White-Box Mutation Testing of Smart Contracts: A Quick Review

Afef Jmal Maâlej(✉) and Mariam Lahami

ReDCAD Laboratory, National School of Engineers of Sfax, University of Sfax, BP 1173, 3038 Sfax, Tunisia
{afef.jmal,mariam.lahami}@redcad.org

Abstract. Once being deployed on the blockchain, smart contracts cannot be altered, requiring more testing. A fault-based testing technique called mutation testing (MT) can significantly increase the utility of a test for smart contracts. MT is a type of white-box testing which is mainly used for unit testing. In fact, certain statements of the source code are changed to check if the test cases are able to find errors in source code. The main objective of MT is ensuring the quality of test cases in terms of robustness in the way that it should fail the mutated source code. In this paper, our goal is to identify and classify the main applications of mutation testing of smart contracts by providing a quick review on the application perspective of mutation testing based on a collection of several papers. In particular, we analysed in which quality assurance processes mutation testing of smart contracts is used, which mutation tools and which mutation operators are employed.

Keywords: White-Box testing · Mutation Testing · Blockchain · Smart Contract

1 Introduction

Blockchain is a modern technology that has revolutionized the way society interacts and trades [18]. It might be described as a network of distributed, decentralized blocks that store data with digital signatures. This approach was initially used to develop digital currencies like Bitcoin and Ethereum. However, recent research and commercial studies have concentrated on the chances that blockchain offers in a variety of other application fields to benefit from this technology's key qualities, such as decentralization, persistency, and anonymity. Healthcare [13], internet of things [15,17] and vehicles [12,14] are just a few of the industries that employ blockchain.

In this context, smart contracts are computer programs implementing business logic that manage the data or assets on a blockchain environment. Although they have been introduced several years ago, the development of smart contracts is still challenging for developers. The latter usually produce vulnerable code which can lead to huge monetary losses. Therefore, it is essential to ensure

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
B. Ben Hedia et al. (Eds.): VECoS 2023, LNCS 14368, pp. 135–148, 2024.
https://doi.org/10.1007/978-3-031-49737-7_10

that smart contracts do not contain such vulnerabilities. The most important verification and validation technique for detecting both semantic errors and vulnerabilities is software testing. Testing smart contracts is even more crucial than testing regular programs, since their source code once deployed on the blockchain cannot be altered or changed due to their immutable nature. Furthermore, it is highly demanded to evaluate the quality of the tests and improve their adequacy. A powerful approach that can perform such assessments is *Mutation Testing* (MT). Indeed, this test technique consists on injecting faults into a given program to check the fault-detection capabilities of test suites [16].

A wide range of papers make use of this mature fault-based software testing technique to detect functional bugs and vulnerabilities in smart contracts because it is widely studied for over four decades. Thus, several approaches and tools are introduced in order to increase confidence on smart contracts [8,19,21]. However, we noticed the absence of surveys that include work done on mutation testing of smart contracts and give researchers new trends and challenges in this emerging research line.

Therefore, this paper presents a quick review that surveyed the most relevant studies related to MT of smart contracts dated from 2019. Particularly, we tackle the following main research questions:

– **RQ1:** What are the methodologies, approaches and tools based on mutation testing to verify smart contracts ?
– **RQ2:** Which mutation operators are mostly used by the studied approaches?

The answers to these questions help researchers to understand the studied topic, to identify the challenges in this research area and their solutions and also to discuss future directions. To do so, we first chose four well-known scientific and electronic databases (ScienceDirect Elsevier[1], ACM Digital Library[2], SpringerLink[3] and IEEE Xplore[4]) with the aim of extracting the most relevant papers related to our research topic. Second, we used the following search keywords which were the same in all databases: *"Mutation Testing AND Smart contract" OR "Mutation Testing AND Blockchain"*. Then, the selection of articles was performed by removing irrelevant articles after checking their titles and their abstracts and after being fully read we selected **14** as primary studies.

