Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts

Yinxing Xue
University of Science and Technology of China
Hefei, China
yxxue@ustc.edu.cn

Mingliang Ma
University of Science and Technology of China
Hefei, China
sa517245@mail.ustc.edu.cn

Yun Lin*
National University of Singapore
Singapore
dcsliny@nus.edu.sg

Yulei Sui
University of Technology Sydney
Sydney, Austrilia
yulei.sui@uts.edu.au

Jiaming Ye
University of Science and Technology of China
Hefei, China
sa517462@mail.ustc.edu.cn

Tianyong Peng
University of Science and Technology of China
Hefei, China
sa517270@mail.ustc.edu.cn

ABSTRACT

Reentrancy bugs, one of the most severe vulnerabilities in smart contracts, have caused huge financial loss in recent years. Researchers have proposed many approaches to detecting them. However, empirical studies have shown that these approaches suffer from undesirable false positives and false negatives, when the code under detection involves the interaction between multiple smart contracts.

In this paper, we propose an accurate and efficient cross-contract reentrancy detection approach in practice. Rather than design rule-of-thumb heuristics, we conduct a large empirical study of 11714 real-world contracts from Etherscan against three well-known general-purpose security tools for reentrancy detection. We manually summarized the reentrancy scenarios where the state-of-the-art approaches cannot address. Based on the empirical evidence, we present Clairvoyance, a cross-function and cross-contract static analysis to detect reentrancy vulnerabilities in real world with significantly higher accuracy. To reduce false negatives, we enable, for the first time, a cross-contract call chain analysis by tracking possibly tainted paths. To reduce false positives, we systematically summarized five major path protective techniques (PPTs) to support fast yet precise path feasibility checking. We implemented our approach and compared Clairvoyance with five state-of-the-art tools on 17770 real-worlds contracts. The results show that Clairvoyance yields the best detection accuracy among all the five tools and also finds 101 unknown reentrancy vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; • **Software and its engineering → Software safety**.

*Yun Lin is the corresponding author.

KEYWORDS

reentrancy vulnerabilities, static taint analysis, cross-contract analysis, smart contracts

ACM Reference Format:
Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416553

1 INTRODUCTION

Powered by the blockchain platform, smart contracts are executable contracts written in the form of computer programs [38], which facilitate, verify, and enforce the trading transactions between buyers and sellers in an automatic and transparent way without involving third parties. On the most popular platform (i.e., Ethereum [1]), contracts are written using Solidity language. Security issues, as surveyed in [8], are the major concerns of Solidity programs.

Reentrancy, also known as the notorious DAO attack [16], has caused a devastating financial loss of around $150 million stolen Ether (digital currency) for many Ethereum accounts. The vulnerability lies in that the attacker can leverage fallback functionalities in Solidity to repetitively incur the payment until exhausting the balance of the victim (an example is shown in Fig. 1). To address this issue, existing approaches [13, 24, 25, 35, 48–50] define a variety of rules to detect and avoid illegal (or suspicious) use of the fallback function in Solidity program. In this regard, most general-purpose security scanning tools (e.g., Slither [49], Oyente [35], Zeus [25], Mythril [13], Manticore [48], Contract-Fuzzer [24] and Securify [50]) claim to support reentrancy bug detection.

However, there lacks a study on how the existing general-purpose security tools actually perform on large-scale reentrancy detection from real-world contracts. To answer this problem, in this paper, we initially conduct a large-scale empirical study on using three general-purpose static tools for reentrancy detection (i.e., Oyente [35] from CCS'16, Securify [50] from CCS'18 and Slither [49] from industry) on 11714 real-world contracts from Etherscan [2]. The empirical evidence shows that:

- Programmers have already invented many programming paradigms to avoid reentrancy attack. Without being aware of those paradigms, existing state-of-the-art security tools report many false positives (FPs);
- On the other hand, existing tools usually define rules at intra-procedural level, which incurs the incomplete modeling of program paths, particularly cross-contract and inter-procedural call paths in Solidity programs. It also causes a considerable number of false negatives (FNs).

In this work, we formally summarize those paradigms to avoid reentrancy attack as *Path Protection Technique* (PPT), and propose a cross-function and cross-contract static analysis approach for reentrancy bug detection. Our approach models more sound interprocedural paths (to reduce FNs) and identifies feasible interprocedural paths by considering PPTs (to reduce FPs). More specifically, we first construct a cross-contract interprocedural control flow graph (XCFG), on which a static taint analysis is performed to identify potential paths that contain a reentrancy, and then perform a light-weight symbolic analysis based on PPTs to eliminate infeasible paths, and finally report reentrancy bugs based on the refined feasible program paths. Based on the technique, we build and publish our tool Clairvoyance (avaiable at [5]). We compared Clairvoyance with five state-of-the-art tools on 17770 contracts from Google BigQuery Open Dataset. The experimental results show that Clairvoyance (1) yields the best detection accuracy among all the tools and (2) finds 101 unknown reentrancy vulnerabilities.

In summary, this paper makes the following contributions:

- We present a large-scale empirical study to (1) evaluate the ineffectiveness of three state-of-the-art static tools to detect reentrancy attack in practice and (2) understand the practical programming paradigm (summarized as PPT) to avoid reentrancy attack.
- We present a new static reentrancy detection approach to enable (1) more sound analysis by modeling cross-function cross-contract behaviors, and (2) more precise analysis by applying a light-weight symbolic analysis based on PPTs.
- Unlike existing static tools which yield a binary result (vulnerable or not), our approach supports a more accurate bug reporting method by providing the feasible call chain(s) that cause the reentrancy. So the *understandability* of detection results is improved.
- We have compared our tool, Clairvoyance, with the three state-of-the-art static tools (i.e., Oyente [35], Securify [50] and Slither [49]) and two dynamic tools (i.e., Mythril [13] and sFuzz [21]) on 17770 contracts from Google BigQuery Open Dataset. The results show that Clairvoyance has *significantly better* accuracy than the compared tools, and successfully finds 101 *unknown* reentrancy bugs that are all missed by the compared tools. Our experimental results are publicly available at [5].

2 BACKGROUND AND MOTIVATION

In this section, we briefly introduce the fallback mechanism of Solidity and explain how it can cause reentrancy. Then, we introduce three state-of-the-art static tools, including Slither [49], Securify [50], and Oyente [35], with examples of the FNs and FPs.

```
1  contract Partner {
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 }
```

Figure 1: A simple example for reentrancy vulnerability.

2.1 Fallback Function of Solidity Program

On Ethereum, most smart contracts are implemented in Solidity, which supports basic structural elements, including contract (similar to class in OOP), variable, function, etc. Once Solidity contract is published (e.g., on Ethereum), the program is regarded as a "law" and can no longer be altered. Given the law property, special mechanisms such as fallback function are introduced. From law point of view, the contract should define its behavior in any case. Hence, from technical point of view, the execution of fallback function is triggered to handle some exceptional cases such as (1) when its owning contract is called with an unknown function name, or (2) when the contract receives plain Ether (i.e., Ehtereum currency) without data. Nevertheless, such a mechanism introduces notorious reentrancy vulnerabilities, causing devastating economic loss [6, 8].

2.2 Reentrancy Vulnerability

Fig. 1 shows an example of a reentrancy vulnerability. The `Partner` contract supports other contracts to deposit money to (via `deposit()` function at line 3) or withdraw money from (via `withdraw()` function at line 6) its account. The other contract account is represented by `balances[msg.sender]` field in `Partner` contract[1]. In the `withdraw()` function, `Partner` contract first checks whether `balances[msg.sender]` has a sufficient amount (line 7). If so, `Partner` transfers to the other contract (line 8) and updates the balances (line 9).

