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

Yinxing Xue  
University of Science and Technology of China  
Hefei, China  
yx.xue@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  
dcs.liny@nus.edu.sg

Yulei Sui  
University of Technology Sydney  
Sydney, Australia  
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-world contracts. The results show that CLAIRVOYANCE yields the best detection accuracy among all the five tools and also finds 101 unknown reentrancy vulnerabilities.

KEYWORDS

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

ACM Reference Format:

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 FN) and identifies feasible interprocedural paths by considering PPTs (to reduce FP). 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 FN and FP.
Can we conduct the light-weight path feasibility analysis, which false 7...

Existing static tools while Giventhe behavior of the possible externally-interacting contract?

denote the read and write operations, w denotes a certain public variable; and reentrancy (1) externCall?
denotes the contract tokenOnSale = TokenOnSale() // Internal Contract defined at line 8 in the same file

function combination() {
    tokenSale.buyTokensWithWei();
tokenSale.buyTokens(beneficiary);
}

contract TokenSale {
    TokenOnSale tokenOnSale;
    // External Contract

    function set(address add) {
        tokenOnSale = TokenOnSale(add);
    }

    function buyTokens(address beneficiary) {
        if (starAllocationToTokenSale > 0) {
            tokenOnSale.mint(beneficiary, tokens);
        }
    }

    function buyTokensWithWei() onlyTrader {
        wallet.transfer(weiAmount);
    }
}

Figure 2: Complicated call graph and control flow graph cross contracts—an FN for Slither, Oyente and Securify.

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:

\[
(r(var) \lor w(var)) > externCall \ encourages w(var) \Rightarrow \text{reentrancy} \quad (1)
\]

In Rule 1, r() and w() denote the read and write operations, respectively; var denotes a certain public variable; > 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 \( 4 \rightarrow 5 \rightarrow 6 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \), 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 \( 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \), and such a path in CFG to exercise the chain can never be feasible. The reason is that \( 5 \rightarrow 7 \) requires \( reEntered = true \) while \( 1 \rightarrow 5 \rightarrow 7 \) 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
Our approach consists of the following four steps: addition, we highlight the input/output in grey. As shown in Fig. 6, how a step takes some artifacts and outputs some new artifact. In Fig. 6, each rectangle represents an artifact, each ellipse represents a step (or process), each arrow 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:

### 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.

### 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 arrow 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:

#### 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 SECURe [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
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. Hence, the corresponding violation pattern is to have writes after call, as shown by the Rule 2:

\[
\text{externCall} > \text{w}(\text{var}_g) \Rightarrow \text{reentrancy} \quad (2)
\]

where \(w(\text{var}_g)\) means write operation(s) to a public variable and \(\text{externCall}\) refers to the external call for money transfer functions. Oyente supports the following Rule 3, according to [35]:

\[
\text{r}(\text{var}_g) \land (\text{gas}_{\text{trans}} > 2300) \land (\text{var}_t.a > \text{var}_d.a) \\
\land (\text{w}(\text{var}_g) > \text{externCall}) \Rightarrow \text{reentrancy} \quad (3)
\]

where \(r()\) and \(w()\) means read and write operation(s) to a public variable, \(\text{gas}_{\text{trans}} > 2300\) means the gas for transaction must be greater than 2300, the transfer amount \(\text{var}_t.a\) must be greater than the deposit amount \(\text{var}_d.a\), and the public variable \(\text{var}_g\) should be changed before the external call (denoted by \(\rangle\)). 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.

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 \(\text{auctions}[\_\text{pepeId}]\); then calls an external function via \(\text{pepeContract}.\text{cozyTime}\()); last, writes (delete instruction belongs to the general write operations) to the public variable \(\text{auctions}[\_\text{pepeId}]\). However, in reality, reentrancy cannot be triggered by external attackers due to the require check \(\text{require}(\text{address}(\text{pepeContract}) = \text{msg}.\text{sender})\) at line 3 in Fig. 7. Access control usually checks the invoker of the payment functions — checking whether the identity of \text{msg}.\text{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.

