Review of Tools for Analyzing Security Vulnerabilities in Ethereum based Smart Contracts

Anishka Moona[a], Rejo Mathew[b]

[a]*Mukesh Patel School of Technology Management and Engineering, NMIMS (Deemed-to-be) University, Mumbai- 400056, India anishka.moona35@nmims.edu.in*

[b]*Mukesh Patel School of Technology Management and Engineering, NMIMS (Deemed-to-be) University, Mumbai- 400056, India rejo.mathew@nmims.edu*

ABSTRACT

One of the most promising features of blockchain technology are software programs; smart contracts. A smart contract is a digital agreement that allows a significant amount of cryptocurrency transactions and processes without the need of third parties. They are the fundamental component for any decentralized application and are irreplaceable. They are legally self-enforced lines of a computer program. This paper is motivated by financial losses that occurred due to known attacks such as DAO attack ($150M) , Parity Multi-Sig Wallet attack (30M USD) and integer underflow/overflow attack during 2016-2018. This paper provides a detailed study of the known security vulnerabilities and security analysis tools that are used to detect them. This review explores possible attack techniques along with presenting a systematic study of the tools such as Oyente, Smart Check, Securify, GASPER, ZEUS, MAIAN and F* Framework used to detect these vulnerabilities.

Keywords: Ethereum, Smart Contracts, Analysis, Attack Techniques, Tools, Detection, Security.

1. Introduction

Blockchain technology implements several mechanisms of which security is guaranteed because of the use of cryptographic methods. In comparison to hostile centralized applications running on one computer, a peer-to-peer network of computers is required for decentralized applications. This helps in evicting the involvement of third parties. [1] This decentralized execution uses distributed ledger technology mechanism. The execution is carried out by deploying the programming scripts called Smart Contracts to process tasks autonomously that have been widely supported on Ethereum platform. For developing these smart contracts on the Ethereum platform, a high-level object-oriented programming language Solidity is used; it is similar to JavaScript.

Smart Contracts comprises the following: a) involving parties and the subject b) an account balance c) a private storage (containing addresses) and d) an executable code, which is immutable and stored on EVM. [2] Storage and balance of the contract together define the state of the contract that is stored on the blockchain. Initially, they are explicitly set to zero. The state is modified every time a contract is activated. Ethereum is the most suitable platform for smart contracts using ether as its cryptocurrency. This is because it supports manipulation of other data-rules i.e. its Turing-completeness feature and allows to carry each transaction only once. Smart contracts are considered to be an essential component of blockchain applications, that operate on the information from peripheral events and help the blockchain in managing the state of the network. Smart Contracts are self-executing programs running on Ethereum Virtual Machine. As the term, 'virtual' suggests the instance of a real computer running on another operating system. The EVM provides limited computation capabilities having a stack-oriented architecture. The instruction set includes various operations such as - Arithmetic, Stack, Process flow, System, Logic, etc. EVM ensures that the state of Ethereum is valid every time a contract is executed, or a transaction takes place. The terms of agreement between buyer and seller is directly written into the lines of code stored as a set of rules. The EVM runtime environment executes the opcode (set of instructions) of smart contracts. Here, the buyer and seller do not need to rely on lawyers or banks as trust is also ensured. It enforces to ensure that the terms of an agreement are met between untrusted parties and then releases the digital assets. Smart contracts technology can be utilized to different applications (e.g., IoT, e-commerce, healthcare, financial transactions, supply chain, etc.) [3]

Despite the features and security enhancements, smart contracts still face challenges to cope up with vulnerabilities and attacks; maybe due to lack of knowledge to developers and less testing facilities. They are just like other programs which might contain flaws. Thus, developers are encouraged to write secure smart contracts before deployment on the blockchain network. This is due to the immutable property of blockchain, that no one can modify a contract once it is deployed. However, anyone on the network can access them. Unless a blockchain network is exploited, the contract can't be removed. Thus, it becomes necessary to test them before deployment. The vulnerable smart contracts can be classified into 3 categories: a) Prodigal contracts (leaks funds to arbitrary users), b) Suicidal contracts (contracts that can be killed by any user) and c) Greedy contracts (contracts that lock funds indefinitely). [4]

Two real-life examples of such unique attacks are given below:

1. Decentralized Autonomous Organization, DAO was established in 2016 by the members of Ethereum society. It had a digital set of contracts that allowed token holders to become contract holders with ether using DAO funds. The DAO was stipulated with a large crowdfund, around $150M (over 3,600,000 Ether) due to the exchange method in the code. However, the DAO contract had loopholes which allowed attackers to remove funds by exploiting a reentrancy vulnerability. This allowed them to call the funds from the smart contract several times before the balance was updated and embezzle millions of dollars of ether. [5] This might have occurred due to the fact that developers didn't think of the potential for a recursive call. This depicts that a simple smart contract vulnerability can be quite devastating.