The root cause lies in that the malicious contract can repetitively incur the payment (line 8) without finishing the call to the method `withdraw()` until exhuasting the balance. Assume a malicious contract `Mali` has two functions, one function `attack()` that calls `Partner.withdraw()` and the other function (the fallback function) that calls `Partner.withdraw()`. The attack happens when `Mali.attack()` calls `withdraw()` function, which executes line 8 of Fig. 1. The execution of `msg.sender.call.value()` function at line 8 will trigger the fallback function of `Mali` contract. Note that `Mali` contract can craft its own fallback function to call `Partner.withdraw()` function for his or her own benefits. Once the fallback function of `Mali` contract is triggered to call `Partner.withdraw()` function, the control flow goes back to line 7 and line 8, before `balances[msg.sender]` can be updated at line 9. Such a trick nullifies the condition check at line 7. By this means, `Mali` can repeatedly withdraw money from `Partner` contract.

2.3 State-of-the-arts and Their Limitations

Existing static security tools such as Slither [49], Securify [50], and Oyente [35] claim to address this vulnerability. These tools

[1]In Solidity, each contract has an address. Keyword `msg.sender` represents the address of contract interacting with the owner contract.

```
1  contract Trader {
2    TokenSale tokenSale = new TokenSale(); // Internal
         Contract defined at line 8 in the same file
3    function combination() {
4      tokenSale.buyTokensWithWei();
5      tokenSale.buyTokens(beneficiary);
6    }
7  }
8  contract TokenSale {
9    TokenOnSale tokenOnSale;   // External Contract
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() onlyTrader {
20     wallet.transfer(weiAmount);
21   }
22 }
```

Figure 2: Complicated call graph and control flow graph cross contracts — an FN for SLITHER, OYENTE and SECURIFY.

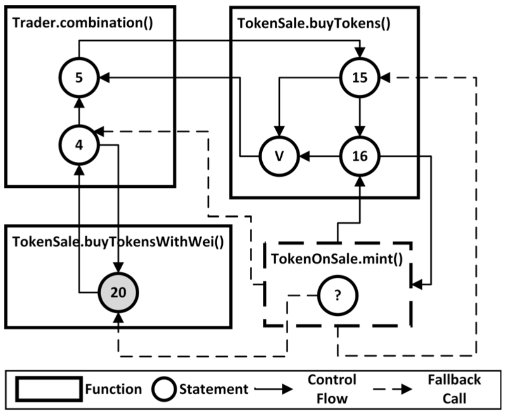

Figure 3: Cross-function and Cross-contract CFG for Fig. 2*.

*Note: The grey node ⑳ denotes a statement of payment function (reentrancy vulnerability requires reentering of a payment function), ⑦ denotes a malicious statement in the external unknown function that may call back.

have their self-defined rules for checking vulnerabilities in a Solidity program. For example, according to SLITHER's rule, a reentrancy may happen if it satisfies the following condition:

$$\boxed{\left(r(var_g) \vee w(var_g)\right) \succ externCall \succ w(var_g) \Rightarrow \text{reentrancy} \quad (1)}$$

In Rule 1, $r()$ and $w()$ denote the read and write operations, respectively; var_g denotes a certain public variable; $\succ$ denotes the execution order based on the program's control flow; *externCall* denotes the external call to the *payment* functions except *built-in* functions `send()` and `transfer()`. Thus, the rule means that if there exists an external call to a payment function and the call is between two write operations to the same public variable, a reentrancy may happen. In Fig. 1, function `withdraw()` is identified as a vulnerable function by this rule. Specifically, (1) a read access to a public state variable (at line 7), then (2) a call to an external address via the low-level function `call.value()` (at line 8), and (3) a write access to the public state variable (at line 9). OYENTE and SECURIFY have similar static rules to detect the reentrancy in Fig. 1 (see §4).

These simple rules have been proved to be effective in some cases, but more often, make detection results fall into FPs or FNs.

False Negative (Missing Complete Call Chain): Existing static tools for smart contracts usually fail to consider cross-function or cross-contract call chains, missing analysis of some important yet suspicious program paths. Fig. 2 shows a contract contains a reentrancy vulnerability that is missed by SLITHER, SECURIFY, and OYENTE. Fig. 3 shows its corresponding cross-contract CFG (XCFG), where each rectangle represents a function, each circle represents a statement denoted by its line number in Fig. 2, and each directed edge represents the control flow. Moreover, we use a virtual node (denoted by V) in the function of `TokenSale.buyTokens()` to show the returned control flow, a grey node to represent payment statement, and dashed rectangle to represent an unknown function of an external contract (i.e., `TokenOnSale.mint()`). *Notably, the dashed rectangle could call any function as its behavior is unknown.*

From Fig. 3, we can see that, as long as `TokenOnSale` contract is malicious, there exists a call chain ④ → ⑳ → ④ → ⑤ → ⑮ → ⑯ → ⑦ → ④ → ..., which allows the attacker to recursively call `wallet.transfer(weiAmount)` (line 20). Such attack crosses multiple contracts and functions, and does not write any public variable, which makes it hard to enumerate rules for its detection.

False Positive (Inaccurate Path Feasibility): Existing static tools can hardly identify path feasibility. Fig. 4 shows a contract that is mistakenly reported as a reentrancy vulnerability by SLITHER and SECURIFY as it conforms to their defined rules. Fig. 5 shows its XCFG. We can see that there exists a reentered call chain ⑤ → ⑦ → ⑧ → ⑨ → ⑦ → ⑤ → ⑦ → ..., as the unknown function `ZTHTKN.buyAndSetDivPercentage.value()` may use the fallback function to call back function `receiveDividends()`. However, in practice, such path in CFG to exercise the chain can never be feasible. The reason is that ⑤ → ⑦ requires $reEntered = true$ while ⑦ → ⑤ → ⑦ requires $reEntered = false$. They are contradictory.

Fig. 4 is a typical example, where *a path protective technique* (PPT) is applied as a countermeasure for reentrancy attack. The logic of the code ensures that the check condition (line 5) can always protect the execution of payment statement. Unfortunately, both SLITHER and SECURIFY fail to recognize this infeasible path. *Notably, to accurately identify this PPT from code, light-weight symbolic analysis is required to check locking at line 8 and unlocking at line 10, and such analysis could be just limited in the internal function.*

To overcoming FPs/FNs is non-trivial, which requires addressing these challenges:

- Given the behavior of the possible externally-interacting contract is unknown, how do we model its unknown call graph?
- The path number explodes when the function/contract number increases, how do we efficiently identify those suspicious reentrancy attack paths from a set of Solidity programs?
- Can we conduct the light-weight path feasibility analysis, which can achieve the efficiency meanwhile not compromising the soundness/completeness of our approach?
- How can we incorporate Solidity-specific features impacting control flow graph like function modifier, address binding, etc.?

We propose a scalable static approach that can systematically compute those suspicious and feasible paths to cause reentrancy vulnerability with respect to both cross-contract call chains and

```
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 { // To be inherited for
       implementing  the function buyAndSetDivPercentage
15   function buyAndSetDivPercentage(address _referredBy,
         uint8 _divChoice, string providedUnhashedPass)
         public payable returns (uint); // Declaration
16 }
```

Figure 4: Complicated path constraints due to using an execution lock `reEntered` — an FP for Slither and Securify.

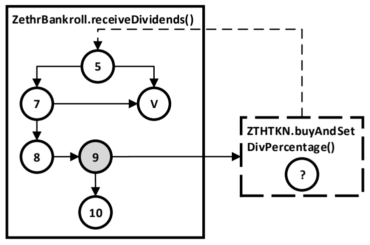
ZethrBankroll.receiveDividends()
5
7
V
8
9
10
ZTHTKN.buyAndSet DivPercentage()
?

Figure 5: Cross-function and Cross-contract CFG for Fig. 4.

path feasibility. The cross-contract call chain construction can effectively mitigate the problem of missing reentrancy control flow. Identifying infeasible paths mainly relies on path protective techniques (PPTs) to reduce the FPs where reentrancy vulnerabilities never exist.

3 OVERVIEW

Problem Statement. We argue that reentrancy attack could happen *when there exists a feasible program path in cross-contract CFG (XCFG) of Solidity contracts, which starts from setting an insecure contract (or address) and ends with the payment function call* (e.g., `call().value()`) of that contract (or address).

