A Comprehensive Review of Testing Blockchain Oriented Software

Mariam Lahami[1], Afef Jmal Maalej[1], Moez Krichen[1,2] and Mohamed Amin Hammami[1]
[1] *ReDCAD Laboratory, National Engineering School of Sfax, University of Sfax, Tunisia*
[2] *Faculty of CSIT, Al-Baha University, Saudi Arabia*

Keywords: Blockchain, Review, Dynamic Testing, Smart Contracts, BoS.

Abstract: This work presents a short review on the most relevant studies in the context of testing Blockchain-Oriented Software (BoS), especially at the smart contract level. Focusing on dynamic testing, we first provide a classification of 20 studies according to the accessibility of smart contract code. Second, we give an overview of each identified work while highlighting its advantages and limitations. Third, we discuss challenges and opportunities in this research area. After carrying out this review, it was noticed that there is much work to be done, especially in the context of model-based testing of smart contracts which can be a good research line for detecting defects and enhancing the quality of such applications.

1 INTRODUCTION

Nowadays, one of the most popular trends of Information Technology (IT) is the blockchain (Jabbar et al., 2022). Firstly introduced in the white paper of Nakamoto et al. in 2008 (Nakamoto et al., 2008), Blockchain (BC) gathered a lot of interest in the last decade by several organizations, especially financial ones. BC is a distributed ledger that follows the Peer-to-Peer architecture in which each node maintains a copy of the blockchain ledger. Its structure makes it distinguishable since it is composed of blocks, linked together by means of cryptographic functions, in which data are securely stored and cannot be changed or tampered. All the participant nodes in the blockchain network are running with respect to a common consensus algorithm without the need of a third party, such as financial institutions. Currently, blockchain platforms, such as Ethereum[1] and Hyperledger Fabric[2], give the possibility of developing decentralized applications with complex business logic. They introduce the concept of “smart contracts” which is defined as a collection of programming code and data that runs on the blockchain and it is executed to fulfill a given task when some events and rules happened. By using smart contracts, blockchain can be used not only to exchange cryptocurrencies but also can be applied on several industry sectors such as property management, assurance and health (Fekih and Lahami, 2020). Although the blockchain technology presents a novel approach to develop decentralized applications between several stakeholders, this technology may suffer from several vulnerabilities and bugs within smart contracts. Due to the decentralized nature of blockchain involving anonymous nodes, malicious attackers may extract digital assets from a contract, cause damage by leading a smart contract into a deadlock, which prevents account owners from withdrawing or spending their assets (Praitheeshan et al., 2019). For instance, 50 million dollars were stolen in the well-known ”DAO attack”, which used a combination of smart-contract vulnerabilities (Finley, 2016). Therefore, applying testing techniques, before the deployment of smart contracts, becomes a crucial task with the aim of increasing confidence in blockchain oriented software and avoiding attacks and asset losses.

In this direction, several efforts were made to detect such vulnerabilities in smart contracts. Researchers have used different approaches including theorem proving, model checking (Nelaturu et al., 2020) and software testing (Antonino and Roscoe, 2020). Although testing blockchain oriented software (BoS) has gained the attention of many researchers, we have not found either recent surveys or systematic mapping studies that provide in depth an overview of the considered research field, and identify exhaustively the amount and quality of available research results related to it. To cope with this lack, this systematic literature review was conducted in which **20** stud-

[1] https://ethereum.org/en/
[2] https://www.hyperledger.org/

Lahami, M., Maalej, A., Krichen, M. and Hammami, M.
A Comprehensive Review of Testing Blockchain Oriented Software.
DOI: 10.5220/0011042800003176
In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 355-362
ISBN: 978-989-758-568-5; ISSN: 2184-4895

ies dealing with dynamic testing of Blockchain oriented Software (BoS), dated from 2017 to November 2021, were deeply discussed. Moreover, we identified several challenges and future research guidelines related to this topic.

