International Journal of Engineering Trends and Technology
ISSN: 2231 – 5381 / https://doi.org/10.14445/22315381/IJETT-V70I9P228

Volume 70 Issue 9, 280-289, September 2022

Original Article

Analysis on Mutation Testing Tools for Smart Contracts

R. Sujeetha[1], C. A. S. Deiva Preetha[2]

[1,2]*Computer Science and Engineering or SRMIST Vadapalani, Tamil Nadu, India.*

[1]*Corresponding Author : sr7092@srmist.edu.in*

Received: 27 June 2022 Revised: 16 September 2022 Accepted: 26 September 2022 Published: 30 September 2022

Abstract *- Smart contracts are codes for executing transactions over the blockchain. Smart contracts play a major role in executing the transactions, which are immutable in nature, and avoid third-party involvement in transactions. Smart contracts are developed in many languages. One of the most popular languages is solidity. This paper focuses on smart contracts written using solidity. Smart contracts are vulnerable and have faced huge losses like with DAO attacks. Smart contracts require exhaustive testing to avoid such loss. Testing is to be qualified; hence mutation testing is the right choice. Mutation testing for smart contracts focuses on vulnerability detection by inserting faults in the code. Also qualifies the test suite executed against a smart contract. The pros and cons of various tools available for this purpose are discussed in this paper. Finally suggested improving the tools to perform the mutation analysis of smart contracts better.*

Keywords *- Mutation Operators, Smart Contract, Solidity.*

1. Introduction

Mutation testing is an error seeding method used at the unit testing level that helps evaluate the test suite effectiveness for a source code. The mutation testing is performed by making slight modifications in lines of code like a syntactic change or an operator change etc., using mutation operators [1]. The operators are applied to the original code, and the modifications are called mutants. Once the mutants are generated, the program is revised to a new version known as the mutated version of the original code. Afterwards, the mutated program is tested against the test suite created for the original code. The results are analyzed. If the Tester finds the results vary from the original code tested, he makes inferences that there are syntactic errors in the code; if the results are the same compared to the original code testing, there is a need to improve the test cases.

One of the white-box testing techniques is mutation testing, which received numerous opinions for the huge investment involved. Several studies have proved that mutation testing is effective in testing the individual application units for the boundary or coverage. It concludes that mutation testing is most effective compared to branch and statement coverage test cases [4]. The strongest comparison of mutation testing with data flow concluded that mutation testing is the strongest [5]. The real program size with errors is difficult to identify appropriately [6]. Researchers started to induce faults to create faulty versions of correct code. These faults are induced manually or even automatically. The automatic version is a coding variant that applies operators to the program. These operators are called mutation operators; the variant resultant is known as mutants and named mutation generation or just mutation. The steps involved in identifying which test suite is faulty and analyzing the mutation failures are called mutation analysis. The mutation generation is advantageous as it helps generate more mutants to produce a statistically significant result.

Proposed 7 research questions and answered those through their experimental analysis [6]. They compared the cost-effectiveness of coverage criteria, like minimal required level of coverage, comparing the coverage criteria to random test suites. The results showed that mutation analysis could assess and differentiate new testing techniques if any. In terms of cost, this study concludes that removing mutants based on their predictive performance by a set of validation suites aids in attaining significant results. Mutation testing can be used at the unit testing level, integration testing level and also at the specification level. It can also be used at the programming level in other software development life cycle phases, like design. Using a design level encourages the designers to improve the design quality by applying Finite state machines, state charts, Petri Nets, Network Protocols, and Web services, to name a few.

The blockchain is featured with special code that helps automatic triggering of lines of code mentioning the action to be taken when the condition is met, called a smart contract. The smart contract plays a major role in the development of dApps used in many domains like healthcare, education, government services etc., The development of smart contracts is processed separately and

deployed over the blockchain network. The nature of blockchain and smart contracts are immutable; hence once the deployment is done, it cannot be modified for versions. Needed to deploy a new smart contract where the older one is of waste. The smart contract to be deployed should be defect free.

The smart contract with defects caused a huge financial loss of $16 billion in the DAO attack that happened in the year 2016. This is caused due to a source code vulnerability in smart contracts. Such attacks must be avoided by performing exhaustive unit and integration testing.

One such test is mutation testing which helps identify the defects in source code. Currently, many research works are carried out in smart contracts to provide secured smart contracts with good quality source code. Vulnerability detection is a field of work done in this domain. The smart contract development standards are required as having for software development SDLC.

The research on mutation testing for smart contracts is available from 2019. Various tools are developed for this purpose. The tools are developed to be specific to the language used for developing smart contracts and use a limited set of mutation operators specific to solidity language and traditional operators.

There is a need for the generic use of mutation operators that can be applied to any smart contract. The selection of mutation operators is not effective in identifying the defects. While executing all the operators, some might not provide effective smart code under test. There is a need for a technique to select the mutation operators.

