Clairvoyance:Enable Inter-contract Static Analysis for Reentrancy Detection in Smart Contracts

ABSTRACT

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the “acmart” document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
. 2018. Clairvoyance:Enable Inter-contract Static Analysis for Reentrancy Detection in Smart Contracts. In *Proceedings of ACM Conference (Conference’17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The popularity of smart contract and Ethereum, and explain the terms

The severity of smart contract security

Brief introduce the existing security scanners, their limitation

Highlight the root of the problem, lacking an accurate yet scalable inter-contract static analysis tool

Brief introduce what such scalable inter-contract static analysis tool brings, the benefits

Brief introduce the techniques and workflow of our tool

Brief introduce the evaluation results

List the contribution items

2 BACKGROUND

In this section, we briefly introduce the property of the Solidity programming language on Ethereum. Based on that, we introduce the security issues of Solidity and the reentrancy vulnerabilities in smart contracts.

Conference’17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

2.1 Solidity Properties

On Ethereum, two types of accounts are supported, including External Owned Accounts (EOA) and Contracts Accounts (CA). Only CAs are allowed to own executable code (i.e., smart contracts code), while EOAs are only operates with GUIs. Most of smart contracts are written in Solidity, a JavaScript-like language in syntax. Solidity is an contract-oriented, high-level language that governs the behaviors of accounts within the Ethereum state. Solidity is statically typed, supporting inheritance, libraries and complex user-defined types among other features [10]. Solidity is Turing-complete and includes four types of elements: contract (similar to class in OOP), variable, function and event. Note that event is used for binding the event handler function, similarly to that in JavaScript. However, solidity differs from the standard OOP — the class can be instantiated in OOP but the contract cannot in Solidity. The variables and functions are accessed and operated directly on the corresponding contract that is bound to a public Ethereum address.

Solidity has its own special feature for smart contracts:

(1) due to the EVM restriction, each instruction compiled from Solidity programs exhausts some gas. The transaction behind Solidity code will fail if gas owned by the corresponding CA is insufficient. For example, maximum gas consumption is up to or limited by 2300 for the built-in functions `transfer()` and `send()`, but the consumption is uncertain and dependent on an external function call for the low-level `add.call.value()` in Fig.1. Owing to the gap limitation, deadlock cases can be avoided.
(2) Solidity has some built-in key words, such as `msg.sender` and address to support communication between two Ethereum accounts. For example, `msg.sender` in Fig. 1 refers to the contract that starts the transaction. Interested readers can refer to the following link [9] for more details.
(3) Solidity support a special type of function — the fallback function, an unnamed function, which have neither arguments nor function return. A fallback function is executed if its contract is called and no other function matches a specified function identifier, or if no data is supplied.

2.2 Reentrancy Vulnerability

introduce the Reentrancy, and explain the example of Fig1a.

Though no vulnerabilities of memory corruption are found in Solidity programs, it cannot be claimed as a secured programming language. Shortly after the advent of smart contracts back in 2016, Atzei et al. [3] have reported six types of vulnerabilities that may affect the security of blockchain. Recently, a public technical blog [2] provides a more detailed taxonomy for vulnerabilities in Solidity code, real cases, and the preventative techniques for these vulnerabilities. Based on the surveys or reports [4, 8], reentrancy is undoubtedly considered as one of most severe and hard-to-detect vulnerability in smart contracts. Notably, in this paper, we focus

```
1  contract Victim {
2    mapping(address => uint256) public balances;
3    function deposit() {
4      balances[msg.sender] += msg.value;
5    }
6    function withdraw(uint256 _amount) public {
7      require(balances[msg.sender] >= _amount);
8      require(msg.sender.call.value(_amount)());
9      balances[msg.sender] -= _amount;
10   }
11 }
```

