

ST-Project

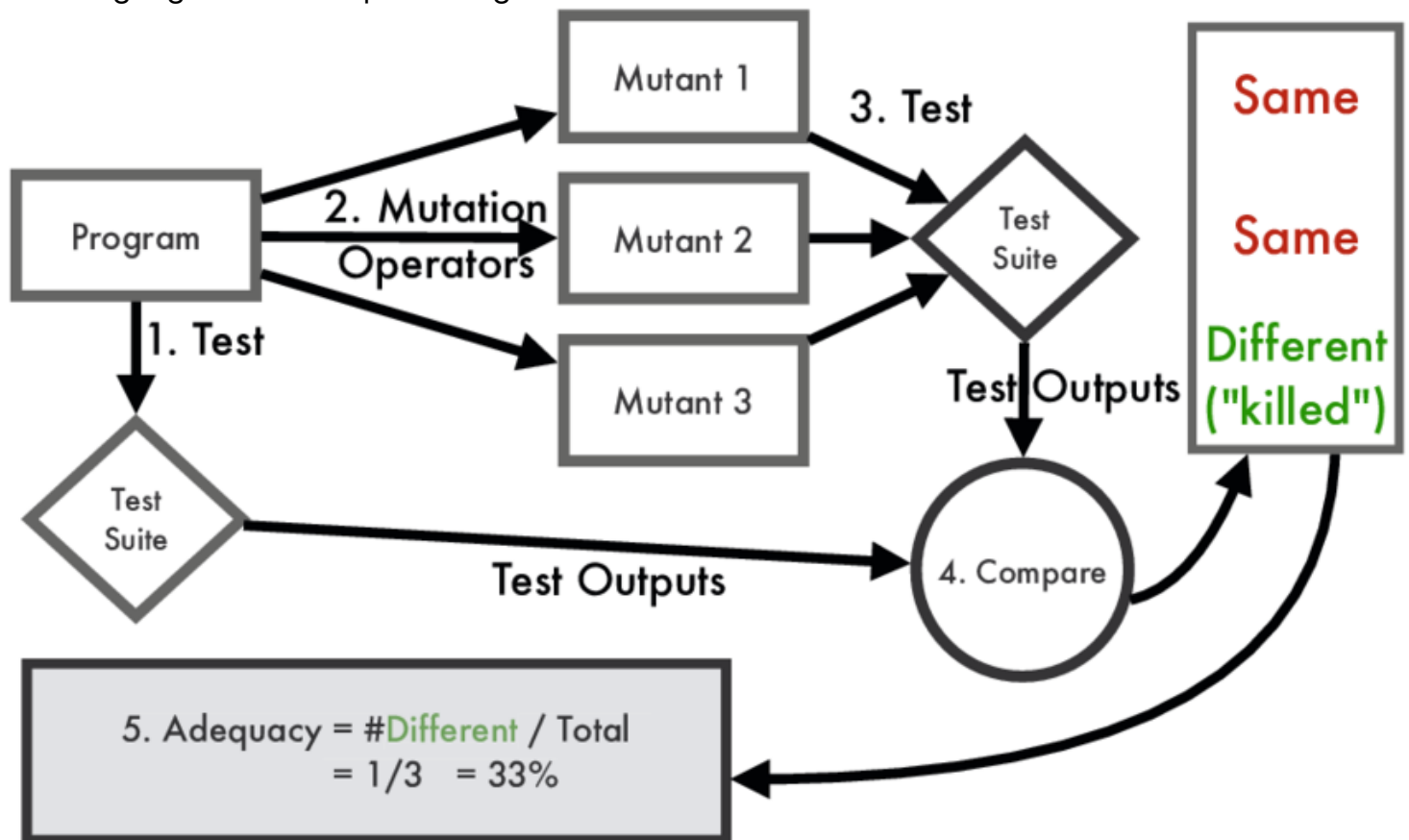
Contributors

- Vidhish Trivedi (IMT2021055): [GitHub \(Vidhish Trivedi\)](#).
- Rohit Shah (IMT2021027): [GitHub \(Rohit Shah\)](#).

What is Mutation Testing?

Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests.

Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program. Faults (or mutations) are automatically seeded into your code, then your tests are run. If your tests fail then the mutation is killed, if your tests pass then the mutation lived. The quality of your tests can be gauged from the percentage of mutations killed.



Why do we need Mutation Testing?

Traditional test coverage (i.e. line, statement, branch, etc.) measures only which code is executed by your tests. It does not check that your tests are actually able to detect faults in the executed code. It is therefore only able to identify code that is definitely not tested.

Libraries and Frameworks Used

- **JavaScript:** Stryker

- Stryker started as a pure JavaScript mutation testing framework, which is why it was referred to as **Stryker**, but it has since been rebranded to **StrykerJS**.
- Today, StrykerJS supports most JavaScript projects, including TypeScript, React, Angular, VueJS, Svelte, and NodeJS.
- **Python: MutPy**
 - Mutpy is a Mutation testing tool in Python that generates mutants and computes a mutation score. It supports the standard unittest module, generates YAML/HTML reports, and has colorful output.

Objective of Mutation Testing

The primary objective of mutation testing is to demonstrate the impact of these mutations by effectively “killing” them.

A mutant can be killed in 2 ways:

1. **Weak Mutant Killing:** Given a mutant $m \in M$ that modifies a location l in a program P , and a test t , t is said to weakly kill m if the state of the execution of P on t is different from the state of execution of m immediately after l .
2. **Strong Mutant Killing:** Given a mutant $m \in M$ for a ground string program P and a test t , t is said to strongly kill m if the output of t on P is different from the output of t on m .

We have chosen mutation testing as our strategy, focusing specifically on strongly killing mutants. This method systematically evaluates the code's robustness by ensuring that mutations cause significant changes in the program's behavior, thereby improving the effectiveness and reliability of the testing process.

Available Mutation Operators

A mutation operator provides a set of rules for making syntactic modifications to a program by substituting parts of the source code. Since mutations rely on these operators, researchers have developed a variety of them tailored to different programming languages, such as Java. The effectiveness of these mutation operators is crucial to the success of mutation testing. Researchers have extensively studied numerous mutation operators.

Below are some examples of mutation operators for imperative languages:

- Statement deletion
- Statement duplication or insertion, e.g. goto fail;
- Replacement of boolean subexpressions with true and false
- Replacement of some arithmetic operations with others, e.g. + with *, - with /
- Replacement of some boolean relations with others, e.g. > with \geq , == and \leq
- Replacement of variables with others from the same scope (variable types must be compatible)
- Remove method body

For the purpose of this project, we have covered some of the many available mutation operators, as covering all of them is not feasible in the given timeframe.

Mutation Testing Levels

- **Unit Mutation:** Unit mutation focuses on individual components like functions or methods. It delves into the internal workings of these units, making changes such as altering operators or tweaking logic. This testing method assesses how well the test suite can detect modifications within these isolated components, ensuring each piece of the software puzzle functions correctly on its own.
- **Integration Mutation:** Integration mutation (sometimes called interface mutation) works by creating mutants on the connections between components. Most of the mutations are around method calls, and both the calling (caller) and the called (callee) method are considered.

For the purpose of this project, we have focused our efforts on Unit Mutation Testing. We have also covered Mutation Integration Testing manually, as the tools we chose do not support it directly.

The Mutation Integration Operators covered are:

1. IPEX - Integration Parameter Exchange
2. IMCD - Integration Method Call Deletion
3. IREM - Integration Return Expression Modification

Original Code:

```
function complexMathCalculation(a, b, c, d) {  
  const sum = add(a, b);  
  const difference = subtract(c, d);  
  const product = multiply(sum, difference);  
  const result = divide(product, 2);  
  let finalResult = result;  
  
  for (let i = 0; i < 5; i++) {  
    if (i % 2 === 0) {  
      finalResult = add(finalResult, i);  
    } else {  
      finalResult = subtract(finalResult, i);  
    }  
  }  
  return finalResult;  
}
```

