2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

Finding Concurrency Exploits on Smart Contracts

Yue Li
Peking University
liyue_cs@pku.edu.cn

***Abstract*—Smart contracts have been widely used on Ethereum to enable business services across various application domains. However, they are prone to different forms of security attacks due to the dynamic and non-deterministic blockchain runtime environment. In this work, we highlighted a general miner-side type of exploit, called *concurrency exploit*, which attacks smart contracts via generating malicious transaction sequences. Moreover, we designed a systematic algorithm to automatically detect such exploits. In our preliminary evaluation, our approach managed to identify real vulnerabilities that cannot be detected by other tools in the literature.**

I. Problem And Motivation

Ethereum smart contracts are programs running on the blockchain to operate on Ethereum cryptocurrencies (*i.e.*, *ether*) and a set of persistent data (*i.e.*, *storage*) [1]. Figure 1 shows a simple smart contract `EtherStore` with four storage variables and public functions except the constructor. External accounts can use `EtherStore` by sending transactions to it. For example, a possible transaction data may look like `a9059cbb7adee867ea91533879d083dd4ea81f0eee3a37e`. Specifically, the first four bytes `a9059cbb` point to the hash of the `foo` function and the remaining bytes indicate the parameter used by `foo`, *i.e.*, a 20-byte address. In the execution of `foo`, the storage `Owner` will be updated to the value passed from the transaction (line 11) and permanently stored on blockchain. Commonly, transactions submitted to the Ethereum network are ordered and packaged by a *miner* node to optimize its own benefit, *e.g.*, efficiently get the most transaction fee. From this perspective, smart contracts can be seen as a form of *concurrent software* since their executions are non-deterministic to transaction senders.

We further explain such concurrency via Figure 2 based on the contract in Figure 1. Particularly, we describe four cases in this context to explain the transaction ordering and storage state transitions accordingly. In 2a, `Alice` first submitted a transaction to call the `foo` function, which sets the storage `Owner` to `Alice`. The following two transactions are from `Alice` (call `bar`) and `Bob` (call `foo`). If `Bob's` transaction is ordered before `Alice's` as in 2a, the call to `bar` will lead to a money transfer to `Bob` (line 17) because `Owner` has been updated to `Bob`. Otherwise, the money goes to `Alice`. Similarly in 2c, when `Alice` acquired the `Owner`, the order of calls to `foo` and `upgrade` from `Bob` will cause the sanity check at line 20 to fail or not. If the check is passed, `Bob` would be able to further invoke a delegated call (line 22), *i.e.*, use a delegate contract (`currentVersion` in this case) to process the transaction input instead of the current contract. In Ethereum, the delegate contract is allowed

```
1   contract EtherStore{
2     address Owner;
3     address Creator;
4     mapping (address => uint256) balances;
5     uint price;
6     constructor (){
7       Owner = msg.sender;
8       Creator = msg.sender;
9     }
10    function foo(address newOwner){
11      Owner = newOwner;
12    }
13    function bar(){
14      price = msg.value;
15      balances[msg.sender]+=price;
16      // tranfer ether to owner
17      Owner.call.value(price)();
18    }
19    function upgrade(address currentVersion) {
20      require(msg.sender == Owner);
21      // contrat currentversion code injection
22      if(!currentVersion.delegatecall(msg.data
            )) revert();
23    }
24    function destruct(){
25      require(msg.sender == Creator)
26      // contract termination
27      selfdestruct(Creator);
28  }   }
```

Fig. 1: A motivating example of smart contract

to directly manipulate storage of the current contract, *e.g.*, `Creator` variable. In such cases, transactions of `upgrade` and `destruct` will produce uncertain scenarios of contract destruction as in 2b, since `Creator` is used as a receiver of the remaining balance after the contract is cleared by the `selfdestruct` API at line 27. The last case of 2d describes two transactions of `bar` from both `Alice` and `Bob`, where different transaction orders will generate the same final storage values of `balances` no matter which one is executed first.

While such concurrency enables normal miners to flexibly make profits, it introduces great security threats to Ethereum. In the aforementioned cases, only the last one is considered secure since the concurrent execution of transactions can commute, *e.g.*, different transaction orders will yield the same state transition. The other three cases are all viewed as potentially malicious, because attackers can employ specific sequence of transactions to make undesired benefits, *e.g.*, transfer money to themselves, pollute the contract storage *etc.* In this work, we refer such attacks as *concurrency exploits* and highlight an informal definition below. Given two accesses p and q from two transactions t_p and t_q on storage s_p and s_q respectively, t_p and t_q can be exploited by a concurrency attack if

- p and q include exact one write and one read operation.
- s_p and s_q point to the same storage index.
- p and q are not mutually exclusive.
- At least one security-critical operation is dependent on s_p or s_q (explained later).

Our work aims to detect *concurrency exploits* in smart

2574-1934/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-Companion.2019.00061

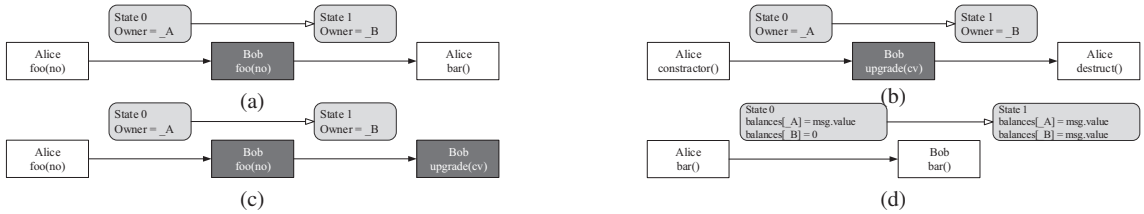

Fig. 2: Possible transaction sequences and state transitions

contract [2]. Oyente identified transaction ordering bugs(TOD) by checking if two different traces have different Ether flows, which is the sub-class of *concurrency exploits* [3] [4]. Securify share the same oracle of TOD with oyente [5]. At technical level, many techniques are used for analyzing security of smart contract: formal verification [6], symbolic execution [7] [3] [5] [8] and fuzzing [9].

