PrettySmart: Detecting Permission Re-delegation Vulnerability for Token Behaviors in Smart Contracts

Zhijie Zhong
Sun Yat-sen University
Zhuhai, China
zhongzhj3@mail2.sysu.edu.cn

Zibin Zheng*
Sun Yat-sen University
Zhuhai, China
zhzibin@mail.sysu.edu.cn

Hong-Ning Dai
Hong Kong Baptist University
Hong Kong, China
hndai@ieee.org

Qing Xue
Sun Yat-sen University
Guangzhou, China
xueq25@mail2.sysu.edu.cn

Junjia Chen
Sun Yat-sen University
Guangzhou, China
chenjj275@mail2.sysu.edu.cn

Yuhong Nan
Sun Yat-sen University
Zhuhai, China
nanyh@mail.sysu.edu.cn

ABSTRACT

As an essential component in Ethereum and other blockchains, token assets have been interacted with by diverse smart contracts. Effective permission policies of smart contracts must prevent token assets from being manipulated by unauthorized adversaries. Recent efforts have studied the accessibility of privileged functions or state variables to unauthorized users. However, little attention is paid to how publicly accessible functions of smart contracts can be manipulated by adversaries to steal users' digital assets. This attack is mainly caused by the permission re-delegation (PRD) vulnerability. In this work, we propose PrettySmart, a bytecode-level Permission re-delegation vulnerability detector for Smart contracts. Our study begins with an empirical study on 0.43 million open-source smart contracts, revealing that five types of widely-used permission constraints dominate 98% of the studied contracts. Accordingly, we propose a mechanism to infer these permission constraints, as well as an algorithm to identify constraints that can be bypassed by unauthorized adversaries. Based on the identification of permission constraints, we propose to detect whether adversaries could manipulate the privileged token management functionalities of smart contracts. The experimental results on real-world datasets demonstrate the effectiveness of the proposed PrettySmart, which achieves the highest precision score and detects 118 new PRD vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Smart Contract, Permission Control, Vulnerability Detection

*corresponding author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639140

ACM Reference Format:
Zhijie Zhong, Zibin Zheng, Hong-Ning Dai, Qing Xue, Junjia Chen, and Yuhong Nan. 2024. PrettySmart: Detecting Permission Re-delegation Vulnerability for Token Behaviors in Smart Contracts. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639140

1 INTRODUCTION

The concept of cryptocurrencies has been prevailing with the development of blockchain technology. Since only a few cryptocurrencies are natively supported by blockchains (e.g., Bitcoin and Ethereum), a variety of *tokens* have been developed on top of smart contracts to meet the need for customized asset management. Tokens have cultivated an enormous ecosystem on Ethereum and many other similar blockchains. In February 2023, the circulating market capitalization of the top-10 tokens accounted for more than 230 billion USD across the blockchains [6]. Correspondingly, numerous smart contracts have been developed for managing users' token assets, such as decentralized finance (DeFi), games, digital wallets, cross-chain exchanges, etc. Since these smart contracts have been granted access to users' digital assets, their permissions should be *properly* assigned to prevent malicious attackers from manipulating their privileged fund-transferring functionalities.

Therefore, ensuring the appropriate permission management of these smart contracts is crucial. Many recent research endeavors have been made to detect permission vulnerabilities for smart contracts. Although previous studies [17, 19, 31, 36] consider *whether* the functions or state variables in smart contracts are accessible to attackers, few of them consider *how accessible functions can be manipulated* by the attackers for malicious use. The latter question plays an important role in the permission management of smart contracts and may consequently incur a severe financial loss if it is not properly addressed. For example, it was reported in October 2022 [4] that a cross-chain decentralized exchange, named *TransitSwap*, was hacked for an estimated 21 million USD. Considering this real-world contract as shown in Figure 1, the attacker launched the attack by calling a publicly accessible function in the *TransitSwap* smart contract with malicious parameters. After several internal contract calls in the decentralized application (DApp), the polluted parameters were propagated to the token transfer function `transferFrom`. As a result, the attacker was able to steal digital assets

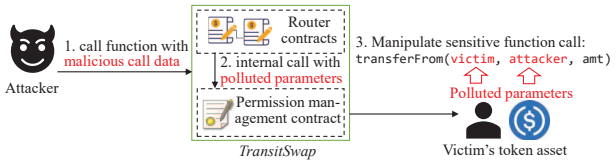

Figure 1: The 2022 *TransitSwap* hack

from arbitrary users who had previously approved smart contracts to manage their tokens. The detailed analysis will be given in § 3.

The above problem is caused by the *permission re-delegation (PRD) vulnerability* of smart contracts. The PRD vulnerability occurs when an unauthorized attacker exploits an execution path to manipulate the sensitive parameters of token transfer functions or state variables in a smart contract with approved privileges. As disclosed by multiple reports [2–4], adversaries attempted to exploit the PRD vulnerabilities of smart contracts to steal user's token assets. These contracts have involved over 200,000 transactions[1] and caused millions of financial losses for a large number of users [2–4]. More importantly, our detection results reveal that 302 real-world smart contracts contain the PRD vulnerability (in § 5.2).

However, detecting the PRD vulnerability from token behaviors is non-trivial due to the following facts: 1) Smart contracts generally adopt *various permission constraints* to place restrictions on the caller's address or attributes. The developers' implementations of these permission constraints are highly diverse due to diverse business logic. 2) Permission constraints can be *bypassed by an unauthorized adversary* to access privileged resources. The bypass activity can be achieved by either manipulating the state variables used for the permission check or making a cross-address call from a permissioned smart contract. Underestimating or overestimating these permission constraints may result in high false positives. For example, a number of existing tools [7, 16, 29, 36] are reported to have high false positives due to unsatisfactory support of permission constraints [7, 39]. While techniques such as symbolic execution and constraint solving can be used for finding a path bypassing the permission constraints, existing methods [18, 29, 31] suffer from the path-explosion problem or require efficiency-improving strategies for effective vulnerability detection [7, 41]. Alternatively, dynamic methods, such as fuzzing tools, can achieve satisfactory scalability, but their effectiveness is limited by the test oracle generation for detecting high-level permission bugs [28].

