

We start by evaluating the prediction performance of *Cerebro*, and the baselines, using four typical metrics, namely, *Precision*, *Recall*, *F-measure*, and *Matthews Correlation Coefficient* (MCC) [40]. A confusion matrix is computed for each one of the studied methods, which stores the correct and incorrect predictions. Given a subsuming mutant, if it is predicted as subsuming, then it is a true positive (TP); otherwise, it is a false negative (FN). Given a non-subsuming mutant, if it is predicted as non-subsuming, then it is a true negative (TN); otherwise, it is a false positive (FP). Then we can use the confusion matrix to quantitatively evaluate the prediction performance of *Cerebro* and *Decision Trees* prediction models

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$

$$F-measure = \frac{2\times Precision \times Recall}{Precision + Recall}$$

$$MCC = \frac{TP\times TN - FP\times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}.$$

Intuitively, *Precision* is the ratio of mutants truly subsuming among all the mutants predicted as subsuming. *Recall* is the ratio of mutants correctly predicted as subsuming among all the subsuming mutants. *F-measure* indicates the weighted harmonic mean of *Precision* and *Recall*. *Matthews Correlation*

11. statementComplexity, expressionComplexity, MutantType, BlockDepth, CfgDepth, CfgPredNum, CfgSuccNum, NumInBlock, NumOutDataDeps, NumInDataDeps, NumOutCtrlDeps, NumInCtrlDeps, AstNodeParentType, NumberOfAstParents, AstNodeType, NumberOfAstChildren

Coefficient (MCC) [40] is a reliable metric of the quality of prediction models [55], that in contrast to the previous metrics, also takes into account the True Negatives (correctly predicted non-subsuming mutants). It is generally regarded as a balanced measure that can be used even when the dataset is unbalanced, i.e., the classes are of very different sizes, e.g., in case of C programs, 10.2% subsuming mutants (Positives) over 89.8% non-subsuming mutants (Negatives). MCC returns a 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 equals 0 indicates that the prediction performance is equivalent to random guessing.

The mutants selected by *Cerebro* are the ones predicted as subsuming. For *Decision Trees* baseline, as it computes a probability of a mutant being subsuming, we followed the probability margin convention and considered those mutants whose predicted probability was higher than 0.5 [11].

To assess the performance we perform a inter-project evaluations. We use 5-folds cross validation, where we evenly split each benchmark in 5 parts (10 programs and 2 projects per fold for C and Java benchmark, respectively). Then, for each benchmark, we repetitively use 1 fold for testing and 4 folds for training (1 part out of 4, is used for validation).

6.5.2 Second Experimental Part

To study the cost and test effectiveness of our approach and the baselines, we simulate a testing scenario where a tester selects a subset of mutants, to use for mutation analysis, and designs tests to kill them. Algorithm 1 provides the pseudocode of the simulation process we follow in our experiments. It takes as input a set `M` of mutants to analyze, the test pool `P` and a target subsuming mutation score `tMS*`, and returns a test suite `T` that kills every mutant from `M` (or reaches the prespecified subsuming mutation score). Additionally, it returns the subsuming mutation score obtained by the test suite `T` (`currMS*`), number of analyzed mutants (`analyzedMut`), number of equivalent mutants analyzed (`equivMut`), and number of test executions (`tExec`) required to generate test suite `T` during the simulated mutation testing scenario.

The simulation starts by picking (`pickNextMutant`) the top mutant `m`, according to the technique used (*Cerebro*, *Decision Trees*, and *Random*), among survived mutants from set `C` (initialized with all mutants from `M`). It then checks if there exists some test in the test pool `P` that kill `m` (this process simulates a tester picking, analyzing, and designing a test to kill a mutant). If no test kills mutant `m`, we judge it as equivalent and remove it from `C`. Otherwise, we randomly pick one test `t` from the pool that kills `m`. Then, we run the test `t` on every mutant from `C` to check if the same test consequently kills other mutants (killed mutants are then removed from `C`). This process continues by taking the next survived mutant and finding a test to kill it until every mutant in `C` has been killed or until the desired subsuming mutation score is reached. We run this simulation with the set of mutants selected by *Cerebro*, *Decision Trees*, and *Random*, respectively, and use the reported values to compare their cost-benefit performance for answering RQ2-4. Since Algorithm 1 includes some random decisions, we repeat this process 1,000 times for all the approaches.

Algorithm 1. Pseudo-Code of the Simulation Procedure to Answer RQ2-4

Input: set of mutants `M`
Input: test pool `P`
Input: target subsuming mutation score `tMS*`
Output: test suite `T` covering mutants in `M`
Output: subsuming mutation score `currMS*` obtained by `T`
Output: `analyzedMut` number of analyzed mutants
Output: `equivMut` number of equivalent mutants analyzed
Output: `tExec` number of test executions

```
1:  T ← ∅
2:  C ← M                                   ▷set of survived mutants
3:  currMS* ← 0
4:  while currMS* < tMS* and ¬isEmpty(C) do
5:    m ← pickNextMutant(C)
6:    analyzedMut ++
7:    if the test pool P can kill mutant m then
8:      t ← randomlyPickTestKilling(m, P)
9:      T ← T ∪ {t}                         ▷add test t to the suite
10:     tExec += size(C)                    ▷run t on mutants from C
11:     remove from set C all mutants killed by t
12:   else
13:     equivMut ++                         ▷m is judged as equivalent
14:   end if
15:   currMS* ← calculateMS*(M, T)
16: end while
17: return T, currMS*, analyzedMut, equivMut, tExec
```

To answer RQ2, we measure the effectiveness (benefit) of the approaches in terms of the *subsuming mutation score* (MS*), i.e., the ratio between killed and total number of subsuming mutants, achieved by the generated test suites when analyzing the selected mutants. The subsuming mutation score reduces the influence of redundant mutants [33], [46].

For assessing the effectiveness of the approaches, we aim at controlling the number of mutants selected by each tool. In the case of *Cerebro*, the mutants selected are the ones predicted as subsuming by our model. For *Decision Trees* baseline, we rank (in descending order) the mutants according to the predicted probability of being subsuming, and follow the ranking to pick mutants (from highest probability to lowest) for analysis. *Random* baseline randomly ranks the mutants to be selected. Initially, we consider the same number of selected mutants for the 3 approaches, defined as the number of mutants predicted as subsuming by *Cerebro*. For instance, if *Cerebro* predicts 20 mutants as subsuming, then *Decision Trees* and *Random* baselines will also select the top 20 ranked mutants. Our intention is to compare the effectiveness reached by each approach, when the number of selected mutants is equal.

Additionally, we study the number of *equivalent mutants* selected by each approach (as these are an important source of redundancy during mutation testing), as well as, the required number of mutants selected by the baselines in order to reach the same subsuming mutation score as *Cerebro*.

To answer RQ3 and RQ4, we study the effort (cost) required by each approach in two ways. We measure the human effort in terms of the number of *analyzed mutants*,

TABLE 3
(RQ1) Prediction Performance of *Cerebro* and *Decision Trees*

Average (and Median) Performance in C-Benchmark				
Approach	MCC	F-measure	Precision	Recall
Decision Trees	0.17 (0.18)	0.25 (0.26)	0.25 (0.25)	0.25 (0.27)
Cerebro-50	0.39 (0.40)	0.34 (0.34)	0.82 (0.82)	0.21 (0.22)
Cerebro-100	0.47 (0.47)	0.41 (0.40)	0.93 (0.93)	0.26 (0.25)
Average (and Median) Performance in Java-Benchmark				
Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Decision Trees	0.16 (0.18)	0.28 (0.30)	0.45 (0.48)	0.21 (0.21)
Cerebro-50	0.38 (0.38)	0.42 (0.42)	0.72 (0.73)	0.31 (0.30)
Cerebro-100	0.45 (0.45)	0.51 (0.52)	0.76 (0.73)	0.39 (0.38)

On Average, Cerebro *Outperforms by 2.78 Times Higher MCC Than* Decision Trees.

killable or not, that are presented to testers for analysis (i.e., either designing a test to kill these or judging these as equivalent), when applying mutation testing. Intuitively, for a given set of mutants, the number of analyzed mutants can be considerably smaller than the entire set's size because a test designed by analyzing one mutant can kill other mutants as well. Hence, we also measure the computational effort in terms of the number of *test executions* performed, during the mutation analysis procedure, i.e., we count the test executions required at every step where a new test is created. As for RQ2, here we also study the number of test executions and the number of mutants that require analysis by the baselines, to reach the same subsuming mutation score as *Cerebro*.