Assumption & Challenges. Our approach follows the close-world assumption of analyzing smart contracts. Given a target contract, our analysis scope includes the source code of all its caller and callee contracts. The Solidity APIs of a program without function bodies (e.g., fallback and low-level built-in functions) are analyzed based on their side-effect summarization following the standard approach in static analysis.

Fig. 6 shows an overview of our approach, which takes as input the source code of Solidity contracts and its caller/callee, and reports call chain(s) with path conditions to explain how a malicious contract can construct a reentrancy attack to exploit the vulnerability. In Fig. 6, each rectangle represents an artifact, each ellipse represents a step (or process), each edge represents the workflow of how a step takes some artifacts and outputs some new artifact. In addition, we highlight the input/output in grey. As shown in Fig. 6, our approach consists of the following four steps:

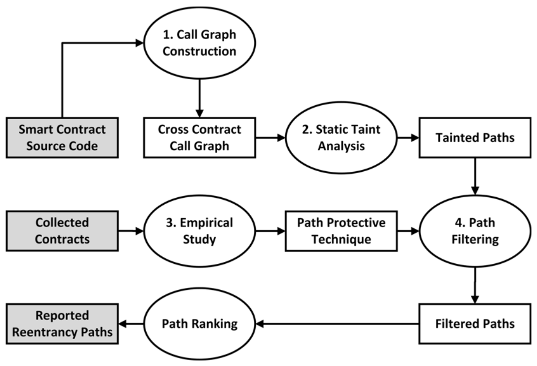
1. Call Graph Construction
Smart Contract Source Code
Cross Contract Call Graph
2. Static Taint Analysis
Tainted Paths
Collected Contracts
3. Empirical Study
Path Protective Technique
4. Path Filtering
Reported Reentrancy Paths
Path Ranking
Filtered Paths

Figure 6: System diagram

S1. XCFG Construction: We first construct cross-contract call graph and CFG among the input smart contracts. In this step, we handle Solidity specific features (e.g., modifier, fallback function) and collect all call chains starting from public functions.

S2. Static Taint Analysis: Next, we identify tainted contract objects or addresses, and track how they are used and propagated along the call chain from the XCFG. In this step, we identify vulnerable call chains from XCFG, which start with a suspicious contract address (or object) and end with the external function call.

S3. Empirical Study (Summarizing PPTs): In order to mitigate the FPs of reported paths (abstracted by the vulnerable call chains), we summarize a set of PPTs for reentrancy attack through an empirical study, which could be used for checking the path feasibility in a scalable way.

S4. Path Filtering: For these PPTs, we define the corresponding filtering patterns via program analysis: paths with access control, paths with hard-coded address, paths with execution locks, paths with internal updating before payment, etc. Then, given the vulnerable call chains from S2 and the filtering patterns behind PPTs from S3, we filter out infeasible paths, which cannot form reentrancy in practice and hence lead to FPs. Finally, the remaining reachable paths will be the outputting results.

S1 (§5.1) and S2 (§5.2) are designed for mitigating FNs, while S4 (§5.3 and §5.4) is designed for addressing FPs. Next, for readability, we first explain S3 (§4) — the empirical study for reasons: (1) quantitatively analyzing how (in)effective existing static tools on 11714 smart contracts in the field; (2) how effective our summarized PPTs are. Then, we proceed to elaborate on each step of our approach.

4 EMPIRICAL STUDY OF TYPICAL FALSE POSITIVES FOR EXISTING RULES

In our empirical study, Slither [20] v0.4.6, Oyente [35] v0.2.7 and Securify [50] v1.0 run on 11714 frequently-used real-world contracts from the well-known third-party website Etherscan for contract indexing and browser [2]. For the detection results, we recruit 4 researchers to spend 2 months in reviewing the results and summarizing the patterns of FPs for rules in these tools.

4.1 Rules of Existing Tools

In §2.3, the detection rule of Slither checks whether there exists an *external* call of money transfer functions (except built-in functions

```
 1 contract CozyTimeAuction {
 2   function buyCozy(uint256 _pepeId, uint256 _cozyCand,
        address _pepeRec) public payable {
 3     require(address(pepeContract) == msg.sender);
 4     PepeAuction storage auction = auctions[_pepeId];
 5     totalFee = ... // ignore the details
 6     auction.seller.transfer(price - totalFee); //transfer
 7     if(!pepeContract.cozyTime(auction.pepeId, _cozyCand,
          _pepeRec)) {...} //external call
 8     delete auctions[_pepeId]; // write after call
 9   }...
10 }
```

Figure 7: A real case of using access control (PPT1) for the account address — an FP for Slither and Securify.

send(), transfer()) that happens between writes to a public variable. Similarly, according to [50], we find that Securify proposes the compliance pattern *No writes after call* to prevent reentrancy[2]. Hence, the corresponding violation pattern is to have writes after call, as shown by the Rule 2:

$$\boxed{externCall > w(var_g) \Rightarrow \text{reentrancy}} \quad (2)$$

where $w(var_g)$ means write operation(s) to a public variable and *externCall* refers to the external call for money transfer functions.

Oyente supports the following Rule 3, according to [35]:

$$r(var_g) \land (gas_{trans} > 2300) \land (var_{t\_a} > var_{d\_a}) \land (w(var_g) > externCall)) \Rightarrow \text{reentrancy} \quad (3)$$

where $r()$ and $w()$ means read and write operation(s) to a public variable, $gas_{trans} > 2300$ means the gas for transaction must be greater than 2300, the transfer amount var_{t_a} must be greater than the deposit amount var_{d_a}, and the public variable var_g should be changed before the external call (denoted by >). As Oyente works on the EVM bytecode instructions, the gas consumption is estimated on the instructions. Notably, Slither and Securify consider *writes after calls* as one condition that forms reentrancy, but Oyente considers *writes before calls* as one key condition[3].

After we investigate the implementations of these three state-of-the-arts, we observe that their detection rules are too simple and coarse-grained, basically ignorant of the possible programming skills used by the developer to prevent the reentrancy. *Shortly, path in-feasibility analysis is insufficiently supported by the three static state-of-the-arts.* Hence, it is expected that these tools will yield many FPs for real-world contracts.

4.2 PPT1: Access Control Before Payment

Fig. 7 gives an FP reported by Slither and Securify based on its Rule 1. The code firstly reads the state variable `auctions[_pepeId]`; then calls an external function via `pepeContract.cozyTime()`; last, writes (`delete` instruction belongs to the general write operations) to the public variable `auctions[_pepeId]`. However, in reality, reentrancy cannot be triggered by external attackers due to the require check `require(address(pepeContract) == msg.sender)` at line 3 in Fig. 7. Access control usually checks the invoker of the payment functions — checking whether the identity of `msg.sender` satisfies certain conditions (e.g., in some authorized list, with a good reputation, or having the dealing history). Notably, PPT1 needs to reside in the same function as external calls, otherwise it can be avoided.

[2]In Securify, two types of detected errors are related to reentrancy, i.e., *DAO* and *DAO Constant Gas*. The two errors will lead to reentrancy, and the only difference is whether the gas will be recursively used or not (with constant gas).

