

Bug Searching in Smart Contract

1st Xiaotao Feng

Swinburne University of Technology
Melbourne, Australia
101973718@student.swin.edu.com

2nd Qin Wang

Swinburne University of Technology
Melbourne, Australia
qinwang@swin.edu.au

3rd Xiaogang Zhu

Swinburne University of Technology
Melbourne, Australia
xiaogangzhu@swin.edu.au

4th Sheng Wen

Swinburne University of Technology
Melbourne, Australia
swen@swin.edu.au

Abstract—With the frantic development of smart contracts on the Ethereum platform, its market value has also climbed. In 2016, people were shocked by the loss of nearly \$50 million in cryptocurrencies from the DAO reentrancy attack. Due to the tremendous amount of money flowing in smart contracts, its security has attracted much attention of researchers. In this paper, we investigated several common smart contract vulnerabilities and analyzed their possible scenarios and how they may be exploited. Furthermore, we survey the smart contract vulnerability detection tools for the Ethereum platform in recent years. We found that these tools have similar prototypes in software vulnerability detection technology. Moreover, for the features of public distribution systems such as Ethereum, we present the new challenges that these software vulnerability detection technologies face.

Index Terms—Blockchain, Smart Contract, Ethereum, Formal Verification, Fuzzing, Symbolic Execution

I. INTRODUCTION

Decentralized cryptocurrencies have gained tremendous attention from both academia and industry. The emerging novel technology originated from these systems is blockchain, a sequentially ordered ledger system. The system possesses the properties of being distributive, irreversible, unforgeable, and traceable. Ethereum [1], as the most accessible blockchain platform, supports distributed applications in different scenarios through the underlying online virtual machine called EVM, which is a fundamental layer for the complete execution of the smart contracts. Smart contract [2] [3] is a collection of code and data (also known as states) executing on the blockchain system. It is Turing-complete which allows us to write the pre-defined rules. Smart contract is pretty suitable for the scenarios requiring dependable security, irreversible persistence, and high trusts, such as the digital assets, online voting, gambling games, insurance, property managements, and financial applications.

However, there are many vulnerabilities in smart contracts [4] [5] [6] [7] [8] [9] [10], and the high financial status brings higher interaction risks. Unlike traditional distributed application platforms, smart contract platforms such as Ethereum allow anyone to join. This high openness makes the EVM environment very vulnerable. Also, some vulnerabilities can only be exploited in some of Ethereum's unique new features

(such as timestamp design, gas settings and fallback function). For example, The DAO [11] exploits a variety of well-documented reentry attacks, resulting in the theft of Ethernet worth more than \$50 million. As a result, the security issue of smart contracts on Ethereum has attracted much attention.

In the world of vulnerability detection, researchers have developed many tools to find vulnerabilities in programs. We focus on three tools, including fuzz testing [12], formal verification, and symbolic execution, that have already found many vulnerabilities. However, as for such tools applying in the smart contract, it is still at the beginning of the research. The nature of fuzzing is to generate inputs for programs and tries to find vulnerabilities based on the results of executing programs [12]. The biggest challenge of fuzzing comes from the fact that it tests a program in random behavior. Formal verification is a method based on logical deduction and tries to prove or disprove the correctness of a program. Researchers have to prove their conclusions strictly so that the target program can be fully trusted or abandoned. The last tool we introduce is symbolic execution, which treats variables in a program as symbolic values. It regards each condition in the program as a constraint and tries to find a possible solution along with an execution path. The main challenge of symbolic execution is path explosion which results from loops or arrays. When such tools are applied in smart contract, researchers have to figure out specific methods to fit tools into smart contracts. The details will be discussed in the following.

In this paper, we survey some common vulnerabilities on EVM and their triggering mechanisms. We also introduce some of the tools in the software vulnerability detection industries on this new platform and their work-flow as well as features. Further, for some new features of the EVM platform, we also present some new challenges for these software vulnerability detection tools.

II. OVERVIEW OF EVM

A. EVM Model

Ethereum Virtual Machine (EVM) provides a practical environment for the execution of smart contracts in the distributed Ethereum platform. It refers to a complete suite of logic processes of deploying, compiling, and executing [1]. Users

