International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2025: 7.089

Mutation Testing with Machine Learning Approaches for Fuzzing, Smart Contracts, and Automated Repair: A Critical Survey

Sobhan Kumar Bedajna[1], Sudipta Roy[2]

[1]Research Scholar, Assam University, Silchar, India
Corresponding Author Email: *sa_exam[at]aus.ac.in*

[2]Professor, Department of Computer Science and Engineering, Assam University Silchar, India

Abstract: ***This study critically examines mutation testing enhanced by machine learning across fuzzing systems, browser-based applications, integer overflow detection in smart contracts, and heuristic-based automated program repair. The analysis synthesizes recent literature to identify methodological trends, strengths, and unresolved limitations. Comparative evaluation indicates that reinforcement learning improves input scheduling and mutation efficiency but often lacks semantic understanding and computational scalability. Domain-specific mutation strategies demonstrate effectiveness in vulnerability detection yet remain limited in the process of generalization. The study highlights the need for intelligent mutation frameworks that integrate semantic modelling and cost-aware execution strategies. The findings guide the development of scalable, context-aware mutation testing methodologies.***

Keywords: Mutation testing, fuzzing, reinforcement learning, computational scalability, browser testing, vulnerability detection

1. Introduction

The principal aim of software testing is to dig out hidden errors, which may occur at runtime or within the program's control flow structure. Typically, test suites or test oracles are used to detect these issues, serving as a form of validation. In contrast, mutation testing introduces artificial bugs into the programme and then applies test oracles to determine whether these injected defects can be detected.

This study focuses mainly on reviews and exploring literatures in the domain of mutation testing specifically on machine learning approaches in case of fuzzing, End-to-End testing of browse-based applications, vulnerability detection in case of integer overflow in Ethereum Smart Contracts (ESCs) and Automated Program Repair(APR) in heuristic approaches. In recent years, Artificial Intelligence (AI) and Machine Learning (ML) methods have been used extensively to identify errors within complex code structures. Since, mutation technique processes are time consuming, comparatively higher in cost parameters, and also difficult to deploy, AI and ML procedures for evaluation of mutants in different areas of coding can give a better result compared to age old techniques.

The rest of the paper is organised as follows- Section 2 is about discussions on the four segments based on which this study is executed. Section 3 contains the discussions on the work done on the considered segments. Section 4 contains the concluding remarks and research gap.

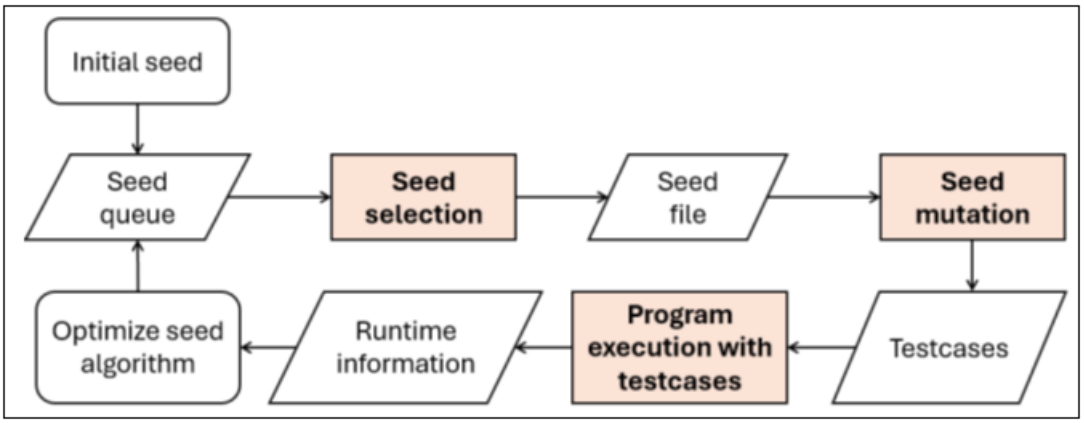
2. Fuzzy Method

The goal is to prioritize inputs that reach code segments which are less explored and aiming to increase the coverage of deeper program areas.

2.1 On Fuzzy Sets

With ongoing advancements and research, the modern fuzzing techniques are generally categorized as (a) black-box fuzzing, (b)white-box fuzzing, and (c)grey-box fuzzing. Among these approaches, grey-box fuzzing is widely used for its balance between efficiency and flexibility [1]. For this reason, the researchers chose to concentrate on grey-box fuzzing in the development of their model [2].