[3]We confirm this rule from the implementation of Oyente.

```
 1 interface HgInterface {
 2   function buy(address _add) 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     rID[rNumber].lastTime = _timestamp;...
14   }
15 }
```

Figure 8: A real case of using the constant value (PPT2) for the contract address — an 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 _amt) external onlyHuman payable{
    require(balances[msg.sender] >= _amt);
    require(msg.sender.call.value(_amt)());
    balances[msg.sender] -= _amt;
  }
}
```

Figure 9: A real case of using a self-defined modifier for protection (PPT3) — an FP for Slither, Oyente and Securify.

4.3 PPT2: Hard-coding Payment Address

Fig. 8 shows an FP reported by Oyente and Securify, which adopts a hard-coded address to prevent the malicious external attack. According to Rule 3, as this code block has a balance check and a low-level-call for money transfer, it is reported by Oyente as a reentrancy. Similarly, according to Rule 2, there is a write operation to the public variable `rID[rNumber].lastTime` after the external call, and hence Securify detects it. However, this example cannot be exploited by any arbitrary external address, owing to the 20 bytes hard-coded address (i.e., `0xB3...efBe`) for contract object `p3d` at line 7 in Fig. 8. Hence, PPT2 restricts the external malicious access.

4.4 PPT3: Protection by Self-defined Modifiers

Fig. 9 gives another FP that is reported by the existing tools ignoring PPT3. This code block actually considers the security issue and adds the self-defined modifier `onlyHuman()` before the potential vulnerable function `buy()`. Since `onlyHuman()` restricts that the transaction can be only conducted by the address of `msg.sender` via the usage of keyword `extcodesize`, which returns 0 if it is called from the constructor of a contract. In such a way, via PPT3, `buy` could not be recursively called by external attackers.

4.5 PPT4: Protection by Execution Locks

Different from the above PPTs relying on access control or restricted addresses to prevent external malicious calls, PPT4 prevents the recursive entrance of the function — eliminating the issue

```
contract PvPCrash {
  function withdraw() gasMin public returns (bool) {
    address _user = msg.sender;
    uint256 _userBalance;
    if (!roundEnded && withdrawBlock[block.number] <=
        maxNumBlock) {
      _userBalance = getBalance(_user);
      if (_balance > _userBalance) {
        if (_userBalance > 0) {
          _user.transfer(_userBalance); //externalCall
          emit Withdraw(_user, _userBalance);
        }
      return true;
      }
    }
    return true;
  }
}
```

Figure 10: A real case of updating internal states before payment (PPT5) — an FP for Oyente.

from root. For instance, in Fig. 4, the internal instance variable `reEntered` will be checked at line 5 before processing the business logic between line 8 and 10. To prevent the reentering due to calling `ZTHTKN.buyAndSetDivPercentage.value()` , `reEntered` will switch to `true`; after the transaction is finished, it will be reverted to `false` to allow other transactions. Hence, by virtue of PPT4, this solution is similar with using mutex, which prevents the reentrancy from both the authorized addresses or external malicious addresses.

4.6 PPT5: Internal Updating Before Payment

PPT5 is required to finish all internal work (i.e., state and balance changes) and then call the external payment function. According to the report from ConsenSys [15], the recommended pattern is composed of three steps: (1) all the `require` checks are at the beginning of function `buy_the_tokens`; (2) in the middle are the internal state changes for bool and numeric variables; (3) at the end is the call of built-in payment functions `transfer()` of some external contracts or addresses. This pattern is also recommended by Solidity official document at [42], named as *Checks-Effects-Interactions pattern* to prevent reentrancy. For example, for the code in Fig. 10, it follows this safe pattern. However, according to Rule 3 for Oyente, this code block is mistakenly reported as vulnerable.

4.7 FP Statistics of Existing Rules

Finally, we audit all the results of the three tools, and summarize the number of FPs due to mis-considerations or unsatisfactory support of these PPTs. As a result, 67.5% of FPs are accounted for the 5 PPTs in Slither, 71.4% of FPs are for PPTs in Oyente, and 95.15% of FPs are for PPTs in Securify. Hence, we can confirm that the available static tools ignore or unsatisfactorily support these PPTs in code. *Note that recall is not statistically evaluated in this empirical study, as our goal is to understand the reasons behind the FPs of these tools.* According to our observation, Securify and Slither have better recalls than Oyente, as Rule 1 and 2 are more general than Rule 3.

5 APPROACH

In this section, we will detail each step of our approach in Fig. 6, including XCFG construction, static taint analysis, path filtering based on PPTs, and finally outputting reachable vulnerable paths.

Table 1: #FPs due to the inconsideration or unsatisfactory support of PPTs for different tools in our empirical study.

	Slither v0.4.6	Oyente v0.2.7	Securify v1.0
PPT1	7	2	16
PPT2	28	2	51
PPT3	15	3	46
PPT4	4	0	6
PPT5	0	3	38
PPTs/all #FP	67.5%	71.42%	95.15%

5.1 XCFG Construction & Call Chain Gathering

XCG and CFG Generation. We first build the cross-contract call graph (XCG) for Solidity programs, which are directed graphs where nodes are Solidity functions and edges between them denote calling relations. The call graph of Solidity programs is generally similar with that of other OOP languages, except that the fallback functions and self-defined modifiers need to be considered to add additional call edges for a more accurate call graph. For each function inside the XCG, we also build its CFG adopted from SlithIR [20] after the abstract syntax tree (AST) parsing.

XCFG Generation. XCFG is a combination of XCG and all the CFGs of functions of a Solidity program (not explicitly supported by Slither). According to the XCG, we can connect all the CFGs for all functions and modifiers, and attain the corresponding XCFG. Let us denote the two parts of a function call cross contracts: a call-site node c_i and a return-site node r_i. There is a cross-contract edge $c_i \rightarrow s_c$ from a call-site node to the start node (s_c) of the called contract c; there is also a corresponding edge $e_c \rightarrow r_i$ for a dedicated exit node e_c. For example, the XCFG is illustrated in Fig. 3 and Fig. 5. Different from call chains starting from the entry of *main* function in Java, a call chain can start from any *public* function in Solidity and is considered valid if its cross-contract edges are matched (i.e., each r_i is matched with the corresponding c_i). In our study, we aim to find *vulnerable call chains* from XCFG that satisfy the condition — call chains with a loop due to calling unknown external functions that may call back to the current callee function. Hence, the problem can be reduced to that — how to find vulnerable call chains on the XCFG as complete as possible, a typical graph traversing problem.

The Collector. We design the call chain collector with the following Algo. 1. Given the XCFG of a Solidity source code file, the input includes the taint input source IS that is listed in Table 2, the set of all public functions F and the set of all call chains C_F in the XCFG. The output yielded by the approach is the set of vulnerable call chains VC, each of which actually abstracts a possible vulnerable path (VP) that leads to a reentrancy attack at execution time. Algo. 1 is composed of three steps: (1) identifying the vulnerable call chains via static taint analysis at line 1 to 8, see §5.2; (2) filtering the call chains that are actually non-vulnerable due to the adoption of PPTs at line 9 to 11, see §5.3; and (3) finally outputting all remaining vulnerable call chains as results at line 12. If the input source file(s) have in total n functions (the maximum length of a call chain without loop) and m lines of code (the maximum length of a path without loop), the time complexity of Algo. 1 is $O(nm)$. The most time-consuming part is the two-level loop at line 1 to 8.

Algorithm 1: CollectingVulCallChain(): traversing the XCFG and collecting vulnerable call chains

