Blake Davis
Department of Computer Science
Colorado State University
Fort Collins, Colorado, USA
December 10th, 2023

Evaluation of the Mutation Testing Approaches used in Mutatest and MutPy on Open-Source Python Programs

Abstract

Mutation testing is a crucial part of evaluating the quality of a test suite. Statement Coverage Percentage is not the only metric that should be used to evaluate how good a test suite is. This study attempts to evaluate the mutation testing approaches used in Mutatest and MutPy on open-source Python programs by evaluating the relationship between Statement Coverage Percentage and Mutation Score as well as evaluating the cost-effectiveness of each mutation testing approach through metrics like Average Execution Time Per Mutant and Mutation Score. To complete my evaluation, I created several subsets of each open-source project's large test suite, found the Statement Coverage Percentage, and collected the other metrics by running each mutation testing tool on each Python program with each test suite. It was found that Statement Coverage Percentage is not strongly correlated with Mutation Score and that each approach to mutation testing has its pros and cons in terms of cost-effectiveness.

Keywords

Mutation Score, Statement Coverage Percentage, Cost-Effectiveness of Mutation Testing Approaches, Industry Application, Open-Source Python Programs

I. Introduction

This project focused on mutation testing in Python using two mutation testing tools, Mutatest and MutPy. Mutation testing is the action of creating mutants by modifying the program slightly to test the quality of the test cases in the test suite. Just like branch and statement coverage is a way to evaluate the quality of test cases, this is another way to evaluate their quality. Many programs are able to achieve a high level of code coverage but may not necessarily achieve a high mutation score, which is one of the things this project aims to evaluate. Mutation testing is another way to potentially determine if the test cases are actually catching faults that may or may not exist within the program, just like code coverage tries to do.

This project interests me as it is an approach to test suite evaluation that I had never heard of before. I've always felt that only using branch and statement coverage to evaluate the quality of a test suite was not enough. Mutation testing is an approach that is quite computationally intensive and is often not used in industry because of its cost. I am hoping to learn how to write better tests from this empirical study as I will be digging into these mutation testing tools and how they work.

For this study, I will be directly comparing two mutation testing tools on several open-source Python programs with 2 main goals:

1. To evaluate the relationship between mutation score and statement coverage percentage.
2. To evaluate each mutation testing approach in terms of cost-effectiveness.

In order to understand how statement coverage percentages and mutation scores differ from each other, we must first understand their relationship. This relationship is important because if statement coverage percentage and mutation score are highly correlated, it doesn't make sense to perform mutation testing in addition to statement coverage as both take time and resources that could be potentially used elsewhere in the development process. To evaluate the mutation testing approaches of the two mutation testing tools for Python, Mutatest and MutPy, and how the statement coverage percentage of a test suite affects the mutation score generated by each of the tools, I started by creating several subsets of each open-source project's test suite. A script was used to complete this to ensure random test file selection and

to ease the process of subsetting each test suite. Subsetting each test suite that was provided with each of the open-source Python projects allowed me to generate test suites with different amounts of statement coverage, but because this was fully random, some of the test suites that contained a larger number of test files had a lower amount of statement coverage. I then ran each test suite and respective Python program with each of the mutation testing tools to get the mutation score and the number of mutants generated by each mutation testing tool.

To apply mutation testing tools in industry, they must be effective and efficient. Companies in the private or public sector will not apply or use these mutation testing tools if they don't understand how to apply them effectively or if they consume too many resources in terms of time, money, and manpower. They are simply businesses, they operate in order to make a profit. If the pros of any tool or process do not outweigh the costs of using that tool or process, it simply will not be used unless absolutely required to complete the project. This project focuses on the cost-effectiveness of the mutation testing tools and their approach to evaluating the quality of test suites.

To answer this question, I used the execution time of the tool from start to finish, the average execution time per mutant as each mutation testing tool has a different approach that generates a different number of mutants, the mutation score generated by each of the tools, and the percentage of the program mutated as one mutation testing tools only mutates a small portion of each Python program. The total time to execute each mutation testing tool and the average amount of time per mutant may seem like redundant metrics but are useful to understand the differences between the two mutation testing approaches. All of these metrics allow an evaluator of the cost-effectiveness of each mutation testing approach to create a well-rounded picture of how much time and how many resources it would take to implement mutation testing into their development process.

