Inducing Subtle Mutations with Program Repair

Florian Schwander  
Universität des Saarlandes  
Deutschland, Saarbrücken  
Email: platin91x@gmail.com

Rahul Gopinath  
CISPA Helmholtz Center  
for Information Security  
Email: rahul.gopinath@cispa.de

Andreas Zeller  
CISPA Helmholtz Center  
for Information Security  
Email: zeller@cispa.de

Abstract—Mutation analysis is the gold standard for assessing the effectiveness of a test suite to prevent bugs. It involves injecting syntactic changes in the program, generating variants (mutants) of the program under test, and checking whether the test suite detects the mutant. Practitioners often rely on these live mutants to decide what test cases to write for improving the test suite effectiveness.

While a majority of such syntactic changes result in semantic differences from the original, it is possible that such a change fails to induce a corresponding semantic change in the mutant. Such equivalent mutants can lead to wastage of manual effort.

We describe a novel technique that produces high-quality mutants while avoiding the generation of equivalent mutants for input processors. Our idea is to generate plausible, near correct inputs for the program, collect those rejected, and generate variants that accept these rejected strings. This technique allows us to provide an enhanced set of mutants along with newly generated test cases that kill them.

We evaluate our method on eight python programs and show that our technique can generate new mutants that are both interesting for the developer and guaranteed to be mortal.

I. INTRODUCTION

Mutation analysis is the gold standard for assessing test suite effectiveness. The first step in mutation analysis is to produce variants of the program using various mutation operators on the source code. These mutation operators typically change or replace syntactic elements of the program generating variants (mutants) from the original. The idea is that these syntactic changes will induce semantic (behavioral) changes in the variants.

Mutation analysis of a program’s test suite starts by verifying that the given test suite passes. That is, there exist no test cases that fail (also called a green test suite). The mutation analysis framework then generates mutants for the program and evaluates how many of the generated mutants fail to be verified using the test suite under evaluation. A test case kills a mutant if the test case executes successfully given the original program (i.e. it verifies the program), but fails when given the mutant. The mutation score of a test suite is defined as the fraction of mutants that failed to be verified relative to the total amount of generated mutants.

Testers often use live mutants for guidance toward identifying what tests to write [1] because live mutants represent possible failures that were not detected by the current test suite. Unfortunately, a significant number of such live mutants may not be of much use for the practitioner due to the so-called equivalent mutants [2], [3]. A mutant is called equivalent if the introduced syntactic change does not lead to a detectable semantic change. For example, consider the program (Listing 1) and its mutant (Listing 2).

```python
# Listing 1: Original Program

def swap(x, y):
    t = x
    x = y
    y = t
    return (x, y)

print(swap(1, 2))
```

```python
# Listing 2: Mutant Program

def swap(x, y):
    t = x
    x = y
    y = t
    return (x, t)

print(swap(1, 2))
```

As you can see, the induced syntactic change — that of swapping y for t — does not result in a corresponding semantic change in the behavior of swap(). That is, the mutant is equivalent to the original program in behavior. Since the presence of those equivalent mutants cannot be detected automatically except in trivial cases [4] they need to be identified and discarded manually. Such equivalent mutants also limit the practical applicability of mutation score as an effectiveness measure [1]. Unfortunately, manual inspection is neither scaleable [5] nor accurate. Researchers found that only about 80% [6] of mutants were classified correctly by manual inspection.

That is, if we are to recommend live mutants as guidance to the testers, we need a way to produce mutants with subtle faults that are guaranteed to be useful. In this paper, we provide a way to generate guaranteed mortal mutants that are useful for the practitioner. We focus on a large subset of programs — input processors — that have some concept of valid inputs and are able to reject inputs that are invalid.

A. Subtle Mutants Using Rejected Inputs

For ease of discussion, we focus on strings. Our technique can also be applied to other data types. Say we have a program that accepts input strings. Such programs typically use a parser
for parsing the input strings to their internal representation. If we have the corresponding input specification (typically a grammar) at hand, then we can use a fast grammar fuzzer [7] to generate numerous inputs, each of which are syntactically valid, and some of which are semantically valid. In case we do not have the input specification, we can also use failure feedback directed fuzzers [8], [9], [10] for generating valid inputs.

Given at least one such syntactically and semantically valid input, we can apply simple mutations such as bit-flip, byte-flip, trim, delete, insert, and swap to the input string. We focus on the minimum number of operations needed to make the string invalid. That is, the input string is closely related to the original string, but is rejected by the program. The idea is to collect such rejected inputs and to repair the program (focusing on the smallest repair needed) to make it accept the input while continuing to pass any test case in the given test suite. Given that the new variant (the repaired program) accepts an input that was rejected by the original program, we can guarantee that the variant is semantically different from the original — that is, the variant is non-equivalent\(^1\).

