A Rich High-Order Mutation Testing Dataset for Software Fortification

Van-Nho Do[1], Giang T.C. Tran[2], Duc-Thuan Nguyen[3], Ngoc-Anh Nguyen-Thi[4], Quang-Vu Nguyen[5], Thanh-Binh Nguyen[6]

[1] Le Quy Don High School for the Gifted, Danang, Vietnam
[2] University of Quebec in Trois-Rivières, Canada. The University of Danang, University of Economics, Vietnam
[3] University of Engineering and Technology, Vietnam National University of Hanoi, Vietnam
[4] University of Engineering and Technology, Vietnam National University of Hanoi, Vietnam
5 The University of Danang, Vietnam-Korea University of Information and Communication Technology, Vietnam
[6] The University of Danang, Vietnam-Korea University of Information and Communication Technology, Vietnam.

Correspondence: Quang-Vu Nguyen, nqvu@vku.udn.vn
Communication: received 26 Jul 2024, revised 8 Nov 2024, accepted 1 Dec 2024
DOI: 10.32913/mic-ict-research.v2025.n1.1277

***Abstract*: High-order mutation (HOM) testing is a rigorous technique for evaluating the effectiveness of test suites by introducing mutations with multiple concurrent faults into the source code. In this study, we present the development and analysis of a comprehensive dataset tailored for HOM testing purposes. The dataset comprises 2,839,792 instances categorized into Survived and Killed classes, representing instances correctly identified as surviving and not surviving the mutation testing process, respectively. We employ four prominent machine learning algorithms—Logistic Regression, Random Forest Classifier, LightGBM, and XGBoost—to classify instances within these categories. Experimental results demonstrate varying levels of accuracy, precision, recall, and F1-score across the algorithms, with LightGBM and XGBoost exhibiting superior performance. These findings underscore the importance of high-quality datasets in facilitating effective HOM testing and provide valuable insights into the capabilities of machine learning algorithms in this context.**

Keywords: *High order mutation testing, dataset, data generation, machine learning.*

I. Introduction

Software testing plays a pivotal role in ensuring the reliability and robustness of modern software systems. Among the abundance of testing methodologies, mutation testing stands out as a rigorous technique for assessing the quality of test suites by injecting artificial faults, or mutants, into the source code. This method, dating back to the pioneering work of DeMillo et al. in the 1970s [1], has since evolved into a powerful tool for evaluating the effectiveness of test cases in detecting faults [2].

While traditional mutation testing typically focuses on introducing single faults into the codebase, recent advancements have led to the development of HOM testing techniques, capable of generating mutants with multiple simultaneous faults [3]. These techniques offer a more comprehensive assessment of test suite quality by exploring the interactions between faults and the ability of tests to detect them [3].

The availability of comprehensive datasets comprising diverse mutants and their associated test cases is a significant factor influencing the effectiveness of HOM testing. However, the success of HOM testing is not exclusively contingent upon data-driven methodologies; a plurality of algorithmic approaches contributes to the overall efficacy of HOM techniques [4]. Such datasets are essential for evaluating the effectiveness of HOM testing tools and algorithms, as well as for benchmarking their performance against existing techniques [4]. However, the advancement of HOM testing is hampered by a lack of suitable datasets.

In this paper, we address this need by presenting the results of our efforts to generate a novel dataset specifically tailored for HOM testing using the MutPy tool. Our dataset aims to provide researchers and practitioners with a valuable resource for advancing the state-of-the-art in mutation testing. Furthermore, this dataset comprises mutants generated using over 20 mutation operators applied to a representative set of software programs selected from diverse domains based on code complexity, programming language diversity, and domain representation. This ensures a broad range of code characteristics.

Our approach to dataset generation follows a systematic and rigorous methodology, leveraging the MutPy mutation testing framework. We carefully selected a representative set of software programs from various domains and applied a comprehensive set of mutation operators to create a

diverse population of mutants. Additionally, we curated a collection of test suites designed to exercise different aspects of the software under test, ensuring thorough coverage of its functionality and behavior.

The selection of software programs and their associated test suites prioritized three key criteria to ensure a representative dataset. First, popularity within the software community was considered to focus on widely used and relevant applications. Second, the reliability of each program was assessed based on its GitHub star rating, indicating community validation and maturity. Finally, the availability of comprehensive test suites was essential to adequately exercise diverse aspects of the software under test, allowing for thorough mutation analysis. This multi-faceted approach aimed to create a robust and representative dataset for evaluating mutation testing techniques.