The organization of this survey is as follows: Section 2 provides background information about blockchain technology, smart contracts, and the software testing techniques. In Section 3, related surveys are discussed. The state-of-the-art testing approaches and techniques for performing different types of testing on smart contracts are provided in detail in Section 4. Section 5 includes a discussion based on blockchain testing challenges and some future directions. Finally, section 6 concludes the paper.

2 PRELIMINARIES

2.1 Blockchain

The concept of Blockchain appeared for the first time in the white paper of Nakamoto et al. (Nakamoto et al., 2008) as the technology underlying Bitcoin. Now, it garnered a lot of attention and is used in several platforms such as Ethereum and Hyperledger. It can be defined as a distributed ledger maintained over a peer-to-peer network.

As illustrated in Fig. 1, BC is composed of a growing list of blocks. Each block contains essentially a given number of transactions that have occurred within the network. The transaction can be seen as token transfers or any manner of data exchange. Each block is composed of two parts: the header and the body. Within the body of the block, transactions are stored while the header contains several fields, particularly a timestamp of when the block was produced and the identifier of the previous block. The latter is obtained by executing a cryptographic hash function (e.g., SHA256, KECCAK256, etc.). In this way, blocks are connected to each other like a linked list (Ali et al., 2018).

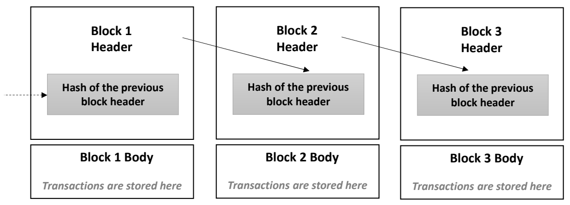
Block 1
Header
Hash of the previous block header
Block 2
Header
Hash of the previous block header
Block 3
Header
Hash of the previous block header
Block 1 Body
Transactions are stored here
Block 2 Body
Transactions are stored here
Block 3 Body
Transactions are stored here

Figure 1: Representation of a blockchain general scheme (Ali et al., 2018).

2.2 Smart Contracts

One of the interesting features of recent blockchain platforms like Ethereum and Hyperledger is the possibility of attaching business logic code to transactions, called Smart Contracts (SC). A SC is seen as a piece of code stored in the blockchain, that is capable of self execution when some conditions are met, without the need of centralized authorizing entities.

For the case of the Ethereum platform, smart contracts are written in a Turing complete language[3], called Solidity[4] and then they are compiled to the Ethereum Virtual Machine (EVM) bytecode. Like JavaScript language, Solidity supports features like libraries, inheritance and user-defined types. The most relevant feature within smart contracts is their immutability. Once deployed on the blockchain, they cannot be altered or changed. Therefore, it is highly required to ensure their correctness and security before their deployment on the blockchain platform.

2.3 Blockchain Testing Techniques

One of the most important activities for proper software development is testing activity. In fact, it is defined in (Freedman, 1991) as the process of validating and ensuring the quality of a System Under Test (SUT). It is usually performed with the aim of assessing the compliance of a system to its intended specifications. Furthermore, software testing can be static or dynamic. Static testing does not involve software execution, but examines or analyses the source code structure, syntax and data-flow, and is usually called *Static analysis*[5].

In contrast with static testing, dynamic testing involves the software execution while feeding inputs and producing outputs. Test cases are typically developed by specifying test inputs and expected outputs. The purpose of testing here is to check whether the actual outputs correspond to the expected ones. For instance, test cases can be designed to validate whether the observed behavior of the tested software conforms to its specification or not. This is mostly referred to in the literature as *Conformance testing*. Dynamic Testing can be used to verify different properties either functional or non-functional.

- **Functional Testing:** can be classified into three categories:

[3]It is worthy noting that Ethereum supports several programming languages (e.g., Vyper) and compilers.

[4]https://solidity.readthedocs.io/

[5]Static analysis can be used with formal verification methods with the aim to prove the correctness of a given software.

- Black-box testing technique: it focuses on examining the software functionalities without any knowledge of source code. It requires the system specification and it is usually performed at the integration and system level. Typical black-box testing techniques include essentially model-based testing (MBT) and fuzz testing[6] (Klees et al., 2018).
- White-box testing technique: it is the detailed investigation of internal logic and structure of the code. This technique requires a full knowledge of source code and it is usually applied at unit testing level.
- Grey-box testing technique: it provides combined benefits of black-box and white-box testing techniques. It requires a partial knowledge of internal structure of the application.

