A Survey on Vulnerability Detection Tools of Smart Contract Bytecode

Junzhou Xu
School of Software
Tsinghua University
Beijing, China
xujunzhou14@163.com

Fan Dang
School of Software
Tsinghua University
Beijing, China
dangfan@tsinghua.edu.cn

Xuan Ding
School of Software
Tsinghua University
Beijing, China
dingx04@gmail.com

Min Zhou
School of Software
Tsinghua University
Beijing, China
zhoumin03@gmail.com

***Abstract*—As the core of present blockchain applications, smart contracts are designed to help multiple parties reach an agreement. Along with the promotion of smart contract applications, a large number of economic losses caused by attacks on smart contract vulnerabilities have emerged. Since most smart contracts only disclose bytecode, in recent years, there have been numerous researches on the vulnerability detection of smart contract bytecode, mainly for Ethereum smart contracts, achieving considerable results. My survey summarizes the methods and supported vulnerability types of these tools, aimed at Ethereum or EOSIO, over the years. The problems reflected in it shed light on the future work of smart contract bytecode vulnerability detection.**

Keywords—vulnerability detection, smart contracts, bytecode, survey

I. Introduction

Since Satoshi Nakamoto invented Bitcoin in 2008 [1], blockchain technology, one of the underlying technologies of Bitcoin, has received increasing attention. Today, the application range of blockchain has spread from digital currency to all aspects of life. Smart contracts, the core of these applications, are agreements written in computer code, allowing people to abide by the agreements without requiring trust.

However, when writing smart contracts, developers may leave some vulnerabilities in the contracts due to the misunderstanding of the code language, the imperfect contract design, or the carelessness. These vulnerabilities are perceived and targeted by hackers, causing a lot of economic losses.

Whether it is the most popular Ethereum launched in late 2013 or the emerging EOSIO appeared in 2018 [2], [3], smart contracts and decentralized applications are gradually gaining attention and promotion. There are currently many tools that use various methods to detect various vulnerabilities in smart contracts on different platforms.

For developers, security analysis tools for high-level smart contract languages may be more in line with demand. However, for users who want to know whether the smart contracts they are using is secure, for the reason that these contracts are usually not open-source, users can only obtain the bytecodes of these contracts, so it is more realistic for users to make use of security analysis tools for smart contract bytecode. Therefore, in order to help developers and users analyze the security of smart contracts, some tools for detecting the vulnerability of smart contract bytecodes have emerged.

The methods of these tools are variable, and the types of vulnerabilities they supported are different. This survey will analyze and summarize these tools based on their methods and supported vulnerability types. Then based on the conclusions, the survey will propose some possible directions for future related work. Different from other surveys, this survey not only focuses on EVM bytecode contracts on the Ethereum platform, which most tools are aimed at, but also on web-assembly(WASM) contracts on the EOSIO platform.

II. Methods

This survey divides the methods used by the tools into three categories and briefly explains these methods.

A. Code Translation

This kind of method translates the code into another form which is easier to analyze. Code translation methods used by vulnerability detection tools include disassembly [4], [6], [9], [10], [12], [14], [18], [20], decompilation [6], *etc.*

Disassembly is a method to translate bytecode into readable assembly language, using symbols and labels to represent operations and addresses. Decompilation is a method to translate bytecode into a higher-level language, trying to reconstruct the original source code. Disassembly and decompilation are two similar methods that improve the readability of the code.

B. Static Analysis

Static analysis is a method that examining the code without actually executing the program. This method will obtain the overall structure of the code and abstract the code information for inference. Most of the tools are based on static analysis. Static analysis methods used by vulnerability detection tools include control flow analysis [4], [9], [12], [14], [18], [20], pattern matching [6], [18], data flow analysis [6], symbolic execution [4], [9], [10], [12], [14], [20], *etc.*

Control flow analysis is a method that uses a control flow graph (CFG) to represent paths traversed through a program during execution. In a CFG, the nodes represent the basic blocks of the program, and the edges represent the running order among blocks. A CFG will help tools confirm the control process of the program.

Pattern matching is a method that searches some patterns in a given sequence of instructions. It first defines some patterns that are secure or vulnerable and then checks the code to find whether there is a match. The main concern is how to define secure patterns and insecure patterns simply but precisely.