Throughout this paper, we provide detailed insights into our methodology for dataset generation, as well as a comprehensive description of the dataset itself. We evaluate the quality and usefulness of the dataset through various experiments and comparisons with existing datasets, demonstrating its efficacy in facilitating HOM testing research and development.

In summary, our work contributes to the ongoing evolution of mutation testing techniques by providing a valuable resource that empowers researchers and practitioners to explore new frontiers in HOM testing. We believe that our dataset will serve as a catalyst for innovation and collaboration in the field of software testing, ultimately leading to the development of more robust and reliable software systems. The rest of the paper is structured as follows. The Related Work section reviews existing literature, recent advancements in HOM testing and current datasets. The Data Generation Process section details the methodology employed to create the dataset, including the use of MutPy and various mutation operators. Following this, the Dataset Evaluation section presents an in-depth analysis of the generated dataset. Finally, the Conclusion summarizes the findings, emphasizes the importance of the dataset for advancing mutation testing research, and suggests potential future work in this domain.

II. Related Work

Mutation testing has been a subject of extensive research, with numerous advancements made in recent years. In this section, we review the latest publications related to HOM testing and data generation using tools like MutPy.

Recent studies have focused on advancing mutation testing techniques to address the limitations of traditional mutation testing. For example, Dang et al. introduced a novel approach for HOM testing, leveraging advanced mutation operators and mutation strategies to generate mutants with multiple faults simultaneously [5]. Their study demonstrated the effectiveness of HOM testing in improving test suite quality and fault detection capability.

In the realm of data generation for mutation testing, the use of automated tools such as MutPy has gained significant attention. MutPy, a mutation testing tool for Python programs, has been widely adopted for generating mutation datasets due to its flexibility and extensibility [4]. Recent study by Dakhel et al. explored the capabilities of MutPy for generating diverse sets of mutants and corresponding test cases, showcasing its effectiveness in facilitating mutation testing research and development [6].

Furthermore, efforts have been made to enhance the mutation testing process by integrating advanced techniques for mutation generation and test case generation. For instance, the study by [7] et al. proposed a novel framework that combines HOM testing with evolutionary algorithms to optimize test case generation for improved fault detection.

Tushar Swamy et al. [8] proposed Homunculus, which is a compiler for efficient machine learning pipelines in network data planes. Addressing limitations of traditional network management, it simplifies machine learning model deployment by automatically generating optimized code from high-level specifications (using the Alchemy interface). Microbenchmarks show superior performance and resource efficiency compared to hand-tuned baselines. Future work focuses on co-compilation with traditional network applications and improving dataset quality and retraining speed. Le Van Phoi and Nguyen Thanh Binh [9] address the performance bottleneck of mutant generation in Lustre mutation testing. It proposes and evaluates a multi-threading approach for parallel generation of both first-order and higher-order mutants, demonstrating significant (nearly 50%) time improvements. Future work suggests exploring machine learning techniques for further optimization.

Additionally, research has been conducted on the evaluation and benchmarking of mutation testing techniques. Studies by [10] et al. and [11] et al. investigated the effectiveness of different mutation operators and mutation strategies in detecting faults and improving test suite quality [10, 11]. These studies provide valuable insights into the strengths and limitations of mutation testing approaches, informing the development of more robust mutation testing methodologies.

Our findings show that previous studies on mutation score prediction mainly rely on the source code of projects [12], or are combined with test suite metrics [13]. It can be said that up to now, there has not been a dataset that

includes mutants of projects. This is the direction of our research.

While these studies have made significant contributions to the field of mutation testing, there remains a need for further research on the generation of comprehensive datasets for HOM testing. Our work builds upon these efforts by presenting a novel dataset specifically tailored for HOM testing, generated using the MutPy tool. By incorporating advanced mutation operators and comprehensive test suites, our dataset aims to provide researchers and practitioners with a valuable resource for advancing the state-of-the-art in mutation testing.

III. Data Generation Process

1. HOM generation strategies background

Mutation testing is a well-established technique in software testing for evaluating the effectiveness of test suites. Traditional mutation testing involves the introduction of single faults, or mutations, into the source code to assess the ability of the test suite to detect these faults. However, as software systems become more complex, there is a growing need for mutation testing techniques capable of generating mutants with multiple simultaneous faults. This is where the HOM strategy of MutPy comes into play.

