

Enhancing Smart Contract Security through Mutation-Aware Test Generation using SAL-GA

Rupendra Jaiswal¹, Sakshi Srivastava²

¹ M.Tech. (CSE) Scholar, Department of Computer Science and Engineering, S. R. Institute of Management and Technology Lucknow, Uttar Pradesh, India

² Assistant Professor, Department of Computer Science and Engineering, S. R. Institute of Management and Technology Lucknow, Uttar Pradesh, India

Abstract- Blockchain technology and smart contracts have transformed digital transactions by enabling secure, transparent, and decentralized execution of agreements without intermediaries. However, ensuring the reliability and security of smart contracts remains a critical challenge due to their immutable nature and potential vulnerabilities. This study explores smart contract testing with a focus on mutation testing techniques to evaluate test suite effectiveness. To address limitations such as high computational cost and inefficient mutant selection, a novel approach named SAL-GA (Self-Adaptive Learning Genetic Algorithm) is proposed for automated test case generation. The method integrates heuristic-based initialization and genetic optimization to improve code coverage and vulnerability detection. Experimental results demonstrate that SAL-GA achieves high performance with 98.1% code coverage, 98.7% vulnerability detection rate, and reduced execution time compared to existing methods. The findings highlight the effectiveness of the proposed approach in enhancing smart contract security and reliability in decentralized finance applications.

Key Words: Blockchain, Smart Contracts, Mutation Testing, Test Case Generation, Genetic Algorithm

1. INTRODUCTION

Blockchain technology has caused a significant change in how transactions are carried out and agreements are upheld in recent years. The technological revolution is centered around smart contracts, which are self-executing contracts where the terms are encoded directly into code. Envision a scenario where contractual agreements are automated, transactions are carried out smoothly, and trust is built without intermediaries. This represents the potential of smart contracts and Blockchain technology. Through immutability and decentralization, smart contracts improve blockchain application security, reliability, transparency, and tamper resistance. Smart contract code and logic cannot be altered after deployment, even by its originator. Tampering or malicious updates that could undermine the application's integrity are eliminated. Users and developers may trust the contract to behave as intended. This reduces unexpected change exploits.

1.1 Smart Contracts And Blockchain Technology

Satoshi Nakamoto introduced a set of regulations for establishing the blockchain framework in the year 2008. Specifically, he formulated the initial digital currency, a cryptographic form of money that emphasizes security levels [1]. The inception of Bitcoin cryptocurrency occurred in 2009, marking its commencement in market trading. Its widespread acceptance was a direct outcome of utilizing the Bitcoin blockchain infrastructure. Bitcoin stands as the pioneering platform that successfully eliminated centralized authority [2]. In the Ethereum blockchain network, a smart contract is a computer code that contains the specifics of an agreement between anonymous individuals [3]. This agreement does not require a centralized system or an external enforcement mechanism. It is the smart contract solidity code responsible for executing transactions, and these transactions cannot be undone. To deploy a smart contract on a blockchain, it is necessary to pay for gas in the same manner that is required for a straightforward transfer. One smart contract can deploy another smart contract, which means that smart contracts are composable. The development of a smart contract is performed in stages that are separated from one another. It is their responsibility to ensure that the client's requirements are satisfactorily met, and that the innovation meets those requirements.

1.2. Smart Contract Testing

Smart Contract testing verifies the correctness, security and functionality of smart contracts deployed over the blockchain network. The evaluation of smart contracts involves verifying the behaviour of the contract under various conditions ensuring the intended functionality and complies requirements. Includes testing like functional testing, security testing, performance testing, integration testing, fault-based testing, gas optimization, regression testing etc.

1.3. Mutation Testing For Smart Contracts

One of the testing methods in unit testing is mutation testing. Mutation testing is a fault-based testing method used to assess the quality of the test suite. Mutation testing is used in

limited quantities by industry because it is both time-consuming and expensive to compute it [14]. There are research studies that can be conducted to reduce this computational cost by reducing the number of mutants that are generated, among other things. In the process of mutation testing, errors are introduced into the code that is being tested, which is referred to as mutated code. The code that has been modified is run against the test case that is associated with the source code [14]. In the event that the test cases are able to identify the faults that have been injected, the mutants will be in a state of being killed; otherwise, the mutants will be either surviving or alive.

