If You Can't Kill a Supermutant, You Have a Problem

Rahul Gopinath*, Björn Mathis†, Andreas Zeller‡
Saarland University
Email: *rahul@gopinath.org, †bjoern.mathis@cispa.saarland
, ‡zeller@cs.uni-saarland.de

***Abstract*—Quality of software test suites can be effectively and accurately measured using mutation analysis. Traditional mutation involves seeding first and sometimes higher order faults into the program, and evaluating each for detection. However, traditional mutants are often heavily redundant, and it is desirable to produce the complete matrix of test cases vs mutants detected by each. Unfortunately, even the traditional mutation analysis has a heavy computational footprint due to the requirement of independent evaluation of each mutant by the complete test suite, and consequently the cost of evaluation of complete kill matrix is exorbitant.**

We present a novel approach of combinatorial evaluation of multiple mutants at the same time that can generate the complete mutant kill matrix with lower computational requirements.

Our approach also has the potential to reduce the cost of execution of traditional mutation analysis especially for test suites with weak oracles such as machine-generated test suites, while at the same time liable to only a linear increase in the time taken for mutation analysis in the worst case.

I. Introduction

Mutation analysis involves exhaustive injection and analysis of first and higher order faults [8]. Traditional mutation analysis involves evaluating each mutant by running individual test cases against that mutant until the mutant is killed (detected) by one of the test cases. One of the problems with mutation analysis is the issue of redundant mutants [17]. That is, a large number of mutants represent faults that are strictly subsumed by other mutants, and hence contributes nothing to the overall effectiveness of mutation analysis [18], [31]. Another major problem with mutation analysis is the heavy computational requirement. Each mutant needs to be evaluated independently by potentially the complete test suite in order to determine if the mutant was successfully detected by one of the test cases [28]. As the number of mutants that can be generated from even a simple program is very large, mutation analysis is computationally intensive, which has limited its practicality.

The best method for accounting for redundancy in mutants is of course to simply evaluate the complete set of mutants against the full set of test cases, and eliminate redundant mutants resulting in the so-called *minimal mutants* [1]. We call the resulting matrix of results the *mutant kill matrix*[1].

The main reason for the high cost of mutation analysis is the requirement of an independent evaluation of each mutant. This is especially aggravated in the case of computing the *mutant kill matrix*[2] because each mutant needs to be evaluated by executing it against each test case. Note that the number of mutants that can be generated from a program is typically of the same magnitude as the number of lines in a program. This in turn results in a much larger number of test executions, which means that any method that can reduce the number of test executions needed can have a large impact on the total cost of analysis.

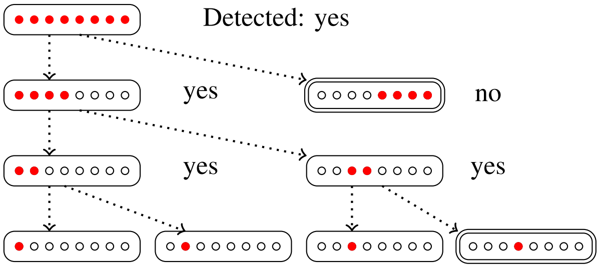

Fig. 1: Evaluation of supermutants. The filled dots indicate mutations applied within the supermutants. The non detected mutants are indicated by doubled borders.

We propose a novel method for computing the mutant kill matrix that can potentially avoid the requirement of individual evaluation of each mutant using supermutants. A supermutant (see Figure 3) is a mutant of a program consisting of several small changes in the program, all applied at once. From previous research [9], we know that the incidence of fault masking is very rare Hence, if a test suite cannot find the supermutant, it is unlikely that it is able to find one of the individual mutants. Hence, when ever we find a supermutant that escapes detection, we mark the constituent mutants as having escaped detection by the test suite. On the other hand, if the supermutant is detected, we partition the component mutations in the supermutant, generate two mutants containing each set of mutations, and recurse into the mutants generated from applying each set.

