2024 11th International Conference on Dependable Systems and Their Applications (DSA) | 979-8-3315-3239-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/DSA63982.2024.00030

An Analysis and Comparison of Mutation Testing Tools for Python

Kadiatou Diallo[1], Zizhao Chen[2,*], W. Eric Wong[2], and Shou-Yu Lee[3]
[1]Lawson State Community College, Birmingham, Alabama, United States
[2]University of Texas at Dallas, Richardson, Texas, United States
[3]Department of Computer Science, Tunghai University, Taiwan
A02394683@alabama.edu, zxc190007@utdallas.edu, ewong@utdallas.edu, shouyu@thu.edu.tw
[*]corresponding author

Abstract—Software testing is a crucial phase in the software development lifecycle, yet it often becomes a challenging task for engineers who must ensure comprehensive test coverage. While python unit testing frameworks like pytest and unit test are widely used to validate code functionality, passing these tests does not necessarily imply meaningful or effective testing. Mutation Testing addresses this gap by introducing deliberate faults, known as mutants, into the program under test to assess the quality of its test suite. This study evaluates five Python-based mutation testing tools: MutPy, Mutmut, Mutatest, Poodle, and Cosmic Ray, by applying them to two different open-source programs. The tools' performances were compared using mutation-specific and tool-specific criteria to determine their strengths, limitations, and analyze which tool created the most competent mutants. In the end, Poodle created the most competent mutants with a 50.9% competency score, Cosmic Ray came next with 25.7% competency score, and Mutmut was inconclusive due to incomplete mutation results.

Keywords-mutation testing; python; software testing

1. INTRODUCTION

Software testing plays a vital role in the software development lifecycle, guaranteeing that programs perform as expected, fulfill user requirements, and uphold high quality standards [1]. The main goal of software testing is to uncover bugs, and it is up to the developer to eliminate them. Software testing can show programmers two things, the existence of bugs, and an estimate of reliability [2]. As a result, more advanced and tailored testing techniques have been continuously developed to improve software quality assurance.

Mutation testing originated in the 1970s, pioneered by Richard DeMillo at the Georgia Institute of Technology, who introduced the concept of generating artificial faults to evaluate the effectiveness of software testing [4]. Early work in mutation testing focused primarily on programming languages such as Fortran with MOTHRA and C with CREAM, where researchers implemented fundamental mutation operators and began developing automated tools to assist in the process [4]. The technique gained popularity in academia due to its theoretical rigor and potential to advance the field of software testing. However, mutation testing remains largely a research-based technique [4], because it is computationally expensive and time-consuming to execute on larger codebases.

Over the years, as computing power increased and testing tools became more sophisticated, mutation testing techniques evolved. In the 1990s and 2000s, research expanded to include object-oriented programming languages like Java, where more advanced mutation operators were developed to handle Object Oriented Programming [4]. With the rise of dynamic programming languages like Python and Ruby, researchers have continued to innovate, developing mutation testing tools tailored to the unique characteristics of these languages.

Despite the advancements and the increasing number of mutations testing tools available for various programming languages, mutation testing has not yet seen adoption in industry [4]. The computational overhead of generating and executing mutants for large codebases can be significant [1], rendering mutation testing impractical for organizations operating under strict time constraints—a challenge faced by nearly all major software companies today. Mutation testing and its many configurations also requires a deep understanding of both the software being tested and the specifics of the mutation testing technique itself [6], which can discourage developers who are not familiar with the approach.

This research aims to emphasize the critical role of software testing by demonstrating mutation testing as a powerful approach for increasing test coverage. The study will conduct a comprehensive analysis of recent Python mutation testing tools to:

- Analyze and compare five different python mutation testing tools focusing on both mutation-specific and tool-specific criteria.
- Identify and discuss the most effective tool in terms of performance, usability, and mutation capability.
- Provide practical recommendations to address the shortcomings and improve the reliability of these tools.

By systematically evaluating these tools, this research will offer valuable insights into their strengths and limitations, contributing to the advancement of mutation testing practices.

2767-6684/24/$31.00 ©2024 IEEE
DOI 10.1109/DSA63982.2024.00030

This paper will go as follows, section 2 will include background information about mutation testing, section 3 will be the experimental setup and necessary criteria listed to compare the tools, section 4 will be the experimental results of the mutation tools, and section 5 will detail the conclusion and future directions for this research.

2. BACKGROUND

This section will give background information on mutation testing and the tools under study.