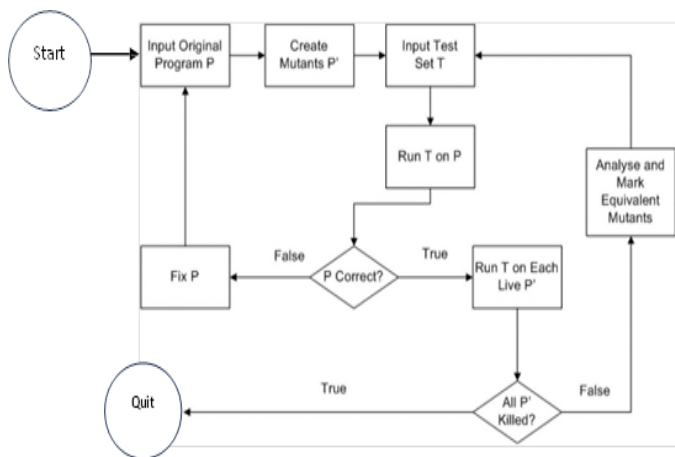


Fig- 1.1: Mutation test Process

2. LITERATURE REVIEW

Regular Mutator [81] mutation happens for each source code that has test suites and smart contracts using mutants generated. The regular expressions library is utilised for the implementation of the mutation injection approach. The Regular Mutator executes project test suites by repeatedly replacing the original files with mutant files that have been created. Upon examination of the test results, the mutant is assigned one of the following designations: 'killed', 'survived', or 'error'. The experiment analysing the performance of regular mutators utilised a smart contract from the POA bridge project. The lines of code coverage metric achieved a rate of 96%, and a total of 129 files were employed for testing by the tool. The achieved mutation score was 18.5%. Upon meticulous analysis of 50 mutants, it was determined that the mutants most likely to endure were those generated by the utilisation of the Operator Replacement for Relational operations technique and operators that are Solidity-specific. Achieved a good level of code coverage and a relatively low mutation score for the "Smart Contracts- POA Bridge". By incorporating the insights from the mutation study into the test suites, we were able to enhance their quality. As stated in [10], the POA project utilises Regular Mutator to assess the quality of test suites, placing greater emphasis on the metric mutation score rather than metric- line coverage. When comparing the mutation adequacy score to the line coverage measure, it was

observed that the mutation score has a substantially lower value than the line coverage statistic. Simultaneously, the test suites were enhanced and a vulnerability was identified through mutation analysis. RegularMutator's development revealed that the most efficient mutation operators are those specific to Solidity and those belonging to the Relational Operator Replacement class. Specifically, the mutants that survived consisted of 26% Solidity-specific operators and 18% ROR operators. It should be emphasised that not all mutants derived from the survivors are beneficial and enhance the quality of the test suite. As an illustration, it is necessary to get rid of comments in the source code because any changes made to them will not have any impact on the way the code is executed. Furthermore, the percentage of mutations resulting in syntactical errors during compilation and the time needed to verify them were substantial. The experiment was conducted for a duration of approximately 50 hours on the machine. Indeed, mutation analysis poses substantial computational obstacles that restrict its practicality in specific circumstances. Moreover, a substantial percentage of mutants that successfully completed the experiment need to be thoroughly examined and assessed. Given the fact that smart contracts cannot be changed once they are deployed and the possibility of financial losses resulting from vulnerabilities in the project, we can conclude that the higher computational expense of testing is reasonable.

Deviant [15] intended to generate variations of a specific Solidity project and execute the provided tests on each variation to assess the efficiency of the testing. Deviant processes one program file at a time, converting it into an Abstract Syntax Tree (AST), and generates mutants by applying mutation operators selected by the user. A Solidity project and its corresponding test code have been submitted to Deviant. Mutation operators are defined based on a comprehensive defect model specifically designed for the Solidity language. The model takes into account the conventional fault categories in traditional programming languages, particular aspects of the Solidity language, and the published faults specific to Solidity. After the creation of each mutant, an exact replica of it will be placed in the project directory and transformed into EVM bytecode, maintaining complete similarity with the original form of the project. The provided Solidity project will undergo testing against the EVM bytecode. Until every mutant has undergone the tests, this procedure is repeated. Deviant records the results of every mutant's test execution and generates a comprehensive test report on the mutation testing, including metrics such as mutation score, number of deceased mutants, and number of alive mutants. Validating the viability of each mutant as a functional program prior to compilation poses a significant technological challenge in mutant generation. A contract that can be built is considered to be a valid anomaly. Demands a comprehensive examination into the reasons behind the survival of the majority of non-equivalent mutants, even while the tests fulfil the requirements of statement or branch coverage.