7 Experimental Results

7.1 Prediction Performance (RQ1)

Table 3 records the average (and median) performance metrics. Fig. 5 shows the performance comparison in box plot format showing the distribution of performance indicators (MCC, F-measure, Precision, and Recall) for both approaches in C, and Java Benchmarks.

On average, *Cerebro* obtains a high Precision, i.e., 0.93 and 0.76 (*Cerebro*-100), and 0.82 and 0.72 (*Cerebro*-50) in C and Java benchmarks, respectively. Testers focusing on mutants selected by *Cerebro* can be confident that these are very likely to be subsuming, providing high utility to the testing process. On the other hand, Recall achieved is low, i.e., 0.26 and 0.39 (*Cerebro*-100), and 0.21 and 0.31 (*Cerebro*-50) in C and Java benchmarks, respectively. This indicates that many subsuming mutants are mistakenly predicted as non-subsuming by *Cerebro*. In practice these mutants can still be collaterally killed by other (mutually subsumed) subsuming mutants correctly predicted as subsuming by *Cerebro* (which is often the case, as we will show when answering RQ2 in the following section). Needless to say, any complementary mutation testing and mutant selection technique can be employed to analyze the remaining mutants that are not killed by test suites designed to kill mutants selected by *Cerebro*.

On comparison with baselines, we observe that *Cerebro* clearly achieves much higher prediction performance in comparison to *Decision Trees* in both benchmarks. The differences are statistically significant.[12]

In C-Benchmark, on average, *Cerebro* with its MCC of 0.47 (*Cerebro*-100), and 0.39 (*Cerebro*-50) outperforms *Random* (0.0 MCC). *Cerebro* also outperforms *Decision Trees*, on average, with 2.76 times higher MCC and 64% improvement in F-measure. It is worth mentioning that while *Cerebro* achieves 3.72 times higher precision than *Decision Trees*, *Cerebro* also offers an improvement of 4% in Recall over *Decision Trees*.

In Java-Benchmark, on average, *Cerebro* with its MCC of 0.45 (*Cerebro*-100), and 0.38 (*Cerebro*-50) outperforms *Random* (0.0 MCC). *Cerebro* also outperforms *Decision Trees*, on average, with 2.81 times higher MCC, and an improvement of 82% in F-measure, 68.88% in Precision, and 85.71% in Recall.

In summary, *Cerebro* offers an improvement in prediction capability (MCC) of 2.78 times higher than *Decision Trees*.

7.2 Effectiveness Evaluation (RQ2)

Figs. 6a and 6d show the average subsuming mutation score (MS*) obtained when selecting the same number of mutants (by all techniques). In C-Benchmark, on average, *Cerebro*-100 obtains an MS* of 87.50%, which is 2.39 and 2.63 times higher MS* than *Decision Trees* and *Random*, respectively. Moreover, *Cerebro*-50 obtains an MS* of 71.43%, which is 2.02 and 2.17 times higher MS* than *Decision Trees* and *Random*, respectively.

In Java-Benchmark, on average, *Cerebro*-100 obtains an MS* of 95.90%, which is twice higher than *Decision Trees*, and 69.53% improvement over *Random*. Moreover, *Cerebro*-50 obtains an MS* of 95.66%, which is 2.20 times higher than *Decision Trees*, and 83.33% improvement over *Random*. The differences are statistically significant, according to the computed $p-value$. We also compared them with the *Vargha-Delaney A measure* ($\hat{A}_{12}$) [62], showing that *Cerebro* achieves better MS* than *Decision Trees*, and *Random*, in 92.4%, and 95.7% of the cases.

We also study the selection size needed by *Decision Trees* and *Random* to achieve the same MS* obtained by *Cerebro*. For C-Benchmark, Fig. 6b shows that while *Cerebro*-100 selects only 2.35% of the mutants, *Decision Trees*, and *Random* need to select 85.42% (36.35 times higher), and 87.61% (37.28 times) of the mutants to achieve same MS* as *Cerebro*. Also, Fig. 6e shows that while *Cerebro*-50 selects only 2.52% of the mutants, *Decision Trees*, and *Random* need to select 34.23% (13.57 times higher), and 42.37% (16.79 times) of the mutants, to achieve same MS* as *Cerebro*. For Java-Benchmark, while *Cerebro*-100 selects 9.85% of the mutants, *Decision Trees*, and *Random* need to select 44.80% (4.55 times higher), and 78.97% (8.02 times) of the mutants, to achieve same MS* as *Cerebro*-100. Also, while *Cerebro*-50 selects 11.60% of the mutants, *Decision Trees*, and *Random* need to select 41.77% (3.60 times higher), and 75.09% (6.48 times) of the mutants, to achieve same MS* as *Cerebro*-50. We obtained a statistically significant $p-value$ and $\hat{A}_{12}$ when

12. We compared the MCC values using *Wilcoxon signed-rank test* and obtained a $p-value < 5.07e{-3}$ in comparison to *Decision Trees*. We also compared the MCC values with the *Vargha-Delaney A measure* [62] and observed that in all (100%) cases, *Cerebro* significantly outperforms baseline techniques.

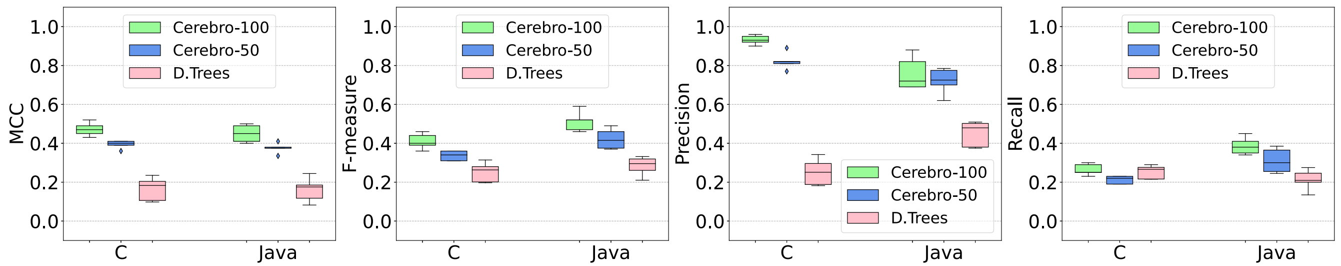
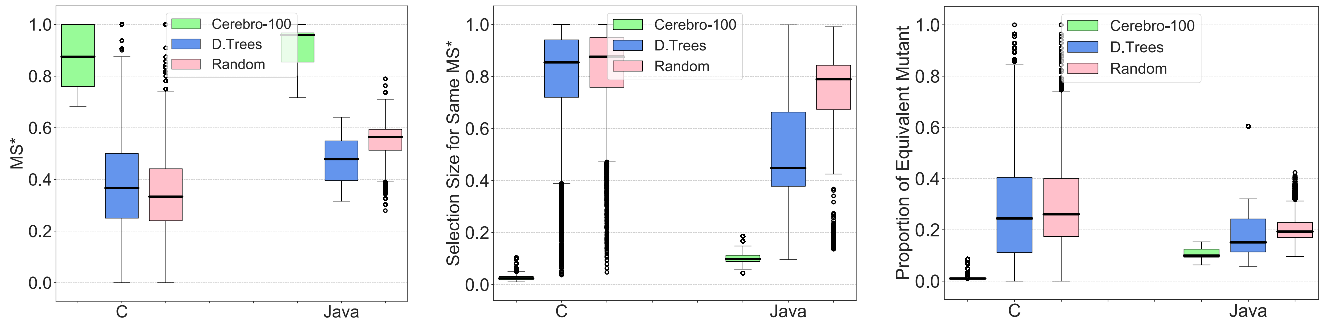


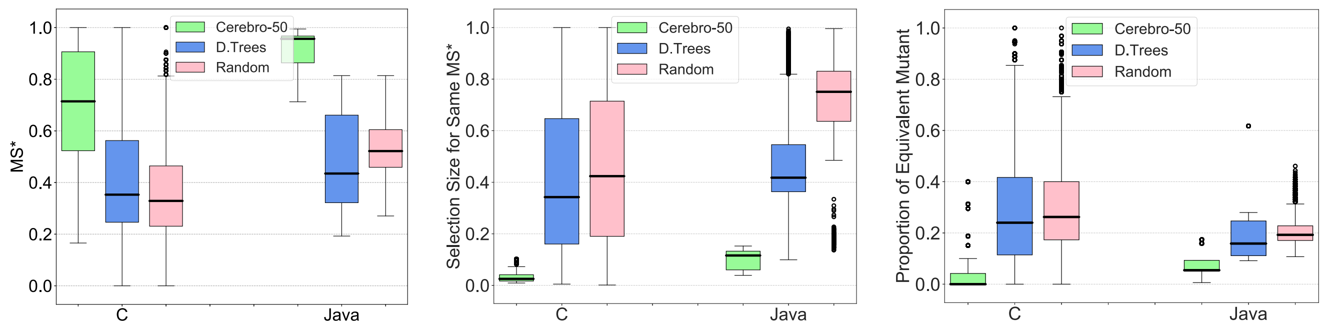
Fig. 5. (RQ1) Prediction Performance Comparison: On average, *Cerebro*-100 outperforms *Decision Trees* by 2.76 times, and 2.81 times higher MCC in C, and Java Benchmark. Moreover, *Cerebro*-50 outperforms *Decision Trees* by 2.29 times, and 2.38 times higher MCC in C, and Java Benchmark. Overall, *Cerebro* outperforms by 2.78 times higher MCC than *Decision Trees*.



