Journal of Theoretical and Applied Information Technology
15th November 2023. Vol.101. No 21
© 2023 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

AUTOMATED MUTATION ANALYSIS FOR SMART CONTRACT USING AMA TOOL WITH ENHANCED GA AND MACHINE LEARNING APPROACH

R SUJEETHA[1], K AKILA[2]

[1,2] Department of Computer Science and Engineering, College of Engineering and Technology, SRM Institute of Science and Technology, Vadapalani Campus, Vadapalani, Tamil Nadu, India.

E-mail: [1]sr7092@srmist.edu.in, [2]akilak@srmist.edu.in

ABSTRACT

Smart Contracts are the critical most popular in the Dapps Blockchain network. Smart contracts play a vital role in safety-critical products. The quality of the smart contract is a vital factor. Test cases are used to ensure the correctness of the smart contract code. The efficiency of the smart contract test suite is assessed using the mutation testing technique. The state-of-the-art tools for assessing test suite quality generate numerous mutants for execution. The test case generation-related state-of-the-art tools code and functional coverage require further research to provide better coverage. This article proposes a tool for performing automated mutation analysis (AMAT) for smart contracts in which the test cases are generated using the proposed enhanced GA used for the mutation analysis. The mutation testing utilizes the effective mutants obtained using a machine learning-based classification algorithm for reducing the number of mutants executed. The results show that the tool effectively generates optimized test cases with high branch and function coverage and achieves up to 98% mutation scores.

Keywords: *Test Case, Genetic Algorithm, Mutation Testing, Classification*

1. INTRODUCTION

Blockchain Technology was first introduced in the concept of Bitcoin [1]. Blockchain technology is maintained by peers where the transactions happening in a time are recorded and packed into a block, and a link is created to join the blockchain. It is a decentralised, traceable and tamper-free technology [2]. A smart contract is an event-driven code written in solidity programming language [3]. These smart contracts can be invoked by initiating a transaction with the peers in the blockchain. Once peers verify the transaction and it is a success, it's recorded in the blockchain, indicating the end of execution of the smart contract.

In blockchain DApps are used as blockchain has decentralised characteristics. There are four types of DApps based on their architecture as a DApp – Native Client, Smart Contract, Web and Contract, Fully Decentralized. Here consider DApp - Web and Contract, where more than 5000 DApps are available. There is an increase in transactions using CrowdFunding DApps. The Decentralized Finance DeFi are popular among the investment-related DApps.

Smart contracts are codes or rules that operate on a blockchain and carry out the conditions of a deal or an agreement. They are used to streamline the exchange of digital assets between parties without the need for middlemen or trust. Smart contracts are binding and provable. Some network nodes, called miners, execute the smart contracts in the blockchain. Smart contracts have many applications in various domains, like security tokens, e-voting, land registration, education certificates, stock exchange, and supply chain management. Millions of dollars are handled through smart contracts; thus, any safety feature or conceptual flaw could result in significant financial losses. For instance, the Ethereum network suffered a loss of $150 million due to the infamous DAO assault [4]. Additionally, because smart contracts cannot be altered, potential problems cannot be fixed. Over '33000 insecure smart contracts on the Ethereum blockchain, containing about Ether 4900, according to Nikolic et al. [5], demonstrate the significance of precise verification of smart contracts.

The smart contracts code verification plays a vital role in providing a quality smart contract deployed in the blockchain network. To provide

© 2023 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

quality smart contract testing, smart contracts are necessary, and this is done by writing test cases or test suites. Test cases are those required to confirm a feature or its working in software testing. Test cases are either written manually or it is generated automatically. Manual writing of test cases involves great effort from testers and consumes a lot of time. Hence, the automatic generation of test cases is focused. The automatic creation of test suites for smart contracts based on solidity is possible using a number of tools [11–14]. Fuzzing, artificial intelligence, and genetic algorithms are some of the techniques used in the tools cited above. The tools provide test cases which lack branch coverage and function coverage. There is a requirement for optimizing the test case generation to improve the quality of test cases.

Mutation testing is an effective method for evaluating test suites of smart contracts that have attracted a lot of attention from researchers. The term mutation is defined as the small changes made in the source code to introduce the faults that reflect the real errors that commonly occur in coding. This version of code is called mutated source code. The mutated source code is executed against the test suite, and the results are analysed by the capability of a test suite to identify the mutated errors. If the mutants are identified by the test suite it's calculated as mutant killed else mutant is alive and is calculated as mutant survives. The mutation score determines the quality of the test suite. The most popular smart contract language used for developing smart contracts is solidity. There are tools for quality checking the solidity of smart contracts test suites [6],[8-10]. For Solidity smart contracts, these [6, 8-10] tools perform mutation testing for the smart contract test suites and accesses the test suite quality. These tools have mutation operators proposed, that includes Mutation testing for Smart contracts MuSC [6], Deviant [8], and SuMo [10]. With the help of mutation testing, these works attempt to report the test suite's quality using mutation score, which involves generating a vast number of mutants and takes longer to get executed. The techniques used in [6,8-10] tools require further investigation to reduce the number of mutants generated or executed.

Further, the execution of automatic test case generation requires an environmental setup different from the mutation testing tool. Both tools are executed on various platforms and consume time to set up the environment for each tool. This paper proposes AMAT tool for Automated Mutation Analysis to bridge the challenges faced in test suite generation and mutation testing as two different platforms. The main goal of this paper is to optimize the test case generation and test the quality of the test case generated using a single framework. The framework comprises three modules: one for generating a test suite, the second for a mutant engine, and a third to perform mutation testing based on the classified effective mutants to assess the test suite quality.