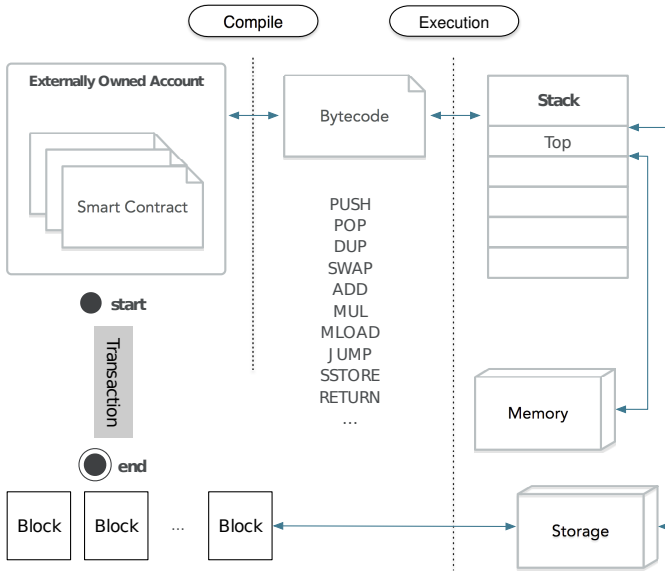


Fig. 1. EVM Model

can make their contracts automatically be executed according to the pre-defined rule via transaction-based state transitions [2]. To achieve the state machine transition, the smart contract can be seen as one type of shared-state. The globally shared-state is made up by small units, called *account*, which can interact with each other through messaging. Each account associates with a state and a 20-byte address as the identifier. There are two different types of accounts:

- *Externally owned account*, controlled by a private key without any associated code.
- *Contract account*, controlled by the corresponding contract code, which sets the actions and operations.

The externally owned account sends the message to another externally owned account or contract account by creating and signing the transaction under the private key. The message transmitted between two externally owned account is just a simple value transfer. But a message from an externally owned account to a contract account will activate the code in the contract account, performing the corresponding actions (such as transferring tokens, logging storage, generating new contracts, minting new tokens, calculating values, etc.). Unlike an externally owned account, the contract account cannot initiate a transaction by its own. Instead, the contract account triggers the transaction only after receiving a transaction. The EVM model is presented in the Fig.1.

There are four components in both types of the accounts:

- *Nonce*: It represents the transaction number sent by the externally owned account, and it also represents the contract number created by the contract account.
- *Balance*: It means the remaining value of the address, dominated as *Wei* where 1 Ether=10¹⁸ Wei.
- *Root*: The root represents the hash value of the Merkle Patricia tree which encodes the Superimposed hash of the stored content in each account.

- *CodeHash*: It is specially used for contract account, where the code of smart contract is saved as codeHash. For externally owned accounts, this field is an empty string.

B. Features in EVM

a) *Gas As Fee*: Every defined calculation generated by transactions requires an amount of cost to be paid. Gas is used to measure the unit of cost in each specific calculation. Correspondingly, Gas price is the value spent on each gas, measured by “gwei” where 1 gwei=1,000,000,000 wei. For each transaction, the sender needs to set two parameters including gas limit and gas price. The total amount represents the cap that the sender is willing to pay for the execution of the transaction. The total amount is calculated by the gas limit and the gas price limit as follow.

$$\text{Gas Limit} \times \text{Gas Price} = \text{Total amount}$$

Any unused gas at the end of the transaction will be returned to the sender for redemption. If there does not exist enough gas to execute the transaction, the transaction will be considered invalid. In this case, the process terminates where all changed states return back to the initial states. Since the system is still working on calculation before running out of gas, no gas will theoretically be returned back to the sender. Instead, all fees used in execution are sent to the miners as the reward who have already made an effort to calculate and verify the transaction.

b) *Fallback Function in Solidity*: The *fallback* function, is an uniquely unnamed function in the solidity. It is automatically executed only when no other function matches the specified function identifier [2]. At the same time whenever the address of the contract receives plain Ether without messages, the function can be called. *fallback* function has no arguments or return nothing. More specifically, when we transfer tokens without any readable data via the function `address.send(ether)`, the contract will automatically execute the *fallback* function to make the transition of state proceed. To make the `send` outputs TRUE, the *fallback* function must be marked *payable*.

Since the `send` function always calls *fallback*, it is dangerous to be attacked by malicious attackers such as DAO [11]. Especially in the scenario of dividend where the `send` operation is deployed on a series of accounts if there exists at least one malicious account holding *fallback* functions to infinite loop, it will cause all `send` processes to fail where the gas is used out. In order to solve this problem, the `send` function sets a limited 2300 gas as the maximum even if the gas is sufficient. Therefore, except for the operations such as log in the *fallback* function, one can hardly do anything to keep the liveness of the whole system.

c) *Timestamp*: Timestamp, as one kind of identifier of blocks and transactions, contained in each block header in the form of Unix time. In addition, the irreversible timestamp avoids the faking of blocks by adversaries. Based on timestamp, blockchain system confirms that each block is sequentially connected. The timestamp proves the sequence

of events in which no one can tamper with it. Timestamp can be seen as the role of notary in blockchain, which is credible for the public participants.