- **Non-functional Testing:** aim to check non-functional requirements, such as reliability, performance and security.

In the case of testing blockchain oriented software, we identify in the literature several kinds of testing techniques that are performed with the aim of increasing confidence and trustworthiness of BoS. For instance, we cite:

- Smart contract testing: It consists in performing detailed functional testing of business logic and process.
- Performance testing: It verifies the performance and the latency within the Blockchain network.
- Security testing: It performs security validation at transaction, block, and network levels.
- Node testing: It aims at checking the integrity of the network and the overall ledger by verifying that all newly added transactions are saved adequately.

3 RELATED SURVEYS

Verification and validation of blockchain oriented software have gained the attention of many researchers. In this direction, we have identified recent comprehensive surveys and systematic literature reviews that focus on verification of smart contracts. For example, authors in (Liu and Liu, 2019) propose an in-depth survey in which they study papers dealing with the security assurance and papers related to the correctness verification (i.e., using formal techniques such as model checking and theorem proving). Moreover, they summarize research done in this area and point out new research directions in smart contract security and correctness.

Similarly, another recent survey, dated on 2020 (Tolmach et al., 2020), analyses and classifies existing approaches to formal modeling, specification, and verification of smart contracts. First, this work outlines various types of properties from different smart contract domains (e.g., functional correctness and security). Second, it discusses the current approaches, tools and frameworks used in verifying such property specifications. It covers a wide range of papers dealing with model checking, theorem proving, program verification, symbolic execution and testing. From testing perspective, it analyses only four fuzzing-based approaches. Also, we notice that its focus is on smart contract functional behavior and does not discuss problems related to other blockchain-related execution aspects, such as performance, scalability, consensus, etc.

Authors in (Sánchez-Gómez et al., 2020) develop a systematic literature review carried out in the field of smart contract development life cycle. They review essentially the state of art of formal smart contract modeling and code generation. A reduced number of studies dealing with testing BoS were briefly discussed. We have identified another study (Koul, 2018) which underlines several challenges faced when testing BoS. It also acknowledges the need to reuse specialized tools and techniques for blockchain-oriented software testing with the aim of ensuring high levels of quality. Nevertheless, the authors do not address nor analyze the current research in this topic. To the best of our knowledge, we have not found recent surveys that provide comprehensive studies related to software testing topic and identify exhaustively the amount and quality of available research results related to it. Therefore, our proposal targets recent research work by analyzing and classifying approaches and tools in the area of software testing of BoS, discussing their challenges as well as the open research directions for future work.

4 TESTING BLOCKCHAIN ORIENTED SOFTWARE

Although testing blockchain oriented software should cover several layers : application layer (e.g., DApps), smart contract layer, blockchain layer (e.g., blocks, transactions), consensus layer and network layer, testing efforts are concentrated essentially on testing smart contracts and ensuring their correctness.

[6]Fuzzing is an automated testing technique that consists in generating random data as inputs to a software.

A wide range of papers focus on security testing and on detecting vulnerabilities with respect to the solidity language. . Up to our knowledge, only (Yuan and Zhu, 2020) proposes a novel approach that deals with performance testing of blockchain system. The proposed test tool measures several test metrics such as delay, transaction execution success rate and resource consumption (i.e., CPU usage, memory usage, etc.). Moreover, existing work on testing BoS can be categorized into static testing approaches like and dynamic testing ones. Due to space limitation, we focus in this review on studying only dynamic testing approaches.

4.1 White-box Testing Approaches

This research line is based on the investigation of internal logic and structure of the smart contract code. Up to our knowledge, the majority of studied papers deal with testing solidity source code and some of them claim that their proposals can be extended to other languages and platforms. For example, the authors in (Driessen et al., 2021) present and partially validate AGSOLT, a tool that creates test suites aiming to achieve branch coverage for Solidity smart contract unit testing. AGSOLT works with both a random testing and a guided-search approaches. It was tested on a set of real-world smart contracts from GitHub. However, AGSOLT was not validated for permissionned/commercial smart contracts.

