SmartReco: Detecting Read-Only Reentrancy via Fine-Grained Cross-DApp Analysis

Jingwen Zhang[†§], Zibin Zheng[†‖], Yuhong Nan[†‖*], Mingxi Ye[†‖], Kaiwen Ning[†§], Yu Zhang[‡§], and Weizhe Zhang[‡§]
[†]Sun Yat-sen University, {zhangjw273, yemx6, ningkw}@mail2.sysu.edu.cn, {zhzibin, nanyh}@mail.sysu.edu.cn
[‡]Harbin Institute of Technology, {yuzhang, wzzhang}@hit.edu.cn
[§]Peng Cheng Laboratory, [‖]GuangDong Engineering Technology Research Center of Blockchain

***Abstract*—Despite the increasing popularity of Decentralized Applications (DApps), they are suffering from various vulnerabilities that can be exploited by adversaries for profits. Among such vulnerabilities, Read-Only Reentrancy (called ROR in this paper), is an emerging type of vulnerability that arises from the complex interactions between DApps. In the recent three years, attack incidents of ROR have already caused around 30M USD losses to the DApp ecosystem. Existing techniques for vulnerability detection in smart contracts can hardly detect Read-Only Reentrancy attacks, due to the lack of tracking and analyzing the complex interactions between multiple DApps.**

In this paper, we propose *SmartReco*, a new framework for detecting Read-Only Reentrancy vulnerability in DApps through a novel combination of static and dynamic analysis (i.e., fuzzing) over smart contracts. The key design behind *SmartReco* is threefold: (1) *SmartReco* identifies the boundary between different DApps from the heavy-coupled cross-contract interactions. (2) *SmartReco* performs fine-grained static analysis to locate points of interest (i.e., entry functions) that may lead to ROR. (3) *SmartReco* utilizes the on-chain transaction data and performs multi-function fuzzing (i.e., the entry function and victim function) across different DApps to verify the existence of ROR. Our evaluation of a manual-labeled dataset with 45 RORs shows that *SmartReco* achieves a precision of 88.64% and a recall of 86.67%. In addition, *SmartReco* successfully detects 43 new RORs from 123 popular DApps. The total assets affected by such RORs reach around 520,000 USD.

***Index Terms*—Decentralize Application; Smart Contract; Vulnerability Detection; Program Analysis**

I. Introduction

Decentralized Applications (DApps) are applications including multiple smart contracts, which are code snippets that contain multiple functions to accomplish specific functionalities and can be executed on the blockchain. Due to the inherent financial property of DApps, the security of DApps is extremely important. For example, in recent years, the entire DApp ecosystem has suffered from various types of attacks, such as front-running [1], price manipulation [2], and access control [3] , resulting in billions of dollars in losses [4].

Reentrancy and Read-Only Reentrancy. As one of the most typical vulnerabilities in smart contracts, the reentrancy vulnerability is leveraged to manipulate global states, such as the contract's state, to make the contract's behavior inconsistent with expectations [5]. In recent years, with the increasing attention on smart contract security, reentrancy has been addressed by many previous research [6]–[9].

Read-Only Reentrancy (ROR), a new type of reentrancy vulnerability first reported in 2022 [10], is a cross-DApp attack that specifically exploits functions in different DApps' contracts. The key difference between ROR and traditional reentrancy is that ROR takes place between the smart contracts of independent DApps, while traditional reentrancy performs within the smart contract(s) of a single DApp. To exploit ROR, the adversary first manipulates a specific state of one DApp by invoking one of its function (i.e., the *entry function*). Then, the adversary performs the attack by hijacking the execution control flow and invoking another DApp's function (i.e., the *victim function*) which relies on the aforementioned state, like calculating the token price with the token supply (see Section II-B for more details). Right now, ROR has already resulted in losses exceeding 30M USD and poses a threat of more than 100M USD. Due to the invisibility and potential risks of ROR, it has been selected as one of the top ten blockchain hacking techniques in 2022 [11].

Challenges in detecting Read-Only Reentrancy. Despite a number of recent research focusing on detecting smart contract vulnerabilities, detecting ROR based on existing techniques is by no means trivial. Specifically, there are three fundamental challenges as we elaborate below.

Firstly, identifying the boundaries between DApps is difficult. Specifically, as ROR is a cross-DApp attack, detecting ROR requires analyzing complex relationships between DApps. However, the anonymity of the blockchain prevents access to smart contract identification, such as which DApp a smart contract belongs to. Prior research [6], [12], [13] simply look up the contract address from the DApp's official website, or query third-party API (e.g., DAppRadar [14] and Thegraph [15]). However, this information is often incomplete, as new smart contracts included by the DApp are not updated in a timely fashion.

Secondly, identifying the entry function (the start point) and the victim function (the end point) of ROR is quite challenging, given the unlimited number of DApps and their publicly accessible functions. Specifically, a function within a DApp can be called by anyone and any other DApp, resulting in a large search space. Moreover, these two functions do not

* Yuhong Nan is the corresponding author.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00057

share any explicit dependency (e.g., a shared state), or within the same call chain. Existing methods such as IcyChecker [6] and DeFiTainter [16] blindly consider all functions as the candidates, which is not feasible for detecting ROR due to the huge exploration space.

Lastly, even with the given entry function and victim function pairs, it is still difficult to find a valid path (call chain) that can trigger ROR. On the one hand, static analysis methods, such as Pluto [17] and Sailfish [18], can not recover the call chain of ROR, due to the lack of concrete runtime contextual information. On the other hand, fuzzing-based methods, such as ConFuzzius [7] and ContractFuzzer [19], randomly generate fuzzing seeds for multi-function (e.g., entry function and victim function) in different DApps. However, these approaches can hardly simulate the actual cross-DApp context (e.g., simultaneously bypassing the internal checks in both DApps), not to mention triggering the potential RORs.

Our work. In this paper, we propose *SmartReco*, a new framework to detect Read-Only Reentrancy vulnerabilities in smart contracts. To address the aforementioned challenges, *SmartReco* comes with the following three unique designs: (1) cross-DApp analysis to identify DApp boundaries, (2) fine-grained static analysis for locating potential entry functions, (3) multi-function fuzzing across different DApps for effectively verifying RORs.

More specifically, the cross-DApp analysis aims to incorporate accurate DApp-based contextual data to identify valid attack surfaces. Here, the DApp-based contextual data refers to records of the interactions between DApps during transaction execution, such as cross-DApp call and cross-DApp state read and write. To facilitate this analysis, we propose an enhanced DApp boundary identification method based on DApp (contract) builders (Section IV-A). Our observation is that a DApp is usually managed by a fixed group of accounts as the builders, allowing us to more accurately identify DApp boundaries. With this method, we build the first DApp builder dataset $D_{builder}$ (with 334 unique builders). To the best of our knowledge, this is the first approach and dataset for accurately classifying smart contracts for DApps.

Based on the collected DApp information, *SmartReco* performs fine-grained static analysis to find potential entry functions. Although there is no explicit dependency between the entry points and the victim points of ROR, the entry points must be associated with specific cross-DApp invocations along the call chain of a ROR. Therefore, with DApp-based contextual data, *SmartReco* analyzes all cross-DApp function invocations on the call chain and identifies potential ROR entry points. (Section IV-B and Section IV-C).

Then, to verify whether the potential ROR entry points can trigger ROR, *SmartReco* adopts a multi-function fuzzing mechanism to capture more accurate context information. To detail, *SmartReco* generates inputs for functions in different DApps based on historical transactions, as such transactions provide a more realistic scenario for finding the valid paths and contexts to invoke ROR (Section IV-D).