In this work, we propose a bytecode-level Permission re-delegation vulnerability detector for Smart contracts (PrettySmart). PrettySmart performs a customized taint analysis to exploit whether the potentially polluted data from an unauthorized user can be propagated to token transfer functions via key parameters, or be propagated to privileged state variables. Our analyzer implements the following mechanisms to detect such vulnerabilities effectively. First, we investigate how smart contracts enforce permission control policies by conducting an empirical study on 0.43 million open-source smart contracts collected from Etherscan [6] till 23/04/2023. It is found that five types of widely-used permission constraints dominate 98% of the studied contracts. Based on this observation, We propose to infer these permission constraints by exploiting the corresponding bytecode instruction sequences and data-flow facts. Moreover, we propose an algorithm to identify permission constraints that an unauthorized adversary can bypass. Inferring permission constraints, PrettySmart avoids overestimating permissioned execution paths that are infeasible in an attacking scenario. Recognizing bypassed permission constraints, PrettySmart avoids underestimating vulnerable paths. As a result, PrettySmart effectively detects PRD vulnerabilities. We evaluated our approach on two real-world datasets, including smart contracts collected from reported vulnerabilities and *SmartBugs-wild* public dataset [14]. Results show that PrettySmart outperforms state-of-the-art approaches by reporting more vulnerabilities while achieving a higher precision.

In summary, this paper makes the following contributions:

- We conduct a large-scale empirical study to understand how smart contracts enforce permission policies.
- We propose PrettySmart[2], a novel bytecode-level analyzer to detect PRD vulnerabilities. We design a set of novel mechanisms to improve the effectiveness of PrettySmart.
- We apply PrettySmart to real-world smart contracts collected from reported vulnerabilities and SmartBugs-wild benchmark. Experimental results demonstrate the effectiveness of PrettySmart in detecting PRD vulnerabilities, as well as permission constraint inference and bypass analysis.

2 BACKGROUND

This section reviews smart contracts, the Ethereum Request for Comments (ERC) token standard, and the PRD vulnerability.

Smart Contract and ERC Tokens. Smart contracts are programs running on virtual machines supported by blockchains, such as Ethereum Virtual Machine (EVM) for the Ethereum blockchain. With the help of smart contracts, developers can deploy diverse applications with complex business logic on top of the blockchain.

Tokens are one of the most influential applications implemented by smart contracts. To regulate the interaction of tokens and relevant applications, a set of token standards have been proposed. As one of the most popular token standards on Ethereum, ERC-20 defines the interaction model of tokens by providing six standard interfaces [5]. Among these interfaces, three of them are designed for token transfer functionalities as shown in Figure 2. Basically, A user can transfer his/her tokens by calling the `transfer` function. When users join a DApp and allow the application to manage their assets, they should first call the `approve(address _spender, uint256 _value)` function of the corresponding token contract. The function specifies that the `_spender` address is allowed by the user to spend the `_value` amount of their tokens. Accordingly, an approved address can spend the tokens on behalf of the user by calling the `transferFrom` function. It is worth noting that a considerable number of users would like to approve an infinite amount of tokens to their trusted smart contracts (called *infinite approval*). By infinite approval, they avoid the trouble of approving each time before use and thereby can save considerable GAS fees. However, the infinite approval undermines the safety of users' token assets. Once the

[1] https://etherscan.io/address/0xbaDc0dEfAfCF6d4239BDF0b66da4D7Bd36fCF05A, https://etherscan.io/address/0xc8d7899f22bc4995c8176e3f2a5ba3f5e87d95e5, https://etherscan.io/address/0x8dFEB86C7C962577deD19AB2050AC78654feA9F7

[2] available at https://github.com/Z-Zhijie/PrettySmart

```
function transfer(address _to, uint256 _value)
function transferFrom(address _from, address _to, uint256
    _value)
function approve(address _spender, uint256 _value)
```

Figure 2: The ERC-20 standard interfaces for token transfers

approved smart contract gets exploited by attackers, all the tokens belonging to the user can be stolen by the attackers.

Permission Re-delegation Vulnerability. The PRD vulnerability has been previously studied in the context of Android apps [17]. We redefine the *PRD* vulnerability in smart contracts as follows.

Definition 1. PRD *occurs when a smart contract with permissions performs a privileged task for the contract without permissions.*

With the prevalence of smart contracts and tokens, PRD also occurs in the context of token management. Consider that an attacker cannot spend users' tokens by directly calling the `transferFrom( _from=user, _to=attacker, _value=amount)` function when no approvals were made to the address, where the three arguments indicate the token sender, the token receiver, and the send amount. A DApp is allowed to spend the users' tokens if she has approved the address of the DApp. However, the PRD attack can happen if the attacker exploits an execution path to manipulate the key arguments of the `transferFrom` function called by the privileged DApp. Specifically, if the first argument (`_from`) and the second argument (`_to`) of the function call are assigned to the victim's address and the address controlled by the attacker, then the DApp can be manipulated to transfer the victim's token assets to the attacker.

3 MOTIVATION

This section introduces a real-world attack and summarizes the limitations of prior studies to motivate our work.