Consider the python program in Listing 3 and its test suite in Listing 4. This program only accepts input strings that exclusively contain digits. We now want to find the missing test cases for this program. For that, we wish to create mutants that pass all five tests while also introducing observable bugs. Say during fuzzing, we mutate this input using a random mutation — insert, and swap to the input string. We focus on the minimum number of operations needed to make the string invalid. That is, the input string is closely related to the original string, but is rejected by the program. The idea is to collect such rejected inputs and to repair the program (focusing on the smallest repair needed) to make it accept the input while continuing to pass any test case in the given test suite. Given that the new variant (the repaired program) accepts an input that was rejected by the original program, we can guarantee that the variant is semantically different from the original — that is, the variant is non-equivalent\(^1\).

Our technique for generating new mutants can also be used to generate test cases — the procedure is reminiscent of Evosuite [11]. The idea is to simply capture the way the original program reacts to the invalid input in a test case (i.e. return a result or throw an exception) and add it to the test suite.

### B. Augmenting Test Suites

Our technique for generating new mutants can also be used to generate test cases — the procedure is reminiscent of Evosuite [11]. The idea is to simply capture the way the original program reacts to the invalid input in a test case (i.e. return a result or throw an exception) and add it to the test suite.

---

\(^1\) We also call a non-equivalent variant a mortal variant.
We note that while we have only tested simple source
will now be rejected as invalid by the mutant. Hence, such
cosmic-ray representing faults that are not covered by
evaluation shows that our technique can produce mutants
augment the mutant set generated by a state-of-the-art mutation
tools. Hence, we evaluate whether our technique can
used for complementary
designed to be
repair technique can yield effective mutants. Our algorithm is
concrete valid input in the original program.

The key idea of our algorithm is that a concrete invalid input
in a mutant has to follow the same conditional branches as a
original, the valid input returns no error while the invalid
input returns an exception. For the mutant, neither the valid
nor the invalid input produce an error. This yields the test case
reverse_int(‘12-’) killing the mutant. In the case
of the second mutant (Listing 6), this was added because the
mutant rejected 0 which was accepted by the original. Indeed,
any input that contains 0 that was accepted by the original
will now be rejected as invalid by the mutant. Hence, such
inputs become new test cases to be added to the test suite.

C. Algorithm

The key idea of our algorithm is that a concrete invalid input
in a mutant has to follow the same conditional branches as a
concrete valid input in the original program.

• We start with a valid input that is accepted by the program.
• We execute the original program with the valid input and
record the truth values of the conditions encountered. We
call this the valid execution.
• Next, we apply simple mutations to the input-producing
inputs that are rejected by the program. We choose inputs
that have high overlap in execution path with the original.
• Next, we record the truth values of the conditions
encountered when processing the invalid input.
• From these records, we compute the source code locations
of conditions that are present in both but differ on the
observed truth value.
• For such pairs of source code locations, we generate
multiple mutants by applying some randomly chosen
mutation operators to the location(s).
• We then test the behavior of all generated mutants when
processing the valid and invalid input respectively and
keep those that reject the valid input or accept the invalid
input.
• For mutants that do not accept the invalid input we re-
compute the difference in observed truth values to the
valid execution and continue as before.
• We stop generating successors of a mutant if the difference
is empty or it accepts the invalid input.

We note that while we have only tested simple source
mutations using mutation analysis operators, any program
repair technique can yield effective mutants. Our algorithm is
designed to be complementary to traditional mutation analysis,
used for augmenting the original mutant set generated by
such tools. Hence, we evaluate whether our technique can
augment the mutant set generated by a state-of-the-art mutation
framework called cosmic-ray[12] on four JSON parsers. Our
evaluation shows that our technique can produce mutants
representing faults that are not covered by cosmic-ray.

Contributions:
• We leverage program repair to generate subtle mutants
that are guaranteed to be non-equivalent.

def H2I(inpt):
    res = 0
    if not inpt:
        raise Exception(‘Not Hex’)
    for c in input.lower():
        if c >= ’0’ and c <= ’9’:
            res = 16 * res + (ord(c)-ord(’0’))
        elif c >= ’a’ and c <= ’f’:
            res = 16 * res + (ord(c)-ord(’a’)+10)
        else:
            raise Exception(‘Not Hex’)
    return res

Listing 7: Hex Converter Test Suite

• We evaluate and show that our technique can augment
the quality of mutants of the state-of-the-art mutation tool
cosmic-ray on four large real-world subjects.

II. INDUCING SUBTLE MUTANTS

For inducing subtle mutants, our strategy involves first
finding some string that is accepted by the given program. Next,
we use simple mutation operators to generate syntactically and
semantically close input strings that are however rejected by the
program. Then we attempt to make the program variant
accept the invalid input string by repairing the program. We generate pairs — (valid-input, invalid-input) — of
such inputs, generating a new pair when either the invalid input
is accepted, or when there are no more simple modifications
2 to be made to the source code. During this process, we also
keep those mutants that reject a designated valid string as they
are also guaranteed non-equivalent to the original program.