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 \(\text{rID}[\_\text{rNumber}].\text{lastTime}\) 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., 0x33...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 \(\text{onlyHuman()}\) before the potential vulnerable function \(\text{buy()}\). Since \(\text{onlyHuman()}\) restricts that the transaction can be only conducted by the address of \text{msg}.\text{sender} via the usage of keyword \text{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 FPs relying on access control or restricted addresses to prevent external malicious calls, PPT4 prevents the recursive entrance of the function — eliminating the issue

```solidity
1 contract CozyTimeAuction { 
2 function buyCozy(uint256 _pepeId, uint256 _cozyCand, 
3 address _pepeRec) public payable { 
4 require(address(pepeContract) != msg.sender); 
5 PepeAuction storage auction = auctions[_pepeId]; 
6 totalFee += _amt; // transfer price - totalFee); 
7 if(!pepeContract.cozyTime(auction.pepeId, _cozyCand, 
8 _pepeRec)) { /* external call */ 
9 } 
10 } 
11 } 
12 } 
13 } 
14 } 
15 }
```

---

2In 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).

3We confirm this rule from the implementation of Oyente.

---

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

Fig. 8: A real case of using the constant value (PPT2) for the contract address — an FP for OYENTE and Securify.

Fig. 9: A real case of using a self-defined modifier for protection (PPT3) — an FP for SLITHER, OYENTE and Securify.
```solidity
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) {
        _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.

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

<table>
<thead>
<tr>
<th>PPT</th>
<th>Slither v0.4.6</th>
<th>Oyente v0.2.7</th>
<th>Securify v1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPT1</td>
<td>7</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>PPT2</td>
<td>28</td>
<td>2</td>
<td>51</td>
</tr>
<tr>
<td>PPT3</td>
<td>15</td>
<td>3</td>
<td>46</td>
</tr>
<tr>
<td>PPT4</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>PPT5</td>
<td>0</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>PPTs/all #FP</td>
<td>67.5%</td>
<td>71.42%</td>
<td>95.15%</td>
</tr>
</tbody>
</table>

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 $r_j$. 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 $c_i$; there is also a corresponding edge $e_c \rightarrow r_j$ 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 $CP$ 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.
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 VF 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.

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 >) 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.

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

<table>
<thead>
<tr>
<th>IS</th>
<th>Rule</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) msg.sender, tx.origin</td>
<td>(1) data assignment: a(address type) = _address</td>
<td>(1) msg.sender, tx.origin</td>
</tr>
<tr>
<td>(2) parameters of public function</td>
<td>(2) address binding: object = ContractOfObject(_address)</td>
<td>(2) parameters of public function</td>
</tr>
<tr>
<td>(3) return value of functions: address/object = function()</td>
<td>(3) return value of functions: address/object = function()</td>
<td>(3) return value of functions: address/object = function()</td>
</tr>
</tbody>
</table>

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

### Table 3: The filtering patterns behind PPT1–PPT5.

<table>
<thead>
<tr>
<th>PPT</th>
<th>Filtering Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPT1</td>
<td>(isAuthorized(msg.sender) * hasPermis(msg.sender)) &gt; msg.sender.externalCall</td>
</tr>
<tr>
<td>PPT2</td>
<td>(var.pD == const * var.pD == Contract(contract)) &gt; (var.pD == contract)</td>
</tr>
<tr>
<td>PPT3</td>
<td>The above two patterns in the self-defined modifiers of functions that have external payments</td>
</tr>
<tr>
<td>PPT4</td>
<td>(isChecked(v1) &gt; w(v1) &gt; externalCall &gt; w(v1) * w(v1)1035)</td>
</tr>
<tr>
<td>PPT5</td>
<td>isChecked(V, n) &gt; w(V, n) &gt; (var.pD, transfer)</td>
</tr>
</tbody>
</table>

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:

1. 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.

2. 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.

3. 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 VF 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.