Data flow analysis is a method of collecting information about the dependencies and the possible ranges of values at various points in a program. The points are usually determined with the aid of the control flow graph. The main concern is how to infer the information at each point efficiently.

Symbolic execution is a method of executing a program at a symbolic level. It treats values as symbols and code instructions as symbolic equations, solving equations to reason about the logic of code execution. Each symbolic path has some constraints, indicating the restriction of the symbolic inputs of this path. By adding additional specific constraints to the equation set, symbolic execution can determine whether a program may have a corresponding output. Most of the symbolic execution tools use Z3-Solver to help to solve the equation set. The main challenges of symbolic execution usually include three parts: path exploration, constraint resolution, and memory modeling.

C. Dynamic Analysis

Dynamic analysis is a method that examining the code by executing it on a real or virtual processor. Among the vulnerability detection tools, dynamic analysis methods are used far less frequently. Some vulnerability detection tools make use of dynamic analysis to validate the results of static analysis [12], [14], while others take advantage of fuzzing [16].

Fuzzing is used to generate unexpected or random inputs for the program and to monitor for exceptions. The first challenge of fuzzing is how to generate the inputs so that they are unexpected enough to find the vulnerabilities but at the same time reasonable enough so the vulnerabilities can be triggered in practice. The second challenge is how to define the conditions for whether a vulnerability is established.

III. Vulnerability Types

In addition to general vulnerability types such as integer overflow, smart contracts also have unique vulnerability types due to the characteristics of the blockchain platform.

A. General Vulnerability Types

1) Integer overflow (IO)

Usually, the integer type in smart contract language, *e.g.*, uint256 in Solidity and i64 in web-assembly, has a limited range. If the value of an integer variable exceeds the range, the value will be adjusted into the range, thereby obtaining incorrect results.

2) Permission verification missing (PVM)

Some key function calls and read/write operations need to verify the user's permission. If there is a lack of permission verification, hackers will be able to perform unauthorized operations, which will cause losses. This kind of vulnerability can be further subdivided into unrestricted write, unrestricted transfer, unrestricted call, suicidal contract, *etc.*

3) Exception handling error (EHE)

When a program receives illegal inputs, the program needs to handle the exception. Specifically, when a contract calls an external function, the contract needs to check the return value to determine whether the call is successful. A specific example in Solidity contracts is unchecked and failed sendings: as the send instruction will not throw any exception or error message when the sending is failed, if there is no exception handling implemented in send method, there may be errors in the balance calculation.

B. Unique vulnerabilities in Smart Contract

1) Transaction ordering dependency (TOD)

The transaction takes a certain time from initiation to confirmation. If someone initiates a transaction to modify the contract during this period and the modified transaction is confirmed earlier, the transaction initiated earlier will be affected. This dependency on the order of transactions is a critical problem in practice, *e.g.* seller may change the price after buyer's buying so that buyer will be forced to pay more without consent.

2) Predictable random number (PRN)

The contract may use predictable seeds, such as block timestamps and block numbers, to generate pseudo-random numbers. In gambling contracts, hackers may take advantage of this to increase their winning rate. In these types of vulnerabilities, most tools detect timestamp dependencies only.

3) Reentrancy (RE)

In a Solidity contract, there is a function called the fallback function that will be called when an account is sent a call method. If the callback function invokes a call method, this process may be repeated until the gas is exhausted.

4) Frozen Tokens (FT)

Some contracts rely on external library contracts to transfer tokens. If the external library contract is terminated or destructed, this contract cannot transfer tokens to other contracts, which means that the tokens are frozen in the contract and cannot be consumed.

5) Fake EOS and fake notice (FF)

In a EOSIO contract, there must be an apply function as a corresponding action handler. In addition, the eosio.token contract is a token standard contract in EOSIO, responsible for all token management in EOSIO. If user A wants to send tokens to B, A will push transfer action to eosio.token and then eosio.token will send a notice to B, triggering B's apply function. Both fake EOS and fake notice involve this process. Regarding fake EOS, if the apply function doesn't verify that the sender of notice is eosio.token, the contract may mistake false tokens for true tokens. About fake notice, if the apply function doesn't verify that the receiver of the notice is the contract itself, the contract may mistake notices forwarded by other contracts as its own.

IV. Tools

In recent years, many vulnerability detection tools have been developed for smart contract bytecode, of which most aim at Ethereum. Table I shows the methods and supported vulnerability types of these tools briefly.