The remainder of this paper is organized as follows. Section 2 provides key concepts related to mutation testing and smart contracts. Related reviews and surveys are discussed in Sect. 3. Next, we investigate in Sect. 4 the most relevant researches on mutation testing applied in the context of smart contract verification. Finally, in Sect. 5, we conclude with a summary of paper contributions, and we identify possible areas of future research.

[1] https://www.elsevier.com.
[2] https://portal.acm.org.
[3] https://www.springerlink.com.
[4] https://www.ieee.org/web/publications/xplore/.

2 Background Materials

2.1 Mutation Testing

The use of mutation testing in software testing has the potential to improve software quality. Indeed, it is defined as a testing technique that injects faults into a program by creating several versions, each one contains one semantic fault. These faulty programs are named *mutants*. The generation of mutants is called mutation and that semantic fault is called mutation operator (MO). There are several traditional mutation operators that depend usually on programming languages such as deleting a statement, replacing boolean expressions, replacing arithmetic, and replacing a variable [24].

As highlighted in Fig. 1, the mutation testing process can be explained simply in following steps:

1. Given a program P and a set of test cases T.
2. Produce the mutant $P1$ from P by inserting only one semantic fault into P.
3. Execute T on both P and $P1$ and save results as R and $R1$.
4. Compare the output of mutant program $R1$ to the expected output R:
   (a) If $R1$ is not equal to R (i.e., $R1 \neq R$), the test cases detect the faults and the mutant is killed.
   (b) If $R1$ is equal to R (i.e., $R1 = R$), this can be due to the inefficiency of the test cases or the equivalence[5] of the mutant to the original program.
5. Calculate the mutation score (MS) which is the number of killed mutants divided by the total number of mutants, multiplied by 100. A mutation score of 100% means the test was efficient.

The process of adding test cases, examining expected output, and executing mutants continues until the threshold proposed by the tester is satisfied.

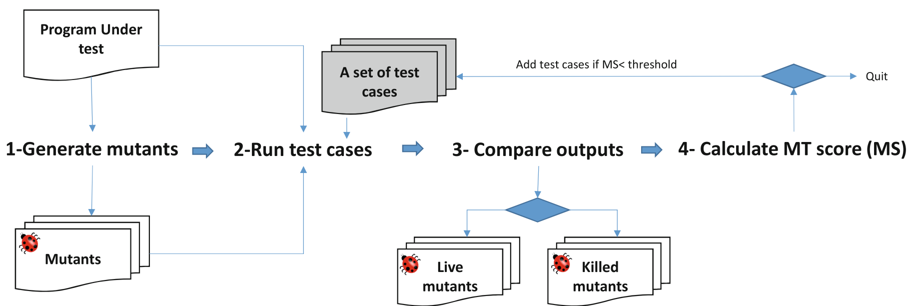

Fig. 1. Mutation Testing process.

[5] An equivalent mutant is a mutant, which is functionally equivalent to the original program.

2.2 Smart Contracts

Smart Contracts (SCs) are one of the most interesting features that have been introduced by several blockchain platforms with the purpose of managing digital assets and attaching business logic code to transactions. A SC is seen as a special program that was designed to be deployed on the distributed ledger, the blockchain. Without the need of third parties, SC is executed when some events occur allowing for making irreversible transactions. In the case of Ethereum[6], blockchain developers make use of a Turing complete language called Solidity[7] to implement Ethereum smart contracts. Similarly to JavaScript, the solidity language supports features like user-defined types, libraries and inheritance. By using the *solc*, the solidity compiler, smart contracts are compiled to the Ethereum Virtual Machine (EVM) bytecode.