2. In July 2017, Parity Wallet Hack was located on the Parity Multisig Wallet. [6] This attack was executed in two transactions. The goal was to own the whole Multisig so that all currency could be removed. Primary transaction was made by calling *initWallet()* function. The bug was all unmatched functions calls were forwarded to the library using *delegatecall()* that contained all public functions. This allowed to exploit the initialization of the *initWallet* function which contained the owner's address. After exploiting, the attacker simply changed m_owners state variable to his address thus emptying the Wallet. This costed around 30M USD (150,000 ether). If *delegatecall()* was not used as the forwarding mechanism this attack could have been prevented.

Such attacks make the study for prevention of higher importance. In this paper, Section 2 comprises the known security vulnerabilities and attack techniques of smart contracts. Section 3 includes security analysis tools and methods that are used to detect these attacks in smart contracts. Lastly, in Section 4 these tools have been compared and concluding remarks are presented in Section 5.

Nomenclature

EVM Ethereum Virtual Machine
CFG Control Flow Graph
SMT Satisfiability modulo theories
AST Abstract Syntax Tree
CLI Command Line Interface
XML eXtensible Markup Language like HTML
IR Intermediate representation
API Application programming interface

2. Attack Techniques

Table 1- A taxonomy of various types of issues in smart contracts.

Issues	
Security Vulnerabilities	Re-entrancy Attack; Transaction Ordering Dependency; Timestamp Dependency; Mishandled Exceptions/Unchecked send-bug; DoS by external contract; Unchecked external call; Gas costly patterns/Gasless patterns; Using tx.origin.
Codifying/Developmental Issues	Complexity of programming languages; Compiler version not fixed; Token API violation; Difficulty in writing correct smart contracts.
Privacy Issues	Lack of transactional privacy; [28] Lack of data feeds privacy. [27]
Performance Issues [26]	Sequential Execution of smart contracts

This paper focuses and provides a detailed study only for analyzing security vulnerabilities on the contracts. However, the literature identifies performance, privacy and codifying issues. Privacy issues include lack of transactional privacy that is, during an interaction the users' balances are publicly visible which limits the adoption of contracts for execution. Similarly, during operating on data feeds the information is publicly visible to everyone on the blockchain. These privacy concerns can be ensured by using tools 'Hawk' [28] and 'Town Crier' [27] respectively. Privacy can also be ensured by encrypting the contracts before deployment. [29] To ensure efficient performance on the network, sequential execution (that is, one contract at a time) must be avoided and parallel execution must be encouraged provided that the contracts are independent. [26]

2.1 Re-entrancy Attack: [7]

Re-entrancy attacks can be classified into following two types - Single function and cross-function. Single function attack occurs during the time of execution. An external call is initiated with a help of a function. The problem arises when this call is made to another untrusted contract or to itself for transaction of ether. This helps the receiver to take advantage of the intermediate state the sender is in. The external call doesn't throw any exceptions if the conditions are met i.e., it returns false otherwise true. As the fallback function does not have any arguments or return values, any code inside the fallback function is executed and the balance of the original contract is emptied as it leads to a recursive call in a loop. In the type of cross-function, this reentrant transaction is guarded, thus the attacker makes use of some other function in the contract which is in the same state.

2.2 Transaction Ordering Dependency:

TOD is a type of race condition attack that arises when the operation of a program depends on the sequence or timing of the processes and threads. This bug occurs when the outcome of one transaction depends on the order of another for e.g. user1 and user2 calls the same smart contract for the transaction. Any one of the transactions occurring first changes the state of the contract before another interaction is processed. This depends on the block mining order. In worst case scenarios, this attack allows to change the price during the processing of the transaction which leaves buyers to pay larger than the expected amount.