Our approach is split into these steps: (1) generating valid
inputs, (2) generating invalid inputs from the valid input, and
(3) repair the program such that the invalid input is accepted.

A. Generating Valid Inputs

For generating inputs, we have a few choices. The essential
idea is to use a fuzzer that can cover the input space fast. If
we have the input specification of the program in question,
we can use a fast grammar fuzzer [7] to quickly generate
syntactically valid inputs and hence, have a good chance of
creating semantically valid inputs if the program at hand
contains semantic validation steps. If on the other hand, we do
not have the input specification, we can use a failure feedback

2 We prefer simple modifications as they are more plausible as a fault in
the program than more complex modifications, and hence more likely to be
found useful by the practitioners.
directed fuzzer [8], [9], [10] for quickly covering the input space, and hence producing a valid input string.

B. Generating Invalid Inputs from the Valid Input

To get invalid inputs, we modify each valid string by applying any of the mutation operators (bit-flip, byte-flip, trim, delete, insert, and swap) randomly, generating mutated input strings. These are then fed to the program to determine if they are rejected. If the string is accepted, it is added to the set of valid strings. If the string is rejected, it is added to the set of invalid strings. When choosing which inputs to modify, we prefer those that have been mutated least often. This ensures that our mutants are within a minimal edit distance [13] from the original. We stop when the budget for allowed mutation attempts is exhausted.

C. Crafting Mutants by Repairing the Program

For crafting mutants, we first identify the candidates for mutation. We only consider conditionals statements as candidates for mutation as these are expected to determine whether a program accepts or rejects an input. Our approach considers the difference in observed evaluated condition state — True or False — to find condition candidates to mutate. We then compute candidate lines for modification in each step for the current (valid-input, invalid-input) pair. The algorithm terminates when the mutation of every pair is completed.

As an example of a possible execution, consider the hex converter (Listing 7) with inputs 19 and 1+9. For the valid input 19, the condition states (these are evaluated in the context of the unmodified SUT) are \{3: False; 6: True\}. For the invalid input 1+9, we get \{3: False; 6: True, False, 8: False\} We call these the condition traces.

Unlike the valid input, the trace of the invalid input is recomputed whenever a new mutant is created. Both agree on the state of line 3 and the valid execution does not encounter line 8 which leaves line 6 as a possible candidate for mutation. Indeed, as its value is fixed for the valid string but varies for the invalid input, line 6 is the only mutation candidate in the current step. Since Mauris (our tool) can generate a large number of mutants (see Table I for the list of mutation operators) for this line, we can’t show them all. Thus we will limit ourselves to one first (Listing 9), one second (Listing 10), and one third-order (Listing 11) candidate to illustrate the procedure. We then pick the first order mutant and first check whether it rejects the valid string 19. Since it does, the current mutant is mortal and can be kept.

Next, we check the invalid string 1+9 to see that it is still being rejected. In the context of the current mutant, its condition trace is \{3: False; 6: False; 8: False\}. This is the same trace as before and to avoid infinite recursion we do not allow a line to be modified more than once. This leaves us with no candidate line which concludes the mutation of our first order mutant. As before, we start analyzing the current state of our second order mutant by checking whether the valid string 19 gets rejected. This is however not the case so this mutant will only be part of the output in case it accepts the invalid string 1+9, which it does. We hence keep our second-order mutant and conclude its mutation.

Finally, we consider our third order mutant again by first checking valid string 19 which it rejects. Again this means that the mutant is non-equivalent and will be part of the output. The mutant also does not accept the invalid string 1+9 and we hence again compute candidate lines. The condition trace is unsurprisingly: \{3: False; 6: False; 8: False\}. As before the only candidate line has been mutated already so we do not create any successors. Note that the presented second-order mutant passes the example test suite.

```python
def H2I(input):
    res = 0
    if not input:
        raise Exception('Not Hex')
    for c in input.lower():
        if not (c >= '0' and c <= '9'):
            res = 16 * res + (ord(c)-ord('0'))
        elif c >= 'a' and c <= 'f':
            res = 16 * res + (ord(c)-ord('a')+10)
        else:
            raise Exception('Not Hex')
    return res
```

Listing 9: First Order Mutant

```python
def H2I(input):
    res = 0
    if not input:
        raise Exception('Not Hex')
    for c in input.lower():
        if not (c >= '0' and c <= '9'):
            res = 16 * res + (ord(c)-ord('0'))
        elif c >= 'a' and c <= 'f':
            res = 16 * res + (ord(c)-ord('a')+10)
        else:
            raise Exception('Not Hex')
    return res
```