As highlighted in Listing 1.1, a code snippet of a smart contract is given. A solidity smart contract is a collection of code (i.e., its functions) and data (i.e., its state variables) which is stored in a particular address on the Ethereum blockchain. In the first line, we specify the compiler version, then the keyword contract declares the contract with its name. In line 3, a state variable called "numbers" is declared as $mapping(address => uint)$. Mapping data structure in solidity acts like a hash table in which data are stored in the form of key-value pairs. Mappings are used here to associate each Ethereum address with its lucky number. Next, several functions are defined either to modify the state variable "numbers" by adding a new address with is associated lucky number or to read the lucky number of a given address.

```
1  pragma solidity ^0.8.17;
2  contract LuckyNumber {
3    mapping(address => uint) numbers;
4    function setNum(uint _num) public {
5      numbers[msg.sender] = _num;
6    }
7    function getNum(address _myAddress) public view returns (uint)
         {
8      return numbers[_myAddress];
9    }
10   function addNumbers(address _myaddress, uint _num)  public {
11     numbers[_myaddress]=_num;
12   }
13 }
```

Listing 1.1. Code snippet of the LuckyNumber smart contract.

It is highly demanded to ensure the correctness and the security of smart contracts before deploying them since executing transactions from boggy smarts contracts can lead to significant financial loss. Meanwhile, testing is one of the most important verification and validation techniques for ensuring software qual-

[6] https://ethereum.org/en/.
[7] https://solidity.readthedocs.io/.

ity. Especially, mutation testing was widely applied in the context of smart contract to check test suite adequacy and their ability to detect defects as more as possible.

3 Related Reviews

Many researchers are now interested in applying mutation testing technique to enhance software quality especially in the context of Blockchain oriented applications. Indeed, we have found recent surveys and methodical literature reviews that concentrate on dressing a literature review either on static testing [20] or dynamic testing of smart contracts [21].

A systematic review was introduced in [20] and it presented static analysis tools for Ethereum blockchain smart contracts. In this review, authors surveyed 86 papers that are published between 2016 to 2021. Among them only one paper dealt with mutation testing, that introduced the Musc tool [23].

A comprehensive survey on blockchain testing was presented in [22]. The authors mentioned academic articles on the subject of testing blockchains. Since it concentrated on static testing, dynamic testing, and formal verification, it had more scope than our work. It included only 6 papers that dealt with mutation testing.

Similarly, authors in [21] published a survey in which they provided a classification of 20 studies according to the accessibility of smart contract code. Among these papers (written from 2017 to 2021), only 6 of them focused on mutation testing and showed that this testing technique has a good effect on smart contract quality.

To the best of our knowledge, there are no current surveys that give thorough investigations connected to the issue of mutation testing of smart contracts and fully list the quantity and quality of relevant research results. Except the survey in [25] which investigated efforts on mutation testing tools while giving the pros and cons of them. The studied tools are only five: MuSC [23], SuMo [4], Deviant [5], Vertigo [10] and RegularMutator [11].

Compared to all these cited surveys, our review focuses on recent research effort by identifying methodologies and tools in this emerging field, assessing them, and highlighting both their difficulties and the unexplored areas that need more study.

4 Mutation Testing of Smart Contracts

In this section, we describe the different 14 selected papers dealing with white-box mutation testing of smart contracts. Figure 2 illustrates the year-wise analysis of the studied papers. It is clear that the increasing interest of academic research on mutation testing is rising over the years. The trend of using this technique to check the quality of test suites has a constant evolution from 2019 until 2022. Also, Fig. 3 highlights the classification of the selected primary studies by types.

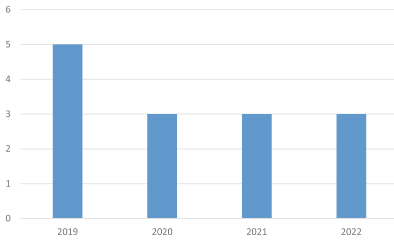

Fig. 2. Year-wise analysis of the selected studies.

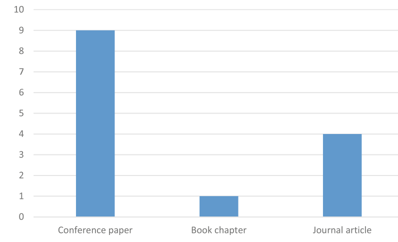

Fig. 3. Analysis of the selected primary studies by type.