III. VULNERABILITIES

In this section, we will discuss three common vulnerabilities in smart contract and present the examples of attacks.

A. Reentrancy

Recursion is a widespread logical processing method in traditional programming languages, but this operation is likely to become a vulnerability in Solidity.

Fig. 2 Ebank contract shows the implementation of a contract for a public bank. Any user can deposit Ether to **EBank** and the contract records the Ether of each account. In this scenario, when the contract withdraws the saving (**withdraw(address to, uint256 amount)**), it ensures that the account has enough balance (**require (balances[msg.sender] > amount)**) and that the bank has sufficient funds for the withdrawal. If the above two conditions are satisfied, the contract will send the Ether to user(**to.call.value(amount)()**) and change the balance of corresponding value(**balances[msg.sender] -= amount**).

In programming languages such as C and C++, the code in Fig.2 Ebank contract can be run correctly. However, such piece of code may be vulnerable in Ethereum's smart contract due to its own grammar. In Fig.2 Ebank contract, the contract utilises **call.value()** to send user Ether. Distinguished from the functions **send()** and **transfer()**, function **call.value()** gives all the rest of gas to external call (fallback function). If the target address is a contract address when making an Ether transaction, the contract's fallback function is called by default.

To show how attackers can exploit this vulnerability, we design an attack contract in Fig.2. In **startAttack()**, the contract firstly **deposit()** the specified amount tokens in bank, and one token is taken through the **withdraw()**. Because the bank contract uses **call** function to send Ether to the target, this will call the target contract's **fallback** function which calls the **withdraw()**. The users balance is modified after the Ether is transferred, the balance on the attackers account remains unchanged and the attacker can always take out Ether from the bank. Therefore, the two contracts of bank and attack fall into a recursive state until that Ether in the bank is sent to the attacker or attacker stops the contract.

B. Gasless

The EVM usually sets the gas limit at 2,300, and the miners who are good at calculating can use the delicate contract structure combined with fallback function to run out of gas. This will result in an error in the running contract so that the miner can get profits from it or achieve other goals.

There is a game called **KingOfTheEtherThrone** [13] and the game is played by sending Ether to a smart contract called KotEt(as shown in Fig.3). Players who want to be king must pay some Ether to the current king, plus a small amount of

fee to KotET contract. Then, the king will get profit from the difference between the price he paid for the throne and the price other player pays to be a new king.

Supposing a player wants to be king, he wants KotET to send a certain amount(**msg.value**) of Ether. The fallback function of KotET is called and it will check if **msg.value** is greater than the quote for the previous king setting. If it is less than the quote for the previous king setting (i.e., the auction failed), it will be abandoned. On the contrary, the player will get the throne and become the new king.

This contract seems to be fine, but there will be a gasless send bug. When **king.send(profit)** fails to execute (gas is not enough to execute fallback()), the throne will be held by this contract.

C. Timestamp

In Solidity language, it defines many block state variables [14] like timestamp, random seed and block number. Since these state variables are written at the head of each block, the malicious miner may modify it and get profit from it. These block state variables can make the Ether flows along different program paths. Here we use timestamp to illustrate how such a vulnerability is exploited by malicious miners.

Block timestamps have traditionally been used for a variety of applications, such as functions for random numbers, locking contract for a period of time, and various conditional statements based on time-varying states. Miners have the ability to adjust the timestamp slightly, and if the block timestamp is misused in a smart contract, it can prove to be quite dangerous.

Block.timestamp (or **now**) can be manipulated by miners if they have incentives to do so. We build a simple contract that is vulnerable to exploitation by miners (Fig.4).

In the Fig.4, this contract is a simple lottery. Each block has a trade to bet 1 Ether and get the chance to win the entire balance in the contract. The assumption here is that the last two digits of block.timestamp are evenly distributed. If so, there will be a 1% chance to win this lottery.

However, as far as we know, miners can adjust the time stamp according to their wishes. In this particular case, if there is enough Ether in the contract, the miner who digs out a block will be motivated to choose a **block.timestamp** (or **now**) to 100 with a timestamp of 0. In doing so, they may win Ether and block rewards in this contract.