MutPy was chosen for dataset generation due to its suitability for our research goals. Its established position as a widely-used Python mutation testing tool, coupled with robust support for HOM testing—including the ability to generate mutants with multiple simultaneous faults—aligned perfectly with our need for a comprehensive and complex dataset. We leveraged MutPy's extensibility to implement a custom "ALL_PAIR" HOM strategy, further enhancing the dataset's scope. The tool's integration with scikit-learn provided essential machine learning capabilities for data analysis and model evaluation. MutPy's focus on Python, the language of our target codebase, ensured seamless integration and facilitated the reproducibility of our results.

Mathematically, let M denote the set of mutation operators and n represent the number of faults to be introduced into the codebase. The HOM strategy involves selecting n mutation operators from the set M and applying them simultaneously to create a mutant. This process can be represented by the combination formula:

$$C(|M|, n) = \frac{|M|!}{n! \times (|M| - n)!}. \quad (1)$$

This formula calculates the total number of distinct n-order mutants that can be generated. Each n-order mutant is created by applying n different mutation operators simultaneously from the set of M operators. By systematically combining mutation operators, MutPy's HOM strategy enables the generation of mutants with varying degrees of complexity, providing a more comprehensive assessment of test suite quality. This approach helps uncover subtle faults and vulnerabilities that may go undetected by traditional mutation testing techniques.

The HOM strategy of MutPy also incorporates advanced mutation operators tailored to target specific aspects of the codebase. These mutation operators cover a wide range of programming constructs and language features, ensuring thorough mutation testing coverage. Additionally, MutPy's HOM strategy allows for the integration of HOM testing with other mutation testing techniques, providing testers with the flexibility to customize their mutation testing approach based on project requirements and constraints. Overall, MutPy's HOM strategy represents a significant advancement in mutation testing methodologies, empowering testers to identify and address potential weaknesses in their test suites more effectively.

2. Dataset Generation Process

The Algorithm 1 defines the AllPairHOMStrategy class, which is responsible for generating second-order mutations by combining compatible first-order mutations. It iterates through the list of mutations, selects a pair of mutations, and yields a combination of these mutations to create a second-order mutant. This code should be integrated into the controller.py file of MutPy as part of the mutation testing process. Additionally, other code snippets are integrated into the MutPy source code to extract important mutation attributes and output them to a CSV file.

The dataset generation process for HOM testing using MutPy involves several steps, primarily utilizing MutPy for mutation extraction and coverage analysis.

Step 1: Utilizing MutPy for Mutation Extraction. In the first step, MutPy is employed to extract attributes of mutations. However, the built-in HOM strategies of MutPy typically generate a limited number of second-order mutants, approximately half the number of first-order mutations. To address this limitation, a custom HOM Strategy named ALL_PAIR is implemented to generate second-order mutants by combining compatible first-order mutants. This custom strategy significantly increases the number of second-order mutants that can be generated, providing a more comprehensive dataset for evaluation.

Step 2: Mutation Attribute Extraction. Following mutant generation, essential attributes of mutations are extracted to create a dataset. These attributes include:

- typeStatement: The type of statement mutated, extracted based on the mutated node or its parent nodes.

Algorithm 1 AllPairHOMStrategy class

```
import random
class AllPairHOMStrategy(HOMStrategy):
    name = 'ALL_PAIR'
    def __init__(self, *args, shuffler=random.shuffle, **kwargs):
        super().__init__(*args, **kwargs)
        self.shuffler = shuffler
    def generate(self, mutations):
        size_of_mutations = len(mutations)
for i in range(size_of_mutations) do
            first_mutations = []
            first_mutations.append(mutations[i])
            available_mutations = []
    for j in range(i + 1, size_of_mutations) do
                available_mutations.append(mutations[j])
    end for
            self.remove_bad_mutations(first_mutations, available_mutations)
    for mutation in available_mutations do
                mutations_to_apply = []
                mutations_to_apply.append(mutations[i])
                mutations_to_apply.append(mutation)
                yield mutations_to_apply
    end for
end for
```

- typeOperator: The mutation operator used, extracted from the mutation.operator field in the code.
- typeReturn: The return type of the mutated operator, obtained from the mutation.node.
- distanceBetweenTwoMutants: The distance between two mutated nodes in the abstract syntax tree (AST), calculated using the lowest common ancestor (LCA) method.
- line and end_line: The starting and ending lines of the mutation.
- depthOnAST: The depth of the mutant on the AST.
- result: The outcome of the mutation when tested, categorized as killed, survived, incompetent, or timeout.