TABLE I. METHODS AND SUPPORTED VULNERABILITY TYPES OF TOOLS

Tool	Method						Supported Vulnerability Type							
	Disassembly	***CFG***	***Pattern matching***	***Symbolic execution***	***Result validation***	***Fuzzing***	***IO***	***PVM***	***EHE***	***TOD***	***PRN***	***RE***	***FT***	***FF***
Oyente	√	√		√					√	√	√	√		
Securify	√		√					√	√	√			√	
Mythril	√	√		√			√	√	√		√	√	√	
Manticore	√			√			√	√	√			√		
teEther	√	√		√	√			√						
MAIAN	√	√		√	√			√					√	
ContractFuzzer						√			√		√	√	√	
EVulHunter	√	√	√											√
EOSafe	√	√		√				√			√			√

A. Tools for Ethereum Contracts

1) Oyente

As a starting milestone in this field, Oyente is a static analysis tool based on symbolic execution that can be run directly on EVM bytecode without accessing high-level languages such as Solidity [4], [5]. Oyente supports the detection of vulnerabilities such as TOD, the predictable random number (timestamp dependency), reentrancy, and exception handling error.

Oyente has four modules: CFGBuilder, Explorer, CoreAnalysis, and Validator. CFGBuilder constructs a CFG of the contract; Explorer symbolically executes the contract; CoreAnalysis takes in the outputs of Explorer, locating the vulnerabilities; Validator uses Z3-Solver to filter out some false positives of TOD detection and reports the final result to users.

Oyente covers most of the EVM opcodes, but due to the lack of context information such as variable types and the reuse of same bytecode by different function calls, it is difficult for Oyente to reconstruct the development intent only from the EVM bytecode, thus it cannot verify some issues on fairness and correctness such as integer overflow. Oyente simplifies the processing of loops by limiting the number of loops to prevent path explosions, which leads to the underreporting of some defects.

2) Securify

Securify is a lightweight and scalable security verifier for Ethereum smart contracts [6], [7]. As a static analysis tool based on symbolic abstraction and pattern matching, Securify defines compliance patterns and violation patterns for each security attribute, and then match the contract with these patterns to detect vulnerabilities. Securify supports detection of vulnerabilities such as frozen tokens, permission verification missing (unrestricted write and unrestricted transfer), TOD, argument validation missing, and exception handling error.

Starting with the EVM bytecode of the contract, Securify decompiles the bytecode into a static-single assignment form (SSA). After symbolically encoding the dependence graph of the contracts in stratified Datalog, Securify uses ready-made Datalog solvers to analyze the Datalog code and get semantic facts of contract efficiently. The semantic facts include data flow dependency and control flow dependency. The compliance and violation patterns are also defined in a designated domain-specific language (DSL). The matching result of these patterns in the contract will reveal whether the contract is safe.

Recently, Securify officially released version 2.0, which supports an updated version of the smart contract language and more detailed types of vulnerability detection [8]. For instance, Securify2 refines TOD into three vulnerabilities: TODAmount, TODReceiver, and TODTransfer.

3) Mythril

Mythril is a security analysis tool for Ethereum contracts based on symbolic execution and taint analysis [9]. After disassembling the EVM bytecode, Mythril initializes the state of the contract account and uses a couple of transactions to explore the state space of the contract. When an undesired state is discovered, Mythril uses Z3-Solver to prove or deny its reachability under certain assumptions. When a vulnerability state is discovered, Mythril will calculate the transactions required to reach that state to verify the existence of the vulnerability.

Mythril supports the detection of vulnerabilities such as integer overflow, permission verification missing (unrestricted write, unrestricted jump, suicidal contract), exception handling error (unchecked call return value), reentrancy, predictable random number, frozen tokens, *etc.*

4) Manticore

Manticore is a symbolic execution framework for the analysis of Ethereum smart contracts as well as Linux ELF binaries [10], [11]. Based on EVM bytecode, Manticore can execute the contract with symbolic transactions where both value and data are symbolic and explore all possible states, generating corresponding concrete inputs for any program state with Z3-Solver. On this basis, Manticore can detect vulnerabilities in contracts. Through event callbacks and instruction hooking, Manticore can control the exploration of the state at a fine-grained level. In a default contract analysis, Manticore gets 66% code coverage on average.