To evaluate the effectiveness of *SmartReco*, we construct a ROR dataset with 45 attack instances based on publicly available attack reports. We manually check and label the contracts (functions) related to these attack incidents. To the best of our knowledge, the dataset covers all reported RORs related to smart contracts as of Mar. 2024 (see Section V-A for more details). The evaluation results show that *SmartReco* has a precision of 88.64% and a recall of 86.67% over this dataset. In the meantime, two state-of-the-art frameworks, ityFuzz [20] and Sailfish [18], can not detect any RORs in the dataset. Besides, with the help of *SmartReco*, we perform an in-the-wild ROR inspection over 123 most popular DApps (with 2,676 smart contracts). Indeed, *SmartReco* successfully detects 43 unreported RORs. The vulnerabilities discovered by *SmartReco* impact approximately 520,000 USD.

To promote smart contract security development, we have open-sourced *SmartReco* and the datasets used in our work [1]. In summary, this paper makes the following contributions:

- We propose *SmartReco*, a novel framework for detecting ROR - an emerging type of reentrancy vulnerabilities in smart contracts. To the best of our knowledge, *SmartReco* is the first research to detect Read-Only Reentrancy.
- We design a series of new mechanisms, such as identifying DApp boundaries, and collecting DApp-based contextual data, to facilitate fine-grained cross-DApp analysis.
- We perform extensive experiments to verify the effectiveness of *SmartReco*. The results indicate that *SmartReco* can detect new RORs while maintaining lower false positives and false negatives.
- We release the artifact and the datasets of *SmartReco*.

The rest of the paper is organized as follows: Section II provides the background of ROR and the motivation of our research. Section III highlights the challenges and solutions for ROR detection. Section IV elaborates on the implementations of *SmartReco*. Section V introduces the experimental setup and evaluation. Section VI gives more discussions about ROR and the limitations of *SmartReco*. Section VII presents related work and Section VIII concludes the paper.

II. Background and Motivation

A. Smart Contract and Contract Execution

Decentralized Application (DApp) typically consists of a UI frontend and a backend that uses smart contracts to store and process data [21]. Smart contracts are codes that can be executed on blockchain platforms (such as Ethereum [22] and BSC Chain [23]) and contain multiple functions to accomplish specific functionalities, e.g., transferring funds. To execute smart contracts, it is necessary to first compile the smart contract into bytecode and deploy it on the blockchain. In blockchain, there are two roles: user (called externally owned account, EOA) and smart contract, and they are both distinguished by a unique identifier called address.

Users and smart contracts can send external and internal transactions respectively to a contract address to invoke functions and the contract is responsible for execution. To invoke

[1] https://github.com/zzz-sysu/smartReco

```
contract Pool{
  // The victim function
  function decrease(uint amount, address asset)
      public nonReentrant  {
    require(allow(msg.sender), "Wrong user!");
    // Calculate balance based on wrong price
    uint256 balance = oracle.getPrice(asset) *
        amount;
    oracle.doHardWork(msg.sender);
    return balance;
  }
}
contract Oracle{
  function getPrice(address asset) public view {
    require(exist(asset), "Wrong Asset!");
    Vault vault = vaults[asset];
    uint (balance, totalToken) = vault.getFunds(
        asset);
    return balance / totalToken
  }
  function doHardWork(address account) public {
    if (account == owner) {
        ...
    }
  }
}
```

(a) The victim DApp.

```
contract Vault{
  // The entry function
  function exitVault() public nonReentrant {
    require(allow(msg.sender), "Wrong user!");
    ...
    updateTokens(msg.sender);
    // Control flow transfers to attacker
    transferBalance(msg.sender);
    updateBalances(msg.sender);
    rate = balance / totalToken;
  }
  function swap(address pool) public nonReentrant{
    uint cur_rate = getRate();
    ...
  }
  function setRate(uint newRate) public onlyOwner{
    rate = newRate;
  }
  // The manipulable function
  function getFunds() public view{
    // Return outdated values
    return (balance, totalToken);
  }
  function getRate() public view{
    return rate;
  }
}
```

(b) The entry DApp.

Fig. 1: An example of Read-Only Reentrancy.

internal transactions, the control of the execution is transferred to the relevant contract, allowing the contract to execute the code and update the state.

B. Reentrancy and Read-Only Reentrancy

Reentrancy. In smart contracts, reentrancy is a specific type of vulnerability that exploits unsynchronized updated data [24]. Specifically, due to the serialized execution mechanism of smart contracts, there may exist an inconsistency between the global states. For example, when one contract calls another contract, the states of the calling contract do not fully update before control transfer, like not updating the token supply before transferring funds. Attackers can exploit this inconsistency by reentering contracts and making a profit.

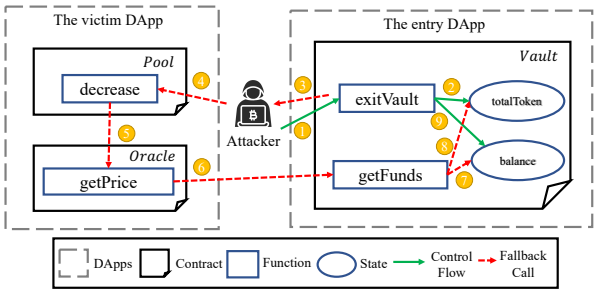


Fig. 2: The attack process of example in Fig. 1.

In smart contracts, there are three types of functions that are closely related to reentrancy attacks, as explained below:

- ***Entry Function***. The entry function is the function where the attacker initializes reentrancy. Entry functions are publicly accessible.
- ***Victim Function***. The victim function is the target function of the reentrancy attack. For each reentrancy attack, the victim function suffers the actual damage such as economic loss.
- ***Manipulable Function***. Manipulable functions are functions that can be controlled by the attacker to achieve a successful reentrancy attack. For example, a function that returns specific token prices as attackers expect.

Read-Only Reentrancy. With the increasing complexity of DApp functionalities, such as token swap [25], lending [26], and collateral [27], different DApps may interact with each other for data access [28]. However, such a deep integration and heavy interactions bring new attack surfaces to DApps, raising significant challenges to DApp security. For example, if the state updates between DApps are not well synchronized, it may lead to potential state inconsistencies in different DApps and further cause attacks such as ROR. Usually, in traditional reentrancy, the victim function and manipulable function are in the same DApp, while the victim function of ROR is in another DApp. This means that the search space for ROR is broader, making it more challenging to detect.

C. Motivating example

Fig. 1 shows a simplified code snippet of three smart contracts in two DApps, victim DApp and entry DApp. The victim DApp in Fig. 1(a) has a contract *Pool*. It allows users to withdraw balances by executing the victim function *decrease*. Although both DApps have added checks, both line 4 in Fig. 1(a) and Fig. 1(b), to ensure security, problems arise when they interact with each other. The attack process is shown in Fig. 2. An attacker can first call the entry function *exitVault* in the entry DApp. During the execution, the control flow

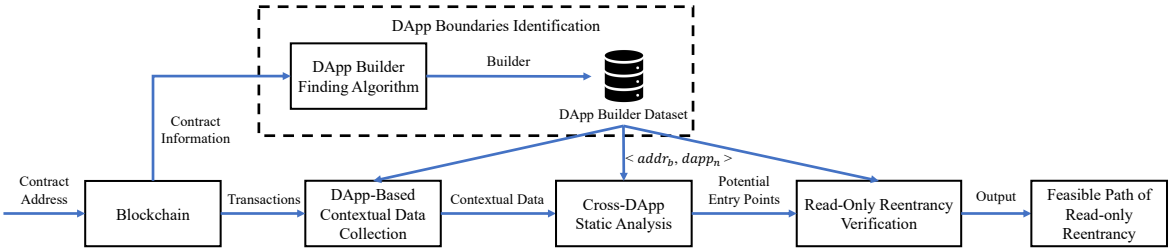


Fig. 3: The workflow of *SmartReco*.

of *exitVault* transfers to the attacker while states not fully update (steps 2 and 3 in Fig. 2), causing a temporary mismatch between token amount and balance. Then, instead of reentering the entry DApp, the attacker executes *decrease* in the victim DApp (step 4 in Fig. 2) and the price is obtained through the manipulable function *getFunds* of the entry DApp (step 6 in Fig. 2). However, since state *balance* has not been updated at this point, *decrease* will get an incorrect price. As a result, attackers can get more assets.