(a) C-Benchmark: For the same mutant selection size, *Cerebro*-100 obtains an MS* of 87.50%, while *Decision Trees*, and *Random* obtains 36.67%, and 33.33%.
Java-Benchmark: *Cerebro* obtains, on average, an MS* of 95.90%, while *Decision Trees*, and *Random* obtains 47.85%, and 56.45%.

(b) C-Benchmark: to reach the same MS*, *Cerebro*-100 uses 2.35% of the mutants, while *Decision Trees*, and *Random* 85.42%, and 87.61%.
Java-Benchmark: *Cerebro*-100 uses 9.85% of the mutants, while *Decision Trees*, and *Random* use 44.80%, and 78.97%.

(c) C-Benchmark: *Cerebro*-100 selects 1.10% equivalent mutants, while *Decision Trees*, and *Random* select 24.44%, and 26.09%.
Java-Benchmark: 9.95% of mutants selected by *Cerebro*-100 are equivalent, whereas 15.11%, and 19.33% of mutants selected by *Decision Trees*, and *Random* are equivalent.



(d) C-Benchmark: For the same mutant selection size, *Cerebro*-50 obtains an MS* of 71.43%, while *Decision Trees*, and *Random* obtains 34.23%, and 32.88%.
Java-Benchmark: *Cerebro*-50 obtains, on average, an MS* of 95.65%, while *Decision Trees*, and *Random* obtains 43.48%, and 52.17%.

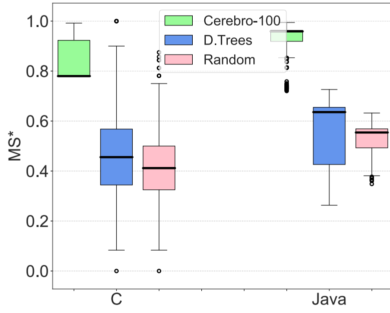
(e) C-Benchmark: to reach the same MS*, *Cerebro*-50 uses 2.52% of the mutants, while *Decision Trees*, and *Random* 34.24%, and 42.37%.
Java-Benchmark: *Cerebro*-50 uses 11.60% of the mutants, while *Decision Trees*, and *Random* use 41.77%, and 75.09%.

(f) C-Benchmark: *Cerebro*-50 selects 4.37% equivalent mutants, while *Decision Trees*, and *Random* select 24%, and 26.23%.
Java-Benchmark: 5.45% of mutants selected by *Cerebro*-50 are equivalent, whereas 15.86%, and 19.26% of mutants selected by *Decision Trees*, and *Random* are equivalent.

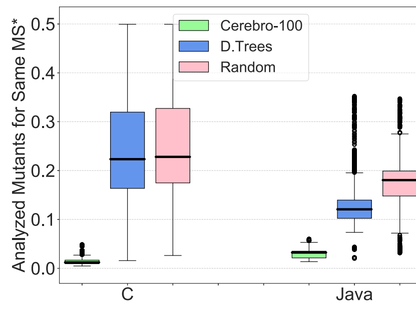
Fig. 6. (RQ2) Results of the Simulation - Trade off between mutant selection size and MS*.

compared these values, evidencing that *Cerebro* in more than 98.5%, and 99.1% of the cases, selects fewer mutants than *Decision Trees*, and *Random*.

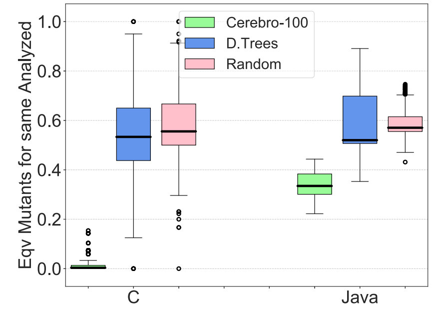
We also measure the percentage of equivalent mutants selected. For C-Benchmark, Fig. 6c shows that 1.10% of mutants selected by *Cerebro*-100 are equivalent, whereas 24.44%, and 26.09%, of the mutants selected by *Decision Trees*, and *Random*, are equivalent. Also, Fig. 6f shows that 4.37% of mutants selected by *Cerebro*-50 are equivalent, whereas 24%, and 26.23%, of the mutants selected by *Decision Trees*, and *Random*, are equivalent. In Java-Benchmark, 9.95% of the mutants selected by *Cerebro*-100 are equivalent whereas for *Decision Trees*, and *Random*, 15.11% (51.86% more), and 19.33% (94.27% more) selected mutants are equivalent. Also, 5.45% of the mutants selected by *Cerebro*-50 are equivalent whereas for *Decision Trees*, and *Random*, 15.86% (2.91 times higher), and 19.26% (3.53 times higher) selected mutants are equivalent. The differences are statistically significant. $\hat{A}_{12}$ shows that *Cerebro* in more than 90%, and 98.4% of the cases selects fewer equivalent mutants than *Decision Trees*, and *Random*. These



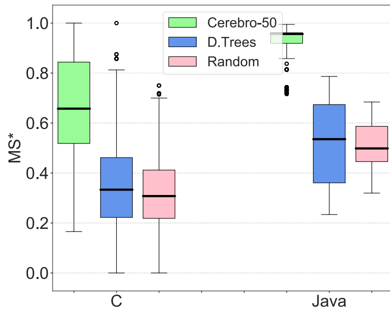
(a) C-Benchmark: Same number of mutants lead to MS* of 78%, 45.56%, and 41.18% for *Cerebro*-100, *Decision Trees*, and *Random*.
Java-Benchmark: *Cerebro*-100 reaches MS* of 94.90%, whereas *Decision Trees*, and *Random* reach 63.59%, and 55.43%.



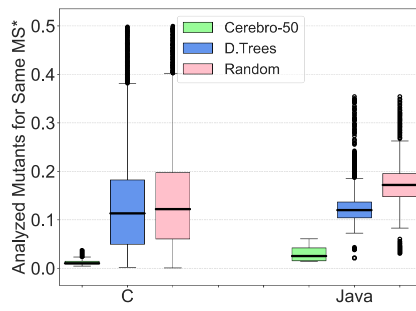
(b) C-Benchmark: *Cerebro*-100 analyzes 1.21% mutants, whereas *Decision Trees*, and *Random* analyze 22.33%, and 22.80% to reach the same MS* as *Cerebro*-100.
Java-Benchmark: *Cerebro*-100 analyze 3.22% mutants, whereas *Decision Trees*, and *Random* analyze 12.07% and 18.05% to reach same MS*.



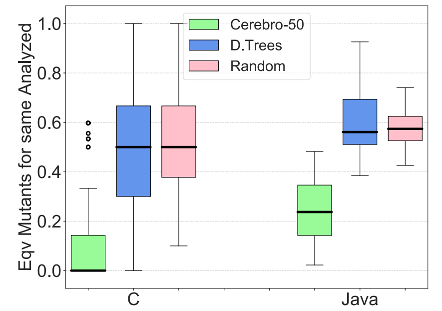
(c) C-Benchmark: 3.70%, 53.33%, and 55.56% of the mutants selected by *Cerebro*-100, *Decision Trees*, and *Random* are equivalent.
Java-Benchmark: 33.48%, 52%, and 57.04% of the mutants selected by *Cerebro*-100, *Decision Trees*, and *Random* are equivalent.