II. Approaches Being Evaluated

A. Mutatest

Mutatest enumerates all of the possible locations to mutate in a program and, under the default setting, randomly selects 10 of the locations to mutate during execution. [7] You can specify the number of locations mutated in the call to Mutatest, but as I did not know the number of mutation locations available upon the first run of Mutatest on a given program, I left the number of mutation locations at the default number 10. [7] The random mutation location selection can be set with a seed to generate the same mutation testing results for each run. The seed 514 will be used for this project. I realized that the mutation testing locations were randomly selected when I ran the same tool on the same open-source program and test suite and got different results every time. For this study, Mutatest will be set to create as many mutations as it can at these 10 locations by using full mode. Mutatest defaults to break out of running that mutation location upon the first survivor, but it was specified in the call to Mutatest that it should exhaustively test each of the 10 locations. [7] In addition, in Mutatest, there is a way to filter the locations mutated with a coverage file, but that feature will not be used in this experiment as the possible mutation locations that are enumerated with each execution of Mutatest would change with each test suite. [7] Mutatest has the ability to be run in parallel and because this study is looking at the application of these mutation testing approaches to industry-grade programs, all of these programs were run and timed with Mutatest running in parallel. [7] All of these tools were run on computers in Colorado State University's Computer Science Building that had the ability to run this tool in parallel.

B. MutPy

MutPy enumerates all of the possible locations to mutate as well but then runs full, exhaustive mutation coverage at every location so there is no random mutation location selection being used. [4] There is also no setting to change how much mutation is done at each location in MutPy, but full mutation coverage is happening anyway and does not need to be specified in the call to MutPy. A difference between Mutatest and MutPy is that you can specify which type of mutation operators are used (i.e. AOD, BCR, LOD, etc.) in MutPy. [4] For this study, no mutation operations were specified so all mutation operators, the default setting, were used. MutPy has no option to be run in parallel and was run sequentially for this project. [4]

III. Evaluation Goals, Research Questions, and Metrics

Goal	- To be able to evaluate the relationship between the statement coverage percentage and the mutation score of each test suite with each Python program.
Questions	- How does the amount of statement coverage in a test suite affect the mutation score of each mutation testing approach?
Metrics	- Mutation Score generated by each mutation testing tool. - Statement Coverage Percentage of each test suite.
Goal	- To be able to evaluate which mutation testing approach is more cost-effective in terms of execution time and performance.
Questions	- Which mutation testing tool is more cost-effective in terms of total run time and the average amount of time it took the mutation testing tool to test each mutant? - Which mutation testing tool generates a higher mutation score for each Python program?
Metrics	- Execution Time of each mutation testing tool on each Python program. - Average amount of time to execute each mutation testing tool per mutant on each Python program (Total Execution Time / Number of Mutants Created and Tested). - Mutation Score generated by each mutation testing tool. - Percentage of Program Mutated (Given by Mutatest, Assumed to be 100% by MutPy).

A. Metric Definitions:

- Mutation Score: Mutation Score is the percentage of mutants killed out of the total number of mutants. ((Number of Mutants Killed/Total Number of Mutants Generated) * 100)
- Statement Coverage Percentage: Statement Coverage Percentage is the percentage of statements that were covered or executed during the execution of the test suite. ((Number of Statements Executed/Total Number of Statements in Program) * 100)
- Execution Time: Execution Time is the amount of time it took to execute the mutation testing tool on the specified program. The execution time is self-contained and recorded by each mutation testing tool starting upon the command line call to the mutation testing tool and ending upon completion. It is measured in seconds up to the thousandth place for each tool.
- Average Execution Time Per Mutant: Average Execution Time Per Mutant is the amount of execution time it took for the mutation testing tool to execute per mutant. This is important because there is variation between the approaches in how many mutants are generated so this allows for a better comparison of execution times as I am making that comparison on a per-mutant basis. Total Execution Time is still important as it still takes that amount of time to complete a run of the mutation testing tool and that is what a project manager and development team in industry would use. (Execution Time/Number of Mutants Generated)
- Percentage of Program Mutated: Percentage of Program Mutated is the percentage of the program that was actually changed upon execution of the mutation testing tool. Mutation testing changes certain lines in the program being tested to kill mutants. Mutatest only changes a small portion of the programs in this project. ((Number of Mutation Locations Mutated/Total Number of Mutation Locations) * 100)

IV. Study Design

A. Independent Variables:

- The Python program that is being tested. (Controlled and Changed Throughout the Experiment)
- The mutation testing tool that is being run on a given Python program. (Controlled and Changed Throughout the Experiment)
- The computer that the Python programs and tools are being run on. (Fixed)
- The Python testing tool being run (Pytest or Unittest). (Fixed. Except for when Pytest wouldn't work for one Python program when run with MutPy, Unittest was used. [14])
- The test suite that is being run on the program. (Controlled and Changed Throughout the Experiment)

B. Dependent Variables:

- The total execution time of each mutation testing tool on each Python program.
- The average amount of time to execute each mutation testing tool per mutant on each Python program.
- The number of mutants created by each mutation testing tool for each Python program.
- The mutation score that is generated by each mutation testing tool on each Python program.
- The Statement Coverage Percentage of each test suite.