Motivating Example. In October 2022, a real-world DApp named `TransitSwap` was hacked for 21 million USD [4]. `TransitSwap` has been used for cross-chain token exchanges. When users swap tokens in the DApp, they need to first approve the DApp's Permission Contract to spend their tokens. After the hacker noticed that there were insufficient validations for the parameters used in their token transfer function `transferFrom()`, he eventually found a path to manipulate these parameters. Figure 3 shows the core data propagation path of the attack. When a user swaps tokens in the DApp, the Proxy Contract with the corresponding `CallData` needs to be called. The Proxy Contract then calls the `claimtokens()` function of the Router Contract with `CallData`. The `claimtokens()` function adopts a permission constraint (`msg.sender == proxy?`) to check whether the contract caller is the Proxy Contract, and next passes the parameters to the `callbytes()` function of the Permission Contract. In the Permission Contract, a similar permission constraint is also applied to check the caller's address. If the permission constraint is passed, the contract further calls the corresponding `transferFrom()` function with the parameters received from the Router Contract. As a result, the attacker may manipulate the parameters of the high-privileged `transferFrom()` function by deliberately generating the initial `CallData` when entering the Proxy Contract.

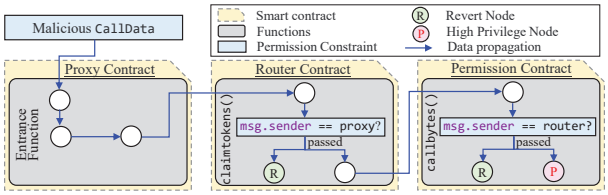

Figure 3: The core execution path of the `TransitSwap` attack

We have two observations from the above example: 1) *The attack can only succeed in a cross-contract call sequence.* Although `callbytes()` is a public function, it only accepts calls from the Router Contract by checking the permission constraint. Effective detection of this vulnerability requires precise recognition of feasible call chains for an unauthorized attacker. Unfortunately, these permission constraints cannot be easily inferred since all of the involved smart contracts are typically not open-sourced. 2) *All of the functions and state variables on the vulnerable path are also accessible in a benign method invocation.* Since the execution path of a malicious attack is the same as that in a benign execution, this vulnerability cannot be detected by simply checking the accessibility of specific functions or state variables.

Limitations of Prior Research. The PRD vulnerabilities cannot be easily detected by off-the-shelf approaches due to the following intrinsic limitations: (1) **Intra-contract analysis.** Existing studies including Securify [36], AChecker [19], and SPCon [28] generally detect permission bugs by intra-contract analysis instead of cross-contract analysis. For example, despite the effectiveness in finding permission bugs by recognizing the inconsistency between extracted permission strategies and testing results, SPCon [28] can only be applied to individual contract addresses since its permission strategies are extracted from limited historical transactions of those contracts. (2) **Flawed detection patterns.** The other studies [19, 28, 29, 31, 36] mainly focus on the accessibility of functions and state variables. Generally, they detect permission bugs by recognizing the *abnormal* accesses of functions or state variables while neglecting benign-yet-risky method invocations. Hence, these approaches do not raise alarms due to the *second observation.*

Although *PRD* attacks are becoming critical threats to users' token assets, an effective detection method for detecting this type of vulnerability is still largely missing in the literature.

4 METHODOLOGY

To overcome the limitations of existing methods, we propose PrettySmart, a bytecode-level cross-contract analyzer for detecting PRD vulnerabilities of smart contracts. Figure 4 depicts the framework of PrettySmart. Overall, PrettySmart takes smart contract bytecode as input and outputs the vulnerable functions. The detection consists of four stages: ❶ Decompilation, ❷ Permission constraint Inference, ❸ Cross-Contract Inter-procedural Control flow Graph (XCFG) Construction and ❹ Customized Taint Analysis. Specifically, PrettySmart starts with the decompilation of smart-contract bytecode into three-address Intermediate Representation (IR) code. We then propose an algorithm to infer permission constraints (PCs)

based on a large-scale empirical study. Next, we construct the cross-contract control-flow graph based on the decompiled results. Thereafter, we conduct a two-step taint analysis to 1) recognize which PCs can be bypassed and which can then be used for identifying feasible and infeasible entry points in an attacking scenario; and 2) find whether there is a vulnerable path in smart contracts. The taint analysis is customized with awareness of the PCs implemented by the contract. The detailed implementation of PrettySmart is elaborated as follows.

4.1 Stage 1: Decompilation

Recent studies on EVM decompilers and static analysis have contributed to the decompilation of smart contract bytecode [1, 9, 20, 21, 26]. In this work, we leverage Gigahorse [20, 21, 26] to decompile the bytecode into three-address IR code. Compared with other decompilers, Gigahorse is reported to have higher performance in terms of resolving operands, recognizing the entry point of private functions, and inferring the function boundaries.

4.2 Stage 2: Permission-Constraint Inference

PCs have been widely adopted in smart contracts to control the accessibility of the corresponding functions. It is *non-trivial* to infer these PCs because permission control is highly customized concerning the diverse business logic of smart contracts. The developers' diverse implementations of PCs may result in various bytecode operations of smart contracts after compilation. To tackle this challenge, we performed an empirical study to investigate how developers enforce these PCs and exploit how these PCs work in the form of EVM operations.