(a) CB1: A intra-function Reentrancy, detected by three tools

```
1  contract Victim {
2    mapping(address => uint256) public balances;
3    function deposit() {
4      balances[msg.sender] += msg.value;
5    }
6    function withdraw(uint256 _amount) public {
7      require(balances[msg.sender] >= _amount);
8      transEth(_amount);
9      balances[msg.sender] -= _amount;
10   }
11   function transEth(uint256 _amount) private {
12     require(msg.sender.call.value(_amount)());
13   }
14 }
```

(b) CB2: A inter-function Reentrancy, detected by Slither and Securify

```
1  contract Victim {
2    EFT eft = new EFT();
3    mapping(address => uint256) public balances;
4    function deposit() {
5      balances[msg.sender] += msg.value;
6    }
7    function withdraw(uint256 _amount) public {
8      require(balances[msg.sender] >= _amount);
9      eft.transEth(msg.sender, _amount);
10     balances[msg.sender] -= _amount;
11   }
12 }
13 contract EFT {
14   modifier onlyPartner() {
15   require(_isPartner(msg.sender) "only partner have
         permission")_;
16   }
17   ...
18   function transEth(address addr, uint256 _amount)
         onlyPartner {
19     addr.call.value(_amount)();
20   }
21   ...
22 }
```

(c) CB3: A inter-contract Reentrancy, not accurately detected by all three tools

Figure 1: Detection results of Slither, Oyente and Securify for reentrancy cases from simple to complicated forms

on the reentrancy vulnerabilities, which cannot be accurately and efficiently by existing analysis tools.

In Fig. 1a, we show a basic case of reentrancy, where all of the elements to meet the condition of a reentrancy are inside a function. According to Slither's rule, a reentrancy may happen if it satisfies the following condition:

$$\boxed{r(var_g) \vee w(var_g) > externCall > w(var_g) \Rightarrow \text{reentrancy} \qquad (1)}$$

where $r()$ and $w()$ denotes the write and read operation, respectively; var_g denotes a certain global variable; $>$ denotes the execution time order based on the program's control flow; *externCall* denotes the external call to the *money-transfer* functions except `send()`, `transfer()`. The intuitive understanding of this rule is that: if there exists an external call to a money-transfer function and the call is between two write operations to the same global variable, the reentrancy could happen.

In Fig. 1a, in function `withdraw()` of the contract `Victim`, Slither can find all the above three elements, i.e., a read access to a state variable (at line 7), then a call to external address via the low level call (at line 8), and a write access to the state variable (at line 9). Hence, CB1 is clearly a valid reentrancy case that can could be handled by Slither, Oyente and Securify, and the attacker can trigger it via the following call chain[1]:

$$msg.sender.attack \rightarrow victim.withdraw \rightarrow$$
$$msg.sender.fallback \rightarrow victim.withdraw \rightarrow ...$$

[1]Due to page limit, we show the detailed attack function and the fallback function that calls the attack function on our tool website.

3 STUDY OF THE EXISTING TOOLS

In this section, we summarize the weakness of the existing tools, the FPs and FNs of rules inside them. Beyond that, we propose the possible improvements towards the desirable static analysis.

3.1 Lack of Inter-contract Analysis in Tools

In this subsection, we show an examples to illustrate various forms of the real-world reentrancy vulnerabilities and how the existing tools handle them.

In Fig. 1b, all the three elements for Slither to judge a reentrancy are not inside the same function any more. Instead, the element of a call to external address via the low level call (line 8 in Fig. 1a) is hidden into another function `transEth` (line 12 in Fig. 1b). However, to detect such a form of reentrancy, the inter-function analysis is required before applying the above Rule 1, regardless of the soundness and completeness of Rule 1 itself. Interestingly, the latest version of Slither supports the inter-function analysis and could successfully detect this form, and Securify can also detect this. The attacker can trigger it via the following call chain:

$$msg.sender.attack \rightarrow victim.withdraw \rightarrow victim.transEth \rightarrow$$
$$msg.sender.fallback \rightarrow victim.withdraw \rightarrow ...$$