C. Steps Taken To Perform Experiment

This project evaluates the approaches to mutation testing used in Mutatest and MutPy, two mutation testing tools for Python, on five Open-Source programs and one CS 220: Discrete Structures and their Applications program. The five Open-Source programs include:

- Augmentor, an image augmentation library for machine learning in Python. [3]
- Asciinema, a terminal session recording software. [1]
- Python-Fire, a library by Google for automatically generating command line interfaces (CLIs) from Python objects. [9]
- The source code and test suite of Mutatest, a mutation testing tool for Python. [6]
- The source code and test suite of MutPy, a mutation testing tool for Python. [11]

To begin working with each mutation testing tool, I used a Python program from when I was a student in CS 220: Discrete Structures and their Applications, an undergraduate Computer Science course at Colorado State University. This program simply returns a boolean value based on a specific operator with 2 inputs. For example: p = True and q = True as inputs, p implies q is True as output. I wrote a small test suite of 8 tests when initially starting this project to begin working with and understanding the mutation testing tools. I tested that these test cases pass using pytest, a Python testing framework, to determine eligibility for the program to be included in this project. [8] I then ran each mutation testing tool on the simple Python program to ensure compatibility.

To determine if an open-source Python program is eligible to be included in this project, I first ran pytest to determine if all of the tests in the provided test suite pass execution. If a test does not pass execution, it will be removed by either being skipped through pytest or commented out in the test suite. Each open-source Python program will then be run through Mutatest and MutPy to test compatibility with each of the mutation testing tools. The initial executions of each Python program to test compatibility were not included in the final dataset and the output from those runs was not recorded.

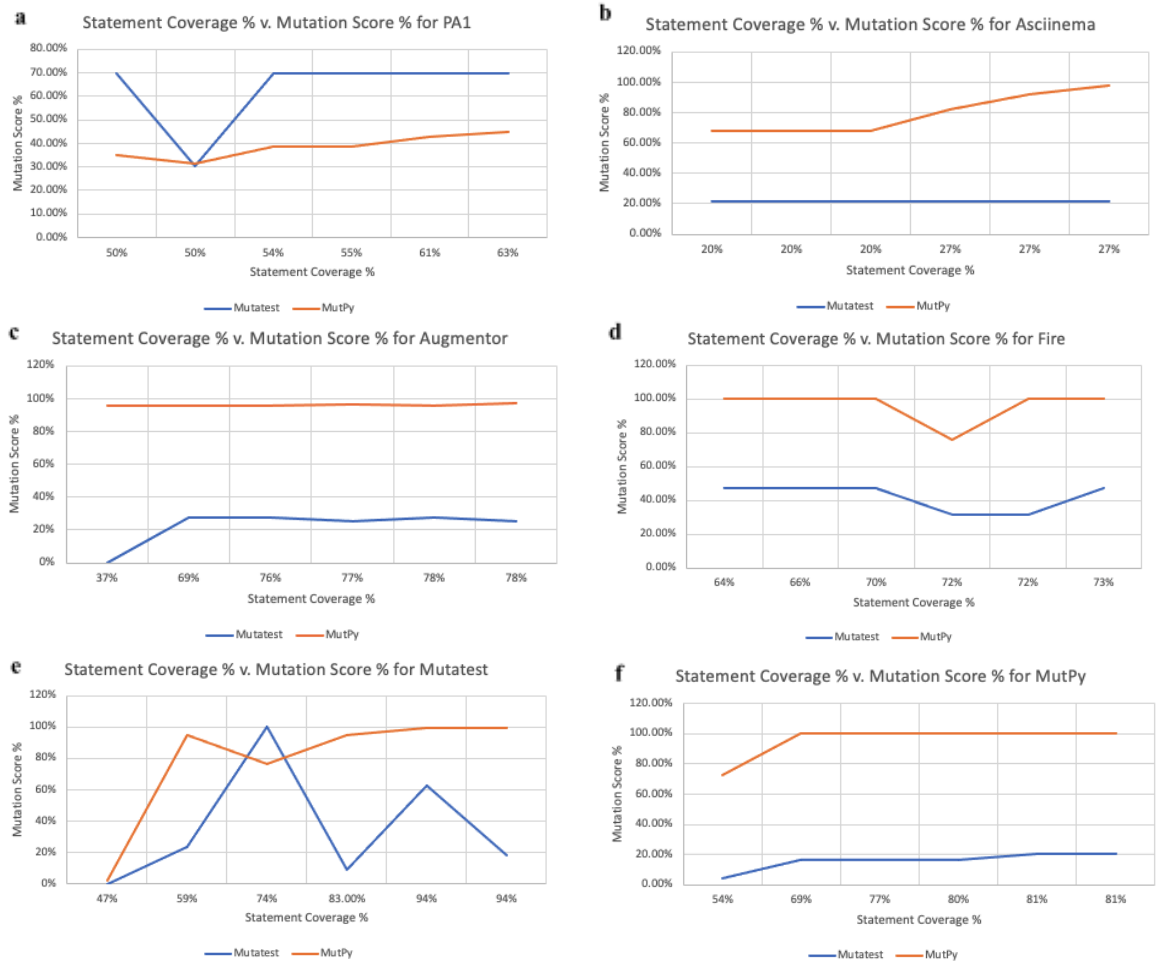
The next step was to write a script that randomly selects test files from each test suite to create 5 subsets of the test suite that was provided with each open-source Python program. I also wrote a separate script to subset the small test suite that I created to test the program that I wrote as a student in CS 220. The test suite subsets will contain roughly 50%, 60%, 70%, 80%, and 90% of the test files in the provided test suite, respectively. This was done for each Python program, meaning that each program will have 6 test suites (5 subsets and the original, provided test suite). With 5 open-source programs and the small Python program from CS 220, I had a total of 36 test suites.