A wide range of papers focus on mutation testing (Li et al., 2019; Ivanova and Khritankov, 2020; Wang et al., 2019b; Akca et al., 2019; Andesta et al., 2020; Hartel and Schumi, 2020) and show that this testing technique has a good effect on smart contract quality. In fact, mutation testing is considered as a fault-based software testing technique generally used to evaluate the adequacy of test cases and their fault detection capabilities. In other words, its purpose is to help testers in identifying limitations of the test suites.

In this direction, a well established approach is proposed in (Li et al., 2019) providing a mutation testing tool for Ethereum smart contracts called, MuSC. This proposal takes as input a smart contract under test and transforms its source files to Abstract Syntax Tree (AST) version. Next, it generates various mutants that implement traditional mutation operators and new ones according to the characteristics of solidity language. The obtained mutants are then transformed back to solidity source files with injected faults for compilation, execution and testing purposes. It also provides user-friendly interface to create test nets and to display test reports. The latter include execution results for each mutant (i.e., pass or fail) and the total mutation score. Similarly, authors in (Ivanova and Khritankov, 2020) developed a RegularMutator tool for mutation analysis. Its major goal is to improve the test suites in order to find defects as well as to increase the effectiveness and the fault detection capabilities of test suites. Taken as input a Truffle project, RegularMutator generates mutants for each source file in the project. Once mutant files are generated, it substitutes the original files with the mutant ones, executes project test suites, and then, the test output is analysed. The main problem within this approach is its high computational cost of executing a set of tests when generating numerous mutants.

In (Wang et al., 2019b), the authors deal with the problem of Ethereum smart contracts test generation, targeting three main objectives, minimizing uncovered branch coverage, time cost, and gas cost. They also proposed a random based and an NSGA-II based multi-objective approach to seek cost-effective test-suites. The authors lead their empirical study on a set of smart contracts in the most widely used Ethereum decentralized applications and verified that their proposed approaches could significantly reduce both the gas cost and the time cost while retaining the ability to cover branches. To ameliorate their work, the authors need to conduct more experiments and to employ more optimization objectives as maximizing mutation killed ability during the test-suite generation.

The authors present in (Akca et al., 2019) a fully automated technique, SolAnalyser, for vulnerability detection over Solidity smart contracts that uses both static and dynamic analysis. They also implemented a tool that injects different types of vulnerabilities in smart contracts and used the mutated contracts for assessing the effectiveness of different analysis tools. The main contributions in this paper concern static checks, developing of a test generator and a fault seeding tool, run-time monitoring and empirical evaluation. The authors intend to generate inputs that provide control flow coverage within functions and transactions in Solidity contracts. This tool is considered effective in terms of execution time due to the use of AST code representation. In contrast to RegularMutator (Ivanova and Khritankov, 2020), it did not compare mutation operators for their effectiveness in detection vulnerabilities and did not explore the possibility of using the results to improve tests.

In (Andesta et al., 2020), the authors propose a testing mechanism for Solidity smart contracts based on mutation testing. They analyzed a comprehensive list of known bugs in such smart contracts and designed classes of mutation operators inspired by the real faults. In this work, the authors extended the Universal Mutator tool with their proposed mutation operators to automatically generate mutants for smart contracts written in Solidity. In (Hartel and Schumi, 2020), the authors evaluate the quality of smart contract mutation testing at scale using replay tests that were automatically generated from the Ethereum blockchain. They propose a set of mutation operators on the basis of existing works and evaluate these operators at scale. The authors introduce a novel killing condition based gas limits for smart contract transactions aiming to improve the mutation score. A first limitation of this work concerns a threat to the validity of the proposed evaluation in the way that the authors only consider a replay test suite that is less powerful than other testing techniques, which might get a higher mutation score. Another limitation regarding the validity of the proposed approach might be that it is not wise to kill a mutant only based on a different gas usage since it could still be semantically equal. Even it is widely considered as an effective method of enhancing the adequacy of testing, mutation testing is challenging and hard to be used in industry due to its high cost. It requires a high understanding of the semantics of the smart contract language to generate only useful mutants.