```
input  : $IS$, all the input source
input  : $F$, all the public functions in XCFG
input  : $C_F$, all the call chains in XCFG
output : $VC \leftarrow \emptyset$, the set of potentially vulnerable call chains
1  foreach call chain $c \in C_F$ do
2      // get the concrete paths for $c$ on the XCFG
3      $P_c \leftarrow c.getConcretePaths()$
4      foreach path $p \in P_c$ do
5          $s \leftarrow getSource(IS, p)$ // get input source for $p$
6          $p_t \leftarrow propagateByRules(p, s)$ taint propagation
7          if $isSinkOfTainted(p_t)$ is True then
8              $VC \leftarrow VC \cup \{c\}$, $break$; // $c$ is vulnerable
9  foreach call chain $c \in VC$ do
10     if $ifExistPPT(c)$ is True then
11         // if a filtering pattern in Table 3 is found
           $VC \leftarrow VC - \{c\}$ // $c$ is non-vulnerable
12 return $VC$
```

Table 2: The sources and rules of the static taint analyzer for Solidity programs.

IS	(1) msg.sender, tx.origin
	(2) parameters of public function
Rules	(1) data assignment: a(address type) = _address
	(2) address binding: object = ContractOfObject(_address)
	(3) return value of functions: address/object = functon()
Sink	object.method(), if object is tainted, then it is tainted
	address.call.value(), if address is tainted, then it is tainted

As n and m are usually not large in real-world contract files, the overall performance is practically good on real-world contracts.

5.2 Static Taint Analyzer

Given the concrete paths represented by call chains from the XCFG, we identify vulnerable functions (**VFs**) and vulnerable paths (**VPs**) by our static taint analysis as follows:

Target of Static Taint. In this study, *vulnerable functions* (VFs) refer to those unsafe functions that are susceptible to reentrancy attack. Vulnerable paths (VPs) refer to the execution paths allowing to read external input parameters P_i (e.g., external address or transaction amount) and lead to a VF with variables values depending on P_i. We also call the set of input source (**IS**) of parameters as tainted data. Thus, we aim to identify the critical VPs that are susceptible to reentrancy attack on the XCFG of a program via the tainted data-dependency paths flowing from an IS to a VF. However, unlike the taint analysis for Java or C++ programs, we need to consider the special features of Solidity language.

Def and Use Relations. We first build the data dependency relationship among the variables in Solidity programs. There are two types of program points (or nodes on XCFG) on the def-use chain relations: 1) a *use* site that only reads one or multiple variables and 2) a *def* site that at least writes to one variable. For the 5 types of variables and 14 types of operations in SlitherIR [41], we build the def-use relations for each of the 14 types of IR operations.

Static Taint Rules. As shown in Table 2, we propose the IS for taint analysis and the propagation rules that suit to Solidity programs. The IS for taint includes the parameters of public functions and the two special features of Solidity, namely `tx.origin` and `msg.sender`. The former refers to the original external address that starts the whole transaction, while the latter just refers to the external address that calls the current contract. All the IS may be tainted with malicious external addresses, and propagated to the VFs via the propagation rules. Similar with other OOP languages, the data assignment operation (Rule 1) and the function return assignment (Rule 3) propagate the tainted data. Uniquely, in Solidity, the address binding operations (creating a contract object from an address in Rule 2) also propagate. Finally, in the XCFG, a VP is identified if there is a path satisfying any of the two conditions: *(1) if a contract object is tainted, calling any of its public methods (potentially invoking the fallback function) is vulnerable; (2) if an address variable is tainted, calling any of its low-level functions is vulnerable.*

Specifically, Rule 1 for program/data assignment involves the following operations in SlitherIR: assignment, binary, unary, new operator, push, convert, array initialization, member for the three types of variables: StateVariable, LocalVariable and SolidityVariable. Rule 2 involves the operations of member, convert, assignment, array initialization and index, as the address binding may refer to one address or an array of addresses. Last, rule 3 involves the operations of member, call, return and assignment.

Table 3: The filtering patterns behind PPT1–PPT5.

PPT	Filtering Pattern
PPT1	$(isAuthorized(msg.sender) \lor hasPermit(msg.sender)) \succ msg.sender.externalCall$
PPT2	$(var_{add} ==$ const $\lor\ var_{obj} ==$ Contract(const)$) \succ (var_{add}.externalCall \lor var_{obj}.externalCall)$
PPT3	The above two patterns in the self-defined modifiers of functions that have external payments
PPT4	$(isChecked(v_l) \succ w(v_l) \succ externalCall \succ w'(v_l)) \land w(v_l)! = w'(v_l)$
PPT5	$isChecked(V_{int}) \succ w(V_{int}) \succ (var_{obj}.transfer())$

5.3 Path Filtering based on PPTs

Given the VPs reported by the static taint analyzer, we need to apply the filters that take into account the PPTs. Behind each PPT, we define a filtering pattern shown in Table 3.

The filtering pattern for PPT1 is to check whether `msg.sender` is within a list of authorized contracts or addresses, or has the permission to do this (e.g., `msg.sender==owner`) and finally the above check is required to be within the same function and *before* (denoted by $\succ$) the external call of the tainted address or contract.

The filtering pattern for PPT2 is to check whether the tainted address or object has been initialized in declaration or modified before the external call, with a 20-byte length string constant.

The pattern for PPT3 is actually applying the above two patterns in self-defined modifiers of the function with the suspicious tainted object or address that has the external call.

The pattern for PPT4 is to check the existence of the execution lock, where v_l denotes the boolean variable to be used as the lock and $isChecked(v_l)$ denotes whether v_l is checked via `require` (or `if`, `assert` checks etc.). $w(v_l)$ denotes the *first* time of write operation to v_l, and $w'(v_l)$ denotes the *second* time of write operation to v_l. Hence, $w(v_l)! = w'(v_l)$ means that the first and the second write operation assign the different values to v_l.

The pattern for PPT5 is to check the existence of checks-effects-interactions pattern, where V_{int} denotes the set of internal variables for the current contract, $w(V_{int})$ refers to the write operations to V_{int} for the internal state or balance changes, and last $var_{obj}.transfer()$ denotes the external call of the build-in function `transfer()`. Notably, even if var_{obj} is tainted, the code is still non-vulnerable, as the built-in function $transfer()$ using a fixed amount of gas and will not repeatedly reenter the payment function.

5.4 Light-weight Symbolic Analysis

To make the PPT-based filtering more accurate, a light-weight symbolic analysis is developed and employed across PPT1-4, assisting the reentrancy detection. Our light-weight symbolic analysis is both intra-procedural path-sensitive and context-insensitive to synthesize a symbolic path from tainted source to the fallback call. Then we feed the path into Z3 solver to check its feasibility.

Given a user defined threshold h, we abstract the loop analysis by unrolling the loop for h iterations. Moreover, we abstract each function call with a new variable of its return type. In other words, our result ensures the soundness of a reported infeasible path. Despite such an abstraction still suffers from FPs in some cases, it largely reduces our run-time detection overhead, while still allows us to avoid potential FPs in a considerable way. For the example in Fig. 5, we synthesize a path "$!reEntered \wedge ActualBalance > 0.01 \wedge reEntered = true$", which is solved to be infeasible, which helps us avoid the false alarm. Last, if the solver returns "unknown" result for the synthesized symbolic path, we discard the path and use the result of PPT for detection.

After the filtering is finished, the remaining reachable call chains (or feasible paths) in VC will be outputted as the detection results.

6 EVALUATION

We have conducted extensive experiments to evaluate the effectiveness of CLAIRVOYANCE. Specifically, we attempt to answer the following research questions (RQs):

RQ1. How effective are the summarized PPTs? Compared with the three available static tools, how is the precision of CLAIRVOYANCE?

RQ2. How useful is the cross-contract static taint analysis? Compared with the three static and two dynamic tools, how is the recall of CLAIRVOYANCE?

RQ3. How efficient is CLAIRVOYANCE in analyzing real-world Solidity programs?

RQ4. Can CLAIRVOYANCE discover real-world unknown reentrancy vulnerabilities, which can lead to DAO attacks?

6.1 Setup

6.1.1 Baselines. In evaluation, we use their latest versions, namely SLITHER v0.6.4, OYENTE v0.2.7 and SECURIFY v1.0. To compare CLAIRVOYANCE with the dynamic tools in terms of recall, we also include MYTHRIL v0.21.20 and sFUZZ v0.0.1 in the tools pool. As no multi-threading options are available, only the default settings are used under the same machine environment.

6.1.2 Dataset for Tool Evaluation. In our evaluation, to fairly compare CLAIRVOYANCE with the other tools, we choose the dataset of 17770 smart contracts, which come from Google BigQuery Open Dataset, a different source other than the 11714 contracts used in our empirical study. In particular, the 11714 contracts used in our empirical study (see §4) are directly crawled from Ethereum block chain. In this experiment, we obtained smart contracts by tracking their deployment addresses from the public Google BigQuery dataset [23]. We downloaded 17770 contracts through the Etherscan API with their deployment addresses.

6.1.3 Experimental Setup. CLAIRVOYANCE is implemented in Python on top of the SlitherIR library. During the evaluation, all the experiments and tools are conducted on a machine running on Ubuntu 18.04, with 8 core 2.10GHz Intel Xeon E5-2620V4 processor, 32 GB RAM, and 4 TB HDD.

6.1.4 Manual Inspection. To validate the false positives (FPs) or true positives (TPs) from these results, we hire four experienced Solidity developers to check the detection results together with their corresponding source code via the aid of the two dynamic tools (i.e., MYTHRIL and sFUZZ) in two months' time. More specially, we asked developers to conduct cross-reference to manually evaluate the results. We divide the developers into two groups (each with two members). In each group, one developer evaluated the results and the other was responsible for double checking the results.

Table 4: The detection accuracy at *function* level for CLAIRVOYANCE and the other three tools on the 17770 contracts*.

	SLI. v0.6.4	OYE. v0.2.7	SEC. v1.0	C.V.
#N	162	28	608	168
#TP	3	4	3	124
#FP	159	24	605	44

*Note: #N refers to the number of detection results, #TP refers to the number of true positives, and #FP refers to the number of false positives.

6.2 RQ1: Evaluating the Precision

Table 4 lists the numbers of detection results of the four static tools, i.e., 608 by SECURIFY, 168 by CLAIRVOYANCE, 162 by SLITHER and 28 by OYENTE. After the validation, it is found that though SECURIFY (i.e., using Rule 2) reports more bugs, most of them are FPs. OYENTE reports fewest bugs and is limited in catching the true bugs found by other tools due to its coarse-grained checking rule (i.e., Rule 3). In comparison, CLAIRVOYANCE has the most TPs and the fewest FPs.

To understand the reasons behind the FPs reported by each evaluated tool, we first classify the FPs into the five categories based on PPTs. The results in Table 5 show that most of the FPs reported by the static tools are due to ignorance or limited support of these PPTs, including 130 of 159 FPs in SLITHER, 22 of 24 FPs in OYENTE, and 478 of 605 FPs in SECURIFY— most of the FPs are due to our summarized five PPTs. This confirms that CLAIRVOYANCE has the best support of these PPTs, and hence achieves the best precision.

Table 5: #FPs due to the inconsideration or unsatisfactory support of PPTs for different tools on the 17770 contracts.

	Sli. v0.6.4	Oye. v0.2.7	Sec. v1.0	C.V.
PPT1	33	2	60	9
PPT2	56	4	121	10
PPT3	25	15	193	0
PPT4	9	0	1	0
PPT5	7	1	103	14
Sum	130	22	478	33
Perc. in All FPs	81.76%	91.67%	79.27%	75.00%