First of all, the authors provide in [4] a mutation testing approach and an associated fully working tool for Solidity smart contracts. For simulating a variety of conventional and Solidity-specific vulnerabilities, SuMo includes a complete set of mutation operators. Seven of the eleven innovative mutation operators were created with a focus on Solidity's distinctive features by means of the research of the Solidity documentation and the available tools. SuMo introduces mutation operators in particular that focus on the overloading mechanism. The SafeMath library, global blockchain variables, function modifiers, cryptographic global functions, enums, return values, and explicit conversions are other areas where SuMo adds additional operators.

ReSuMo [3] provides a regression testing and mutant selection technique to the SuMo tool [4] to accelerate mutation testing on evolving projects without affecting the accuracy of testing results. During a regression mutation testing initiative, ReSuMo chooses a selection of smart contracts to mutate and a subset of test files to run again using a static, file-level technique. ReSuMo continually updates the results of mutation testing while taking into account the results of the previous program version; this allows it to accelerate mutation testing on evolving projects without reducing the mutation score. The authors should concentrate on examining additional fine-grained regression mutation testing methodologies in order to increase ReSuMo's efficacy, particularly for small and tightly-coupled smart contract projects. The tester would then be able to define a preferred level of computation granularity.

The authors specifically address in [27] the EtherFuzz mutation fuzzy testing technique to find TOD (Transaction-Ordering Dependent) vulnerabilities in smart contracts. They create test cases for the intelligent contract using the ABI (Application Binary Interface), test the byte code of the contract using TOD to find vulnerabilities, then they change the tested data to create new test cases. After recording the execution of the smart contract, the fuzzy test process is regulated until the vulnerability is identified.

The study in [26] proposes five mutation operators specifically for integer overflow vulnerability and applies mutation testing to the integer overflow vulnerability test in Ethereum smart contracts. According to the empirical research,

mutation operators are able to produce these vulnerability mutants and assess the appropriate testing tools. The experiment choice of ERC20 token smart contracts, however, could skew the findings in one direction.

Besides, the authors of [6] present AGSOLT (Automated Generator of Solidity Test Suites). They use two search algorithms to automatically construct test suites for standalone Solidity smart contracts, taking into account some of the specific challenges of the blockchain. However, the used data set is not typical of Solidity smart contracts in general, although showing each of the characteristics that are indicative of the identified blockchain specific issues.

In [7], the authors provide a tool called TestSmart that can create a set of efficient test cases for Ethereum smart contracts automatically. It is made up of a module to generate test suites, a module for generating mutants, and a module to select test cases based on the mutants. The generation of the test suite was performed using the Manticore tool. An expansion of the Universal Mutator was used for mutant generation; it comprises the mutation operators previously introduced for Solidity smart contracts. The test cases against the mutants were examined using the Manticore API. The inability to generate test cases with transactions generated neither by the contract owner nor the attacker is a significant weakness of Manticore.

The authors suggest in [11] applying mutational analysis to enhance Solidity smart contracts reliability. They developed a RegularMutator tool for mutation analysis after finding widespread errors in the source code of existing contracts. However, it took around 50 h of machine time to complete the experiments. Actually, mutation analysis is computationally challenging, which prevents it from being useful in some contexts. Additionally, a large number of mutants that survived the experiment need to be manually checked and analysed.

In addition, the authors of [9] evaluate the efficiency of large-scale smart contract mutation testing. They select among the available specific mutation operators for smart contracts, assess their effectiveness in regards to killability, and identify critical vulnerabilities that can be exploited by the mutations. The authors only take into account a replay test suite, which is less efficient than other testing methods and can yield a higher mutation score. The objective of this work was to develop a mutation-based test quality assurance approach that can also act as a starting point for other testing techniques, even though there are better testing methods.

In [2], the authors provide a mutation-based testing system for smart contracts written in the Solidity programming language. They reviewed a comprehensive list of known Solidity smart contract faults and developed 10 classes of mutation operators that were designed based on the actual errors. Furthermore, they added mutation operators to the Universal Mutator tool, enabling it to automatically produce mutants for Solidity-written smart contracts.