(d) C-Benchmark: Same number of mutants lead to MS* of 65.75%, 33.33%, and 30.77% for *Cerebro*-50, *Decision Trees*, and *Random*.
Java-Benchmark: *Cerebro*-50 reaches MS* of 95.65%, whereas *Decision Trees*, and *Random* reach 53.54%, and 49.83%.



(e) C-Benchmark: *Cerebro*-50 analyzes 1.02% mutants, whereas *Decision Trees*, and *Random* analyze 11.92%, and 13.17% to reach the same MS* as *Cerebro*-50.
Java-Benchmark: *Cerebro*-50 analyze 2.52% mutants, whereas *Decision Trees*, and *Random* analyze 12% and 17.19% to reach same MS*.



(f) C-Benchmark: 11.31%, 50%, and 50% of the mutants selected by *Cerebro*-50, *Decision Trees*, and *Random* are equivalent.
Java-Benchmark: 23.72%, 56.08%, and 57.38% of the mutants selected by *Cerebro*-50, *Decision Trees*, and *Random* are equivalent.

Fig. 7. (RQ3) Results of the Simulation - Trade off between percentage of mutants analyzed and MS*.

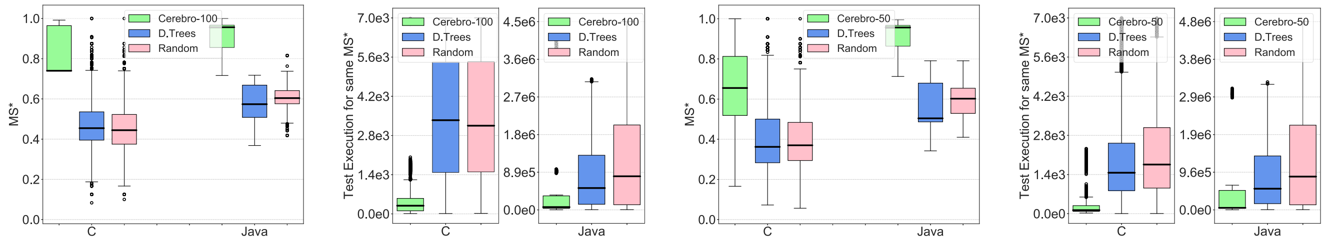
results provide evidence that our approach can reduce significantly this long-standing problem of mutation analysis.

7.3 Number of Analyzed Mutants (RQ3)

Figs. 7a and 7d show the average subsuming mutation score (MS*) obtained by each technique for the same number of analyzed mutants. In C-Benchmark, on average, *Cerebro*-100 achieved an MS* of 78%, which is an improvement of 89.41%, and 71.20% over the MS* of *Random*, and *Decision Trees*, respectively. Moreover, *Cerebro*-50 achieved an MS* of 65.75%, which is 2.14 times higher than *Random* and an improvement of 97% over *Decision Trees* In Java-Benchmark, on average, *Cerebro*-100 achieved an MS* of 94.90%, an improvement of 49.24% and 71.21% over *Decision Trees* and *Random*, respectively. Moreover, *Cerebro*-50 achieved an MS* of 95.65%, an improvement of 78.65% and 91.94% over *Decision Trees* and *Random*, respectively. The differences are statistically significant, according to the computed $p-value$ and $\hat{A}_{12}$. We observed that *Cerebro* in more than 96.2%, and 98.4%, of the cases is better than *Decision Trees*, and *Random*.

We also study what should be the percentage of mutants to be analyzed by *Decision Trees* and *Random* to achieve the same MS* as *Cerebro*. For C-Benchmark, Fig. 7b shows that while *Cerebro*-100 analyzes 1.21% mutants, *Decision Trees*, and *Random* need to analyze 22.33% (18.45 times higher), and 22.80% (18.84 times higher) of mutants to reach same MS* as *Cerebro*-100. Also, Fig. 7e shows that while *Cerebro*-50 analyzes 1.02% mutants, *Decision Trees*, and *Random* need to analyze 11.92% (11.58 times higher), and 13.17% (12.78 times higher) of mutants to reach same MS* as *Cerebro*-50. In Java-Benchmark, while *Cerebro*-100 analyzes 3.22% mutants, *Decision Trees*, and *Random* need to analyze 12.07% (3.75 times higher), and 18.05% (5.61 times higher) of mutants to reach same MS* as *Cerebro*-100. Moreover, while *Cerebro*-50 analyzes 2.52% mutants, *Decision Trees*, and *Random* need to analyze 12.00% (4.76 times higher), and 17.19% (6.82 times) of mutants to reach same MS* as *Cerebro*-50. We obtained a statistically significant $p-value$ and $\hat{A}_{12}$, showing that *Cerebro* in more than 99% of the cases analyzes less mutants than *Decision Trees* and *Random*.

We also measure the percentage of equivalent mutants analyzed by each technique. For C-Benchmark, Fig. 7c shows that, on average, *Cerebro*-100 analyzes 3.70% equivalent mutants, while 53.33% (14.41 times higher), and 55.56% (15.02



(a) C-Benchmark: For same number of test executions, *Cerebro*-100 obtains an MS* of 74%, while *Decision Trees*, and *Random* obtain 45.45%, and 44.44%.
Java-Benchmark: *Cerebro*-100 obtains MS* of 95.65%, while *Decision Trees*, and *Random* obtain 57.38%, and 60.40%.

(b) C-Benchmark: *Cerebro*-100 requires 291 test executions, while *Decision Trees*, and *Random* require 3,345, and 3,149 to reach the same MS* as *Cerebro*-100.
Java-Benchmark: 65,741 test executions are required by *Cerebro*-100, while *Decision Trees*, and *Random* require 517,040, and 795,304.

(c) C-Benchmark: For same number of test executions, *Cerebro*-50 obtains an MS* of 65.52%, while *Decision Trees*, and *Random* obtain 36.20%, and 36.99%.
Java-Benchmark: *Cerebro*-50 obtains MS* of 95.65%, while *Decision Trees*, and *Random* obtain 50.41%, and 60.21%.

(d) C-Benchmark: *Cerebro*-50 requires 125 test executions, while *Decision Trees*, and *Random* require 1,785, and 2,182 to reach the same MS* as *Cerebro*-50.
Java-Benchmark: 50,622 test executions are required by *Cerebro*-50, while *Decision Trees*, and *Random* require 560,866, and 894,494.

Fig. 8. (RQ4) Results of the Simulation - Trade off between number of test executions and MS*.

times higher) of the mutants analyzed by *Decision Trees*, and *Random* are equivalent. Also, Fig. 7f shows that *Cerebro*-50 analyzes 11.31% equivalent mutants, while 50% (4.42 times higher) of the mutants analyzed by *Decision Trees* and *Random* are equivalent. For Java-Benchmark, on average, 33.48% of the mutants analyzed by *Cerebro*-100 are equivalent, while *Decision Trees*, and *Random* analyze 52% (55.31% more), and 57.04% (70.37% more) equivalent mutants. Also, 23.72% of the mutants analyzed by *Cerebro*-50 are equivalent, while *Decision Trees*, and *Random* analyze 56.08% (2.36 times higher), and 57.38% (2.42 times higher) equivalent mutants. This indicates that the baselines suggest the consumption of a large effort to analyze redundant mutants, in comparison to *Cerebro*. The differences are statistically significant. $\hat{A}_{12}$ suggests that *Cerebro* in more than 98% of the cases analyzes fewer equivalent mutants than *Decision Trees*, and *Random*.

7.4 Number of Test Executions (RQ4)

Figs. 8a and 8c show the average subsuming mutation score (MS*) when the number of test executions are fixed. In C-Benchmark, on average, *Cerebro*-100 achieves an MS* of 74%, outperforming *Decision Trees*, and *Random* by 62.82%, and 66.52% (*Decision Trees*, and *Random* achieve 45.45%, and 44.44% of MS*). Also, *Cerebro*-50 achieves an MS* of 65.52%, outperforming *Decision Trees*, and *Random* by 80.95%, and 77.14% (*Decision Trees*, and *Random* achieve 36.21%, and 36.99% of MS*). In Java-Benchmark, on average, *Cerebro*-100 and *Cerebro*-50 achieve an MS* of 95.65% in both simulations, an improvement of approx. 67%, and 58% over *Decision Trees*, and *Random* (*Decision Trees*, and *Random* achieve 57.38%, and 60.40% of MS* in first simulation when compared against *Cerebro*-100, and 50.41%, and 60.21% of MS* in the second comparison simulation against *Cerebro*-50). We obtained a statistically significant $p-value$. Also $\hat{A}_{12}$ suggests that *Cerebro* in 94.15%, and 95.7%, of the cases is better than *Decision Trees*, and *Random*.