IPEX Mutant:

```
// mutant ipex  
function complexMathCalculation1(d, b, c, a) {  
  const sum = add(a, b);  
  const difference = subtract(c, d);  
  const product = multiply(sum, difference);  
  const result = divide(product, 2);
```

```

let finalResult = result;

for (let i = 0; i < 5; i++) {
  if (i % 2 === 0) {
    finalResult = add(finalResult, i);
  } else {
    finalResult = subtract(finalResult, i);
  }
}
return finalResult;
}

```

IMCD Mutant:

```

// mutant imcd
function complexMathCalculation2(a, b, c, d) {
  const sum = 100;
  const difference = subtract(c, d);
  const product = multiply(sum, difference);
  const result = divide(product, 2);
  let finalResult = result;

  for (let i = 0; i < 5; i++) {
    if (i % 2 === 0) {
      finalResult = add(finalResult, i);
    } else {
      finalResult = subtract(finalResult, i);
    }
  }
  return finalResult;
}

```

IREM Mutant:

```

// mutant irem
function complexMathCalculation3(a, b, c, d) {
  const sum = add(a, b);
  const difference = subtract(c, d);
  const product = multiply(sum, difference);
  const result = divide(product, 2);
  let finalResult = result;

  for (let i = 0; i < 5; i++) {
    if (i % 2 === 0) {
      finalResult = add(finalResult, i);
    } else {
      finalResult = subtract(finalResult, i);
    }
  }
  return -finalResult;
}

```

Mutation Score

The mutation score is defined as the percentage of killed mutants with the total number of mutants:

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Mutants}} \times 100$$

Test cases are deemed mutation adequate if they achieve a score of 100%. Experimental studies have demonstrated that mutation testing is a highly effective method for evaluating test case adequacy. However, its primary limitation lies in the significant cost of generating mutants and running each test case against the mutated program.

Details of Source Code

The codebase comprises of various algorithms and computations. Our idea was to create a set of codes with decent number of operators, such as `+`, `-`, `*`, `/`, `>`, `<`, etc. It is our belief that such pieces of codes are good candidates for mutation testing as they provide a significant chance for mutations to occur.

The algorithms under test span from simple programs such as addition, to interesting mathematical concepts such as computing the convex hull for a set of points and Simpson integration. The source code in between covers interesting applications of programming, such as credit card number validation, and glosses over a few popular DSA algorithms like KMP string matching and connected components in a graph. This showcases the usefulness of Mutation Testing (and Testing in general) across different programming tasks.

Lines of Code

- **Source Code:** 456 (JavaScript) + 366 (Python) = 822 lines
- **Test Cases:** 535 (JavaScript) + 466 (Python) = 1001 lines

Besides these components, the project codebase also includes a PowerShell script to automate running MutPy on multiple files in a directory.

Mutation Operators Used

Python (MutPy)

1. AOR - Arithmetic Operator Replacement
2. ASR - Assignment Operator Replacement
3. BCR (incompetent) - Break Continue Replacement
4. COI - Conditional Operator Insertion
5. EHD (incompetent)
6. EXS (incompetent)
7. ROR - Relational Operator Replacement
8. AOD - Arithmetic Operator Deletion
9. LCR - Logical Connector Replacement
10. COD - Conditional Operator Deletion

The details for Mutation Testing using MutPy can be found in the generated HTML reports at `./python/src/Output/{algorithm_name}`. A couple of examples are attached below (convex hull and temperature conversion, respectively):

MutPy mutation report

🕒 22.11.2024 22:25

Target

- `convex_hull_graham_scan`

Tests [5]












- `convex_hull_graham_scan_test` [0.107 s]

Result summary

- 📊 Score - 77.0%
- 🕒 Time - 7.2 s

Mutants [74]

- 🔴 killed - 57
- 🟢 survived - 17
- 🟡 incompetent - 0
- ⌛ timeout - 0

77.0%					23.0%	
#	Module	Operator	Tests run	Duration	Result	
1		AOD [4]	5	0.089 s	🔴 survived	➡
2		AOD [6]	5	0.057 s	🔴 survived	➡
3		AOD [19]	2	0.139 s	🟢 killed	➡
4		AOD [40]	2	0.08 s	🟢 killed	➡
5		AOD [41]	1	0.065 s	🟢 killed	➡
6		AOD [41]	2	0.053 s	🟢 killed	➡
7		AOD [41]	2	0.049 s	🟢 killed	➡
8		AOD [45]	1	0.053 s	🟢 killed	➡
9		AOD [45]	1	0.058 s	🟢 killed	➡
10		AOD [52]	1	0.05 s	🟢 killed	➡
11		AOD [52]	1	0.056 s	🟢 killed	➡