Vertigo mutation testing tool was designed for truffle and solidity smart contract. Mutation testing involves with test suite initially and checks for if the test suite fails for the smart contracts. If not, the next step is generation of mutations in which the filter is used to select a random sample of mutants from the available mutant list. Then the mutations are applied to the code and is executed against the test suite by the mutation runner. Provides the entire mutation process. The major limitation in this vertigo is, it does not support for testing individually the tests and code coverage report are not available. The vertigo tool executes the entire test suite for each mutant which requires optimization.

The aforementioned tools vary in their methods of generating mutants and the operators employed by each tool differ. The created mutants are abundant in number and all of them are being utilised to execute the modified source code in comparison to the testing set. The execution time of the mutants is directly proportional to the number of mutants. The mutant sampling method was employed by certain instruments to decrease the quantity of mutations. There is no assurance that the mutations that are decreased and utilised in the execution process are effective for mutation analysis. The efficacy of mutants is crucial in the examination of mutants. Utilising time on the execution of all the redundant mutants is unnecessary.

Many researchers are currently studying the construction of test cases automatically for smart contracts, which is comparable with the process used in software engineering to create test cases for other software. Current methods for generating test cases continue to face challenges in identifying vulnerabilities. May experience challenges when confronted with the unique components of Ethereum smart contract's statement of code like require, view and pure. The primary obstacle in utilising the mutation testing technique is the creation of a substantial number of mutants. For mutants' reduction, random sampling and obtaining subset of mutants are the strategies employed in current mutation testing technique. However, these techniques face challenges in picking appropriate mutants to test the quality of the test suite. To make mutation testing practical for experts, it is essential to address the high volume of mutants produced by current mutation systems, many of which are deemed unproductive according to Samuel et al. The efficacy of utilising a mutant as a test subject in mutation testing relies on its capacity to produce a pertinent test that improves test coverage. This study presents a new method called the Test Completeness Advancement Probability (TCAP) metric for assessing the effectiveness of mutant usefulness. This work further illustrates that the static program environment of a mutant can accurately forecast its TCAP (Test Case Adequacy Prediction). Additionally, the selection of mutants based on their TCAP can serve as a valuable approach to guide mutation testing.

3. TEST CASE GENERATION USING SAL-GA

Blockchain's intrinsic characteristics, which include resistance to assault and independence from a central authority, make it ideal for managing smart contracts, especially in digital management. Complicated bugs caused by attackers or malicious developers might result in false positive errors. Additionally, autonomous test case generation for smart contracts encounters difficulties related to test generation, test automation, and oracle issues. Symbolic execution models have been utilised to create test cases for code paths in smart contracts concerning financial transactions. However, challenges remain in terms of coverage, adaptability, scalability, diversity of vulnerabilities, automation, and efficiency. Here provides a new automated way for generating test cases for smart contracts to solve these issues using SAL-GA that improves code coverage and detection of vulnerabilities.

3.1. Architecture

The proposed architecture for automated test case generation is depicted in Figure 3.1, consisting of six blocks. At first, various smart contracts are selected with code errors and vulnerable presence within the Decentralized Financial related projects encompassing lending platforms, synthetic assets and decentralised exchange. The construction of the Application Binary Interface (ABI) provides with the required information for test cases generation and executing test cases in testing environment. The parsing process is created to analyse the source code written in Solidity.

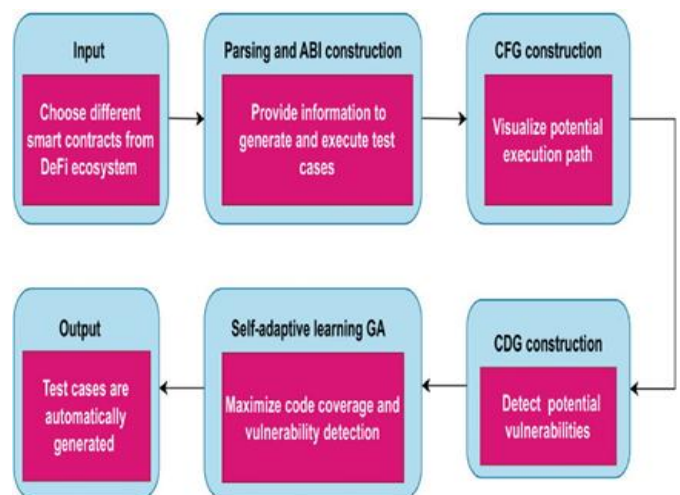


Fig- 3.1: Automated Test Cases Generation using SAL-GA

3.2. Initialization of Solution

Devised a method that combines random value creation with heuristic-based value generation to create an initial population of solutions. In order to generate arbitrary values, a randomizer function was developed for each data

