Runtime Verification and Vulnerability Testing of Smart Contracts

Misha Abraham(✉) and K. P. Jevitha

Amrita School of Engineering, Amrita Vishwa Vidyapeetham, Coimbatore, India
abrahammisha@gmail.com, kp_jevitha@cb.amrita.edu

Abstract. Smart contracts are programs that help in automating agreement between multiple parties involving no external trusted authority. Since smart contracts deal with millions of dollars worth of virtual coins, it is important to ensure that they execute correctly and are free from vulnerabilities. This work focuses on smart contracts in Ethereum blockchain, the most utilized platform for smart contracts so far. Our emphasis is mainly on two core areas. One involves the runtime verification of ERC20 tokens using K framework and the other involves the comparison of tools available for detecting the vulnerabilities in smart contract. The six core functions of ERC20, namely allowance(), approve(), total-supply(), balanceof(), transferfrom() and transfer() were considered for runtime verification. ERC20 contracts were tested with ERC20 standard and the results showed that only 30% in allowance() function, 50% in transferfrom() function, and 90% in transfer() function, were compliant to the standard. The other focus area involves the comparison of existing tool that could identify vulnerabilities in smart contract. Five tools were taken for the comparison, namely Oyente, Securify, Remix, Smartcheck and Mythril and were tested against 15 different vulnerabilities. Out of the 5 tools taken, Smartcheck was found to detect the highest number of vulnerabilities.

Keywords: Blockchain · Smart contract · Vulnerabilities · Runtime verification · ERC20 token · K framework

1 Introduction

A blockchain is essentially a distributed database of records of all transactions or digital events that have been shared among participating parties. It is also known as a distributed ledger which uses cryptographic methods to store information and provides better security [1]. Applications like financial transactions, Internet of Things etc., uses blockchain technology. It was initially implemented by Satoshi Nakamoto on bitcoin platform [2]. Ethereum is one among the best blockchain platform which enables decentralized applications. As stated by Vitalik Buterin, Ethereum is a blockchain platform which can understand programming languages. Unlike bitcoin, Ethereum provides one common platform for all the use cases.

Smart contracts are the decentralized applications that are designed to run on Ethereum using Ethereum Virtual Machine (EVM). Once a smart contract is deployed in the blockchain, it cannot be modified or updated. This can be both advantage as well as disadvantage. As an advantage, it provides better security by preventing the users from editing/modifying the smart contract that are deployed into the Ethereum blockchain. The disadvantage is that if there are any bugs/vulnerabilities in smart contract, they can be addressed only by deleting the entire smart contract from the blockchain. As Dijkstra stated "testing shows the presence not the absence of bugs" there is a requirement to formally verify the smart contract code to provide better compliance of the contract to its standard.

In this work we focus on using runtime verification to verify smart contract functionality against ERC20 standard to ensure that the contract behaves as they are designed for. This is done by taking ERC20 standard rules written in K language [4] and the EVM semantics (KEVM). It works by symbolically executing the bytecode of the contract in the KEVM and identifying its compliance with ERC20 standard using the K framework [3, 5, 6]. We have also examined the tools, to detect security vulnerabilities in smart contracts in Ethereum, which can analyze solidity code to identify vulnerabilities in smart contract. In our work, we compared 5 different tools, namely Oyente, Securify, Remix, Smartcheck and Mythril against 15 vulnerabilities.

The paper is organized as follows. In Sect. 2, background study of Smart contract and K framework is explained in detail. Section 3 describes the related work done in the field of formal verification and vulnerability assessment of smart contracts. Section 4 explains the proposed work on runtime verification and vulnerability testing of smart contracts, the results of which are explained in Sect. 5. Finally Sect. 6 concludes about the work done and briefly explain about the scope of this work in future.

2 Background Study

2.1 Smart Contract

Smart contracts are used to eliminate the need of trusted third parties and are usually written using programming languages like Solidity, Vyper, Java etc., Smart contracts work in the same way as that of classes in object oriented programming and they are made up of functions and fields [8]. The smart contract code is arranged in a low-level stack-based bytecode. Since the bytecode is openly available, it may be said that the smart contract code can be examined by each and every node in the network. So if the contract is vulnerable everyone in the network can see and understand the vulnerability and can exploit it easily.