Manticore supports the detection of vulnerabilities such as integer overflow, reentrancy, permission verification missing (external call or ether leak, suicidal contract), exception handling error (unchecked call return value), *etc.*

Manticore is currently under development and does not cover all opcodes. The official recommendation is to compile the contract with Solidity version 0.4.x to ensure the validity of the tool analysis.

5) teEther

teEther is an analysis tool for Ethereum EVM contracts based on symbolic execution and result validation, focusing on detecting permission verification missing (unrestricted call) [12], [13]. The process of teEther analyzing contracts can be divided into five steps: The first step is to build a CFG for the contract; The second step is to scan the contract for important instructions, including critical instructions and state-changing instructions, *e.g.* DELEGATECALL, SELFDESTRUCT, SSTORE, *etc.*; The third step is to explore paths to these instructions; The fourth step is to generate a set of path constraints through symbolic execution; The last step is to solve the constraints of these paths to detect the vulnerabilities. To validate the detection results, teEther will test the contract on a private blockchain.

6) MAIAN

MAIAN is an analysis tool for Ethereum EVM contracts based on symbolic execution and result validation [14], [15]. It symbolic executes the contract with Z3-Solver, checking the paths of execution. To validate the detection results, MAIAN will test the contract on a private blockchain, attacking the contract with concrete transactions.

MAIAN supports the detection of vulnerabilities including permission verification missing (unrestricted transfer and suicidal contract) and frozen tokens.

7) ContractFuzzer

ContractFuzzer is a fuzzer to detect vulnerabilities in Ethereum EVM contracts [16], [17]. In this work is proposed the first fuzzing framework and a set of new test oracles for detecting vulnerabilities in Ethereum contracts.

ContractFuzzer consists of two parts: offline instrumentation and online fuzzing. The offline instrumentation part is to instrument the EVM code in order for the fuzzing part to monitor the execution of the contract. In the online fuzzing part, after analyzing the application binary interface (ABI) and the EVM bytecode of the contract, ContractFuzzer will extract the information of ABI functions, which helps the tool generate valid fuzzing inputs. The fuzzing inputs of function calls to the external contracts will be randomly selected from the smart contracts crawled on Ethereum by the tool.

ContractFuzzer supports the detection of vulnerabilities including exception handling error, reentrancy, predictable random number, frozen tokens, *etc.*

B. Tools for EOSIO Contracts

1) EVulHunter

As the first vulnerability detection tool designed for EOSIO, EVulHunter is a static analysis tool for EOSIO WASM contracts based on pattern matching [18], [19]. Unlike other tools, this tool is designed specifically for fake EOS and fake notice detection.

EVulHunter consists of three modules: CFG Builder, WASM Simulator, and Detector Engine. In the CFG Builder module, EVulHunter builds the CFG of the contract based on a ready-made tool Octopus, a security analysis framework for WASM and smart contracts. WASM Simulator module works as a virtual machine for further analyses, modifying a Stack and Memory structure during tracing instruction of the WASM code. In order to recover the semantic type information, the module also observes and summarizes several special patterns including some important parameters and strings (in a format of 32-bit encoding integer). In the Detector Engine module, two detectors for fake EOS and fake notice are implemented respectively, each detector including a pattern of corresponding vulnerability for matching.

Considering the difference between the various versions of CDT, EVulHunter covers and analyzes all variants, including patterns, pairs, and elements in the comparison mechanism. In the validation, the tool got full accuracy for fake notice vulnerability. For fake EOS vulnerability, EVulHunter got some false positives for the reason that the contracts acknowledged the legitimacy of an additional account.

2) EOSafe

EOSafe is a static analysis framework for vulnerability detection in EOSIO WASM smart contracts, based on symbolic execution [20], [21]. EOSafe supports the detection of vulnerabilities such as fake EOS, fake notice, predictable random number, and permission verification missing.

EOSafe is composed of three modules mainly: Engine, Emulator, and Scanner. The engine is short for Symbolic Execution Engine, designed as a platform for execution imitation of a contract. Receiving the CFG and dissembled instructions of the contract as the inputs, Engine symbolic executes the code within basic blocks, exploring all workable paths and gathering path constraints. The emulator is short for EOSIO Library Emulator, emulating the side effects of imported functions in the contract. The scanner is short for Vulnerability Scanner, locating suspicious functions, and detecting vulnerabilities.