Step 3: Code Coverage Analysis. Next, code coverage analysis is performed using the coverage tool. This involves running the test suite with coverage to determine which lines of code are covered. Additionally, each test case in the test suite is executed individually with coverage to assess its code coverage.

Step 4: Additional Attribute Generation. Based on the results of MutPy and coverage analysis, five additional attributes are generated:

- ancestor: Indicates whether two first-order mutants have a parent-child relationship.
- start_line_coverage: Determines if the starting line of the mutation is covered by the test suite.
- start_line_test_coverage: Calculates the percentage of tests covering the starting line of the mutation.
- body_coverage: Measures the overall code coverage of the mutated code body.
- body_test_coverage: Calculates the percentage of tests covering each line of the mutated code body.

By following these steps, a comprehensive dataset is generated for high-order mutation testing, facilitating the evaluation and improvement of test suites and mutation testing tools.

3. Dataset Description

Table I encapsulates the results of HOM testing conducted using the MutPy framework across various software modules. Each module is assessed based on several key metrics, including the number of survived mutants, killed mutants, timeouts, instances categorized as incompetent, and a specific metric denoted as Nummodulo.

A ”killed” mutant refers to a mutated version of the code that is successfully detected by the test cases. This means that the test cases, when executed against the mutated code, identify the introduced fault and fail, indicating that the test cases are effective in detecting this specific type of mutation. Conversely, a ”survived” mutant represents a

mutated version of the code that is not detected by the test cases. This implies that the test cases, when executed against the mutated code, do not identify the introduced fault and pass, indicating that the test cases are ineffective in detecting this specific type of mutation.

Survived mutants are those that remain undetected after the test cases have been executed. High numbers of survived mutants might indicate that the test cases are not thorough or effective enough to catch certain types of faults. This could be due to insufficient test coverage, focusing on specific code paths, or failing to address certain types of errors.

Opposite to Survived mutants, Killed mutants are the mutants that are detected by the test cases. A high number of killed mutants indicates that the test cases are effective in finding faults, thus demonstrating good coverage and robustness. This suggests that the test cases are capable of identifying a wide range of potential errors, leading to a higher confidence in the software's quality.

The classification of mutants into "killed" and "survived" categories is crucial for evaluating the effectiveness of test cases in detecting software faults [14]. The mutation score, often calculated as the percentage of killed mutants, provides a quantitative measure of test case effectiveness, guiding researchers and developers in improving the quality and coverage of their test cases.

Timeouts are instances where test cases take too long to execute and exceed a predefined time limit. This indicates performance issues or infinite loops within the software. Incompetent mutants are the mutants that are invalid or cause the program to crash immediately, suggesting that the mutation operators might need refinement to produce more realistic and useful mutations. Nummodulo is a metric representing the number or type of mutation operations applied during the testing process. This reflects the diversity and range of mutations tested, which is crucial for comprehensive coverage.

The *AirBnB_clone* module exhibits a relatively low count of 6 survived mutants, suggesting a limited capacity to withstand alterations, juxtaposed with a higher count of 30 killed mutants, indicating instances where the software failed to meet expected outcomes. Notably, this module demonstrates no occurrences of timeouts or incompetence.

Conversely, the *asynctest* module demonstrates robustness with a substantial count of 79,978 survived mutants, although it encounters occasional timeouts, as evidenced by 253 occurrences. Additionally, a notable number of instances, 6,939, are categorized as incompetent, indicating areas where the software may require optimization or refinement.

The *avocado* module showcases impressive resilience with over 1.2 million survived mutations, yet it is plagued by a significant count of timeouts and instances of incompetence, pointing towards potential areas for improvement.

Similarly, the *connect_four* module exhibits a blend of successes and challenges, with notable counts in both survived and killed mutants, alongside occurrences of timeouts and incompetence.

The *cpython3.7* module stands out with an extensive dataset, indicating substantial testing efforts, with millions of survived and killed mutants, alongside occurrences of timeouts and incompetence.

Overall, the dataset[1] provides a comprehensive snapshot of the performance and robustness of each software module under the rigorous scrutiny of HOM testing, offering insights into areas of strength and opportunities for enhancement in software quality and reliability.