The coverage-guided fuzzing process for an application involves several key stages, which are illustrated in the figure below. Each phase has a crucial role in the overall fuzzing workflow. Functions and purposes of these stages are described as follows [3-7]:



1) First, an input value (seed) is provided, which the fuzzing tool places into a queue of input values[7].
2) From this queue, the fuzzing tool selects an input value for mutation and each fuzzing tool employs different algorithms for this selection. The effectiveness of these seed selection algorithms significantly influences the fuzzing process's efficiency.
3) Next, using the chosen input value, the fuzzing tool generates various test cases by applying specific actions to the input (e.g., flipping bits, adding bits, shifting bits, etc.). The choice of actions and the effort devoted to this process also impact the fuzzing process's performance.
4) Subsequently, the fuzzing tools test the generated inputs with the target program, collecting runtime information (program crashes, execution time, code coverage, etc.).
5) Using the collected information, the fuzzing tool decides whether to continue, remove, replace, or adjust the inputs in the input queue. When a crash is detected, the fuzzing tool records the corresponding test case along with detailed crash information so that the fuzzing operator can analyse it later. The process then returns to step 2 and the cycle continues.

- **Creating test suites**: Researchers generally apply static and dynamic analysis techniques to derive information from the executable files and create an input that can achieve extensive program survey. This is different from using a random input or a pre-defined input class. In cases where the input involves typical data structures (file formats, etc.), some research employs artificial intelligence to find the data structure and form test cases that can bypass initial structural blocks of the program [8-11].
- **Input preference and mutation**: When multiple good test suites are examined and kept aside for subsequent steps, the selection of the next input for transformation, the choice of transformation operations, and the amount of effort provided in testing that input significantly influence the model's performance. The focus is on reducing redundant changeover efforts, quickly selecting accurate changeover to increase code coverage, thereby enhancing the fuzzing process's efficiency [8-11].
- **Post-fuzzing evaluation**: In certain programs, the number of false positives found during fuzzing can be high, or duplicate vulnerabilities might be discovered. This can be cumbersome for the analysts. Therefore, researchers also address this by implementing algorithms to detect duplicates, evaluating the return of detected vulnerabilities, or employing machine learning algorithms to score the exploitability of vulnerabilities [8-11].

Yue et al. proposed EcoFuzz [9], where input scheduling was depicted as a Multi-Armed Bandit (MAB) problem, and an enhanced alternative was introduced as the Adversarial Multi-Armed Bandit model to improve its performance. The common idea of both techniques was to enhance the input selection and thereafter scheduling, prioritizing inputs to get a higher likelihood of containing errors and leading to better code coverage. Ji et al. introduced AFLPro [12], which enhanced input selection and scheduling by combining static analysis with a basic block synthesis model.

The issue with using static analysis techniques always lies in the lack of complete runtime data and often produces low-accurate or false-positive results. Additionally, it may not work effectively on applications which utilizes code obfuscation or packing mechanisms.

Chen et al. created Matryoshka [13], used taint analysis to solve conditional statements and penetrate deeper into the program. However, this technique demanded significant resources and slowed down the fuzzing process. Taint analysis also faced challenges with under-tainting and over-tainting. Zhang et al. proposed a lightweight and convenient mechanism to excel input checks by combining static analysis with mutating key bytes in InsFuzz [14]. They identified bytes influencing conditional statement results and then mutate them.

Recently, with the huge advancement of artificial intelligence and machine learning techniques, researchers are also using these methods to enhance the fuzzing process. Surveys [15] indicate that the use of machine learning methods to fuzzing is diverse as well as creative, yielding promising results. The steps typically addressed by artificial intelligence include- (1) selection of input, (2) scheduling of input, (3) generation of input, (4) mutation action selection, etc. RapidFuzz [16] and CGFuzzer [17] employ Generative Adversarial Networks (GANs) to learn the complex structural inputs, aiming to generate higher similarity patterns for fuzzing protocols or specific file formats. This approach helps in saving time by avoiding mutating invalid samples and increasing the likelihood of passing structure checks. NeuFuzz by Wang et al. [18] modelled the bug-finding process akin to natural language processing, utilizing deep-learning Long Short-Term Memory (LSTM) for grasping the arrangement of error containing paths, predicting which paths were more likely to

have errors and prioritizing them for input scheduling. The Reinforcement Learning (RL) approaches have been applied also to fuzzing for the first by Böttinger et al. [19]. The authors transformed the fuzzing problem into an RL problem, where the selection of the next mutation action is analogous to choose the next move in a chess game. The proposed model was specifically designed for PDF files, lacking objective results when compared to modern fuzzing tools and not addressing the balance between exploitation and exploration. Kuznetsov et al. also employed deep Q-learning to choose action of mutations for application testing [20]. They demonstrated that combining RL can reduce the time needed to create expected test cases by up to 30%, yet their evaluation method does not suit real-world applications.

Reddy et al. improved mutation action selection using the Monte Carlo Control algorithm, creating more valid samples for applications with complex input structures [21]. The results enhanced the rate of passing structure checks for samples. However, their model skewed towards exploitation rather than exploration, focusing on generating diverse inputs with similar features instead of exploring new behaviours.

Liu et al. introduced Reinforcement Compiler Fuzzing [22], which also utilized deep Q-learning for mutation selection at the compilation level. The implementation requires the source code to work effectively. Drozd and Wagner combined Deep Double Q-learning to select mutation actions and accelerate libFuzzer [23]. However, they acknowledged that it is not sufficient and that further enhancements are needed in terms of input selection and filtering.