2.1 Mutation Testing Theory

Mutation testing is based on two key hypotheses: the Competent Programmer Hypothesis and the Coupling Effect Hypothesis [8]. The Competent Programmer Hypothesis suggests that while programmers may not write flawless programs, the programs they write are nearly correct. The Coupling Effect Hypothesis is when test data capable of detecting faults like the original program (mutants) are also likely to uncover more complex faults, indicating that complex faults are linked to simple ones.

Mutation testing is a fault-based technique designed to enhance the quality of test suites. The resulting versions of the program, called mutants, are executed against the test suite to compare outputs [6]. If a test case causes a mutant to fail, the mutant is considered "killed"; if not, the mutant "survives" and indicates a deficiency in the test suite's fault detection ability. The effectiveness of a test suite is quantified by the mutation adequacy score [6], which is the ratio of killed mutants to the total number of generated mutants.

Mutation operators introduce errors to a program under test to evaluate the strength of a test suite [9]. These operators simulate common coding mistakes by modifying code elements, such as changing arithmetic operations, logical expressions, or control flow. As shown in Table 1, each mutation testing tool may implement operators differently based on programming language constraints and developer design choices. In dynamic languages like Python, mutation operators must account for the flexibility of runtime type changes and variable behavior [9]. Recently, mutation testing has expanded to include object-oriented mutation operators that alter class structures and behaviors, making it especially useful for testing object-oriented designs.

Table 1. Python Mutation Testing Tools discussed in this paper

Tool-Specific Criteria	Tools				
	Mutmut	*Poodle*	*Cosmic Ray*	*Mutpy*	*Mutatest*
Creation Date	Dec 1, 2016	Dec 12, 2023	July 11, 2014	Feb 6, 2014	Jan 6, 2019
Last Update	v2.5.0 May 2024	v1.3.3 February 2024	v8.3.15 July 2024	v0.6.1 November 2019	V3.1.0 February 2022
Interface	CLI	CLI	CLI	CLI	CLI
Documentation	GitHub, readthedocs	GitHub, readthedocs	GitHub readthedocs	GitHub	GitHub, readthedocs
Test Libraries	Unittest, pytest	Unit test, pytest	Unit test, pytest	Unit test, pytest	Unit test, pytest
Configuration	yes	yes	yes	yes	yes

In python-based mutation testing, two main approaches are used to introduce faults into a program: Source Code mutators and Abstract Syntax Tree (AST) mutators. Source code mutators [10] directly alter the human-readable code. This approach is straightforward for developers, making it easy to understand and debug the mutations. However, it can be time-consuming because each mutation requires recompilation of the source code. AST mutators [11] manipulate the abstract syntax tree, an intermediate representation of the code structure. By modifying the AST, these mutators can apply more sophisticated changes that respect the language's syntax and semantics. However, implementing and maintaining AST-based tools can be complex.

2.2 Existing Mutation Testing Tools

The tools below were chosen based on two key criteria: the presence of relevant information in published research and the availability of comprehensive documentation on GitHub or ReadtheDocs. These five Python tools met these criteria, leading to their selection for experimentation. Every tool has a command line Interface, and is compatible with both pytest and unit test frameworks.

- *MutPy*

MutPy [12] is a mutation testing tool specifically designed for Python applications to assess and improve their test suites. This is one of the oldest mutation tools developed for python (Est. 2012) and has not been updated since 2019. Mutpy applies mutations on the AST level and allows for custom mutation operator creation, as shown in Table 2.

Table 2. Mutpy Mutation Operators [12]

Mutpy Mutation Operators: AST Level	
AOD - arithmetic operator deletion	Removes an arithmetic operator
AOR - arithmetic operator replacement	Replaces an arithmetic operator / to -
ASR- assignment operator replacement	Replaces assignment operators = to +=
BCR-break continue replacement	Replaces break with continue or vice versa
COD-conditional operator deletion	Removes a conditional operator
COI-conditional operator insertion	Inserts a conditional operator
CRP-constant replacement	Replaces a constant True to False
DDL-decorator deletion	Removes a decorator from a function
EHD-exception handler deletion	Deletes an exception handler block form a try catch structure
EXS-exception swallowing	Modifies exception handling to ignore the exception
IHD-hiding variable deletion	Removes a variable that hides another variable with the same name in a parent scope
IOD-overriding method deletion	Deletes a method that overrides a method in a parent class
IOP-overridden method-calling position change	Changes the position or way an overridden method is called
LCR-logical connector replacement	Replaces logical connectors and to or
LOD-logical operator deletion	Deletes logical operators
LOR-logical operator replacement	Replaces logical operators && to ||
ROR-relational operator replacement	Replaces relational operators >To =
SCD-super calling deletion	Deletes a call to a superclass method within a subclass
SCI-super calling insert	Inserts a call to a superclass method within a subclass
SIR-slice index remove	Removes an index from a slice operation in an array or list