IV. Dataset Evaluation

1. Evaluation metrics

In this study, we employed several evaluation metrics to assess the performance of the machine learning algorithms in classifying instances within the HOM testing dataset. These metrics provide valuable insights into the algorithms' effectiveness in correctly classifying instances across different categories.

Accuracy: Accuracy measures the proportion of correctly classified instances among all instances in the dataset. It provides an overall assessment of the algorithm's classification performance and is calculated as the ratio of the number of correctly classified instances to the total number of instances.

Precision: Precision measures the proportion of correctly identified instances among those predicted to belong to a particular category. It focuses on the algorithm's ability to avoid false positives and is calculated as the ratio of the number of true positive instances to the total number of instances predicted to belong to the positive category.

Recall (Sensitivity): Recall, also known as sensitivity, quantifies the proportion of correctly identified instances among all instances that actually belong to a particular category. It assesses the algorithm's ability to identify all positive instances and is calculated as the ratio of the number of true positive instances to the total number of instances belonging to the positive category.

F1-score: The F1-score is the harmonic mean of precision and recall, offering a balanced measure of a classifier's performance. It takes into account both false positives and

[1] https://rb.gy/1928xn

Table I
MUTANTS GENERATED BY MUTPY

Module	Survived	Killed	Timeout	Incompetent	Nummodulo
AirBnB_clone	6	30	0	0	4
asynctest	79978	0	253	6939	2
avocado	1287486	26426	15448	219190	15
callee	187537	0	0	36335	9
connect_four	12452	89592	8662	5080	4
cpython3.7	13373132	12227565	3118778	2792673	73
cricri	682	9067	4796	1386	3
green	309015	178806	19082	12920	8
nose2	36722	0	0	4784	4
RinnBlackJackPro	951269	20177	5536	89989	1
virtue	198693	0	8	19551	3

false negatives and is calculated as the harmonic mean of precision and recall. A higher F1-score indicates better overall performance.

2. Predictive machine learning models

To rigorously assess the suitability of the newly generated dataset for machine learning-based HOM testing, we employed four machine learning classification algorithms and evaluated their performance using evaluation metrics in section 4.1. These metrics provide valuable insights into the algorithms effectiveness in correctly classifying instances, while the successful classification across diverse algorithms provides strong empirical evidence of the dataset's internal consistency, absence of significant bias, and overall structural integrity—essential characteristics for a reliable HOM testing resource. This multifaceted evaluation, therefore, directly demonstrates the dataset's utility and robustness for future research in data-driven HOM testing methodologies.

Logistic Regression: Logistic Regression is a classic linear model widely used for binary classification tasks [15]. Despite its simplicity, Logistic Regression provides interpretable results and is computationally efficient, making it a popular choice for baseline classification tasks.

Random Forest Classifier: Random Forest Classifier is an ensemble learning method that constructs multiple decision trees during training and combines their predictions through voting or averaging [16]. It excels in handling high-dimensional data and is robust against overfitting, thanks to its built-in mechanism of feature randomness and bagging.

LightGBM: LightGBM is a gradient boosting framework developed by Microsoft that uses a novel technique called Gradient-Based One-Side Sampling (GOSS) to handle large-scale datasets efficiently [17]. It partitions the data in a depth-wise manner and can handle categorical features directly, making it highly efficient and accurate for classification tasks. Additionally, LightGBM supports GPU acceleration, enabling faster training times and scalability to larger datasets.

XGBoost: XGBoost, or Extreme Gradient Boosting, is another popular gradient boosting framework known for its scalability, speed, and performance [18]. It employs a regularized objective function and parallel tree boosting to achieve high accuracy and generalization on various datasets. XGBoost's advanced regularization techniques and parallel computing capabilities make it suitable for large-scale classification tasks with complex data structures.

3. Experimental Results

Table II shows the experimental results, provides a granular analysis of the performance metrics of four distinct machine learning algorithms—Logistic Regression, Random Forest Classifier, LightGBM, and XGBoost—across two pivotal categories: "Survived" and "Killed". These categories represent the outcomes of a binary classification task, where "Survived" denotes instances correctly identified as surviving and "Killed" represents instances correctly identified as not surviving.

In the "Survived" category, there are 1,633,032 instances for Logistic Regression, 1,633,844 instances for Random Forest Classifier, 1,634,353 instances for LightGBM, and 1,633,032 instances for XGBoost. Similarly, in the "Killed" category, there are 1,205,089 instances for Logistic Regression, 1,205,901 instances for Random Forest Classifier, 1,204,580 instances for LightGBM, and 1,205,901 instances for XGBoost.