2.3 Timestamp Dependency: [7]

This attack is exploited by corrupt miners. They are responsible for setting a timestamp for the block, generally to the current time of the miner's local system. Code instructions such as block.timestamp and block.number gives the output of the current time or a time delta. In the case of block.timestamp, attackers can attempt to use it as a triggering condition to execute time dependent actions such as sending money. It is necessary for the miner to manipulate the timestamp in the standard range and must be larger than the previous block. However, miners can't set a timestamp smaller than the previous one nor can they set the timestamp too far ahead in future. Thus, considering these points, programmers must not depend on the accuracy of the provided timestamp.

2.4 Mishandled Exceptions/Unchecked-send bug: [8]

When there is a transaction between two contracts, a smart contract calls another contract. There is a caller and a callee. Often, during this interaction an exception might be thrown if conditions are not satisfied; this causes the callee contract to terminate, reverse its state and return a false value. And if these exceptions are not handled properly by the caller (e.g. not checking the return value), they make room for a variety of vulnerabilities. In Ethereum, sending payment to a contract or calling a contract, that is carrying out a transaction costs "gas". An example is the King of Ether Throne contract. [9] In this, a wallet contract called the KotET contract using a method function. However, the payment sent to the wallet contract by KotET contract was not processed. And so, the payment in terms of ether stayed with KotET. The KotET was unaware that the payment had failed. It kept on carrying out the transaction with another contract and the process, which led to making the caller King. The particular line of Solidity code used to send payments was:

currentMonarch.etherAddress.send(compensation);

Here send is used instead of transfer. This shows that use of transfer must be encouraged and is safest. As it checks the return value automatically instead manually. Thus, not allowing the attacker to drain the balance.

2.5 Using tx.origin:

This is a form of phishing attack, called transaction origin attack that is capable of draining a contract of all the funds. Transaction Origin (tx.origin) is the first user in the chain, who initiated the chain of interactions between contracts. This is usually an externally owned, original sender. Similarly, msg.sender is the immediate caller or sender. The attacker can relay the transaction using the address of tx.origin. It starts "posing" as the first contract and passes the entire balance of the immediate sender to its own contract. Suppose, A to B to C, here C will see A as first and B as immediate. Thus, the usage of tx.origin for authentication must be avoided, as it is easily spoofed.

2.6 Unchecked External Call: [8]

External calls are considered to be risky in smart contracts. This is because a malicious code can be executed in that external contract. To avoid this while sending ether, errors must be handled, and return value must be checked. *pull* over *push* is more favorable.

2.7 DoS by External Contract: [10]

DoS is a situation created when during a transaction a conditional statement such as *if , for , while* depends on an external call. The attacker here is the callee which reverts the payment and stops the process. This is possible as the attacker makes use of Solidity's fallback function (it is a function that doesn't have any arguments). This vulnerability is similar to the previous vulnerability and to prevent this form of attack from happening, any *throw* exceptions from external calls needs to be handled properly, and also, looping behavior must be avoided.

2.8 Gas costly patterns/Gasless Send: [11]

Gas is defined as the fee or the price amount required to carry out a transaction. Depending on the transaction fees, the maximum gas limit on the network may alter during time. This technique becomes vulnerable when the solidity codes in Ethereum smart contracts have the potential to be executed with expensive patterns which cost more gas than required during operation of each instruction. Hence, to save users money smart contract coding practices must be improved.

Other vulnerabilities include – integer overflow/underflow attack [12] that occurs due to the reason that Solidity data types are only of 256 bits. This makes the variables in a loop vulnerable after they reach a maximum/minimum value of the standard stack size. Another attack that also occurs due to constrained Solidity language is Call-Stack Depth Limitation. In EVM, the limit is up to 1024 frames only. After every instruction is invoked, the value increases by one. Similarly, Token API violation occurs when

the token functions with incorrect parameters are implemented. That is, the function parameters and return types do not match with the standard application programming interface for implementing tokens. API's act as a software gateway allowing the backend of two applications to interact with one another.

3. Tools

The solutions or countermeasures to the above-mentioned attacks have been highlighted in this section. For analyzing smart contract codes, majority tools must convert the Solidity code to EVM bytecode. As bytecode helps in efficient storage of the opcodes. Three methods are used to analyze these codes – Static Analysis, Dynamic Analysis and Formal Verification.