Another promising research line is identified that consists in testing Decentralized Applications (DApp). A DApp is a Web application composed of two parts: the front-end and the back-end. The identified approaches (Gao et al., 2019; Wu et al., 2020) touch several research areas including smart contract analysis and automated Web application testing. They overcome the lack of effective methods and tools for testing DApps since the existing ones either focus on testing front-end code or back-end programs but they ignore the interaction between them. For instance, authors in (Gao et al., 2019) propose a tool called Sungari that employs first random events to infer the interaction between browser-side code and smart contracts. Second, the proposed technique generates test cases and orders them based on a data flow graph of smart contracts. In the same direction, the Kaya framework cited in (Wu et al., 2020) reduces the complexity of testing DApps by providing GUI and CLI tool. It consists of three modules: a Kaya Graphical User Interface (KGUI), Core Function Module (CFM) and Log Analyzer (LA). Within Kaya, test engineers may produce easily test cases with a simple setting and then Kaya will execute them in an automatic manner. This is done by simulating front-end events, setting Blockchain pre-states (i.e., runtime environment of test cases) and finally running smart contracts. The Log Analyzer compares the initial values of parameters in SCs and the generated outputs, then, it produces a report illustrating such analysis.

4.2 Black-box Testing Approaches

As already mentioned, Black-box testing includes several kinds of testing techniques that apply testing activities without having any knowledge of the internal structure of BoS. The most used ones in the studied context are fuzz testing and model-based testing (Krichen, 2007; Krichen, 2018).

From fuzz testing perspective, we identify the Fuse project (Chan and Jiang, 2018) which is a fuzz testing service for smart contracts and Dapp testing. It assists developers for test diagnosis via test scenario visualization. The first prototype developed in the context of Fuse project is ContractFuzzer (Jiang et al., 2018) that detects seven security vulnerabilities of Ethereum smart contracts. The proposed approach generates fuzzing inputs from the ABI[7] specification of the smart contract. It also defines test oracles for detecting the supported real world vulnerabilities within smart contracts. ContractFuzzer was performed on 6991 real-world Ethereum SCs showed that it has identified 459 SCs vulnerabilities, including the DAO and Parity Wallet attacks.

A closely approach to ContractFuzzer is sFuzz, an adaptive fuzzing engine for EVM smart contracts (Nguyen et al., 2020). sFuzz is composed of three components: *runner* that manages test case execution, *liboracles* that supports eight oracles inspired by the previous researches (Jiang et al., 2018; Luu et al., 2016) and *libfuzzer* which implements the test suite generation algorithm. The latter is based on a feedback-guided fuzzing technique which transforms the test generation problem into an optimization problem and uses feedbacks as an objective function in solving the optimization problem. This proposal is based on adaptive strategy since it is possible to change the objective function adaptively based on the feedback to evolve the test suite with the aim of improving its branch coverage. Due to its effectiveness and its reliability, sFuzz has already gained interest from multiple companies and research organizations. However, fuzz-based approaches may suffer from false positive detection as a reported vulnerability may be a false positive[8].

From the model-based testing perspective, authors in (Liu et al., 2020) introduce an MBT tool, namely ModCon, that uses an explicit abstract model of the target SC in order to derive tests automatically. ModCon shows its effectiveness specifically for enterprise SC applications written in Solidity from permissioned/consortium blockchains. It allows SC devel-

[7]Application Binary Interface

[8]Some test cases fail but there is no bug and the program is working correctly.

opers to input their test model for the SC under test. Certainly, the efficiency of MBT depends both on the input test model and fault model (or test hypotheses). A promising research challenge could be to investigate SCs and their faults to determine suitable fault models or test assumptions. In the same direction, authors in (Sánchez-Gómez. et al., 2019) propose a model driven approach that generates smart contract code from UML diagrams (i.e., Use Cases and Activity diagrams). They also point out the necessity of applying testing technique in the early stage of Software Development Life Cycle (SDLC), especially in the context of blockchain oriented software. It outlines the necessity of applying model-based testing process to generate test scenarios from requirements (UML use cases) and test cases from UML activity diagrams. However, this approach is still immature since no transformation rules for the smart contract generation code are given and no real tool implementation for the discussed ideas were presented. Similarly, the work in (Kakadiya, 2017) proposes a complete software testing life cycle to test BoS projects. The proposal is composed of four phases including system overview, test design, test planning and test execution.