### 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 >) 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 \( v_j \) denotes the boolean variable to be used as the lock and \( \text{isChecked}(v_j) \) denotes whether \( v_j \) is checked via \texttt{require} (or \texttt{if}, assert checks etc.). \( w(v_j) \) denotes the first time of write operation to \( v_j \), and \( w'(v_j) \) denotes the second time of write operation to \( v_j \). Hence, \( w(v_j) = w'(v_j) \) means that the first and the second write operation assign the different values to \( v_j \).

The pattern for PPT5 is to check the existence of checks-effects-interactions pattern, where \( V_{\text{int}} \) denotes the set of internal variables for the current contract, \( w(V_{\text{int}}) \) refers to the write operations to \( V_{\text{int}} \) for the internal state or balance changes, and last \( \text{var}_{\text{obj}}, \text{transfer()} \) denotes the external call of the build-in function \text{transfer}(). Notably, even if \( \text{var}_{\text{obj}} \) is tainted, the code is still non-vulnerable, as the built-in function \text{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 "\( \text{reEntered} \land \text{ActualBalance} > 0.01 \land \text{reEntered} = \text{true} \)" 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.

### 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 4: The detection accuracy at function level for Clairvoyance and the other three tools on the 17770 contracts.

<table>
<thead>
<tr>
<th></th>
<th>Slith v0.6.4</th>
<th>Oyn. v0.2.7</th>
<th>Secur v1.0</th>
<th>C.V.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#TP</td>
<td>162</td>
<td>28</td>
<td>608</td>
<td>168</td>
</tr>
<tr>
<td>#FP</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>124</td>
</tr>
<tr>
<td>#N</td>
<td>159</td>
<td>24</td>
<td>605</td>
<td>44</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Tool</th>
<th>SLT v.0.6.4</th>
<th>OVE v.0.2.7</th>
<th>Sec v.1.0</th>
<th>CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPT1</td>
<td>33</td>
<td>2</td>
<td>60</td>
<td>9</td>
</tr>
<tr>
<td>PPT2</td>
<td>56</td>
<td>4</td>
<td>121</td>
<td>10</td>
</tr>
<tr>
<td>PPT3</td>
<td>25</td>
<td>15</td>
<td>193</td>
<td>0</td>
</tr>
<tr>
<td>PPT4</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PPT5</td>
<td>7</td>
<td>1</td>
<td>103</td>
<td>14</td>
</tr>
<tr>
<td>Sum</td>
<td>130</td>
<td>22</td>
<td>478</td>
<td>33</td>
</tr>
</tbody>
</table>

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         if (total_bettors <= 1) { forceVoidRace(); }  
8         else {
9             require(house_fee < address(this).balance);
10            total_reward = total_reward.sub(house_fee);
11            betContrInterface.payHouseFee.value(house_fee);  
12         }
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 PPT3). (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 PPT3 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.

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 FPs cannot be handled by static analysis. Due to page limit, more details and examples of other FPs 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
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—i.e., 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.

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.

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Sl. v0.6.4</th>
<th>Ove. v2.0</th>
<th>Sec. v1.0</th>
<th>Clairvoyance</th>
</tr>
</thead>
<tbody>
<tr>
<td>17770</td>
<td>52</td>
<td>1352</td>
<td>8859</td>
<td>181</td>
</tr>
</tbody>
</table>

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 0x7b6d1b9a6be26fb15d58f845dad027feab0bfa0. 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 logedTransfer(), 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 0x526af336d614de5e6c252a407062e886e1af98f5. 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 tube().deosit.value(msg.value()) is the low-level-call of tainted object tube. We can recursively call this function for causing gas exhaustion.

Answer to RQ4: Based on the vulnerable call chains reported by Clairvoyance, triggering the effectiveness of TPs 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 self destruct() 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 inter-procedural 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 EtherRacer [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., S-TaHorn [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 precise 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.