```
1  contract Betting {
2    function Betting() public payable { //   constructor
3      owner = msg.sender;
4      betContrInterface = BettingContrInterface(owner);
5    }
6    function reward() internal {
7      ... // calculate house_fee
8      if (total_bettors <= 1) { forceVoidRace(); }
9      else {
10       require(house_fee < address(this).balance);
11       total_reward = total_reward.sub(house_fee);
12       betContrInterface.payHouseFee.value(house_fee)();
13   }
14 }
```

Figure 11: A real case using PPT2 (the hard-coded address) in constructor at deployment time, an FP for Clairvoyance.

More specifically, we summarize the following observations from the FP results: (1) Securify fails to consider permission controls, hard-coded addresses and self-defined modifiers. It also falsely reports the write operations after calling built-in functions `send()` and `transfer()` as vulnerable, causing FPs since it does not consider PPT5. (2) Oyente basically ignores the protections in self-defined modifiers and has many FPs mostly because it ignores PPT3. (3) Among all the FPs of Slither, it generally has a good support for PPT3 by considering the code of security check in modifiers. However, it still has many FPs due to lack of any symbolic analysis for PPT1 (33 cases) and PPT2 (56 cases). Considering a relatively small number of cases in using execution lock (PPT4), Slither ignores the protection by execution lock(s).

FP example of Clairvoyance. As shown in Table 5, the FPs of Clairvoyance root in the limited support of PPT1, PPT2 and PPT5. After looking into the code of these FPs, the reason for 9 FPs of PPT1 mainly lies in the complicated path conditions (e.g., user-defined functions in condition control), which cannot be well-handled by our light-weight symbolic analysis — actually the path is infeasible due to strict constraints in the user-defined functions, but Clairvoyance reports it feasible. The reasons of 10 FPs of PPT2 are twofold: (1) 5 FPs are due to the hard-coded address in a constructor (see Fig. 11), (2) 5 FPs are also due to the complicated condition solving (e.g., involving string operations) for address constants. Last, the 14 FPs of PPT5 all belong to this kind of corner cases — the statements relevant to PPT5 are all inside an externally unreachable branch, but Clairvoyance falsely reports them reachable.

In Fig. 11, we show an FP example, where our filtering pattern for PPT2 fails because of the hard-coded address in a constructor. According to Algo. 1, we build and search all call chains starting from a public function — *however, the constructor is not considered as an ordinary public function in call chains, as it is only executed at the contract deployment time and cannot be called by other internal/external functions after that.* For function `reward()` in Fig. 11, our taint analyzer suggests that the low-level call `***.payHouseFee.value()` at line 12 is vulnerable, as `betContrInterface` is in taint input source and without any protection from PPTs. However, this case is an FP, as the constructor has assigned the contract address with `msg.sender` during the deployment time (`msg.sender` is the account address of the deploying programmer). As `betContrInterface` is never modified in other places, its address is the account address of the deploying programmer. No external address can trigger the reentrancy, and hence it is an FP due to overlooking of PPT2 in the constructor.

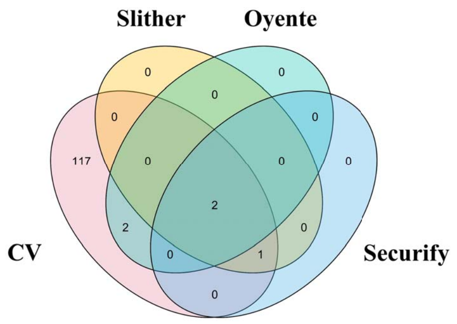

Figure 12: The Venn diagram for illustrating the overlapping and unique TPs among the evaluated static tools.

Table 6: The detection results for two dynamic tools on the 122 vulnerable contracts detected by Clairvoyance*.

Tool	#N	#TP	R%	Detection Time (min.)
Mythril 0.21.20	13	13	10.65%	275
sFuzz 0.0.1	11	11	9.01%	115

*Note: the two dynamic tools cannot finish on 17770 contracts within one week, and hence are applied on 122 vulnerable contracts.

Besides, *there are 11 FPs of Clairvoyance due to other PPTs (not included in §4): (1) strict gas consumption checks for one entrance to prevent reentrancy of the payment function, (2) runtime condition checks on the basis of using* `block.number` *or* `block.time`*, which sometimes rely on the chain status.* In general, these two other PPTs cannot be handled by static analysis. Due to page limit, more details and examples of other PPTs are available at the website [5].

Answer to RQ1: In general, Clairvoyance exhibits a significantly better precision than the other three static tools owing to better support of five PPTs. The other three tools need to improve in terms of PPT1 to PPT3, while Clairvoyance needs the deployment analysis and more accurate symbolic analysis.

6.3 RQ2: Evaluating the Recall

In Fig. 12, this Venn diagram shows that 122 TPs are found by Clairvoyance, 3 TPs by Slither, 3 TPs by Securify and 4 TPs by Oyente. For the overlapping part, only 2 TPs are found by all the four static tools; Slither and Securify report the same 3 TPs; Oyente has 2 TPs that are also found by Clairvoyance. Notably, our tool reports 117 unique TPs missed by the three static tools.

After we inspect the 122 TPs of Clairvoyance, we find the low recall of the three static tools is attributed to two factors: (1) 55 out of 122 (45%) own the vulnerable call chains that involve multiple contracts (except the external attack contract), which could be

