2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

Towards Automated Generation of Bug Benchmark for Smart Contracts

Anonymous

***Abstract*—Smart Contracts bring Ethereum transactions great convenience, meanwhile they can have potentially devastating financial consequences. To address the lack of security guarantee, we investigate state-of-art bug detection tools and conduct experiments to analyze their pros and cons. Finally, we conduct research to answer why they have false positives and a convincing bug benchmark is provided.**

***Index Terms*—Ethereum, Smart Contract, Solidity, Vulnerability, Fasle Positive**

I. Introduction

Nowadays, Block-chain and Ethereum have attracted plenty of attention from academia and industries. Though in the early stage, Block-chain and Ethereum are advocated by IT giants and reputed banking systems, as the public foresee their great potential of being applied in various industries, including financial investment, smart traffic, smart home and etc [1]. In future, Block-chain and Ethereum may transform our lifestyle and bring up revolutionary changes in daily-life, as Internet did so in recent decades.

On Ethereum, smart contracts play a major role, since they intend to digitally facilitate, verify, or enforce the negotiation or performance of a contract without third parties [2]. To make contracts first class citizens on Ethereum, the Solidity language is proposed for implementation. Although being easy to understand and master, Solidity programs could have various bugs or security issues that may lead to financial loss [3]. Hence, detecting bugs in smart contracts is important for securing the property and privacy of the users of Ethereum.

Witnessing the severity of bugs in smart contracts, researchers from academia and industry have developed various bug-detection tools [4]–[6]. However, due to the variety of possible bugs in smart contracts, it is very difficult for each tool to cover all types of bugs. Furthermore, on account of the internal detection mechanisms, some tools (e.g., rule-based) are good at detecting pattern-based bugs while other tools (e.g., testing-based) are good at bugs that are hard to identify via rules. Last, it is also unclear to the research community that how agreed when these tools are applied on the same smart contracts for bug detection. For each tool, it is also desired to understand the false positive cases.

To address the above questions and understand the state-of-the-art tools, in this paper, we aim to conduct an insightful inspecting of these tools and summarize their pros and cons. For the open-source tools that are executable, we will apply them on the same set of smart contracts and analyze the results. Furthermore, to understand the exact detection mechanisms of the available tools, we also look into the source code. Last, on the basis of detections results of the tools, our goal is to build an automatically labeled benchmark for bugs in smart contracts. Certainly, total automation might be problematic, as each tool will produce false positive cases. Hence, we also manually inspect the results to identify those false positive cases, and improve the tools by removing those cases.

To summarize, in this paper, we make two major contributions: inspecting the available state-of-the-art tools and combining the results of tools for building a bug benchmark.

II. Background

In this part, we will introduce several influential vulnerabilities on smart contracts. Subsequently, we will introduce some bug detection tools against these vulnerabilities. The features of each tool will be explicitly listed for ease of comparison.

A. *Influential Smart Contract Vulnerabilities*

Among most influential vulnerabilities on Solidity, the *Re-entrancy Attack* is the most famous one. Like JavaScript, Solidity has fall-back functions[1]. Every contract on Ethereum could have exactly one unnamed function. When there is a call to this contract and none of the functions matches the given function identifier, this function will be called. Furthermore, the fall-back function is executed whenever the contract receives ethers [7]. In other words, if attackers want to launch Re-entrancy attack, all he needs to do is defining a malicious fall-back function and calling the victim function constantly until the gas was used out or ethers were stolen. As shown in Code Block 1 and Code Block 2, the pay function in the victim contract will do simple sent verification before sending an ether. The attack function is a fall-back function, containing a function call to the victim contract. Every time when the fall-back function is called, the pay function will be triggered. Ethers will be stolen once the attacker function launches. In another case, the Re-entrancy vulnerability could perform a kind of *Denial of Service (DoS)* attack. Because gas[2] will be consumed when the victim is executed and ends until gas depleted [8]. This attack was launched when DAO (Decentralized Autonomous Organization) was raised and caused 3.6 million worth of ethers stolen in 2016 [9].

[1]A kind of function has no arguments and cannot return anything.
[2]An unit to measure executing cost.