The development of a mutation testing framework and its application to the field of smart contracts were both studied by the authors in [10]. They demonstrated how developers may use mutation testing to evaluate the effectiveness of their test suite and make improvements to it in order to make it more efficient.

They also produced a tool called Vertigo, which should identify the precise tests that cover the line on which the mutant causes a syntactic change rather than executing the complete test suite for each mutant. The final result of tests that do not include this line should not be impacted by the modification.

The authors describe in [1] a fully automated method, called SolAnalyser, for Solidity smart contract vulnerability detection that combines static and dynamic analysis. The proposed SolAnalyser tool can be expanded to handle different vulnerability types and allows the automatic detection of 8 different vulnerability classes that are currently underrepresented in existing technologies. In addition, the authors included a fault seeding tool that introduces various vulnerabilities into smart contracts. However, by enhancing the quality of the generated inputs, SolAnalyser precision can further be improved.

The challenge of Ethereum smart contracts test generation was described by the authors of [28] as a Pareto minimization problem. Minimizing uncovered branch coverage, time costs, and gas costs are three objectives that are taken into consideration. Then, in order to identify test suites, the authors suggest a multi-objective strategy based on randomness and NSGA-II (a representative multi-objective genetic algorithm).

The authors of [23] introduce MuSC, an Ethereum Smart Contract (ESC) mutation testing tool. It facilitates autonomous processes including building test nets, deploying them, and running tests, besides it has the capacity to quickly produce large numbers of mutants. With regard to the Solidity ESC programming language, MuSC implements a number of unique mutation operators in particular. As a consequence, it can expose the defects of smart contracts to a certain degree. However, there are several issues that need to be enhanced, like handling errors.

Deviant, a mutation testing tool for Solidity smart contracts, is presented in [5]. It generates mutants of a particular Solidity project automatically and analyses each mutation against the specified tests to determine its efficiency. Deviant offers mutation operators for all of Solidity's special features in accordance with the Solidity fault model, in addition to conventional programming constructs, that simulate various problems in Solidity smart contracts. Using Deviant, the authors evaluated the effectiveness of the tests for three Solidity projects. The findings show that these tests have not yet attained high mutation scores and that a test suite that meets the requirements of Solidity smart contracts for statement and branch coverage does not always guarantee the highest level of code quality.

We highlight that all the previously introduced papers are depicted in Table 1 such as:

- Column **Paper:** refers to the surveyed paper.
- Column **Tool:** refers to the name of the proposed testing tool (if it exists).
- Column **Testing objective:** refers to the aim behind performing mutation testing.

- Column **Number of MO:** refers to the number of adopted mutation operators in each paper (if mentioned), considering the known real bugs made by smart contract developers.
- Column **MS calculation:** refers to the mutation score calculation or not for each paper.
- Column **Number of vulnerabilities:** refers to the number of weaknesses that malicious actors can exploit in smart contracts (if mentioned).
- Column **Year:** refers to the publication year of each paper.

Table 1. Surveyed approaches on white-box mutation testing of smart contracts.

Paper	Tool	Testing objective	Number of MO	MS calculation	Number of vulnerabilities	Year
[3]	ReSuMO	Regression testing	44	Yes	Not mentioned	2022
[27]	EtherFuzz	Security testing	14	No	1	2022
[26]	No proposed tool	Security testing	5	Yes	1	2022
[4]	SuMo	Functional testing	44	Yes	6	2021
[6]	AGSolT	Functional testing	Not mentioned	No	Not mentioned	2021
[7]	TestSmart	Security testing	57	Yes	Not mentioned	2021
[11]	RegularMutator	Security testing	6	Yes	3	2020
[9]	ContractMut	Scalability testing	14	Yes	7	2020
[2]	Extension of the Universal Mutator tool	Security testing	57	No	8	2020
[10]	Vertigo	Functional testing	6	Yes	Not mentioned	2019
[1]	SolAnalyser	Security testing	Not mentioned	No	8	2019
[28]	No proposed tool	Performance testing	3	No	Not mentioned	2019
[23]	MuSC	Security testing	15	Yes	Not mentioned	2019
[5]	Deviant	Functional testing	61	Yes	Not mentioned	2019

To respond the first research question **RQ1** and as presented in Table 1, both [26] and [28] did not propose testing tools as an automation of their solutions, while the other approaches implemented their frameworks in form of different mutation tools, even for the majority source codes are open on GitHub.