Static Analysis:

3.1 Oyente: [13] (Applicable to Section 2.1, 2.2, 2.3, 2.4)

Oyente uses symbolic execution to find potential security threats. In symbolic execution the program variables are first represented using symbolic expressions. Then the paths and branches are characterized by operating on these symbols. These operations lead to algebraic terms and conditional statements giving rise to propositional formulas. A path is not traversed if its path condition is poor. Oyente has a modular design, consisting of four main components Control Flow Graph Builder, Explorer, Core Analysis and Validator (Figure 1)
The steps followed are:

a) A control flow graph is constructed for the bytecode using CFGBuilder module.
b) A control flow graph consists of nodes and edges. Here, the node represents an execution block i.e. a deed between the parties and edges represent execution jumps between the blocks.
c) After that, symbolic execution of the contract takes place in the explorer.
d) Suppose output X is received from the explorer. X is then fed as an input to the Core Analysis component.
e) When the vulnerabilities are catched, they are operated by the logic in the Core Analysis module.
f) In the end, the Validator module filters out the false positives and final results are visualized. Satisfiability modulo theories (SMT) solver, Z3 bit-vector solver is used to prove the correctness of outputs received.

TOD is detected when the Explorer returns a set of traces i.e. the sequence of instructions executed during run time and the corresponding Ether flow of each trace. A contract is flagged when any timestamp dependent symbolic variable is identified. Re-entrancy is identified by making use of path conditions and the execution of *ISZERO* instruction is checked to handle exceptions.

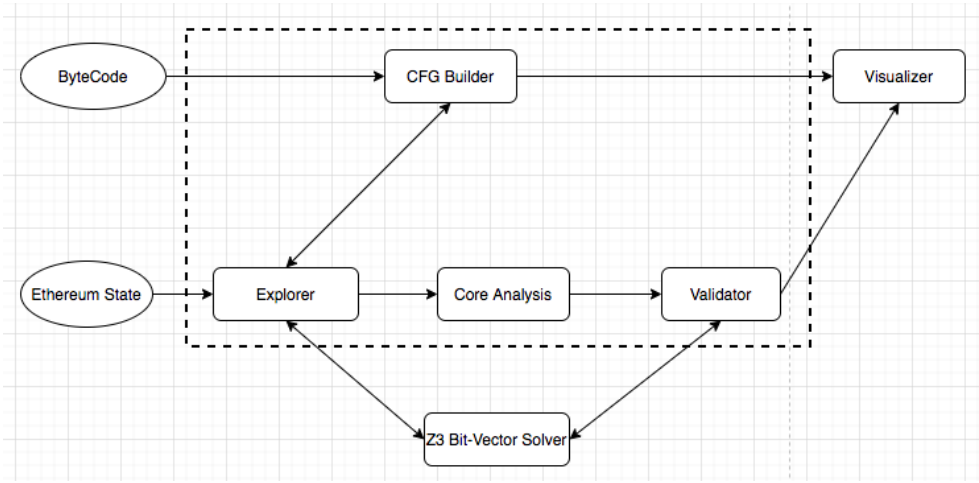

Fig. 1 – Architecture of OYENTE. Main components within the dotted area. Outside boxes are publicly available.

3.2 SmartCheck: [10] (Applicable to Section 2.1, 2.3, 2.5, 2.6, 2.7)

Smart Check is a web-based automated static analysis tool. It uses Another Tool for Language Recognition (ANTLR) and custom Solidity grammar to generate XML parse tree of the source code. It is then generated as an intermediate representation (IR). To

search the vulnerabilities and patterns, XQuery path expressions are used on the IR. XQuery helps in selecting nodes from XML documents. It automatically detects bad coding practices and highlights and flags the vulnerability (line of code). SmartCheck also describes the vulnerability along with a viable solution to avoid a particular security issue. In this way, it not only identifies the attack, but also clarifies and recommends. Each vulnerability is correlated with its severity level. Devastating vulnerabilities such as Re-entrancy, DoS by external contract, tx.origin usage, timestamp dependency and unchecked external call are recognized. The others are given low warning such as redundant fallback function, private modifier, Token API violation.