type to produce random values. Consider a scenario where a smart contract function necessitates an integer parameter. In such cases, the randomizer function operates by generating an integer randomly chosen from the permissible range of integers for the Ethereum Virtual Machine (EVM). Similarly, when an argument requires an Ethereum address, the randomizer function generates a random Ethereum address as the argument. The process of deriving values based on heuristics entails a meticulous and sophisticated approach. It involves analysing both the functionality and historical transaction data of the smart contract to infer probable input patterns. For instance, if past transactions consistently provided integer arguments within a specific range for a particular function, the randomizer function would prioritize generating initial values within the same range.

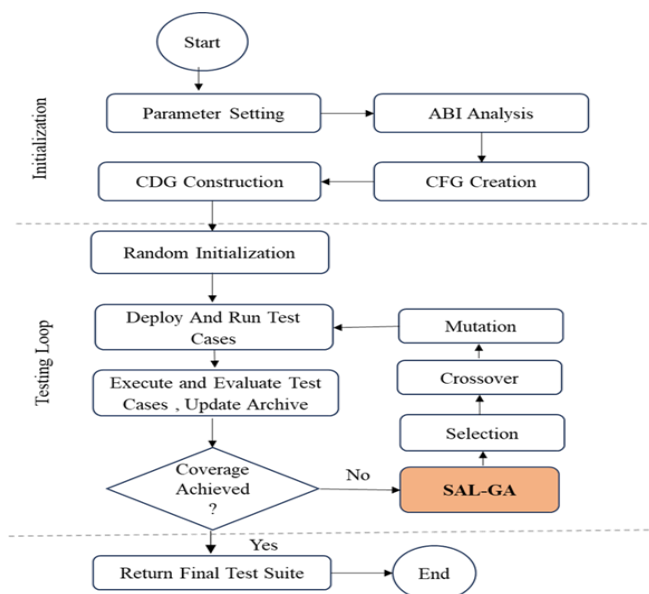


Fig- 3.2: Test Case Generation Process

3.3. Experimental Setup

Conducted studies on test case creation using Python 3.10.9. These trials were executed on a Windows 11 operating system, leveraging a system configuration featuring 16 GB of RAM and an Intel(R) Core(TM) i5-1005G1 processor operating at a clock speed of 1.20GHz. The system was equipped with a 64-bit operating system to support the computational demands of the experiments. Test environment for Ethereum is used with Truffle and local blockchain with Ganache, and for smart contract compilation used Solidity v0.5.16, along with Node v18.15.0 and Web3.js v1.8.2. This setup provided a robust environment for evaluating the efficacy and performance of various test case generation techniques, allowing for comprehensive analysis and comparison of results.

3.3.1: Parameter Settings

The SAL-GA parameters were optimised in order to attain maximum performance and supremacy. The software was

meticulously designed and adjusted to incorporate a wide range of parameters, script packages, and methodologies. Table 3.1 is a concise summary of the parameter configurations utilised in study.

Table 3.1: SAL-GA Parameter Settings

Algorithm	Parameters	Optimal Values
SAL-GA	Size of Population	200
	Probability of Mutation	0.05
	Probability of Crossover	0.9
	Probability of mutation strategy Selection	0.25
	Generators	10
	Learning rate	0.1667
	Total Populations	1000
	Each Solution Weight	0.4

4. RESULT AND DISCUSSIONS

This study evaluates the efficacy of a proposed method by analysing various metrics such as average code coverage rate, vulnerability detection rate, path uniqueness rate, execution rate, false positive rate (FPR), false negative rate (FNR), test case generation time, total vulnerabilities identified, precision, and recall. Assessed the proposed method's superiority by comparing it with several existing approaches such as ReL-GA, SaFReL, aDynaMOSA, GA, and Improved GA. Table 4.1 provides results obtained using proposed SAL-GA.

Table 4.1: Test Case Generation using SAL-GA.