```
1  contract RedEnvelope {
2    function create(address token, ...) public payable {
3      ... // checks and calculation for other variables
4      infos[hash] = Info(token, msg.sender, amount,..., 0);
5      emit RedEnvelopeCreated(hash);
6    }
7    function sendBack(bytes32 hash) public {
8      Info storage info = infos[hash];
9      require(info.owner == msg.sender);
10     ... //  more checks
11     uint back = info.amount - info.fill;
12     if (info.isSpecialERC20){
13       SpecialERC20(info.token).transfer(msg.sender,back);
14     }else {ERC20(info.token).transfer(msg.sender,back);}
15     emit RedEnvelopeSendBack(hash, msg.sender,back);
16   }
17 }
```

Figure 13: A real case of misusing PPT1 at line 9 due to the issue of supporting struct `Info`, an FN for Clairvoyance.

similar with the call chain in Fig. 3; and the three static tools fail to conduct cross-contract analysis. (2) The other 67 TP residing in single contract mostly have the call to externally defined high-level functions, not those low-level functions (e.g., `call()`) — however, the three static tools mainly check the call to low-level functions.

In addition to static tools, two dynamic tools are compared in terms of recall in Table 6. Out of 122 TPs found by Clairvoyance, Mythril and sFuzz detect 13 and 11, respectively. Hence, the two dynamic tools show a better recall than the three static tools, but still miss most of the TPs. As dynamic tools are significantly slower than static tools, it will take too much time to run on the whole 17770 contracts. So we just randomly pick up the other 1000 contracts to have testing in the wild for the two dynamic tools, and find two vulnerable contracts (FNs) that are missed by Clairvoyance.

FN example of Clairvoyance. TPs might be missed by Clairvoyance due to the misuse of the PPTs. For example, the FN shown in Fig.13 mistakenly applies PPT1 (access control) for line 9 and removes this case from the set of VC. Nevertheless, after manual auditing, we find that *the* `require` *check at line 9 is not to check the key member* `info.token`, *but to check an irrelevant member* `info.owner` According to `hash` (the source), our taint analyzer reports the *whole* struct `Info` as tainted due to the initialization of the struct at line 4, and then applies PPT1 at line 9. Occasionally, due to the granularity issue of the taint analyzer, it cannot accurately answer which member of a struct object is tainted. Apart from the FN in Fig.13, another tricky case of misusing PPT1 causes the other FN for Clairvoyance— the external low-level call of a tainted address and a permission check are meanwhile inside an `if` condition control, which is mistakenly matched with the PPT1.

Answer to RQ2: In general, Clairvoyance exhibits much better recall than the other three static and two dynamic tools, owing to cross-contract call chain analysis. But misusing PPTs will falsely filter some TPs and cause FNs for Clairvoyance.

6.4 RQ3: Evaluating the Efficiency

On the 17770 contracts, we observe that — light-weight static analysis tools are most efficient, tools based on verification or symbolic execution are relatively slower. In Table 7, Slither and Clairvoyance take the least time, 52 *min* and 181 *min*, respectively. Oyente is less efficient, taking 1352 *min*. Surprisingly, Securify is the least efficient, using 8859 *min*. Although Oyente supports 4 types of vulnerability and Securify supports 7 types, the analysis overheads do not increase very much even when supporting more types.

Table 7: The time (*min.*) of reentrancy vulnerability detection for each tool on 17770 contracts.

Dataset	Sli. v0.6.4	Oye. v2.0	Sec. v1.0	Clairvoyance
17770	52	1352	8859	181

Slither and Clairvoyance both depend on light-weight data/control flow analysis. Our tool has more overheads due to the support of cross-contract static taint analysis. Oyente is based on symbolic execution of EVM IR disassembled from EVM bytecode. The performance issue of Securify roots in the time-consuming conversion of IRs into Datalog format and then the verification on the datalog. To sum up, the efficiency is essentially determined by the inherent complexity of different techniques.

Answer to RQ3 : In general, Clairvoyance is efficient as a light-weight cross-contract static taint analysis tool, which avoids using verification or heavy-weight symbolic execution.

6.5 RQ4: Reproduction of DAO Attack

To confirm the vulnerabilities of TPs found by Clairvoyance, we manually inspect 101 TPs (not detected by any other static or dynamic tool) and experiment with the triggering of them. According to the hints (e.g., the vulnerable call chains) provided by Clairvoyance, we conclude that 82 out of 101 TPs could be easily triggered with high confidence, owning to two reasons: 1) no clear PPTs are found along the vulnerable call chains, indicating their feasibility; 2) the contexts of the call chains are not too complicated to understand, allowing us (or external attackers) to understand the flaws in the business logic. The remaining 19 TPs, for which we are with low confidence, have complicated logic for understanding and own complex constraints in paths that may be undecidable for static analysis. In the tool website [5], we have added the exploitation code (succeed on *our private chain*) for 20 out of the 82 TPs, and we will gradually add more. Here, we detail two interesting TPs:

Case 1 at `0x7bc51b19abe2cfb15d58f845dad027feab01bfa0`. A simple case is the contract `DividendDistributor` with 6 transaction times and a total transaction amount of 1.6 Ethers. The vulnerable call chain involves 5 functions from 2 contracts and has an internal vulnerable function `loggedTransfer()`, which proceeds the transaction with unchecked the transfer target and the amount. The low-level-call of the transfer function is tainted. We can easily trigger this reentrancy and steal money from it.

Case 2 at `0x526af336D614adE5cc252A407062B8861aF998F5`. A complicated case is the contract `SaiProxy` with 9,987 transaction times and a total transaction amount of 107,172 Ethers. The shortest vulnerable call chain involves 4 external contracts and its length is 8 functions. The function call `tub.gem().deposit.value(msg.value)()` is the low-level-call of tainted object `tub`. We can recursively call this function for causing gas exhaustion.

Answer to RQ4: Based on the vulnerable call chains reported by Clairvoyance, triggering the TPs of reentrancy is facilitated and proves the effectiveness of Clairvoyance in practice.

6.6 Discussion

6.6.1 Generality of PPTs. PPTs can be applied for other bug types, like *self-destruct abusing* [8]. For example, ignoring PPT3 (self-defined modifiers) when detecting self-destruct abusing causes FPs, since strict permission control could be done via PPT3 and make the function $selfdestruct()$ not accessible by the non-owner. Besides, PPT3 is also found to prevent the bug of *unexpected revert* [15], and tools ignoring PPT3 will inevitably yield FPs. More details on using PPTs for detecting other types of bugs are available in [5].

6.6.2 Threats to Validity. Threats to *internal validity* come from the threshold h used in the light-weight symbolic analysis. Currently, h is set to 2 for unrolling every loop (e.g., `while`) up to twice. We find that increasing the value (e.g., up to 5) does not have significant impact on our results. The reason is that not many loops are solved along the paths for the PPTs. Threats to *external validity* mainly come from the representativeness of the two datasets of smart contracts. However, there is no dataset available in previous papers [35, 49, 50]. During the past two years, we have tried our best to collected and downloaded 11714 real-world contracts from Etherscan to assess the existing tools and summarize the PPTs. To avoid the bias of using one dataset, we use the 17770 contracts in Google BigQuery Open Dataset for tools evaluation. We will release and contribute our datasets to the community.

6.6.3 Future Enhancement. Our Clairvoyance is not designed for the compositional analysis, i.e., how can we avoid redundant interprocedural analysis when a few new contracts are added into our analysis scope. The major runtime overhead for non-compositional analysis of our approach lies in that Clairvoyance needs to search through all the contract scope for its caller contracts. The larger the scope, the more overhead it will incur. In the future, we plan to investigate incremental and compositional analysis for reentrancy detection by caching and indexing the call relation among contracts. Note that, the overhead for re-analyzing for callee contract is acceptable as a contract consists of only up to a few hundred lines of code, which requires only about 0.5 second for the callee analysis of individual contract.

7 RELATED WORK

7.1 Reentrancy Detection in Smart Contracts

Recently, many security scanners have been proposed for vulnerability detection in smart contracts. Among them, Manticore [48], Mythril [13], sFuzz [21], MythX [14], Echidna [47], ContractFuzzer[24] and Ethracer [26] belong to dynamic testing/fuzzing tools. Other scanners, including Slither [49], Oyente [35, 36], SmartCheck [11, 46], Securify [10, 50], Octopus [4], Zeus [25], Maian[39] and sCompile [12], are based on static analysis. Notably, Maian and sCompile support inter-contract function call analysis and apply verification technique, but do not support reentrancy detection.