3.3 Securify: [14] (Applicable to Section 2.1, 2.2, 2.4, 2.5)

Securify is an open-source fully automated security analyzer tool. EVM bytecode and security properties are given as input to the Securify system. This tool defines 2 patterns – compliance patterns and violation patterns. Violation pattern is found in the following manner – Symbolic analysis is carried out on the dependency graph. This extracts semantic information. And the critical code is checked with sufficient condition to prove whether a property exists or not.
The steps followed are:

a) The stack-oriented bytecode is decompiled into an assignment-based form i.e. each variable in the program is assigned exactly only once
b) From this representation of the code, DataLog facts or semantic facts are inferred along with data and control flow dependencies.
c) Lastly, the set of compliance and violation security patterns are checked. The patterns are coded as DataLog rules.

According to their website [15], they guarantee for finding specific vulnerabilities and scope of development to capture all newly discovered vulnerabilities. This tool increases efficiency because of domain specific verifiers, reduced manual effort and auditing smart contracts.

3.4 Gasper: [11] (Applicable to Section 2.8)

Gasper is a static analysis security tool which focuses on automatically recognizing gas costly programming patterns and patterns with inefficient gas consumptions. This tool is in the research stage and not been released yet. It takes input only as the bytecode. It follows symbolic execution and constructs CFG. It then searches for dead code and loops containing expensive operations. Seven Solidity gas costly patterns have been identified and segregated into 2 categories.

The steps followed are:

a) The bytecode of the contract is disassembled using the disasm command.
b) After that a control flow graph is built.
c) Then symbolic execution of CFG is carried out by traversing through the graph, starting from the root node.
d) If a conditional jump is reached during the traversal, Z3 solver is used to query the condition to find its feasibility.

3.5 Zeus: [16] (Applicable to Section 2.1, 2.2, 2.3, 2.4)

Zeus is a framework designed to verify the correctness and validate the fairness of smart contracts. It has three components – a) policy builder b) source code translator and c) verifier. It is necessary to provide two inputs to Zeus: the Solidity source code and a security policy written in a specific language. To analyze the contract, a sound approximation of the semantics of the program and symbolic model checking is done. The steps followed by the three components are:

a) Policy Builder (Taint Analysis): The user must define the policy which has the safety properties expressed in it.
b) Solidity to LLVM Translator: Here, the code and policy specifications are translated to Low level Virtual Machine (LLVM) bytecode.
c) Verifier: The verifier states assertion violations. This is done by performing static analysis on the code and then appending the assert statement at the right place in the program.

Formal Verification:

3.6 F Framework: [17]*

F* is a functional programming language. [18] This method follows formal verification that is verifying the functional correctness of a program using mathematical techniques.

The steps followed by F* framework architecture are:

a) Firstly, the Solidity code is translated into F* as it is difficult to directly modify EVM.
b) On the other hand, a de-compiler is used that converts EVM bytecode to F* programs.

F* focuses on providing program verification. This is done as the system proves and disproves the correctness of a program using SMT solving. In this way, source-level correctness and low-level properties such as gas consumption, execution time, etc. both are analyzed and verified. This tool is still in the research stage, there are other frameworks also using formal verification such as Formalization using Isabelle/HOL [19] , FEher interpreter using Coq [20].

Dynamic Analysis:

3.7 MAIAN: [4]

This tool follows dynamic analysis to find the vulnerabilities in the execution traces. An execution trace is the sequence of running a contract recorded on the blockchain as per the order of test cases. It is necessary to provide the tool 2 inputs – contract bytecode and analysis specifications. The analysis specifications component defines the vulnerabilities and the depth of the search within the contract (Figure 2) This tool follows symbolic execution and thus execution traces of the program are generated.
The steps followed are:

a) When a bug is found, the symbolic analysis component returns concrete values for the variables assigned to the traces.
b) After that, the concrete validation component validates the obtained results. And to discard false positives, dynamic analysis of the contracts is conducted. This testing is done by deploying them on a private blockchain and attacking them with computed transactions.

It was observed in this tool that the state of the contracts remains intact on the main Ethereum blockchain. This is promising as it has no effect on the state.