The scope of this study covers all open-source smart contracts on Ethereum till 23/04/2023. We collected nearly 0.78 million contracts with source codes via Etherscan [6]. After deduplication, there are 0.43 million distinct smart-contract source codes. We analyzed the enforcement of permission control policies *w.r.t.* the `require` statements and the keyword `msg.sender`. The keyword choice is motivated by two observations: 1) The `require` statement in Solidity is used for checking conditions and throwing an exception if the check fails. Hence, it becomes a natural choice for implementing permission checks for smart contracts. For example, a popular implementation of permission check is "`require(msg.sender==owner)`" [8], where the `msg.sender` is the contract caller and the `owner` is the owner address. The caller can pass the permission check only if he/she is the owner of the smart contract. 2) The object of permission control is the contract caller, which is represented by the `msg.sender` variable in Solidity. Therefore, we filtered out `require` statements that are irrelevant to the contract caller (i.e., `require` statements that do not contain the keyword `msg.sender`). Detailed analytical results will be reported in § 5. As a result, we analyzed the functionality of 284,787 permission control statements. We found that most of the permission control policies implemented for smart contracts can be divided into the following two categories.

- *Address restriction* requires the caller's address to be a specific address or belong to a specific group. Similar to permissions in Linux systems, this type of permission policy can be divided into *User* (U)-oriented and *Group* (G)-oriented, where each type has two implementations according to our study (i.e., two U-oriented implementations and two G-oriented implementations).
- *Data-driven restriction* does not restrict the contract caller's address. Instead, it checks whether an attribute of the caller meets a specific condition, e.g., checking whether the balance of the caller is larger than a specific value.

In the following, we describe these five types of implementations and the corresponding EVM operation sequences.

I. The *compare2Owner* Constraint. The upper part of Figure 5 shows a code snippet of the *compare2Owner* Constraint. The `require` statement in line 2 checks if the caller address (i.e., `msg.sender`) is the same as a predefined address in a state variable (in line 1). Developers can check the constraint by adding the `require` statement to the target function. The lower part of Figure 5 demonstrates the corresponding operation sequence. When the permission constraint is activated, EVM first loads the `owner` state variable and the caller address by using the `SLOAD` and `CALLER` operations. The two variables will be then compared by `EQ` operation. Depending on the comparison result, the `JUMPI` operation continues the execution or jumps to a path that ends with a `REVERT` operation.

II. The *queryOwner* Constraint. This type of permission constraint is usually used in the implementation of a Non-Fungible Token (NFT) smart contract. Each NFT is uniquely identified with a specific identifier (such as a unique token ID) and each NFT is associated with an owner address. As illustrated in Figure 6, the code in line 1 uses a mapping variable `nftOwner` to record the owner of each NFT id. The code in line 2 checks if the contract caller is the owner of the NFT with ID `_nftId`. It first queries the owner of the `_nftId` from the mapping variable `nftOwner`, then compares it with the caller's address `msg.sender`. The check result is used in a `require` statement. Therefore, if the check fails, i.e., the contract caller is not the owner of the corresponding NFT, the transaction will be reverted. The lower part of Figure 6 depicts the EVM operation sequence of this permission constraint. EVM performs the permission check by first accessing the value in the mapping variable with respect to the key `_nftId`. It then checks if the owner is the same as the contract callers. In the low-level execution of EVM, the storage location of a value in the mapping is calculated by the hash operation `SHA3` with two inputs: the corresponding key and the storage location of the mapping variable itself. In this case, the operation takes the `nftId` and the location of the mapping variable `nftOwner` as input and outputs their hash value. The value of `nftOwner[_nftId]` is loaded by the `SLOAD` operation from the storage. An `EQ` operation is then used to check if it is equal to the caller's address `CALLER`. Depending on the result, the `JUMPI` operation will jump to a `REVERT` operation or continue the following execution.

III. The *groupBased* Constraint. Another widely-used strategy is the "allowList" constraint, which gives permission to a group of accounts, where the privileged accounts can be flexibly added, deleted, or modified. As shown in Figure 7, the smart contract maintains an allow list for the users by keeping a mapping variable in line 1. The mapping variable associates the address of an account to a boolean value indicating its permission. If the corresponding value is `True`, then the account can pass the permission check in line 2. Similar to the queryOwner constraint, EVM first accesses the value of the mapping variable with key `msg.sender` and then checks whether the boolean value is `True`. The lower part

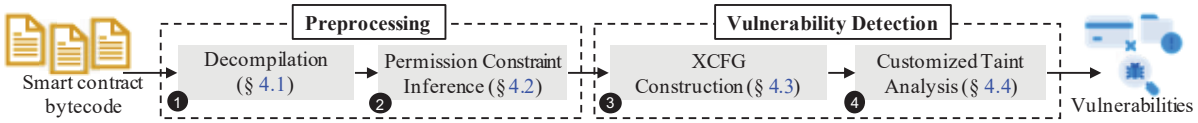

Figure 4: Framework of PrettySmart

```
1 address private owner;
2 require(msg.sender == owner);
```

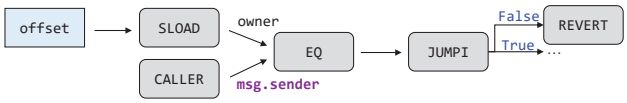

Figure 5: Code snippet and corresponding operation sequences of the *compare2Owner* constraint.

```
1 mapping (uint256 => address) private nftOwner;
2 require(nftOwner[_nftId] == msg.sender);
```

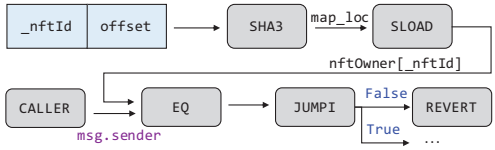

Figure 6: Code snippet and corresponding operation sequences of the *queryOwner* constraint.

```
1 mapping(address => bool) private allowlist;
2 require(allowlist[msg.sender]);
```

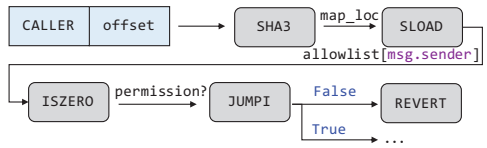

Figure 7: Code snippet and corresponding operation sequences of the *groupBased* constraint.

of Figure 7 illustrates the detailed operation sequence. Specifically, EVM first calculates the storage location of the boolean variable `allowlist[msg.sender]` by a SHA3 operation. The parameters of this operation are the caller's address (obtained by the CALLER operation) and the location of the mapping variable. The output is the hash value of them. The value of `allowlist[msg.sender]` is then loaded by the SLOAD operation and passed to the ISZERO check. Depending on the check result, the execution will be reverted or continued.