I then ran Coverage.py, a code coverage evaluation tool for Python, to retrieve the statement coverage percentage that each test suite and its subsets has on its respective Python program. [2] I then ran each mutation testing tool, Mutatest and MutPy, on each new test suite with their respective Python

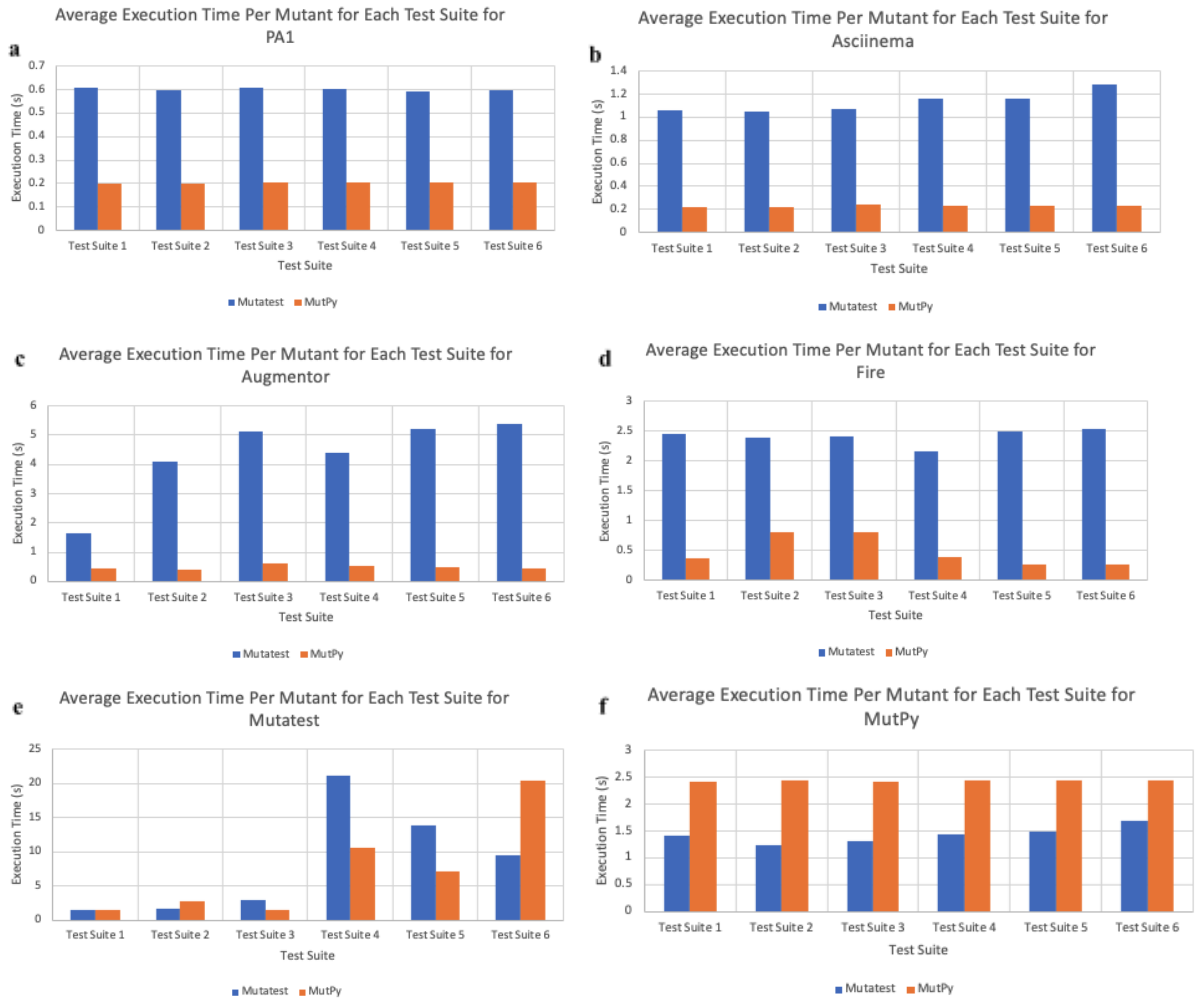
program to get the mutation score for each test suite from each mutation testing tool. I am interested in the relationship between the statement coverage percentage and the mutation score so retrieving these two metrics for each test suite allowed me to generate enough data to plot the statement coverage percentage versus the mutation score for each Python program for both of the mutation testing tools. The total execution time of each mutation testing tool and the number of mutants generated are tracked by each of the tools and are directly reported in the Results section of the paper.