Apart from creating decentralized applications, a smart contract can also be used to create tokens for the purpose of developing their own economy on top of Ethereum [7]. In order to standardize token creation, Ethereum community has come up with token standards like ERC20, ERC721 etc., ERC stands for Ethereum Request for Comment, and 20 is the number assigned to the request. It includes six core functions, namely allowance(), approve(), totalsupply(), balanceof(), transferfrom() and transfer() [9]. The use of ERC20 standard makes the risk of tokenization less since every token adhere to the same standards. It also makes the token interaction less complex and also brings in uniformity to the network. If the tokens doesn't match with the ERC20 standard it will not be possible to use the ERC20 compliant wallets like metamask. So it is necessary to check for the compliance of the core functions of ERC20 tokens with the ERC20 standard functions.

2.2 K Framework

K is a framework in which programming languages can be defined and programs can be symbolically executed according to semantics of the language written in K. In order to write semantics of a language in K, we need to define three sections, namely configurations, computations and rules [10]. The configuration part contain the initial values of parameters to be used while writing the semantics. In computations part, we will define the computations that are to be performed to the parameters declared in configuration part. Finally, the rule contains the conditions that should be satisfied while executing the program. The execution of language in K framework will take two inputs. One is the semantics of the language written in K and the other is the program that is written in that language. Executing the program using K will run the program according to the semantics defined in K.

3 Related Work

This section describes the formal verification of smart contracts and also the security vulnerabilities present in smart contracts.

3.1 Formal Verification of Smart Contracts

This section gives an overview of related work on formal verification of smart contracts. In Karthikeyan et al. [19], they focused mainly on formal verification of smart contracts at the source level and also at the EVM level. This is done by converting the solidity code and bytecode into solidity* and bytecode* which can be used to verify the smart contract properties. They have also explained in detail about the conversion of solidity code and bytecode to solidity* and bytecode* respectively. Another formal verification method is given in [20] in which they characterized trace vulnerabilities efficiently by analyzing a smart contract multiple times. The main focus was on three main properties of trace vulnerabilities such as finding contracts that sends ether to arbitrary users, locks funds indefinitely and the contracts that are vulnerable to suicidal attack. They created a tool to detect the above mentioned trace vulnerabilities and named it as MAIAN. It is also known to be the first tool that utilizes symbolic analysis and indicates the trace properties for vulnerability detection in smart contracts. In [16], the authors focuses on creating a semantics for ethereum virtual machine and is known to be the first completely executable formal semantics of the EVM in which the smart contracts can be executed in the form of bytecodes. It deals with identifying the vulnerabilities such as integer overflow and unhandled exceptions.

3.2 Security Vulnerabilities

This section explains about the related work on analysis of security vulnerabilities in smart contracts from various research papers and articles. In Atzei et al. [11], the levels chosen to represent the vulnerabilities are in the following manner: solidity, EVM and blockchain. It is classified into different levels to group all the vulnerabilities in each and every level. So, if a new vulnerability is found, it can be matched to any of these levels. Another classification is given by Alharby and Moorsel [8, 14, 15] in which they found four major issues in smart contracts. They are security, performance, coding and privacy issues. In security issues, they identified the bugs and vulnerabilities where as in privacy they identified the issues like data exposure to unauthorized people. In coding issues they found out flaws in smart contract code and methods to improve them. Finally performance deals with the capacity of blockchain code. Some research papers and articles mainly focused on tools that could detect security vulnerabilities. Ethereum community found Oyente as the first and most important investigation tool in terms of security. It was developed by Luu et al. [16] and uses symbolic execution to identify the security vulnerabilities.

4 Methodology

4.1 Runtime Verification of Smart Contracts

In order to perform runtime verification ERC20 contracts were taken and checked for its compliance with ERC20 standard. For this purpose, K framework was used. Three main inputs to K are contract specification, ERC20 contract bytecode and a trusted input to execute the bytecode. Contract specification states what the contract "should do" in K language. The contract bytecode is the low level code of the smart contract

Table 1. Contracts and its description.

ERC20-token contracts	Description
Storj (STJ)	Decentralized cloud storage network
Odyssey (OCN)	Decentralized sharing economy and peer-to-peer ecosystem
Dentacoin (DCN)	Blockchain designed for improving the quality of dental care worldwide
Wax	Decentralized platform designed to serve video gamers to trade virtual assets
Populous (PPT)	Peer-to-peer invoice discounting platform
Rchain (RHOC)	Decentralized applications platform powered by Rho virtual machine
ICON (ICX)	Decentralized transactions network
Gifto (GTO)	Decentralized universal gifting protocol
OpenZeppelin	OpenZeppelin is a library for secure smart contract development
Dragonchain (DRGN)	Simplifies the integration of real business applications on blockchain