Evaluation Metrics	Performance Ranges
Time of execution	25 seconds
Time to generate test cases	16 seconds
Average code coverage	98.1%
Total vulnerability identified	3500
Vulnerability detection	98.7%
Recall	98.2%
Precision	98.8%
False positive rate	1.3
False negative rate	3.50
Rate of Path uniqueness	96.4 %

4.1. Comparative Analysis

The proposed SAL-GA is compared with various other algorithms that are used to generate test cases. The

comparative results in this section are provided in terms of execution time, test case generation time, code coverage rate, total vulnerabilities detected and vulnerability detection rate.

The analysis of execution time dependent on the number of test cases that were generated is depicted in Figure 4.1. The comparison takes into account the SAL-GA approach that has been proposed and developed, as well as ReL-GA, SaFReL, aDynaMOSA, GA, and Improved GA. Execution time is the amount of time that is spent on the generation of each individual test case. In especially for smart contracts, the execution time has a tendency to increase in tandem with the number of test cases that are generated. According to the findings of the analysis, the SAL-GA model proposed has a duration of execution that is shorter than that of other approaches for the development of test cases. In particular, the proposed model was able to obtain an execution time of 25 seconds when it was performing the 2000th test case, it recorded 102 seconds when it was performing the 10000th test case. This demonstrates that the proposed method is effective in producing test cases in a shorter amount of time, which makes it a desirable option for the activities that include the development of test cases.

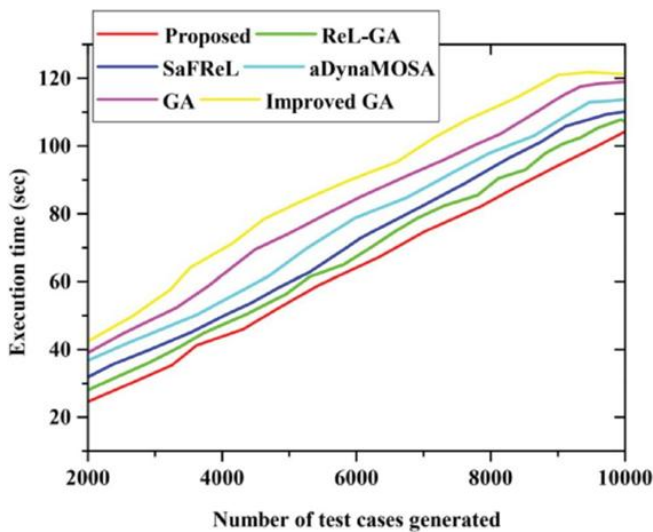


Fig-4.1.: Execution Time Analysis

Figure 4.2. displays the evaluation of the time taken for test case generation using different comparison methods: ReL-GA, SaFReL, Improved GA, aDynaMOSA, GA and the proposed approach. Test case creation time is amount of time it takes to create test cases. The research demonstrates that the proposed method surpasses the other methods in terms of the time required for generating test cases. The proposed method took 16 seconds to generate test cases at the 2000th test case, and 112 seconds at the 10000th test case. Nevertheless, it is important to acknowledge that the time required for test case generation tends to rise proportionally with test cases quantity developed. The results clearly specifies that proposed methodology provides robust test

cases generated with low time when compared with other methods been considered in evaluation.