IV. TOOLS

A. Fuzzing

Fuzzing [12] is a technique that randomly generates inputs to examine testing programs. In most situations, fuzzing intends to crash a testing program. Moreover, when it crashes, we can check whether the crash is a bug or not. Fig.5 shows the typical procedure of fuzzing. At the very first execution, fuzzing needs original inputs to be mutated. Then, new inputs are mutated and generated from the original inputs, which will examine the testing program. After the examination, fuzzing will check whether the new inputs are interesting or not. The interesting inputs are saved to be seeds, which will be chosen

<pre> Contract Ebank { address owner; mapping (address => uint256) balances; function Ebank(){ owner = msg.sender; } function deposit() payable { balances[msg.sender] += msg.value; } function withdraw(address to, uint256 amount) { require (balances[msg.sender] > amount); require (this.balance > amount); to.call.value(amount)(); balances[msg.sender] -= amount; } function balanceOf() returns (uint256) { return balances[msg.sender]; } function balanceOf(address addr) returns (uint256) { return balances[addr]; } /* Other functions hide */ } </pre>	<pre> Contract Attack { address owner; address bank; function Attack() payable { owner = msg.sender; } function deposit(uint256 amount) payable { if (this.balance > amount){ bank.call.value(amount)(bytes4(keccak256("deposit()"))); } } function withdraw(uint256 amount) { bank.call(bytes4(keccak256("withdraw(address,uint256)")), this, amount); } function startAttack(uint256 amount) { deposit(amount); withdraw(1); } function() payable{ if (msg.sender == bank){ bank.call(bytes4(keccak256("withdraw(address,uint256)")), this, msg.value); } } } </pre>
---	--

Fig. 2. Ebank and Attack contracts

```

Contract KotET {
    address King;
    address owner;
    unit public claimPrice = 1;

    function KotET() {
        owner = msg.sender;
        King = msg.sender;
    }

    function() {
        if (msg.value < claimPrice) throw;

        unit profit = calculateProfit();
        King.send(profit);
        King = msg.sender;
        claimPrice = newPrice();
    }
    /* Other functions hide */
}

```

Fig. 3. KotET contract

```

Contract Lottery {
    unit pastBlockTime

    function() payable{
        require(msg.value == 1 ether);
        require(now != pastBlockTime);
        pastBlockTime = now;
        if(now % 100 == 0){
            msg.sender.transfer(
                this.balance);
        }
        /* Other functions hide */
    }
}

```

Fig. 4. Lottery contract

as inputs. If the testing program crashes, we have to verify whether the crash is a bug.

In Jiang's work [15], they create a new fuzzer, named ContractFuzzer, which is a novel fuzzer to fuzz Ethereum smart contracts. ContractFuzzer is vulnerability detecting tool which is built based on traditional fuzzing combining with static analysis. Fig.6 shows how ContractFuzzer works on fuzzing smart contract. It provides an offline **EVM instrumentation tool** which can monitor the execution of smart contracts for subsequent analysis. The ContractFuzzer firstly works on analyzing the **ABI interface and bytecode** of the smart contract which is collected in **contract dataset**. Then the ABI function arguments and signatures will be extracted for the

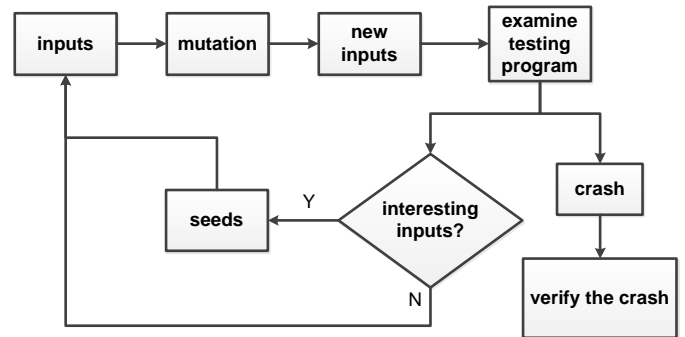


Fig. 5. The typical procedure of fuzzing. Original inputs are mutated into new inputs, and these new inputs will examine the testing program. The aim is to crash the testing program, and then check whether the crash is a bug.

ABI signature analysis. ContractFuzzer generates the input seed for online fuzzing based on the two types of analysis, and finally, it starts to do fuzzing test and detect security