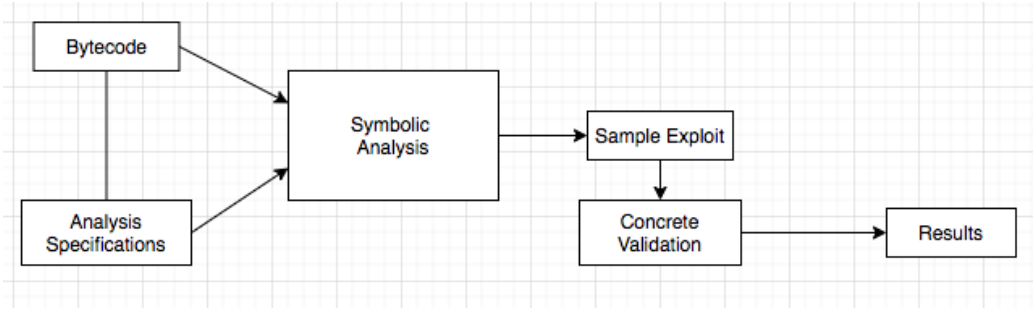

Fig. 2 – Architecture of MAIAN

Few other dynamic analysis methods include – contractLarva [21] , ReGuard [22] , teEther [23].

4. Analysis/Review

For best outcome, the identified tools must be analyzed and compared on the basis of various parameters and 3 assessments: effectiveness, accuracy and consistency. Control flow graphs can be considered as simpler than AST's as they are less redundant and use simpler expressions. Control flow graphs turn helpful for syntax analysis as most contracts fail there. However, Control flow graphs can fail to check initialization nodes and also miss some paths. Similarly, the tools can be accessed via an online web interface or on a command line interface, that is a text-based interface used to enter commands. It prompts on the user's screen and the key commands are entered and processed.

Table 2 - A comparison table for Tools.

Name of Tool / Features	OYENTE [13]	SmartCheck [10]	Securify [14]	GASPER [11]	ZEUS [16]	F* Framework [17]	MAIAN [4]
Type of Analysis	Static	Static	Static	Static	Static	Formal Verification	Dynamic
Code Transformation method	Disassembly, Control flow graph	Abstract Syntax Tree Analysis	Disassembly, De-compilation	Disassembly, Control flow graph	Abstract Syntax Tree Analysis	Formal Methods – translation to formal programming language	Disassembly, Control flow graph
Methodology for analysis	Symbolic Execution and constraint solving for Bytecode	Performs analysis on Solidity code	Abstract interpretation and Datalog (Horn Logic) for Bytecode	Symbolic Execution and constraint solving for Bytecode	Constraint Solving on Solidity code	Uses theorem provers and constructs program logics	Symbolic Execution and constraint solving for Bytecode
Usability & setup	CLI and Web Interface both. Provides docker image to deploy the app.	Web Interface.	Web Interface online.	N/A	N/A	CLI Initial setup takes significant time	Written in python, CLI tool.
Performance metrics	This tool provides high accuracy but detects only 4 vulnerabilities.	This tool fails to detect bugs that can be affected by user input, i.e. taint analysis	This tool generated a positive rate of true warnings. However, it fails to comprehend numerical analysis	This tool detects only 1 vulnerability – that is Gas costly patterns	This tool generated zero false negatives and attributes involving mathematical operations are not validated.	Only focuses on functional correctness and run-time safety. This tool is unable to detect bugs.	This tool traced through an invocation method and generated a high true positive rate.

Prevention Mechanism	This tool consists of verification methods for eliminating the false positives.	This tool flags the code and suggests solutions for preventing the attack.	This tool generates a command when a violation is found. It is then tried to match with a compliance pattern. A warning is generated if the compliance and violation pattern don't match.	N/A	This tool appends assert statement in the program. In this tool, there is scope of development to scan contracts on other platforms.	N/A	To discard the false positives, the contracts are analyzed on a live block in this tool.

5. Conclusion & Future Work

Smart contract technologies evict the involvement of intermediaries such as lawyers or banks due to their decentralized nature. This allows users to shift to digital agreements from traditional paper-based contracts. They are transparent, time-efficient, irreversible, precise, and enable users to make savings. However, code-only executable contracts are difficult to adapt. This paper reviewed various security vulnerabilities on Ethereum based smart contracts and analysis tools used to detect them. The tools should not fail to detect all possible vulnerabilities and should not give a false sense of security. It is clear from the study that there are 3 methods; static analysis, dynamic analysis [24] and formal verification. [25] Static analysis consists of the advanced automated tools such as Securify and SmartCheck detecting maximum vulnerabilities. It was observed that these two proved to be most efficient. However, they fail to detect vulnerabilities during execution time. Formal verification methods use functional programming methods. They aim to prove if the properties in smart contracts are performing correct or not. They validate run-time safety and functional correctness. Dynamic Analysis methods similarly are easy to use and can turn out to be handy while detecting bugs. Although the paper summarizes and identifies as many tools and countermeasures as possible, there is a long way to go.