We also measure the test executions required by the baselines to achieve the same MS* as *Cerebro*. Fig. 8b shows that, in C-Benchmark, *Cerebro*-100 requires 291 test executions (median), while *Decision Trees*, and *Random* require 3,345, and 3,149. Also, Fig. 8d shows that *Cerebro*-50 requires 125 test executions (median), while *Decision Trees*, and *Random* require 1,785, and 2,182. This shows that *Cerebro*-100 is 10-12 times less and *Cerebro*-50 is 14-17 times less expensive (computationally) than the baselines.

In Java-Benchmark, *Decision Trees*, and *Random* require 517,040, and 795,304 test executions (median) to achieve the same MS* as *Cerebro*-100, for which 65,741 test executions are required. Moreover, *Decision Trees*, and *Random* require 560,866, and 894,494 test executions to achieve the same MS* as *Cerebro*-50, for which 50,622 test executions are required. This shows that the baselines require 7 to 12 times, and 11 to 17 times higher computational effort than *Cerebro*-100, and *Cerebro*-50.

These differences are statistically significant. $\hat{A}_{12}$ value indicates that in more than 98.7% of the cases, *Cerebro* executes fewer tests than *Decision Trees* and *Random*.

8 Discussion

Cerebro is a learning-based method, and thus its performance depends on a number of parameters and design decisions we made. To this end, we discuss the key (intuitive) parameters that make the Machine Translation approach we use effective (Section 8.1), together with empirical results demonstrating the potential impact on the model's performance given the design decisions of using unabstracted code sequences (Section 8.2), sequences with decreased length during training (Section 8.3), and the impact of assuming unkilled mutants as equivalent mutants during testing (Section 8.4).

8.1 Why *Cerebro* is a Good Candidate for Subsuming Mutant Prediction?

There are three main factors that make Machine Translation a good candidate for subsuming mutant prediction. The first one is that it learns to select mutants using the exact local context (entire code snippet composed of 50-100 tokens, represented as a sequence), while previous work considers AST and data-flow abstractions [11], ignoring the exact formulation of the code snippet. In a sense, the key determining factor is the sequence that code tokens appear in the local context (considered code snippet). The second reason is that the

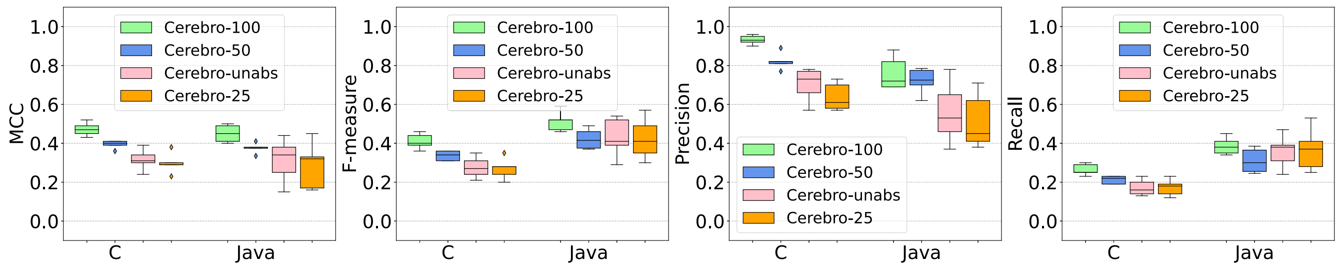


Fig. 9. Impact of the abstraction process and sequence length in *Cerebro*'s prediction performance: On average, MCC is decreased by 18% with unabstracted code and decreased by 24% with sequence length 25.

machine translator includes a powerful self-attention mechanism, which together with the encoder-decoder architecture makes the learning resistant to noise [59], and able to learn out of imbalanced data. Overall, previous research has shown that this architecture often makes the best predictions for many NLP tasks [20]. This is actually the reason why Machine Translation has been successfully used in code analysis tasks such as mutant generation, code clone detection, test assertions generation, etc. The third reason is the diversity of the selected mutants, i.e., Cerebro selects a few mutants per code block, which allows eliminating local redundancies, while spreading testing across the entire code-base.

8.2 Impact of Removing Code Abstraction

We analyzed the impact of using unabstracted code sequences to train our models instead of proposed abstracted code sequences and how it affects the model prediction performance (RQ1). In this experiment, we just removed the code comments and kept everything else as it is. We found a prediction performance reduction for projects in both C and Java benchmarks. For C-Benchmark, the model performance deteriorated by 18.9% in MCC, 14.4% in Precision and 18.1% in Recall. For Java-Benchmark, although we found an improvement of 15.4% in Recall, the overall performance deteriorated by 17.9% in MCC and 22.5% in Precision.

8.3 Impact of Reducing the Sequence Length

We also analyzed the impact of reducing the length of sequences that we use to train our models and how it affects the model prediction performance (RQ1). In this experiment, we reduced the sequence length from 50 tokens per sequence to 25 tokens per sequence. Fig. 9 and Table 4 shows the average and median scores achieved by the models. For simulation details on Effectiveness Evaluation (RQ2), Number of Analyzed Mutants (RQ3) and Number of Test Executions (RQ4), please refer to our online repository. From these results we found that reducing the length of sequences used by the models to train also deteriorated the model prediction performance for projects in both C and Java benchmarks. For C-Benchmark, the model performance deteriorated by 23.5% in MCC, 22.2% in Precision and 18.1% in Recall. For Java-Benchmark, although we found an improvement of 18.7% in Recall, the overall performance deteriorated by 24.7% in MCC and 28.6% in Precision.

8.4 Impact of Considering Equivalent, Mutants That are Subsuming, i.e., Impact of Potential Mistakes in Our Evaluation

In our experiments, we considered the mutants that were not killed by our test suite as unkillable a.k.a. equivalent. Although this being an undecidable problem (as we elaborated in Section 6.2), we analyzed the impact of what would have happened if the mutants that we considered as equivalent were subsuming instead. Hence, we addressed this by introducing noise in our evaluation, i.e., we assumed 2% equivalent mutants in our evaluation set as subsuming and analyzed the change in performance (MS* achieved) for all the approaches (*Cerebro*, *Decision Trees* and *Random*). We gradually increased the noise percentage from 2% till 10% (i.e., 2%, 4%, 6%, 8%, 10%) and analyzed the change in behaviour for all the approaches (i.e., change in MS*), if it increases or decreases with increase in noise.

We found that *Cerebro*'s and *Decision Trees*' performances are more or less inversely related to the noise in evaluation (Fig. 10). Higher the noise, lower the MS* achieved by both the approaches (with an exception of 10% noise in C benchmark for *Decision Trees* where *Decision Trees* performed better than in case of 8% noise, as detailed in Table 5). For *Random* selection, the performance also deteriorated in most of the cases, with an exception of 10% noise in C benchmark, and 6% and 8% noise in Java benchmark where *Random*'s performance improved by 6.48%, and 0.23% and 1.26% improved MS*, respectively. Despite the reduction in performance due to introduced noise, *Cerebro* still achieves higher MS* than the baselines (Fig. 10).