- *Mutmut*

Mutmut [13] is a powerful mutation testing tool designed to mutate Python test suites. It simplifies the mutation testing process by allowing programmers to apply found mutants directly to the source code files with a straightforward command. After generating mutations, Mutmut automates the execution of the current test suite against these altered code versions. Mutmut applies mutations on the source code level and allows for configurations such as whitelisting line by line, as shown in Table 3.

Table 3. Mutpy Mutation Operators [13]

Mutmut Mutation Operators:	
number_mutation	Alters numeric values by incrementing them by one
string_mutation	Modifies string literals by inserting 'XX' near the start and end
partition_node_list_mutation	Identifies a split point in a list of nodes based on a specified value
lambda_mutation	Changes lambda expressions by mutating the result to either *0* or *None*
Argument_mutation	Alters arguments of dictionary-style calls by appending 'XX' to the name of the argument
arglist_mutation	Removes selected arguments from argument lists
Keyword_mutation	Switches specific keywords (is, not, in, etc.) to their opposites
Operator_mutation	Alters arithmetic, bitwise, comparison, and assignment operators to their opposites
And_or_mutation	Changes logical operations between *and* and *or*.
Expression_mutation	Alters assignment expressions by changing assigned values to None or empty strings
Decorator_mutation	Mutates decorators by yielding only the final newline

Name_mutation	Changes names for boolean literals
Trailer_mutation	Specifically targets array-like access ([ ])
Subscript_mutation	Alters subscript (index) expressions, like setting the index to *None*

- *Poodle*

Poodle [14] is a mutation testing tool designed to enhance the efficiency and flexibility of mutation testing in Python environments Poodle operates by copying the source code to a temporary location before applying mutations, thus ensuring that the original code remains unaffected by the mutations. Poodle applies mutations on the source code level and has configurations such as whitelisting line by line, and code blocks, as shown in Table 4.

Table 4. Poodle Mutation Operators [14]

Poodle Mutation Operators: Source Code level (with a twist)	
"BinOp"- Binary Operation Mutator	Changes binary operators + to -
"AugAssign"- Augmented Assignment Mutator	Alters augmented assignments += to *=
"UnaryOp"- Unary Operation Mutator	Modifies unary operators -x to +x
"Compare"- Comparison Mutator	Changes comparison operations == to >=
"Keyword"- Keyword Mutator	Alters control flow keywords and to or
"Number"- Number Mutator	Modifies numeric literals 1 to 0
"String"- String Mutator	Changes string literals "hello" to "world"
"FuncCall"- Function Call Mutator	Alters function call arguments or function names
"DictArray"- Dict Array Call Mutator	Modifies dictionary accesses
"Lambda"- Lambda Return Mutator	Changes the return value of lambda functions
"Return"- Return Mutator	Alters the return statements in functions
"Decorator"- Decorator Mutator	Modifies decorators on functions or classes.

- *Mutatest*

Mutatest [15] is a Python-based mutation testing tool which introduces mutations, or small changes, into a program's source code. The tool operates by first scanning and creating an abstract syntax tree (AST) from the source files to identify locations in the code that can be mutated. randomly samples these locations and applies mutations such as altering operators, changing conditions, or modifying assignments. Mutatest then runs the existing test suite against these mutated versions using the corresponding *pycache* files, which ensures that the original source code remains untouched, as shown in Table 5.