Across the algorithms, we observe varying levels of accuracy, precision, recall, and F1-score, which offer nuanced insights into their classification capabilities. Logistic Regression achieves an accuracy of 0.81. In comparison, Random Forest Classifier achieves slightly higher accuracy scores of 0.85, and moving to more advanced algorithms, LightGBM and XGBoost exhibit even higher accuracy levels, with LightGBM achieving 0.90 and XGBoost achieving 0.91.

Analyzing precision, recall, and F1-score metrics provides further insights into algorithm performance. Light-

Table II
EXPERIMENTAL RESULTS

Algorithm	Accuracy	Kind of SOM	Amount	Precision	Recall	F1-score
Logistic Regression	0.81	Survived	1,633,844	0.27	0.79	0.54
		Killed	1,205,089	0.27	0.70	0.50
Random Forest Classifier	0.85	Survived	1,633,032	0.23	0.85	0.71
		Killed	1,205,901	0.23	0.78	0.67
LightGBM	0.90	Survived	1,634,353	0.90	0.94	0.92
		Killed	1,204,580	0.91	0.85	0.88
XGBoost	0.91	Survived	1,633,032	0.91	0.94	0.93
		Killed	1,205,901	0.92	0.88	0.90

GBM demonstrates remarkable precision values of 0.90 and 0.91 for ”Survived” and ”Killed” categories, respectively, indicating a high proportion of correctly classified instances among those predicted to belong to a particular category. Similarly, XGBoost exhibits impressive recall values of 0.94 and 0.88 for ”Survived” and ”Killed” categories, respectively, showcasing its ability to correctly identify a substantial proportion of instances belonging to each category.

By employing these established classification algorithms, the suitability of the generated dataset for machine learning applications is evaluated. The results provide insights into key dataset characteristics, including size and completeness (a large dataset with minimal missing values is desired), data quality (the absence of errors, inconsistencies, and bias), and transparency (a detailed description of the dataset creation process, encompassing selection criteria, data collection methods, and preprocessing steps). These findings inform the development of robust criteria for evaluating the quality of datasets for mutation testing research.

V. CONCLUSION

The dataset presented herein contributes significantly to the advancement of research in the field of HOM testing. By meticulously generating a comprehensive dataset using MutPy and coverage analysis, this research provides a valuable resource for researchers and practitioners interested in evaluating and enhancing the effectiveness of mutation testing techniques.

One of the primary contributions of this dataset lies in its focus on HOM testing, a cutting-edge approach that introduces mutants with multiple simultaneous faults. Traditional mutation testing techniques often overlook complex faults that can only be detected through HOM testing. By generating mutants with varying degrees of complexity, this dataset enables researchers to explore the effectiveness of HOM testing in uncovering subtle faults and vulnerabilities in software systems.

Moreover, the dataset encompasses a diverse set of mutation operators and strategies, including custom strategies like the AllPairHOMStrategy, designed to overcome limitations in existing mutation testing tools. This diversity allows researchers to evaluate the performance of different mutation operators and strategies and identify the most effective approaches for mutation testing.

This dataset provides a valuable resource for researchers in the Software Engineering (SE) community to advance their understanding and development of HOM testing techniques. Researchers can utilize the dataset to evaluate the effectiveness of existing HOM testing tools and algorithms, benchmark their performance against existing techniques, and explore new approaches for generating and analyzing mutants. The dataset's rich collection of mutants, categorized as ”Survived” and ”Killed,” allows researchers to investigate the impact of various mutation operators and strategies on test suite quality. Furthermore, the dataset can be used to train and evaluate machine learning models for predicting mutation outcomes, thereby improving the efficiency and effectiveness of HOM testing.

Overall, this dataset serves as a valuable resource for advancing the state-of-the-art in mutation testing and software quality assurance. By providing researchers with access to a diverse and meticulously curated dataset, this research aims to foster collaboration, innovation, and knowledge sharing in the field of HOM testing and beyond.

While the dataset presented here offers valuable insights into HOM testing and data generation, it is essential to acknowledge its limitations and identify areas for future research.

One limitation of the dataset is its reliance on MutPy for mutation generation and coverage analysis. While MutPy is a versatile and widely used tool, it may have inherent limitations in its mutation operators and strategies, which could impact the diversity and effectiveness of the generated mutants. Future research could explore alternative mutation testing tools or develop custom mutation operators to enhance the diversity and quality of mutants generated.

