# Predicting the Fault Revelation Utility of Mutants

Thierry Titcheu Chekam, Mike Papadakis, Tegawendé Bissyandé and Yves Le Traon
SnT Centre, University of Luxembourg
{thierry.titcheu-chekam,michail.papadakis,tegawende.bissyande,yves.letraon}@uni.lu

## ABSTRACT

Mutation testing is one of the strongest code-based test criteria. However, it is expensive as it involves a large number of mutants. To deal with this issue we propose a machine learning approach that learns to select fault revealing mutants. Fault revealing mutants are valuable to testers as their killing results in (collateral) fault revelation. We thus, formulate mutant reduction as the problem of selecting the mutants that are most likely to lead to test cases that uncover unknown program faults. We tackle this problem using a set of static program features and machine learning. Experimental results involving 1,629 real faults show that our approach reveals 14% to 18% more faults than a random mutant selection baseline.

## KEYWORDS

Mutation testing, Fault Revelation, Machine Learning

## 1 INTRODUCTION

Mutation testing is considered as one of the most effective, at finding faults, testing technique [2]. However, it is expensive. This is mainly due to the large number of mutants that testers need to manually analyse. To deal with this issue, mutant selection methods have been devoted [4]. However, these techniques focus at selecting, among a large set of mutants, those mutants that are of the same power, i.e., every test suite that kills the mutants of the smaller set also kills the mutants of the large set. However, we argue that the actual objective should be to select only those mutants that are of practical value, such as those that lead to fault revelation.

Our goal is to form a mutant selection technique that precisely identifies the valuable mutants, prior to any mutant generation or execution. Depending on the use we intend to do, we can consider different mutants as valuable. Thus, one might consider that the valuable mutants are those that help developers rethink their implementation, someone else might consider those that are killable or subsuming. Nevertheless, we focus on the fault revealing mutants. However, our approach is general and could be tuned to identify additional types of valuable mutants.

We advance in this research direction by proposing a machine learning approach that learns the static properties of the valuable mutants, such as mutant type, location, code complexity, control and data dependencies. We thus leverage the knowledge we gain from historical data, (train on a set of known faults prior to any testing or test case design) and classify the mutants of the program under test as fault revealing. This way testers can focus on the most promising mutants and apply mutation on a best-effort basis.

*ICSE 2018, May 27 - 3 June 2018, Gothenburg, Sweden*
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

**Table 1: Description of the static code features**

| | |
|---|---|
| **Complexity** | Complexity of statement $S_M$ approximated by the total number of mutants on $S_M$ |
| **CfgDepth** | depth of $B_M$ according to CFG |
| **CfgPredNum** | number of predecessors basic blocks, according to CFG, of $B_M$ |
| **CfgSuccNum** | number of successors basic blocks, according to CFG, of $B_M$ |
| **AstNumParents** | number of AST parents of $S_M$ |
| **NumOutDataDeps** | number of mutants on statements data-dependents on $S_M$ |
| **NumInDataDeps** | number of mutants on statements on which $S_M$ is data-dependent |
| **NumOutCtrlDeps** | number of mutants on statements control-dependents on $S_M$ |
| **NumInCtrlDeps** | number of mutants on statements on which $S_M$ is control-dependent |
| **NumTieDeps** | number of mutants on $S_M$ |
| **AstParentsNumOutDataDeps** | number of mutants on statements data-dependent on$S_M$'s AST parent statement |
| **AstParentsNumInDataDeps** | number of mutants on statements on which $S_M$'s AST parent statement is data-dependent |
| **AstParentsNumOutCtrlDeps** | number of mutants on statements control-dependent on $S_M$'s AST parent statement |
| **AstParentsNumInCtrlDeps** | number of mutants on statements on which $S_M$'s AST parent statement is controle-dependent |
| **AstParentsNumTieDeps** | number of mutants on $S_M$'s AST parent statement |
| **TypeAstParent** | statement type of AST parent statement of $S_M$ |
| **TypeMutant** | mutant type of M as matched code pattern and replacement. Ex: $a + b \to a - b$ |
| **TypeStmtBB** | CFG basic block type of $B_M$. Ex: $if-then,\ if-else$ |
| **AstParentMutantType** | mutant type of M's AST parent |
| **OutDataDepMutantType** | mutant types of mutants on statements data-dependents on $S_M$ |
| **InDataDepMutantType** | mutant types of mutants on statements on which $S_M$ is data-dependent |
| **OutCtrlDepMutantType** | mutant types of mutants on statements control-dependents on $S_M$ |
| **InCtrlDepMutantType** | mutant types of mutants on statements on which $S_M$ is control-dependent |
| **AstChildHasIdentifier** | AST child of statement $S_M$ has an identifier |
| **AstChildHasLiteral** | AST child of statement $S_M$ has a literal |
| **AstChildHasOperator** | AST child of statement $S_M$ has an operator |
| **DataTypesOfOperands** | Data types of operands of $S_M$ |
| **DataTypeOfValue** | Data type of the returned value of $S_M$ |

Experimental results using 10-fold cross validation on 1,629 faults, from the CodeFlaws benchmark [5], show a high performance of our approach. In particular our mutant selection method achieves significantly better results than random mutant selection by revealing 12% to 20% more faults. We also show that static program features related to control and data program dependencies are the most effective ones at determining the mutants' utility.

## 2 MUTANT SELECTION PROBLEM

We define the mutant selection problem as the problem of statically (prior to any test execution) selecting a subset of mutants, given a large set of mutants, that maximize a dynamic property (only known after the test execution) of interest. Here, we focus on selecting the mutants that are most likely to lead to test cases that uncover unknown defects, given a set of static, computationally inexpensive, features.