Limitations of existing works. Existing methods (both static and dynamic methods) can not effectively identify ROR due to the following unique challenges.

• *Identifying DApp boundaries for contracts in ROR.* Current contracts' DApp information collection methods [6], [12], [13] are inaccurate. For example, when analyzing *decrease* in Fig. 1(a), contracts and functions under the same DApp, such as contract *Oracle* and function *doHardWork* may be introduced. As a result, the search space becomes large.

• *Finding entry functions of ROR.* Existing methods for detecting reentrancy [9], [29] cannot trace DApp data, which makes it difficult to identify potential entry functions of ROR. In our example, all functions in contract *Vault* need to be analyzed, causing significant performance overhead and false positives.

• *Verifying the presence of ROR.* Due to the lack of runtime contextual information, existing static analysis techniques such as DeFiTainter [16] can hardly detect and verify RORs. In the meantime, state-of-the-art fuzzing approaches [20], [30] have proved effective for vulnerability detection in smart contracts. Unfortunately, these approaches are still inadequate for detecting RORs as their randomly generated fuzzing inputs can hardly trigger RORs with high complexity. For example, it is with little chance to generate valid fuzzing inputs that can bypass line 4 in Fig. 1(a) and line 4 in Fig. 1(b) simultaneously.

Our solution. *SmartReco* uses cross-DApp analysis to guide the detection of RORs. Turning to the example in Fig. 1, *SmartReco* first collects and replays the transactions of *decrease*. During the replay, *SmartReco* records DApp-based contextual information, such as DApps of called contracts. Then *SmartReco* finds out manipulable functions, like *getFunds*. Such manipulable functions are the key to uncover the entry functions of ROR due to the shared state dependencies with entry functions. For example, since function *exitVault* modifies the state *balance* that *getFunds* relies on, *SmartReco* regards *exitVault* as a potential entry point. *SmartReco* then performs multi-function fuzzing for *exitVault* and *decrease*. In detail, *SmartReco* tries to reenter *decrease* during the execution of *exitVault*. When *SmartReco* finds a reachable path, it reports relevant inputs and functions and explains how an attacker can initialize ROR. Turning to this example, *SmartReco* will report that an attacker can hijack control flow during executing *exitVault* and reenter *decrease*.

III. Design of *SmartReco*

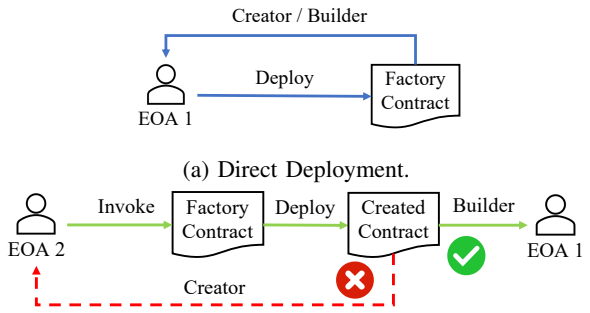
A. Technical Challenges and Our Idea

Identifying DApp boundaries. Due to the anonymity of the blockchain, smart contracts do not inherently contain information about the DApp it belongs to. To identify DApp boundaries, a straightforward approach is to collect the contract addresses of each DApp from the official websites or third-party APIs (e.g., DAppRadar [14] and Thegraph [15]). However, the DApp information from these sources is mostly outdated and inaccurate, as a large number of new smart contracts are created and deployed every day.

To address this, *SmartReco* identifies the DApp boundaries (i.e., DApp builders) based on creators of various smart contracts. Compared to the contract-DApp mapping information scattered in various sources, information about DApp builders (contract creators) is better documented and easier to access. Therefore, the DApp builder information is a more reliable source to identify the DApp boundary.

Finding potential entry functions. As mentioned earlier, the entry function and the victim function of ROR have no direct relationship. For example, we can not find a call chain from function *decrease* to function *exitVault*, and vice versa. To find potential entry functions, the simplest approach is to traverse all functions and contracts involved in the execution path. However, this is rather impractical due to the large exploration space, with a significant number of false positives. Existing tools [19], [31] can not detect ROR due to the ignorance of these important factors.

To narrow down the search space, *SmartReco* employs a two-step filtering process to locate entry functions. The key insight here is that while the entry point of ROR is not directly related to the victim function, it must be associated with manipulable functions in the call chain. For example, in Fig. 1, the entry function *exitVault* updates states that the manipulable function *getFunds* depends on. To this end,



(a) Direct Deployment.

(b) Indirect Deployment.

Fig. 4: Two methods for deploying smart contracts.

SmartReco first points out all functions of other DApps in the call chain. Then, only functions that share state dependency with those manipulable functions are considered as potential entry functions of ROR. To achieve this, *SmartReco* collects DApp-based contextual data, including the state changes and invoking of each DApp, and comprehends the entire call chain. These information allows *SmartReco* to further identify all manipulable functions and the candidate ROR entries. We will give more details about this process in Section IV-B and IV-C

Verifying the existence of ROR. As mentioned earlier, the last challenge lies in how to provide valid fuzzing inputs that can precisely trigger the critical path of ROR across two DApps. More specifically, with random input generation mechanisms as in prior works [20], [32], it is hard to satisfy the internal check in both DApps. Even if we bypass the check, there is no guarantee that the input can trigger the ROR path (line 8 in Fig. 1(b)).

To generate valid inputs for both potential entry functions and victim functions, *SmartReco* utilizes their corresponding on-chain historical transactions to detect ROR. This is because on-chain transaction data, such as the sender, timestamp, or inputs, provides realistic execution environments for fuzzing. Compared with randomly generating such fuzzing inputs as in prior works, the on-chain data allows *SmartReco* to bypass the internal checks, such as *require* statements in different functions. In this way, our fuzzing mechanism can trigger a more complex call chain between the entry function and the victim function. Besides, the successful execution of the combined transactions indicates the existence of the vulnerable path of ROR. More details of this fuzzing process are elaborated in Section IV-D.

B. Workflow of SmartReco

Fig. 3 shows the workflow of *SmartReco*. Specifically, given an unknown contract address as input, *SmartReco* outputs its corresponding ROR detection results along with the valid ROR paths if exist.

In DApp Boundaries Identification, to identify which DApp the contract belongs to, *SmartReco* retrieves a set of information related to contract creation from the blockchain, including the deployment transaction, internal transaction list

Algorithm 1 DApp Builder Finding Algorithm.