Zhang et al. proposed rlfuzz [10], a method to balance exploitation and exploration in a deep-Q-learning fuzzing models by randomly selecting trial inputs for subsequent transformations when the model did not experience an increase in code coverage. However, this selection method was not yet optimal, as inputs with low code coverage in the queue still have an equal chance of being transformed as those with higher potential.

In a different approach, Wang et al. [5][7] utilized RL for input scheduling rather than selecting mutation actions like other studies. They proposed a multilevel code coverage model to enhance fuzzing detail and introduced a scheduling mechanism to support this multilevel code coverage model using RL[10][19]. The results showed a balance between exploitation and exploration in the generated test cases, but there was no significant improvement in selecting more effective mutation actions. Current combined RL and fuzzing solutions tend to focus heavily on designing RL algorithms, states, rewards, and parameters without considering factors like the balance between exploitation and exploration [22]. This leads to RL fuzzing models delving deep into one code branch, missing opportunities to find vulnerabilities in other branches.

Moreover, the focus often lies solely on mutation action selection, without mechanisms for effective input selection and scheduling that are crucial for real-world fuzzing tools.

Furthermore, no RL fuzzing study provides a comparative perspective on performance, strengths, and weaknesses compared to modern fuzzing tools [10][22]. The work investigated and proposed an RL-based guided coverage aware fuzzing model to address weaknesses by integrating it with an effective input selection and scheduling algorithm. Additionally, the authors proposed a multi-level input transformation algorithm that can be applied to RL-based fuzzing models, coupled with a waste reduction mechanism to improve model efficiency.

2.2 On browse-based application.

Although tools for mutation testing are available for various programming languages but there are no ready-made solutions for managing the complete mutation testing process for entire browser-based implementations within the scope of End-to-end(E2E) testing [24]. E2E validation of browser-based implementations is a form of black-box testing that relies on the fundamentals of mutation.

2.2.1 E2E and Mutta Framework

A test action represents a series of actions applied on the browser-based implementations being tested. For Example, providing a username and a password, and then pressing the login button. From a single test scenario, multiple test oracles can be produced by defining the specific input data for each step and specifying the expected results through declarations.

The running of the test oracles may be programmed by using programming language, such as Java or Python, and using a dedicated E2E testing model that can control a web surfer in a manner similarly how human being interacts with it (Leotta et al.) [25].

When multiple E2E test suites or testing frameworks are available for browser-based applications, it will be crucial to get a systematic process for impartially examining the effectiveness in detecting errors. Such evaluations also help in deciding which test oracle or model to accept [24][26][27][28]. These types of situations are occurring frequently in real life. For instance, managers may face situations such as: (1) evaluating different E2E techniques or tools to determine which is most effective at uncovering errors, or (2) needing to lower the volume of a test oracle because its full execution is too time-consuming [29][30]. In these cases, a balance must be struck between execution time and the error-unfolding capacity of the validation suite.

MUTTA, a new framework was designed for making the mutation testing process automated in case of browser-based applications. This model can apply mutations to the several components of a web application in server-side, generating multiple versions of mutants, each having an exclusive mutation. After generating the mutants, this model executes the E2E test cases under evaluation against the marked browser-based implementations and collects the test results [25][26][28]. To assess the effectiveness of MUTTA, the authors, Leotta et al. conducted a case study aimed at equating two separate processes to E2E web testing: (1) test suits based on traditional contention, and (2) test suites exploiting distinctive testing.

Differential testing (McKeeman, 1998) [31] for web applications involves comparing the running page under test

with a reference exposure from an earlier page-version that was assumed to be correct. This approach shows as an alternative to traditional assertions for detecting regressions that arise during the advancement of the browser-based applications under consideration [24-28].

2.3 Overflow of integers

The study finds out vulnerabilities present for integer overflow and proposes specialized mutation process for addressing integer overflows in Ethereum smart contracts (ESCs).

2.3.1 vulnerability On Integer overflow

Integer overflow is the most common vulnerability in Ethereum Smart Contracts (ESCs). Kalra et al. [32] found that One thousand ninety five (1095) out of One thousand Five Hundred Sixty Four (1,564) smart contracts that contains integer overflow issues. The vulnerability arised for two main reasons. First, the Ethereum Virtual Machine (EVM) uses only two fifty six-bit integers, whereas solidity supports multiple type of integers, considering eight-bit and sixteen-bit integers. As a result, Solidity integers were not always strictly mapped to EVM types. Second, both EVM and Solidity, lack strict handling of overflow of integers. At the time of overflow, the program never shows an exception rather continues execution of consequent instructions, potentially leading to unexpected behaviour.

Thus, integer overflow testing vulnerabilities in smart contracts is very critical. Currently, most research in this area focuses primarily on detecting these vulnerabilities within smart contracts [32]. Whatsoever, most of the research on examining smart contract testing methodology still remains imperfect. Also, most ESCs live in the shape of bytecodes. Therefore, the total number of smart contract source codes which are applicable for evaluation and testing in realm are very less. Also, in the existing smart contract, it is very difficult to label integer vulnerabilities manually.