9 Threats to Validity

External Validity: Threats may relate to the subjects we used. Although our evaluation expands to both C and Java projects

TABLE 4
Impact of the Abstraction Process and Sequence Length in *Cerebro*'s Prediction Performance: On Average, MCC is Decreased by 18% With Unabstracted Code and Decreased by 24% With Sequence Length 25

Average (and Median) Performance in C-Benchmark				
Approach	MCC	F-measure	Precision	Recall
Cerebro-100	0.47 (0.47)	0.41 (0.40)	0.93 (0.93)	0.26 (0.25)
Cerebro-50	0.39 (0.40)	0.34 (0.34)	0.82 (0.82)	0.21 (0.22)
Cerebro-unabs	0.32 (0.31)	0.28 (0.27)	0.70 (0.73)	0.17 (0.16)
Cerebro-25	0.30 (0.29)	0.27 (0.28)	0.64 (0.61)	0.17 (0.18)
Average (and Median) Performance in Java-Benchmark				
Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Cerebro-100	0.45 (0.45)	0.51 (0.52)	0.76 (0.73)	0.39 (0.38)
Cerebro-50	0.38 (0.38)	0.42 (0.42)	0.72 (0.73)	0.31 (0.30)
Cerebro-unabs	0.31 (0.34)	0.43 (0.41)	0.56 (0.53)	0.36 (0.38)
Cerebro-25	0.29 (0.32)	0.42 (0.41)	0.51 (0.45)	0.36 (0.37)

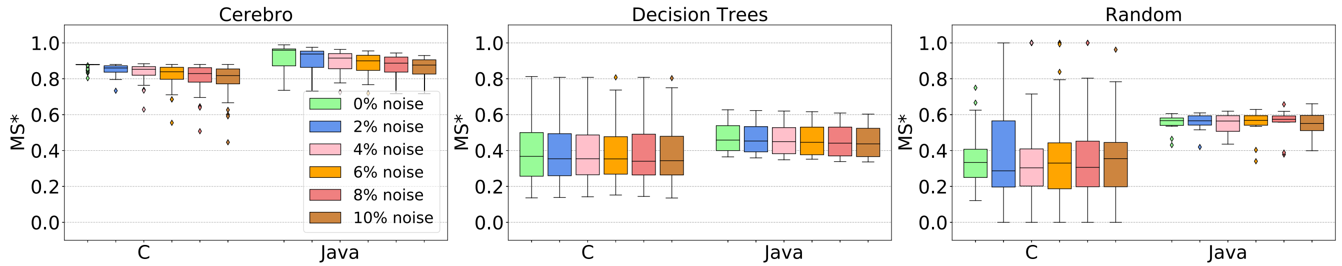


Fig. 10. Impact of noise in evaluation on all approaches' performance (MS*): *Cerebro*'s and *Decision Trees*' performances are more or less inversely related to the noise in evaluation. For *Random* selection, the performance also deteriorated in most of the cases, with exceptions of 10% noise in C benchmark, and 6% and 8% noise in Java benchmark where *Random*'s performance improved by 6.48%, and 0.23% and 1.26% improved MS*, respectively.

of different sizes, the results may not generalize to other projects or programming languages. We consider this threat as low since we have a large sample of programs, i.e., we perform one of the largest mutation testing studies to date.

Other external threat lies in the operators we used, since our prediction approach might not work for different types of mutants. To reduce this threat, we employ modern mutation tools, for both C and Java that implement a large variety of mutation operators. For the C-Benchmark, taken from [12], 816 simple operators across 18 categories were considered; while for creating our Java-Benchmark, we consider the group "ALL" of mutation operators provided by Pitest [17], resulting in 112 simple operators across 29 categories.

Internal Validity: Threats may relate to the restriction that we impose on sequence length, i.e., a maximum of *100* tokens. This was done to enable reasonable model training time, approximately *740* hours. Moreover, restricting the sequence length to *50* assisted to reach an appropriate training time of 360 hours. However, it resulted in a prediction performance deterioration of approximately 15%, as discussed in Section 8. Other threats maybe due to the use of machine translation for classification. This choice was made for simplicity, to use the related framework out of the box, similar to the related studies [60], [61]. Still a potential "sequence to class classifier" may yield better results, though such improvements should be marginal given the low number of unexpected labels we get, i.e., on average, 2.15% of the mutants do not get a valid label (4.2% in C and 0.1% in Java).

Threats may also relate to the features we implemented for training the *Decision Trees* baseline. We follow the guidelines provided in [11], to extract the 16 features for our Java dataset. Unfortunately, many of the 28 features for C programs presented in [11] depend on the semantic of the C language, that we found unfeasible to be replicated in Java. However, the prediction performance of *Decision Trees* in Java are in line with the results obtained for C, indicating that the impact of this threat is low.

Other internal validity threats could be related to the test suites we used and the mutants considered as subsuming and equivalent. To deal with this issue, we used well-tested programs and state-of-the-art tool to generate extensive pools of tests (KLEE [10], SEMu [12], and EvoSuite [23]). Since identifying subsuming and equivalent mutants is an undecidable problem, in our experimental setup, we approximate them through an extensive pool of tests. This has been a typical process followed in related mutation testing studies [3], [27], [34], [45], [46]. To be more accurate, our underlying assumption is that the extensive pool of tests used in our experiments are a valid representation of all possible tests that a tester can manually or automatically generate. This assumption allowed us to identify the minimal set of mutants (i.e., subsuming mutants) that a tester needs to kill in order to kill every other killable mutant (i.e., subsumed mutants). Also, we assumed that unkilled mutants are equivalent. Even if this may not be the case, it is likely that the testers guided by mutation won't be able to kill all the killable mutants. Here it must be noted that since Cerebro is quite precise, its feeding with less noisy data, i.e., correct labels, will make it perform better, i.e., more accurate labelling in training will result in better predictions. Nevertheless, we also investigate the impact of having such noisy data and found minor discrepancies, please refer to Section 8.4.

Cerebro's use may also pose additional threats. In particular, *Cerebro* required approximately 5 minutes for preprocessing of the projects and 5 minutes for classification (decoding results). While this time overhead is low, compared to the hours of test executions, it may still be important. Although our implementation is non-optimal and involves no parallelism, however our encoding and decoding can easily be parallelized, since mutant instances are independent of one another.

TABLE 5
Impact of Noise in Evaluation on All Approaches' Performance (MS*): *Cerebro*'s and *Decision Trees*' Performances are More or Less Inversely Related to the Noise in Evaluation

Performance Change % (Median) in MS* w.r.t. noise for C-Benchmark			
Noise (%)	*Cerebro*	*Decision Trees*	*Random*
2%	↓ -2.24%	↓ -3.61%	↓ -13.91%
4%	↓ -3.10%	↓ -3.59%	↓ -8.89%
6%	↓ -4.67%	↓ -3.83%	↓ -0.95%
8%	↓ -5.78%	↓ -7.40%	↓ -8.17%
10%	↓ -7.06%	↓ -6.53%	↑ +6.48%
Performance Change % (Median) in MS* w.r.t. noise for Java-Benchmark			
Noise (%)	***Cerebro***	***Decision Trees***	***Random***
2%	↓ -2.19%	↓ -1.17%	↓ -0.16%
4%	↓ -4.55%	↓ -1.89%	↓ -0.39%
6%	↓ -6.15%	↓ -2.76%	↑ +0.23%
8%	↓ -7.50%	↓ -3.76%	↑ +1.26%
10%	↓ -8.61%	↓ -4.63%	↓ -2.80%

For Random *selection, the performance also deteriorated in most of the cases, with exceptions of 10% noise in C benchmark, and 6% and 8% noise in Java benchmark where* Random*'s performance improved by 6.48%, and 0.23% and 1.26% improved MS*, respectively.*

Construct Validity: Our assessment metrics, subsuming mutation score, number of equivalent mutants and number of test executions may not reflect the actual testing cost / effectiveness values. These metrics have been suggested by literature [5], [34], [47] and are intuitive, i.e., number of selected and analyzed mutants essentially simulate the manual effort involved by testers, subsuming mutation score the level of covering the test requirements [3], [46], and number of test executions capture the computational effort involved. Here it should be noted that automated test generation tools may reduce this cost but they require testers to check the related test oracles. Similarly, equivalent detection techniques and related heuristics may also reduce the manual effort involved [32]. Though, in C we applied TCE (Trivial Compiler Equivalence) [25], [31] to filter out equivalent and duplicated mutants and our approach still provided significant benefits. Similarly, the use of test executions capture the computational effort involved independently of the test execution framework and optimizations used [15], [47], [64], [69], the machines and the level of parallelization used during test execution. Nevertheless, the differences are substantial making such threats unlikely to happen. Overall, we mitigate these threats by following suggestions from mutation testing literature [5], [34], [47], using state-of-the-art tools, performing several simulations, forming very large and diverse test pools, and got consistent and stable results across our subjects.

10 Related Work

Mutation testing has been established as one of the strongest test criteria [3], [14]. Despite its potential, mutation is considered to be expensive since it introduces too many mutants. To this end, random mutant sampling [19], [49] and selective mutation [43] (restricting mutant instances according to their types) have been proposed as potential solutions. Unfortunately, these approaches fail to capture relevant program semantics and performing similarly to random mutant sampling [11], [34], [68].

Other attempts regard the selection of relevant program locations, which should be mutated. Sun *et al.* [57] proposed selecting mutants that reside in diverse static control flow graph paths. Gong *et al.* [24] identified dominator nodes (using static control flow graph) to select mutants.