Listing 10: Second Order Mutant

```python
def H2I(input):
    res = 0
    if not input:
        raise Exception('Not Hex')
    for c in input.lower():
        if c <= '0' or c <= '9':
            res = 16 * res + (ord(c)-ord('0'))
        elif c >= 'a' and c <= 'f':
            res = 16 * res + (ord(c)-ord('a')+10)
        else:
            raise Exception('Not Hex')
    return res
```

Listing 11: Third Order Mutant
TABLE I: Mutation operations

<table>
<thead>
<tr>
<th>AST elements (class)</th>
<th>Replace with</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any AST element e</td>
<td>not e</td>
<td>x → not x</td>
</tr>
<tr>
<td>Any AST element with multiple components</td>
<td>all possible combinations of negation</td>
<td>x and y</td>
</tr>
<tr>
<td>+, -, *, /, %, **, &lt;&lt;, &gt;=, &lt;=, &amp;</td>
<td>other operand of this class</td>
<td>1+2 → 1+2</td>
</tr>
<tr>
<td>constant n</td>
<td>constant between ¬abs(n)−1 and abs(n)+1</td>
<td>4 → −2</td>
</tr>
<tr>
<td>and, or, !=, !~, &lt;=, &gt;=, ≠, is, in, is not in, not in</td>
<td>other operand of this class</td>
<td>a in b → a !~ b</td>
</tr>
<tr>
<td>∼, ¬ (unary), + (unary)</td>
<td>other operand of this class</td>
<td>x → −x</td>
</tr>
<tr>
<td>e[1:3] (slice)</td>
<td>empty index</td>
<td>a[1:3] → a[:].</td>
</tr>
</tbody>
</table>

For mutants that do not accept the invalid input we recompute the difference in observed truth values to the valid execution and continue as before. We stop generating successors of a mutant if the difference is empty or it accepts the invalid input.

III. Evaluation

For evaluation, we chose the following test subjects:

- cgi.py: A simple CGI decoder (72 SLOC)
- mathexpr.py: A script that evaluates simple mathematical expressions (169 SLOC)
- urljava.py: A simple URL parser (230 SLOC)
- xsum.py: a script which computes the cross-sum of a given integer (13 SLOC). This served as our basic test subject since a full run on this subject is fairly easy to fully comprehend.
- ijson.py[14]: An iterative JSON parser (309 SLOC)
- microjson.py[15]: A minimal JSON parser (311 SLOC)
- nayajson.py[16]: A fast streaming JSON parser (546 SLOC)
- simplejson.py[17]: A simple, fast, extensible JSON parser (1486 SLOC)

We obtained cgi.py, mathexpr.py, urljava.py, and xsum.py from the Pychains [18] examples. For these subjects, there were no existing test suites. Hence, we added them ourselves.

A. Verifying Mutant Behavior

To verify that the mutants we generated are guaranteed non-equivalent, we infer the actual behavior when passing the used inputs via the command line. This is to make sure that the way we inject the inputs has no influence on the execution.

1) Mutant mortality: The first question concerns whether the mutants we produce are actually mortal or not.

RQ1: Are the mutants created non-equivalent?

That is, are the mutants produced using our technique semantically different from the original? In particular, do these generated mutants show different behavior from the original by accepting some originally rejected strings, or rejecting some originally accepted strings?

2) Unique faults: We aim to generate strong mutants that represent unique faults. Hence, we need to assess the quality of the generated mutant set. The next research question tackles whether the generated mutants improve the quality of the mutant set.

RQ2: Do the generated mutants encode new unique faults?

In mutation analysis, the theoretical minimal set of mutants [19] corresponding to an original set of mutants is defined as the smallest set of mutants such that a test suite that can kill every mutant in that minimal set is guaranteed to be capable of killing every mutant in the original set. That is, every semantic fault in the original mutant set is represented uniquely in the minimal set. That is, the size of the minimal set of mutants captures the number of unique faults that the mutants can produce. Given that one often does not have access to a fine-grained adequate test set that can compute the theoretical minimal set of mutants, we make do with computing the minimal set of mutants that correspond to a given test suite. The size of such a minimal set of mutants is used as a measure of how many different behaviors a mutant set shows in the context of a given test suite. Hence, this question is evaluated by comparing the size of the minimal set to the size of the minimal set. That is, the size of the minimal set. Hence, we need to assess the quality of the minimal set. That is, the size of the minimal set.

We aim to generate strong mutants that represent unique faults. Hence, we need to assess the quality of the generated mutant set. The next research question tackles whether the generated mutants improve the quality of the mutant set.

RQ2: Do the generated mutants encode new unique faults?