It is important to mention here that mutation testing can productively address all these challenges. Some of the researchers [33] suggested to use mutation testing to validate test suites and testing procedures although there were no comprehensive guideline for mutation operators in case of integer overflow. Based on this, the authors surveyed integer overflow susceptibilities in exhaustively used Solidity smart contracts on Ethereum and suggested five typical mutation operators [33][34].

The study first classifies errors in integer overflow in smart contracts into three categories**:** overflow in arithmetic data type, overflow in case of truncation, and overflow in signed data type. It then proposes 5- mutation operators designed to target integer faults of these types. The operators consist of both new operators introduced specifically in this study and existing operators from traditional mutation testing.

Overflow in Arithmetic data type is the most familiar type of integer overflow issue. It typically arises from the lack of boundary checks during arithmetic operations and can manifest as either overflow or underflow. Overflow happens when the value assigned to a variable is declared as integer exceeds its upper limit.

In truncation overflow, Solidity integers from eight (8) to two hundred fifty six (256) bits, expand in step-up of 8 bits. Allocating a smaller-bit-width integer to a larger-bit-width variable does not result in overflow. However, providing a larger-bit-width integer to a smaller-bit-width variable leads to truncation, resulting in a reduced value.

Signed overflow occurs due to mismatches between the type of the assigned value and the type of the target variable, and it can happen in two main cases-

1) *Signed-to-unsigned conversion*: The sign bit represents the highest bit of a signed integer. When a signed integer is allocated to an unsigned integer of the similar size, the value can overflow, producing a large positive number.
2) *Unsigned-to-signed conversion*: Assigning an unsigned integer to a signed integer of the similar size can also result in overflow, potentially producing an incorrect negative or wrapped value.

Now, using five mutation operators, the authors Fioraldi et al. developed an integer overflow mutation tool that can automatically inject mutations for integer overflow. The authors collected and labelled Forty smart contracts from open source containing vulnerabilities in case of integer overflow with 2,099 generated mutants, and conducted a comprehensive study. The outcome demonstrated that the operators proposed produce effective mutants but also provided a powerful support for enhancing the adequacy of testing of ESC. Both tools and the dataset are accessible on GitHub [35].

A pragmatic survey was performed on forty open-source Ethereum Smart Contracts (ESCs). The outcome demonstrated that (1) the suggested mutation operators were capable of reproducing all vulnerabilities present as integer overflow in original contracts, with the created mutants exhibiting a compilation pass rate as high [17]; and (2) these mutants were able to reveal weaknesses in existed testing methods in case of detecting integer overflow vulnerabilities, and thus delivered valuable base for enhancing the comprehensiveness and effectiveness of test oracles.

Smart contracts, first suggested by Szabo [36], summarize specified states and transformation rules, enabling the automatic delivery of contract schema and corresponding responses under certain conditions. When these delivery criteria were met, the smart contract executed spontaneously, producing results according to the logic defined in its code.

Blockchains are well-suited for smart contracts because of their distribution, stability, and trackability. Today, numerous blockchain platforms, including Ethereum [34] and Bitcoin [37], bear smart contract functionality. Ethereum, in particular, offers a comprehensive programming language and a suite of development methods, making it a popular base for creating Decentralized Applications (DApps).

The Ethereum blockchain features two types of accounts- (1) external accounts and (2) contract accounts. External accounts are generated by individuals and managed via

private key, while contract accounts are generated when a contract is implemented and are primarily governed by the contract code. The operations like transferring Ether and invoking contracts by sending transactions to the appropriate contract address can be performed by both types of accounts [38-40].

The commonly used smart contract programming language on Ethereum is Solidity, which is therefore the focus of this study. Solidity is a Turing-complete language with a common syntax as JavaScript. Programmers write smart contracts in Solidity, compile it using EVM bytecode, and implement all on the Ethereum blockchain via transactions.

Security vulnerabilities in smart contracts arise for several reasons. First, Solidity was a comparatively new language, and its design may still have drawbacks. Second, many programmers are inexperienced with Solidity, making it easy to introduce errors when coding smart contracts. Third, once implemented, smart contracts on the blockchain are immutable; even susceptibilities are discovered, the contract cannot be patched or modified, leaving the errors permanent.

Ethereum smart contracts (ESCs), however, are often vulnerable and immutable once deployed, making pre-deployment testing a critical event. In this context, mutation testing can be used as a dominant approach to enhance the quality and reliability of ESC testing before deployment on the blockchain. Kalra et al. [32] explained that integer overflow was very critical and frequently occurring problem in smart contracts. The outcome of their survey provided that one thousand ninety five out of one thousand five hundred sixty four smart contracts are having integer overflow issue.

2.4 Heuristics approach

Heuristic-based approaches select operators for mutation randomly, which can lead to over-exploration of certain operators that tend to generate more failing program variants. This result in more incorrect patches and wasted computational resources, reducing the overall efficiency and effectiveness of the repair process.