In Fig. 1c, the form become more complicated, as the element of a call to external address via the low level call (line 8 in Fig. 1a) is hidden into function `transEth` of another Contract `EFT` (line 19 in Fig. 1c). Hence, to detect such a form, the inter-contract analysis is required. As the inter-contract analysis could be computationally costly (accessing other contracts is quite common in Solidity programs), most of existing static analysis tools are not designed or aimed to support this. Hence, the existing tools (e.g., Slither and Securify) at most conduct inter-function analysis, resulting in either FPs or FNs . Slither reports warns due to the use of low-level

```
1  contract YumeriumManager {
2      function getYumerium(address sender) external payable
           returns (uint256);
3  }
4  contract Sale {
5      address public creator;
6      YumeriumManager public manager;
7      constructor(address _manager_address) public {
8          manager = YumeriumManager(_manager_address);
9          creator = msg.sender;
10     }
11     function buy(uint amount) public {
12         require(amount < address(this).balance);
13         uint256 amount = manager.getYumerium.value(amount
               )(msg.sender);
14         address(this).balance -= amount;
15     }
16     function changeManagerAddress(address
            _manager_address) external {
17         require(msg.sender==creator, "You are not a
               creator!");
18         manager = YumeriumManager(_manager_address);
19     }
20 }
```

Figure 2: CB4: a real case of using constant value for the account address — a FP for SLITHER and SECURIFY.

calls, but not bugs for reentrancy. SECURIFY reports as a benign reentrancy which unfortunately is a true reentrancy..

$$msg.sender.attack \longrightarrow victim.withdraw \longrightarrow EFT.transEth \longrightarrow$$
$$msg.sender.fallback \longrightarrow victim.withdraw \longrightarrow \ldots$$

To summarize, the static analysis for smart contract needs to support inter-function and inter-contract analysis for the complicated reentrancy vulnerabilities. Besides, none of these tools is able to report a possible inter-function and inter-contract call chain that leads to reentrancy, meanwhile effectively tackling the path exploration problem.

3.2 Empirical Study of Typical FPs and FNs for Existing Rules

Even for the intra-contract analysis in existing tools, their detection rules are neither sound nor complete, due to the inconsideration of possible DMs. In our preliminary study, we run the three state-of-the-art tools, i.e., SLITHER [5], OYENTE [6] and SECURIFY [11], on 10000 frequently-used real-world contracts from the well-known third-party website ETHERSCAN for contract indexing and browser [1]. For the detection results, we recruit 4 researchers to spend around 1.5 months in reviewing the results, and summarize the patterns of FPs and FNs for rules in these tools.

FPs of Reentrancy for Existing Rules. As the reentrancy caused some significant losses in the past [12], the real contracts on Ethereum have already adopted some DMs to prevent from the actual invocation of reentrancy. We summarize the three main categories of DMs: **DM1**, hard-coding the constant value for the address of payee or payer; **DM2**, adding the self-predefined modifier in the function declaration; **DM3**, using `if` lock(s) to prevent reentrancy. However, these DMs are seldom discussed in relevant studies or considered in existing scanners. Hence, the ignorance about possible DMs will result in a high FP rate during detection.

Fig.. 2 gives a false alarm (CB4) reported by Slither based on its Rule 1 that fails to recognize DM1. The code firstly reads the state variable `address(this).balance`; then calls an function of the

```
1  interface HgInterface {
2      function buy(address _player) payable external
           returns(uint256);
3  }
4  contract Richer3D {
5      ...
6      mapping(uint256=>DataModal.RoundInfo) rID;
7      HgInterface constant p3d = HgInterface(0
           xB3775fB83F7D12A36E0475aBdD1FCA35c091efBe);
8      function calculateTarget() public {
9          if(increaseBalance >= targetBalance) {
10             if(increaseBalance > 0) {
11                 p3d.buy.value(ethForP3D)(p3dAddress);
12             }
13         }
14         ...
15         rID[rNumber].lastTime = _timestamp;
16         ...
17     }
18 }
```