This paper is organized as follows: section 2 analyses the outcomes and discusses the tools available by prior studies to offer an updated picture of the existing research. Section 3 presents the smart contracts and requirements for testing the smart contracts and also lists the vulnerabilities present in smart contracts and issues associated with them. A description of the tools that are available for mutation testing of a smart contract is presented in section 4. Section 5 provides the results and discussion on the tools available, and finally, section 6 finishes the main conclusions of this study.

2. Related Work

2.1. Mutation Testing in SDLC

Mutation testing applications are discussed in detail. The mutation testing can be used as an assessment tool and involves test case generation, prioritization, unit test level, and structural testing levels [7]. Mutation testing most probably begins with the presumption of the "Competent Programmer Hypothesis" (introduced by Demillo et al. [8] in 1978): "The competent programmers create programs that are close to being correct." According to this supposition, the bugs that qualified programmers put into their codes are straightforward errors that can be fixed with a few straightforward syntactical changes.

Following the previous idea, mutation testing frequently modifies the source code in a minor syntactical way; therefore, the introduced flaws are minor and bear a resemblance to errors made by "competent programmers". In terms of defect revealing capabilities, many experimental and observational investigations have demonstrated that mutation testing is substantially more efficient than other test adequacy criteria [9], [10], and [11]. The review provided by [7] inspired a much more detailed discussion of how mutation testing can be applied in various testing activities and what works are being contributed by other researchers for the equivalent mutant detection and cost reduction methods. Also, the characterization data of mutation operators help classify and compare the operators used in experiments.

2.2. Mutation Testing Process

The mutation process introduces minor modifications to lower the errors occurred during coding. Mutation operators are a form of rules that compares the data and provides a suitable environment to generate mutants. There are three types of mutations: decision, value and statement. The process is performed as given in below Fig. 1.

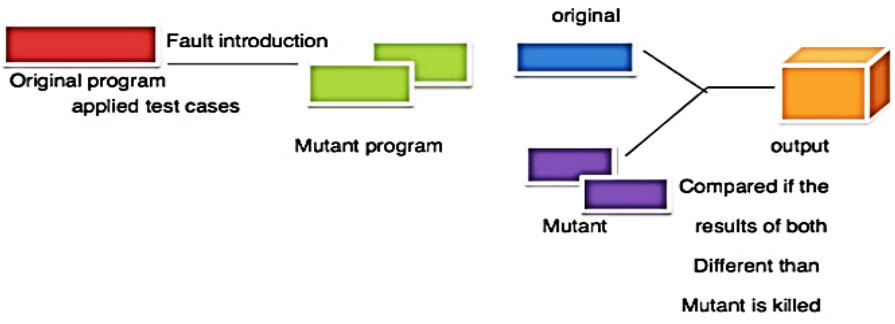

Fig. 1 Mutation Process

The process includes the following steps. Firstly, faults are introduced into the actual code producing variants called mutants. Every mutant with one fault makes the mutant unsuccessfuland validates the efficiency of test cases. Next, the test cases are executed on the mutated program, and the actual code finds the faults. Once the errors are detected, the actual and mutant code outputs are compared. If the output of actual and mutant code is not similar, then mutant is executed by the test cases. If it is similar, its syntactically different, operationally the same, or the test case is insufficient to identify mutants.

2.2.1. Mutation Operators

The cost of computing mutation score becomes very large when performed with more mutation operators generated automatically [12]. Hence the author advised identifying a minimal number of mutation operators sufficient for determining the test suite's success. Employeda statistical approach to identify a selection of mutation operators and a linear model that accurately predicted mutation scores. Addresses the challenges of mutation analysis, like a time-consuming and computational cost while using many lines of code [13]. Concentrates on reducing the generated mutants by using a reduced but required set of mutants for mutating COR and ROR operators. Also proposes an optimized execution flow that uses redundancies and execution time differences of test cases to order it again and split the respective test suite. The combination of no redundant operators and prioritization information about the execution and coverage of tests thus reduces the cost involved in mutation analysis greatly by 65%.

2.2.2. Mutant Selection Techniques

The mutation testing generates a greater number of equivalent and redundant mutants. Equivalent mutants are unkillable and hence are no different from the original code, whereas redundant mutants are killed largely. The greater number of mutants generated by mutation testing makes industries avoid its use. The researchers indicate redundant mutants are less likely to affect the work effort of the test engineers, whereas equivalent mutants are more likely to affect the work effort linearly. A minimal number of operators is sufficient for measuring the mutation score [14]. Predictor variables help to predict a required variable for selecting the subset of operators.

Statistical techniques such as greedy algorithms like forwarding selection and least angle regression can predict such subsets. Other methods like elimination-based correlation analysis and cluster analysis can also be used for subset selection. An evaluation method like cross-validation is used to evaluate the statistical method chosen for subset selection. The cross-validation procedure helps researchers to estimate the effectiveness of the test suite for a code with the subset of mutants instead of considering all mutants.

2.2.3. Cost Reduction Techniques

Mutation Sampling

Mutation sampling takes a tiny sample of the mutations created and performs mutations depending on that sample set. Mutant sampling is a cost-cutting method aimed at reducing the number of mutants. It comprises a subset of the mutants created and performed at random [18]. Sahinoglu and Spafford [19] proposed a sampling method based on the Bayesian consecutive frequency ratio test to calculate the mutant ratio. This novel method selects a selection of mutations to be analyzed randomly until a mathematically appropriate number of observations is reached.