IV. The *roleBased* Constraint. To support a fine-grained permission control strategy for groups, OpenZepplin provides the Role-based Access Control Library. As shown in Figure 8, this strategy allows the developer to define a group of roles for using the smart contract (line 5). Each group is maintained by a `struct` variable `RoleData` (line 1). The mapping variable `members` in the `struct`

```
1 struct RoleData {
2     mapping(address => bool) members;
3     bytes32 adminRole;
4 }
5 mapping(bytes32 => RoleData) private _roles;
6 require(_roles[role].members[msg.sender]);
```

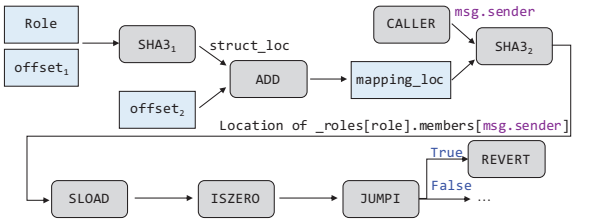

Figure 8: Code snippet and corresponding operation sequences of the *roleBased* constraint.

records the account addresses for this role in an allow list manner (line 2). The permission check can be activated by using the `require` statement defined in line 6. It checks whether the contract caller is in the `role` group. Specifically, when the `require` statement is executed, EVM will first access the corresponding `RoleData` `struct` variable (i.e., `_roles[role]`). If the mapping variable in the `struct` indicates that the contract caller is in the `role` group (i.e., `_roles[role].members[msg.sender]` is `True`), the permission check is then passed. The lower part of Figure 8 depicts the operation sequence of this permission constraint. Specifically, EVM will first find the location of the `struct` variable (i.e., `RoleData`) by calculating the hash of the mapping key (i.e., `role`) and the storage location of the mapping variable `_roles`. Then, an ADD operation is performed to find the location of the mapping variable `members` in `RoleData`, as the elements of structs are stored one after one in EVM storage. To get the location of `members[msg.sender]`, the storage location of the mapping `members` together with the caller's address will be passed to a SHA3 operation. Then, a SLOAD will be used to get the value of `members[msg.sender]`. Lastly, EVM will decide to revert or continue the execution depending on the value of `members[msg.sender]`.

V. The *dataDriven* Constraint. This type of permission constraint does not require the contract caller to be a specific user or be in a specific group of users. Instead, it checks if an attribute (i.e., user's balance) of the caller meets the requirement. In such cases, a mapping data structure is a natural choice for storing the attributes of users since it directly connects the key (user address) and the value (attributes). For example, line 1 of Figure 9 defines a mapping variable `balances` to store the balance of user accounts. The permission constraint in line 2 checks if the balance of the caller `balance[msg.sender]` is no less than a constant value `_value`. The low-level operation sequence is shown in the lower part of Figure 9.

```
mapping(address => uint256) balance;
require(_value <= balance[msg.sender]);
```

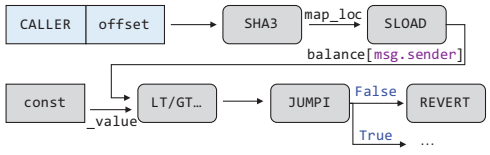

Figure 9: Code snippet and corresponding operation sequences of the *dataDriven* constraint.

Similar to the group-based permission constraint, EVM first queries the mapping value `balance[msg.sender]` with a `SHA3` operation. The mapping value is then passed to a compare operation such as `LT` (Less Than) or `GT` (Greater Than) to check if the mapping value is less or greater than a constant value `_value`. Similarly, EVM will revert or continue the execution according to the comparison.

Recognizing the corresponding operation sequences in the three-address code, PrettySmart is able to infer PCs adopted by smart contracts, thereby benefiting the following two analyses: (1) **Locating the permission-dependent variables.** PrettySmart can locate the storage location of the state variable used for the permission constraint, which can be used as the identity of the state variable. For simplicity, we call these permission-dependent variables PDVs. (2) **Recognizing the PCs that can be bypassed.** PrettySmart can identify which of these permission-dependent variables can be changed by an unauthorized attacker and thus recognize which of the PCs can be bypassed. In this way, PrettySmart can avoid making false positives by filtering out infeasible entry points.

4.3 Stage 3: XCFG Construction

In this stage, we build the XCFG from the decompiled three-address code. We adopt the XCFG building algorithm used in SmartDagger [27] and Clairvoyance [39]. The key idea is to connect the CFGs of each individual contract function by adding function call edges.

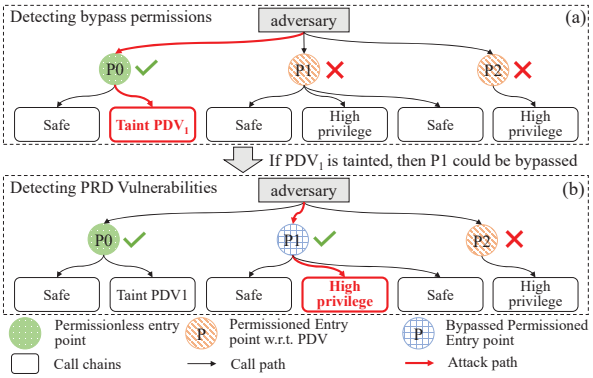

Figure 10: Taint analysis with awareness of the permission constraint: (a) detecting bypassed permission constraints, and (b) detecting PRD vulnerabilities based on the permissionless or permissioned-but-bypassed entry points.