MutPy mutation report

🕒 22.11.2024 22:28

Target

- `temperature_conversion`

Tests [7]











- `temperature_conversion_test` [0.056 s]

Result summary

- 📊 Score - 88.2%
- 🕒 Time - 1.7 s

Mutants [34]

- 🔴 killed - 30
- 🟢 survived - 4
- 🟡 incompetent - 0
- ⌛ timeout - 0

88.2%					11.8%	
#	Module	Operator	Tests run	Duration	Result	
1		AOR [3]	1	0.092 s	🟢 killed	➡
2		AOR [3]	1	0.043 s	🟢 killed	➡
3		AOR [3]	1	0.043 s	🟢 killed	➡
4		AOR [3]	7	0.037 s	🔴 survived	➡
5		AOR [3]	1	0.034 s	🟢 killed	➡
6		AOR [3]	1	0.04 s	🟢 killed	➡
7		AOR [6]	2	0.035 s	🟢 killed	➡
8		AOR [9]	3	0.047 s	🟢 killed	➡
9		AOR [9]	3	0.043 s	🟢 killed	➡
10		AOR [9]	3	0.034 s	🟢 killed	➡
11		AOR [9]	7	0.034 s	🔴 survived	➡

JavaScript (Stryker)

All files

Mutants

Tests

All files

531

81

101

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
All files	74.47	86.76	490	81	41	101	0	0	0	531	182	713
JS add.js	100.00	100.00	2	0	0	0	0	0	0	2	0	2
JS catalan-numbers.js	100.00	100.00	18	0	3	0	0	0	0	21	0	21
JS convex-hull-graham-scan.js	90.00	90.00	86	10	4	0	0	0	0	90	10	100
JS determinant.js	92.65	94.03	62	4	1	1	0	0	0	63	5	68
JS find-month-calendar.js	38.99	66.67	50	31	12	66	0	0	0	62	97	159

convex-hull-graham-scan.js

Mutants

Tests

All files / convex-hull-graham-scan.js

531

81

101

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
convex-hull-graham-scan.js	90.00	90.00	86	10	4	0	0	0	0	90	10	100