Random Selective Mutation

Random Selective Mutation (RSM) [15] reduces the number of mutation operators from the total mutants generated, assuming that the mutants selected the small number of operators would produce a small count of mutants which will be enough to conduct the required testing. RSM is carried out in two steps. It starts by selecting an application and calculating a mutation score for each operator. Finds the operators with less than 50% of mutation scores to consider only the test effective operators and puts them into subset 1. Other operators are discarded.

In the next step, the application size is computed based on it, and the operators are selected from subset 1 and filled in subset 2. Mutants are generated from subset 2. This approach is compared with strong mutation and selective mutation. Results showed that the mutation score got using selective mutation, strong mutation, and RSM are similar to the use of 10 operators on average by the RSM method.

RSM Method saves the mutation cost while maintaining a similar mutation score and test effectiveness. Evaluated the savings using the percentage of saving measures. RSM demonstrates that a limited number of mutation operators can be used for mutation testing. It's also likely that the mutation operator used impacts mutant detection effectiveness.

Do-Fewer, Do-Smarter, Do-Faster Approach

The do-fewer approach runs a small number of mutants without considering any information loss. Selects a subgroup of mutants derived from the created mutants. This strategy is followed by preferential mutation and mutant sampling [16]. Developing a series of cost-cutting algorithms, the do-faster technique seeks to produce and execute variants as quickly as possible. [18]. Two strategies for getting things done faster are mutation analysis based on the schema and independent compilation. Finally, the do-smarter strategy aims to spread processing costs across multiple implementations [17]. The do-smarter technique is

well-exemplified by weak mutation and distributed architectures. According to previous research, commonly used mutation operators produce 40% to 60% of all mutants [19]. Test cases that kill other mutation operators frequently kill mutants created by these operators as well. This novel technique presents a strategy for identifying a smaller subset of mutation operators that saves the most money while maintaining full mutation effectiveness.

3. Smart Contract

Smart contracts are computer code automatically executes full or part of terms in an agreement and saves on a blockchain platform like Ethereum. The code controls the transaction execution that is trackable and irreversible. Smart contracts can be used for fund transfers between the parties. They are a more efficient, cost-effective, and secure method of executing and administering agreements. Smart contracts have several flaws that must be addressed for them to receive widespread adoption.

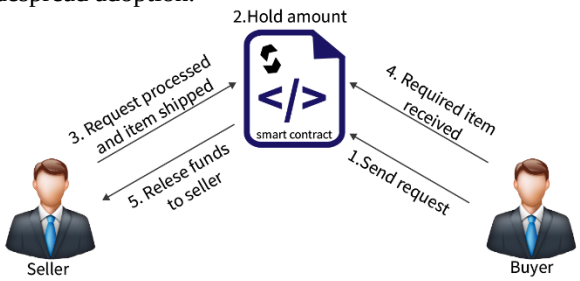

Fig. 2 Smart Contract Working

These include the technical complexity of making updates and the incapacity to process complex transactions. Large organizations can use smart contracts, including medicare, logistics management, and banking sectors.

3.1. Working of Smart Contract

Computing protocols known as smart contracts enable the electronic assessment, management, and implementation of contracts. The blockchain technology underpinning smart contracts conducts every transaction in a contract without the use of a mediator. Smart contracts are written using simple if/then phrases. Once the predefined parameters are met and accepted, the operations are carried out by a dispersed network, as shown in Fig. 2. These tasks include sending alerts, releasing payments to the right recipients, and issuing a ticket. When a transaction occurs, the blockchain is updated, so it can't be changed, and the results are only available to those who have been assigned access. There might be as many requirements as necessary in a smart contract to satisfy the parties that the job will be executed correctly.

3.2. Testing for Smart Contract

The development of smart contracts aids in the automatic facilitation, verification, and enforcement of many untrustworthy parties' negotiations and agreements. On the other hand, smart contracts raise several worries about security threats, weaknesses, and legal challenges. After the DAO attack, it became a challenging task for the developers even to create a simple, smartcontract. Unit tests and formal specification for smart contract verification is becoming mandatory.

The unit test coverage is not always satisfactory, and formal specification becomes complicated, similar to implementation. To ensure the quality of the smart contract, it is required to check everything in the code using mutation testing. We review the major flaws that could cause big issues in smart contract applications. Vulnerable patterns in smart contract execution and code are caused by reentrancy concerns, difficulties with temporal constraints, failure managing, and transaction ordering dependencies. Before launching their live Ethereum or other blockchain platform contracts, developers should be aware of these vulnerabilities and rigorously undertake quality assurance test cases.

The below table1 tabulates a few of the vulnerabilities associated with attacks and security issues related to that vulnerability. The tools like securify, Oyente, MAIAN, and Zeus are used as security analysis tools for smart contracts. These tools identify the vulnerabilities present in the source code by performing static and dynamic analysis and formal verification.