In mutation analysis, the theoretical minimal set of mutants [19] corresponding to an original set of mutants is defined as the smallest set of mutants such that a test suite that can kill every mutant in that minimal set is guaranteed to be capable of killing every mutant in the original set. That is, every semantic fault in the original mutant set is represented uniquely in the minimal set. That is, the size of the minimal set of mutants captures the number of unique faults that the mutants can produce. Given that one often does not have access to a fine-grained adequate test set that can compute the theoretical minimal set of mutants, we make do with computing the minimal set of mutants that correspond to a given test suite. The size of such a minimal set of mutants is used as a measure of how many different behaviors a mutant set shows in the context of a given test suite. Hence, this question is evaluated by comparing the size of the minimal set to the size of the minimal set. That is, the size of the minimal set. Hence, we need to assess the quality of the minimal set. That is, the size of the minimal set.

3) Are the crafted mutants useful?: We need to ensure that our mutants are useful for the programmer. In particular, we need to be wary of mutants that are irrelevant to the programmer. For example, a plausible mutant could be an inserted check such as if i == 42: raise Error() for a function that takes i as a parameter. The problem with such mutants is that they require overly specific tests to detect, and are not representative of actual faults. That is, we need to make sure that our generated mutants do not contain irrelevant mutants that require overly specific conditions to manifest.

RQ3: Are the subtle mutants generated plausible?

For analyzing this, we looked at each mutant and checked whether they are plausible. That is, whether they are representative of real faults that a programmer can make in the real world. In particular, we checked whether this mutant would be a mutant that a programmer would likely write a test case to detect. Since there is a subjective metric, it requires human
analysis. Hence, we analyzed each generated mutant manually. The following procedure was used to judge whether the given mutant was plausible or not:

- Produce a `diff` of the original and mutant program.
- Simplify the condition as much as possible.
- Consider what kind of test case would kill the mutant.
- Judge whether the emerging test case was reasonable enough to add to the test suite. That is, it was not overly specific.

### B. Execution Details

For the evaluation, we used a Windows 10 64-bit system running an Intel Xeon E3-1230 V2 (stock settings) and 16GB of DDR3 RAM. The time budget for valid string generation was set to 40 minutes. For the four JSON subjects, we generated valid inputs for `microjson` using pychains and filtered them to match the respective subject. The time budget was chosen such that mutation and full verification of `simplejson`, the most costly subject with regards to time, did not exceed 24 hours. We used these settings to execute `Mauris` five times per subject. This was found to be enough for the coverage induced by the generated inputs to become asymptotic. Since our comparison is based on mortal mutants which cannot be guaranteed for cosmic-ray live mutants, we considered them equivalent. These were removed before comparison. For cosmic-ray (v5.4.0) we used default settings which yielded all available mutants.

### C. Results

Table II contains details of our subjects. The terms used are as follows: `Conditions` are the amount of `if` and `elif` statements in the code. `Condition/SLOC` is the relative frequency of conditions in the source code. `Coverage (Test Suite)` is the branch or statement coverage measured with `coverage.py` [20] when running the test suite. `Coverage (Mauris Inputs)` is the branch or statement coverage measured with `coverage.py` [20] when using the union of inputs of all `Mauris` executions of the given subject.

The results for the subjects can be found in corresponding tables: `nayajson` is given in Table III, `simplejson` is given in Table IV, `json` is given in Table V, `microjson` is given in Table VI, `cgi` in Table VII, `mathexpr` in Table VIII, `urljava` in Table IX, and `xsum` in Table X.

The following are the columns in the table: `M` is the time in minutes generating the mutants took. For `Mauris` this consists of generating inputs, generating candidate mutants, executing the test suite, and checking the behavior of the live mutants. `Minimal` is the amount of mutants in the minimal mutant set. `Live` is the number of mortal mutants that passed the test suite. `Subtle` is the percentage of live `Mauris` mutants considered subtle.

The following are the rows in the table: `cosmic-ray` contains results for cosmic-ray. `Mauris + cosmic-ray: Average` contains the average results of all five test runs again including cosmic-ray. `Mauris + cosmic-ray: Best` contains the results for the test run with the minimal set of mutants of the union of `Mauris` and cosmic-ray being the biggest.

---

**IV. Discussion**

We discuss the results of our experiments next.

### A. Mutant mortality

Our first research question was about whether our guarantee of non-equivalence was validated in practice.

**RQ1:** Are the mutants created non-equivalent?

The results show that our strategy can indeed produce mutants that are mortal. This was verified by performing behavior validation on live mutants.

### B. Unique faults

For our second research question, we evaluated whether the mutant set that was produced from cosmic-ray and augmented with `Mauris` was more effective than the baseline set of mutants from cosmic-ray.

**RQ2:** Do the generated mutants encode new unique faults?

We observed that between 10.76% (urljava, Table IX) and 80.53% (microjson, Table VI) of mutants are unique to `Mauris`. Hence we conclude that `Mauris` can indeed find mutants that encode new unique faults and hence augment the baseline mutant set of cosmic-ray.