The biggest threat to the effectiveness of our technique is

[1] The *mutant kill matrix* is also used for other research [24].

[2] While in traditional mutation analysis, one stops evaluation of a mutant as soon as any of the covering test cases manage to kill the mutant being considered. However, if one wants to compute the *mutant kill matrix*, one needs to execute the complete set of covering test cases. That is, traditional mutation analysis can have a complexity of $n \times 1$ in the best case, and $n \times t$ in the worst case, where n is the number of mutants, and t the number of test cases, for computing the mutant kill matrix, one always has $n \times t$ complexity.

the presence of trivially killed mutants that can force recursion into the child nodes containing first order mutants. Indeed, a majority of mutants are trivial, and can be detected by a covering test case. Hence, we filter out these mutations by using a minimal[3] coverage-adequate[4] test-suite with assertions removed (We also ensure that any single mutant is evaluated by exactly one test case). Any mutant that is killed by a covering test suite with its assertions removed may be marked as killed by all other test cases that covers that mutant, and hence need not be evaluated by supermutants again.

The steps for evaluation of supermutants are as follows:

A. Evaluating supermutants for the mutant kill matrix

We only consider *lexically non-overlapping* mutations such as first-order deletions initially. We then discuss how other mutations can be evaluated similarly.

1) Generate a test suite that is minimally *coverage adequate*.
2) Identify mutants that can be killed by this minimal test suite. Next, verify that the mutants are indeed trivial to kill, and hence any covering test case will detect them (This can be accomplished by removing the assertions before executing the test suite).
3) Generate a composite supermutant containing all the first-order mutations possible in the program.
4) Pick any test case that covers any of the mutations and check whether it kills the composite supermutant.
5) If the composite supermutant was not killed, declare the test case as killing none of the component mutants.
6) If the composite supermutant was killed by the test case, partition the mutations randomly into two equal groups, and combine each group into a child supermutant.
7) Evaluate the test case on each child supermutant.
8) If the child supermutant is actually a first-order mutant, we evaluate the test case on it and stop and return the result of evaluation for that mutation because there is nothing left to recursively evaluate.
9) If one or both of the child supermutants are detected by the test case, we recursively evaluate smaller and smaller partitions until we reach the first-order mutants.
10) If none of the children are detected, we prioritize this mutant for later evaluation, but return with an empty mutant kill matrix.
11) The evaluation results in the mutant kill matrix for that specific test case.
12) Continue the evaluation for each test case in the same manner to get the complete mutant kill matrix.

How do we deal with mutation points such as arithmetic operators and boolean operators that can result in multiple overlapping mutations? While these mutations cannot be included in the same supermutant, one can produce multiple supermutants each containing non-overlapping mutations, and evaluate them as we detailed above.

What is the runtime for the evaluation of supermutants? In the best case, we will find that the supermutant is not killable by any of the test cases in which case, we require only a *single* execution of the supermutant per covering test case. If the mutations and test suites are such that we have a high mutation score test suite, but with low redundancy, the first supermutant would be detected, and we will be forced to recurse into the child supermutants. However, it is likely that at each step a significant fraction of test cases would fail to detect the supermutant. If a test case fails to detect the supermutant, then there is little chance of it detecting its component mutants due to the rarity of fault masking. Hence these test cases can be be eliminated from deeper recursion on that particular supermutant. Thus, even in this scenario, our technique has the potential to be significantly cheaper. If on the other hand, *all* mutants are indeed killable by *each* test case, then we will require $2n - 1$ evaluations of mutants per test case rather than n mutant evaluations per test case where n is the number of mutations possible. Indeed, the stronger the mutants are, or lower the mutation score is, the higher the probability that supermutants may result in significant cost savings.

The same procedure may be adapted for traditional mutation analysis, and can prove gainful when the test suite has weak oracles, and the probable mutation score is low. The adaptation of the supermutants for traditional mutation analysis is simple.