3.3. Challenges in Smart contract testing

The smart contract provides solutions for organising cryptocurrencies, sensitive data and valuable assets. The complexity of the blockchain platform makes it difficult to validate and verify smart contract-based software [23]. It is necessary to ensure the reliability of the smart contract code. Deploying secure and quality contracts is the biggest concern for smart contract developers. The following are critical things to be considered in testing smart contracts. High stakes- the contracts execute valuable or sensitive data in the such scenario; if testing is not done properly, this may cause unknown behavior leading to financial loss, or attackers can drain out the cryptocurrencies after its deployed in the blockchain.

3.3.1. Complex Interaction Dynamics

Smart contracts communicate between the blockchain and the real world, providing data exchange. The smart contract performance depends on external computations, the outcome of legacy systems, and off-chain data sources. While performing testing, the association between smart contracts and interaction with outside components must be taken into account for the derivation of test cases. This is a challenging job because the computing execution of a blockchain application relies not just on the internal operations and states of the system but also on the network's state. There is a chance that many programmers aren't accustomed to evaluating their software in this

situation. For instance, the lack of software checks in the smart contract code led to assaults like the Parity Wallet and Decentralized Autonomous Organization (DAO) hacks, which cost millions of dollars each.

3.3.2. State Setting and Exploration

Due to their stateful nature, Smart Contracts present a number of testing challenges. As a result, different pre-states may result in drastically different test results. Although most testing frameworks allow the user to construct a contract, doing so is frequently laborious and time-consuming. The amount of gas used during setup procedures is another thing that Ethereum developers need to be careful of. For instance, the well-known Truffle2 framework for Ethereum offers the developer various test hooks that are each processed as a separate transaction to perform setup and teardown actions.

A testing campaign conducted in a lab setting cannot completely replicate the intricate interactions in real life. As a result, serious problems and vulnerabilities could bypass the pre-release stage and appear only after the program has been installed on the primary network. Unfortunately, it is difficult to continue testing in the live environment due to the immutability of the transactions. There is a definite demand for organized testing methods that can explore important execution states and boost testing efficiency.

3.3.3. Lack of Testing Procedure

Deploying dependable code is difficult due to the lack of established best practices and guiding methods for testing Smart Contracts. Indeed, these programmes' unconventional lifecycle is not considered by the test cases and best practices currently in use for standard software systems.

It would be beneficial to look at how switching to a different blockchain platform affects these patterns and whether they can be modified to account for the special nature of smart contracts. New testing methodologies should also be developed expressly for Smart Contracts, considering the characteristics of the execution environment underneath.

Although certain works already published [24, 25, 26] suggest and analyse best practices for blockchain-based applications, these mainly focus on the design and implementation stage. A recent set of security practices for Daps that includes addresses software lifecycle testing and deployment was put forth by Marchesi et al. [27]. In order to determine whether all pertinent patterns were implemented in the under consideration Smart Contract, the authors also present the user with various vulnerability scanning questionnaires.

Guidelines and standardised best practices are crucial to the testing process for smart contracts because of the nature of the blockchain environment. These guiding policies should also specify the testing techniques and equipment to use. Given the wide range of testing tools and platforms available today, this would be incredibly helpful for developers of Smart Contracts.

3.3.4. Test Suite Adequacy Assessment

There are currently no commonly used approaches or tools for evaluating the suitability of Smart Contract test suites [28]. Test engineers frequently use static analyzers, compute branch and statement coverage, or even do manual test inspections to enhance test effectiveness. The research group soon began paying more attention to stronger test suite sufficiency evaluation methods like mutation testing due to the business-critical nature of smart contracts.

Mutation testing is one of the best ways to enhance the effectiveness of a test suite, but it is rarely used in practice because of its high computing costs [23]. It is required to look into cost-reduction and performance analysis further to promote this technology's use in actual Smart Contract development environments.

Additionally, there aren't any tools for mutation testing that include built-in assistance for regression testing activities. Smart Contracts go through a number of changes during development, just like conventional software (i.e., code refactoring, bug fixes, new features). Existing tools should gradually update the mutation score on developing Smart Contracts to speed up mutant execution.

In the next session, the available tools for assessing the effectiveness of the test suite in finding defects in the source code of smart contracts are briefed, and their pros and cons are discussed.

4. Existing Tools

The tools for automatically creating and testing Solidity mutants are covered in this section.

4.1. Musc

Regarding the Solidity programming language, Musc provides a set of mutation operators considered for the mutation analysis. To some extent, the outcome aids in exposing smart contract flaws [21]. The truffle project provides the tool with a test suite and smart contract as input. The smart contract code is converted into AST files for mutants' generation. Solidity-parser-antlr solidity parser is used for transforming the source file into AST files. AST file is mutated and is stored as a line number where the mutant is generated. This method of storing the mutants helps reduce the storage space greatly.