### C. Are crafted mutants useful?

As our mutants are often of higher order we had to check whether they can be useful for a programmer. We would, for example, consider a mutant that alters a number processor (Listing 12) to not accept input 42 (Listing 13) irrelevant because even though it represents a fault that the test case likely did not check for, it is not a plausible fault that can be produced by a competent programmer.

```python
if len(inpt) > 0:
    print('ok')
else: raise Exception()

Listing 12: A number processor
```

```python
if len(inpt) > 0 and inpt != '42':
    print('ok')
else: raise Exception()

Listing 13: An irrelevant mutant
```

**RQ3:** Are the subtle mutants generated plausible?

Our analysis of whether the observed modifications in source code are plausible mistakes that a programmer may make shows that across all subjects, 84.38% of live mutants were relevant, with `CGI` having the lowest share with just over 51% (Table VII). We thus conclude that most (over 84%) live mutants created by `Mauris` are useful to test against.
TABLE II: Coverage and Code measures

<table>
<thead>
<tr>
<th></th>
<th>Statement (Tests)</th>
<th>Statement (Mauris Inputs)</th>
<th>Branch (Tests)</th>
<th>Branch (Mauris Inputs)</th>
<th>Conditions</th>
<th>SLOC</th>
<th>Condition/SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nayajson</td>
<td>88.00%</td>
<td>62.00%</td>
<td>85.00%</td>
<td>85.00%</td>
<td>162</td>
<td>546</td>
<td>29.67%</td>
</tr>
<tr>
<td>Simplejson</td>
<td>94.00%</td>
<td>39.00%</td>
<td>92.00%</td>
<td>92.00%</td>
<td>22</td>
<td>1486</td>
<td>13.12%</td>
</tr>
<tr>
<td>Json</td>
<td>99.00%</td>
<td>89.00%</td>
<td>98.00%</td>
<td>98.00%</td>
<td>23</td>
<td>309</td>
<td>7.44%</td>
</tr>
<tr>
<td>Microjson</td>
<td>95.00%</td>
<td>93.00%</td>
<td>94.00%</td>
<td>94.00%</td>
<td>37</td>
<td>311</td>
<td>11.90%</td>
</tr>
<tr>
<td>CGI</td>
<td>98.00%</td>
<td>97.00%</td>
<td>96.00%</td>
<td>96.00%</td>
<td>9</td>
<td>72</td>
<td>12.50%</td>
</tr>
<tr>
<td>Mathexpr</td>
<td>93.00%</td>
<td>94.00%</td>
<td>90.00%</td>
<td>91.00%</td>
<td>21</td>
<td>169</td>
<td>12.43%</td>
</tr>
<tr>
<td>Urljava</td>
<td>89.00%</td>
<td>69.00%</td>
<td>86.00%</td>
<td>64.00%</td>
<td>40</td>
<td>230</td>
<td>17.39%</td>
</tr>
<tr>
<td>Xsum</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>1</td>
<td>13</td>
<td>7.69%</td>
</tr>
</tbody>
</table>

D. The quality of mutants crafted

By design, four factors directly influence the quality of mutants created by Mauris:

- The quality of used input strings (i.e., their branch/statement coverage): Only conditions observed during execution are modification candidates.
- The density and structure (number of modifiable tokens) of conditions: Mauris exclusively changes if and elif statements. Each modifiable token yields multiple candidate mutants.
- The structure of its test suite: Test cases for inputs used by Mauris can kill the corresponding mutants. Consider a mutant that accepts invalid input ‘xyz’ for a number processor. If the test suite has a test case which checks that ‘xyz’ is rejected the mutant will be killed. Tests that require certain exception names for a given input work similarly.

For the four JSON subjects (nayajson III, simplejson IV, ijson V, and microjson VI) the first two reasons can explain the observed results. On the subject with the lowest condition density (ijson, 7.44%) and the one with the worst inputs (simplejson, 39% statement coverage) respectively (Table II) Mauris performed worse than on the other two. Given that nayajson has the highest condition density (Table II) of all subjects the number of initially created mutants being high is expected as well. The fourth and final subject in this class is microjson for which Mauris generated inputs with coverage close to the test suite’s (93% and 95% statement coverage respectively, Table II) producing overall good results.

Regarding the four Non-JSON subjects CGI and mathexpr have both a sufficient condition density II (both above microjson’s 11.9%) and very strong inputs making Mauris produce good results. It is however surprising how for urljava with both a fairly high condition density (17.39%) and fairly good inputs (69% statement coverage) (Table II) our results were not as good as for the other two. Looking into the details however we noticed that the test cases are similar to the inputs used by Mauris. In effect, the corresponding Test Suite is particularly good at killing those mutants.

V. RELATED WORK

When implementing a piece of software it is in the best interest of all parties that the program is correct. That is, it has few bugs. Test suites are a commonly used tool for that, making a means to evaluate their effectiveness invaluable.