Input: Contract address $addr_s$
Output: Contract builder $builder$
1: Tx_D := *fetchDeploymenTx*($addr_s$);
2: $creator$:= *fetchCreator*($addr_s$);
3: L_D := *fetchInternalTxList*(Tx_D);
4: **while** *isNotEmpty*(L_D) && *contain*(L_D, $addr_s$) **do**
5: $addr_{in}$:= *fetchFactoryContract*($addr_s$);
6: Tx_D := *fetchDeploymenTx*($addr_{in}$);
7: $creator$:= *fetchCreator*($addr_{in}$);
8: L_D := *fetchInternalTxList*(Tx_D);
9: **end while**
10: $builder$:= $creator$;
11: **return** $builder$

of deployment transaction, and the transaction sender. The above information helps *SmartReco* to get the actual builder of the contract, and further identify the DApp it actually belongs to. We will detail this process in Section IV-A.

In DApp-based Contextual Data Collection, *SmartReco* collects the contract's historical transactions, faithfully replays them, and filters DApp-based contextual data by DApp boundaries. In Cross-DApp Static Analysis, *SmartReco* uses contextual data to extract potential entry points (entry functions) of RORs. In Read-Only Reentrancy Verification, *SmartReco* employs a multi-function fuzzing strategy to verify the existence of ROR in potential entry points and output feasible ROR paths if exist.

IV. Approach Details

A. DApp Boundaries Identification

SmartReco uses DApp builders to identify DApp boundaries. Note that due to the unique DApp development mode, the builder of a given DApp cannot simply attributed to the creator(s) of the smart contracts it contains. As the examples shown in Fig. 4, there are two ways to deploy smart contracts, namely *direct deployment* and *indirect deployment*. For direct deployment (Fig. 4(a)), users can deploy a new contract by directly sending a transaction to the blockchain. In this case, the builder of contract *Factory* is its creator `EOA1`. Differently, a more complicated case is the scenario of *indirect deployment*. As shown in Fig. 4(b), users can send a transaction to a contract (i.e., *Factory Contract*) that further creates another new contract (*Created Contract*). Since in smart contract, a contract's creator is always recorded as the account that invokes the transaction, simply using this information from transaction records can easily cause misclassification. In this case, the actual builder of *Created Contract* should be *EOA1*, not its creator *EOA2*.

DApp builder identification. To identify the builder of a given contract, *SmartReco* employs a heuristic-based DApp Builder Finding (DBF) algorithm, as shown in Algorithm 1. The core idea here is to exhaustively find the actual builder of the contract, based on various information related to contract creation. In detail, to find the builder $builder$ of a contract $addr_s$, *SmartReco* first fetches its deployment transaction Tx_D, the first transaction of the contract, from the blockchain

and searches for the creator $creator$ in Tx_D. Then *SmartReco* analyzes how Tx_D is deployed by checking the internal transaction list L_D, which records all internal transactions executing in Tx_D. When $addr_s$ is created by indirect deployment, there will be an internal creation transaction related to $addr_s$ in L_D. Thus, if L_D is empty or *SmartReco* can not find $addr_s$ in L_D, it means $addr_s$ is created by direct deployment and $builder$ is $creator$. Otherwise, *SmartReco* gets the internal creator $addr_{in}$, which means $addr_s$ is deployed by another smart contract $addr_{in}$ and they belong to the same DApp. Then, *SmartReco* checks Tx_D, $creator$, and L_D of $addr_{in}$ iteratively until L_D is empty or $addr_{in}$ not exists in L_D. In this way, *SmartReco* finally obtains $builder$ of $addr_s$.

We use the examples in Fig. 4 to illustrate this process. For *Factory Contract* in Fig. 4(a), *SmartReco* can find its $creator$ is `EOA1`. As *Factory Contract* is directly deployed, its L_D is empty. Thus, *SmartReco* can verify that `EOA1` is the builder of *Factory Contract*. For *Created Contract* in Fig. 4(b), its $creator$ is `EOA2`. As *Created Contract* is deployed indirectly, its L_D is not empty and *SmartReco* can find the internal builder $addr_{in}$ of *Created Contract* is *Factory Contract*. In this case, *SmartReco* considers both of the two contracts to belong to the same DApp. Therefore, *SmartReco* investigates how *Factory Contract* is deployed. Since *Factory Contract* is directly deployed by `EOA1`, *SmartReco* can determine that the builder of the *Created Contract* is `EOA1` instead of `EOA2`.

Note that, *SmartReco* may fail to identify some contracts if their DApp builders are totally absent from $D_{builder}$. This could potentially lead to inaccuracies in boundary identification. However, this will not diminish the effectiveness of *SmartReco* in detecting ROR. Specifically, *SmartReco* employs a conservative approach to address this potential misinformation. In *SmartReco*, all contracts whose builders are not listed in $D_{builder}$ will be deemed unsafe, and these contracts will then undergo further security analysis.

B. DApp-based Contextual Data Collection

In this step, *SmartReco* replays transactions as in prior works [6], [33], [34] to obtain DApp-based contextual information such as contract address and state read and write. The difference is that *SmartReco* replays at transaction level, while others perform based on synchronizing blockchain state. The advantage is that *SmartReco* does not require a significant amount of time and storage to gain the entire blockchain state and can be easily extended to other blockchain platforms like BNB Chain [23]. More specifically, during the replay of a transaction, whenever *SmartReco* needs to invoke another contract $addr_s$, it will first identify the DApp information $dapp_n$ of this contract and record the $operation$ executed in $addr_s$ in a three-tuple $<<addr_s, dapp_n>, operation>$. For example, when *SmartReco* meets opcodes related to state read and write, such as *SLOAD* and *SSTORE*, *SmartReco* records *read* and *write* in $operation$ respectively. Similarly, *SmartReco* records invocation information whenever it encounters opcodes related to method execution, such as *CALL*, *CALLCODE*, *DELEGATECALL*, and *STATICCALL*. The $operation$ here is *invoke*. We will demonstrate that replay at transaction level is feasible in Section V-C.

C. Cross-DApp Static Analysis

This step comprises two tasks: (1) prioritizing manipulable functions from a given call chain for efficiency, and (2) identifying the set of potential entry functions in other DApps based on the selected manipulable functions.

Manipulable function prioritization. Given the huge amount of manipulable functions in cross-DApp interactions, *SmartReco* needs to prioritize their order as it can significantly improve detection efficiency. Thus, *SmartReco* introduces a metric called *importance* to order the manipulable functions. Here, *importance* is calculated based on the sum of contextual data (i.e., invoke, read, and write) of each function in the call chain. Our main idea is that if a function frequently appears or performs many operations in historical transactions replay, it is more likely to provide high-quality contextual information (i.e., state dependency) that is critical to find ROR entry functions. The calculation formula is as follows:

$$importance = C_{invoke} + C_{read} + C_{write} \tag{1}$$

where C represents the count.

To this end, *SmartReco* generates the inter-DApp data flow graph based on DApp-based contextual data. The graph records the operations in each manipulable function, such as *call*, *read*, and *write*. Then, *SmartReco* orders these manipulable functions based on their *importance* in descending order.

Potential entry functions determination. Based on the characteristics of real-world ROR attacks, we have summarized four rules (shown in Fig. 5) for constructing an intra-DApp graph and identifying potential entry functions. To the best of our knowledge, these rules are relatively comprehensive, as they sufficiently cover all known ROR attacks (8 in total). Meanwhile, it is practical to incorporate new rules into *SmartReco* should new varieties of ROR attacks emerge. Below, we use contracts in Fig. 1 to show how *SmartReco* constructs the intra-DApp graph.