Table 5. Mutatest Mutation Operators [15]

Mutatest Mutation Operators: Source Code Level	
AugAssign	Modifies augmented assignment operations: += to minus=
BinOp	Alters binary operators + to -
BinOp Bitwise Comparison	Changes bitwise operations x= a &y to x= a | y
BinOp Bitwise Shift	Adjusts shift operations x << y to x >> y
BoolOp	Modifies Boolean operations if x **and** y to if x **or** y
Compare	Changes comparison operators x >= y to x < y
Compare In	Alters membership checks x in [1,2,3] to x not in [1,2,3]
Compare Is	Changes identity checks x is None to x is not None
If	Replaces conditional checks in if statements with True or False
Index	Mutates index values for list or array accesses x[0] to x[1] or x[-1]
NameConstant	Switches between constants x=True to x=False
Slice	Alters slice boundaries in list or array X[:2] to x[2:]

- *Cosmic Ray*

Cosmic Ray [16] is a powerful mutation testing tool designed to enhance the robustness of the test suite by introducing small alterations (mutations) in a program. Cosmic Ray utilizes the concept of "sessions" to manage a full mutation

testing suite, since mutation testing runs can be time-consuming and may need to be paused or restarted. Cosmic ray introduces mutations on the AST level and allows for custom mutation operators to be created by the user, as shown in Table 6.

Table 6. *Cosmic Ray* Mutation Operators [16]

Cosmic Ray Mutation Operators: AST Level	
Binary_operator_replacement	Replaces binary operators
Boolean_replacer	Replaces Boolean operators
Break_continue	Replaces break with continue and vice-versa
Comparison_operator_replacement	Replaces one comparison with another
Exception_replacer	Modifies exception handlers
Keyword_replacer	Replaces one keyword with another
No_op	An operator that makes no changes
Number_replacer	Modifies numeric constants
Remove_decorator	Removes decorators
Unary_operator_replacer	Changes Unary Operators
Variable_inserter	Replaces usages of named variables to statements
Variable_replacer	Replaces usages of named variables
Zero_iteration_for_loop	Modifies for-loops to have zero iterations

2.3 Challenges and Limitations of Mutation Testing

Mutation testing faces several limitations that hinder its ability to be applied in practical use. One of the challenges is the large number of mutants generated during testing. Each syntactic change in the program code can create multiple mutants, leading to a high execution cost due to the need to test each mutant individually. For instance, a simple program might be mutated in various ways, resulting in a considerable number of test executions to cover all mutants. This high computational cost [17], especially for larger programs, can be prohibitive.

Mutation testing is also limited by the difficulty of identifying equivalent mutants [17], those that behave identically to the original program despite mutations. These equivalent mutants do not provide useful information but still consume resources. The concept of incompetent mutants [17] further complicates the process because they do not functionally alter the program's behavior in a meaningful way and are not representative of real faults. These mutants can skew the results of mutation testing and contribute to a misleading assessment of test suite effectiveness. The process of detecting these mutants often involves substantial manual effort, adding to the overall complexity and cost of mutation testing.

3. EXPERIMENTAL SETUP

We will now go into the details of setting up the environment for the experiments, the research questions, the programs under test and how the test cases were produced, and an explanation of how to format the mutation testing tools.

3.1 Research Questions

RQ1: *What is the difference between the mutation testing tools based on tools and mutation specific criteria?*

Every tool performs mutations and give results, but there are key differences in design, configuration, operators used, application technique, runtime, and displayed results. We want to give a comprehensive comparison to show these differences between all 5 tools.

RQ2: *Which tool is the most effective in creating competent mutants?*

We want to run these tools to determine which makes the most competent mutants in two different environments. We will also be calculating how many equivalent and incompetent mutants were created to calculate a new *competent mutant score:* The number of competent mutants (passed or failed) generated over the total mutants created. This information will tell us which tool utilizes computational storage best which makes for easier alterations of unit tests.

3.2 Programs Under Test

Table 7. Program Under Test Metadata

PUT	PUT Metadata		
	Description	*# of functions*	*# of test cases*
bagels	Small program: a program that asks users to guess a 3 digit number.	2	8
diff-match-patch	Big Program: The Diff Match and Patch libraries offer robust algorithms to perform the operations required for synchronizing plain text.	31	184

As shown in Table 7, both *bagels* [18] and *diff-patch-match* [19] were selected from GitHub. We selected *bagels* because we wanted to generate *mutation-adequate* test suites to have the highest possible mutation adequacy score, and we selected *diff-match-patch* to observe how mutation tools operate on non-mutation-adequate test suites, a larger code base, and more advanced tests.

3.3 Generating Test Cases

A test suite had to be generated for the *Bagels*, so we designed test cases using a comprehensive approach involving branch adequacy and mutation adequacy. This technique of creating mutation-adequate tests [11] was used to achieve the highest possible mutation adequacy score. The process goes as follows:

1. Generate branch adequate tests for a PUT
2. Run the mutation tool, and check for live mutants.
3. Attempt to kill the live mutants by changing the unit tests accordingly.