An early approach to evaluating test suites was debugging[21], which required a programmer to artificially insert bugs into the program to check whether the tests can find them. Since this does, however, require significant human effort DeMillo et al. introduced an alternative approach called mutation testing [22]. It is based on two major hypotheses from his previous work [23], the competent programmer hypothesis and the coupling effect. The competent programmer hypothesis (also called the finite neighborhood hypothesis) states that programmers tend to create near-correct versions of a program that is just a couple of tokens away from being correct. The coupling effect [23] states that complex faults mostly arise from the interaction between multiple small faults. This implies that test cases that find simple faults are also capable of finding

\[3\] also called fault seeding or fault injection
more complex faults [24], [25]. For mutation analysis these small errors usually take the form of a so-called first-order mutant (FOM), i.e. a single token was modified (mutated). In case a mutant is created by modifying two or more tokens at once it is called a higher-order mutant (HOM).

Mutants are classified according to their quality as equivalent, redundant, trivial, and stubborn. Redundant mutants are mutants that are easier to detect than some other mutant (or in other words, the faults represented by the redundant mutants are already represented by other mutants). According to Rice’s theorem [26], it is not possible to find which mutants are redundant. Trivial and stubborn mutants are opposites. Trivial mutants can be detected by just about every test case, whereas stubborn mutants can not be detected by most test cases. Stubborn mutants are the ones that we are most interested in. Finally, there are also the equivalent mutants, which act exactly like the original program does and can hence never be detected.

The number of equivalent mutants is dependant on a given program’s unique characteristics, which makes statistical estimation infeasible [27], [28]. The presence of equivalent mutants artificially deflates mutation scores, and hence, makes mutation score unreliable as a measure of test suite quality.

Since they are a big problem for many mutation techniques there has been a significant amount of research on the topic and many publications dedicated to them. One of them is the work of Gruen et al. [29] who explored how big of a problem equivalent mutants are. They concluded that equivalent mutants can make up a significant portion of the overall mutants (up to 40% in their example) making trying to get rid of them a worthwhile effort.

One way to achieve this is by trying to avoid creating them in the first place. This was the method of choice for Kaminski et al. who tried to select mutation operators that do not create equivalent mutants [30]. Another common approach is trying to remove them by coming up with a heuristic, such as the effort by Baldwin and Sayward [31] which introduced various means to check for likely equivalent mutants. Similarly, Offutt et al. [32] tested using some compiler optimization techniques and feasible paths [33] as a heuristic to eliminate equivalent mutants with partial success. Another compiler optimization technique - program slicing - was the tool of choice for Hierons et al. [34] and Kintis et al. [35]. The last approach in this category that we want to mention is by Schuler et al. [36] who used program invariants to achieve a low equivalent mutant rate. A somewhat different idea was pursued by Adamopoulos et al. [37] who used evolutionary algorithms to generate better mutants. Yet another way is to get a non-equivalence guarantee by sacrificing generality. A relevant example for this is by Ji et al. [38] who introduced a way to detect all equivalent mutants related to exception blocks.

Similarly, we guarantee non-equivalent mutants by requiring a specific input domain (programs that take string inputs) and a specially tailored mutation operator (modifying if statements based on some invalid input string). Regarding mutation in python, we borrowed some knowledge on which operators to consider from Derezińska et al. [39] and adjusted and augmented them for usage in if-conditions. While pursuing a different goal the work of Shahriar et al. [40] is somewhat similar to our work as it proposes mutation operators to specifically meet certain conditions. The same holds for the efforts of Just et al. [41] who walked a path similar to ours but used test cases instead of actual inputs and inference instead of execution.

Even though the idea of creating test cases using mutation is somewhat different from the topic of this work, we still want to mention the paper of Papadakis et al. [42] as crafting tests that kill each generated mutant from Mauris output is possible. Efforts to link mutation analysis and program repair are not all new as both require strategies to deal with their inherently large complexity and the implied stress on the available resources such as memory and time. Common techniques to tackle these issues involve not executing the program directly and instead analyzing it using e.g. symbolic execution as suggest by Wong et al. [43] among others. Approaches as by Ali Ghanbari et. al.[44] which is lightweight and focuses on generality by using simple mutation operations at Java bytecode level to generate patch candidates4 show that the two topics pair well. Rothenberg et. al.[45] also used mutation and additionally combined SAT and SMT solvers to limit the number of modifications performed. Taking into account the findings of Timperley et. al.[46] which suggest that patch

---

TABLE VII: Results - cgi

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>Minimal</th>
<th>Live</th>
<th>Subtle</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosmic-ray</td>
<td>3</td>
<td>11</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Mauris + cosmic-ray: Average</td>
<td>62</td>
<td>27</td>
<td>16</td>
<td>51.25%</td>
</tr>
<tr>
<td>Mauris + cosmic-ray: Best</td>
<td>61</td>
<td>51</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