First, a tool for the automated creation of test suites for Solidity smart contracts is developed. Secondly, a mutation testing tool is developed to examine the test suites generated. The two tools are integrated. The findings are fascinating from two perspectives. Firstly, demonstrates how the test cases are generated using the proposed enhanced GA. Secondly, assess the test suite's quality using a mutation testing tool, in which the mutants not killed by the test suite highlight the type of problems not identified by test cases. Also, this tool can be used to assess various available test suites for smart contracts.

The rest of this article is organised as follows. The preliminary notions are briefly discussed in Section II. Section III introduces AMAT for automated mutation analysis for smart contract, and Section IV describes the experiment results in terms of the useful test cases that were produced and the assessment of the test suite's quality. Section V contains concluding remarks.

2. BACKGROUND

The related works are selected from recent publications in reputed journals like IEEE Transactions, Springer, etc.; these were selected based on keyword searches like mutation testing tools for smart contracts, smart contract testing challenges and test case generation for smart contracts. The following sections discuss the techniques and limitations of tools for smart contract mutation testing and test case generation.

2.1 Ethereum Smart Contracts

The Ethereum Virtual Machine (EVM) is where Ethereum smart contracts are carried out and are created using solidity, a statically typed curly-braces programming language. Smart contracts are written codes implemented within a peer-to-peer network where no one has particular ownership over the execution, allowing anybody to implement tokens of value, ownership, voting, and other types of logic. It is required to utilise the most recent version of Solidity while deploying contracts. This is due to the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

frequency with which breaking updates, new features, and bug fixes are introduced. The blockchain technology known as Ethereum was created specifically for creating and deploying smart contracts. The Ethereum framework works on a model based on accounts, where the people participating in the network are represented through their accounts. Every transaction that happens in the Ethereum blockchain can be represented as an interaction between an externally owned account and a contract account.

Externally owned account - To execute transactions on Ethereum, users must first create an account. A unique private-public pair of keys is established for each account type. The private key allows the user to access the account's funds, whereas the public key is used to authenticate the account. Each External Owned Account can send transactions to other accounts of the same type and Contract Accounts.

Contract Account – A Contract account is created due to deploying a smart contract on the Ethereum network. Other accounts can then communicate with this account via its linked address. A contract can execute transactions on its storage calls, and fire calls on other contracts as a reaction to a transaction.

2.2 Mutation Testing

Mutation testing is a highly effective technique used to assess and enhance the quality of a test suite. It works by introducing minor flaws into copies of the code's source, which helps ensure the accuracy of test data in detecting real flaws and identifying constraints in the implemented test suite. The ultimate goal of mutation testing is to deploy trustworthy code by replicating the majority of defects through the identification of a subset of simple faults that can be found in real-world programs. This approach is based on the assumption that by identifying these defects, we can better understand how to improve the overall quality of our code.

Mutation testing creates mutants, slightly altered versions of the original code. A mutant has a minor difference from the original code that simulates a common programming mistake. The main element of mutation testing is mutation operators, which are rules for modifying the source code to produce a mutant.

Mutation testing evaluates the test suite by running it on each mutant code and checking if the tests can find the inserted error. If the test suite identifies the mutant on a mutated code, the mutant is said to be killed; otherwise, it is marked as alive. The compiler can also discard some mutants. The mutant is referred to be stillborn if the mutation results in a compilation error.

The Mutation Score (MuScore), or percentage of mutations eliminated by the test suite (EQ. (1)), is a measure of the test suite's suitability. The test suite can be improved iteratively by the developer until the Mutation Score satisfies the adequacy condition.

$$MuScore = \frac{Total\ Mutants - Live\ Mutants}{Total\ Mutants} \times 100 \quad (1)$$

However, the equivalent mutants must be identified and removed before computing the Mutation Score. A code with syntactic modifications that behaves exactly like the original code is called an equivalent mutant. Valid live mutants can be examined to gain valuable knowledge about the kinds of faults that standard testing cannot pick. Based on this feedback, the test suite can be improved, and missed instances during the test design process can be covered.

Mutation analysis is a potent but expensive testing method. Large code bases and test suites can make a thorough mutation process expensive and time-consuming. Researchers and professionals have investigated effective cost-reduction strategies [15][16]. Typically, either the number of mutants or the amount of time required to run the program can be decreased to make mutation testing simpler [15]. The quality of the final test suite could be jeopardised if either technique is used carelessly. Mutant sampling, Mutant clustering, and selective mutants are some of the mutant reduction techniques followed. These techniques use a subset of mutants, selective operators alone are considered if these techniques are done properly, the small set can be as effective as the whole set of mutants. The cost of execution of a mutant code can be reduced by slacking the killing condition. The weak mutation approach doesn't involve running the complete program since it assumes that the mutant is destroyed whenever it deviates from the original code and changes its state. The test assessment is not as expected.

The articles [6,8-10] have proposed test suite assessment methods using mutation scores for smart contracts. These tools commonly use user-selected mutation operators specific to smart contracts. Based on the selected operators, generated the mutants and the mutated source code is used for execution against the test suite. This involves random sampling and selective mutants techniques for reducing the mutants. These does not satisfy the usage of effective mutants and requires further research in reducing the mutants and using the only the effective mutants. This motivated the further study of this research.

The predictive mutation testing [23] predicts the mutation results without the mutants executions. Constructed a classification model with the features related to mutants and tests that predicts whether the mutant is killable or survivable. This model has been evaluated using projects written in JAVA language achieved small accuracy loss and 0.80 AUC score. The effectiveness of the model can be improved by using more test appropriate features than the categorical features and also to use large dataset.