We did not strive for a perfect mutation score in *Diff-match-patch* unlike *Bagels,* because we wanted to record how mutation tools behave when mutants survive, what mutants were injected, and why tests won't always be perfect.

3.4 Running Mutation Tools

Mutation testing is performed via the terminal, requiring careful setup to ensure the tools function correctly. Organizing the source code and test files is essential, as troubleshooting can become tedious if configurations are not well-structured. While each mutation testing tool has documentation outlining the steps for execution, beginners or those unfamiliar with terminal operations may find the process challenging. Therefore, learning to read and interpret open-source tool documentation is crucial for successful mutation testing. Before running any mutation testing tool, all test cases must pass at 100% using a testing framework such as *pytest* or *unit test*. Next, the tools should be installed ideally within a virtual environment *(venv)*.

Cosmic Ray needs a TOML file to specify the configurations which will include which file programmers want to mutate, which file is the test file, excluded modules for things programmers don't want to test, which *distributor* programmers want mutants to run on which can give programmers the option to run multiple mutations at one time in parallel, and *timeout* which is how long (in seconds) programmers want each mutant to run and try each mutant before moving onto the next mutant, this can greatly alter how long the tests run. Then programmers must set up a session and baseline which prepares the mutations for the code and stores them in a database file. It's crucial to re-run this command if programmers modify the configuration, change the code-under-test, or alter the tests, as these changes affect which mutations and tests are applied. Before running the mutation suite, ensure the test suite passes unmutated code by executing the baseline command confirming that everything functions correctly without any mutations. Cosmic Ray has a lengthy setup process [16].

Poodle is one of the simpler tools, programmers list its name to be able to run the tool, there is no mandatory configuration file, but it is an option. Poodle also uses inline comments; whitelisting, such as *#pragma: no mutate* to exclude singular lines of code and *# nomut: on* excludes certain code blocks from mutation testing and *#nomut: off* to resume testing) depending on where programmers place the command [14]. Mutmut requires whitelisting to be added line by line, using the comment *# pragma: no mutate* to ensure that parts of the code won't mutate. There is also a configuration file that is strongly suggested to set up, even though it is not mandatory because Mutmut will mutate every executable file in a directory [13].

It is important to acknowledge that certain tools, despite proper setup, may not function as expected during the course of this study. Specifically, Mutpy and Mutatest could not be executed successfully. This highlights the potential unreliability of some open-source tools, which may hinder research progress. Recognizing anomalous behavior in a tool's functionality is crucial, as it may signal underlying issues. Addressing these challenges will be a focus of future investigations.

4. EXPERIMENTAL RESULTS

In this section we will go over the mutation testing results, special observations, and answer the research questions, mutators used by different tools have been shown in Table 9.

As shown in Table 8, Poodle's mutation score was the lowest with a 95.6% mutation score with only 1 mutant that was not discovered. In Diff-match-patch, poodle created 2832 Mutants with a total mutation adequacy score of 80.6%.

Mutmut created 31 mutants for *bagels* and 1913 mutants for *diff-match-patch*. Bagels got a 100% mutation adequacy score, and diff-match-patch got an 80.29% mutation adequacy score. Mutmut used 6 different operators across the two PUT's.

Cosmic Ray created the most mutants with 113 mutants generated for *bagels* and 6158 mutants for *diff-match-patch* Cosmic Ray used 9 different operators across both PUT's. The mutation score for bagels was 100% and diff-match-patch was 81.55%.

Table 8. The comparison between *Diff-match-patch* and *Bagels*

Tools	*Diff-match-patch*				*Bagels*			
	Generated	Survived	Killed	Percentage	Generated	Survived	Killed	Percentage
Poodle	2832	548	2284	80.6%	23	1	22	95.6%
Cosmic Ray	6158	1136	5022	81.55%	26	0	26	100%
Mutmut	1913	377	1536	80.29%	31	0	31	100%

Table 9. Mutators used by different tools

Cosmic Ray Used Mutators	Mutmut Used Mutators	Poodle Used Mutators
Binary Mutator	Number mutator	FuncCall
Comparison Replacement Mutators	Expression mutator	AugAssign
Unary mutator	Keyword mutator	BinOp
Boolean mutator	And/or mutator	UnaryOp
Break/continue mutator	String mutator	DictArray
Exception mutators	Operator mutator	Number
Number replacer mutator		String
Zero Iteration Mutator		Keyword
Add/not Mutator		Compare
		Return