B. Adapting supermutants for traditional mutation analysis

1) Generate a test suite that is minimally *coverage adequate*.
2) Use this coverage adequate test suite in the first phase of mutation analysis to identify trivial mutants that are easily killed, and the test cases sufficient to kill this set of trivial mutants.
3) Collect the remaining test suites and remaining mutations possible on the program.
4) Produce a supermutant with all remaining mutations. (Steps until this point is common.)
5) Identify and produce a test suite that covers the program locations of mutations.
6) Run the test suite against the supermutant by choosing one test case at a time. Stop at the *first* test case that was able to kill the super mutant.
7) If the supermutant was not detected, None of the mutants could be found by the given test suite.
8) If the supermutant was detected, partition the mutations in the supermutant into two equal groups randomly. Generate a new test suite by discarding the test cases from the previous test suite that were *evaluated, and failed to detect* the parent supermutant. Apply the new test suite against each smaller supermutant.
9) If neither mutant was detected by the test suite, mark the parent supermutant for later analysis
10) If one of the child supermutants were detected, proceed to analyze the child mutant similarly until we reach

[3] We use a minimal test suite to ensure that we use the least effort necessary to remove these trivial mutants.

[4] From here on, we consider any subset of test cases of the complete test suite to be *coverage adequate* if it has covered all the statements possible by the complete test suite. We also assume that we have the coverage information for each of the test cases.

first-order mutants.

11) If both of the child supermutants were detected, proceed to analyse both child mutants similarly.
12) Once we reach the leaf node, mark the simple mutants as detected if they are detected.

The procedure of evaluation of a supermutant is summarized in Figure 1. As can be seen in the figure, at each step, the supermutant is decomposed into smaller and smaller sets of mutations, ending when the mutations are the smallest possible.

There are a few possible gains from using such a technique. In the first case, where one is interested in speeding up traditional mutation analysis, as long as the oracle being tested against is weak[5] or the mutants are stubborn[6], or there are numerous equivalent mutants, we have a reasonable chance of being faster than the traditional – mutation at a time – analysis. Second, complex mutations where the supermutant is detected but neither the child mutants are detected, are interesting in their own right, because the child mutants have a higher probability of being non-equivalent and *stubborn* on account of there being at least some effect in the program due to the mutation. The supermutant can also be minimized to *1-minimal form*[7] using delta debugging to produce a strong higher-order mutant. There are further avenues for optimization. For example, the utility of this approach is enhanced if there is a means to classify stubborn mutants so that they can be clubbed together. It was previously found [33] that once we consider reachability, specific operators that produce off-by-one errors are more likely to result in stubborn mutants. These mutations could then be chosen to form a supermutant that may be evaluated separately.

C. Reduction of supermutants to 1-minimal form

The evaluation procedure of supermutants may often result in situations where the parent supermutant was detected by a particular test case while neither of the children are detected by the same test case. When this happens, it is an indication that the combination of mutations expresses a fault that is different from the faults originally expressed by the individual mutations. In such cases, one may apply the delta debugging [38] algorithm to isolate the particular mutations necessary to trigger the fault. By carefully varying the partitions during each mutation analysis, one may collect different higher-order mutants that express different faults and hence improve the effectiveness of mutation analysis.

We note that traditional mutation analysis is most expensive when it evaluates stubborn and equivalent mutants – where a majority of test cases do not detect the mutant being examined. It is precisely for these mutants that our technique is most effective. Hence we believe that supermutants can be more effective provided one is able to identify these mutants in advance.

Contributions:

- A novel method for reducing the computational overhead in producing a complete mutant kill matrix provided the mutants could be pre-classified as trivial or stubborn.
- A novel method for reducing the computational overhead for traditional mutation analysis especially for test suites with weak oracles.