- **Implicit dependency expanding.** When there are two functions in a contract, where one modifies the state and the other reads the state, *SmartReco* regards these two functions have an implicit dependency and adds an edge between them. For example, in Fig. 5(a), *exitVault* modifies the state *balance*, while *getFunds* reads *balance*. *SmartReco* will add an implicit dependency from *getFunds* to *exitVault*.
- **Access control pruning.** Based on access control, *SmartReco* performs implicit dependency pruning. More specifically, if the modification of states within functions is protected by access control, like *setRate* in Fig. 5(b), *SmartReco* does not consider it as an unsafe function and prunes all implicit dependency edges that depend on it, like *swap* in the example.
- **NonReentrant pruning.** Although modifier *nonReentrant* is ineffective in defending against ROR. However, within the same contract, *nonReentrant* is feasible.

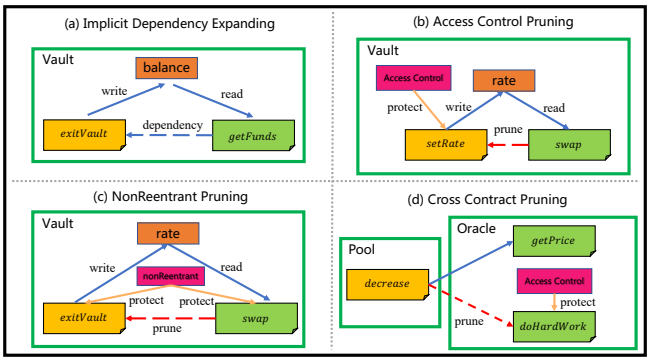


Fig. 5: Rules for constructing intra-DApp graph.

Specifically, when two functions in the same contract are both protected by *nonReentrant*, like *exitVault* and *swap* in Fig. 5(c), even if they have a relationship on the same state, they are unable to cause ROR. Therefore, it is safe to remove the dependency edge between them.

- **Cross contract pruning.** As mentioned before, ROR is a cross-DApp attack. For functions in the same DApp with sufficient access control, *SmartReco* considers them as safe functions and prunes their corresponding execution paths. As shown in Fig. 5(d), since *doHardWork* in contract *Oracle* is protected by access control, line 19 in Fig. 1 (a), *SmartReco* prunes *doHardWork* in *decrease*.

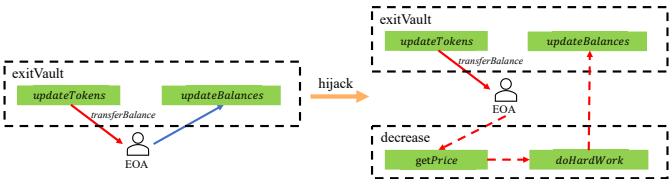
Based on the above rules, *SmartReco* can construct an intra-DApp control flow graph. More specifically, for any manipulable function, all endpoints of implicit dependency edges have the potential to become an entry point and need further testing, like *exitVault* in Fig. 1(b).

D. Read-Only Reentrancy Verification

To verify ROR, *SmartReco* generates valid inputs for entry functions and attempts to trigger ROR by performing control flow hijacking.

Inputs generation. For each potential entry function fun, *SmartReco* attempts to generate its valid input during the replay of a transaction tx_o extracted from the detected contract. In detail, if there is no historical transaction of fun, *SmartReco* will use the same environment in tx_o, such as the same caller, to construct a transaction of fun. More specifically, *SmartReco* will generate transaction tx_{fun} based on the ABI of fun and add it to candidate list l_c. Otherwise, if historical transaction tx_{fun} of fun is fetched from the blockchain like Ethereum [22], *SmartReco* pushes it into the l_c. Since most normal transactions may not trigger the critical path of ROR, *SmartReco* mutates tx_{fun} using the following two strategies:

- **Fuzzing funds**. If fun is *payable*, then it can receive funds, like ETH, and should have complete logic to handle the assets involved in the transaction. Therefore, *SmartReco* randomly changes funds values in tx_{fun}, gets a new transaction, and adds it to l_c.



(a) execution path of tx. (b) execution path of (tx,tx_o).

Fig. 6: The process of verifying potential entry function for contracts in Fig. 1.

- **Fuzzing input.** If fun can accept parameters, to explore unusual paths, *SmartReco* will attempt to mutate them randomly based on the original inputs in tx_{fun}, get a new transaction, and add it to l_c.

ROR verification. For each transaction tx in l_c, *SmartReco* traces the execution of tx and attempts to hijack its control flow. In detail, whenever tx interacts with an external parameter (e.g., an arbitrary contract address), *SmartReco* tries to invoke tx_o. If tx_o executes successfully, *SmartReco* continues to track the execution of tx and monitors state modifications, e.g., updating balances. If tx updates states that tx_o has read, *SmartReco* considers the presence of ROR and reports it.

We use contracts in Fig. 1 as an example. During replaying the transaction tx_o of *decrease*, *SmartReco* identifies *exitVault* as a potential entry function. *SmartReco* then generates the transaction tx_{fun} for *exitVault*, mutates tx_{fun} to create candidate list l_c and analyzes each transaction tx in l_c. Fig. 6(a) shows the original execution path of tx. Specifically, when tx executes to line 8 in Fig. 1(b), the execution control is handed over to `EOA` to perform the transfer. *SmartReco* then simulates an attacker hijacking the control flow and executes tx_o, as shown in Fig. 6(b). Since tx_o executes successfully and tx modifies the state *balance* that tx_o depends on after executing tx_o, *SmartReco* will report the presence of ROR.

V. Evaluation

In this section, we first present the dataset used in the evaluation and introduce our experimental setup of *SmartReco*. Then we show the evaluation results of *SmartReco*.

A. Implementation and Evaluation Setup

Dataset. We construct the following three datasets to perform our evaluation.

- **Manual-labeled ROR Dataset** ($D_{labeled}$). This dataset contains 45 ROR vulnerabilities from 25 contracts through a carefully designed manual-labeling process. Particularly, 14 RORs from 8 contracts are collected from publicly reported attack incidents, while the remaining 31 RORs from 17 contracts all written in Solidity [35] are manually discovered by our security experts. More details are shown in Section VI-B. To the best of our knowledge, this is the most comprehensive and up-to-date ROR dataset.

The manual labeling process is performed by three researchers, including one professor and two senior PhD candidates. Each researcher has 2+ years of experience in auditing

smart contract vulnerabilities. To ensure the quality of the labeled dataset, we first give a detailed tutorial about ROR to researchers, helping them better understand the nature of RORs. Then, we collect all the 8 publicly reported ROR attacks, and extract their entry functions and manipulable functions. We then ask researchers to independently extract the attack patterns (i.e., call chains) from these RORs. Meanwhile, to construct a potential ROR set, we use a similarity-matching approach to search for functions that call manipulable functions in all historical transactions. Finally, we collect 67 suspicious functions from 36 contracts. Based on the extracted ROR patterns, we attempt to construct call chains from the entry functions to functions in the set with the multi-function fuzzing method. Subsequently, the experts independently verify these cases and align disagreements together. Note that during the whole labeling process, only the vulnerabilities confirmed by all three researchers are considered as RORs. While this criterion is relatively conservative, our labeled dataset provides a lower bound for evaluating the effectiveness of *SmartReco*, excluding potential FPs caused by data labeling.