All data was collected on a Linux machine owned and maintained by the Department of Computer Science at Colorado State University. The Linux machine that was used to collect this data has an Intel Core i7-12700K with 12 cores and 20 threads. This hardware that this data was collected on matters for 2 reasons. The first is that Mutatest was run in parallel for this project and the number of threads affects how many jobs can run in parallel. The second is that the sequential processing speed of the machine matters as it could affect the execution time of the mutation testing tools running sequentially and in parallel.

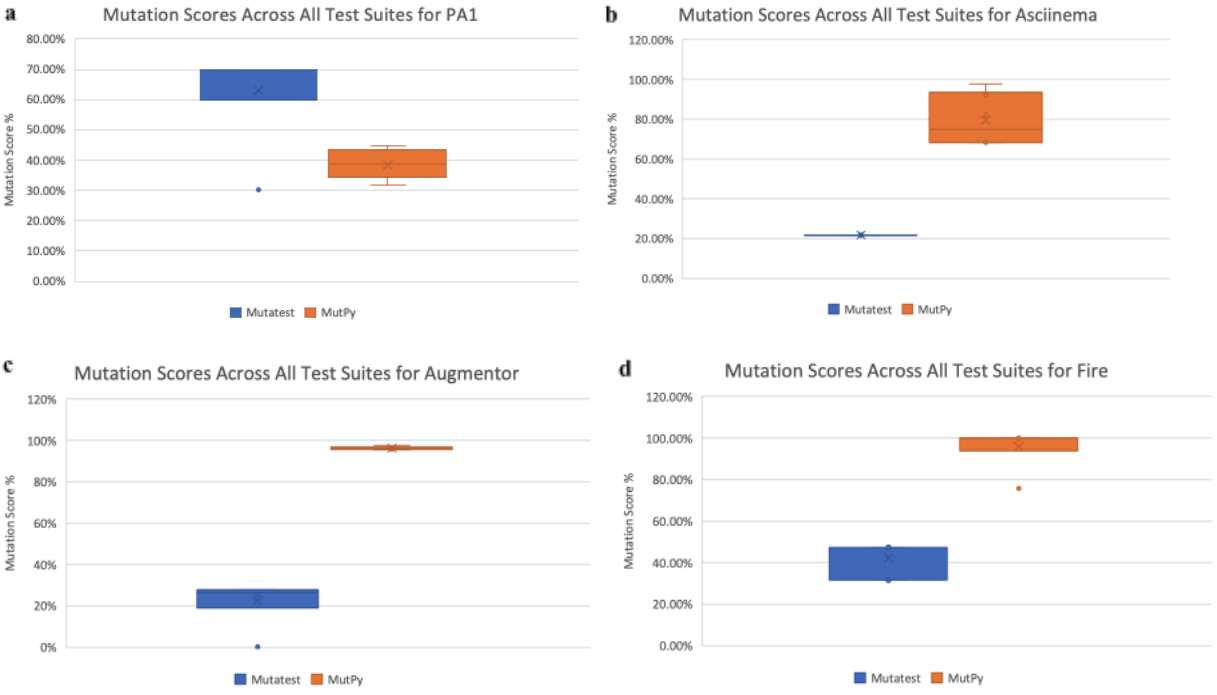
V. Results

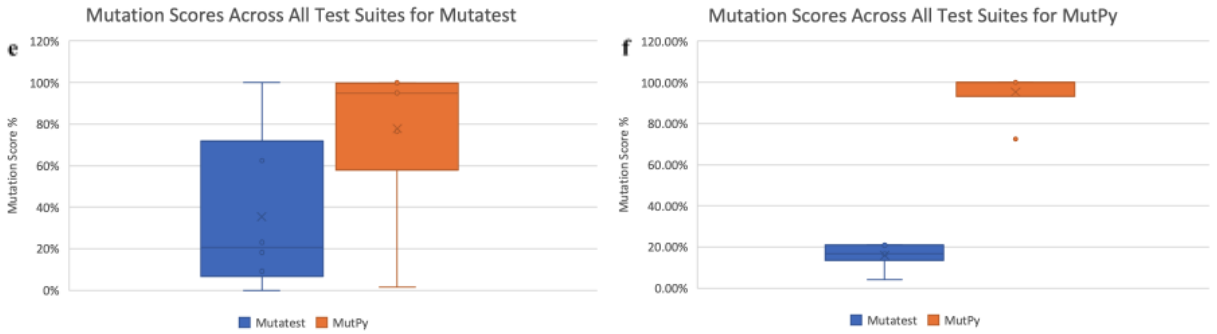


Figures 1a - 1f. Statement Coverage % versus Mutation Score % for each Python program with the x-axis sorted by Statement Coverage %.