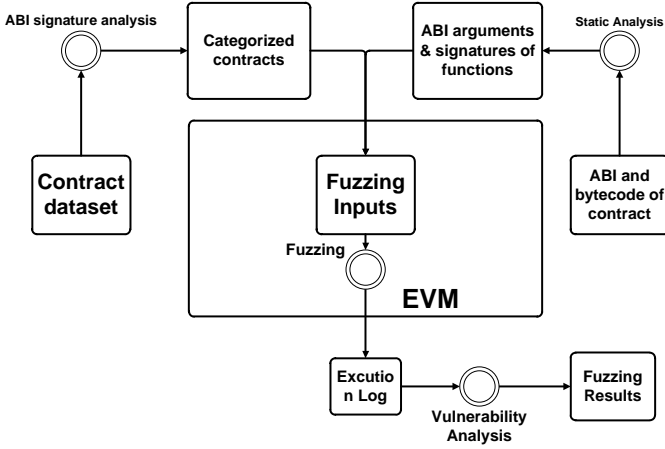


Fig. 6. Overview of the ContractFUZZer Tool

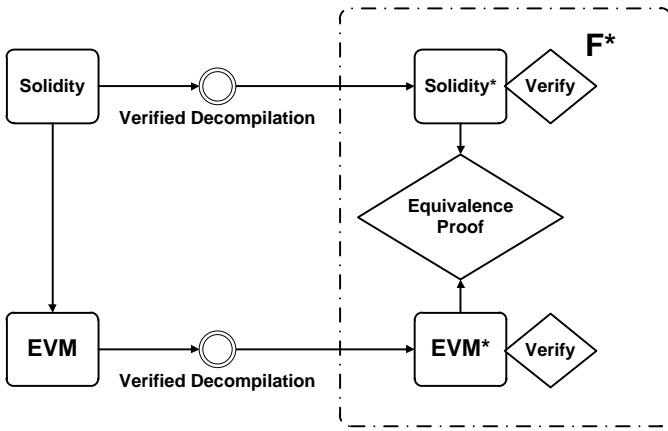


Fig. 7. Outline of our verification architecture

vulnerabilities via analyzing **execution logs**.

B. Formal Verification

In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system concerning a certain formal specification or property, using formal methods of mathematics.

Compare with the dynamic detection methods like fuzzing, for Ethereum, static methods such as the formal verification does not require a simulated execution environment and provides better precision and false positive rate in vulnerability analysis [16]. In general, automatic formal verification can be divided into three main types: 1) Automated theorem proving, 2) Model checking and 3) Abstract interpretation. For smart contract platforms, model checking is appropriate because of the smaller size of smart contracts.

Inspired by the process of processing JavaScript [17], Bhargavan’s team [18] outlines a framework to verify Ethereum smart contracts using formal verification. After translating and decompiling the Solidity code and EVM bytecode into a functional programming language named F*, they will

determine the existence of a vulnerability in the contract by verifying the equivalence of the two in the F* language results (See Fig.7 for details).

However, there are still significant limitations to this approach. Bhargavan also mentioned in their evaluation part that this language-based process cannot support many Solidity language features. It can only translate and typecheck 46 out of the 396 contracts they collected from <https://etherscan.io>.

ZEUS [19] did more work on translating the smart contract language. Zeus consists of three parts: a) policy builder, b) source code translator, and c) verifier. The policy builder performs static analysis on the smart contract code, extract the predicate from the policy condition and then insert the assertion into the contract code. Unlike [18], Zeus does not directly deal with the source code of Solidity but converts it into LLVM bytecode by the source code translator. Finally, the verifier will check the assertion inserted by the policy builder before, and then determine violations. Fig.8 is an example to explain how the Zeus works on smart contract. Firstly, ZEUS formalises Solidity Semantics into Abstract language. For the condition policy in these codes, ZEUS creates an XACML-Styled [20] five-tuple policy specification to describe and convert these policies into assert statements. The LLVM translator then helps ZEUS convert these solidity codes into LLVM’s bytecode and finally validate the code with the CHC [21] symbolic model checker.

C. Symbolic Execution

Symbolic execution is a technique based on formal verification. It analyses programs to test whether specific properties can be violated. This technique can yield strong guarantees on the checked property due to the nature of symbolic execution, which is it can simultaneously explore multiple paths. The key idea of symbolic execution is to analyze programs based on symbolic values, rather than concrete input values.

In symbolic execution, the execution part [22], which is performed by an engine, maintains each explored control flow path. This engine contains two parts: 1) a first-order Boolean formula describes the conditions included in branches of paths, and 2) a symbolic memory store maps variables to symbolic expressions. Branch execution updates the formula while a model checker verifies whether there are any violations.

Luu [16] created a static analysis tool named OYENTE based on symbolic execution technique to help developers avoid the vulnerabilities in writing smart contracts. Fig.9 is an overview of OYENTE. After extracting the smart contract bytecode from the EVM, **CFGBuilder** plots the main control flow graph (CFG) for each smart contract. Then, **Explorer** performs the Symbolic execution on these CFGs, and uses the Z3 solver [23] to complete the CFG block entry condition according to the path constraints. **Core Analysis** performs a vulnerability analysis on the collated CFGs, and at the end **Validation** will verify the analysis results.