2.4.1 On Heuristic based study

Fixing software errors are typically manual and time-consuming task, sometimes exceeding the time allocated to the programmers [41],[42]. In a fast-paced industry, this frequently resulted in the implement of error-prone products to meet fixed delivery deadlines [12]; [43]. Research in Automated Program Repair (APR) seeks to handle this challenge by automated the method of generating proper patches for software errors.

The categorization of Automated Program Repair (APR) method varies in the survey of literatures. As per the survey by Le Goues et al. [44-50], APR techniques can be categorized as following**.**

- *Based on constraint* –This approach leverages the semantics of the faulty program to generate constraints and then synthesize restorations that satisfies the need (Xuan et al. [51]).
- *Based on learning*-- End-to-end restoration techniques forecast patches by learning features from erroneous programme segments and the corrected developer-written versions (Chen et al. [52][53]).
- *Based on heuristic approaches*-- APR has been the oldest and, so far, the widely adopted in industry. Examples include its application in a medical management system (Haraldsson et al. [54]), large-scale automatic end-to-end repair at Meta (Marginean et al. [55]), and fixing regularly happening errors at Bloomberg (Kirbas et al. [56]).

Heuristic-based APR uses searching approaches for navigating the area of software development [57][58]. The process needs a test suite for the faulty program to judge the accuracy of the programme variants. In APR, the suite consists of test oracles along with minimum one wrong test oracle which finds the error. The process of repair typically begins with fault localization technology to correctly pointed out the faulty region of the code. A code variant is then generated by selecting a suspicious location and applying a mutation operator, with the most commonly used operators like deletion, insertion, or replacement of code fractions.

This process produces a huge number of candidate variants, which are evaluated against the test suite to ascertain the fitness. The procedure is iterated again and again till a patch is found which satisfies all test cases, effectively fixing the bug.

To overcome this, Smigielska et al. [59] suggested one selection criterion, where the probability of choosing operators is proportionate to the size of its search area. Also, Soto et al. and Le Goues [60][61] leveraged programmer's error-fixing histories to assign a more balanced and informed distribution of mutation operators.

To determine which strategies could increase the productivity and efficiency of search-based APR, the authors carried out a literary survey. Their review concentrated on two key regions relevant to the study- mutation operators and the use of machine learning techniques for automatically repair of programmes and improvement of genetics**.**

The literary survey was carried out using four major computer science search engines**:** IEEE Xplore, ACM Digital Library, ScienceDirect, and the DBLP Computer Science Bibliography**.**

All searches were performed on 25th April 2024. The scope for pertinence included conference papers, workshop papers, journal articles, and PhD theses published as on date, covering the fields of of genetic, incremental programming, and algorithms for genetics.

The first meta data used in the search was the exact phrase "mutation operators". After an initial relevancy screening, the next filter was applied to identify papers that specifically addressed mutation rate alteration. Upon completing this primary search stage, the authors performed snowballing by examining the references of the relevant papers, which yielded twenty four more papers that met the study's adoption criteria.

A second search was carried out using the exact phrases "machine learning" and "program repair", applying the same

scope of relevancy. To identify the most relevant papers, the focus was on studies which used machine learning to enhance the procedure of finding source-level repairable aspirants in the software. This initial search resulted in 20 papers, among which only those applying machine learning for mutation operator selection were specifically related to progressive computation.

3. Discussion on these Models

Overall, Pham et al. [6] proposed a coverage-guided fuzzing model that leverages reinforcement learning for automated software vulnerability detection. The algorithm selected simple inputs and used scheduling strategies to balance investigation and utilization of the software. By combining multi-level input mutation with reinforcement learning, the system generated diverse input sequences, improving code coverage. The resulting CTFuzz model achieved an execution speed of 28.5 seconds**.**

The MUTTA framework supported end-to-end (E2E) mutation testing in software applications. MUTTA automatically mutates the server-side source files of a browser-based application, executed the E2E test oracle on the mutated versions, and finds the results. It was evaluated using web applications to compare assertion-based and differential testing approaches. In total, over 15,000 mutants were generated and 87,000 test runs were performed. The results showed that approximately 95% of the mutations were effectively handled, demonstrating MUTTA's dependability and efficacy in E2E mutation testing.

The study by Sun et al. [63] introduced a mutation testing approach for detecting integer overflow vulnerabilities in programs. An experimental survey on 40 open-source Ethereum smart contracts evaluated the proposed mutation operators. The operators successfully reproduced all 179 known integer overflow vulnerabilities and achieved a high compilation success rate. Additionally, they highlighted limitations in existing testing techniques, helping to improve their effectiveness. The overall vulnerability detection rate was 42%, indicating room for further improvement.