- **Popular DApp Dataset ($D_{unknown}$).** To further understand the impact of ROR in real-world DApps, we construct an unknown dataset ($D_{unknown}$) consisting of 2,676 smart contracts from 123 popular DApps. More specifically, this dataset is a union of public-available, top-popular DApp data released by two prior research (Stefan et al. [12] and IcyChecker [6]). The identifications of 2,676 smart contracts are determined based on the identified DApp boundaries by *SmartReco*. Since all contracts we analyzed are from popular DApps, we believe the analysis results are sufficiently representative of the smart contract ecosystem.
- **DApp builder dataset ($D_{builder}$).** We use all DApps in $D_{unknown}$ to construct the builder dataset with the DBF algorithm described in Section IV-A. All data in $D_{builder}$ are recorded in a tuple $<builder,\ dapp_n>$, consisting of each unique builder address and DApp(s) it belongs to. Note that for DApps belonging to the same project, if these DApps share the same builder, we will merge these DApps and their builder as a single item in $D_{builder}$, as they are mutually trusted. Finally, our $D_{builder}$ consists of a total number of 334 builders and their corresponding DApps. To this end, given the builder of an unknown contact, $D_{builder}$ can tell which DApp the contract exactly belongs to.

Implementation. We implement *SmartReco* with around 1,700 lines of code in Python and about 2,300 lines of code in Rust. *SmartReco* replays historical transactions and verifies RORs based on ityFuzz [20], which is an online fuzzing-based framework for smart contracts. *SmartReco* finds potential entry functions based on Slither [36]. All experiments in our evaluation are conducted on a machine with two Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz, 512GB RAM, and Ubuntu 20.04.4 OS.

Evaluation setup. We inspect the latest 1000 transactions for each contract and use the top 300 transactions of each potential entry function to test and manually analyze the reported results. If *SmartReco* needs data during execution, such as reading storage, it directly fetches the corresponding storage from the blockchain and caches it locally. Besides, like other transaction-based frameworks (e.g., sFuzz [32] and Smartian [31]), *SmartReco* implements a customized Ethereum Virtual Machine (EVM) [22] to execute transactions, monitor the opcodes, and record data.

TABLE I: Overall effectiveness of *SmartReco* on $D_{labeled}$.

D_{label}	Recall			Precision		
	TP	FN	Rate	TP	FP	Rate
Attack incidents	8	6	57.14%	8	0	100%
Manual annotation	31	0	100%	31	5	86.11%
Total	39	6	86.67%	39	5	88.64%

Evaluation Metrics. Specifically, we focus on the following research questions.

RQ1. How effective is *SmartReco* in detecting ROR?
RQ2. What is the impact of each module in *SmartReco* on detecting ROR?
RQ3. Is *SmartReco* more effective in detecting ROR compared to other advanced tools?
RQ4. Can *SmartReco* effectively detect unknown ROR in real-world smart contracts?
RQ5. What is the performance overhead of *SmartReco* for detecting ROR?

B. Effectiveness of SmartReco

To answer RQ1, we use $D_{labeled}$ to test *SmartReco*, and the results are shown in Table I. In detail, *SmartReco* successfully detects 39 out of 45 RORs with a precision of 88.64% and a recall of 86.67%. In addition, thanks to the fine-grained cross-DApp analysis, *SmartReco* successfully identifies 4 attack contracts out of the 8 attack incidents and accurately pinpoints the specific attack paths.

False Negatives. Five out of six false negatives are all written in Vyper [37], which is outside of our scope. Due to *SmartReco* relying on Slither, *SmartReco* cannot detect such contracts. In fact, there are currently no stable static analysis tools specifically for Vyper [16]. If the state dependencies of these contracts are provided, *SmartReco* can determine the dependency relationship between functions and identify potential entry functions for analysis. Another false negative is due to the use of a modifier that has similar logic to *onlyOwner*, causing *SmartReco* to mistakenly assume that the function does not need to be analyzed.

False Positives. After analyzing the false positive cases, we identify the main reasons as follows: (1) When searching potential entry functions, *SmartReco* primarily focuses on checking modifiers because most smart contracts use modifiers for control. However, some contracts do not follow these patterns and instead perform checks within the function, resulting in *SmartReco* failing to recognize them and causing false positives. (2) Some DApps have forwarding modules, such as the Gnosis multi-signature wallet. It helps users to forward and execute operations directly. However, *SmartReco*

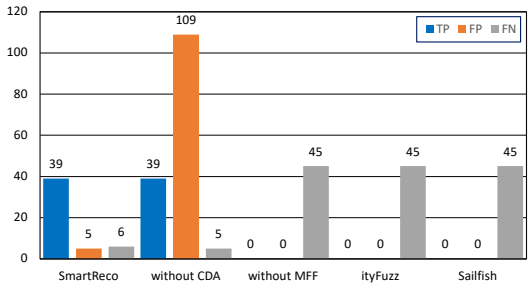


Fig. 7: Result of *SmartReco*, *SmartReco* without cross-DApp analysis (without CDA), *SmartReco* without multi-function fuzzing (without MFF), ityFuzz and Sailfish based on $D_{labeled}$.

currently considers only interactions in the same DApp as safe and thus leads to false positives.

Answer to RQ 1:

SmartReco demonstrates its effectiveness in ROR detection with a precision of 88.64% and a recall of 86.67% on the manually-labeled dataset $D_{labeled}$.

C. Effectiveness of Each Module in SmartReco

To answer RQ2, we evaluate the effectiveness of DApp boundaries identification method with top-10 DApps in $D_{unknown}$. Besides, we verify the effectiveness of the transaction-based replay method with $D_{unknown}$. We also conduct ablation experiments on cross-DApp analysis and multi-function fuzzing methods based on $D_{labeled}$ separately.

Effectiveness of DApp boundaries identification. To verify whether the newly created contracts will cause the DApp boundaries identification method to fail, we select the top-10 DApps from $D_{unknown}$, as top DApps provide more timely and comprehensive maintenance of contract information. Specifically, we collect the latest contracts of all these DApps from their official websites, totaling 545 contracts. Then, we use the DBF algorithm to find the builders of these contracts, determine whether these builders are in the $D_{builder}$, and whether the corresponding builders are classified correctly. The results show that the DApp boundary identification method accurately recognizes 489 contracts, reaching a precision of 89.72%. The remaining 56 contracts are due to the contract builders not being included in $D_{builder}$. Overall, our method does not produce any misclassification.

Effectiveness of transaction-based replay. To test the effectiveness of transaction-based replay, we compare the original execution results of all transactions, currently a total of 678,380, with the results obtained through replay for contracts in $D_{unknown}$. Finally, 659,196 transactions, around 97%, have consistent results. After analysis, we identify the main causes of inconsistency as follows: (1) When multiple transactions in a block invoke the same contract, replaying one of these transactions may lead to inconsistency. For example, if a user deposits tokens and then performs a transfer in two external transactions, directly replaying the transfer transaction may result in a revert, as the account's balance might be insufficient. (2) To test more transactions, *SmartReco* does not consider the gas costs. Thus, *SmartReco* may successfully execute transactions that revert due to insufficient gas.

Effectiveness of cross-DApp analysis. To validate the effectiveness of cross-DApp analysis, we remove the cross-DApp analysis module (without CDA). More specifically, we replace all cross-DApp analysis in *SmartReco* with traditional cross-contract analysis and *SmartReco* will consider all the contracts encountered in the execution as unsafe. The experimental result is shown in Fig. 7. Although without CDA successfully reports the same amounts of true positives as *SmartReco* does, the failure to recognize the boundaries between contracts results in reporting an excessive number of false positives. Besides, without CDA needs to test all contracts blindly, leading to more time costs (Section V-F).

Effectiveness of multi-function fuzzing method. To validate the effectiveness of the multi-function fuzzing method, we replace it with random fuzzing (without MFF). In other words, for each potential entry function, *SmartReco* attempts to generate input based on the original transaction and the function's ABI to simulate a user interacting with two DApps simultaneously. The result is shown in Fig. 7. Due to most users not having states in both DApps, such as holding assets in both DApps, most transactions revert as they cannot pass the internal check in both functions or cannot find the critical path of ROR, resulting in a significant amount of false negatives.