More recent attempts regard the identification of interesting mutants (pairs of mutant types and related locations). Petrovic and Ivankovic [51] and Just *et al.* [29] proposed using the code AST in order to identify “useful” mutants. Petrovic and Ivankovic used what they called arid nodes (special AST nodes), while Just *et al.* used the AST parent and child nodes, in order to identify high utility mutants. Mirshokraie *et al.* [41] employed complexity metrics together with test executions to select killable mutants. Similarly, Titcheu *et al.* [11] employed static features, including data flow analysis, complexity and AST information, in order to perform mutant selection, wrt mutants linked with real faults.

In our analysis we approximate the performance of the above approaches through the two baselines we adopt and show that our approach significantly outperforms these. Random mutant sampling is performing comparably to operator mutant selection [68], while the supervised baseline we consider simulates the AST-based and complexity-based approaches.

Perhaps the closest work to ours, is from Marcozzi *et al.* [38], which attempts to identify subsumed mutants using verification techniques (such as weakest precondition). While Marcozzi *et al.*’s approach is particularly powerful, it targets weak mutation and not strong as we do. This results in several false positives in the strong mutation case due to failed error propagation [14]. Moreover, Marcozzi *et al.*’s approach is time consuming, requires complex computations and infrastructure while *Cerebro* is fast and simple. Nevertheless, future research should attempt to combine these methods.

Tufano *et al.* [60] proposed using Neural Machine Translation to learn mutations from bug fixes with the aim of introducing mutations that are syntactically similar to real bugs. *Cerebro* relies on the same technology, though it targets a different problem; the identification of high utility mutants, among those given by regular mutation testing tools, while Tufano *et al.* aim at generating mutants regardless of their potential. This indicates that *Cerebro* can complement Tufano et al by selecting relevant mutants. Nevertheless, we focus on subsuming mutants, that could help measuring test adequacy and designing test suites, which are unlikely to be supported by Tufano *et al.* as there is no notion of subsumption in the bug-fixing sets they use. Moreover, we make no assumption on the availability and repetitiveness of historical bugs and their fixes.

Predictive mutation testing (PMT) [67] attempts to predict whether a given test can kill a given mutant without performing any mutant execution. The approach relies on a set of both static and dynamic features (relying on coverage and code attributes) and achieves relatively good results (on average with 10% error). Though, PMT mainly targets intra-project predictions, while *Cerebro* targets inter-project. Nevertheless, PMT is incomparable to *Cerebro* since it aims at evaluating test execution results, while we do mutant selection prior to any test execution. In other words, we aim at identifying the mutants to be used for test design/generation, while PMT to verify whether mutants are killed by some tests. Therefore, the two methods target different but complementary problems.

Evolutionary Mutation Testing (EMT) [21] utilises dynamic features (execution traces) in order to identify interesting locations and mutant types. As such, EMT requires tests and user feedback, which make it different but complementary to ours; *Cerebro* can set a starting point for EMT or integrate its predictions within EMT’s fitness function. Higher-order mutation [27] aims at dynamically optimizing mutants based on given test suites. This means that Cerebro can be directly applied to support test generation prior to any test generation, while higher-order mutation is only applicable after test generation. Perhaps more importantly, Cerebro does not introduce any expensive dynamic mutant execution, while higher-order mutation introduces major mutant execution overheads.

11 Conclusion and Future Work

We presented *Cerebro*, a method that learns to select subsuming mutants (subset of mutants that subsumes the others, i.e., tests killing them also kill all the mutants of the given mutant set) from given mutant sets. Experiments

with 58 programs showed that *Cerebro* identified subsuming mutants with 0.85 precision and 0.33 recall at an inter-project scenario (trained on different projects than the ones it was evaluated). These predictions enable testers designing test cases capable of killing more than two times the subsuming mutants that they would kill if they were using either randomly selected mutants or another previously proposed machine learning-based mutant selection technique. At the same time *Cerebro* entails the analysis of 66% fewer equivalent mutants and 90% less mutant executions, indicating a large reduction on the practical effort/cost of the approach.

Recently, it has become increasingly common to pre-train the entire model on a data-rich task, which causes the model to develop general-purpose abilities and knowledge that can then be transferred to downstream tasks [52]. In this practice *aka* Transfer Learning and its applications to computer vision [44], [63], pre-training is typically done via supervised learning on a large labeled data set like ImageNet [53]. In contrast, modern techniques for transfer learning in Natural Language Processing (*NLP*) often pre-train using unsupervised learning on unlabeled data [20], [36]. The resulting pre-trained models are further trained on specialized datasets to accomplish the desired tasks. Unsupervised pre-training for *NLP* is attractive and seems a good fit for neural networks as it have been shown to exhibit remarkable scalability, i.e., it is often possible to achieve better performance simply by training a larger model on a larger data set [26], [28], [37], [54]. It will be worthwhile to explore such available pre-trained models [22], [39] and if these can be further refined to address our specific prediction task.

On the other hand, as we have shown that *Cerebro* is proficient in capturing the silent features and patterns of the code context, it is promising to explore *Cerebro* in security-specific task such as prediction of zero-day vulnerabilities, which pose a very high risk [66]. Vulnerabilities are fewer in comparison to defects, limiting the information one can learn from. Also, their identification requires an attacker's mindset [42], which developers or code reviewers may not possess. Lastly, the continuous growth of codebases makes it difficult to investigate them entirely and track all code changes. For instance, Linux kernel, which is one of the projects with the highest number of publicly reported vulnerabilities, reached 27.8 million LoC (Lines of Codes) at the beginning of 2020 [35]. Hence, it will also be rewarding to explore *Cerebro* in this line of work.

References

[1] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: tensorflow.org.

[2] A. T. Acree, "On Mutation," Ph.D. thesis, School of Information and Computer Science, Georgia Inst. Technol., Atlanta, Georgia, 1980.

[3] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Proc. IEEE 7th Int. Conf. Softw. Testing, Verification Valid.*, 2014, pp. 21–30.

[4] P. Ammann, J. Offutt, "Introduction to Software Testing," 1st ed. New York, NY, USA: Cambridge Univ. Press, 2008.

[5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[6] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate, in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.

[7] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2016, pp. 4945–4949.

[8] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, "Massive exploration of neural machine translation architectures," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2017, pp. 1442–1451.

[9] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Inf.*, vol. 18, no. 1, pp. 31–45, Mar. 1982.

[10] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, 2008, pp. 209–224.

[11] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Yves Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empir. Softw. Eng.*, vol. 25, no. 1 pp. 434–487, 2020.

[12] T. T. Chekam, M. Papadakis, M. Cordy, and Y. Le Traon, "Killing stubborn mutants with symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Jan. 2021.

[13] T. T. Chekam, M. Papadakis, and Y. Le Traon , "Mart: A mutant generation tool for LLVM," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, page 1080–1084.

[14] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 597–608.

[15] L. Chen, L. Zhang, "Speeding up mutation testing via regression test selection: An extensive study," in *Proc. 11th IEEE Int. Conf. Softw. Testing Verification Valid.*, 2018, pp. 58–69.

[16] K. Cho *et al.*, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1724–1734.

[17] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for java (demo)," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 449–452.

[18] M. L. Collard, J. I. Maletic, "srcML 1.0: Explore, analyze, and manipulate source code," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution*, 2016, pp. 649–649.

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, 2019, pp. 4171–4186.

[21] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo , "Evolutionary mutation testing," *Inf. Softw. Technol.*, vol. 53, no. 10, pp. 1108–1123, 2011.

[22] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics: EMNLP*, 2020, pp. 1536–1547.

[23] G. Fraser, A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proc. ACM Int. Symp. Softw. Testing Anal.*, 2010, pp. 147–158.

[24] D. Gong, G. Zhang, X. Yao, and F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," *Inf. Softw. Technol.*, vol. 81, pp. 82–96, 2017.

[25] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler intermediate representation," in *Proc. 12th IEEE Conf. Softw. Testing, Valid. Verification*, 2019, pp. 114–124.

[26] J. Hestness *et al.*, "Deep learning scaling is predictable, empirically," 2017, *arXiv:1712.00409*.

[27] Y. Jia, M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, 2009.

[28] R. Józefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," 2016, *arXiv:1602.02410*.

[29] René Just, B. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 284–294.

[30] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Proc. Asia Pacific Softw. Eng. Conf.*, 2010. pp. 300–309.