The other significant problem with mutation testing is the tester's need for manual labour. To enhance the test suite, each living mutant must be manually examined. Finding identical mutations, which is an impossible undertaking, comes at an additional cost. Selective mutation, HOMT (higher-order mutation testing), and the TCE (Trivial Compiler Equivalence) [17] can all assist in resolving this issue. Nonetheless, the screening procedure requires human participation because no automated technique can remove all similar mutations.

2.3 Automated Test Case Generation

The creation of automated test cases for smart contracts similar to other software in software engineering using automated methods is a current field of study by many researchers.

To overcome problems and build test suites, Driessen et al. [18] created an Automated Generator of Solidity Test Suites (AGSOLT). To gain greater branch coverage for smart contracts, the AGSOLT compared random and guided searches. However, whether this approach could yield reduced test cases for final test suites was unclear.

Ji et al. [19] enhanced the Genetic Algorithm (GA) to improve its ability to locate global optima. Test cases for smart contracts built on solidity were created using the GA. The findings show that the algorithm reduced execution time while obtaining good coverage. The method, however, proved unsuitable for large-scale research and did not detect smart contract weaknesses.

Olsthoorn et al. [20] created SynTest-Solidity, an automated test case production tool incorporating the fuzzing technique as a framework for smart contracts. The framework proved useful for testing solidity smart contracts using meta-heuristic search-based algorithms. The method, however, was limited to the solidity programming language and did not include other languages such as TypeScript and JavaScript. The code coverage 70%, function coverage 91% and branch coverage 64% are achieved. The evaluation of the tool requires further research.

To more efficiently generate test cases with excellent coverage for smart contract data flow testing, S. Ji et al. [21] describe an improved GA based test-case generating method. The method incorporates particle swarm optimisation theory into the genetic algorithm, which decreases the role of randomness in genetic operations and improves its capacity to locate global optima. More complex fitness functions should be designed to assess if test cases cover the same def-use pairs and to lead the search for appropriate test cases.

The studies show continuous efforts to enhance smart contract automated test creation to improve coverage, lower execution time, and boost test generation effectiveness. However, the literature also reveals various areas for improvement, such as extending these methods to more programming languages and increasing their ability to find errors.

Existing test case generation approaches still suffer from detecting vulnerabilities. Some encounter difficulties when faced with the distinctive elements of Solidity smart contracts, such as require statements. Coverage of code and branch still suffers from maximum coverage. The major challenge in using the mutation testing tool is the generation of numerous mutants. Random sampling and subset are the techniques used in existing tools for mutant reduction which suffers in selecting effective mutants in testing the quality of test suite. The issues mentioned above are addressed in this article.

3. PROBLEM STATEMENT

Smart contracts have emerged as a crucial technological advancement, facilitating secure and decentralized functionalities across diverse fields such as banking, supply chain management, and governance. With these applications' increasing intricacy and significance, a corresponding need arises for thorough testing of smart contracts. The test case generated using a different tool has to be tested using the mutation testing tool which is executed on different platforms. There are no existing tools available for smart contracts which perform both test case generation and quality assessment using mutation testing in the same framework. The environmental setup for generating test cases and mutation testing takes time. All types of test suites cannot be executed by the available mutation testing tool as it is compatible with one type of test suite. The central challenge revolves around refining the construction of test suites for smart contracts, aiming to enhance coverage and efficiency while minimizing vulnerabilities and bugs. Quality assurance of the test suite generated for the smart contracts is challenging while using the available mutation testing tool.

4. AUTOMATED MUTATION ANALYSIS TOOL (AMAT)

This section provides the procedure to integrate the automatic test case generation tool with the mutation testing tool. The different construction processes are considered to generate higher code coverage, vulnerability detection and mutation analysis. Detailed descriptions are given in the following sections.

4.1 Proposed Architecture

The overall architecture of this paper is shown in Figure 1 which contains three modules. The user interface is provided to input the smart contracts for which the test suite is generated, and the test suite's quality is verified using the mutation testing technique. The test case generator uses various steps to generate test cases with better coverage using the proposed Genetic algorithm, and the mutation testing module constructed with the predictive model is used to assess the test suite quality.

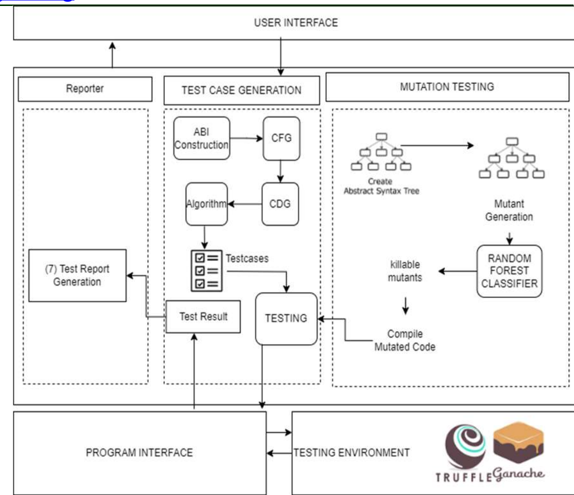


Figure 1: Architecture Diagram

4.2 Generation of Test Case for Smart Contracts

The test case generation follows the steps depicted in Figure 2.