Answer to RQ 2:

Each module of *SmartReco* indeed helps reduce false positives and false negatives, and improves detection effectiveness.

D. Comparison with other Tools

To answer RQ3, we use $D_{labeled}$ to test with the two most advanced tools, ityFuzz [20] and Sailfish [18], as they are currently the most effective dynamic analysis tool and static analysis tool for detecting reentrancy vulnerabilities. We obtain the publicly released artifacts of these two tools from their respective publications. Note that, we do not compare *SmartReco* with other popular tools such as icyChecker [6], which can only detect reentrancy within DApps and is not aligned with our scope. Besides, Mythril [38], Oyente [8], and Vandal [39] have been proven less effective than Sailfish in detecting reentrancy [18]. We conduct the tests using the default configurations of these tools with a time limit of five minutes and one hour for Sailfish and ityFuzz respectively and manually identify the results as *SmartReco* does. After manual inspection, the results are presented in Fig. 7. We can observe that ityFuzz and Sailfish are unable to detect any RORs.

Our further investigation finds that although ityFuzz has made optimizations in the fuzzing process, such as using

on-chain states to emulate the real environment and using waypoints to improve input quality, it is not effective when it comes to ROR, as ROR requires generating multiple function inputs simultaneously. Besides, Sailfish is a static analysis tool that lacks information about the execution context, such as the specific address called by opcode CALL. Consequently, its call chain is incomplete and cannot detect RORs. Additionally, both tools randomly select functions for testing, which results in lower efficiency compared to *SmartReco*.

Answer to RQ 3:

Compared to advanced tools, *SmartReco* detects ROR with higher precision and recall.

TABLE II: Overall effectiveness of *SmartReco* on $D_{unknown}$.

Total Smart Contracts	Reported RORs	Detection results		
		TP	FP	Precision
2,676	55	50	5	90.91%

E. Large-scale Analysis for Finding Unknown RORs

To answer RQ4, we evaluate *SmartReco* based on $D_{unknown}$ and Table II shows the detailed results. From Table II we can find *SmartReco* reports 55 RORs. To validate the results detected by *SmartReco*, our three domain experts independently verified the detection results, and 50 out of the 55 vulnerabilities are confirmed by the experts and the overall precision is 90.91%. In addition, out of these 50 RORs, we discover that 43 RORs have not been publicly reported before and we find 35 new functions that are suffered with RORs. The total asset affected by these RORs is around 520,000 USD. We will analyze a case in Section V-G that can only be detected by the *SmartReco* but not by other advanced tools.

Answer to RQ 4:

SmartReco is effective in RORs under complex DApp interactions in real-world scenarios.

TABLE III: Performance of *SmartReco* and without CDA on $D_{labeled}$.

	Avg. Time (seconds)		Execution Count
	Per Execution	Finding ROR	
SmartReco	4.12	265.84	150421
without CDA	5.32	485.26	558438

F. Efficiency of SmartReco

To answer RQ5, we use $D_{labeled}$ to compare the efficiency of *SmartReco* and without CDA in terms of average time per execution, the average time to detect ROR, and number of executions. The results are shown in Table III. Note that we do not compare with without MFF, ityFuzz, and Sailfish, because they do not detect any ROR. Due to the caching mechanism, although *SmartReco* fetches on-chain data, there is no need to repeatedly retrieve after the initial acquisition. Therefore, the average execution time of *SmartReco* is not high. Although the average time per execution of without CDA and *SmartReco* are similar, without CDA has a larger search space and needs to execute more times, resulting in lower efficiency in detecting ROR compared to *SmartReco*. To this end, the performance of *SmartReco* will not be affected by the size of the dataset. Specifically, since the cross-DApp analysis can effectively filter the search space, a larger dataset only leads to at most a linear growth in the number of functions and transactions that need to be analyzed.

Answer to RQ 5:

SmartReco is rather efficient in detecting ROR with large-scale analysis.

G. Case Study

Fig. 8 is a real case detected by *SmartReco*. In detail, *joinPool* is the entry point for becoming a member of entry DApp. Normally, users need to synchronously transfer a certain amount of funds, like ETH, as collateral. Besides, *joinPool* provides a protection mechanism, line 5 in Fig. 8(b), to ensure that users do not transfer funds than expected. To exploit ROR in Fig. 8, the first step is to trigger line 5 and invoke line 7 in Fig. 8(b). After obtaining control of the transaction, attackers can obtain more funds through *removeLiquity* in Fig. 8(a). However, normal transactions are unlikely to trigger this path, as this will incur higher costs. We analyze all the transactions of *joinPool* in contract *Vault* and find that none of them trigger this mechanism. Therefore, solely relying on transactions for testing can result in potential false negatives. *SmartReco* can discover that *joinPool* is payable and will try to mutate the funds carried by *joinPool*, ultimately triggering the protection mechanism and finding the ROR.

VI. Discussion

A. Detection on deployed contracts

SmartReco primarily targets the deployed, on-chain contracts, as simulating real DApp interactions, such as obtaining a user's balance or a token price, in an off-chain environment is difficult. In the meantime, we would like to note that for DApps and their smart contracts before deployment, *SmartReco* can utilize existing united test suites or fuzzers to generate transactions and perform detection, similar to existing research [40]. Additionally, detecting deployed contracts is with high-value, as it can help the DApp owners find risks in advance and take measures to reduce losses. For example, deploying a new contract and transferring the funds from the old contract to the new one.

B. Threats to Validity

Internal threats. The internal threats of *SmartReco* mainly come from its dependence on clear DApp boundaries. The effectiveness of *SmartReco* could be affected if it fails to accurately recognize the boundary of a given DApp. However,

```
contract Periphery{
  function removeLiquity() public nonReentrant{
    ...
    // get outdated balance
    uint balance = Vault.getBalance();
    IAsset asset = convertERC20ToAssets(balance);
    withdrawTokens(assets);
  }
}
```

(a) The victim DApp contract.

```
contract Vault{
  function joinPool(address pool) public payable
      nonReentrant {
    ...
    _processJoinPoolTransfers();
    if (balanceTotal - balanceUsed > 0) {
        // root entry point of ROR
        _handleRemainingBalance();
    }
    updatePoolBalance();
    ...
  }
  // The manipulable function
  function getBalance() view{
    return balance;
  }
}
```

(b) The entry DApp contract.

Fig. 8: Simplified real-world smart contracts with ROR.

for contracts with unclear boundaries, *SmartReco* will perform analysis on all unknown contracts. As the results are shown in Section V-C, although *SmartReco* may miss some contract DApp information, there will be no misclassification, ensuring the effectiveness. Another major internal threat comes from the evaluation over a limited number of DApps (i.e., 123 popular DApps). Since *SmartReco* needs to test multiple functions simultaneously, the time overhead is larger than traditional single-contract fuzzing tools, which has limited our ability to conduct larger-scale evaluations. However, compared to randomly selected DApps, the DApps in our dataset are representative as they are top-popular DApps.

External threats. The external threats of *SmartReco* mainly come from the inability to analyze non-Solidity languages, such as Vyper, and contracts without source code. This is because *SmartReco* relies on Slither, which is based on Solidity source code for analysis (although the latest version of Slither claims to cover Vyper, it currently only supports version 0.3.7 [41]). We consider this to have a relatively small impact on the effectiveness of *SmartReco*. On the one hand, Solidity is currently the most mainstream smart contract development language [42]. On the other hand, most DApps tend to open-source code [43], as users are more willing to invest in open-source DApps. To this end, *SmartReco* only misses a small portion of contracts.