II. Case Studies

Our case study is aimed at investigating the effectiveness of supermutants in reducing the computational effort required for mutation analysis. We consider two programs for our study – the *triangle* program and *urlparse* program. The effectiveness of supermutants is linked strongly to the test suite being used. Hence, for both programs, we evaluate the effectiveness of our technique using two test suites of different oracular strengths. For evaluating the mutation score of both programs, for demonstration purposes, we chose simple statement deletion of *leaf* statements. That is, for the triangle, only *return* statements would be replaced with *pass*. We evalute supermutants on these two programs below.

A. Analyzing the triangle program

Figure 2 contains the well known triangle classification program [22] used in numerous mutation studies [13], [20], [27], along with two test suites given in Figure 5 and Figure 6. The *triangle* program classifies any given valid numeric triple as one of the three: *Equilateral*, *Isosceles*, *Scalene*. The test suite in Figure 5 is a strong test suite (100% coverage) with a weak oracle which only checks for the validity of the triple, while the test suite in Figure 6 is a test suite with similar coverage, but with a stronger oracle. The small triangle marks in Figure 2 indicate the possible mutations. We first use a statement coverage adequate test suite to filter out trivial mutants. The ▶ mark represent trivial mutants that are removed by the test suite in Figure 5. Figure 3 shows the preliminary supermutant generated after removing the trivially detected mutants.

In this case, evaluating the supermutant in Figure 3 with the test suite in Figure 5 results in the supermutant not being found, which immediately suggests that none of the mutants are likely to be found, resulting in saving 5 mutant evaluations. However, we note that the savings are also dependent on the strength of test suite. For example, the test suite in Figure 6 will not result in any savings as this test suite is both minimally coverage adequate and strong enough to detect all mutants with the first test case that covers them. That is, the first test suite is an example where our approach can lead to savings, while the second test suite is an example of a pitfall, where our approach can be more expensive than traditional analysis.

[5] A weak oracle is an oracle that asserts weak facts such as no crash.

[6] A mutant is stubborn if a high-quality test suite is unable to detect it, but is not equivalent [37].

[7] A *1-minimal* supermutant is a mutant that is detected by some test suite such that removal of any of the component mutations results in it not being detected by that test suite.

Fig. 2: The *Triangle* program

```
  def triangle(a, b, c):
▶   assert a + b > c and a + c > b and b + c > a
    if a = b and b = c:
▷           return 'Equilateral'
    if a = b:
▷           return 'Isosceles'
    if b = c:
▷           return 'Isosceles'
    if a = c:
▷           return "Isosceles"
▷   return "Scalene"
```

Fig. 3: The *Triangle* supermutant

```
def triangle(a, b, c):
  assert a + b > c and a + c > b and b + c > a
  if a = b and b = c:
         pass
  if a = b:
         pass
  if b = c:
         pass
  if a = c:
         pass
  pass
```

Fig. 4: ⊥ indicates an assertion failure is expected, ⊤ indicates no assertion failure, and ⊢ indicates that an assertion is *true*

Fig. 5: Test Suite 2 (weak oracle)

```
def testsuite():
    ⊥ triangle(0,1,1)
    ⊤ triangle(1,1,1)
    ⊤ triangle(2,1,2)
    ⊤ triangle(2,2,1)
    ⊤ triangle(1,2,2)
    ⊤ triangle(3,4,5)
```

Fig. 6: Test Suite 1 (strong oracle)

```
def testsuite():
    ⊥ triangle(0,1,1)
    ⊢ triangle(1,1,1) = 'Equilateral'
    ⊢ triangle(2,1,2) = 'Isosceles'
    ⊢ triangle(2,2,1) = 'Isosceles'
    ⊢ triangle(1,2,2) = 'Isosceles'
    ⊢ triangle(3,4,5) = 'Scalene'
```