Figure 3: CB5: another real case of using constant value for the account address — a FP for OYENTE and SECURIFY.

```
contract RTB2 {
    modifier onlyHuman() {
        address _addr = msg.sender;
        uint256 _codeLength;
        assembly {_codeLength := extcodesize(_addr)}
        require(_codeLength == 0, "sorry humans only");
        _;
    }
    function buy(uint256 _amount) external onlyHuman
         payable{
        require(balances[msg.sender] >= _amount);
        require(msg.sender.call.value(_amount)());
        balances[msg.sender] -= _amount;
    }
}
```

Figure 4: CB6: a real case of using self-defined modifier for protection — a FP for SLITHER, OYENTE and SECURIFY.

contract of an external address via `***.value(amount)(msg.sender)`; last, writes to the state variable `address(this).balance`. However, in reality, reentrancy will never be triggered by external attackers due to the require check `msg.sender==creator` at line 17 in Fig. 2. Similarly, in Fig. 3, we show an FP reported by OYENTE, according to its run-time detection rule below[2]:

$$\boxed{\begin{aligned}&(r(var_g) \wedge (gas_{trans} > 2300) \wedge (amt_{bal} > amt_{trans}) \wedge \\ &\quad var_g \text{ is changad before external call}) \Rightarrow \text{reentrancy}\end{aligned} \quad (2)}$$

where $r(var_g)$ means read operation(s) to a public variable, $gas_{trans} > 2300$ means the gas for transaction must be larger than 2300, $amt_{bal} > amt_{trans}$ means the balance amount must be larger than transfer amount, and lastly the global variable could be changed before external calls. Although CB5 in Fig. 3 satisfies all the four conditions, it actually could not be triggered by external attackers — the only hard-coded address (line 7 in CB5) is allowed in the transaction.

Fig. 4 gives another false alarm example that reported by the existing tools that ignore DM2. CB6 actually takes into account the security issue and adds the self-defined modifier `onlyHuman()` before the possibly vulnerable function `buy`. Since `onlyHuman()` restricts that the transaction can be only done by the `admin` or `owner` role, otherwise the transactions will be reverted. In such a way, `buy` could not be recursively called by external attackers.

[2]We summarize this rule from the implementation of OYENTE.

Different from the above two DMs on permission control to prevent external malicious calls, the last DM is to prevent the recursive entrance for the function—eliminating the issue from root. For instance, in Fig. 5, the internal instance variable `reEntered` will be checked at line 5 before processing the business logic between line 8 and 10 of CB7. To prevent the reentering due to calling `ZTHTKN.buyAndSetDivPercentage.value`, `reEntered` will switch to `true`; after the transaction is done, it will be reverted to `false` to allow other transactions.

Update the data for 10000 contracts and also add the data for Securify

Totally, 457 FP cases are manually identified for Slither. Among them, 216 FPs are attributed for the first DM, 76 FPs for the second, 47 FPs for the third, and only 1 FP for the forth DM and 117 for other causes. In contrast, Oyente has fewer FP cases (only 53 in total), among which the distribution for the four caused DMs is 5, 5, 22 and 0. **TODO:** Add the data for Securify. Details on causes other than the 4 DMs can be found from our tool website [7].

Typical FNs of Reentrancy for Existing Rules. CB8 in Fig. 6 is missed by Slither, as there is no read/write operation to global variable(s) before the external call. Being no better, Oyente does not report this CB, as it fails to find the balance check according to Rule 2. **TODO:** Add the reason for Securify. Such missed reentrancy cases share the same root cause of reentrancy — allowing function calls from external address, e.g., reentering function `buyTokens()` by taking advantage of the external address `_add` at line 5 in Fig. 6.