Future work includes development of tools carrying out complete analysis of the code for all possible attacks. And developers should take more care while writing the programs as the distributed and immutable nature of blockchain can have unpredictable consequences.

REFERENCES

[1] Elva L., Besnik S. and Luis L. : Systematic Literature Review of Blockchain Applications-Smart Contracts. IEEE International Conference on Information Technologies, Bulgaria (2019)
[2] M. Alharby, A. Moorsel (2017). Blockchain-based smart contracts: A systematic mapping study. *ArXiv, abs/1710.06372*.
[3] Wang S., Yuan Y., Wang X., Li J., Qin R., Wang F. : An overview of Smart Contract: Architecture, Applications and Future Trends. 2018 IEEE Intelligent Vehicles Symposium (IV), 108-113. (2018).
[4] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. : Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv:1802.06038*. (2018)
[5] Osman G. : The DAO Hack Explained. https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562
[6] Santiago P. : The Parity Wallet Hack Explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/
[7] https://consensys.github.io/smart-contract-best-practices/known_attacks/
[8] N. Atzei, M. Bartoletti, and T. Cimoli.: A survey of attacks on ethereum smart contracts (sok). In: International Conference on Principles of Security and Trust. Springer, pp. 164–186. (2017)
[9] King of Ether Throne Investigation, http://www.kingoftheether.com/postmortem.html
[10] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov.: SmartCheck: Static Analysis of Ethereum Smart Contracts. In: WETSEB: IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. (2018)
[11] T. Chen, X. Li, X. Luo, X. Zhang. : Under-Optimized Smart Contracts Devour Your Money. IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). (2017)
[12] T. Min, W. Cai. : A security case study for blockchain games. *arXiv preprint arXiv:1906.05538*. (2019)
[13] L. Loi, C. Duc-Hiep, O. Hrishi, P., S., and A. Hobor.: Making Smart Contracts Smarter. ACM SIGSAC Conference on Computer and Communications Security, New York USA (2016)

[14] Petar T., Andrei D., Dana C., Arthur G., Florian B., Martin V. : Securify: Practical Security Analysis of Smart Contracts. In Proceedings ACM SIGSAC Conference on Computer and Communications Security. (2018)
[15] Securify tool. https://securify.ch/
[16] M. Dhawan, S. Kalra, S. Goel, S. Sharma. : ZEUS: Analyzing Safety of Smart Contracts. Network and Distributed System Security Symposium. (2018)
[17] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier: Formal Verification of Smart Contracts: Short Paper. Proceedings of 2016 ACM Workshop on Programming Languages and Analysis for Security. (2016)
[18] https://en.wikipedia.org/wiki/F*_(programming_language)
[19] S. Amani, M. Begel, M. Bortin, M. Staples.: Towards verifying ethereum smart contract bytecode in isabelle/hol. Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. (2018)
[20] Z. Yang, H. Lei. : Fether: An extensible definitional interpreter for smart contract verifications in coq. IEEE Access. (2019)
[21] S. Azzopardi, J. Ellul, and G. J. Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. 18th Int. Conf. on Runtime Verification (RV'18). (2018)
[22] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding re-entrancy bugs in smart contracts. Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ACM. (2018)
[23] J. Krupp and C. Rossow. teEther: Gnawing at Ethereum to automatically exploit smart contracts. 27th USENIX Security Symposium (USENIX Security 18). USENIX Association. (2018)
[24] https://devqa.io/static-analysis-vs-dynamic-analysis-software-testing/
[25] https://runtimeverification.com/blog/how-formal-verification-of-smart-contracts-works/
[26] M. Vukolić: Rethinking permissioned blockchains. Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts. (2017)
[27] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi: Town crier: An authenticated data feed for smart contracts. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. (2016)
[28] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. IEEE Symposium on Security and Privacy (SP). (2016)
[29] H. Watanabe, S. Fujimura, A. Nakadaira, Y. Miyazaki, A. Akutsu, and J. J. Kishigami: Blockchain contract: A complete consensus using blockchain. IEEE 4th Global Conference on Consumer Electronics (GCCE). (2015)