In the validation, EOSafe got full accuracy in detecting permission verification missing, fake EOS and fake notice. EOSafe got one false negative in detecting predictable random number, for the reason that the tool did not explore enough paths before timeout.

V. Discussion

Although there have been considerable achievements in the vulnerability detection of smart contract bytecode, these tools still have some common problems or areas that can be improved.

A. Platform and Language

As the most popular blockchain platform, Ethereum is naturally the main target of these tools. Compared to Ethereum, the emergence and popularity of EOSIO are the latter. With the overall architecture of the platform design far more complicated than Ethereum, it is reasonable that the tools for EOSIO are far less than Ethereum. However, with the development of decentralized applications on EOSIO, the security analysis of EOSIO contract should naturally receive attention. In addition, Ethereum is also designing Ethereum flavored web-assembly (ewasm), a subset of WASM to be used for Ethereum contracts, so the development of vulnerability detection tools for WASM contracts will be an important direction next.

B. Vulnerable Types

Difficulties in detecting different types of vulnerabilities vary. For example, simple pattern matching can detect fake EOS and fake notice precisely, but neither symbol execution nor fuzzing can guarantee 100% accurate detection of TOD. Even if the methods used are similar, the range of vulnerability types detected by different tools is also inconsistent. Therefore, how to accurately detect all currently known types of vulnerabilities in a single system is still a problem to be solved.

C. Automated Validation of Result

There are two tools in this survey, teEther, and MAIAN, that validate the results of static analysis through dynamic analysis, but the specific methods of these two tools are manually testing the contract on private blockchains. If the results of static analysis can be automatically validated by the tools, not only can the accuracy of tool detection be improved, but also the labor cost can be reduced.

VI. Conclusion

Whether on the most popular blockchain platform Ethereum or the rising platform EOSIO, smart contracts and distributed applications are gradually gaining attention and promotion. Compared to tools for high-level languages, tools for bytecode are more versatile for users and developers. There are currently many tools that use various methods to detect various vulnerabilities in smart contract bytecode on different platforms. These tools have their advantages and disadvantages in supporting language versions, detecting vulnerability types, accuracy, and performance. This survey summarizes the types of vulnerabilities that have emerged in smart contracts, sorts out the methods and types of vulnerabilities detected by current vulnerability detection tools, and finally puts forward some ideas for possible directions for future work.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Nov. 1, 2008. Accessed on: Jun. 14, 2020. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum project yellow paper 151, 2014, pp. 1-32.

[3] "EOS.IO Technical White Paper v2," Mar. 16, 2018. Accessed on: Jun. 14, 2020. [Online]. Available: https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md

[4] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 254-269.

[5] "Oyente: An Analysis Tool for Smart Contracts," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/melonproject/oyente

[6] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 67-82.

[7] "Securify: Security Scanner for Ethereum Smart Contracts," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/eth-sri/securify

[8] "Securify2: Securify v2.0," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/eth-sri/securify2

[9] "Mythril: Security analysis tool for EVM bytecode," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/ConsenSys/mythril-classic.

[10] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019, pp. 1186-1189.

[11] "Manticore: Symbolic execution tool," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/trailofbits/manticore

[12] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, 2018, pp. 1317–1333.

[13] "teEther: Analysis and automatic exploitation framework for Ethereum smart contracts," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/nescio007/teether

[14] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," arXiv:1802.06038, 2018.

[15] "MAIAN: automatic tool for finding trace vulnerabilities in Ethereum smart contracts," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/ivicanikolicsg/MAIAN

[16] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 259-269.

[17] "ContractFuzzer: The Ethereum Smart Contract Fuzzer for Security Vulnerability Detection," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/gongbell/ContractFuzzer

[18] L. Quan, L. Wu, and H. Wang, "EVulHunter: Detecting Fake Transfer Vulnerabilities for EOSIO's Smart Contracts at Webassembly-level," arXiv:1906.10362, 2019.

[19] "EVulHunter," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/EVulHunter/EVulHunter

[20] N. He, R. Zhang, L. Wu, H. Wang, X. Luo, Y. Guo, T. Yu, and X. Jiang, "Security analysis of EOSIO smart contracts," arXiv:2003.06568, 2020.

[21] "EOSafe: A powerful on-chain smart-contract wallet on the EOSIO platform," Accessed on: Jun. 14, 2020. [Online] Available: https://github.com/xJonathanLEI/EOSafe