Figures 2a - 2f. Average Execution Time Per Mutant For Each Test Suite for each Python program.





Figures 3a - 3f. Mutation Score Percentage Across All 6 Test Suites for each Python program.

	PA1	Asciinema	Augmentor	Fire	Mutatest	MutPy
Mutatest	12.5%	5.59%	1.19%	0.67%	1.09%	1.80%
MutPy	100%	100%	100%	100%	100%	100%

Table 1. Percentage of Program Mutated for each Python program for each mutation testing tool.

	PA1	Asciinema	Augmentor	Fire	Mutatest	MutPy
Mutatest	19.833 s	26.000 s	202.433 s	45.658 s	109.355 s	34.327 s
MutPy	23.249 s	330.125 s	1094.303 s	3507.250 s	82816.590 s	3839.042 s

Table 2. Average Total Execution Time of All 6 Executions (One for Each Test Suite) for each Python program.

VI. Discussion

After some analysis, I found that Statement Coverage Percentage is not strongly correlated with Mutation Score. Figures 1a - 1f show that there is not a specific trend that Mutation Score follows as Statement Coverage increases. This makes sense as more tests do not necessarily mean you will have a higher Statement Coverage Percentage or Mutation Score, highlighting quality over quantity. In Figure 1a, the mutation score for Mutatest is higher for most of the test suites. I suspect this is because the size of PA1 is much smaller in comparison to the size of the open-source programs. This is also evident in Figure 3a. Additionally, I found that Mutatest has a higher average execution time per mutant for the first 4 Python projects as seen in Figures 2a - 2d, but has a lower average execution time per mutant for the source code and test suites of Mutatest and MutPy, as seen in Figures 2e - 2f. My hypothesis is that this may be because of executing Mutatest in parallel mode and the overhead that comes with that. We cannot say there is a trend of increasing average execution time per mutant as the test suites on the x-axis are not sorted in Figures 2a - 2f. In addition, I found that the test suite used with the mutation testing tool heavily affects the mutation score, especially if you have quality, new tests with each updated test suite. This is obvious because that is what is supposed to happen with mutation testing, but I included Figures 3a - 3f to show that both of these approaches have the ability to evaluate a test suite and produce a mutation score that represents the quality of the test suite through mutation testing.

On that same note, I noticed that Mutatest consistently returns lower mutation scores for most of the Python programs. There could be two reasons for this, the first being that Mutatest may generate mutants that are harder for the test suite to kill. The second is that Mutatest randomly selects locations to mutate in each program. I made the random mutation location selection consistent using a seed, but it is possible that Mutatest selected locations where the test suites were unable to kill as many mutants.

Looking at the percentage of program mutated table in Table 1, it is evident that only small portions of the programs were actually mutated during the executions of Mutatest. This is because I used the default setting of 10 mutation locations for Mutatest whereas MutPy mutated every possible mutation location in all of the programs. I included these results because when considering the cost-effectiveness of each mutation testing approach, the amount of work the tool actually completes must be included in the consideration. It is noted in the Threats to Validity section that this default setting should be set to a larger number to be used on larger programs to get a better feel for the quality of the entire test suite. Something to consider is that a user of Mutatest could configure how many locations they want to actually test based on how much of their test suite they want to be tested and how much time they have. In Table 2, it can be seen that MutPy takes a lot longer across the board to complete mutation testing which makes sense since it generates a lot more mutants than Mutatest. These results were included because when considering which mutation testing tool to use on these larger open-source Python programs, the total execution time must also be considered. After all, you don't always have time to let the mutation testing tool run for days. This is where the configurability of Mutatest may be better suited for projects on a time crunch even though the user has less control over which locations are actually mutated, aside from setting the seed.

VII. Threats to Validity

A. External Validity

The results of this study can generalize rather well to other open-source Python programs as the programs that this study used were of varying sizes to demonstrate how these tools would generalize to other open-source Python programs. My comparison of Average Execution Time Per Mutant may not generalize as well as the other metrics to other contexts as I left the number of mutated locations at the default value of 10. It is possible that by increasing the number of mutated locations, the Average Execution Time Per Mutant for Mutatest may drop below the Average Execution Time Per Mutant for MutPy for a majority of mutated projects. These results are still useful because if a user was trying to get the mutation score for a small sample of their large project using Mutatest with a sample of 10 mutation locations, the results of this project apply. It is also possible that running Mutatest in parallel is adding extra overhead which is increasing the Average Execution Time Per Mutant in comparison to MutPy which runs sequentially. In addition, it is also possible that these tools will not run on other open-source Python projects. I searched for open-source Python programs that had passing tests and that the mutation testing tools executed on and many of them either did not have passing test suites, making mutation testing unhelpful, or the mutation testing tool simply would crash when running while open-source program.