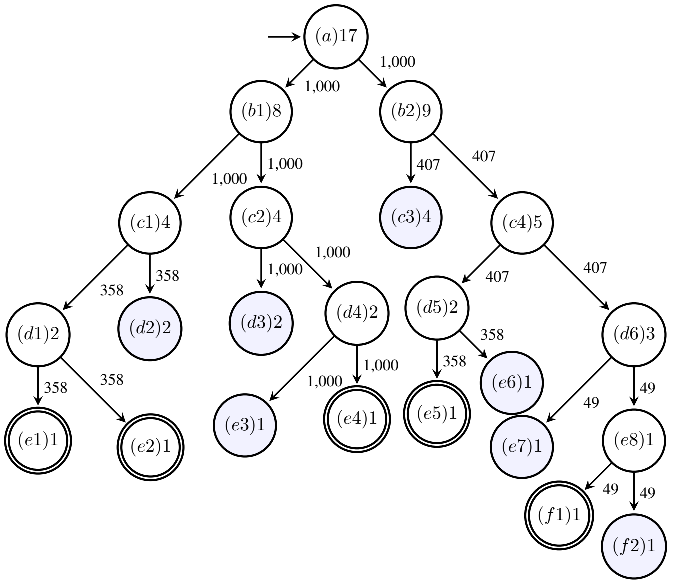

Fig. 7: Evaluation of supermutants. Each node represents a supermutant. The node values indicate number of mutations in each. The shaded nodes mark non-detected mutants. The edge labels show the number of tests that detected the supermutant and hence transmitted to children. The doubled border indicates first order mutants that were detected.

B. Analyzing the urlparse program

The *urlparse* API is a part of the *Python* standard library[8]. It contains 37 statements that can be mutated (We manually confirmed that the 37 mutants were non-equivalent) and are covered by our test suite. The mutant kill matrix contains $1,000 \times 37 = 37,000$ test case evaluations. This can be reduced to 25,260 test case evaluations using techniques such as coverage based test-case selection.

The API *urlparse* accepts different kinds of URLs, parses these and breaks them into their components. We used an automatic grammar based fuzzer for generating input URLs and used two test suites of varying strengths. The first test suite had the strongest oracle. This oracle verified that the result of unparsing the parsed URL matches the input URL. Using this test suite, the maximum coverage we obtained was 32% with 1,000 test inputs. On delta debugging the test suite, we found that a coverage of 32% could be obtained by just 4 test inputs. Using these 4 inputs and using traditional mutation analysis, we removed 20 trivially killed mutants where reachability was sufficient for detection. This required 37 mutant executions with one of the 4 different test cases. We verified that any reachable test case would be able to detect these mutants by checking that these mutants would be detected even after removing assertions from test cases. We next constructed a supermutant with the remaining 17 mutants. We evaluated our test cases against this supermutant, and passed the detecting test cases to the resulting child supermutants. Here, out of the

[8] https://docs.python.org/3/library/urllib.parse.html

17 mutations, 5 could be detected by supermutant analysis, and the remaining 12 could not be detected either as a single supermutant or individually by our test suite. This resulted in a total of 13,009 test evaluations. This compares to 25,260 test case evaluations for traditional analysis. The details are given in Figure 7. The mutation score was $\frac{25}{37} = 67.6\%$.

For the next cycle, we weakened our oracle. Rather than verifying that the unparsed output object exactly matched the input URL, we only verified that the return was a valid object (not *None*). Using this oracle, only one of the 5 mutants we previously detected could be found, and the supermutant containing all remaining mutations could not be detected by the test suite either. This resulted in a total of $9,037$ test case evaluations compared to 25,260. The mutation score was $\frac{21}{37} = 56.8\%$.

III. Discussion

As we showed in the two case studies, using the supermutant technique has the potential to reduce the computational overhead for evaluating the complete mutant kill matrix. A simple adaptation that we described in Section I can also make it effective for traditional mutation analysis.

The applicability of our technique is dependent on the following factors.

A. Strength of the test suite

The main requirement for applicability of the technique is that the generated test suites are strong (coverage adequate), but with weak oracles. We note that most machine generated test suites [11] tend to belong to this category. Our technique works best if individual test cases cover numerous mutants so that a number of these mutants can be evaluated in a single supermutant.