TABLE VIII: Results - mathexpr

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>Minimal</th>
<th>Live</th>
<th>Subtle</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosmic-ray</td>
<td>3</td>
<td>37</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Mauris + cosmic-ray: Average</td>
<td>65</td>
<td>102</td>
<td>18.2</td>
<td>86.81%</td>
</tr>
<tr>
<td>Mauris + cosmic-ray: Best</td>
<td>56</td>
<td>116</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

TABLE IX: Results - urljava

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>Minimal</th>
<th>Live</th>
<th>Subtle</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosmic-ray</td>
<td>7</td>
<td>100</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Mauris + cosmic-ray: Average</td>
<td>45</td>
<td>112.2</td>
<td>5.4</td>
<td>88.89%</td>
</tr>
<tr>
<td>Mauris + cosmic-ray: Best</td>
<td>43</td>
<td>119</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

TABLE X: Results - xsum

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>Minimal</th>
<th>Live</th>
<th>Subtle</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosmic-ray</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Mauris + cosmic-ray: Average</td>
<td>41</td>
<td>5</td>
<td>1</td>
<td>100.00%</td>
</tr>
<tr>
<td>Mauris + cosmic-ray: Best</td>
<td>41</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

---

4Automatically generated program variants that fix a given bug
candidates induced by mutations modify parts of the code that looks alien to human programmers reinforced our conviction to stick to modifying conditional statements only.

Since the number of mutants we can create is very hard to estimate, we refer to the papers by Jia et al. [47] and Harman et al. [48] which suggest that our results are relevant even if the number of mutants is small. This is due to us not limiting their mutation order which allows higher-order mutants.

Finally, we also need to mention the mechanism we used to compare the quality of mutants by Mauris to the baseline. As stated in the introduction we used the minimal mutant set approach introduced by Ammann et al.[19] which suggests condensing a mutant set to mutants causing unique faults.

VI. LIMITATIONS AND FUTURE WORK

Currently, our approach is restricted to programs that accept input and have some concept of valid and invalid input. However, the approach of inducing subtle mutations with program repair does not need to be restricted to only input processors. One can also think of mutating test cases themselves. For example, given a test case, one can mutate it until it fails for the program under test, then repair the program until it passes. This mutation can take place anywhere in the test case, including the order of API sequence called, the content of the API calls, or even the oracle. That is, our approach has the potential of being applicable across a much wider variety of programs. This would be one of the major focuses for us going forward.

Another limitation of our approach is the cost of exploration. Each valid input is generated after expending some computational cost in fuzzing, and each invalid input is generated again after expending further effort. Next, each repair needs to be generated and evaluated. All this can mean that depending on the program under test, the cost of mutant generation can be high. Hence, reducing the cost of finding repairs would be one of our major considerations going forward.

Finally, we have used trivial mutations for the most part because they are easy to generate, and are more plausible than more complex mutations. However, plausible complex mutations can exist, which may lead to more subtle mutations, and hence more effective mutants. Exploring this avenue would again be one of our priorities for the future.

VII. THREATS TO VALIDITY

A. Threats to External Validity

Threats to external validity are factors that may reduce the generalizability of our findings. Since our approach is novel the parameters, as well as used subjects, had to be chosen by us. This might have induced threats to external validity caused by subject amount, subject type, test suites, and amount of test repetitions per subject. That is, results obtained by testing eight subjects may not generalize well. Furthermore, the sizable amount of JSON-processors (50% of subjects) and simpleness of the programs pose threats as well. So does the fact that we had to write half of the test suites ourselves (for subjects that had none). Additionally testing each subject five times may not generalize well as having encountered a statistical fluke is possible.

B. Threats to Internal Validity

Threats to internal validity are factors concerning the judgment of cause and effect relationships. Misjudging the actual order of a mutant poses such a threat as does our subjective judgment of mutant relevance. Furthermore, the evaluation of mutant relevance was carried out by a single human which may require further research.

VIII. CONCLUSION

Testers often use live mutants as targets when writing new test cases. However, the presence of equivalent mutants reduces the effectiveness of this technique, resulting in wasted manual labor. Hence, we need subtle mutants that encode likely faults in the program that are guaranteed to be mortal — i.e non-equivalent.

We showed how to use program repair techniques to obtain subtle mutants that are guaranteed to be non-equivalent, and relevant to the programmer. We used a fuzzer to generate inputs for the program under test and identified the conditions that behaved differently for valid and invalid inputs. The source locations of these identified conditions were repaired to accept the invalid inputs, resulting in subtle mutants.

We found that more than 84% of the mutants produced were relevant for programmers in that they were plausible and not overly specific. The created mutants are guaranteed to be detectable, and hence, useful for software practitioners to target to kill when writing (or automatically generating) new test cases.

REFERENCES