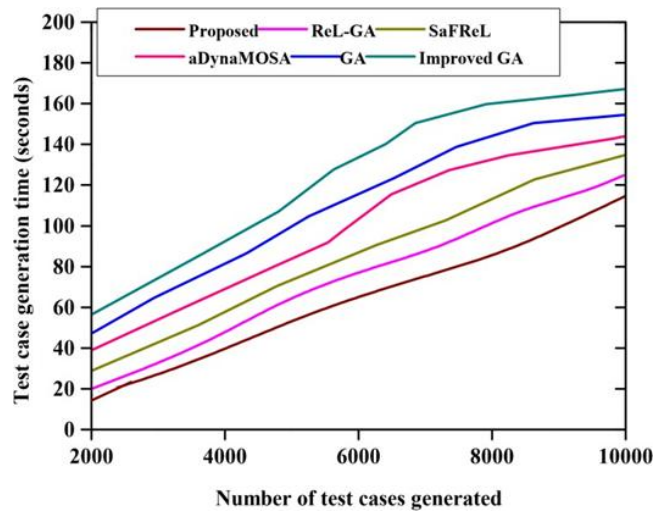


Fig- 4.2.: Test Case Generation Time Analysis

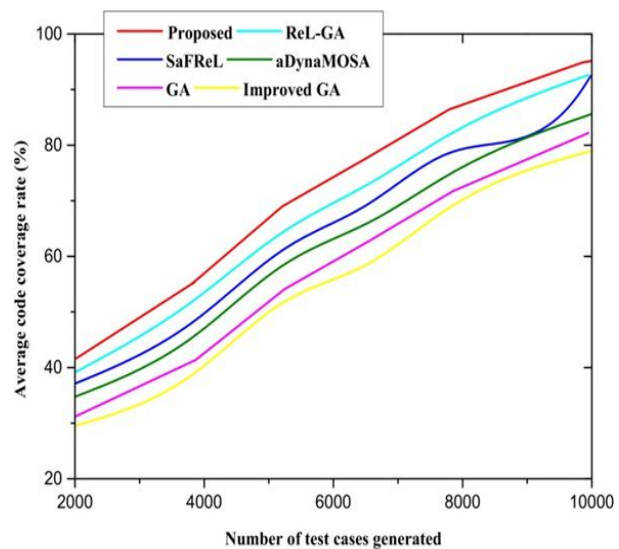


Fig-4.3.: Average Code Coverage Rate

Figure 4.3. illustrates the analysis of the average code coverage rate, which is determined by the number of automatically generated test cases. Different approaches to the development of test cases are compared in this comparative analysis. These approaches include the proposed method, ReL-GA, GA, Improved GA, SaFReL, and aDynaMOSA. This implies that a higher code coverage rate offers more chances to detect coding issues and ensure the proper behavior of the application. In order to get average code coverage rate, it's necessary to identify sections of code that are not covered by the test cases that have been developed. Typically, when more test cases are generated, observed increase in the average code coverage rate. The approach for the production of test cases that has been proposed has highest average rate among all alternatives that have been assessed. In particular, the proposed

technique was able to obtain 98.1% of code coverage rate on average when it was applied to the ten thousandth test case. The fact that this is the case demonstrates that the approach that was proposed, effectively covers major percentage of code, which ultimately results in improved detection of potential programming problems. In general, the findings demonstrate that the proposed model has superior performance in terms of reaching a high average rate in code coverage when compared with other approaches that were taken into consideration during the analysis.

Figure 4.4 depicts comparative analysis of the total vulnerabilities detected. The comparison involves various approaches for generating test cases, including SaFRel, Improved GA, aDynaMOSA, the suggested method, ReL-GA, and GA. The data demonstrates a positive correlation between quantity of generated test cases and the overall number of vulnerabilities found across all test case generating methods

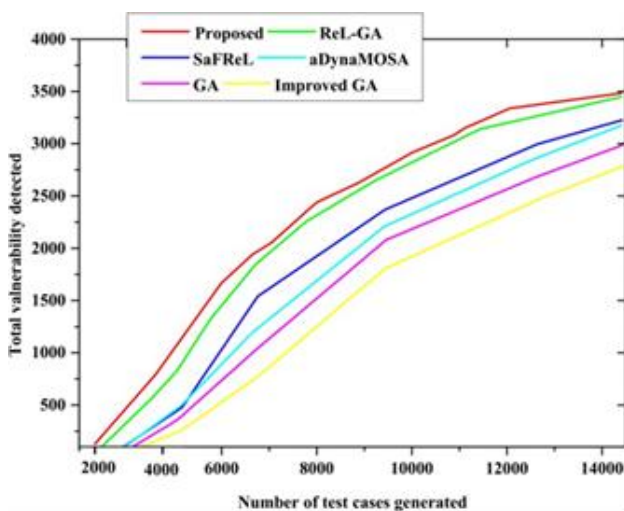


Fig-4.4.: Total Number of Vulnerabilities Detected

Nevertheless, proposed approach exhibits advantages in detecting vulnerabilities. At the 14000th test scenario, the suggested method successfully identifies 3500 vulnerabilities, demonstrating its efficacy in detecting potential vulnerabilities. Yet, the other test case generation methods found varying numbers of vulnerabilities: 3250 for SaFRel, 2850 for Improved GA, 3240 for aDynaMOSA, 2490 for ReL-GA, and 3000 for GA. The results show that the suggested test case generation process is highly effective in finding vulnerabilities, outperforming other methods studied. The higher number of vulnerabilities identified by proposed technique indicates its ability in uncovering potential weaknesses and enhances overall system security.