This is the main limitation of our approach. That is, it is applicable only to a specific type of test suite (either machine generated or containing weak oracles such that most mutants are not detected immediately). A second problem is that we assume that if a test case without assertion is able to detect the mutants it covers, then any other test case that covers them will also detect them (or other cheap means of classification). However, this may not hold for all test cases.

B. Strength of the mutant

Our technique works best with mutants that are stubborn or for evaluating possibly equivalent mutants. The reason is that, for these mutants, traditional mutation analysis has to evaluate the full matrix ($n \times t$), when compared to trivial mutants where the detection may happen with the first covering test case ($n \times 1$). Note that the statement deletion mutation operator that we have used in Section II-A and Section II-B requires smaller number of test cases [7], and hence easier to detect (and results in a smaller number of equivalent mutants) than other mutation operators. Hence our approach works with even relatively easier to detect mutants.

Indeed, we suggest that any mutant likely to be hard to detect or equivalent be marked, and evaluated exclusively with our algorithm rather than even checking whether they can be detectd by a simple covering test suite. Given that it is comparatively easy to classify mutants as weak or strong once reachability is taken into account [33] we believe that our approach will prove useful in reducing the cost of execution.

C. Overlapping mutations

A third limitation of our approach is that only non-overlapping mutations can be evaluated in a single supermutant. However, one can evaluate multiple supermutants each containing different non-overlapping mutations to work around this issue.

D. Effect of fault masking

The final limitation of our approach is that in the case of weak oracles that rely on exceptions, fault masking starts to become a problem. This can be illustrated by considering a weak oracle for *urlparse* that only checks for exceptions and assertion failures. Because *urlparse* relies on other procedures to perform actual work, replacing these procedure calls by *pass* removes any chance of exceptions, and hence mask the supermutants in these procedures. This is in some sense predicted by previous research [9] which suggests that fault masking is dependent on the co-domain of a function. In the case of a weak oracle that is unable to distinguish anything further than two cases of a function behavior– *exception* and *no exception*, the co-domain is essentially binary, and chances of fault masking increase correspondingly. Again, another observation from *urlparse* is that, if there are opportunities for incorrect parsing without throwing exceptions, these too can potentially produce masking supermutants when using weak oracles. These are problems that need to be solved before mutation analysis using supermutants can be practical.

IV. Threats to Validity

Our evaluation is subject to the following threats:

- Our case study was conducted on two simple programs *triangle*, and the *urlparse* the results from which are indicative of the potential of our technique, but are not representative of the real world.
- We rely on the rarity of fault masking from *theory of composite faults* to avoid redundant computation. While the theory has sufficient empirical evidence, it is possible that specific programs may have different characteristics.
- We assume that mutants that are not killed by a covering test case are likely hard to be killed. However, this assumption has not been empirically evaluated.
- We use a covering test-suite with weak oracle (without assertions) to eliminate trivial mutants. Here we assume that a mutant killed by a covering test case without assertions will be killed by any test case that covers it. However, this assumption may not be correct in every instance.
- We assume that supermutants especially when numerous mutants are applied are likely easier to kill, and effects of

fault masking is negligible in such cases. However, this assumption may not be correct in all cases.
- Some of the mutation patterns such as those that affect the control flow can lead to some spurious mutant kill matrix results. For example, a special optimization branch could be rendered non-executable by a mutation, which may likely make other test cases that test the correctness of the optimization also fail to detect the supermutant, while detecting the component mutations on the optimization.

V. Related Work

The idea of mutation analysis was proposed by Lipton [19] and its concepts formalized by DeMillo et al. [8], , and was first implemented by Budd [4]. Previous research [5], [21], [29] suggests that it subsumes different test coverage measures, including *statement*, *branch*, and *all-defs* dataflow coverage. Research shows that mutants are similar to real faults in terms of error trace produced [6], the ease of detection [2], [3], and effectiveness [16]. The foundational assumptions of mutation analysis—"the competent programmer hypothesis" and "the coupling effect"—have been validated both theoretically [9], [34], [35] and empirically [25], [26].