Code Block 1. Victim Code Block

```
contract victim
{
    //victim contract
    bool sent = false;
    function pay(address add)
    {
        if (!sent){
            add.call.value(2);
            sent = true;
        }
    }
}
```

Code Block 2. Attack Code Block

```
contract victim{function pay();}

contract attacker
{
    //malicious contract
    function(){
        victim(msg.sender).pay(this);
    }
}
```

B. State-of-art Analysis Tools

Various techniques were used to analyze Solidity software, including static and dynamic methods, fuzzing tests, verification methods. Table 1 lists the tools for analyzing Ethereum software. Among these tools, MYTHRIL and MYTHX are out of maintenance, having building errors on our machine, we cannot successfully execute and inspect them. ECHIDNA uses fuzzy testing methods. It needs to manually locate test function, which is awkward facing large number of contracts, so we will not include it in our experiments. MANTICORE uses similar techniques with OYENTE. SECURIFY was once closed source but opens now. It uses reverse engineering methods to analyze softwares. These two tools both have performance issues, they spend too much time for each contract, thus we will not elaborate them in our experiment. SMARTCHECK is an exciting tool for high accuracy rate, however, closed sources make them unavailable for us to explorer their mechanisms. Therefore, we cannot include it into the detailed comparison in the following discussion. Regarding the OCTOPUS, as a program analysis tool, it can well support the generation of CFG and IR, but not support bug detection. In short, SLITHER and OYENTE will be our target.

TABLE I
STATE-OF-ART TOOLS FOR SOLIDITY ANALYSIS

Tool Name	Main Features	Open Source
MYTHRIL-CLASSIC [10]	Multiple ability for contract analysis	open source
MYTHX [11]	A saas platform service for contract analysis	open source
SLITHER [12]	Static analysis framework with detectors	open source
ECHIDNA [13]	The only fuzzing tool for Ethereum software	open source
MANTICORE [14]	Dynamic binary analysis tool with several detectors	open source
OYENTE [15]	Dynamic binary analysis tool with several detectors	open source
SECURIFY [16]	Fully automated online static analyzer for smart contracts	open source
SMARTCHECK [17]	Static analysis of Solidity source code for security vulnerabilities and best practices	closed source
OCTOPUS [18]	Reverse Analysis tool for Blockchain smart contracts	open source

III. VULNERABILITY DETECTION EXPLORATION

In this part we will mainly talk about the working processes of SLITHER and OYENTE. To introduce the process in a concise and elegant way, we will illustrate it with figures as shown in Fig.1 and Fig.2.

A. Slither Working Process

SLITHER leverages a static analyzing method to search vulnerabilities. As shown in Fig.1, it first uses Solidity compilers to compile source code into AST (Abstract Syntax Tree), and translates the AST into self-defined intermediate representations, Slith IR. As for the reason why it can make the translate so fluently, the Solidity compiler will generate well-structured syntax trees. Detailed classes of statements and operations make the tree slightly complicated but easy to analyze. Meanwhile the IR analyzer use ASTs to performs a gigantic rule-based matching then generate IRs. Moreover, it extracts highly abstract features from statements. Meanwhile, the AST will be used to produce the CFG (control flow graph). Nodes instead of blocks are generated, which contain several abstract statements and terminate with function calls. After all these process have terminated, SLITHER could begin to search vulnerabilities by detectors. For the Re-entrancy vulnerability, the detector will search if there exist value-transfer statements in nodes, if so, it will take further steps to check if operations on global variables follow specific rules, namely write-after-read.

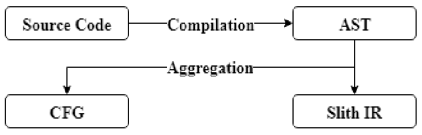

Fig. 1. The Working Process of Slither

B. Oyente Working Process

OYENTE [19] chooses symbolic execution as main method and takes the advantage of analyzing a smart contract path-by-path. The working process as shown in Fig.2, OYENTE starts with analyzing the AST, the same as SLITHER, and builds CFG by dividing blocks and analyzing function calls. When CFG is built successfully, symbolic executions implemented by Z3 [20] will check instructions path by path. If one path is finished, the core analyzer, right behind symbolic executor, will take rules to search the vulnerabilities. At last, the validator is a filter which helps reduce false positives. When detecting *Re-entrancy* vulnerability, OYENTE will firstly search the path containing global variables, then check the gas remained. If gas is more than a external call spending, OYENTE will judge it a *Re-entrancy*.