B. Internal Validity

In this study, I concluded that Statement Coverage Percentage is not strongly correlated with Mutation Score. Both are ways to measure the quality of a test suite, but they both measure different things about quality. There may be some other factor that is changing the relationship between these two metrics, such as a confounding variable.

C. Construct Validity

One threat to validity in the construction of my study is that I did not average the execution times of the mutation testing tools across multiple trials. It is very possible that the Linux machine I was using was under heavy load while I was collecting my data. I collected all of my results over Fall Break and did my best to find a machine that had a light CPU load, but I did not monitor the load while these projects were running throughout the week. The timing results of my project are probably still in the ballpark of the execution times that I would have reported if I averaged the times across multiple trials.

VIII. Related Work

There are several works related to this project. The first is about a mutation testing tool called Judy and how it compares against other mutation testing tools for Java. The paper presents "an innovative approach to mutation testing that takes advantage of a novel aspect-oriented programming mechanism… to speed up mutation testing." [10] In their study, they compare several mutation testing tools for Java

through many distinct characteristics and features, including Traditional Selective Mutation Operators (ABS, AOR, LCR, ROR, UOI), Object-Oriented (OO) Mutation Operators, Mutant Generation Level, Mutants Format, JUnit Support, and more. [10] This paper is related to this project as it discusses how to compare several Java Mutation Testing tools and their approaches where this project will compare two mutation testing tools for Python, Mutatest and MutPy. This project used a few similar metrics such as mutation score and a few different metrics such as statement coverage percentage and average execution time per generated mutant. The similarities and differences between this project and this paper are important as I generated ideas on how to approach my evaluation of each mutation testing tool for Python from this paper.

Derezińska and Hałas discuss mutation operators and their application to the Python language through MutPy, a mutation testing tool for Python that includes mutation operators that make sense for the unique and dynamically typed Python language. [4] The researchers also discuss how some mutation operators are more risky than others, specifically operators that add a new element to mutate a statement are riskier than the operators that change or delete a program element/statement to create a mutated version of the program as adding new elements often creates incompetent mutants in Python. [4] In addition, the researchers included a discussion of why a few Object-Oriented Mutation Operators were included in MutPy as Python allows for the use of basic Object-Oriented concepts, but most standard Object-Oriented Mutation Operators are not suitable for Python. Python operators that required type verification during the mutation process were also omitted from the operators included in the MutPy mutation testing tool. [4] This heavily relates to this project as the approach that MutPy takes on mutation testing in Python was one of the two approaches evaluated in this project. Reading this paper allowed me to better understand the approach that MutPy takes upon execution to make a better comparison of mutation testing approaches in Python.

Derezińska and Hałas evaluate several different problems with mutation testing including dealing with incompetent mutants, the distinct types of structural and Object-Oriented Mutational Operators, the procedural generation of mutants, and killing mutants. [5] The paper also discusses how these problems were solved in version 0.3.0 of MutPy. [5] The researchers used 4 Open-Source Python projects to run MutPy on (they used version 0.3.0 while this project used version 0.6.1) and analyzed several metrics including mutation time, code coverage (assuming statement coverage), and mutation score. [5] This project will analyze two mutation testing tools and several open-source projects while using similar metrics to analyze the performance of each tool. The researchers also ran their program on version 0.2.0 of MutPy, a previous working version, which is where I got the idea to use the source code and test suite of MutPy as one of the open-source Python programs to include in this project. [5] This project used version 0.6.1 of MutPy and used the source code and test suite from 0.6.1 as a Python project which I did not see as a threat to validity. The authors of this paper also used two very different types of machines to run and collect their data on, a MacBook and an IBM x3755 server, to see how hardware affects the results generated by the mutation testing tool. [5] This project used only one computer to collect all data, a Linux machine with an Intel Core i7-12700K 12-core processor with 20 threads.

Some related work to this project is about the industrial application of mutation testing. The authors of this paper learned several lessons throughout their study including that not all mutants should be killed and that mutation adequacy, getting a perfect or high mutation score, is too expensive. [13] Starting with the first lesson, the researchers stated that there are several mutants that mutation tools generate that are redundant and unproductive as they are equivalent to other mutants which causes developer frustration. [13] They also learned that mutation adequacy is too expensive as the overall goal of a developer is to improve the overall quality of the test suite and not to kill every possible mutant. [13] The researchers are trying to communicate that mutation testing should be used effectively by using the mutation testing tool and mutation score percentage to write better tests, but mutation testing can only do so much. The authors also discuss mutants that are out of the scope of unit testing, such as changing the initial capacity of an ArrayList which only changes the performance of a program or creates an Out of Bounds Exception. [13] This paper applies to this project as I analyzed the cost-performance relationship of each mutation testing approach for Python. Currently, mutation testing is not used very much in

industry as it consumes a lot of time and resources. This project tries to analyze which approach to mutation testing in Python is more cost-effective which could be used to make a recommendation of which tool should be used.