Researchers have suggested several approaches to reducing the cost of mutation analysis, which were categorized as *do smarter*, *do faster*, and *do fewer* by Offutt et al. [28]. The *do smarter* approaches include space-time trade-offs, weak mutation analysis, and parallelization of mutation analysis, and *do faster* approaches include mutant schema generation, code patching, and other methods to make the mutation analysis faster as a whole. Finally, the *do fewer* approaches try to reduce the number of mutants examined, and include selective mutation and mutant sampling.

While a number of researchers have investigated redundant mutants [17] and have used the complete mutant kill matrix for their research [1], [10], [24], we are unaware of any previous research that targets mutant kill matrix computation specifically.

A. Higher-order mutants

An important and related area of research is that of *higher-order* mutants [14], [15], [23]. The key idea here is that of *subsuming* higher order mutants, which are mutants that are harder to kill than their constituent mutants. These are mutants where there is at least a partial masking of effect of the first mutant by the second mutant. While such mutants are rare [9], they are important for the simple reason that these mutants represent the *hard to find* bugs that tend to interact, and hence represent a different class of bugs. While this research also investigates *higher-order* mutants, we rely on the rarity of subsuming mutants for ensuring that generated higher order mutants are easier to detect than its component mutants.

The mutant schemata approach [32] embeds multiple mutants in the same source file, and chooses the mutant to evaluate using global switches. Wang et al. [36] and Gopinath et al. [12] suggest removing execution redundancy between different mutants and hence making the mutation analysis faster. Papadakis et al. [30] also combines multiple mutants but for performing weak mutation.

VI. Future Research

Numerous modifications to our algorithm may be investigated for further improvements. We detail some of these below.

A. Partitioning strategy

While generating the supermutants, the question of how to partition a supermutant into two child mutants can have different answers. The simplest method is to randomly choose which mutations end up in each child. If one is interested in avoiding fault interaction, a plausible partitioning strategy is to choose mutants that are far apart, and unlikely to influence each other as possible within the same supermutant. If on the other hand, our focus is on generating stronger higher order mutants, or avoiding equivalent mutants, it may be better to partition the set of mutations by natural boundaries such as function, class, module etc. such that there is a better chance of interaction between mutations.

B. Order of execution

Currently, we generate a supermutant, and identify all the test cases that cover it. Instead, one could envision a tests-first approach where one identifies a particular test case, and identify all mutations that can be applied to the program elements covered by that test case. Depending on whether one is interested in mutant kill matrix or traditional mutation score, one can decide to eliminate the killed mutants from further consideration. It is as yet unclear whether this approach can yield better performance as compared to the approach considered in this paper.

C. Identifying probable fault masking patterns

One of the limitations with the supermutant approach is that of fault masking, especially for test suites with weak oracles. We identified that one of the recurring patterns was that of method calls that can be removed without producing an exception, and hence mask any exceptions that may be thrown from within the procedure call. Another pattern is that of optimization branches, where some optimization that does not affect the correctness of results is performed for specific kinds of inputs. Say a mutation changes the control flow such that the optimization branch is never taken. In such a case, a supermutant that contains mutations in the optimization branch will never be detected by test cases that check for the correctness of optimization even though the individual mutations can be detected separately. These may not be the only patterns, and a further empirical study using real world programs is necessary to identify the fault masking patterns that are likely to occur.

VII. Conclusions

Mutation analysis is used to accurately evaluate the quality of software test suites. Unfortunately, traditional mutation analysis is computationally intensive, requiring as many test suite evaluations to complete as there are possible mutations in the software, which often makes it impractical. The problem of heavy computational footprint is aggravated when one wishes to find the mutant kill matrix where the results of all test cases against all mutants are required, especially for evaluating redundancy in the mutants generated.