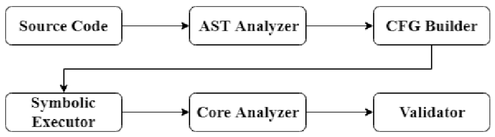

Fig. 2. The working process of Oyente

Code Block 3. An Example Of Slither's Reentrancy False Positives

```
function finish() public onlyOwner{
    require(!isFinalized); \\ global variable
        readed
    ...
    atxToken.transfer(vaultWallets[amount],
        atxToken.balanceOf(address(this)));
    \\ Transaction launched
    ...
    isFinalized = true;\\ global variable
        write
}
```

Code Block 4. An Example Of Oyente's Reentrancy False Positives

```
modifier onlyOwner(){
    require(msg.sender == owner);
    _;
}
...
function buyer(Crowdsale crowdsale) onlyOwner
    {
    uint bal = this.balance;\\ global variable
        readed
    crowdsale.buyTokens.value(bal)(investor)
        ;\\ send ethers
    state = States.Success;
}
```

IV. EXPERIMENT

A. Experiment Preparation

In our experiment, we have 1010 smart contracts in 200 files. All these files are randomly selected from our smart contract database crawled from Etherscan (https://etherscan.io/). As the hardware environment preparation, we make it by a computer with i7-4750HQ, 8GB ram. And all of our experiment are conducted in Ubuntu 18.04 LTS. We run OYENTE in the docker environment and Slither directly in Linux. Manticore is built successfully but failed when running most samples. Therefore, it will not be included.

B. Vulnerability Detection Range

We list the detection range of SLITHER and OYENTE, as shown in Table II. Instead of listing all vulnerabilities, we will only concern vulnerabilities with high impact. Apparently, huge differences exist between SLITHER and OYENTE. On one hand the ability to detect vulnerabilities is strictly decided by working method. For example, the *Integer over/underflow* could only be detected by dynamic methods, as *Unintialized state/storage* could only be detected by static methods. On the other hand, the definition of vulnerabilities varies among tools, e.g. *Suicidal* in SLITHER and *Money concurrency* in OYENTE.

TABLE II
DETECTION RANGE

Detector Name	SLITHER Support	OYENTE Support	MANTICORE Support
Reentrancy	Yes	Yes	Yes
Indistinct declaration	Yes		
Public Self-destruct	Yes	Yes	Yes
Uninitialized state/storage	Yes		Yes
Delegatecall used	Yes		Yes
Locked ether	Yes		
Constant function	Yes		
Tx origin	Yes		Yes
Callstack fullfilled		Yes	
Integer over/underflow		Yes	Yes
Unresolved assertion		Yes	
Unused return	Yes		Yes

C. Experiment Results

Through our experiments, we aim at answering two questions (Qs) categorized into two topics, namely effectiveness (Q1) and false positives (Q2).

Q1. What is the effectiveness of SLITHER and OYENTE on our dataset?

Q2. What causes false positives?

In Q1, we list SLITHER and OYENTE's detection results on our dataset, and a breifly comparison is given. In Q2, we manually check their detection result and compare their false positive rate, then we will analyze what cause false positives on detailed examples.

1) Answer to Q1: We performed OYENTE and SLITHER in our well-prepared dataset. The result of our vulnerability detection is shown in Table III. The numbers in this table is the vulnerable contracts these two tools reported. Clearly, the *Re-entrancy* was both supported by SLITHER and OYENTE, but the number of reported varies. As we've discussed, the different methods result in huge diverges in detection result. SLITHER detects 24 files contain *Re-entrancy* bug, while OYENTE only detects 4 files, which means the two tools has little intersection in bug detection results. Besides, SLITHER detects more *Re-entrancy* bug than OYENTE, however, it's too early to judge whose behavior is better. Further false positive check is needed. What's more, OYENTE detect 546 *Integer over/underflow* bug. Almost every file is suspicious in OYENTE's rules. Consequently, this rule will lose abilities to detect bugs as it has a high false positive rate.

TABLE III
DETECTION RESULT

Detector Name	SLITHER Detected	OYENTE Detected
Re-entrancy	33	4
Indistinct declaration	11	Unsupported
Public Self-destruct	1	0
Uninitialized state/storage	113	Unsupported
Delegatecall used	0	Unsupported
Locked ether	5	Unsupported
Constant function	4	Unsupported
Tx origin	0	Unsupported
Callstack fullfilled	Unsupported	11
Integer over/underflow	Unsupported	546
Unresolved assertion	Unsupported	0