Besides, each surveyed approach focuses on a specific testing objective using mutation testing. It could be about:

- *Regression testing* [3]: concerns testing existing software applications to make sure that a change has not broken any existing functionality.
- *Security testing* [1,2,7,11,23,26,27]: concerns a cybersecurity technique that organizations use to identify, test and highlight vulnerabilities in their security posture.
- *Functional testing* [4–6,10]: concerns a type of testing that seeks to establish whether each application feature works as the software requirements.

- *Scalability testing* [9]: concerns a testing of a software application to measure its capability to scale up or scale out in terms of any of its non-functional capability.
- *Performance testing* [28]: concerns evaluating how a system performs in terms of responsiveness and stability under a particular workload.

Besides, the authors of [6] and [1] did not mention explicitly the number of introduced mutation operators, whereas this latter criterion varies widely from 3 to 61 mutation operators among the other publications. In fact, it is up to the tester to choose the scope or specificity of the operators. Some authors prefer to introduce a specific operator for every singular change, others choose to group together similar changes into one operator. Note that the power of mutation testing is very much dependent on its mutation operators, and the operators that can mimic the real bugs can select more effective test cases.

As a response to **RQ2**, and based on a collection of most repeated bugs that may happen in the implementation of a smart contract in Solidity programming language, the majority of researchers categorize them to two groups:

1) Classic Bugs: these bugs occur in almost any programming language, from which we can mention arithmetic issues or logical bugs (inside conditions).
2) Solidity Bugs: these faults are mostly related to the Solidity programming languages, and the distributed nature of blockchain and smart contracts. Hence, it is noticed that classical mutation operators designed for general-purpose programming languages, e.g. JavaScript, are not sufficient for the Ethereum platform, and other mutation operators need to be designed to simulate the Solidity specific bugs. So, mutation operators are divided as well into two groups: **(i)** Classic mutation operators, and **(ii)** Solidity mutation operators.

In addition, 9 out of 14 papers calculate the mutation score for a set of test cases, which corresponds to the percentage of mutants killed by these scenarios, and is a metric for evaluating the effectiveness of test cases.

50% of studied works mentioned the number of treated vulnerabilities, among basically eight well-known vulnerabilities that are reported frequently in the smart contract weakness classification (SWC) registry[8]. It is about: integer overflow/underflow, division by zero, timestamp dependency, authorisation through tx.orgin, unchecked send, repetitive call function and finally out of gas.

The assessment of the selected studies is based on several criteria highlighted in Table 2. Then, the obtained results are introduced in Table 3. Only four studies achieved 100% on quality evaluation [6,9,10,27]. A wide range of papers are greater than 80%.

[8] https://swcregistry.io/.

Table 2. Quality criteria

ID	Criteria
QC1	Are the study context and objectives appropriately described?
QC2	Is the proposed approach described in detail?
QC3	Are the study findings discussed?
QC4	Is the effectiveness of the proposed approach evaluated on at least an example of case study?
QC5	Are the proposed approach limitations outlined and discussed?
QC6	Does the study include a positioning among existing related works?

Table 3. Quality assessment scores of the selected and analysed studies

Paper	QC1	QC2	QC3	QC4	QC5	QC6	Quality %
[3]	1	1	1	1	0	1	83.34%
[27]	1	1	1	1	1	1	100%
[26]	1	1	1	1	0	1	83.34%
[4]	1	1	1	1	0	1	83.34%
[6]	1	1	1	1	1	1	100%
[7]	1	1	1	1	0	1	83.34%
[11]	1	0	1	1	0	1	66.67%
[9]	1	1	1	1	1	1	100%
[2]	1	1	1	1	0	1	83.34%
[10]	1	1	1	1	1	1	100%
[1]	1	1	1	1	0	1	83.34%
[28]	1	1	1	1	0	0	66.67%
[23]	1	1	1	1	0	0	66.67%
[5]	1	1	1	1	0	1	83.34%

5 Conclusion

In this quick review, we investigated the state of art related to mutation testing of smart contracts. We included **14** studies published from 2019 to 2022 and we analysed them to provide researchers relevant information about the used mutation operators and the calculation of mutation score. Moreover, a deep classification of these studies were discussed while giving their strengths and weaknesses.