The attack logic is the man-in-the-middle attack, which exploits the weakness of unique identification of a smart contract — in Solidity, the uniqueness of a contract is identified by the 20 bytes address, not by the contract name. Hence, if no security check is on the contract-address binding `TokenOnSale(_add)` at line 5, malicious attackers can forge a faked contract `TokenOnSale` via an malicious address and make function `TokenSale.buyTokens()` reenter-able from function `mint()` of the faked `TokenOnSale`. Note that `TokenOnSale.mint` of the external malicious address needs to trigger the fallback function on the faked attack contract `TokenOnSale`. In the following call chain, `msg.sender` refers to the faked contract `TokenOnSale` that starts the transaction.

$$msg.sender.attack \rightarrow TokenSale.buyTokens \rightarrow$$
$$TokenOnSale.mint \rightarrow msg.sender.fallback \rightarrow$$
$$TokenSale.buyTokens \rightarrow ...$$

3.3 Summarized Challenges and Insights

According to the examples in §3.1 and §3.2, we can summarize the following challenges for the desirable reentrancy analysis:

C1. Reentrancy can be so complicated that the inter-contract analysis is required. However, inter-contract analysis can be computationally costly, and an efficient solution is desired.

C2. Due to the properties of Solidity (e.g., fallback function, self-defined modifier and contract-address binding, etc.), DMs could cause FPs and the accurate analysis should consider them.

C3. Owing to the weakness in contract identification, calling unchecked external contracts via name could be problematic and cause FNs. However, the analysis needs an algorithm to trace an external contract in all control flow paths and identify one path that accesses it before checking it.

```
1  contract ZethrBankroll is ERC223Receiving {
2    ZTHInterface public ZTHTKN;
3    bool internal reEntered;
4    function receiveDividends() public payable {
5      if (!reEntered) {
6        ...
7        if (ActualBalance > 0.01 ether) {
8          reEntered = true;
9          ZTHTKN.buyAndSetDivPercentage.value(ActualBalance
               )(address(0x0), 33, "");
10         reEntered = false;  }
11     }
12   }
13 }
14 contract ZTHInterface {
15   function buyAndSetDivPercentage(address _referredBy,
         uint8 _divChoice, string providedUnhashedPass)
         public payable returns (uint);
16 }
```

Figure 5: CB7: a real case of using an `if` lock for reentrancy protection — a FP for Slither and Securify.

```
1  contract Victim {
2      TokenSale tokenSale = new TokenSale();
3      function combination() {
4          tokenSale.buyTokensWithWei();
5          tokenSale.buyTokens();
6      }
7  }
8  contract TokenSale {
9      TokenOnSale tokenOnSale;
10     ...
11     function set(address _add) {
12         tokenOnSale = TokenOnSale(_add);
13     }
14     function buyTokens(address beneficiary) {
15         if (starAllocationToTokenSale > 0) {
16          tokenOnSale.mint(beneficiary, tokens);
17        }
18     }
19     function buyTokensWithWei() onlyPartner {
20         wallet.transfer(weiAmount);
21     }
22 }
```

Figure 6: CB8: a real case susceptible to the faked external contract attack — a FN for Slither, Oyente and Securify.

To address these challenges, we propose an scalable static approach of inter-contract analysis that adopts taint analysis for effectively tracing the contract objects and addresses on all possible control flow paths. Our approach can build complete CFG via supporting the modifier analysis, report a shortest call chain that leads to reentrancy.

4 SYSTEM OVERVIEW

In this section, we briefly introduce the workflow of our inter-contract detection approach. The input just includes the source code files of smart contracts to be scanned, and the output of the detection reports two things: which contracts have reentrancy vulnerability and what call chain in the source code will constitutes a reentrancy. Notably, most of the existing tools of static analysis will not provide the information on the call chain leading to a reentrancy.