Specifically, our XCFG analysis is mainly composed of two operations: 1) Adding control flow edges between intra-contract functions (*inter-procedural analysis*). The destination of intra-contract function calls can be determined by the `JUMP` destination of the bytecode. Therefore, the callee function can be directly found by the `JUMP` operands and the decompilation results of Gigahorse [20]; 2) Adding control flow edges between smart contract functions in different addresses (*cross-contract analysis*). The destination of cross-contract function calls in EVM is determined by the callee's address and the function signature hash (i.e., the first 4 bytes of the hash of the function signature). For the off-chain developing and testing process, the callee's address can be manually given. For on-chain smart contracts, the callee's address is either stored in an on-chain state variable slot or given as a parameter. The former case can be referred to by querying the state variable in the corresponding slot through on-chain storage query services such as *ethereum.storage*. Similar to prior research such as Sailfish [7], Smartdagger [27], and Pluto [30], PrettySmart cannot recover those contract addresses given at runtime. Recovering such addresses is non-trivial as it requires runtime simulation or in-depth analysis over the potential call addresses. Admittedly, in this case, Prettysmart may miss some of the PRD. We leave this as part of the future work. Given the callee's address, the destination node of the cross-contract call can be found by comparing the callee's public function signature hash and the signature hash given in the function call. We note that EVM bytecode does not contain explicit function call information. An external function call is executed by a `CALL` operation with seven stack inputs. The function signature hash and corresponding parameters are stored in the memory address indicated by the 4th input `argsOffset` and the 5th input `argsSize`. Therefore, we can identify the function signature hash by tracking the values stored in the corresponding address `argsOffset`. In this way, we combine individual CFGs to get a cross-contract inter-procedural CFG.

4.4 Stage 4: Customized Taint Analysis

In this stage, we perform a customized taint analysis with awareness of the PCs adopted by the entry function. As shown in Figure 10, the detection process contains two rounds of taint analysis: R1) identifying PCs that can be bypassed, and R2) detecting vulnerability based on the permissionless or permissioned-but-bypassed entry points. The first round is generic for all smart contracts and the second round is PRD-specific.

R1) Bypassed Permission Detection. We first locate PDV for each PC. We then exploit taint analysis to recognize which PDV could be modified by adversaries. According to the patterns described in § 4.2, each of the PCs depends on one or several state variables (i.e., PDVs). These PDVs can be uniquely identified by the storage locations of the corresponding state variables. We identify the storage location of the PDV as follows: 1) For the *compare2Owner* constraints, the PDVs are of the `address` type (e.g., `address owner` in Figure 5). It can be directly read from a constant storage location. The storage location of the corresponding PDV can be identified by recognizing the operand used for the `SLOAD` operation. 2) For the other constraints, the PDVs are stored in mapping data structures. Generally, for a mapping variable `map<key, value>`, EVM reads the storage location of a mapping `value` through

Table 1: Key components of the first round taint analysis

Sources	(1) `msg.sender` and `tx.origin` operations; (2) input of the entry point function; (3) return data of external calls with undetermined address.
Propagation rules	(1) data assignment and algorithmic operation; (2) `SSTORE/MSTORE` tainted variables to storage or memory location; (3) `SLOAD/MLOAD` tainted storage or memory locations to variables; (4) external function calls with tainted parameters.
Sinks	(1) `SSTORE` to locations of PDVs.

a `SHA3` operation with two parameters: the `map<key` and the identity of the mapping variable itself. Therefore, we can mark the location of the corresponding PDV as `SHA3(key,map_id)`.

Given the storage locations of PDVs, we leverage taint analysis to detect whether untrusted inputs can be assigned to these PDVs. As shown in Table 1, the taint sources are chosen as input values determined by untrusted users such as the contract caller's address `msg.sender`. The taint propagation rules are data flow rules as used in Clairvoyance [39] and Smartdagger [27]. Taint sinks are chosen as `SSTORE` operation to the storage locations of PDVs since `SSTORE` is the only way to modify state variables in EVM [37]. If the taint sources are propagated to taint sinks, unauthorized users can set the PDV as malicious value, e.g., setting the `address owner` variable in Figure 5 as an adversary-controlled address or increasing their account balances. Hence, they can satisfy the permission constraint "`require(msg.sender==owner)`" or "`require(_value <= balances[msg.sender])`". In other words, an adversary can bypass those PCs dependent on these PDVs.

We propose Algorithm 1 for permission bypass recognition. The algorithm takes four inputs: 1) the set of PCs $\mathbb{P} = \{P_c(p)\}$, where $P_c(p)$ is a permission constraint dependent on PDV p; 2) the set of permission-free entry points $\mathbb{E}_{\text{free}}$, i.e., publicly available entry points without PCs; 3) the set of permissioned entry points $\mathbb{E}_{\text{p}}$, i.e., entry points with at least one PCs, where each permissioned entry point $e \in \mathbb{E}_{\text{p}}$ is labeled with a Permission Constraint $P_c(p)$, indicating the implemented permission constraint and its corresponding PDV; and 4) XCFG. The output of Algorithm 1 is the set of all the PCs P_{bypassed}, which can be bypassed.