Table 1. List of vulnerabilities and associated security issues

Vulnerabilities	Mechanism	Associated Attacks	Software Security Issues
Reentrancy Problem	Recursive call fromfallback function	DAO attack	Failure in storing andprotecting data
Transaction Ordering	Inconsistent transactions-orders based on time of invocations	-	Race conditions
Exception handling	Fails to check the return values on afunction call	DAO Attack, Integer overflow or Underflow attack	Failure to handle errors
Integer overflow / Underflow attack	Subtracting positiveintegers from zero results	Integer overflow orUnderflow attack	Integer range errors
No restricted write	Writes to storage variables are restrictedby the modifierprivate.	The DAO Attack, Multisig walletAttack	Failure to store and protect data.

Musc provides the user interface to view the mutants. Figure 3 gives the architecture of thetool in which it carries out the test suites by first launching the smart contract across the blockchain. Userdefined testnet is supported by Musc. During the execution of the test, compilation errors causing mutants are avoided. The outcomes are recorded, and a report that includes the mutation score and test findings for each mutant run can be generated. The performance of the tool is evaluated against smart contracts like Skincoin. The mutation operators used in Musc are effective, as per the authors.

The altered files are then run through the test suites,with the results being logged and shown at the end. Mutants that use Relational Operator Replacement and Solidity Specific Operators are likely to be alive, according to the results of the analysis. The manual analysis made changes to the original test suite and was executed again, and no new errors were discovered; thus, mutation testing helps improve the test suite's quality.

4.2. Regular Mutator

This research provided a method for increasing the stability of smart solidity contracts via mutation analysis [19]. They discovered frequent mistakes committed by contract developers based on the product. Results help in improving the test suite quality and, in turn, identify the defects present in code for smart contracts. The paper also points out the disadvantages of using source code lines as metrics to qualify the test suite. The source code lines have no connectivity with the test suite quality and do not help identify the defects. The mainadvantage of this metric is its simplicity, and the computation time is much less.

Mutations deals with errors made by coders, which helps identify the new test cases that were not taken into account earlier and thus initiates the creation of test cases for the newly identified ones. The test suite quality can be measured using the mutation concept. The ratio of the number of killed mutants to all is used as the metric for quality assessment.

Only those mutation operators that can be represented as regular expressions are considered from the common errors identified. The mutants are added to the source code using the regular expressions library, and the mutated source code is tested against the test suite to assess its quality of the test suite.

4.3. SuMO

Proposed a smart contract tool to assess a test suite's fault detection potential. The tool used 44 mutation operators considering the solidity documentation and various other mutation tools to help animate the errors made by developers [20]. Reported evaluation of tools on freelyavailable projects with test suites. The tool uses the mutation strategy, such as customizing the operators for the mutation process, which was achieved by reducing the number of mutants generated. The other strategy was to limit the redundant mutants by combining the mutants that are producing the redundant mutants. Also, the tool limited the stillborn mutants by removing those operators like those of semantic errors that are injected as faults in code which causes compilation errors.

The tool's implementation uses solidity- parser-antlr to produce the code's AST. The AST is then mutated with the selected mutation operators. Complies with the mutated code and tests each mutant generated by executing the test suites. Finally, the mutation score and the coverage values are logged into a file. On careful analyses of the survived mutants, some mutations are not reached by the test suites. The tool addresses wrong addresses assignment, address balances and event parameters not concentrated by other tools [20]. The tool requires deep analysis on how to reduce stillborn mutants. All the novel mutation operators were not evaluated by the tool. In future, experiments are further required to assess if the surviving mutants lead to identifying the faults to concentrate on the evaluation process [20].

4.4. Deviant

Deviant is a mutation testing tool for the Ethereum smart contract, designed with an objective to produce mutants of a given contract code spontaneously, to execute the tests against each mutant [24] instinctively. Deviant contributes mainly to solidity-specific features in addition to the traditional operators. The results of deviant display that the statement coverage and branch coverage of contract codes do not mandatorily guarantee code quality.

As shown in figure 3, the design selects one program at a time and converts it into AST, after which the user selected Mutation operators are applied, and mutants are generated. The Mutation operators are based on a thorough solidity language defect model. According to the tool, each mutation operator generates one or more mutants. Only one mutation is copied and compiled into bytecode from the project code. The tests are conducted against the generated bytecode.