4.3 Grey-box Testing Approaches

Another research line is identified. It consists in applying testing techniques with a partial knowledge of internal BoS structure. In this direction, the work of Wang et al. (Wang et al., 2019a) combines fuzz and mutation testing in order to detect several smart contract vulnerabilities triggered during transactions. The proposed tool called ContraMaster is a grey-box fuzzing approach that dynamically executes transactions and observes their actual effects on the contract states in order to detect vulnerabilities.

ContraMaster includes a novel feedback mechanism to guide efficiently the fuzzing process, by taking into consideration the data-flow information, control-flow information and states of smart contracts. It also monitors the executions of target smart contracts, and validates the obtained results against a general purpose semantic test oracle to detect vulnerabilities. Compared to traditional fuzzing approaches like ContractFuzz (Jiang et al., 2018), this proposal has not shown any false positives, and it easily generalizes to unknown types of vulnerabilities.

HARVEY (Wüstholz and Christakis, 2019) is another grey-box Fuzzer for smart contracts, which is the first one used in industry by one of the largest blockchain-security consulting companies. It is based on a novel technique that systematically forecasts new inputs for the smart contract under test in order to increase the performance and effectiveness of the grey-box fuzzing approach. In contrast to existing works based on grey-box fuzzing, this proposal suggests concrete input values based on data from past executions, instead of performing arbitrary mutations.

Similar to ContraMaster tool, HARVEY generates, executes, and fuzzes sequences of transactions, which invokes the contract's functions. To alleviate the exploration of the search space of all possible sequences, authors use a demand-driven sequence fuzzing technique, avoiding the generation of transaction sequences that cannot further enhance the coverage.

Another promising study was proposed in (Liao et al., 2019). This work introduces SoliAudit tool that combines machine learning and Fuzz testing for smart contract vulnerability assessment. Indeed, it employs machine learning using Solidity machine code as learning features to verify 13 kinds of vulnerabilities. The latter are listed as Top 10 threats by an open security organization[9].

Moreover, the tool includes a grey-box fuzz testing mechanism, composed of a fuzzer contract and a simulated blockchain environment for on-line transaction verification. Similar to (Wang et al., 2019a), SoliAudit can detect vulnerabilities without predefined patterns. The source code is only required if the user wants to identify the vulnerability location in the source code. It only needs the ABI and the bytecode to deploy on the blockchain.

Yet another potential work to explore is the HF-ContractFuzzer tool (Ding et al., 2021). Compared to the majority of cited studies that focus on Ethereum smart contracts written in solidity language, this tool is specific to a Hyperledger Fabric smart contract written in Go language. Indeed, authors deal with the security vulnerability detection while combining Fuzzing technology and Go language testing tool, called go-fuzz.

To do so, HFContractFuzzer takes as input smart contracts and test cases. Then, it automatically generates test results. In order to improve the efficiency of smart contract fuzzing, this proposal gives two optimization methods of the go-fuzz tool : optimizing the generation of the initial Fuzzing corpus and optimizing the mutation process. However, it requires more effort to evaluate its vulnerability detection and to measure its performance.