We tackle the mutant selection problem by using 28 static features that are listed in Table 1. In this table, $B_M$ is the basic block, of the program Control Flow Graph (CFG), associated with a mutated statement $S_M$.

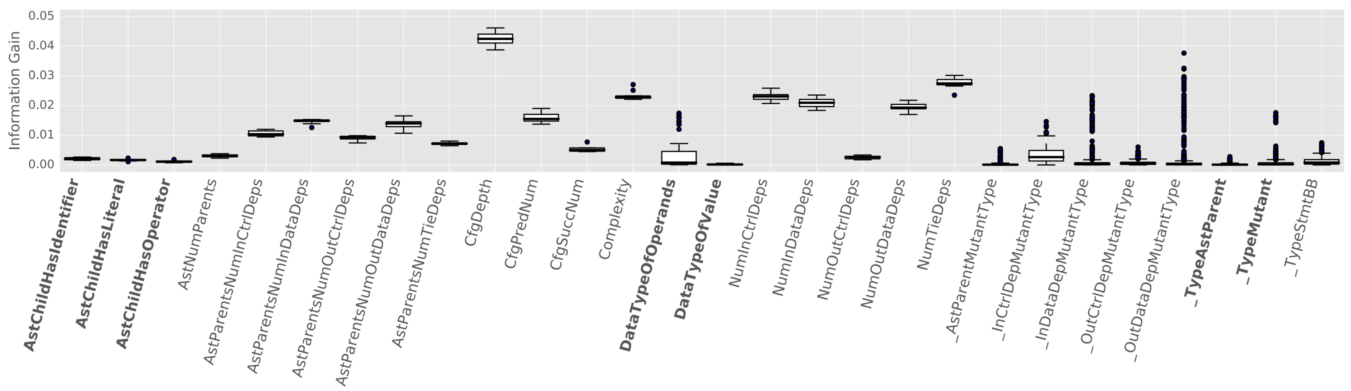

**Figure 1: Information Gain distributions of ML features**

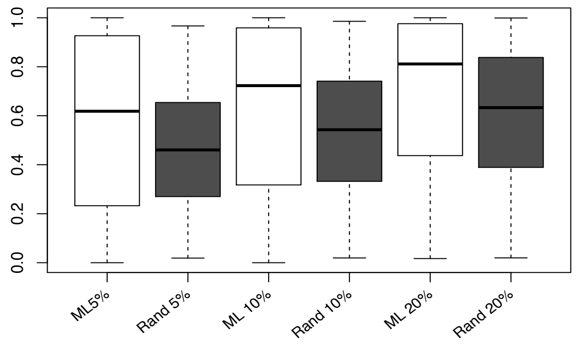

**Figure 2: Fault revelation of the mutant selection strategies**

## 3 RESULTS

To evaluate our approach we used CodeFlaws [5]. The benchmark has 3,902 faulty program versions of 40 defect classes. The programs are accompanied by validation test suites, which we augment with KLEE [1]. We thus formed large pools of test cases from which we sample multiple times and construct mutation adequate test suites (for the mutant selection strategies we study). To evaluate our approach, we used the faults that are revealed by less than 25% of the tests in our test pools. These were 1,629 faults.

We applied mutation testing on the faulty program versions, so that we are faithful to real settings and avoid making the CPA hypothesis [2]. We used 18 mutation operators[1], applied at the LLVM bitcode. To reduce the influence of redundant and equivalent mutants, we applied TCE [3] using the LLVM-diff utility.

During the training phase, for each mutant we extract its feature values and we feed them along with the fault revealing information to a machine learning classification algorithm (stochastic gradient boosting decision trees). The outcome, is a prediction model that predicts the mutants' fault revealing potential (given the feature values). In other words, given the feature values of the candidate mutants the classifier predicts their potential. During testing time, the classifier ranks the mutants according to their (predicted) fault revealing potential, which is then used by the tester.

Here, we performed a 10-fold cross-validation (train on 90% of the faults and evaluate on the 10% for 10 times) and evaluate the actual fault revealing ability of the top 5%, 10% and 20% of the ranked mutants.

Figure 1 depicts the distribution of information Gain values for the features we use. These data enable the assessment of the potential contribution of every feature to the prediction model we built. Interestingly, together with complexity, the features related to control and data dependencies are the most important ones.

Figure 2 shows the distribution of the fault revelation of the mutant selection strategies when selecting the top 5%, 10% and 20% of the ranked mutants. As can be seen from the plot our approach outperforms the random selection. At the 5%, 10% and 20% thresholds the difference of the median values is 14%, 18% and 18%. These differences are also statistically significant[2].

## 4 CONCLUSION

The large number of mutants involved in mutation testing has long been identified as a barrier to the practical application of the method. To deal with this issue, we introduce a new perspective to the problem, the fault revelation mutant selection. We demonstrate that fault revealing mutants can be identified through simple static program features and standard machine learning techniques. We provide results showing that machine learning helps and leads to significantly higher fault revelation than random mutant selection. We also show that mutants' utility can be captured through control and data dependence related features.

## REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. 209–224.

[2] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *ICSE*. 597–608.

[3] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *ICSE*. 936–946.

[4] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2018. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* (2018).

[5] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *ICSE*. 180–182.

[1]Statement Deletion, Trap statements, Operand Swapping, Left/Right operand, Logical connector Replacement, Absolute Value Insertion, Non-Pointer Unary Operator Insertion, Non-Pointer Unary Operator Stripping, Constant Value Replacement, Non-Pointer Binary Operator Replacement, Function Call arguments shuffling, Switch Cases shuffling, Pointer Binary Operation Replacement, Pointer Unary Operation Replacement, Pointer Dereference, Binary Operation to Unary operator

[2]we performed a Wilcoxon rank-sum test, with significance level $a < 0.001$