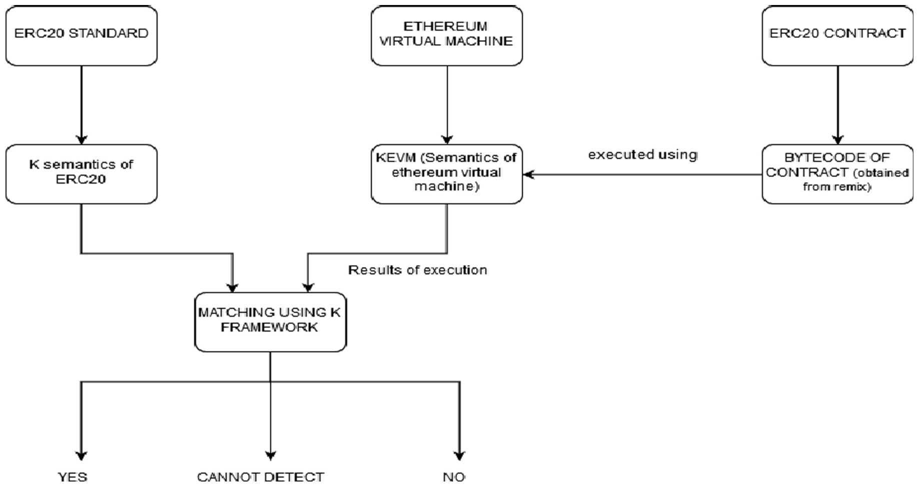

Fig. 1. Runtime verification of smart contracts.

which ethereum virtual machine can understand. And the trusted input is the semantics of the ethereum virtual machine in which the bytecode can be executed. Contracts that were taken for compliance check with ERC20 standard are shown in Table 1. 60 functions from these 10 ERC20 contracts were matched with six core functions of ERC20 standard, namely allowance(), approve(), totalsupply(), balanceof(), transferfrom() and transfer(). The results would show whether an ERC20 contract follow the ERC20 standard. The above process is shown step-by-step in Fig. 1.

The bytecode of each smart contract is obtained from Remix, an online interface for solidity [22]. The contract when written in the interface have to be compiled. Successful compilation of a contract will produce details like bytecode, assembly code, application binary interface etc., The bytecode in Ethereum are executed using ethereum virtual machine and involves gas. So a semantics of ethereum virtual machine is used for executing the bytecode. In our work, we have used K semantics of ethereum virtual machine for symbolically executing the smart contract. The execution result and the ERC20 contract standard in K are given as input to the K framework which then check for its compliance.

4.2 Vulnerability Testing

According to the research paper [16] out of 19,366 existing Ethereum contracts, 8,833 are vulnerable in many ways. We selected 15 different, namely timestamp dependence, use of untrusted input, transaction-ordering dependence, reentrancy, insecure coding patterns, unexpected ether flows, mishandled exceptions, tx.origin usage, blockhash usage, gas costly patterns, DoS by external contract, unchecked external call, locked money, unprotected functions and integer overflow/underflow whose occurrence is very frequent and conducted a study on the tools that could detect these vulnerabilities. Smart contract with above vulnerabilities were taken from different websites given in

[16, 23–25]. The results of this work would allow us to have an idea about the tools to be used to detect the above mentioned vulnerabilities. It would also allow us to identify the tool which could detect highest number of vulnerabilities. The above process falls into 3 steps as depicted in Fig. 2.

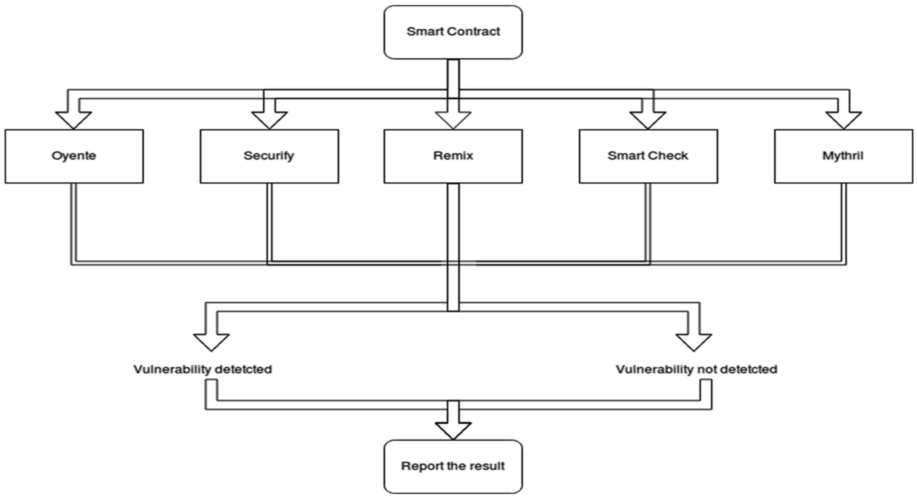

Fig. 2. Vulnerability testing of smart contracts.