Up to our best knowledge, existing surveys focused on static analysis and dynamic testing. Our survey has a significant contribution in the literature since

it was the first one that dealt specifically with mutation testing of smart contracts. Definitely, more refinement should be accorded in the future to review the latest studies on mutation testing of smart contracts and more meaningful research questions should be proposed.

In conclusion, the study's findings showed that most research publications come from conference proceedings and all of them focused on Ethereum smart contracts written in Solidity language. We think that further research and development are needed in order to advance the state of the art in this research line. For instance, we can investigate the application of mutation testing techniques on others programming languages used for smart contracts (e.g., Serpent, Vyper, Go, etc.) and also other blockchain platforms like HyperLedger Fabric.

References

1. Akca, S., Rajan, A., Peng, C.: SolAnalyser: a framework for analysing and testing smart contracts. In: Proceedings of the 26th Asia-Pacific Software Engineering Conference (APSEC), pp. 482–489 (2019)
2. Andesta, E., Faghih, F., Fooladgar, M.: Testing smart contracts gets smarter. In: Proceedings of the 10th International Conference on Computer and Knowledge Engineering (ICCKE), pp. 405–412. The Organization (2020)
3. Barboni, M., Casoni, F., Morichetta, A., Polini, A.: ReSuMo: regression mutation testing for solidity smart contracts. In: Vallecillo, A., Visser, J., Pérez-Castillo, R. (eds.) QUATIC 2022. CCIS, vol. 1621, pp. 61–76. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-14179-9_5
4. Barboni, M., Morichetta, A., Polini, A.: SuMo: a mutation testing strategy for solidity smart contracts. In: Proceedings of the IEEE/ACM International Conference on Automation of Software Test (AST), pp. 50–59 (2021). https://doi.org/10.1109/AST52587.2021.00014
5. Chapman, P., Xu, D., Deng, L., Xiong, Y.: Deviant: a mutation testing tool for solidity smart contracts. In: Proceedings of the IEEE International Conference on Blockchain (Blockchain), pp. 319–324 (2019). https://doi.org/10.1109/Blockchain.2019.00050
6. Driessen, S., Nucci, D.D., Monsieur, G., van den Heuvel, W.: AGSoLT: a tool for automated test-case generation for solidity smart contracts. CoRR **abs/2102.08864** (2021). https://arxiv.org/abs/2102.08864
7. Fooladgar, M., Arefzadeh, A., Faghih, F.: TestSmart: a tool for automated generation of effective test cases for smart contracts. In: Proceedings of the 11th International Conference on Computer Engineering and Knowledge (ICCKE), pp. 476–481 (2021). https://doi.org/10.1109/ICCKE54056.2021.9721448
8. Hammami, M.A., Lahami, M., Maâlej, A.J.: Towards a dynamic testing approach for checking the correctness of ethereum smart contracts. In: Kallel, S., Jmaiel, M., Zulkernine, M., Hadj Kacem, A., Cuppens, F., Cuppens, N. (eds.) CRiSIS 2022. LNCS, vol. 13857, pp. 85–100. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-31108-6_7
9. Hartel, P., Schumi, R.: Mutation testing of smart contracts at scale. In: Ahrendt, W., Wehrheim, H. (eds.) TAP 2020. LNCS, vol. 12165, pp. 23–42. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50995-8_2