[31] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, “Detecting trivial mutant equivalences via compiler optimisations,” *IEEE Trans. Softw. Eng.*, vol. 44, no. 4, pp. 308–333, Apr. 2018.
[32] M. Kintis, M. Papadakis, and N. Malevris, “Employing second-order mutation for isolating first-order equivalent mutants,” *Softw. Testing Verification Rel.*, vol. 25, no. 5/7, pp. 508–535, 2015.
[33] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, “Mutant subsumption graphs,” in *IEEE 7th Int. Conf. Softw. Testing Verification Valid. Workshops*, 2014, pp. 176–185.
[34] B. Kurtz, P. Ammann, J. Offutt, Márcio, E. Delamaro, M. Kurtz, and Nida Gökçe, “Analyzing the validity of selective mutation with dominator mutants,” in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 571–582.
[35] Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd,” Accessed: Oct. 12, 2020. [Online]. Available: https://www.theregister.com/2020/01/06/linux_2020_kernel_systemd_code/
[36] Z. Liu, W. Lin, Ya Shi, and J. Zhao, “A robustly optimized bert pre-training approach with post-training,” in *Proc. China Nat. Conf. Chin. Comput. Linguistics*, 2021, pp. 471–484.
[37] D. Mahajan *et al.*, “Exploring the limits of weakly supervised pre-training,” in *Proc. Eur. Conf. Comput. Vis*, 2018, pp. 181–196.
[38] M. Marcozzi, S. Bardin, N. Kosmatov, M. Papadakis, V. Prevosto, and Loïc Correnson, “Time to clean your test objectives,” in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 456–467.
[39] A. Mastropaolo *et al.*, “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 336–347.
[40] B. W. Matthews, “Comparison of the predicted and observed secondary structure of t4 phage lysozyme,” *Biochimica et Biophysica Acta (BBA) - Protein Struct.*, vol. 405, no. 2, pp. 442–451.
[41] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, “Guided mutation testing for javascript web applications,” *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 429–444, May 2015.
[42] P. Morrison, K. Herzig, B. Murphy, and L. Williams, “Challenges with applying vulnerability prediction models,” in *Proc. Symp. Bootcamp Sci. Secur.*, 2015, pp. 1–9.
[43] 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.
[44] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, “Learning and transferring mid-level image representations using convolutional neural networks,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 1717–1724.
[45] M. Papadakis, T. T. Chekam, and Y. Le Traon, “Mutant quality indicators,” in *Proc. IEEE Int. Conf. Softw. Testing Verification Valid. Workshops*, 2018, pp. 32–39.
[46] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon , “Threats to the validity of mutation-based test assessment,” in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 354–365.
[47] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, “Chapter six - Mutation testing advances: An analysis and survey,” *Advances Comput.*, vol. 112, pp. 275–378, 2019.
[48] M. Papadakis and N. Malevris, “Automatic mutation test case generation via dynamic symbolic execution,” in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, 2010, pp. 121–130.
[49] M. Papadakis and N. Malevris, “An empirical evaluation of the first and second order mutation testing strategies,” in *Proc. 3rd Int. Conf. Softw. Testing Verification Valid.*, 2010, pp. 90–99.
[50] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, “Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults,” in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 537–548.
[51] G. Petrovic and M. Ivankovic, “State of mutation testing at google,” in *Proc. 40th IEEE/ACM Int. Conf. Softw. Eng. Softw. Eng. Pract. Track*, 2018, pp. 163–171.
[52] C. Raffel *et al.*, “Exploring the limits of transfer learning with a unified text-to-text transformer,” 2019, *arXiv:1910.10683*.
[53] O. Russakovsky *et al.*, “Imagenet large scale visual recognition challenge,” *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.
[54] N. Shazeer, *et al.*, “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer,” in *Proc. 5th Int. Conf. Learn. Representations*, 2017.
[55] M. Shepperd, D. Bowes, and T. Hall, “Researcher bias: The use of machine learning in software defect prediction,” *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, Jun. 2014.
[56] A. Shewalkar, D. Nyavanandi, and S. Ludwig, “Performance evaluation of deep neural networks applied to speech recognition: RNN, LSTM and GRU,” *J. Artif. Intell. Soft Comput. Res.*, vol. 9, pp. 235–245, 2019.
[57] C.-AI Sun, F. Xue, H. Liu, and X. Zhang, “A path-aware approach to mutant reduction in mutation testing,” *Inf. Softw. Technol.*, vol. 81, pp. 65–81, 2017.
[58] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks, in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[59] G. Tang, M. Müller, A. Rios, and R. Sennrich, ”Why self-attention? A targeted evaluation of neural machine translation architectures,” in *Proc. 2018 Conf. Empir. Methods Natural Lang. Process.*, 2018, pp. 4263–4272.
[60] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, ”Learning how to mutate source code from bug-fixes,” in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution*, 2019, pp. 301–312.
[61] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, ”An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.
[62] A. Vargha and H. D. Delaney, ”A critique and improvement of the ”CL” common language effect size statistics of McGraw and wong,” *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.
[63] J. Yangqing *et al.*, ”Convolutional architecture for fast feature embedding,” in *Proc. ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[64] B. Wang, Y. Xiong, Y. Shi, Lu Zhang, and D. Hao, ”Faster mutation analysis via equivalence modulo states,“ in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 295–306.
[65] X. Yao, M. Harman, and Y. Jia, “A study of equivalent and stubborn mutation operators using human analysis of equivalence,” in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 919–930.
[66] Zero-day vulnerability,” Accessed: Oct. 12, 2021. [Online]. Available: https://www.trendmicro.com/vinfo/us/security/definition/zero-day-vulnerability
[67] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, “Predictive mutation testing,” *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, pp. 898–918, Sep. 2019.
[68] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, “Operator-based and random mutant selection: Better together,” in *28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 92–102.
[69] L. Zhang, D. Marinov, and S. Khurshid, “Faster mutation testing inspired by test prioritization and reduction,” in *Proc. Int. Symp. Softw. Testing Anal*, 2013, pp. 235–245.

Aayush Garg received the MS degree in computer science with a concentration in security from Boston University, Boston, Massachusetts, in 2019. He is a doctoral researcher with the Department of Computer Science, Faculty of Science, Technology and Medicine (FSTM), University of Luxembourg. He has several years of industrial experience as a software developer in Fintech Organizations. His research areas comprise computer security, computational intelligence in software engineering, and mutation testing.

Milos Ojdanic received the MSc degree from the Faculty of Innovation, Design, and Technology, Mälardalen University, Västerås, Sweden, in 2019. He is a doctoral researcher with the Interdisciplinary Center for Security, Reliability, and Trust (SnT), University of Luxembourg. His research interests include software development, testing, and evolution. In particular, he focuses on evolving systems, change-aware testing criteria, mutation testing, and prediction modeling.

Renzo Degiovanni received the PhD degree in computer science from the National University of Cordoba, Cordoba, Argentina. He is a research associate with the Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg. His research interests include software engineering, specifically the validation and verification of software. His research has contributed to the automation of requirements engineering activities, software testing, and formal software verification.

Thierry Titcheu Chekam received the BSc degree in computer science and technology from the University of Science and Technology of China, Hefei, China, in 2013, the MEng degree in software engineering from the School of Software, Tsinghua University, Beijing, China, in 2015, and the PhD degree in computer science from the University of Luxembourg, Esch-sur-Alzette, Luxembourg. He is a software engineer with SES, Luxembourg. His research areas comprise software testing, mutation analysis, symbolic execution, and cloud computing/storage.

Mike Papadakis received the PhD degree in computer science from the Athens University of Economics and Business, Athens, Greece. He is a senior research scientist with the Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg. He is recognised for his work on software testing and in particular in the area of mutation testing. His research interests also include static analysis, prediction modelling, and search-based software engineering.

Yves Le Traon is professor with the University of Luxembourg, where he leads the SERVAL (SEcurity, Reasoning and VALidation) Research Team. His research interests within the group include (1) innovative testing and debugging techniques, (2) Android apps security and reliability using static code analysis, machine learning techniques and, (3) model-driven engineering with a focus on IoT and CPS. His reputation in the domain of software testing is acknowledged by the community. He has been general chair of major conferences in the domain, such as the 2013 IEEE International Conference on Software Testing, Verification and Validation (ICST), and program chair of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). He serves at the editorial boards of several, internationally-known journals (*Software Testing, Verification & Reliability*, *Software and Systems Modeling*, *IEEE Transactions on Reliability*) and is author of more than 150 publications in international peer-reviewed conferences and journals.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.