D. Language Translation

Language translation is a common support tool for software vulnerability detection. In general, it can transform some

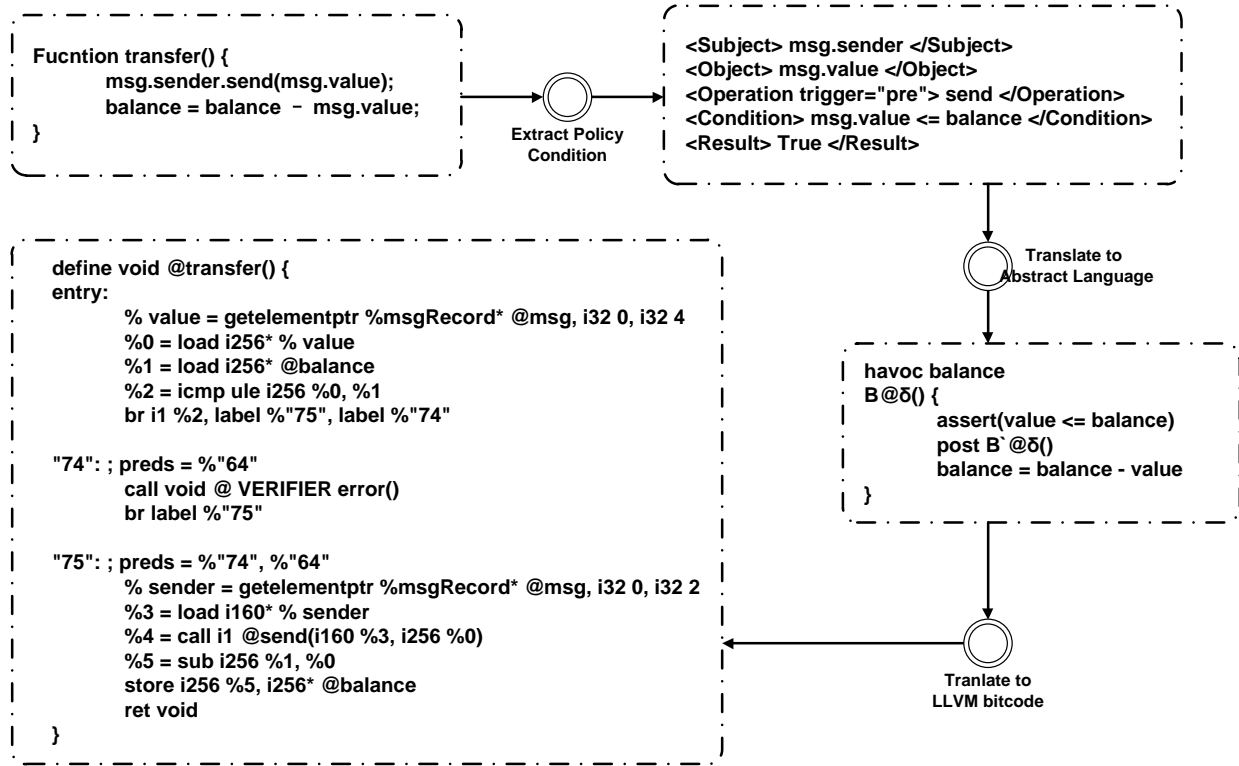


Fig. 8. An example of Zeus working flow

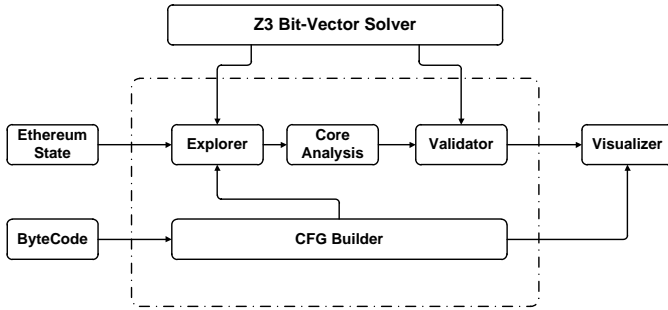


Fig. 9. Overview of the OYENTE Tool. Oyente was built based on modular design, and it consists of four main components: 1) **CFG Builder** draws a control flow graph of whole **Byte Code**; 2) **Explorer** is an interpreter loop, and it runs a single instruction when it gets a state from last the run; 3) **Core Analysis** contains several sub-components that help it analyze different security vulnerabilities; 4) **Validation** helps OYENTE remove false positive via manually verifying the results provided by **Core Analysis**.

hard-to-read or unpopular languages into some well-known programming languages. For the static method of vulnerability analysis, if you need to analyze an executable file, the general practice is to convert it into a high-level language that we know well with the decompilation tool [24] for the researchers to further processing. This kind of decompilation is a translation of assembly language into high-level languages such as C, C++, etc. Similarly, disassembly can help researchers solve problems on binary files, and in essence, it is also a translation technique. This translation technology can be used not only

in different levels of programming language but also help us achieve the conversion of peer languages, such as C++ to JAVA [25]. This kind of translation is not meaningless. It can turn some hard-to-read languages (e.g., machine code) into popular languages with many supporters, and then the program will be easier to understand, and researchers can use some sophisticated analytical techniques according to the language.