RQ1: *Is there a difference between the existing mutation testing tools based on tool and mutation specific criteria?*

Poodle made the least mutants, took the least time, and did not mutate the test file, gives 2 different viewing options for html results, one is the actual source code displayed with each mutation showing up line by line, and another list style result with only failed mutants. Poodle used 10 different mutation operators across *bagels* and *diff-match-patch.*

Mutmut unfortunately did not give any extra information about the killed mutants like which mutants were applied to which areas in its overall mutation score, so the analysis is incomplete. We do not know what operators makes up the successful mutations, what percent is equivalent or incompetent. Mutmut used 6 different operators in *diff-match-patch*, but we were only able to analyze the surviving mutants, so the analysis is incomplete. Mutmut also mutates the test file, even when including the whitelisting onto the file.

Cosmic Ray organizes the mutation results by operators and makes so many because it applies a maximum level of application by applying every single type of operator onto a mutation location. The whitelisting feature on cosmic ray also counted skipped mutants into the total mutant percentage, so this means that to get an accurate calculation programmers need to go in and count the mutants by hand that programmers want included. The amount of mutants cosmic ray produced for diff match patch was almost twice as much as poodle, and 3 times as much as Mutmut, and this could be due to the AST application technique being able to apply more sophisticated mutants. Cosmic ray also utilized its bitwise operators, which added onto the number of mutations created. Most of the mutations turned out to be incompetent, may of the Binary operator mutants were incompetent because every binary operator was applied to each place a binary operator showed up.

In Mutpy's case every single mutant either survived or was incompetent, and the mutation score would be 0%, even with the simplest of programs and tests. There is a bug in the tool's source code, we were not able to locate it.

Mutatest would start to run but would crash a few seconds after that and there would be a few errors which would lead to a bug in programs and tests. A "Population" error would show up, indicating there was a bug in the source code as well.

RQ2: *Which tool creates the most competent mutants?*

As shown in Figure 1, Poodle had 128 mutants timeout which means the mutants were trying to be applied, but exceeded the amount of time a single mutant is allowed to compile. Poodle created 1463 competent mutants, 1415 incompetent mutants, 3 equivalent mutants, generating a 50.9% competency score.

Cosmic ray had no mutants timeout with more mutants over Poodle. 1584 mutants were competent, 25 were equivalent, and 4598 were incompetent, which is a 25.7% competency score.

Competent mutants apply to both surviving and killed mutants because both are important to the mutation adequacy score. As programmers can see in the graph above, Poodle had the higher percentage of competent mutants. We did not include Mutmut in this part of the study because the results of the mutation trial were incomplete.

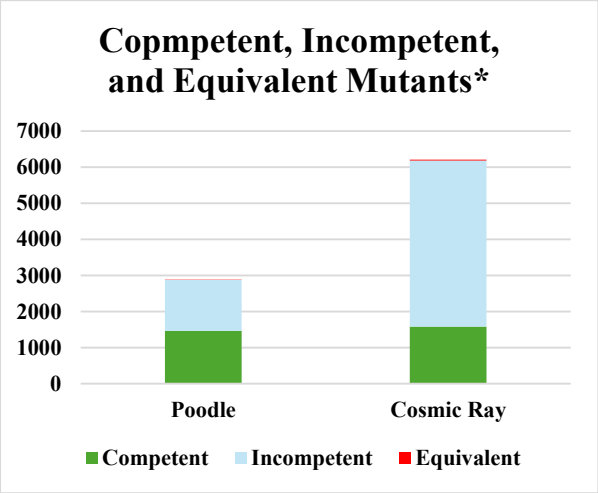

Figure 1. Comparison between Poodle and Cosmic Ray

5. CONCLUSION AND FUTURE DIRECTIONS

Mutation testing is a widely researched software testing technique, though it still faces significant challenges in reaching widespread adoption in industry [4]. One major obstacle is the manual effort required to review both incompetent and equivalent mutants, which involves ensuring that detected mutants align meaningfully with the intended test outcomes.