We propose a novel means of reducing the resource requirements of mutation analysis by combining multiple mutants into supermutants and evaluating them together. Our approach has the potential to reduce the resource requirements of mutation analysis especially for test suites with weak oracles — in particular machine generated test suites — where traditional mutation analysis is most expensive (because each mutant likely needs to be evaluated by each test case). Our approach can also yield strong higher order mutants that can improve the effectiveness of mutation analysis.

References

[1] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 21–30.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering*. IEEE, 2005, pp. 402–411.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[4] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1980, pp. 220–233.

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

[6] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1996, pp. 158–171.

[7] M. E. Delamaro, L. Deng, N. Li, V. Durelli, and J. Offutt, "Experimental evaluation of sdl and one-op mutation for c," in *International Conference on Software Testing, Verification and Validation*, Cleveland, Ohio, USA, 2014.

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

[9] R. Gopinath, , C. Jensen, and A. Groce, "The theory of composite faults," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2017.

[10] R. Gopinath, A. Alipour, A. Iftekhar, C. Jensen, and A. Groce, "Measuring effectiveness of mutant sets," in *Workshop on Mutation Analysis*. IEEE, 2016.

[11] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *International Conference on Software Engineering*. IEEE, 2014.

[12] ——, "Topsy-turvy: A smarter and faster parallelization of mutation analysis," in *International Conference on Software Engineering*. New York, NY, USA: ACM, 2016, pp. 740–743.

[13] C. Iida and S. Takada, "Reducing mutants with mutant killable precondition," in *International Conference on Software Testing, Verification and Validation Workshops*, March 2017, pp. 128–133.

[14] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008, pp. 249–258.

[15] ——, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, Oct. 2009.

[16] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 654–665.

[17] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 720–725.

[18] B. Lindstrom and M. Marki, "On strong mutation and subsuming mutants," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2016, pp. 112–121.

[19] R. J. Lipton, "Fault diagnosis of computer programs," Carnegie Mellon Univ., Tech. Rep., 1971.

[20] Y.-S. Ma and S.-W. Kim, "Mutation testing cost reduction by clustering overlapped mutants," *J. Syst. Softw.*, vol. 115, no. C, pp. 18–30, May 2016.

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

[22] G. J. Myers, "The art of software testing," *A Willy-Interscience Pub*, 1979.

[23] Q. V. Nguyen and L. Madeyski, "Problems of mutation testing and higher order mutation testing," in *Advanced Computational Methods for Knowledge Engineering*. Springer, 2014, pp. 157–172.

[24] R. Niedermayr, E. Juergens, and S. Wagner, "Will my tests tell me if i break this code?" in *International Workshop on Continuous Software Evolution and Delivery*. New York, NY, USA: ACM, 2016, pp. 23–29.

[25] A. J. Offutt, "The Coupling Effect : Fact or Fiction?" *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, Nov. 1989.

[26] ——, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[27] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337–344, 1994.

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

[29] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, Tech. Rep., 1996.

[30] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19, no. 4, pp. 691–723, Dec. 2011.

[31] D. Shin and D. Bae, "A theoretical framework for understanding mutation-based testing methods," in *International Conference on Software Testing, Verification and Validation*, 2016, pp. 299–308.

[32] R. H. Untch, J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," *ACM SIGSOFT Software Engineering Notes*, pp. 71:1–71:4, 1998.

[33] W. Visser, "What makes killing a mutant hard," in *IEEE/ACM Automated Software Engineering*. ACM, 2016, pp. 39–44.

[34] K. S. H. T. Wah, "A theoretical study of fault coupling," *Software Testing, Verification and Reliability*, vol. 10, no. 1, pp. 3–45, 2000.

[35] ——, "An analysis of the coupling effect I: single test data," *Science of Computer Programming*, vol. 48, no. 2, pp. 119–161, 2003.

[36] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2017, pp. 295–306.

[37] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," *International Conference on Software Engineering*, pp. 919–930, 2014.

[38] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.