In terms of future work, we will explore novel mutation operators and strategies tailored to specific application domains or programming paradigms and focus on developing automated techniques for mutation testing, such as machine

learning-based approaches for mutation generation and test case prioritization. Additionally, enhancing the dataset's scope and documentation to improve its generalizability for broader mutation testing research should be focused. This includes expanding the dataset to encompass a wider variety of software projects and programming paradigms, providing a more detailed analysis of the dataset's statistical properties, and thoroughly documenting all data collection, preprocessing, and selection criteria. Furthermore, we plan to investigate the application of the dataset to different mutation testing techniques and explore the development of more robust evaluation metrics beyond the mutation score.

REFERENCES

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

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[3] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 147–158.

[4] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[5] X. Dang, D. Gong, X. Yao, T. Tian, and H. Liu, "Enhancement of mutation testing via fuzzy clustering and multi-population genetic algorithm," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2141–2156, 2021.

[6] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Information and Software Technology*, p. 107468, 2024.

[7] Y. Li, W. Shen, T. Wu, L. Chen, D. Wu, Y. Zhou, and B. Xu, "How higher order mutant testing performs for deep learning models: A fine-grained evaluation of test effectiveness and efficiency improved from second-order mutant-classification tuples," *Information and Software Technology*, vol. 150, p. 106954, 2022.

[8] T. Swamy, A. Zulfiqar, L. Nardi, M. Shahbaz, and K. Olukotun, "Homunculus: Auto-generating efficient data-plane ml pipelines for datacenter networks," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 329–342.

[9] N. T. Binh *et al.*, "Optimizing mutant generation for lustre programs with multi-threading," in *2020 5th international conference on innovative technologies in intelligent systems and industrial applications (CITISIA)*. IEEE, 2020, pp. 1–5.

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

[11] Q. Zhu, A. Panichella, and A. Zaidman, "A systematic literature review of how mutation testing supports quality assurance processes," *Software Testing, Verification and Reliability*, vol. 28, no. 6, p. e1675, 2018.

[12] Y. Gil and D. Ma'ayan, "Better prediction of mutation score," *Authorea Preprints*, 2023.

[13] K. Jalbert and J. S. Bradbury, "Predicting mutation score using source code and test suite metrics," in *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*. IEEE, 2012, pp. 42–46.

[14] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[15] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[16] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[17] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[18] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

Van-Nho Do heads the Informatics group at Le Quy Don Gifted High School, Da Nang, Vietnam. He is a doctoral candidate in Computer Science at Da Nang University of Technology, with a research specialization in advanced mutation testing. His research interests include software engineering, which he applies to his leadership role in education.

Email: dovannho@gmail.com

Giang T.C. Tran is a postdoctoral researcher at the University of Quebec in Trois-Rivières, specializing in the application of artificial intelligence to improve and enhance information systems. She completed her Ph.D. in Computer Science and Engineering at Chung-Ang University (2023) and holds a B.S. (with honors) in Management Information Systems from Da Nang University of Economics (2019). Her research utilizes data mining, machine learning, and logical reasoning techniques.

Email: thi.chau.giang.tran@uqtr.ca

Duc-Thuan Nguyen is a fourth-year student majoring in Information Technology at the University of Engineering and Technology, Vietnam National University, Hanoi.
Email: thuanbn03@gmail.com

Ngoc-Anh Nguyen Thi is a fourth-year student majoring in Information Technology at the University of Engineering and Technology, Vietnam National University, Hanoi.
Email: hannibalvera1412@gmail.com

Quang-Vu Nguyen is PhD in field of Computer Science from Wroclaw University of Science and Technology, Poland and currently is Head of the Department of Science – Technology and International Cooperation at Vietnam-Korea University of Information and Communication Technology, the University of Danang. His research focus is on artificial intelligence, software engineering, data science, software quality assurance, and testing.
Email: nqvu@vku.udn.vn

Thanh-Binh Nguyen graduated in Information Technology from the University of Danang - University of Science and Technology in 1997. He received PhD. degree in Information Technology at Grenoble Institute of Technology, France in 2004. He has been qualified as Associate Professor since 2013. He is currently working at the University of Danang - Vietnam-Korea University of Information and Communication Technology. His research interests include software engineering and software quality.
Email: ntbinh@vku.udn.vn