Algorithm 1 first records those entry points with no PCs as "*to be visited*" (line 1). The bypassed permission constraint set P_{bypassed} is initialized as an empty set (line 2). Then, Algorithm 1 traverses all the entry points in the ToVisit set $\mathbb{T}$ (line 3-4). We traverse XCFG to get the inter- and intra-contract paths starting from each entry point and perform taint propagation analysis (line 5-8). As discussed, if the taint source is propagated to PDV p, an adversary can modify the PDV as a malicious value to bypass the permission constraint. In this case, Algorithm 1 appends these bypassed PCs to P_{bypassed} (line 9-11). Additionally, an adversary can further exploit these permissioned-but-bypassed entry points to taint other PDVs, thereby bypassing other PCs. Therefore, Algorithm 1 also appends these entry points "*to be visited*" (line 12). In this manner, Algorithm 1 can further exploit all the PCs that can be bypassed and collect them into P_{bypassed} (line 17).

R2) PRD Detection. Given the bypassed permission constraint set, we can identify feasible entry points for an unauthorized attacker. Specifically, we exploit whether the key parameters of sensitive function calls or privileged state variables can be tainted.

Algorithm 1: Bypassed Permission Detection

Input: $\mathbb{P} = \{P_c(p)\}$: PC Set $\mathbb{P}$ containing constraint $P_c(p)$;
$\mathbb{E}_{\text{free}}$: Permission-free Entry Points set;
$\mathbb{E}_{\text{p}}$: Permissioned Entry Points set;
XCFG: The cross-contract inter-procedural CFG
Output: P_{bypassed}: bypassed Permission Constraints

```
1  T ← E_free //T is a set of point to be visited
2  P_bypassed ← ∅ //Initialize P_bypassed to be empty
3  while T is not empty do
4      e ← T.pop() //get a entry point e
5      Π ← xCFG.getPathsFrom(e)
6      for each path π ∈ Π do
7          s ← π.getTaintSource()
8          π.taintPropagtion(s)
9          for each P_c(p) ∈ P do
10             if p is tainted then
11                 P_bypassed.add(P_c(p))
12                 T.add{e|(e, P_c(p)) ∈ E_p}
13             end
14         end
15     end
16 end
17 return P_bypassed
```

For a sensitive function call, its taint sink is the key parameter of the function. We identify the function call information by tracking the 4th stack input (i.e., `argsOffset`) of the EVM `CALL` operation (as discussed in § 4.3). To recognize sensitive function calls, we track the values stored in the corresponding memory address `argsOffset` and check 1) whether the function signature matches the target functions (i.e., the signatures of `transferFrom, transfer, approve`); and 2) whether the parameters of the corresponding functions can be tainted from the taint source (the untrusted input). For example, if the parameters `_from` and `_to` in the `transferFrom` function are tainted, the adversary can call `transferFrom` with parameters determined by attacker-controlled input. Therefore, the attacker can transfer the users' token assets to an attacker-controlled address.

For privileged state variables, we assume that the state variables used for the *dataDriven* PCs are privileged for token behaviors. Because most of these variables are used for storing the user's token balance or approved token amount. If these variables are tainted from untrusted input, adversaries can simply modify these variables to change the balance of the users. There are situations where users deposit money into the contract to change their balance. Therefore we omit the state variables sinks in widely-used deposit functions such as `fallback` function. In this way, the taint analysis result in the bypass recognition can be reused in detecting the manipulation of privileged state variables.

To this end, we can recognize whether an unauthorized attacker can manipulate the parameters of token transfer functions or privileged state variables. Thus, the PRD vulnerability can be detected.

5 EVALUATION

In this section, we evaluate PrettySmart to answer the following three research questions (RQs):

(RQ1): How do smart contracts enforce permission control policies and how representative are the five types of permission constraints introduced by PrettySmart?

(RQ2): How PrettySmart performs in detecting the Permission Re-delegation vulnerability?

Table 2: Distribution of permission constraints in all `require` statements that occur more than 200 (or 100) times

Type	# >200		# >100	
	% Per.	No.	% Per.	No.
compare2Owner	61.78%	175,941	60.87%	186,831
dataDriven	29.44%	83,841	29.08%	89,245
groupBased	4.40%	12,531	5.31%	16,314
roleBased	1.87%	5,326	1.92%	5,897
queryOwner	1.55%	4,414	1.58%	4,852
others	0.97%	2,762	1.24%	3,804
# total	-	284,787	-	306,943

Table 3: Top 5 most used permission constraints.

Code	No.	Type
`msg.sender == owner`	58,320	compare2Owner
`_value ≤ balances[msg.sender]`	16,004	dataDriven
`_value ≤ allowed[_from][msg.sender])`	13,003	dataDriven
`_value ≤ allowance[_from][msg.sender])`	12,220	dataDriven
`msg.sender == governance`	11,316	compare2Owner

(**RQ3**): How effective is the proposed Permission Constraint Inference (PCI) method and the bypass-recognition method?