More related work to this project is on practical mutation testing at scale in industry, specific to Google. The paper presents cases where mutation testing outperforms statement coverage as a test case may utilize the statements by calling the function, but the mutant would survive as it did not check the state of the returned object effectively. [12] This is a great way to show why mutation testing is important and how it can make an impactful difference in evaluating the quality of a test suite if mutation testing is used effectively. The authors of this paper also introduce something that I hadn't thought of, incorporating mutation testing into code reviews. [12] The development team at Google can review surviving mutants and determine if they are useful to kill in testing or if it would be counterproductive to spend time writing tests to kill the mutant. [12] The paper also suggests only mutating the code changed during a code review rather than mutating the entire code base to prevent execution redundancy and save time. [12] This is a way to make mutation testing scale to large projects without using too many resources and too much time. This applies to this project as I am looking at the cost-effectiveness of mutation testing approaches on open-source projects in Python. If a company could take a mutation testing tool with an approach that works for them with an efficient development process, they would be able to implement mutation testing into their process with a reasonable amount of setup.

IX. Conclusions

I learned that Statement Coverage Percentage is not strongly correlated with Mutation Score. Figures 1a - 1f show that there is not a specific trend that Mutation Score follows as Statement Coverage Percentage increases. This makes sense as larger test suites do not necessarily mean you will have a higher Statement Coverage Percentage or Mutation Score highlighting quality over quantity. I also found that having a larger test suite does not necessarily mean that the test suite will have a higher Statement Coverage Percentage. This can be seen well in Figure 1f as the number of tests changed for each test suite, but the amount of Statement Coverage changed minimally or did not change at all on the second half of the x-axis. I also learned that Mutatest's approach to mutation testing in Python is generally slower on a per mutant basis. There are a variety of possibilities of why this happens so some future research including a deep dive into how Mutatest makes decisions would be required. In addition, I found that the test suite used with the mutation testing tool heavily affects the mutation score, especially if you have quality, new tests with each updated test suite. This is obvious because that is what is supposed to happen with mutation testing, but I included Figures 3a - 3f to show that both of these approaches can evaluate a test suite and produce a mutation score that represents the quality of the test suite through mutation testing. If I were to perform this experiment again, I would follow an approach that more closely resembles how mutation testing is performed in industrial contexts. Overall, this study taught me a lot about how to use mutation testing tools effectively.

References

[1] Asciinema, "Asciinema," *GitHub*, Accessed 1 November 2023, https://github.com/asciinema/asciinema.

[2] N. Batchelder, "Coverage.py," *Read the Docs*, Accessed 10 November 2023, https://coverage.readthedocs.io/en/7.3.2/.

[3] M. Bloice, "Augmentor," *GitHub*, Accessed 1 November 2023, https://github.com/mdbloice/Augmentor.

[4] A. Derezińska and K. Hałas, "Analysis of Mutation Operators for the Python Language," *Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX*, Brunów, Poland, 2014, vol 286, pp. 155-164, https://doi.org/10.1007/978-3-319-07013-1_15.

[5] A. Derezińska and K. Hałas, "Experimental Evaluation of Mutation Testing Approaches to Python Programs," *2014 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Cleveland, OH, USA, 2014, pp. 156-164, doi: 10.1109/ICSTW.2014.24.

[6] E. Kepner, “Mutatest,” *GitHub*, Accessed 1 November 2023, https://github.com/EvanKepner/mutatest.
[7] E. Kepner, “Mutatest,” *Read The Docs*, Accessed 1 November 2023, https://mutatest.readthedocs.io/en/latest/.
[8] H. Krekel, “pytest,” *Pytest*, Accessed 1 November 2023, https://docs.pytest.org/en/7.4.x/.
[9] Google. “Python-Fire,” *GitHub*, Accessed 1 November 2023, https://github.com/google/python-fire.
[10] L. Madeyski and N. Radyk, “Judy - a mutation testing tool for Java,” *IET Software*, vol. 4, no. 1, pp. 32-42, 2010, doi: 10.1049/iet-sen.2008.0038.
[11] MutPy, “MutPy,” *GitHub*, Accessed 1 November 2023, https://github.com/mutpy/mutpy.
[12] G. Petrović, M. Ivanković, G. Fraser, and R. Just, “Practical Mutation Testing at Scale: A view from Google,” *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900-3912, Oct. 2022, doi: 10.1109/TSE.2021.3107634.
[13] G. Petrović, M. Ivanković, B. Kurtz, P. Ammann, and R. Just, “An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions,” *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Västerås, Sweden, 2018, pp. 47-53, doi: 10.1109/ICSTW.2018.00027.
[14] Python Software Foundation, “Unittest,” *Python*, Accessed 1 November 2023, https://docs.python.org/3/library/unittest.html.

Appendix

The GitHub Repository that holds all results:
https://github.com/bdavis12302/Mutation-Testing-of-Open-Source-Projects-in-Python