[9]DASP Top 10: https://dasp.co/

5 CHALLENGES AND FUTURE DIRECTIONS

5.1 Challenges

1. Smart Contract Testing still a hard task for developers: Despite the growing interest generated by its use in a wide range of applications, the use and the validation of smart contract technology is still considered as tough mission for many developers, mainly because of its innovative design.
2. Formal methods for smart contract testing still overlooked: Formal testing is regarded as a crucial part of the software development process. Formal testing, on the other hand, has been largely disregarded in the smart contract development cycle. Although a shift of formal testing from standardised software to smart contract-based software is quite conceivable.
3. Performance tools still not enough evaluated: Despite the fact that several techniques for evaluating Blockchain performance have been developed, only a few of them have been thoroughly examined due to the lack of standardised interfaces while running workloads.
4. Choice of performance metrics: Many developers utilize various types of coverage to demonstrate the effectiveness of testing. Although this is an intuitive indicator, there is no inescapable link between increased coverage and improved vulnerability detection efficiency.

5.2 Future Directions

1. Making Smart Contract Testing easier: The most important thing regarding this point is to define some specific techniques which may enable developers to establish a testing procedure dedicated for smart contracts.
2. Formal methods for smart contract testing: At this level too, there is a real need to make formal testing techniques more accessible for smart contracts developers by providing more sophisticated tools which are both efficient and easy to use at the same time.
3. Performance tools: these tools must evolve in order to meet the need of developers and to meet high-quality levels. In particular, they need to be equipped with standardised interfaces which may be used while running workloads.
4. Choice of performance metrics: At this level, new criteria and performance metrics have to be defined in order to improve the efficiency of vulnerability detection.

6 CONCLUSION

The main motivation for this study was to investigate the state of art in testing Blockchain oriented Software. This review was conducted to determine what issues have been studied and by what means, also to provide a guide for researchers in this emergent research area. Up to our best knowledge, the existing surveys covered essentially static testing and analysis. Dynamic testing approaches were rarely addressed. Therefore, this paper is a significant contribution in the literature which includes over **20** research studies published from 2017 to 2021. A classification of these approaches was reported while providing their strengths and weaknesses. In summary, the obtained results of this study indicated that most research papers are from conference proceedings, suggesting that the area of research is still young. The proposed research direction may lead to advances in three relatively open fields of software engineering research: (1) Model based testing of BoS, (2) Performance testing of blockchain networks and (3) Testing consensus algorithms. We believe that to improve the current state of art in these three lines, further work needs to be done and more exploration is required.

REFERENCES

Akca, S., Rajan, A., and Peng, C. (2019). Solanalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 482–489.

Ali, M. S., Vecchio, M., Pincheira, M., Dolui, K., Antonelli, F., and Rehmani, M. H. (2018). Applications of blockchains in the internet of things: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 21(2):1676–1717.

Andesta, E., Faghih, F., and Fooladgar, M. (2020). Testing smart contracts gets smarter. In *2020 10th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 405–412.

Antonino, P. and Roscoe, A. (2020). Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity. *arXiv preprint arXiv:2002.02710*.

Chan, W. and Jiang, B. (2018). Fuse: An architecture for smart contract fuzz testing service. In *The 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 707–708.

Ding, M., Li, P., Li, S., and Zhang, H. (2021). Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection. *CoRR*, abs/2106.11210.

Driessen, S., Nucci, D. D., Monsieur, G., and van den Heuvel, W.-J. (2021). Agsolt: a tool for automated test-case generation for solidity smart contracts.

Fekih, R. B. and Lahami, M. (2020). Application of blockchain technology in healthcare: A comprehensive study. In *The Impact of Digital Technologies on Public Health in Developed and Developing Countries - 18th International Conference, ICOST 2020, Hammamet, Tunisia, June 24-26, 2020, Proceedings*, pages 268–276.

Finley, K. (2016). A $50 million hack just showed that the dao was all too human.

Freedman, R. (1991). Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6):553 –564.

Gao, J., Liu, H., Li, Y., Liu, C., Yang, Z., Li, Q., Guan, Z., and Chen, Z. (2019). Towards automated testing of blockchain-based decentralized applications. In *IEEE/ACM 27th Int. Conf. on Program Comprehension (ICPC)*, pages 294–299.

Hartel, P. and Schumi, R. (2020). Mutation testing of smart contracts at scale. In Ahrendt, W. and Wehrheim, H., editors, *Tests and Proofs*, pages 23–42, Cham.