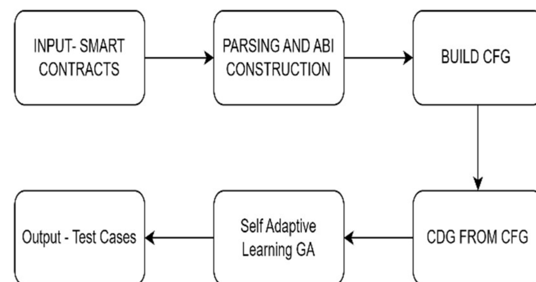


Figure 2: Automatic Test Case Generation

1) The various smart contracts chosen from projects relevant to DeFi ecosystem are considered as initial contracts.
2) To extract the information required for generating and executing the test cases parsing is done and Application Binary Interface (ABI) is constructed for the smart contract.
3) To visualize the execution paths in the smart contract CFG(Control Flow Graph) is built.
4) To represent the dependencies of various control elements in the smart contract, CDG (Control Dependency Graph) is constructed

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

using CFG. CDG helps in identifying inconsistencies and control dependent issues.

5) Self-Adaptive learning Genetic Algorithm is proposed to ensure better coverage in branch, function and line of code.

Algorithm 1: Pseudocode of Enhanced GA

1) Start
2) Initialization of Parameter
3) Evaluation of fitness value f_v
4) while (f_v <max_generation)

for (b $= 1: P_S$)
Selection of Mutation
Update the mut_strat using the equation

$$EP_N = EP'_N/(EP'_1+\ldots+EP'_4)$$

Update the speed and pos by selected mut_strat
if(new_pos of solution >current_pos of solution)
Update new_pos of solution
end if
if (new_pos of solution >global_best solution)
Update new_pos of solution
end if
if (new_pos of solution > prev_best solution)
Update new_pos of solution
end if
end for
for (b $= 1: P_S$)
Update Acc_N
end for
if (f_v $mod\ Gen_{f_v} = 0$)
for (N $= 1: 4$)
Update EP_N using equation $EP'_N = (1-\eta)EP_N + \eta Acc_N/Gen_{f_v}$ $EP_N = EP'_N/(EP'_1 + \cdots + EP'_4)$
Assign $Acc_N = 0$
end for
end if
$f_v = f_v + 1$
end while

5) Test cases are generated
6) Stop

4.2.1 Algorithm explanation

Due to applying some complex problems, the GA model is stuck in a local optima dilemma where the optimal solutions are impossible to reach. Therefore, Enhanced GA is used to minimize local optima and improve global search capabilities, increasing the diversity of solutions. The GA has the potential to learn on its own to increase robustness. The following tactics are part of the proposed GA model: 1) by adaptively modifying the algorithm parameters utilizing linear and non-linear techniques 2) Develop various population plans 3) uses multiple populations rather than just one. 4) GA incorporates a bio-inspired mechanism 5) include mutation update techniques with the enhanced GA algorithm [22].

4.3 Mutation Testing

Mutation testing comprises the following steps as shown in Figure 3,

1. The initial contract is mutated with the defined mutation operators to generate mutated code.
2. The mutated contract is compiled for each mutant created.
3. The Mutants are then classified using a random forest classifier as effective or ineffective based on killability.
4. Testing is carried over each mutant against the test suite generated. Depending on the test results, the mutants can be killed or survived.
5. The killed and total mutants generated are used to calculate the Mutation Score (1).
6. The final mutation score for the test suite is reported as a test report.

4.3.1 Classifier

The classifier based on binary classification machine learning algorithm is used here to classify effective mutants that are effective based on killing capacity of the mutants. The machine learning algorithm contains various binary classification algorithms like SVM, Naïve bayes, Decision tree, Random Forest etc., The proposed model is built using random forest as it gives higher accuracy in classifying the mutants than other algorithms. The classifier model greatly reduces the time taken for executing the mutants.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

Experiment Environment: Conducted experiments in a computer system with Intel(R) Core(TM) i7-1260P 2.10 GHz, 16.00GB RAM, and Windows 11. Python is used to build the model for GA, truffle and ganache is used to execute the Solidity smart contracts.

5.2 Evaluation

The tool is evaluated for the performance of the proposed approach using several metrics: average

code coverage rate, path uniqueness rate, execution rate, false positive rate, false negative rate, test case generation time, precision, recall and mutation score.

1. *Number of test cases (*TC_n*):*

Test cases describe actions needed to validate specific functions during software testing. Each test case includes inputs and expected outputs for verifying the actual program output. The Wilcoxon rank-sum test assesses the significance of the number of several test cases. During program testing, the number of test cases achieved by one algorithm may differ from that of other algorithms, either greater or smaller. TC_n is calculated using eq 2

$$TC_n = FP_n \times 1 \cdot 2 \quad (2)$$

Where FP_n is the number of functional points.

2. *Average code coverage (AC):*

Code coverage serves as a metric for gauging the extent to which the source code has been tested while assessing the test suites' effectiveness. This metric quantifies the proportion of code within the application exercised by the generated test cases, with a higher value indicating more comprehensive coverage and better-quality testing. AC is evaluated based on equation (3) as,

$$AC = \frac{n_{exe}}{TC_n} \times 100 \quad (3)$$

where N_{exe} is depicted as the number of lines executed, and TC_n is denoted as the total number of generated test cases.

3. *Path uniqueness rate (PR):*

This assessment encompasses the distinct execution paths traversed by the generated test cases, indicating the evaluation of the rate of path uniqueness. This uniqueness rate is determined by calculating the ratio of the count of unique execution paths (N_u) to the total number of test cases generated TC_n. PR is calculated using eq (4).

$$PR = \frac{N_u}{TC_n} \times 100 \quad (4)$$

4. *Test case generation time (TGT):*

The duration required for generating test cases for smart contracts is called test case generation time, with a shorter duration reflecting superior performance.