5. CONCLUSIONS

In navigating the intricacies of automated test case creation for Ethereum smart contracts, encountered multifaceted obstacles revolving around code coverage, vulnerability spectrum, and the automation of vulnerability detection. Recognizing the imperative nature of these challenges in ensuring the robustness and security of decentralized finance (DeFi) ecosystems, a sophisticated genetic algorithm named SAL-GA was proposed, meticulously designed to refine and enhance the process of Ethereum smart contract testing. Drawing data from a diverse array of smart contracts within the DeFi collection of smart contracts related to Finance, rigorously evaluated SAL-GA's performance using a comprehensive set of metrics. These metrics encompassed crucial aspects such as average code coverage, vulnerability detection rate, path uniqueness rate, execution time, false positive and false negative rates, time taken for test case generation, and the overall efficacy in identifying vulnerabilities. The comparative analysis involved benchmarking SAL-GA against several established methodologies including Genetic Algorithm, adaptive DynaMOSA, Improved Genetic Algorithm, ReL-GA, and self-adaptive fuzzy reinforcement learning (SaFRel). SAL-GA achieved impressive results, generating test cases quickly in just 16 seconds, with 98.1% code coverage rate being high on average and an outstanding vulnerability detection rate of 98.7%. These results underscore SAL-GA's efficacy in surmounting the challenges of Ethereum smart contract testing, thereby fortifying security measures and resilience within the DeFi ecosystem

REFERENCES

- [1] Bitcoin Whitepaper. (2008). *Bitcoin: A peer-to-peer electronic cash system*. Retrieved from <https://bitcoin.org/bitcoin.pdf>
- [2] Bitcoin.org. (n.d.). *Bitcoin official website*. Retrieved from <https://bitcoin.org/en/>
- [3] Nakamoto Institute. (n.d.). *Micropayments and mental transaction costs*. Retrieved from <https://nakamotoinstitute.org/static/docs/micropayments-and-mental-transaction-costs.pdf>
- [4] Macrinici, D., Cartoceanu, C., & Gao, S. (2018). Smart contract applications within blockchain technology: A systematic mapping study. *Telematics and Informatics*, 35(8), 2337–2354.
- [5] Liu, J., & Liu, Z. (2019). A survey on security verification of blockchain smart contracts. *IEEE Access*, 7, 77894–77904.
- [6] Rouhani, S., & Deters, R. (2019). Security, performance, and applications of smart contracts: A systematic survey. *IEEE Access*, 7, 50759–50779.
- [7] Vacca, A., Di Sorbo, A., Visaggio, C. A., & Canfora, G. (2021). A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software*, 174, 110891.
- [8] Nanayakkara, S., Perera, S., Senaratne, S., Weerasuriya, G. T., & Bandara, H. M. N. D. (2021). Blockchain and smart