VII. Related Work

Vulnerability detection in smart contract. At present, most research uses static and dynamic analysis to detect vulnerabilities in smart contracts. Static analysis detects vulnerabilities by analyzing the source code [36], [44], [45] or bytecode [3], [16], [46] of contracts. For example, DeFiTainter uses decompiled bytecode to detect price manipulation. Slither translates source code into an intermediate representation, called SlitherIR, and performs analysis on it. Due to their inability to obtain execution context information, they miss critical information and cannot fully recover the call graph when detecting ROR. On the other hand, dynamic analysis technologies try to generate inputs that satisfy specific requirements, such as covering more branches. Then by executing the inputs, they can monitor the runtime status of the program and detect vulnerabilities [5], [7], [19], [47]. However, the lack of effective input generation methods in existing tools leads to a significant number of false negatives in ROR, as it is difficult for them to bypass the internal check in functions of several DApps.

Reentrancy in smart contract. Reentrancy is one of the most common vulnerabilities in smart contracts, and in recent years, many studies have focused on reentrancy [8], [18], [20], [31], [48]. For example, ReVulDL uses deep learning methods to detect reentrancy. ityFuzz employs online fuzzing methods to provide realistic testing. Clairvoyance [29] uses path-protective techniques to identify potential paths for detecting reentrancy. icyChecker [6] uses replay to find out potentially vulnerable functions. However, existing works on reentrancy appear to have very limited capability [24], as they overlook the potential correlations between contracts, such as using DApp information to detect vulnerabilities. Unlike existing approaches, *SmartReco* incorporates cross-DApp analysis to obtain more fine-grained information and uses a multi-function fuzzing method to effectively detect ROR, a novel type of reentrancy vulnerability.

VIII. Conlucsion

In this paper, we focus on the detection of ROR based on fine-grained cross-DApp analysis. We introduce *SmartReco*, a novel detection framework to detect RORs. Specifically, we propose a new DApp identification method to identify DApp boundaries from smart contracts. Additionally, *SmartReco* collects DApp-based contextual data based on replay, and uses this information to find potential entry functions. Then, *SmartReco* employs a multi-function fuzzing method to detect RORs. Experimental results show that *SmartReco* achieves good effectiveness, outperforming existing advanced tools. Besides, *SmartReco* successfully detects 43 RORs that have never been reported in real-world smart contracts.

Acknowledgment

This research is supported by the National Key Research and Development Program of China (2023YFB2704801), NSFC/RGC Collaborative Research (62461160332, CRS_HKUST602/24), the Major Key Project of PCL (Grant No. PCL2023A05).

REFERENCES

[1] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 363–373.

[2] S. Wu, D. Wang, J. He, Y. Zhou, L. Wu, X. Yuan, Q. He, and K. Ren, "Defiranger: Detecting price manipulation attacks on defi applications," *arXiv preprint arXiv:2104.15068*, 2021.

[3] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[4] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2444–2461.

[5] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 65–68.

[6] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, "Detecting state inconsistency bugs in dapps via on-chain transaction replay and fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 298–309.

[7] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 103–119.

[8] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[9] Z. Zhang, Y. Lei, M. Yan, Y. Yu, J. Chen, S. Wang, and X. Mao, "Reentrancy vulnerability detection and localization: A deep learning based two-phase approach," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[10] "Decoding 220k read-only reentrancy exploit — quillaudits," 2022. [Online]. Available: https://quillaudits.medium.com/decoding-220k-read-only-reentrancy-exploit-quillaudits-30871d728ad5

[11] "Top 10 hacking techniques of 2022," 2023. [Online]. Available: https://www.openzeppelin.com/security-audits/top-hacking-techniques-2022

[12] S. Kitzler, F. Victor, P. Saggese, and B. Haslhofer, "Disentangling decentralized finance (defi) compositions," *ACM Transactions on the Web*, vol. 17, no. 2, pp. 1–26, 2023.

[13] V. von Wachter, J. R. Jensen, and O. Ross, "Measuring asset composability as a proxy for defi integration," in *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 2021, pp. 109–114.

[14] "Dappradar - the world's dapp store — blockchain dapps ranked," 2023. [Online]. Available: https://dappradar.com/

[15] "The graph," 2023. [Online]. Available: https://thegraph.com/

[16] Q. Kong, J. Chen, Y. Wang, Z. Jiang, and Z. Zheng, "Defitainter: Detecting price manipulation vulnerabilities in defi protocols," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, p. 1144–1156.

[17] F. Ma, Z. Xu, M. Ren, Z. Yin, Y. Chen, L. Qiao, B. Gu, H. Li, Y. Jiang, and J. Sun, "Pluto: Exposing vulnerabilities in inter-contract scenarios," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4380–4396, 2021.

[18] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 161–178.

[19] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 259–269.

[20] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 322–333.

[21] W. Metcalfe *et al.*, "Ethereum, smart contracts, dapps," *Blockchain and Crypt Currency*, vol. 77, 2020.

[22] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[23] "Bnb chain," 2023. [Online]. Available: https://www.bnbchain.org

[24] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 295–306.

[25] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, "Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–50, 2023.

[26] M. Bartoletti, J. H.-y. Chiang, and A. L. Lafuente, "Sok: lending pools in decentralized finance," in *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 2021, pp. 553–578.

[27] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (defi)," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, 2022, pp. 30–46.

[28] A.-D. Popescu, "Decentralized finance (defi)–the lego of finance," *Social Sciences and Education Research Review*, vol. 7, no. 1, pp. 321–349, 2020.

[29] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1029–1040.

[30] Y. Xue, J. Ye, W. Zhang, J. Sun, L. Ma, H. Wang, and J. Zhao, "xfuzz: Machine learning guided cross-contract fuzzing," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[31] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 227–239.

[32] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[33] Y. Kim, S. Jeong, K. Jezek, B. Burgstaller, and B. Scholz, "An {Off-The-Chain} execution environment for scalable testing and profiling of smart contracts," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 565–579.

[34] S. Wu, L. Wu, Y. Zhou, R. Li, Z. Wang, X. Luo, C. Wang, and K. Ren, "Time-travel investigation: toward building a scalable attack detection framework on ethereum," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–33, 2022.

[35] C. Dannen, *Introducing Ethereum and solidity*. Springer, 2017, vol. 1.

[36] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[37] Vyper, "Vyper document," 2024. [Online]. Available: https://docs.vyperlang.org/en/latest/

[38] "Mythril," 2023. [Online]. Available: https://github.com/ConsenSys/mythril

[39] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[40] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 716–727.

[41] "Vyper != 0.3.7 support is a best effort and might fail," 2023. [Online]. Available: https://github.com/crytic/crytic-compile/blob/master/crytic_compile/platform/vyper.py#L81

[42] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[43] W. Zhang, Z. Zhang, Q. Shi, L. Liu, L. Wei, Y. Liu, X. Zhang, and S.-C. Cheung, "Nyx: Detecting exploitable front-running vulnerabilities

in smart contracts," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 146–146.
[44] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.
[45] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.
[46] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," *Proc. ACM ICSE*, 2023.
[47] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
[48] Z. Zhang, Y. Lei, M. Yan, Y. Yu, J. Chen, S. Wang, and X. Mao, "Reentrancy vulnerability detection and localization: A deep learning based two-phase approach," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.