5. *False positive rate (*FP_r*):*

The false positive rate is a metric that erroneously categorises a secure execution path as vulnerable. A false positive (FP) refers to those execution paths that are inaccurately identified as vulnerable, whereas true negatives (TN) indicate accurately identifying the safest execution paths. The FP_r is computed by,

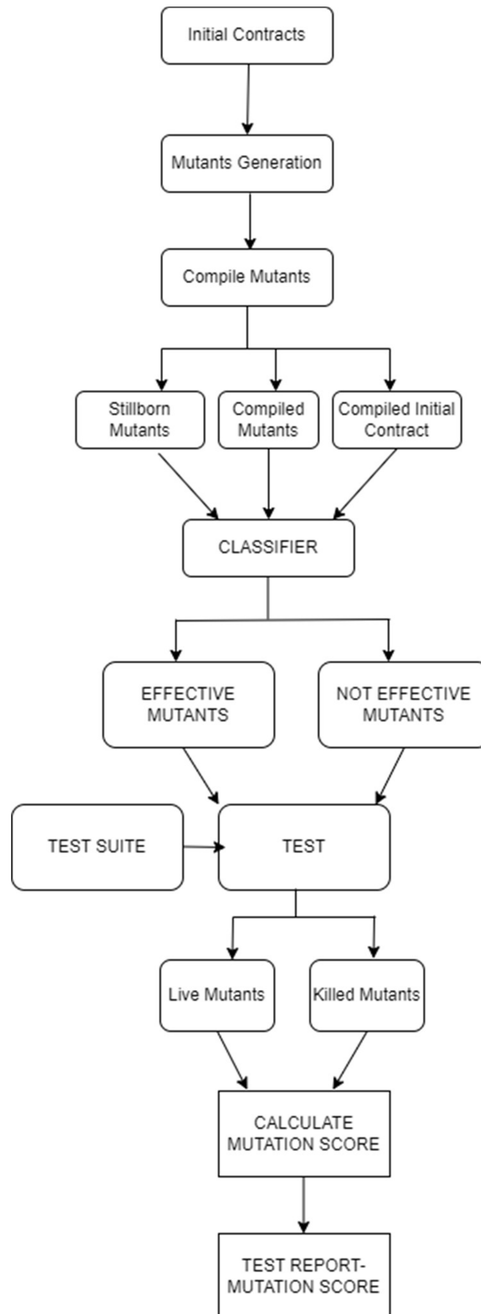


Figure 3: Experimental Process-Mutation Testing

$$FP_r = \frac{FP}{FP+TN} \quad (5)$$

6. *False negative rate (*FN_r*):*

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

When an insecure execution path is incorrectly labelled as secure, it is referred to as the false negative rate. This rate is calculated using the equation provided below,

$$FN_r = \frac{FN}{FN+TP} \quad (6)$$

From the above-mentioned equation, the terms FN and TP are represented as false negative and true positive values.

7. *Precision (P_V):*

This metric gauge the ratio of identified vulnerabilities that are indeed genuine. True positive values accurately forecast the vulnerable execution paths. Precision values are assessed using the equation provided below:

$$P_v = \frac{TP}{TP+FP} \quad (7)$$

8. *Recall (R_r):*

This measures the proportion of actual vulnerabilities correctly detected by the test cases. Higher values indicate that the test cases are effective at detecting vulnerabilities. The recall rate is determined by,

$$R_r = \frac{TP}{TP+FP} \quad (8)$$

9. *Mutants Generated:*

The total number of mutants generated using the mutation operators.

10. *Mutants Killed:*

Killed mutants are the total number of injected mutants or flaws identified by the test suite.

11. *Mutation Score:*

The ratio of number of mutants killed to the total number of mutants generated gives the mutation score. A mutation score reaching 100% implies that the test suite quality is good, while less than 100 % tells test suite requires improvement.

$$MuScore = \frac{Total\ Mutants - Live\ Mutants}{Total\ Mutants} \times 100 \quad (1)$$

5.3 Results

This section provides a discussion of the experimental results. Table 1 provides the results of the testing tool. For each smart contract, the table shows the total number of mutants generated, the number of mutants killed, and the Mutation Score for both the existing and proposed tools. The results, in comparison with the existing and proposed tool, can be discussed regarding test case generation metrics like average code coverage, execution time, recall, precision, number of mutants generated, mutants killed and mutation score obtained.

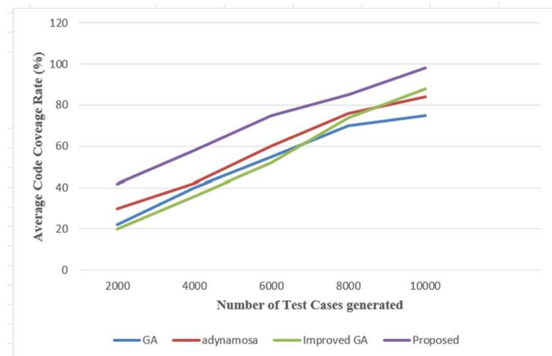


Figure 4: Average Code Coverage

Figure 4 analyses the average code coverage rate based on the quantity of generated test cases. The proposed technique, GA, Improved GA, and aDynaMOSA, are among the several test case-generating methods included in the comparative study. A higher code coverage rate indicates a greater possibility of identifying coding faults and ensuring proper application behaviour. The average code coverage rate is assessed by determining the areas of the code that are not covered by the created test cases. The average code coverage rate and the amount of generated test cases tend to rise. The proposed test case generation model had the highest average code coverage rate of all the approaches tested. In particular, the suggested strategy achieved an average code coverage at the 10,000th test case. This shows that the suggested method effectively covers a substantial amount of the code, improving the identification of potential programming problems. The findings demonstrate the suggested test case generation model's superior performance in reaching a high average code coverage rate compared to the other approaches considered during the analysis.