10. 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.) DPM/CBT -2019. LNCS, vol. 11737, pp. 289–303. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31500-9_19
11. Ivanova, Y., Khritankov, A.: RegularMutator: a mutation testing tool for solidity smart contracts. Procedia Comput. Sci. **178**, 75–83 (2020)
12. Jabbar, R., Fetais, N., Kharbeche, M., Krichen, M., Barkaoui, K., Shinoy, M.: Blockchain for the internet of vehicles: how to use blockchain to secure vehicle-to-everything (V2X) communication and payment? IEEE Sens. J. **21**(14), 15807–15823 (2021)
13. Jabbar, R., Fetais, N., Krichen, M., Barkaoui, K.: Blockchain technology for healthcare: enhancing shared electronic health record interoperability and integrity. In: 2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT), pp. 310–317. IEEE (2020)
14. Jabbar, R., Kharbeche, M., Al-Khalifa, K., Krichen, M., Barkaoui, K.: Blockchain for the internet of vehicles: a decentralized IoT solution for vehicles communication using ethereum. Sensors **20**(14), 3928 (2020)
15. Jabbar, R., Krichen, M., Kharbeche, M., Fetais, N., Barkaoui, K.: A formal model-based testing framework for validating an IoT solution for blockchain-based vehicles communication. In: 15th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 595–602. SCITEPRESS-Science and Technology Publications (2020)
16. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng. **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
17. Krichen, M.: Strengthening the security of smart contracts through the power of artificial intelligence. Computers **12**(5), 107 (2023)
18. Krichen, M., Ammi, M., Mihoub, A., Almutiq, M.: Blockchain for modern applications: a survey. Sensors **22**(14), 5274 (2022)
19. Krichen, M., Lahami, M., Al-Haija, Q.A.: Formal methods for the verification of smart contracts: a review. In: 15th International Conference on Security of Information and Networks, SIN 2022, Sousse, Tunisia, 11–13 November 2022, pp. 1–8. IEEE (2022). https://doi.org/10.1109/SIN56466.2022.9970534
20. Kushwaha, S.S., Joshi, S., Singh, D., Kaur, M., Lee, H.N.: Ethereum smart contract analysis tools: a systematic review. IEEE Access **10**, 57037–57062 (2022). https://doi.org/10.1109/ACCESS.2022.3169902
21. Lahami, M., Maâlej, A.J., Krichen, M., Hammami, M.A.: A comprehensive review of testing blockchain oriented software. In: Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2022, Online Streaming, 25–26 April 2022, pp. 355–362. SCITEPRESS (2022)
22. Lal, C., Marijan, D.: Blockchain testing: challenges, techniques, and research directions. CoRR **abs/2103.10074** (2021). https://arxiv.org/abs/2103.10074
23. Li, Z., Wu, H., Xu, J., Wang, X., Zhang, L., Chen, Z.: MuSC: a tool for mutation testing of ethereum smart contract. In: Proceeding of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1198–1201 (2019)
24. Nguyen, Q.V., Madeyski, L.: Problems of mutation testing and higher order mutation testing. In: van Do, T., Thi, H.A.L., Nguyen, N.T. (eds.) Advanced Computational Methods for Knowledge Engineering. AISC, vol. 282, pp. 157–172. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06569-4_12

25. Sujeetha, R., Preetha, C.A.S.D.: Analysis on mutation testing tools for smart contracts. IJETT J. **70**, 280–289 (2022)
26. Sun, J., Huang, S., Zheng, C., Wang, T., Zong, C., Hui, Z.: Mutation testing for integer overflow in ethereum smart contracts. Tsinghua Sci. Technol. **27**(1), 27–40 (2022). https://doi.org/10.26599/TST.2020.9010036 https://doi.org/10.26599/TST.2020.9010036
27. Wang, X., Sun, J., Hu, C., Yu, P., Zhang, B., Hou, D.: Etherfuzz: mutation fuzzing smart contracts for TOD vulnerability detection. Wireless Commun. Mob. Comput. **2022** (2022). https://doi.org/10.1155/2022/1565007
28. Wang, X., Wu, H., Sun, W., Zhao, Y.: Towards generating cost-effective test-suite for ethereum smart contract. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 549–553 (2019)