← →

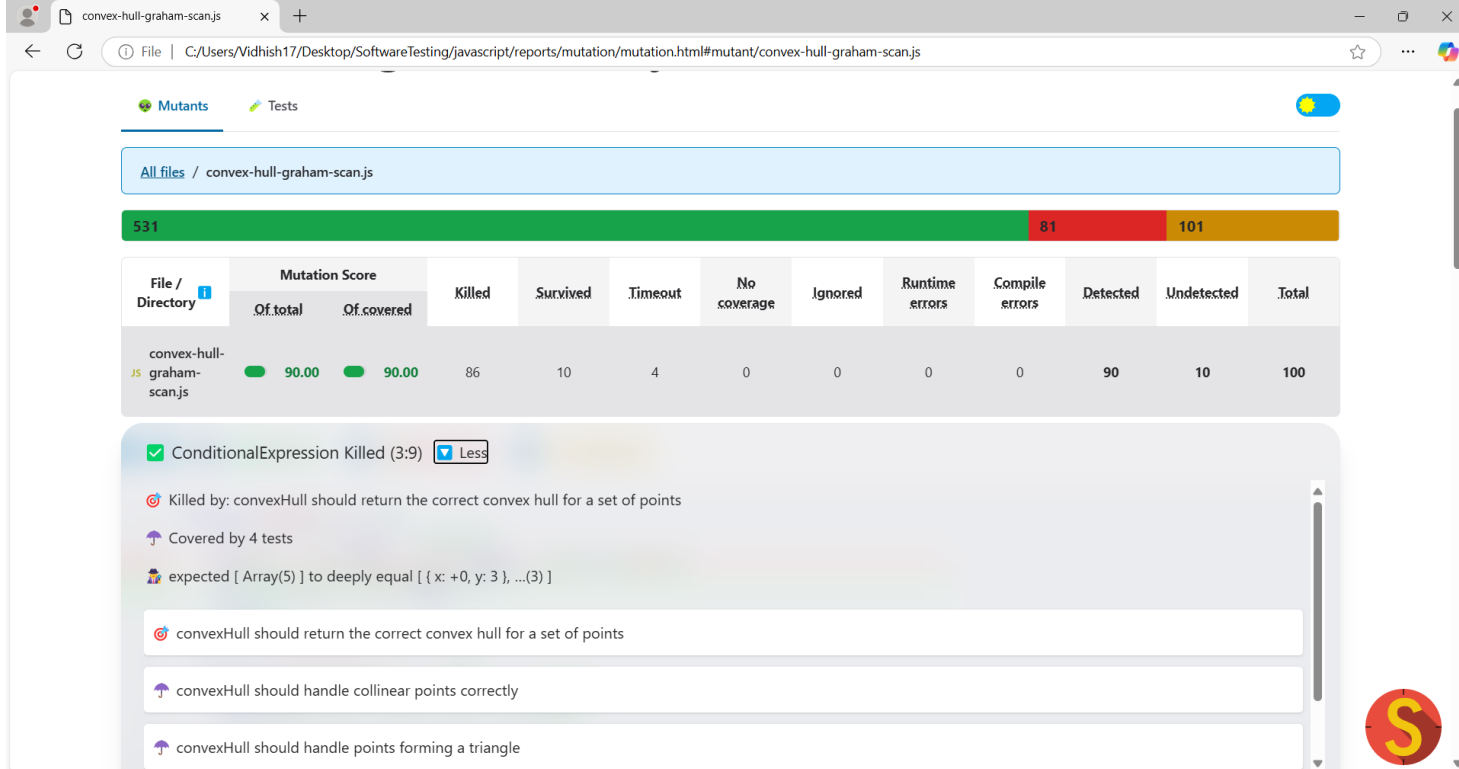
☐ Killed (86)

☒ Survived (10)

☒ Timeout (4)

```
1 function compare(a, b) {
2   if (a.x < b.x) return -1
3   if (a.x === b.x && a.y < b.y) return -1
4   return 1
5 }
6 function orientation(a, b, c) {
7   const alpha = (b.y - a.y) / (b.x - a.x)
8   const beta = (c.y - b.y) / (c.x - b.x)
9
10  if (alpha > beta) return 1
11  else if (beta > alpha) return -1
```

```
1 function compare(a, b) {
2   if (a.x < b.x) return -1
3 - if (a.x === b.x && a.y < b.y) return -1
+ if (true && a.y < b.y) return -1
4   return 1
5 }
6 function orientation(a, b, c) {
7   const alpha = (b.y - a.y) / (b.x - a.x)
8   const beta = (c.y - b.y) / (c.x - b.x)
```



Running The Project

Common Steps

```
git clone https://github.com/Vidhish-Trivedi/SoftwareTesting.git
cd SoftwareTesting
```

JavaScript

```
cd javascript
npm install
```

To run the test cases:

```
npm test
```

For mutation testing:

```
npx stryker run
```

Python

```
pip install pytest mutpy
```

For mutation testing:

```
mut.py --target src_file --unit-test test_file -m --runner pytest --report-html Output/
```


Alternatively, you may try:

```
python.exe  
C:\Users\Vidhish17\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8  
p0\LocalCache\local-packages\Python311\Scripts\mut.py --target src_file --unit-test  
test_file -m --runner pytest --report-html Output/
```

For both Stryker and MutPy, we are able to generate interactive HTML reports which summarize the results for mutation testing.

Future Scope

The scope of the current project is limited to Mutation Testing over unit tests of various algorithms and computations which we believe are good candidates for mutation testing. This is achieved through the use of tools such as Stryker (for JavaScript) and MutPy (for Python).