As a new language with the birth of new technologies, the programming language environment for smart contracts is far less sophisticated than other mature languages like C++ and so on. As a result, many research teams have chosen to translate the language of smart contracts into the form they want, and then use existing sophisticated software analysis techniques to detect the vulnerability problems in smart contracts. Bhargavan's team [18] translates Ethereum and smart contracts into F*, which is a functional programming language designed for program verification, to analyze the security and functional correctness of the platform while it is running. Similarly, in ZEUS's [19] work, they implemented a tool to translate solidity code into LLVM bytecode and proposed a model detection scheme based on LLVM bytecode. Leveraging LLVM bytecode helps their analytical work take advantage of the support of robust industry tools on LLVM platform. Brent et al. [26] used the decompilation method. They decompiled and analyzed the bytecode of the smart contract extracted from Ethereum, and got readable low-level mnemonics that are annotated with program counter addresses and help Vandal [26] generate the control flow graph of these smart contracts.

V. CHALLENGE

As we mentioned in this paper, software vulnerability detection methods have been well applied to smart contract platforms. However, due to the many differences among the smart contracts, the traditional software programs, and the bottlenecks of these vulnerability detection technologies, we still face challenges in the detection of vulnerability in smart contracts.

A. Fuzzing

Due to the concurrency of smart contracts and the distributed design of blockchain, fuzzing on smart contract platforms has many differences from fuzzing in the traditional sense. In software vulnerability mining, whenever fuzzing causes the target program to crash once, we consider it a potential vulnerability (requires tools or manual verification). In smart contracts, the vulnerability we mentioned earlier is almost impossible to crash the EVM, even though the collapse of a single contract does not affect the entire EVM. Therefore, as we mentioned in ContractFuzzer, when using fuzzing in smart contracts, we also need to record the results of each execution result of fuzzing and perform additional analysis to verify vulnerabilities. ContractFuzzer uses a predefined test oracle to solve this problem.

For example, for a gas-free problem, ContractFuzzer will repeatedly execute a single contract and check if the residual gas value of the `send()` function in the result analysis is zero. However, the results of some single contract executions do not reveal the vulnerability which requires special circumstances (such as reentrancy attacks, which require interactive calls between two smart contracts). In [15], for the reentrancy attack, they created three kinds of accounts and two options for `call.value()`. In this scenario, if a single contract is to be fuzzed k times, the ContractFuzzer will perform $6(2*3)*k$ times fuzzing, and then analyze the results. For this kind of reentry attack, we can make this scene design more complicated to cover more situations, but at the same time, the fuzzing execution time and analysis time will be accompanied by complexity growth. Therefore, how to find the balance between scene complexity and vulnerability patterns coverage is a challenge for fuzzing method.

Similarly, some of the problems that appear in the software fuzzing test also exist in smart contracts. Sanity checks (like magic number or checksum condition checks) in programs has always been a challenge for fuzzing, and its presence has also increased the false positive rate of fuzzing experiments. Due to the input generated by fuzzing has a high degree of randomness, fuzzing is difficult to pass for a condition check such as `Str == "HelloWorld"`, resulting in some paths becoming difficult or inaccessible. This problem has become more severe in smart contracts. Since many smart contracts include external attributes as part of the verification (such as timestamps), this leads to the fact that if the environment at this time does not meet the requirements of the contract (the date limit has passed). Then, the contract may not be able to

be entered, and the vulnerability hidden in the contract may not be detected.

B. Language Translation

Because there are no related models or tools designed for smart contract platform, both symbolic execution and formal verification require language translation to transform the unfamiliar languages into a familiar one.

For example, in symbol execution, its core model ‘SMT solver’ has many existing tools (such as Z3, SMT-ART, etc.) available, but these tools only support specific languages like C++, Java or LLVM bytecode. This means that if we want to use these tools, we must first convert the contract code written by solidity into the language corresponding to the tools. However, due to some unique features in the solidity language, converting it to another language usually requires additional manipulation of some of the instructions. For example, the invocations in solidity can be divided into three types: internal, external, and `call()`. Internal calls and external calls exist in most programming languages, but the `call()`, which can call methods in other contracts, is rare in other programming languages and requires special handling when translating. Formal verification also faces the same problem. Often, when we use model checking to guide the development of an application, we need to convert the requirements into specifications using abstract language or paradigm. After passing the detection of the model detector, we can convert the specifications into application code. When we need to verify that an application has a problem, we need to convert the code to specifications first, but at this time, the conversion may encounter many problems. Bhargavan’s team [18] wants to convert the code in solidity to F^* and also decompile bytecode in EVM to F^* then verify their equivalence. However, in the evaluation part, they also mentioned that F^* does not support many of the syntactic features in Solidity, resulting in only 46 of the 396 contracts to be translated.