- contracts: A solution for payment issues in construction supply chains. *Informatics*, 8(2), 36.
- [9] Wang, Y., He, J., Zhu, N., Yi, Y., Zhang, Q., Song, H., & Xue, R. (2021). Security enhancement technologies for smart contracts in the blockchain: A survey. *Transactions on Emerging Telecommunications Technologies*, 32(12), e4341.
- [10] Lin, S. Y., Zhang, L., Li, J., Ji, L. L., & Sun, Y. (2022). A survey of application research based on blockchain smart contract. *Wireless Networks*, 28(2), 635–690.
- [11] Barboni, M., Morichetta, A., & Polini, A. (2022). Smart contract testing: Challenges and opportunities. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain* (pp. 21–24).
- [12] Nzuba, S. (2019). Smart contracts implementation, applications, benefits, and limitations. *Journal of Information Engineering and Applications*, 9(5), 63–75.
- [13] Krupa, T., Ries, M., Kotuliak, I., & Bencel, R. (2021). Security issues of smart contracts in Ethereum platforms. In *2021 28th Conference of Open Innovations Association (FRUCT)* (pp. 208–214). IEEE.
- [14] Andrews, J. H., Briand, L. C., Labiche, Y., & Namin, A. S. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8), 608–624.
- [15] Wood, G. (2014). *Ethereum: A secure decentralised generalised transaction ledger* (Ethereum Yellow Paper).
- [16] Hassan, M., Ali, I., Ahammed, R., Khan, M. M., Alsufyani, N., & Alsufyani, A. (2021). Secured insurance framework using blockchain and smart contract. *Scientific Programming*, 2021, 1–11.
- [17] Muneeb, M., Raza, Z., Haq, I. U., & Shafiq, O. (2021). Smartcon: A blockchain-based framework for smart contracts and transaction management. *IEEE Access*, 10, 10719–10730.
- [18] Górski, T. (2021). Towards continuous deployment for blockchain. *Applied Sciences*, 11(24), 11745.
- [19] Ibrahim, R., Harby, A. A., Nashwan, M. S., & Elhakeem, A. (2022). Financial contract administration in construction via cryptocurrency blockchain and smart contract: A proof of concept. *Buildings*, 12(8), 1072.
- [20] Investopedia. (n.d.). *Smart contracts*. Retrieved from <https://www.investopedia.com/terms/s/smart-contracts.asp>
- [21] Taş, R. (2023). Smart contract security vulnerabilities. *Erzincan University Journal of Science and Technology*, 16(1), 196–211.
- [22] Azaria, A., Ekblaw, A., Vieira, T., & Lippman, A. (2016). MedRec: Using blockchain for medical data access and permission management. In *International Conference on Open and Big Data* (pp. 25–30).
- [23] Ribeiro, S. L. I., & Barbosa, A. P. (2020). Risk analysis methodology to blockchain-based solutions. In *Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)* (pp. 59–60).
- [24] Xu, J. J. (2016). Are blockchains immune to all malicious attacks? *Financial Innovation*, 2(1), 25.
- [25] Wang, S., Wang, C., & Hu, Q. (2019). Corking by forking: Vulnerability analysis of blockchain. In *IEEE Conference on Computer Communications* (pp. 829–837).
- [26] Li, X., Jiang, P., Chen, T., Luo, X., & Wen, Q. (2020). A survey on the security of blockchain systems. *Future Generation Computer Systems*, 107, 841–853.
- [27] Dey, S. (2018). Securing majority-attack in blockchain using machine learning and algorithmic game theory: A proof of work. In *Computer Science and Electronic Engineering* (pp. 7–10).
- [28] Ramezan, G., Leung, C., & Wang, J. Z. (2018). A strong adaptive, strategic double-spending attack on blockchains. In *International Conference on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, and Smart Data* (pp. 1219–1227). IEEE.
- [29] Zhao, X., Chen, Z., Chen, X., Wang, Y., & Tang, C. (2017). The DAO attack paradoxes in propositional logic. In *International Conference on Systems and Informatics* (pp. 1743–1746).
- [30] WIRED. (2016). *The \$50 million DAO hack explained*. Retrieved from <https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/>
- [31] Investopedia. (n.d.). *Ethereum smart contracts vulnerable to hacks*. Retrieved from <https://www.investopedia.com/news/ethereum-smart-contracts-vulnerable-hacks-4-million-ether-risk/>
- [32] FindLaw. (n.d.). *Reports: Ethereum smart contracts are far from secure*. Retrieved from <https://www.findlaw.com/legalblogs/technologist/reports-ethereum-smart-contracts-are-far-from-secure/>
- [33] Zhao, H., Li, X., & Gai, K. (2022). A dynamic taint analysis-based smart contract testing approach. In *International Conference on Smart Computing and Communication* (pp. 403–413). Springer.
- [34] Krichen, M., Lahami, M., & Al-Haija, Q. A. (2022). Formal methods for the verification of smart contracts: A review. In *2022 15th International Conference on Security of Information and Networks (SIN)* (pp. 1–8). IEEE.
- [35] Du, J., Huang, S., Wang, X., Zheng, C., & Sun, J. (2022). Test case generation for Ethereum smart contract based on data dependency analysis of state variable. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)* (pp. 710–720). IEEE.
- [36] Akca, S., Peng, C., & Rajan, A. (2021). Testing smart contracts: Which technique performs best? In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1–11).
- [37] Viglianisi, E., Ceccato, M., & Tonella, P. (2020). A federated society of bots for smart contract testing. *Journal of Systems and Software*, 168, 110647.
- [38] Bhardwaj, A., Shah, S. B. H., Shankar, A., Alazab, M., Kumar, M., & Gadekallu, T. R. (2021). Penetration testing framework for smart contract blockchain. *Peer-to-Peer Networking and Applications*, 14, 2635–2650.
- [39] Xu, Y., Hu, G., You, L., & Cao, C. (2021). A novel machine learning-based analysis model for smart contract vulnerability. *Security and Communication Networks*, 2021, 1–12.