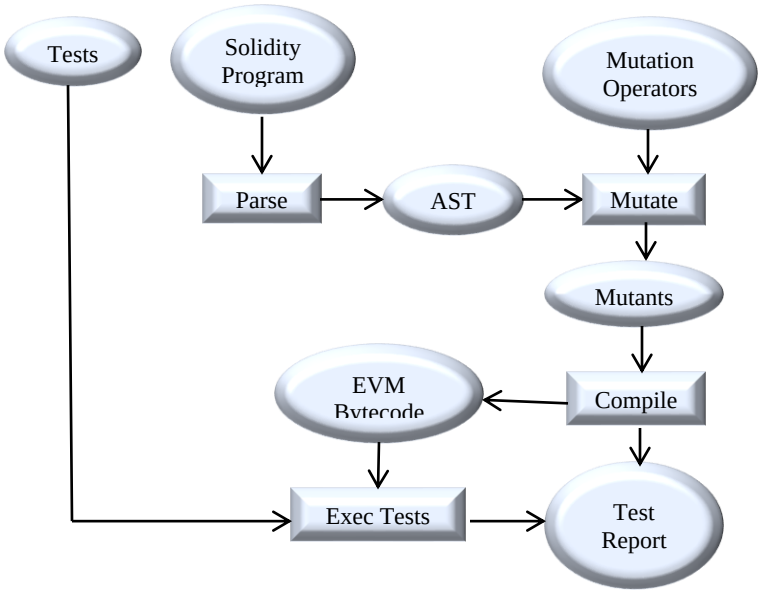

Fig. 3 Architecture of Deviant

The process is repeated for each mutant, culminating in a report that includes a mutation score, a count of died mutants, and a countof live mutants. The tool takes time to iterate for every mutant. Deviant provides a wide range of operators used to mutate the smart source contract, which other related tools do not consider.

4.5. Vertigo

The Vertigo framework for smart contracts was developed to assess the feasibility of mutation testing in the area of smart contracts built on the Ethereum platform. As shown in Fig. 4, users can launch mutation testing using the user's command line interface (CLI) [25].

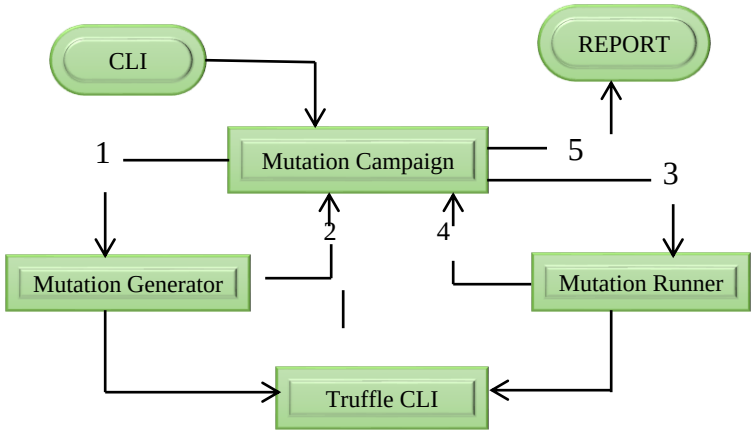

Fig. 4 Vertigo Architecture

The command line parameters can be used to configure the Ethereum networks that will be used for the test runs. Using the mutation method, vertigo can also be sampled. Vertigo, by default, alerts all mutants who a mutation campaign has impacted. Instead, a thorough report with all of the additional mutations can be saved to a file.

Table 2. Comparison of Mutation tools for Smart Contract

Tool Name	Methodology Used	Demerits
Musc	Source code is converted into AST and then mutated with the user-selected mutation operators. Then it's compiled into EVM bytecode, and tests are executed.	Several operators are missing, which could lead to serious contract risks. The operators are designed based on known bugs. The mutation score is not published by the authors.
Deviant	Source code is converted into AST and then mutated with the user-selected mutation operators. Then it's compiled into EVM bytecode, and tests are executed against one mutant at a time. Theprocess is repeated for every mutant generated by the mutation operators.	Although the majority of non-equivalent mutants aren't eliminated, they pass the branch and statement coverage checks. Produces a large number of mutations with a high computational cost.
Regular Mutator	Source code is converted into Regular Expression and then mutated with the user-selected mutation operators. Then it's compiled into EVM bytecode, and tests are executed.	The compilation error causing mutants is avoided while calculating the mutation score.
SuMO	Source code is converted into AST using solidity-parser-antlr and then mutated with the user-selected mutation operators. Then its compiled into EVM bytecode and tests are executed against each mutant generated.	More research is needed to see if the surviving mutants can also lead to the discovery of important flaws. All of the proposed novel operators could not be evaluated by the tool.
ContractMut	Implement the four solidity-specific and general mutation operators derived from the minimal standard Mothra set. Used selective mutation techniques to reduce stillborn mutants' generation.	Concentrated primarily on the four-solidity language-specific mutation operators.

The area for mutation assessment, which consists of three connected parts, is where most of the work is done. These elements carry out everything, from creating mutants to managing the testing procedure. The vertigo tool can be optimized with the tool able to detect specific tests that execute the line of code to which syntactic changes are done instead of executing a test suite for each mutant.

The above-discussed tools are available tools for mutation analysis of smart contracts that can provide the test suite quality assessment. The demerits are discussed in the next session.

5. Results and Discussion

5.1. Result Analysis

Table 3 shows the data obtained from various tools like Musc, Vertigo and Sumo. Fig. 5 compares existing tools with the parameters: number of mutants killed, number of mutants survived, number of mutants generated and the mutation score achieved by each tool.

The available tools are executed for the smart contract, and the results are compared with the parameters mentioned earlier. The tools show variations in Fig. 5 in generating mutants and killing and surviving mutants.

The surviving mutants are those that are not detected by the test suite; hence, it can be inferred that test suite quality is to be improved, or the surviving mutants may cause risks like unidentified defects. The tool must be capable of generating effective and killable mutants. The surviving mutants can be equivalent, yet another direction for future research. The mutation testing tool provides a mutation score as the final result, which determines the quality of the test suite of the smart contract under test.