In general, both the symbolic execution and the formal verification are developed well in software vulnerability mining technology. For the new platform of smart contracts, the challenge in applying this technology is how to translate Solidity into the required language fully.

C. Analysis

Whether it’s fuzzing, symbolic execution or formal verification, they end up using static analysis methods when mining vulnerabilities. In [15], for the 7 different vulnerabilities proposed, they design corresponding oracles to determine whether the contract is vulnerable when analyzing the fuzzing execution results. For example, for the gasless problem, if the gas value of the `call()` function in the contract is 0 during the analysis, the contract is considered to have a gasless problem. Here, whether the remaining number of gas of the `call()` in the contract is 0 will be considered as a constraint for detecting this vulnerability. Such constraints make the analysis simple and easy to automate, but also bring false positives and false negatives.

The highly customized static analysis method also brings limitations to scalability. The existing analysis aims at some vulnerabilities that have been studied, so these methods cannot identify other vulnerabilities that are not discovered or appear in unnoticed scenarios. Of course, a good constraint can help us cover multiple manifestations of the same vulnerability, but whether we recognize all the patterns of this vulnerability depends on the triggering logic of the vulnerability itself.

In a nutshell, how to design a constraint that captures most of the vulnerability patterns is a challenge for these static analysis methods.

VI. CONCLUSION

In this paper, we survey several vulnerabilities focused on smart contracts in Ethereum such as reentrancy attacks, gasless send and vulnerability about timestamps. Then we present their triggering logic. Also, we briefly introduce fuzzing, symbolic execution, formal verification, and language translation methods that are often used for software vulnerability detection, and overview some tools for applying these methods to smart contract vulnerability detection.

These software vulnerability detection tools excel in detecting vulnerabilities in smart contracts, but at the same time, for such a new platform, the new features in smart contracts also pose challenges to the application of these tools. For different tools, we analyze the limitations imposed by the new features of Ethereum and raise the challenges that may constrain their development.

REFERENCES

- [1] "Ethereum white paper," 2019. [Online]. Available: <https://www.ethereum.org>
- [2] "Introduction to smart contracts," 2019. [Online]. Available: <https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html>
- [3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Springer-Verlag New York, Inc., 2017, pp. 164–186.
- [4] A. Juels, A. Kosba, and E. Shi, "The ring of gyges: Investigating the future of criminal smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 283–295. [Online]. Available: <http://doi.acm.org/10.1145/2976749.2978362>
- [5] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, 2018, pp. 67–82. [Online]. Available: <http://doi.acm.org/10.1145/3243734.3243780>
- [6] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18, 2018, pp. 653–663. [Online]. Available: <http://doi.acm.org/10.1145/3274694.3274743>
- [7] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Büenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 67–82.
- [8] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. ACM, 2018, pp. 65–68.
- [9] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, July 2018, pp. 204–217.
- [10] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 270–282. [Online]. Available: <http://doi.acm.org/10.1145/2976749.2978326>
- [11] (2016) analysis of the dao exploit. [Online]. Available: <http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>
- [12] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
- [13] "King of the ether throne smart contract," 2016. [Online]. Available: <https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html>
- [14] "Units and globally available variables," 2016. [Online]. Available: <https://solidity.rtfid.io/en/develop/unitsandglobalvariables.html>
- [15] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.
- [16] 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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: <http://doi.acm.org/10.1145/2976749.2978309>
- [17] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman, "Gradual typing embedded securely in javascript," in *ACM SIGPLAN Notices*, vol. 49, no. 1. ACM, 2014, pp. 425–437.
- [18] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *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, pp. 91–96.
- [19] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium (NDSS18)*, 2018.
- [20] (2013) extensible access control markup language (xacml) xml media type. [Online]. Available: <https://tools.ietf.org/html/rfc7061>
- [21] K. L. McMillan, "Interpolants and symbolic model checking," in *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 89–90. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1763048.1763057>
- [22] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 50, 2018.
- [23] (2018) z3. [Online]. Available: <https://github.com/Z3Prover/z3>
- [24] (2015) ida. [Online]. Available: <https://www.hex-rays.com/products/ida/>
- [25] (2019) the most accurate and reliable source code converters. [Online]. Available: <https://www.tangiblesoftware.com/index.html>
- [26] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.