Hanna et al. [64] suggested a method for choosing mutation operators in heuristic-based program repair using reinforcement learning. This approach generated more test-passing variants but it did not notably increase the number of errors actually fixed. With a median of 29 and an average of 96 successful variants, the method achieves a 45.1% success rate**.**

4. Research Gap and Conclusion

This part emphasizes on the lacuna and uncovered area of the studies and conclusion at the end.

4.1 Research Gap

The coverage-guided fuzzing model enhances exploration but lacks semantic understanding of the source code and does not analyse internal logic, resulting in incomplete vulnerability detection. The Mutta framework suffers from high computational costs due to large-scale test executions and redundant mutant generation, limiting its efficiency in real-world applications. The existing mutation testing approach for integer overflow is domain-specific, targeting only the Ethereum blockchain, and cannot be generalized to other software systems. The automated programme in heuristics approach produces a huge number of candidate variants, which are evaluated against the test suite to ascertain the fitness of the test suite. Additionally, all models produce a limited number of test-passing mutants, although these changes rarely result in actual bug fixing and reflecting poor semantic repair capability. Also, these approaches do not include semantic mutation modelling and at the same time does not focus on computational cost reduction.

4.2 Conclusion

This study critically synthesizes machine learning driven mutation testing across fuzzing, browser validation, smart contract security, and automated program repair. The analysis demonstrates that while learning based mutation improves exploration efficiency, challenges remain in semantic reasoning, scalability, and domain generalization. The efficiency of each model relies heavily on the availability of high-quality, well-labelled mutant datasets; any inherent limitations or biases in these datasets can reduce detection accuracy.

Future research should prioritize hybrid semantic mutation models, standardized benchmarking datasets, and cost-aware execution strategies. These directions can advance mutation testing toward robust industrial applicability. Further, the research could explore the integration of Transformer-based architectures to improve mutant detection, optimize cost-reduction strategies, and incorporate explainable AI to enhance transparency in decision-making. Additionally, the proposed methodology could be extended to automated testing tools and applied across a wider range of domains beyond traditional software engineering, thereby increasing its practical impact.

References

[1] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, ''The art, science, and engineering of fuzzing: A survey,'' *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.
[2] F. Rustamov, J. Kim, J. Yu, and J. Yun, ''Exploratory review of hybrid fuzzing for automated vulnerability detection,'' *IEEE Access*, vol. 9, pp. 131166–131190, 2021.
[3] X. Zhu, S. Theyn, S. Camtepe, and Y. Xiang, ''Fuzzing: A survey for roadmap,'' *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–36, Jan. 2022.
[4] S. Mallissery and Y.-S. Wu, ''Demystify the fuzzing methods: A comprehensive survey,'' *ACM Comput. Surv.*, vol. 56, no. 3, pp. 1–38, Mar. 2024.
[5] X. Zhao, H. Qu, J. Xu, X. Li,W. Lv, and G.-G.Wang, ''A systematic review of fuzzing,'' *Soft Comput.*, vol. 28, no. 6, pp. 5493–5522, Mar. 2024.
[6] M. Böhme, V.-T. Pham, and A. Roychoudhury, ''Coverage-based greybox fuzzing as Markov chain,'' *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.