Table 3. Analysis of Results of Mutation Testing Tool

Tool	No. of mutants generated	No. of mutants Killed	No. of Mutants survived	Mutation score Achieved
Musc	184	44	140	24%
Sumo	681	584	97	39%
Vertigo	391	303	30	92%
Deviant	397	147	150	37%
Regular Mutator	871	25	110	18.50%

Based on the mutation score, the test suite can be improved to provide defect-free smart contracts before deploying on the blockchain platform.

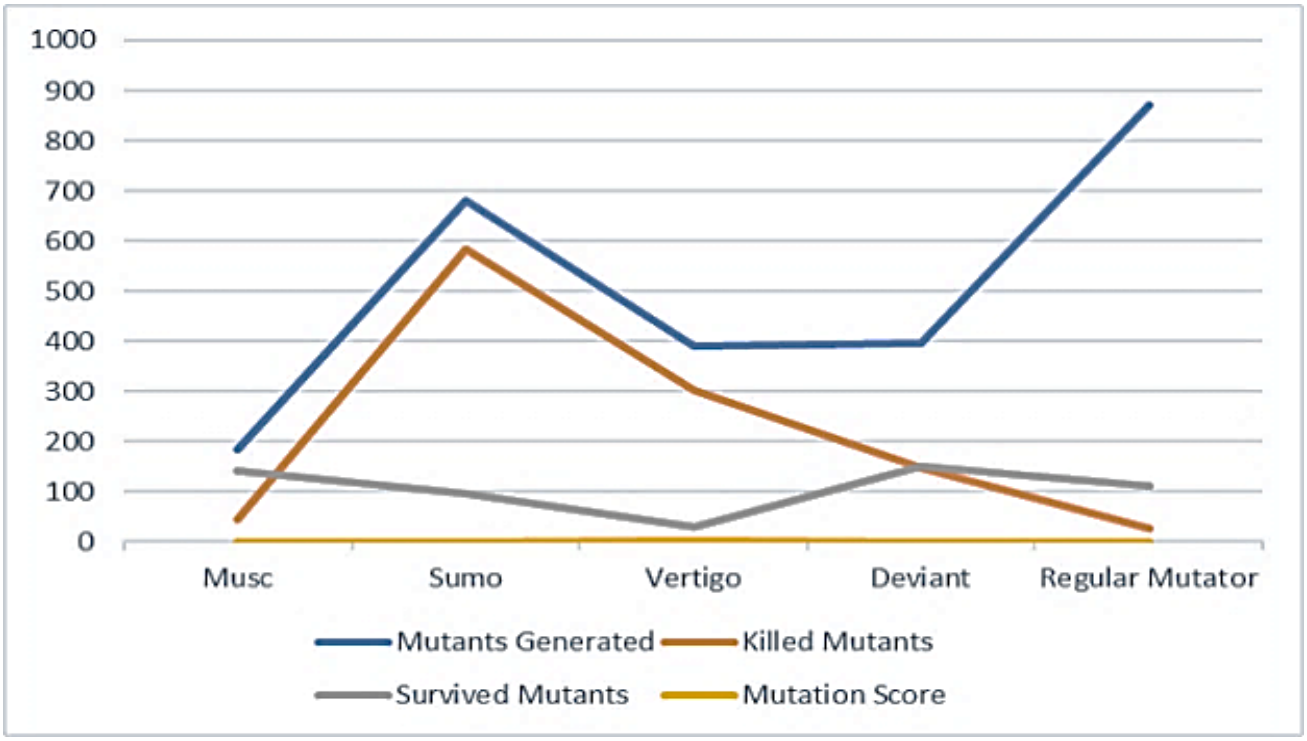

Fig. 5 Comparison of results of Existing Tools

Table 2 compares the tools available for evaluating smart contract mutations. The technique used by each tool is provided in columns 2 and column 3, giving the research gaps that can be addressed for future research that helps to provide better performing mutation testing tools for assessing the test suite quality.

6. Conclusion

On Ethereum, there are more than 34000 vulnerable smart contracts. Thus the paper demonstrates the requirement for having a better performing mutation analysis for the smart contracts. Needs developing an alternate method for mutation insertion such that they do not cause compilation errors. The high computational cost due to the increase in mutants generated and the undecidable equivalent mutant problem restricts the process of full automation of equivalent mutation analysis. The key benefit of employing mutation testing is that it is a good substitute for real problems, indicating the test suite's capacity to identify the faults introduced. The tools like musc, vertigo, sumo and deviant help the smart contract developers to qualify their test suites. Smart contract with mutation analysis helps developers to deploy bug-free ESC in the blockchain. In future, a new tool will be developed for assessing the test suite quality that considers the gaps addressed in this paper in generating the mutants and selecting the mutation operators for the smart contract.

References

[1] Munawar, H, "Mutation Testing Tool For Java," 2004.

[2] Jefferson Offutt, Jie Pan, Kanupriya Tewary, Tong Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software—Practice & Experience*, vol. 26, no.2, pp.165-176, 1996.