II. Approach

Smart contract is compiled to bytecode that is executable on the Ethereum virtual machine (EVM) [1]. We proposed a systematic technique to automatically detect concurrency exploits from EVM bytecode. Before we describe the details, we first introduce *security-critical* operations as mentioned in §I. Specifically, we focus on four EVM instructions, as below.

- `CALL` that allows a contract to directly transfer ether to a given address [10].
- `CALLCODE` / `DELEGATECALL` that enables another contract to process the current transaction [10].
- `SELFDESTRUCT` that terminates a contract, and transfers all remaining ether to the specified account [10].

Furthermore, as defined above, a security-critical operation is *dependent* on a storage, if any of the following two conditions is satisfied.

- The storage as part of the parameters of security-critical operations. Such as the concurrency exploits in Figure 2a, `Owner` is the receiver of `CALL` instruction.
- The storage as part of security-critical operations execution path condition constraint. Such as the concurrency exploits in Figure 2b, the value of `Owner` is a condition for `DELEGATECALL` 's execution.

Our technique using symbolic execution explores branches and leverages program paths to identify concurrency exploits from EVM bytecode (smart contract). The detection flow mainly contains three steps:

Step 1: Marking Storage Operation. Accessing storage variable is a major source of concurrency exploits. Our work using symbolic execution traces two EVM instruction to operate storage: `SLOAD` loads word from storage and `SSTORE` saves a word to storage. Step 1 marks operation of storage to construct access flow graph(AFG).

Step 2: Merging Access Flow. Since an AFG contains sequential relations of storage operation. Step 2 aims to classify in-sensitive operation in AFG. Consider `foo` function(branch) in Figure 1, it firstly writes(`SSTORE`) `price`(line 14) then loads(`SLOAD`) `price` to use(line 15,17). Since the value of price is determined while loading and it is not affected by concurrency, Step 3 removes `SLOAD` marking in line 15,17. Furthermore, if there are multiple writing to a storage index in a branch, only the marking of last writing is retained.

Step 3: Verifying Concurrency Exploits. Step 3 obtains pairs of concurrent operations on storage from Step 2 to check concurrency exploits based on our definition in §I. Consider concurrency exploits in Figure 2a, the `CALL` instruction is dependent on `Owner`. Step 3 checks the pair of operations on `Owner`(line 11, line 17 in Figure 1) that are not mutually exclusive by solving the path condition constraints of them.

TABLE I: Comparison of Concurrency Exploits Detection

Contract Address	Type	CESC	Oyente	Securify
0x352dbBA201aF66f98a47F2b280bFf45f9050DBf8	●	2	2	1
0x4ceae42d5fb3bd6956b2463fdd4a2209382d722c	●	1	0	1
0x33b44a1D150F3feAA40503aD20a75634Adc39B18	○	1	1	0
0xaf71b19e6292c6e1491ff3a54b3e63dbd41ef023	○	1	0	0
0xfa82f0a05b732deaf9ae17a945c65921c28b16dd	◐	1	0	0
0x5aef06ec39e98c05201ee1e54b653c372ecb9cf3	●	2	0	2
0x43efc486d1c7c5cb0193e409a73aa33786f5197c	●	2	0	2
0xd43cbd8a74535327a8a196ea36cd44fc799ca289	○	1	0	0
0x7b47e1473F97040689812204113FF549b2E2C31B	None	0	2	4

III. Preliminary Evaluation

To evaluate our algorithm, we apply it to verify real-world smart contracts from etherscan [11]. And we compare our technique(marked as CESC) with Oyente [3] and Securify [5]. While Oyente and Securify allow checking on a set of bug patterns, we focused on transaction-ordering dependency bug (TOD) in this setting. The results are shown in Table I. The first column is contract address in Ethereum [11]. The concurrency exploits on `CALL` instructions marked as ●. The concurrency exploits on `DELEGATECALL`/`CALLCODE` instructions marked as ◐. The concurrency exploits on `SELFDESTRUCT` instructions marked as ○. We marked false positives as gray.

By comparing with our approach in detecting concurrency exploits, our approach highly improve the scope of detection for potentially concurrent attacks and reduce false positives.

IV. Contribution

We summarize our main contributions below.

- We highlighted *concurrency exploit* as a new and general form of security attacks on Ethereum smart contracts.
- We proposed a systematic detection technique to identify *concurrency exploits* based on symbolic execution.
- We conducted a preliminary evaluation on real contracts deployed on Ethereum.

REFERENCES

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[2] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 478–493.

[3] 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*. ACM, 2016, pp. 254–269.

[4] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 79–94.

[5] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," *arXiv preprint arXiv:1806.01143*, 2018.

[6] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Beguelin, "Formal verification of smart contracts," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS'16*, 2016, pp. 91–96.

[7] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." NDSS, 2018.

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

[9] 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.

[10] Ethereum, "Solidity — solidity 0.4.19 documentation," 2017. [Online]. Available: https://solidity.readthedocs.io/en/develop/

[11] E. Team, "Etherscan: The ethereum block explorer," 2017.