The analysis of execution time based on the quantity of generated test cases is shown in Figure 5. The suggested self-adaptive learning GA approach, aDynaMOSA, GA, and Improved GA are all included in the comparison. The amount of time needed to generate each test case is called execution time. Particularly for smart contracts, the execution

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 1: AMA Tool Results – Mutation Testing

AUTOMATED MUTATION ANALYSIS TOOL						
Smart Contract	**No. of Mutants Generated**		**No. of Mutants Killed**		**Mutation Score**	
	Existing	**Proposed**	**Existing**	**Proposed**	**Existing %**	**Proposed %**
Core_Fi_V3	858	515	458	455	53.38	88.38
INS	1819	1091	978	968	53.77	88.69
Rootkit_finance	1024	614	505	567	49.32	92.29
Straight_Fire_Finance	853	512	378	412	44.31	80.50
ThriftToken	1374	824	574	735	41.78	89.16
WOLF	2190	1314	1098	1190	50.14	90.56
Migrations	117	70	69	62	58.97	88.32
HelloEthSalon	51	31	29	28	56.86	91.50
hashforether	77	46	44	44	57.14	95.24
origin	269	161	112	145	41.64	89.84
rubixi	1371	823	478	792	34.87	96.28
timelock	191	115	99	98	51.83	85.51
IdentityManager	872	523	369	505	42.32	96.52
LotteryMultipleWinners	598	359	356	335	59.53	93.37
MultiSigWallet	1021	613	694	560	67.97	91.41
Identity	994	596	385	545	38.73	91.38

time tends to increase along with the amount of generated test cases. According to the analysis, the proposed self-adaptive learning GA model runs faster than alternative techniques for creating test cases. The proposed model specifically obtained an execution time of 20 seconds at the 2000th test case and a duration of 104 seconds for the 10000th test case. This shows how effective the suggested method is at producing test cases.

In Figure 6, the evaluation of the recall rate depending on the quantity of generated test cases is shown. Different methodologies are compared, including GA, aDynaMOSA, Improved GA, and the proposed method. The recall metric gauges how well the models perform to accurately identify vulnerabilities. The data shows that the recall rate improves for all approaches as the number of test cases generated rises. This suggests that the models are better able to identify vulnerabilities with a larger amount of test scenarios correctly.

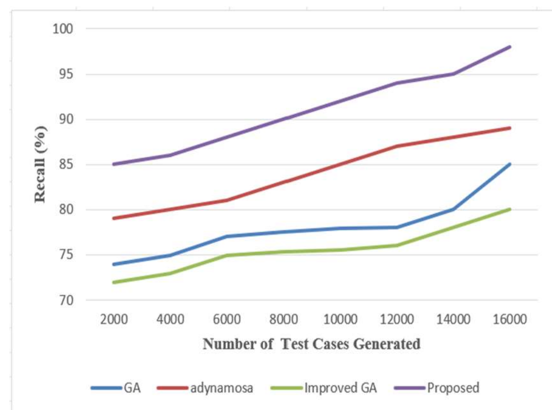


Figure 5 Execution Time

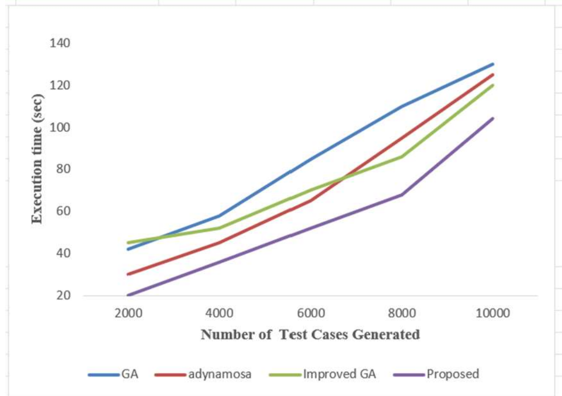


Figure 6 Recall comparison

The proposed self-adaptive learning GA strategy outperforms the other methods among all those assessed regarding recall rate. The proposed method specifically obtains a recall rate of 85% at test case 2000 and a recall rate of 98.2% at test case 16000. These findings demonstrate how well the suggested self-adaptive learning GA technique accurately identifies vulnerabilities. The suggested method's higher recall rates show that it can identify more vulnerabilities, improving the system's overall security.

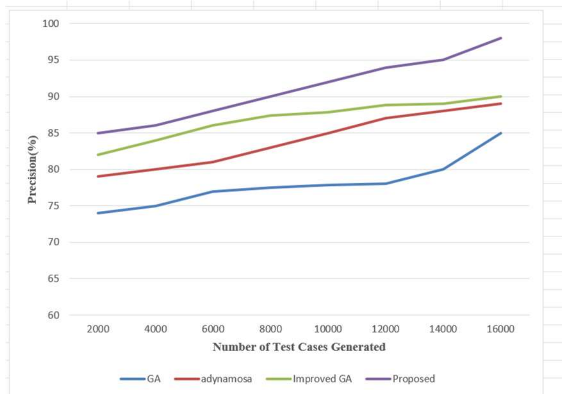


Figure 7 Precision

Figure 7 shows the study of the precision rate dependent on the number of test cases produced. Precision provides more dependable and precise results by measuring how closely the generated results resemble the original values. Compared to previous comparison approaches, the study shows that the suggested test-case-generating process provides a greater precision rate. In particular, the precision rates for GA, aDynaMOSA, Improved GA, and the suggested approach are 92%, 92.8%, 91%, and 98%, respectively, at the 16000th test instance. These results show that the suggested test case generation approach outperforms the other methods taken into account in the analysis regarding the precision of results.