5 Analysis and Results

5.1 Analysis of Contracts Based on ERC20 Standard Using K Framework

Among the 10 contracts chosen 6 contracts were not compliant with the ERC20 standard and 4 contracts were compliant with the ERC20 standard, the results of which are shown in Table 2. The analysis also showed that 3 contracts were not compliant with allowance() function, 5 contracts were not compliant with trans ferfrom() function

Table 2. Results of compliance check of contracts with the standard.

Smart contracts	Compliant/Non compliant
Storj	Non compliant
Odyssey	Compliant
Dentacoin	Non compliant
Wax	Non compliant
Populous	Compliant
Rchain	Compliant
ICON	Non compliant
Gifto	Non compliant
OpenZeppelin	Compliant
Dragonchain	Non compliant

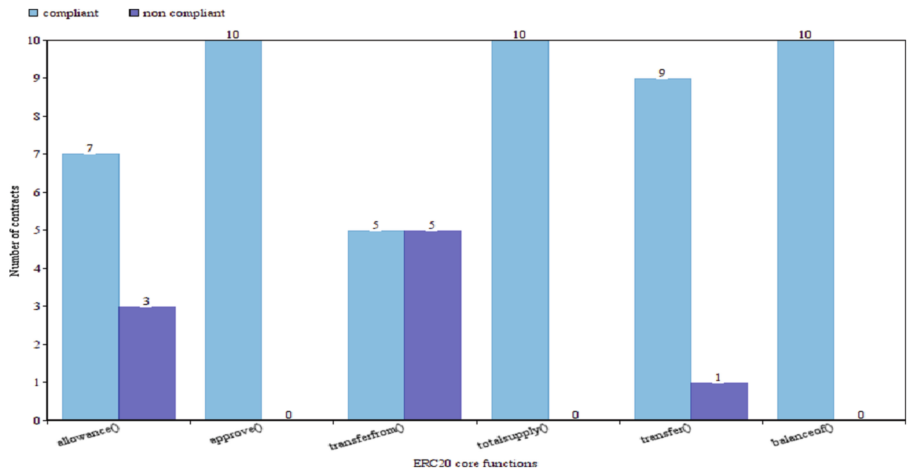

Fig. 3. Analysis of smart contracts based on ERC20 standard.

and 1 contract were not compliant with transfer() function. The Fig. 3 shows the analysis result of 60 functions with the ERC20 standard.

5.2 Analysis of Smart Contract Vulnerability Testing Tools

The analysis on security detection methods helped in understanding different tools and was able to classify them based on solidity analysis or bytecode analysis performed. As shown in Table 3, it was found that Securify and Mythril was the tools that performed both solidity and bytecode analysis. The tools like oyente, Remix and smartcheck performed only solidity analysis.

Table 3. Comparison of tools based on detection method and number of vulnerabilities detected.

Security tool	Method	Bytecode analysis	Solidity analysis	No. of bugs detected
Oyente	Symbolic execution		√	5
Securify	Formal verification	√	√	4
Remix	Formal verification		√	5
Smart check	Symbolic execution		√	7
Mythril	Formal verification	√	√	4

The analysis was also done based on the vulnerability detection capabilities of different tools. Comparison results are briefly explained in Table 4. Oyente was able to detect transaction ordering dependence, timestamp dependence, mishandled exceptions, reentrancy and integer overflow/underflow. Securify was able to detect transaction-ordering dependence, reentrancy, unexpected ether flows and use of untrusted input. Remix an online tool was able to detect timestamp dependence, reentrancy, tx.origin usage, blockhash usage and gas costly patterns. Smartcheck was able to identify timestamp dependence, reentrancy, tx.origin usage, gas costly patterns, DoS by external contract, locked money and unchecked external call. Mythril detected reentrancy, tx.origin usage, unprotected functions and integer overflow/underflow.

Table 4. Results of comparison of tools based on vulnerabilities detected.