[3] Patrick Joseph Walsh, "A Measure of Test Case Completeness (Software, Engineering)," Ph.D. Dissertation. State University of New York at Binghamton, Binghamton, NY, USA, 1985.

[4] J. H. Andrews, L. C. Briand, Y. Labiche and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *In IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608-624, 2006, doi: 10.1109/TSE.2006.83.

[5] Zhu Q, Panichella A, Zaidman A, "A systematic literature review of how mutation testing supports test activities," *PEERJ Preprints* 4:e2483v1 https://doi.org/10.7287/peerj.preprints.2483v1, 2016.

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

[7] P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[8] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses VS mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[9] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," *in Software Testing, Verification and Validation Workshops*, 2009. ICSTW'09. International Conference on, pp. 220–229, IEEE, 2009.

[10] Siami Namin, J. Andrews and D. Murdoch, "Sufficient mutation operators for measuring test effectiveness," *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 351- 360. doi: 10.1145/1368088.1368136.

[11] R. Just, G. M. Kapfhammer and F. Schweiggert, "Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis," *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 2012, pp. 11-20. doi: 10.1109/ISSRE.2012.31.

[12] Falah, B., Akour, M., & Bouriat, S, "RSM: Reducing Mutation Testing Cost Using Random Selective Mutation Technique," *Malaysian Journal of Computer Science*, vol.28, no.4, pp.338–347, 2015. Retrieved from https://ejournal.um.edu.my/index.php/MJCS/article/view/6885.

[13] Maryam Umar, "An Evaluation of Mutation Operators for Equivalent Mutants," Master Thesis. King's Colledge, London, United Kingdom. Advisor Mark Harman, 2006.

[14] Hong Zhu, Patrick A. V. Hall, and John H. R. May, "Software unit test coverage and adequacy," *ACM Computer Survey*, vol. 29, no.4, pp. 366-427, 1997.

[15] Moohebat, M., Raj, R.G., Kareem, S.B.A., Thorleuchter, D., "Identifying ISI-indexed articles by their lexical usage: A text analysis approach," *Journal of the Association for Information Science and Technology*, vol. 66, no. 3, pp. 501–511. doi: 10.1002/asi.23194.

[16] M. Sahinoglu and E. H. Spafford, "A Bayes Sequential Statistical Procedure for Approving Software Products," *In Proceedings of the IFIP Conference on Approving Software Products (ASP'90).* Garmisch Partenkirchen, Germany: Elsevier Science, pp. 43–56, 1990.

[17] A.J. Offutt, and R.H. Untch, "Mutation 2000: Uniting the Orthogonal‖, Mutation Testing for the New Century," *W.E. Wong* (Ed.) Kluwer, 2001.

[18] 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)*, pp. 1198-1201, 2019. doi: 10.1109/ASE.2019.00136.

[19] Y.Ivanova A.Khritankov, "Regular Mutator: A Mutation Testing Tool for Solidity Smart Contracts", *Science Direct, Procedia Computer Science*, vol.178, pp.75-83, 2020.

[20] M. Barboni, A. Morichetta and A. Polini, "SuMo: A Mutation Testing Strategy for Solidity Smart Contracts," *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 50-59, doi: 10.1109/AST52587.2021.00014.

[21] P. Chapman, D. Xu, L. Deng and Y. Xiong, "Deviant: A Mutation Testing Tool for Solidity Smart Contracts," *2019 IEEE International Conference on Blockchain (Blockchain)*, pp. 319-324, 2019. doi: 10.1109/Blockchain.2019.00050.

[22] Honig J.J., Everts M.H., Huisman M, "Practical Mutation Testing for Smart Contracts," 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, 2019. Springer, Cham. https://doi.org/10.1007/978-3-030-31500-9_19.

[23] M. Barboni, A. Morichetta and A. Polini, "Smart Contract Testing: Challenges and Opportunities," *2022 IEEE/ACM 5th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 21-24, 2022. doi: 10.1145/3528226.3528370.

[24] Qinghua Lu, Xiwei Xu, Yue Liu, and Weishan Zhang, "Design pattern as a service for blockchain applications," *In IEEE International Conference on Data Mining Workshops (ICDMW). IEEE*, 128–135.

[25] Vijay Rajasekar, Shiv Sondhi, Sherif Saad, and Shady Mohammed, "Emerging Design Patterns for Blockchain Applications," *In ICSOFT*. ScitePress, 242–249.

[26] Xiwei Xu, Cesare Pautasso, Liming Zhu, Qinghua Lu, and Ingo Weber, "A pattern collection for blockchain-based applications," *In Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pp. 1–20.

[27] Lodovica Marchesi, Michele Marchesi, Livio Pompianu, and Roberto Tonelli, "Security checklists for ethereum smart contract development: patterns and best practices," arXiv preprint arXiv:2008.04761.

[28] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu, " Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*.

[29] G. Petrović, M. Ivanković, G. Fraser and R. Just, "Does Mutation Testing Improve Testing Practices?," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 910-921, 2021. doi: 10.1109/ICSE43902.2021.00087.