2) Answer to Q2: We list their false positive rates in Table IV. Some lines in Table III disappear because these vulnerabilities were not reported by both tools (*Tx origin*, *Delegatecall used*, *Unresolved assertion*). Some vulnerabilities are reported too many times to manually check (*Uninitialized state/storage*, *Integer over/underflow*). Meanwhile the *Callstack fullfilled* bug could only be analyzed through dynamic methods, so we will not check their false positives. Apparently, SLITHER has high false positive rate when detecting *Re-entrancy* vulnerability. For other items with high FP rates, like *Public Self-destruct*, *Locked ether* and *Constant function*, as the lack of data quantity, we will not include them in discussion. Let's talk about one of the false positives judged by SLITHER's Re-entrancy detector, as show in Code Block 3. In this function, the global variable *isFinalized* was readed in the head, and there's a transaction launched, finally the global variable was wrote. We take it as a representative of false positives because it reveals the drawback of SLITHER's detection rules. In SLITHER's rule, if a global variable was read, then transaction happens and the global variable was wrote, it is a *Re-entrancy*. Such inflexible rule works not well when global variables have nothing to do with transactions.

While OYENTE performs not sufficiently well, it fails to judge every case it found. As we have talked above, when detecting Re-entrancy vulnerability, OYENTE will firstly search the path containing global variables, then check the gas remained. If gas is over than a external call spending, OYENTE will judge it a *Re-entrancy* bug. As shown in Code Block 4, this function firstly calls global variable to check balances, then send ethers to the target. In this case, this function has a self-defined modifier onlyOwner, which guarantees this function could only be executed by the owner of this function. However, the function modifier is ignored in the working process of OYENTE, which results in a false positive. Besides, the same thing happens in other three false postitives. The lackness of modifier check makes results of OYENTE untrusted.

V. CONCLUSION

Many smart contracts in Ethereum lack of security guarantees. Some of them revealed then caused huge economy losses, while others still hide in mist. These contracts control gigantic balances worth billion USDs. To investigate these contracts, we surveyed some state-of-art tools and discussed their features. We identified that the mechanism behind tools remains imporved. Meanwhile, larger amounts of dataset is expected for a precise evaluation among methods. We hope our benchmark will be useful for Ethereum community users in offering a criterion for evaluating security tools.

TABLE IV
VALIDATION RESULT

Detector Name	SLITHER FP Count	OYENTE FP Count
Reentrancy	51.5% (17 of 33)	100% (4 of 4)
Indistinct declaration	18.2% (2 of 11)	Unsupported
Public Self-Destruct	100.0% (1 of 1)	None
Uninitialized state/storage	Unable to Check	Unsupported
Locked ether	60.0% (3 of 5)	Unsupported
Constant function	25.0% (1 of 4)	Unsupported
Callstack fullfilled	Unsupported	Unable to Check
Integer over/underflow	Unsupported	Unable to Check

REFERENCES

[1] I. Makhdoom, M. Abolhasan, H. Abbas, and W. Ni, "Blockchain's adoption in iot: The challenges, and a way forward," *J. Network and Computer Applications*, vol. 125, pp. 251–279, 2019.

[2] ethereum, "Whitepaper," github, https://github.com/ethereum/wiki/wiki/White-Paper/f18902f4e7fb21dc92b37e8a0963eec4b3f4793a.

[3] G. Greenspan, "Smart contracts and the dao implosion," website, https://www.multichain.com/blog/2016/06/smart-contracts-the-dao-implosion/.

[4] *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society, 2018.

[5] M. Huchard, C. Kästner, and G. Fraser, Eds., *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.* ACM, 2018.

[6] D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* ACM, 2018.

[7] ethereum, "Solidity document," Website, https://solidity.readthedocs.io/en/v0.4.24/contracts.html?highlight=fallback.

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," *IACR Cryptology ePrint Archive*, vol. 2016, p. 1007, 2016.

[9] David Siegel, "Understanding the DAO Attack," Website, https://www.coindesk.com/understanding-dao-hack-journalists.

[10] ConsenSys, "Mythril," github, https://github.com/ConsenSys/mythril-classic.

[11] ——, "Mythx," website, https://mythx.io/.

[12] trailofbits, "Slither," github, https://github.com/trailofbits/slither.

[13] ——, "Echidna," github, https://github.com/trailofbits/echidna.

[14] ——, "Manticore," github, https://github.com/trailofbits/manticore.

[15] melonproject, "Oyente," github, https://github.com/melonproject/oyente.

[16] ChainSecurity, "Securify," website, https://securify.chainsecurity.com/.

[17] ——, "Smart check," website, https://tool.smartdec.net/.

[18] quoscient, "Octopus," github, https://github.com/quoscient/octopus.

[19] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," *IACR Cryptology ePrint Archive*, vol. 2016, p. 633, 2016.

[20] C. R. Ramakrishnan and J. Rehof, Eds., *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008.