[7] J. Wang, C. Song, and H. Yin, ''Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing,'' in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021.
[8] H. Liang, X. Pei, X. Jia,W. Shen, and J. Zhang, ''Fuzzing: State of the art,'' *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018.
[9] T. Yue, P.Wang, Y. Tang, E.Wang, B. Yu, K. Lu, and X. Zhou, ''EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit,'' in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, Aug. 2020, pp. 2307–2324.
[10] Z. Zhang, B. Cui, and C. Chen, ''Reinforcement learning-based fuzzing technology,'' in *Proc. 14th Int. Conf. Innov. Mobile Internet Services Ubiquitous Comput. (IMIS)*. Cham, Switzerland: Springer, 2020, pp. 244–253.
[11] J. Ye, R. Li, and B. Zhang, ''RDFuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation,'' *Math. Problems Eng.*, vol. 2020, pp. 1–12, Mar. 2020.
[12] T. Ji, Z.Wang, Z. Tian, B. Fang, Q. Ruan, H.Wang, andW. Shi, ''AFLPro: Direction sensitive fuzzing,'' *J. Inf. Secur. Appl.*, vol. 54, Oct. 2020, Art. no. 102497.
[13] P. Chen, J. Liu, and H. Chen, ''Matryoshka: Fuzzing deeply nested branches,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 499–513.
[14] H. Zhang, A. Zhou, P. Jia, L. Liu, J. Ma, and L. Liu, ''InsFuzz: Fuzzing binaries with location sensitivity,'' *IEEE Access*, vol. 7, pp. 22434–22444, 2019.
[15] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, ''Fuzzing vulnerability discovery techniques: Survey, challenges and future directions,'' *Comput. Secur.*, vol. 120, Sep. 2022, Art. no. 102813.
[16] A. Ye, L. Wang, L. Zhao, J. Ke, W. Wang, and Q. Liu, ''RapidFuzz: Accelerating fuzzing via generative adversarial networks,'' *Neurocomputing*, vol. 460, pp. 195–204, Oct. 2021.
[17] Z. Yu, H. Wang, D. Wang, Z. Li, and H. Song, ''CGFuzzer: A fuzzing approach based on coverage-guided generative adversarial networks for industrial IoT protocols,'' *IEEE Internet Things J.*, vol. 9, no. 21, pp. 21607–21619, Nov. 2022.
[18] Y. Wang, Z. Wu, Q. Theyi, and Q. Wang, ''NeuFuzz: Efficient fuzzing with deep neural network,'' *IEEE Access*, vol. 7, pp. 36340–36352, 2019.
[19] K. Böttinger, P. Godefroid, and R. Singh, ''Deep reinforcement fuzzing,'' in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2018, pp. 116–122.
[20] A. Kuznetsov, Y. Yeromin, O. Shapoval, K. Chernov, M. Popova, and K. Serdukov, ''Automated software vulnerability testing using deep learning methods,'' in *Proc. IEEE 2nd Ukraine Conf. Electr. Comput. Eng. (UKRCON)*, Jul. 2019, pp. 837–841.
[21] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, ''Quickly generating diverse valid test inputs with reinforcement learning,'' in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Oct. 2020, pp. 1410–1421.
[22] X. Liu, R. Prajapati, X. Li, and D.Wu, ''Reinforcement compiler fuzzing,'' in *Proc. ICML Workshop*, 2019.
[23] *LibFuzzer—A Library for Coverage-Guided Fuzz Testing*. Accessed: Dec. 2023. [Online]. Available: https://llvm.org/docs/LibFuzzer.html
[24] Leotta, M., Clerissi, D., Ricca, F., Tonella, & P.: Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013)*, IEEE, pp. 272–281. https:// doi. org/ 10. 1109/ WCRE. 2013.
[25] Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2016). Approaches and tools for automated end-to-end web testing. *Advances in Computers, 101*, 193–237. https:// doi. org/ 10. 1016/ bs. adcom. 2015. 11. 007
[26] Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2014). Visual vs. DOM-based Theyb locators: An empirical study. In: Casteleyn, S., Gustavo Rossi, M. W. (eds.), *Proceedings of 14th International Conference on web Engineering (ICTHEY 2014)* (vol. 8541), LNCS, Springer, pp. 322–340. https:// doi. org/ 10. 1007/
[27] Leotta, M., Ricca, F., Stoppa, S., & Marchetto, A. (2022). Is NLP-based test automation cheaper than programmable and capture & replay? In: Vallecillo, A., Visser, J., Pérez-Castillo, R. (eds.), *Proceedings of 15th International Conference on the Quality of Information and Communications Technology (QUATIC 2022)* (vol. 1621), CCIS, Springer, pp. 77–92. https:// doi. org/ 10. 1007/ 978-3- 031- 14179-9_6
[28] Leotta, M., Paparella, D., & Ricca, F. (2022). Comparing the effectiveness of assertions with differential testing in the context of web testing. In: Vallecillo, A., Visser, J., Pérez-Castillo, R. (eds.), *Proceedings of 15th International Conference on the Quality of Information and Communications Technology (QUATIC 2022)* (vol. 1621), CCIS, Springer, pp. 108–124. https:// doi. org/ 10. 1007/978-3- 031- 14179-9_8
[29] Olianas, D., Leotta, M., & Ricca, F. (2022). SleepReplacer: A novel tool-based approach for replacing thread sleeps in selenium webdriver test code. *Software Quality Journal (SQJ), 30*, 1089–1121. https:// doi. org/ 10. 1007/ s11219- 022- 09596-z
[30] Olianas, D., Leotta, M., Ricca, F., Biagiola, M., & Tonella, P. (2021). STILE: A tool for parallel execution of E2E webtest scripts. In *Proceedings of 14th IEEE International Conference on Software Testing, Verification and Validation (ICST 2021)*, IEEE, pp. 460–465. https:// doi. org/ 10. 1109/ ICST4 9551. 2021. 00060
[31] McKeeman, W. M. (1998). Differential testing for software. *Digital Technical Journal, 10*(1), 100–107.
[32] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, ZEUS:Analyzing safety of smart contracts, in Network and Distributed System Security Symp., San Diego, CA, USA, doi: 10.14722/ndss.2018.23092.
[33] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen, Mutation testing for ethereum smart contract, arXiv preprint arXiv: 1908.03707, 2019.
[34] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, http://gavwood.com/Paper.pdf, 2014.
[35] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. (2020). *American Fuzzy Lop Plus Plus (AFL++)*. [Online]. Available:https://github.com/AFLplusplus/AFLplusplus