As shown in Fig. 7, our approach is composed of five steps:

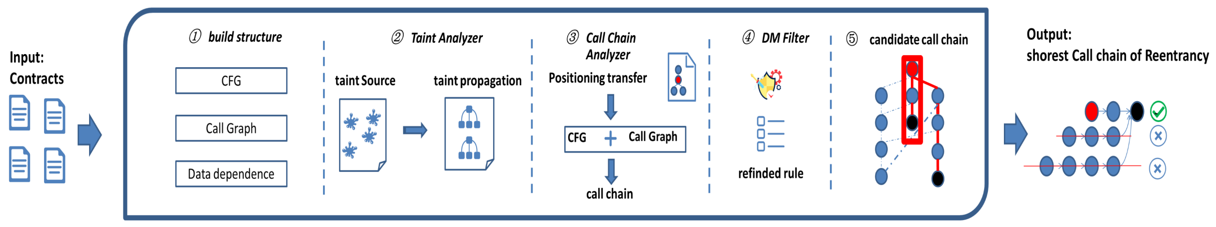

Figure 7: System diagram

S1. Based on CFG and call graph, we need to generate the ICFG for the source code of any given smart contract, which facilitates the following call chain analysis.

S2. Given the special features of Solidity language, we need to define taint source and taint propagation rules for a accurate taint analyzer.

S3. Given an ICFG and the taint analyzer, we need to collect all the possible call chains that potentially lead to a reentrancy attack. To improve recall, the collector should be complete yet efficient for finding call chains in various forms.

S4. Based on the DM summarized in 3.2, we need to remove the FPs from the collected call chains. To improve precision, various DMs are encoded as different filtering patterns on the call chain.

S5. Given the remaining call chain after DM filtering, we still need to rank them and select one from them as the *desired* call chain that most likely leads to an reentrancy.

Clearly, S1 and S3 are designed to address the C1 in §3.3 for the sound and complete call chain analysis. S2 and S4 are designed for the purpose of accurate taint analysis on Solidity language that address the C2 in §3.3. Last, S5 is to address the C3 in §3.3.

5 APPROACH

5.1 ICFG Generator

5.2 Taint Analyzer

5.3 Call Chain Collector

5.4 Filter based on DM

5.5 Call Chain Selector

6 EVALUATION

RQ1: Accuracy, comparison with other tools

RQ2: Scalability

RQ3: Case study for new Vulnerabilities

7 DISCUSSION

8 RELATED WORK

9 CONCLUSION

REFERENCES

[1] 2019. A Block Explorer and Analytics Platform for Ethereum. https://etherscan.io/. Online; accessed 29 January 2019.

[2] Adrian Manning. 30 May 2018. Solidity Security: Comprehensive List of Known Attack Vectors and Common Anti-patterns. https://blog.sigmaprime.io/solidity-security.html. Online; accessed 29 January 2019.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2016. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive* 2016 (2016), 1007.

[4] ConsenSys Diligence. 2019. Ethereum Smart Contract Best Practices:Known Attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/. Online; accessed 29 January 2019.

[5] Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: A Static Analysis Framework For Smart Contracts. In *2nd IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montréal, Canada*. to appear.

[6] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS 2016*. 254–269.

[7] Mavs Dev Team. 2019. Mavs: Semi-automated Construction of Vulnerability Benchmark for Smart Contracts. https://mavspublic.github.io/mavs_test_cases/. Online; accessed 29 January 2019.

[8] NCC Group. 2019. Decentralized Application Security Project (or DASP) Top 10 of 2018. https://dasp.co/. Online; accessed 29 January 2019.

[9] Solidity Dev. Team. 2019. Solidity in Depth: Types. https://solidity.readthedocs.io/en/v0.5.10/types.html. Online; accessed 30 June 2019.

[10] Solidity Dev. Team. 2019. Solidity, the Contract-Oriented Programming Language. https://github.com/ethereum/solidity. Online; accessed 30 June 2019.

[11] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS 2018*. 67–82.

[12] Xiangfu Zhao, Zhongyu Chen, Xin Chen, Yanxia Wang, and Changbing Tang. 2017. The DAO attack paradoxes in propositional logic. In *ICSAI 2017*. 1743–1746. https://doi.org/10.1109/ICSAI.2017.8248566