In this study, we thoroughly analyzed five Python mutation testing tools, examining each tool's architecture, interface, configuration options, and release dates. We tested each tool on both mutation-adequate and non-mutation-adequate test suites, with tests spanning small to very large codebases. Our findings highlight that incompetent mutants significantly impede the progression of mutation testing. Specifically, over half of the mutants generated by Poodle and Cosmic Ray were deemed incompetent. Mutmut also presented incomplete mutation results, while MutPy and Mutatest encountered functional issues, suggesting that improvements and updates are necessary to enhance their effectiveness. Overall, our study indicates there is substantial room for improvement in Python mutation testing tools.

In the future, we hope that developers consider our findings and address the limitations identified in this paper. We also plan to extend our research by testing larger suites, exploring a wider range of mutation operators, and using diverse test case scenarios to gain deeper insights into the reliability of mutation testing tools in Python.

ACKNOWLEDGEMENTS

This work was supported in part by the U.S. National Science Foundation under Grant 2050869 and Grant 2349347. It was also supported by the National Science and Technology Council, Taiwan under Grant No. NSTC 113-2222-E-029-003-MY2. It was partially completed by the first author under the supervision of Professor W. Eric Wong and Mr. Zizhao Chen while attending the Research Experience for Undergraduates (REU) program at the University of Texas at Dallas in Summer 2024.

REFERENCES

[1] Ahamed, S.S., 2010. Studying the feasibility and importance of software testing: An Analysis. *arXiv preprint arXiv:1001.4193*.

[2] Bertolino, A. and Marré, M., 1996. How many paths are needed for branch testing?. *Journal of Systems and Software*, *35*(2), pp.95-106.

[3] Zou, Y., Li, H., Li, D., Zhao, M. and Chen, Z., 2024, March. Systematic Analysis of Learning-Based Software Fault Localization. In *2024 10th International Symposium on System Security, Safety, and Reliability (ISSSR)* (pp. 478-489). IEEE.

[4] Offutt, J., 2011. A mutation carol: Past, present and future. *Information and Software Technology*, *53*(10), pp.1098-1107.

[5] Mathur, A.P. and Wong, W.E., 1993, October. Evaluation of the cost of alternate mutation strategies. In *Anais do VII Simpósio Brasileiro de Engenharia de Software* (pp. 320-334). SBC.

[6] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y. and Harman, M., 2019. Mutation testing advances: an analysis and survey. In *Advances in computers* (Vol. 112, pp. 275-378). Elsevier.

[7] Wong, W.E. ed., 2001. *Mutation testing for the new century* (Vol. 24). Springer Science & Business Media.

[8] Sánchez Jerez, A.B., Delgado Pérez, P., Medina Bulo, I. and Segura Rueda, S., 2022. Mutation testing in the wild: findings from GitHub. *Empirical Software Engineering, 27, 1-35.*

[9] Derezińska, A. and Hałas, K., 2014. Analysis of mutation operators for the python language. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland* (pp. 155-164). Springer International Publishing

[10] Ahmed, Z., Zahoor, M. and Programmersnas, I., 2010, February. Mutation operators for object-oriented systems: A survey. In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)* (Vol. 2, pp. 614-618). IEEE

[11] Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E. and Malevris, N., 2016, October. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (pp. 147-156). IEEE.

[12] "MutPy," *PyPI*, Nov. 17, 2019. https://pypi.org/project/MutPy, last available in November 2024

[13] "mutmut - python mutation tester — mutmut documentation." https://mutmut.readthedocs.io/en/latest last available in November 2024
[14] "Poodle documentation." https://poodle.readthedocs.io/en/latest/index.html, last available in November 2024
[15] "Mutatest: Python mutation testing — Mutatest 3.1.0 documentation." https://mutatest.readthedocs.io/en/latest/index.html, last available in November 2024
[16] "Cosmic Ray: mutation testing for Python — Cosmic Ray documentation." https://cosmic-ray.readthedocs.io/en/latest/index.html, last available in November 2024
[17] Nguyen, Q.V. and Madeyski, L., 2014. Problems of mutation testing and higher order mutation testing. In *Advanced Computational Methods for Knowledge Engineering: Proceedings of the 2nd International Conference on Computer Science, Applied Mathematics and Applications (ICCSAMA 2014)* (pp. 157-172). Springer International Publishing.
[18] Sweigart, A., 2021. *The Big Book of Small Python Projects: 81 Easy Practice Programs*. No Starch Press, pp. 1-5.
[19] google. (n.d.). *GitHub - google/diff-match-patch: Diff Match Patch is a high-performance library in multiple languages that manipulates plain text.* GitHub. https://github.com/google/diff-match-patch, last available in November 2024