The classifer used in the predictive model helps in reducing the number of mutants for execution. The figure 8 shows the classification report for classifying the effective mutants. Various binary classification is used for the comparison of the performance of the classifier. Random Forest provides accurate results and thus used in the proposed method.

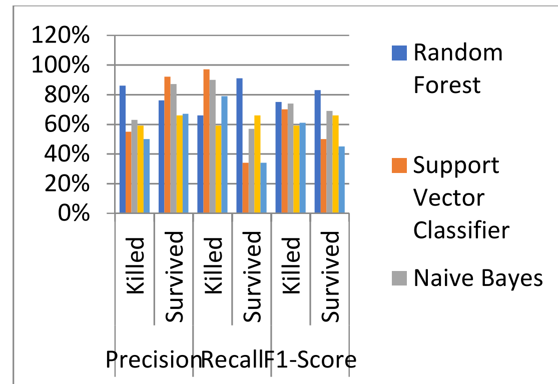
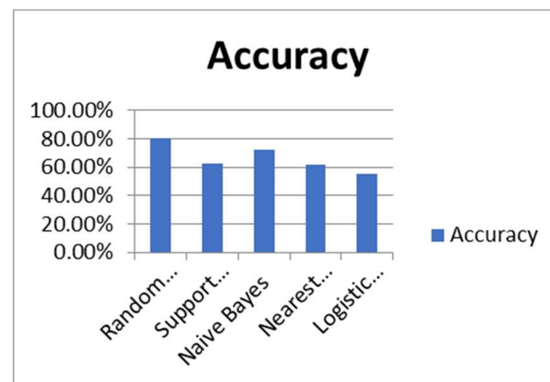


Figure 8 Classification Report

The accuracy obtained by the model for various binary classification algorithms are given in the figure 9. Based on the accuracy obtained random forest is chosen for the model in AMA tool.

Figure 9 Classifier's Accuracy



The analysis based on the number of mutants generated for the smart contracts under study is shown in Figure 10. The proposed technique is compared with one of the existing mutation testing tools. Based on the number of lines of code and operators available in the smart contract, the mutation operators are used to generate mutants.

The number of mutants generated plays a vital role in the performance of the mutation testing tool. The more mutants, the execution time for each mutant increases. The analysis shows that the

© 2023 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

proposed technique considers fewer mutants for execution than the existing method. The reduction of mutants is based on the effective mutant classification, which is the new concept introduced into the mutation testing of smart contracts instead of using random sampling and clustering. This helps in reducing the time taken to execute the mutants.

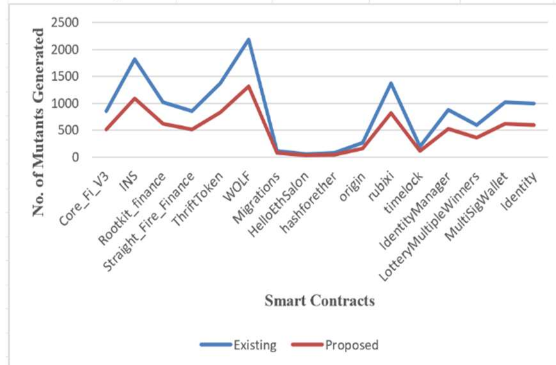


Figure 10: Number of Mutants Generated

The mutants killed count assesses the test suite quality. The test suite must be able to identify the mutants in the mutated source code. Once it identifies the flaw, it said as mutant is killed. The number of mutants killed is calculated based on number of mutants executed. The analysis between the existing and proposed techniques in terms of the number of mutants killed for the various smart contracts under study is shown in Figure 11. The analysis shows that the test suite generated using the proposed self-adaptive GA is capable of identifying the mutants. The number of mutants killed is increased with the proposed tool when compared with the existing one, shows the effectiveness of the test suite generated.

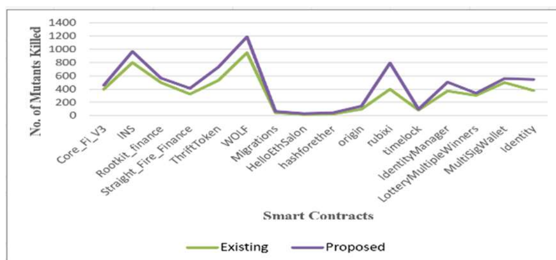


Figure 11: Number of Mutants Killed

The test suite quality is assessed using the mutation score calculated using the formula given in equation (1). The ratio between the number of mutants killed to the total mutants generated. The mutation score is the metric that reveals the test suite quality. The higher the mutation score better the quality of the test suite. Lower the mutation score insist the improvement in the test suite generated. The proposed integrated tool with the test suite generation using the self-adaptive GA and machine learning-based mutation testing shows that the mutation score achieved is better than the existing method is shown in figure 12. The analysis is based on the various smart contracts and the mutation score achieved for each smart contract.

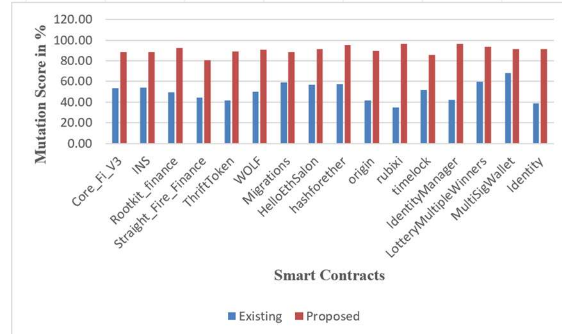


Figure 12: Mutation Score

6. CONCLUSION

The difficulties with autonomous test case development and mutation testing in smart contracts are discussed in this article. Proposed a comprehensive mutation analysis method to address the issues. The proposed tool's performance is assessed using various measures, including average code coverage, execution time, false positive and false negative rates, recall, precision, and mutation score. Other approaches, like GA, aDynaMOSA, Improved GA, and SuMo, are used for the comparative analysis. The experimental research showed that the suggested tool performed better in terms of code coverage, branch coverage in terms of test case generation and reduced number of mutants in mutation testing for smart contracts. The tool achieved average code coverage rates of 98%, 98.1% recall, 98.4% precision, and 96% mutation score. The developed AMA tool is hence useful in testing the smart contract by generating the test cases automatically and assessing quality of the test cases by using mutation score within the same framework. This reduces the time involved in experimental setup of two different tools for generating the test cases and assessing the quality.