However, in static tools, only Slither, Oyente, Securify and Zeus support the detection of reentrancy vulnerability. Among them, Slither [49] serves as an analysis framework and runs the built-in reentrancy detector on the basis of intra-contract control/-data flow analysis. Oyente [35] works on the EVM IR and leverages Z3-solver [17] for constraint solving in symbolic execution. Securify [50] converts EVM bytecode into datalog representation and then applies verification on the datalog. Zeus [25] adopts XACML as a language to write the safety and fairness properties, converts them into LLVM IR [3] and applies a verification engine (e.g., SeaHorn [22]). Notably, Zeus is not included in tool comparison, as it is not open-sourced. Recently, VerX [40], an automated verifier, has been proposed to prove functional properties of Ethereum smart contracts. VerX could detect a wide range of vulnerabilities via techniques of reducing temporal property verification for reachability checking, a new symbolic execution engine, and delayed predicate abstraction. Compared with all above tools, Clairvoyance makes the first attempt at adopting PPTs for path feasibility analysis.

7.2 Interprocedural Analysis

Call graph construction is the fundamental requirement for interprocedural static analysis, which is used in many program analysis applications such as software debugging and testing [30–32, 37, 51], code recommendation [29, 33], and template-based code reuse [28, 29]. There exist many call graph construction algorithms for resolving virtual dispatches and indirect calls for traditional Java and C/C++ programs by using class hierarchy analysis [18], rapid type analysis [9], variable type analysis [45], and pointer analysis [19, 27]. Unlike conventional Java or C/C++ programs, Solidity has new and complicated language features, such as low-level calls, explicit manipulations of contract addresses via keywords (e.g., `msg.sender` and `address(this)`), fallback mechanisms and inter-contract callbacks (similar to resolving callbacks in event-driven programs such as Android apps [52, 53]). Interprocedural static taint analysis has also been studied as an instance of data-flow analysis. The analysis normally conducts reachability analysis on top of a program's data-flow graph which is either pre-computed using fast and imprecise analysis [34, 44] or being built on-the-fly in a demand-driven manner [7, 43]. Different from previous approaches on Java and C/C++, this work conducts inter-contract static taint analysis on top of smart contract programs by considering Solidity's 14 types of IR operations to support precise taint propagation in detecting reentrancy vulnerability, one of the most important types of vulnerabilities in smart contracts.

8 CONCLUSION

In this paper, we present a reentrancy detection approach based on (1) applying the light-weight cross-contract static taint analysis to find reentrancy candidates and (2) integrating the PPTs to refine the results. On the publicly collected 17770 contracts, Clairvoyance significantly outperforms the three static tools in terms of precision and recall, and two dynamic tools in terms of recall. In the future, we will extend our approach by combining with dynamic approaches for detecting more vulnerability types.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Grant No. 61972373) and partially supported by the Australian Research Council under Grant DP200101328. The research of Dr. Xue is supported by CAS Pioneer Hundred Talents Program.

REFERENCES

[1] 2015. Ethereum: Blockchain App Platform. https://www.ethereum.org/. (2015). Online; accessed 29 January 2019.

[2] 2019. A Block Explorer and Analytics Platform for Ethereum. https://etherscan.io/. (2019). Online; accessed 29 January 2019.

[3] 2019. LLVM Language Reference Manual. https://blog.sigmaprime.io/solidity-security.html. (2019). Online; accessed 29 January 2019.

[4] 2019. Octopus. https://github.com/quoscient/octopus. (2019). Online; accessed 29 January 2019.

[5] 2020. Clairvoyance:. https://toolman-demo.readthedocs.io/en/latest/index.html. (2020). Online; accessed 1 May 2020.

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

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Acm Sigplan Notices*, Vol. 49. ACM, 259–269.

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

[9] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. *ACM Sigplan Notices* 31, 10 (1996), 324–341.

[10] ChainSecurity. 2019. Securify. https://securify.chainsecurity.com/. (2019). Online; accessed 29 January 2019.

[11] ChainSecurity. 2019. Smart Check. https://tool.smartdec.net/. (2019). Online; accessed 29 January 2019.

[12] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, and Zijiang Yang. 2018. sCompile: Critical Path Identification and Analysis for Smart Contracts. *CoRR* abs/1808.00624 (2018). arXiv:1808.00624 http://arxiv.org/abs/1808.00624

[13] ConsenSys. 2019. Mythril. https://github.com/ConsenSys/mythril-classic. (2019). Online; accessed 29 January 2019.

[14] ConsenSys. 2019. MythX. https://mythx.io/. (2019). Online; accessed 29 January 2019.

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

[16] David Siegel. [n. d.]. Understanding the DAO Attack. Website. ([n. d.]). https://www.coindesk.com/understanding-dao-hack-journalists.

[17] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008*. 337–340.

[18] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.

[19] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 329–340.

[20] 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, MontrÃl'al, Canada*. to appear.

[21] GuardStrike. 2019. sFuzz: An AFL based fuzzer for smart contracts. https://fuzzing.gitbook.io/sfuzz/. (2019). Online; accessed 27 May 2019.

[22] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV 2015*. 343–361.

[23] Google Inc. 2019. Google Big Query Open Dataset. https://cloud.google.com/bigquery/public-data. (2019). Online; accessed 29 January 2019.

[24] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *ASE*. ACM, 259–269.

[25] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS 2018*.

[26] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. 363–373. https://doi.org/10.1145/3293882.3330560

[27] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *International Conference on Compiler Construction*. Springer, 153–169.

[28] Yun Lin, Guozhu Meng, Yinxing Xue, Zhenchang Xing, Jun Sun, Xin Peng, Yang Liu, Wenyun Zhao, and Jinsong Dong. 2017. Mining implicit design templates for actionable code reuse. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 394–404.

[29] Yun Lin, Xin Peng, Zhenchang Xing, Diwen Zheng, and Wenyun Zhao. 2015. Clone-based and interactive recommendation for modifying pasted code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 520–531.

[30] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering fitness gradients for interprocedural Boolean flags in search-based testing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 440–451.

[31] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. 2018. Break the dead end of dynamic slicing: localizing data and control omission bug. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 509–519.

[32] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 393–403.

[33] Yun Lin, Zhenchang Xing, Xin Peng, Yang Liu, Jun Sun, Wenyun Zhao, and Jinsong Dong. 2014. Clonepedia: Summarizing code clones by common syntactic context for software maintenance. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 341–350.

[34] V Benjamin Livshits and Monica S Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. *ACM SIGSOFT Software Engineering Notes* 28, 5 (2003), 317–326.

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

[36] melonproject. 2019. Oyente. https://github.com/melonproject/oyente. (2019). Online; accessed 29 January 2019.

[37] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the 42nd International Conference on Software Engineering*.

[38] Nick Szabo. 1996. Smart Contracts: Building Blocks for Digital Markets. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html. (1996). Online; accessed 29 January 2019.

[39] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. 653–663. https://doi.org/10.1145/3274694.3274743

[40] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE S&P 2020*.

[41] SlithIR Dev. Team. 2019. SlithIR Types. https://github.com/crytic/slither/wiki/SlithIR. (2019). Online; accessed 30 June 2019.

[42] Solidity Dev. Team. 2019. Solidity — Security Considerations. https://solidity.readthedocs.io/en/v0.5.0/security-considerations.html. (2019). Online; accessed 30 June 2019.

[43] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. ACM, 460–473.

[44] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 254–264.

[45] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. *Practical virtual method call resolution for Java*. Vol. 35. ACM.

[46] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *WETSEB@ICSE 2018*. 9–16.

[47] trailofbits. 2019. Echidna. https://github.com/trailofbits/echidna. (2019). Online; accessed 29 January 2019.

[48] trailofbits. 2019. Manticore. https://github.com/trailofbits/manticore. (2019). Online; accessed 29 January 2019.

[49] trailofbits. 2019. Slither. github. (2019). https://github.com/trailofbits/slither.

[50] 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.

[51] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering* (2019).

[52] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 89–99.

[53] Yifei Zhang, Yulei Sui, and Jingling Xue. 2018. Launch-mode-aware context-sensitive activity transition analysis. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 598–608.