Ivanova, Y. and Khritankov, A. (2020). Regularmutator: A mutation testing tool for solidity smart contracts. *Procedia Computer Science*, 178:75–83.

Jabbar, R., Dhib, E., ben Said, A., Krichen, M., Fetais, N., Zaidan, E., and Barkaoui, K. (2022). Blockchain technology for intelligent transportation systems: A systematic literature review. *IEEE Access*.

Jiang, B., Liu, Y., and Chan, W. K. (2018). Contractfuzzer: Fuzzing smart contracts for vulnerability detection. ASE 2018, page 259–269.

Kakadiya, A. (2017). Block-chain oriented software testing approach. *International Research Journal of Engineering and Technology (IRJET)*.

Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. (2018). Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 2123–2138.

Koul, R. (2018). Blockchain oriented software testing - challenges and approaches. In *3rd International Conference for Convergence in Technology (I2CT)*, pages 1–6.

Krichen, M. (2007). *Model-based testing for real-time systems*. PhD thesis, PhD thesis, PhD thesis, Universit Joseph Fourier (December 2007).

Krichen, M. (2018). *Contributions to model-based testing of dynamic and distributed real-time systems*. PhD thesis, École Nationale d'Ingénieurs de Sfax (Tunisie).

Li, Z., Wu, H., Xu, J., Wang, X., Zhang, L., and Chen, Z. (2019). Musc: A tool for mutation testing of ethereum smart contract. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1198–1201.

Liao, J.-W., Tsai, T.-T., He, C.-K., and Tien, C.-W. (2019). Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 458–465.

Liu, J. and Liu, Z. (2019). A survey on security verification of blockchain smart contracts. *IEEE Access*, 7:77894–77904.

Liu, Y., Li, Y., Lin, S.-W., and Yan, Q. (2020). Modcon: A model-based testing platform for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1601–1605.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 254–269.

Nakamoto, S. et al. (2008). Bitcoin: A peer-to-peer electronic cash system.

Nelaturu, K., Mavridou, A., Veneris, A., and Laszka, A. (2020). Verified development and deployment of multiple interacting smart contracts with verisolid. In *Proc. of the 2nd IEEE International Conf. on Blockchain and Cryptocurrency (ICBC)*.

Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., and Minh, Q. T. (2020). Sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, page 778–788.

Praitheeshan, P., Pan, L., Yu, J., Liu, J. K., and Doss, R. (2019). Security analysis methods on ethereum smart contract vulnerabilities: A survey. *CoRR*, abs/1908.08605.

Sánchez-Gómez., N., Morales-Trujillo., L., and Torres-Valderrama., J. (2019). Towards an approach for applying early testing to smart contracts. In *Proceedings of the 15th International Conference on Web Information Systems and Technologies - APMDWE,*, pages 445–453.

Sánchez-Gómez, N., Torres-Valderrama, J., García-García, J. A., Gutiérrez, J. J., and Escalona, M. J. (2020). Model-based software design and testing in blockchain smart contracts: A systematic literature review. *IEEE Access*, 8:164556–164569.

Tolmach, P., Li, Y., Lin, S., Liu, Y., and Li, Z. (2020). A survey of smart contract formal specification and verification. *CoRR*, abs/2008.02712.

Wang, H., Li, Y., Lin, S.-W., Artho, C., Ma, L., and Liu, Y. (2019a). Oracle-supported dynamic exploit generation for smart contracts.

Wang, X., Wu, H., Sun, W., and Zhao, Y. (2019b). Towards generating cost-effective test-suite for ethereum smart contract. In *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 549–553.

Wu, Z., Zhang, J., Gao, J., Li, Y., Li, Q., Guan, Z., and Chen, Z. (2020). Kaya: A testing framework for blockchain-based decentralized applications. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 826–829.

Wüstholz, V. and Christakis, M. (2019). Harvey: A greybox fuzzer for smart contracts. *CoRR*, abs/1905.06944.

Yuan, C. and Zhu, J. (2020). A new performance testing scheme for blockchain system. In Zhang, J., Dresner, M., Zhang, R., Hua, G., and Shang, X., editors, *LISS2019*, pages 757–773.