In future planning, the proposed integrated tool will be applied to a wider collection of commercial smart contracts. Additionally, in order to improve the efficiency and performance of the test case-generating process, it is intended to investigate parameterisation with other search techniques. To enhance the automation, dependability, and effectiveness of testing smart contracts in multiple

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

sectors by broadening the scope and introducing new algorithms.

REFERENCES:

[1] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, Apr. 2008, [online] Available: https://bitcoin.org/bitcoin.pdf.

[2] Z. Zheng, S. Xie, H. Dai, X. Chen and H. Wang, "An overview of blockchain technology: Architecture consensus and future trends", Proc. IEEE Int. Congr. Big Data, pp. 557-564, 2017.

[3] N. Szabo, "The idea of smart contracts", Apr. 1997, [online] Available: https://nakamotoinstitute.org/the-idea-of-smart-contracts/.

[4] D. Siegel, Understanding the dao attack, https://www.coindesk.com/ understanding-dao-hack-journalists, [Online; accessed 18-Dec-2019].

[5] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, A. Hobor, Finding the greedy, ´ prodigal, and suicidal contracts at scale, in: Proceedings of the 34th Annual Computer Security Applications Conference, ACM, 2018, pp. 653–663.

[6] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang and Z. Chen, "MuSC: A Tool for Mutation Testing of Ethereum Smart Contract," 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1198-1201, doi: 10.1109/ASE.2019.00136

[7] Y.Ivanova A.Khritankov, "Regular Mutator: A Mutation Testing Tool for Solidity Smart Contracts", 2020, Science Direct, Procedia Computer Science, Volume 178, Page No. 75-83.

[8] Chapman, Patrick, et al. "Deviant: A mutation testing tool for solidity smart contracts." 2019 IEEE International Conference on Blockchain (Blockchain). IEEE, 2019.

[9] Honig J.J., Everts M.H., Huisman M. "Practical Mutation Testing for Smart Contracts",2019. In: Pérez-Solà C., Navarro-Arribas G., Biryukov A., Garcia-Alfaro J. (eds) Data Privacy Management, Cryptocurrencies and Blockchain Technology. DPM 2019, CBT 2019. Lecture Notes in Computer Science, vol 11737. Springer, Cham. https://doi.org/10.1007/978-3-030-31500-9_19

[10] Barboni, Morena, Andrea Morichetta, and Andrea Polini. "SuMo: A mutation testing approach and tool for the Ethereum blockchain." Journal of Systems and Software 193 (2022): 111445.

[11] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, A. Dinaburg, Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts, arXiv preprint arXiv:1907.03890v3.

[12] V. Wustholz, M. Christakis, HARVEY: A Greybox Fuzzer for Smart Monsieur ¨ Contracts, arXiv preprint arXiv:1905.06944v1.

[13] S. So, S. Hong, H. Oh, SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution (2021). URL http://prl.korea.ac.kr/smartest

[14] S. W. Driessen, D. D. Nucci, G., D. A. Tamburri, W.-J. Van Den Heuvel, Automated Test-Case Generation for Solidity Smart Contracts: the AGSolT Approach and its Evaluation, arXiv preprint arXiv:2102.08864v2

[15] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649-678, Sept.-Oct. 2011, doi: 10.1109/TSE.2010.62.

[16] Zhang, Lu, et al. "Is operator-based mutant selection superior to random mutant selection?." Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. 2010.

[17] Papadakis, Mike, et al. "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique." 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1. IEEE, 2015.

[18] Driessen, S., Di Nucci, D., Monsieur, G., Tamburri, D.A. and Heuvel, W.J.V.D., 2021. Automated test-case generation for solidity smart contracts: the agsolt approach and its evaluation. *arXiv preprint arXiv:2102.08864*.

[19] Ji, S., Zhu, S., Zhang, P., Dong, H. and Yu, J., 2022. Test-Case Generation for Data Flow Testing of Smart Contracts Based on Improved Genetic Algorithm. *IEEE Transactions on Reliability*.

[20] Olsthoorn, M., Stallenberg, D., Van Deursen, A. and Panichella, A., 2022, May. SynTest-solidity: automated test case generation and fuzzing for smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (pp. 202-206).

[21] S. Ji, S. Zhu, P. Zhang, H. Dong and J. Yu, "Test-Case Generation for Data Flow Testing of Smart Contracts Based on Improved Genetic Algorithm," in IEEE Transactions on Reliability, vol. 72, no. 1, pp. 358-371, March 2023, doi: 10.1109/TR.2022.3173025.

[22] Tan, Z., Tang, Y., Huang, H. and Luo, S., 2022. Dynamic fitness landscape-based adaptive mutation strategy selection mechanism for differential evolution. Information Sciences, 607, pp.44-61

[23] Zhang, Jie, et al. "Predictive mutation testing." Proceedings of the 25th International Symposium on Software Testing and Analysis. 2016.