Vulnerabilities	Oyente	Securify	Remix	Smart check	Mythril
Transaction-ordering dependence	√	√			
Timestamp dependence	√		√	√	
Mishandled exceptions	√				
Reentrancy	√	√	√	√	√
Insecure coding patterns					
Unexpected ether flows		√			
Use of untrusted input		√			
tx.origin usage			√	√	√
Blockhash usage			√		
Gas costly patterns			√	√	
DoS by external contract				√	
Locked money				√	
Unchecked external call				√	
Unprotected functions					√
Integer overflow/underflow	√				√

6 Conclusion and Future Work

A runtime verification and security analysis of the existing Smart contract was performed and it was discovered that many of the contracts were not compliant with standards like the ERC20 standard. Also the solidity code which was used to write smart contracts in ethereum was vulnerable in many aspects. Some of vulnerabilities are integer overflow, timestamp dependency, invalid random entropy sources, exception handling, unnecessary usage of delegate call, denial of service and callstack depth problem.

The Runtime verification of smart contracts was performed using K framework. 60 functions from 10 contracts were taken and it was found that, some of the functions does not follow ERC20 standard. Also, a comparison of the tools that are developed for identification of vulnerabilities was performed and analyzed different vulnerabilities detected by different tools. Our research showed Smart check as one of the best vulnerability testing tool which can detect seven vulnerabilities. The above two research helped us to perform runtime functionality verification of smart contracts and also to check for vulnerabilities in smart contract.

There are several lines of research arising from this work which should be pursued.

Firstly, on developing a platform in which all vulnerabilities in smart contract can be detected instead of using different tools. A second line of research, which follows from Sect. 4.1, is to write k rules for ERC721 which can be used to check the functionality of ERC721 contracts. Finally, performance checking of semantics of Ethereum Virtual Machine.

References

1. Zheng, Z., et al.: An overview of blockchain technology: architecture, consensus, and future trends. In: 2017 IEEE International Congress on Big Data (BigData Congress). IEEE (2017)
2. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
3. K Framework - An Overview. https://runtimeverification.com/blog/k-framework-an-overview/. Accessed 22 Sept 2018
4. Rosu, G.: ERC20-K: Formal Executable Specification of ERC20. https://runtimeverification.com/blog/erc20-k-formal-executable-specification-of-erc20/. Accessed 21 Sept 2018
5. Formal verification of ERC-20 contracts. https://runtimeverification.com/blog/erc-20-verification/. Accessed 21 Sept 2018
6. How Formal Verification of Smart Contracts Works. https://runtimeverification.com/blog/how-formal-verification-of-smart-contracts-works/. Accessed 21 Sept 2018
7. Sajana, P., Sindhu, M., Sethumadhavan, M.: On blockchain applications: hyperledger fabric and ethereum. Int. J. Pure Appl. Math. **118**, 2965–2970 (2018)
8. Alharby, M., van Moorsel, A.: Blockchain-based smart contracts: a systematic mapping study. arXiv preprint arXiv:1710.06372 (2017)
9. https://theethereum.wiki/w/index.php/ERC20TokenStandard. Accessed 22 Aug 2018
10. https://runtimeverification.com/blog/k-framework-an-overview/. Accessed 1 Aug 2018
11. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
12. Egbertsen, W., et al.: Replacing paper contracts with Ethereum smart contracts (2016)
13. Kosba, A., et al.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE (2016)
14. del Castillo, M.: The dao attacked: code issue leads to $60 million ether theft. Saatavissa (viitattu 13.2.2017). http://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft. Accessed 15 Sept 2018
15. Reentrancy Woes in Smart Contracts. http://hackingdistributed.com/2016/07/13/reentrancy-woes/. Accessed 22 Aug 2018
16. Luu, L., et al.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM (2016)
17. Kalra, S., et al.: ZEUS: analyzing safety of smart contracts. In: NDSS (2018)

18. Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: lessons and insights from a cryptocurrency lab. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 79–94. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_6
19. Hildenbrandt, E., et al.: KEVM: a complete semantics of the ethereum virtual machine (2017)
20. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. ACM (2016)
21. Nikolic, I., et al.: Finding the greedy, prodigal, and suicidal contracts at scale. arXiv preprint arXiv:1802.06038 (2018)
22. https://remix.ethereum.org. Accessed 13 Sept 2018
23. https://github.com/smartdec/smartcheck/tree/master/src/test/resources/rules. Accessed 07 Aug 2018
24. https://github.com/eth-sri/securify/tree/master/src/test/resources/solidity. Accessed 07 Aug 2018
25. https://github.com/trailofbits/not-so-smart-contracts. Accessed 07 Aug 2018