[36] N. Szabo, Smart contracts: Building blocks for digital markets, https://kameir.com/smart-contracts/, 1996.
[37] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, https://bitcoin.org/en/bitcoin-paper, 2008.
[38] Z. X. Li, H. R. Wu, J. H. Xu, X. Y. Wang, L. M. Zhang, and Z. Y. Chen, MuSC: A tool for mutation testing of ethereum smart contract, in Proc. 34th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), San Diego, CA, USA, 2019.
[39] C. F. Torres, J. Sch¨utte, and R. State, Osiris: Hunting for integer bugs in ethereum smart contracts, in Proc. 34th Ann. Computer Security Applications Conf. (ACSAC), San Juan, PR, USA, 2018.
[40] Y. Hirai, Formal verfication of Deed contract in Ethereum name service, https://yoichihirai.com/deed.pdf, 2016
[41] Weiß, C., Premraj, R., Zimmermann, T., Zeller, A.: How long will it take to fix this bug? MSR (2007)
[42] Böhme, M., Soremekun, E.O., Chattopadhyay, S., Ugherughe, E., Zeller, A.: Where is the bug and how is it fixed? An experiment with practitioners. ESEC/FSE, pp. 117–128 (2017)
[43] Tassey, G.: The economic impacts of inadequate infrastructure for software testing. NIST (2002)
[44] Le Goues, C., Pradel, M., Roychoudhury, A.: Automated program repair. Communications of the ACM (2019)
[45] Le Goues, C., Wemer, W., Forrest, S.: Representations and operators for improving evolutionary software repair. GECCO, pp. 959–966 (2012)
[46] Le Goues, C., Nguyen, T.V., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE TSE **38**(1), 54–72 (2012)
[47] Le, X.-B.D.: Towards efficient and effective automatic program repair. ASE (2016)
[48] Motwani, M., Soto, M., Brun, Y., Just, R., Le Goues, C.: Quality of automated program repair on real world defects. IEEE TSE **48**(2), 637–661 (2022)
[49] Smith, E.K., Barr, E.T., Le Goues, C., Brun, Y.: Is the cure worse than the disease? Overfitting in automated program repair. ESEC/FSE, pp. 532–543 (2015)
[50] Whitacre, J.M., Pham, T.Q., Sarker, R.A.: Use of statistical outlier detection method in adaptive evolutionary algorithms. GECCO (2006)
[51] Xuan, J., Martinez, M., Demarco, F., Clément, M., Lamelas, S., Durieux, T., Berre, D.L., Monperrus, M.:Nopol: automatic repair of conditional statement bugs in java programs. IEEE TSE **43**, 34–55 (2018)
[52] Wu, D., Shen, B., Chen, Y., Jiang, H., Qiao, L.: Automatically repairing tensor shape faults in deep learning programs. Inf. Softw. Technol. **151**, 107027 (2022)
[53] Chen, L., Pei, Y., Pan, M., Zhang, T., Wang, Q., Furia, C.A.: Program repair with repeated learning. IEEE TSE (2022)
[54] LNCS, pp. 159–165 (2021) https:// doi. org/ 10. 1007/ 978-3- 030- 88106-1_ 12/ FIGUR ES/1 Haraldsson, S.O., Woodward, J.R., Brownlee, A.E.I., Siggeirsdottir, K.: Fixing bugs in your sleep: how genetic improvement became an overnight success. GECCO, pp. 1513–1520 (2017).
[55] Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A., Scott, A.: Sapfix: Automated end-to-end repair at scale. ICSE-SEIP (2019).
[56] Kirbas, S., Windels, E., Mcbello, O., Kells, K., Pagano, M., Szalanski, R., Nowack, V., Winter, E., Counsell, S., Bowes, D., Hall, T., Haraldsson, S., Woodward, J.: On the introduction of automatic program repair in bloomberg. IEEE Software (2020).
[57] Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann Publishers Inc., 340 Pine Street, 6th Floor, San Francisco, CA 94104 (2004).
[58] Koza, J.R.: Genetic programming as a means for programming computers by natural selection. Stat. Comput. **4**(2), 87–112 (1994).
[59] Smigielska, M., Blot, A., Petke, J.: Uniform edit selection for genetic improvement: empirical analysis of mutation operator efficacy. International Workshop on GI, 1–8 (2021).
[60] Soto, M., Le Goues, C.: Using a probabilistic model to predict bug fixes. SANER 2018-March, pp. 221–231 (2018).
[61] Soto, M.: Improving patch quality by enhancing key components of automatic program repair. ASE, pp.1230–1233 (2019).
[62] Geethal, C., Bohme, M., Pham, V.T.: Human-in-the-loop automatic program repair. IEEE Trans. Softw. Eng. **49**, 4526–4549 (2023). https:// doi. org/ 10. 1109/ TSE. 2023. 33050 52.
[63] Zhu, Q., Sun, Z., Xiao, Y.A., Zhang, W., Yuan, K., Xiong, Y., Zhang, L.: A syntax-guided edit decoder for neural program repair. ESEC/FSE, pp. 341–353 (2021).
[64] Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of chatgpt. In: Proceedings - 2023 IEEE/ACM International Workshop on Automated Program Repair, APR 2023, pp. 23–30 (2023) https:// doi. org/ 10. 1109/ APR59 189. 2023. 00012