5.1 Experiment Setup

The benchmark used in our study contains three datasets: 1) *Open-source smart contracts*. We collected the verified source code of all open-source smart contracts from Etherscan (One of the most popular Ethereum blockchain explorer platforms) [6] till 23/04/2023. This results in 0.78 million open-source smart contracts on Ethereum. We conducted an empirical study on this dataset to answer RQ1. 2) *Reported Vulnerabilities* (RV). We collected a set of smart contracts from reported hacks and Common Vulnerabilities and Exposures (CVEs) and manually inspected whether they contained PRD vulnerabilities. For hack reports, we went through the hacks reported since 01/01/2020, and collected three reported hacks with six smart contracts. These hacks involved over 200 million USD (i.e., *TransitSwap* [4], *MEVBot* [3], and *Bancor* [2]). We collected vulnerable smart contracts with CVEs based on the benchmark given by SP-Con [28], which checked 531 CVEs and selected 17 smart contracts with permission bugs as the rest of them are mainly integer underflow or overflow vulnerabilities. AChecker [19] further confirmed that two of them are not exploitable, resulting in 15 smart contracts. Additionally, since the main contract does not inherit the vulnerability in CVE-2020-17753 in a subcontract, the vulnerable function is not compiled into the contract's runtime bytecode. Therefore, we omit this case since bytecode-based detectors cannot work on it. As a result, we collected 20 smart contracts and conducted experiments on this dataset to answer RQ2 and RQ3. 3) *Smartbugs-wild dataset* [14], which contains 47,518 smart contracts from Ethereum. Among them, 3,801 smart contracts were marked to have access control bugs by several analysis tools. However, these labels are not ground truth. This dataset was also used to evaluate the effectiveness of PrettySmart in detecting permission bugs (i.e., RQ2). All experiments were conducted on a Ubuntu 20.04.1 LTS workstation equipped with an Intel i9-10980XE CPU and 256GB RAM.

5.2 Representativeness of introduced permission constraints

We first investigate how smart contracts enforce permission control policies through an empirical study. Moreover, we show the representativeness of the permission constraints introduced in § 4.2.

This study was conducted on an *open-source smart contracts* dataset, in which we collected 0.78 million smart contract source codes and deduplicated them to obtain 0.43 million unique smart contract source codes. Our analysis focused on all the `require` statements with the keyword `msg.sender` (§ 4.2 gives the rational of this keyword choice). In this way, 512,689 `require` statements are extracted from the 0.43 million unique smart contracts. We counted the occurrences of each `require` statement and manually analyzed statements occurring ≥ 200 times. As a result, we analyzed the functionality of 284,787 `require` statements. We found that most of the permission control policies used for smart contracts can be divided into two categories: *address restriction* and *data-driven restriction*. Depending on the implementations, these two categories can be further divided into five specific types as introduced in § 4.2.

The first two columns of Table 2 report the distribution of these PCs. The *compare2Owner* type accounts for the majority, i.e., 61.78% of the analyzed permission control policies adopting this type. The second widely-used permission constraint is the *dataDriven* constraint, accounting for 29.44% of the analyzed permission control policies. The *queryOwner*, *groupBased*, and *roleBased* PCs account for 4.40%, 1.87%, and 1.55% of the analyzed results, respectively. Moreover, 0.97% of the PCs belong to neither of the above types. Table 3 lists the top 5 most used PCs. All the five PCs are of the *compare2Owner* type or *dataDriven* type and all of them occurred more than 10,000 times in smart contracts. Moreover, the 2nd, 3rd, and 4th PCs listed in Table 3 have strong correlation with specific token implementations. For example, the `allowance` or `allowed` variable is a typical variable used in ERC token contracts to record the amount of allowed tokens to be spent from one user to another after calling the `approve()` function introduced in § 2. Moreover, we noticed in our dataset that the top 10 popular implementations of the *dataDriven* type are all related to token implementations. In most cases, they check if the `balances` or the `allowance` of a user is enough in token transfer functions. Thus, these data-driven token-specific constraints can be covered by the proposed pattern.

There might be threats due to misunderstanding of PCs. To provide a more stable result, we introduced a validation process in our categorization following the design of [10]. This process consists of two iterations: 1) We randomly chose 20% of the statement groups. We employ two developers with more than two years of smart contract development experience to determine the categorization. They first read the code to understand in what cases the `require` statement succeeds. Then they discuss the permission policy implemented by the statement. In case of an unclear permission policy, it is treated as an undetermined type (i.e., *others*). After this iteration, we obtained the above five categories. 2) The same developers independently categorized the remaining 80% of the statement groups. We use Cohen's Kappa index [13] to measure the agreement between the classification results. The overall Kappa value is 0.99, indicating a strong agreement.

We performed the other two studies to analyze the potential bias brought by the threshold of occurrence and keyword choices. 1) **Threshold of occurrence.** We analyzed `require` statements occurring ≥ 100 times to see if the conclusion still holds under a different occurrence threshold. As shown in the 3rd and 4th columns of Table 2, there are 306,943 corresponding statements and the order of the permission control policies stays unchanged. Moreover, the proportion of each policy does not change much (<%1) when the threshold changes to 100. 2) **Keyword choice.** It is possible for other statements to be used for permission control. We treated `if-then` conditions as a typical alternative and analyzed their occurrence. Similarly, we counted the occurrence of these conditions with `msg.sender` that occurs ≥ 100 times. As a result, there are only 18,110 corresponding statements, which accounts for less than 6% of the occurrence of `require` statements. Moreover, we found that over 70% of the `if-then` conditions can fit into the *compare2Owner* and *Datadriven* patterns. Considering the bytecode forms, the only difference between `if-then` and `require` implementations is that the conditional jump operation in § 4.2 may not lead to a `REVERT` operation. Therefore, PrettySmart can fit the majority of these permission implementations with minor changes.

Answer to RQ1: The five introduced permission constraints have dominated 98% of the permission constraint implementations in the scope of our analysis.

5.3 Effectiveness of PrettySmart

We evaluate the effectiveness of PrettySmart on the RV and *Smartbugs-wild* datasets [14]. We evaluate its performance with comparison of six state-of-the-art permission bug detectors: AChecker [19], SPCon [28], Mythril [31], Securify [36], Maian [33], and Slither [16]. Since most of the prior studies for permission bug detection focus on intra-contract analysis and cannot be directly applied to DApp hacks, which contain multiple smart contracts, we apply these tools to the core vulnerable contracts inside DApps and report whether they find permission bugs. We evaluated the precision score of PrettySmart on the *Smartbugs-wild* dataset by checking the number of True-Positives (TPs) of the detected results. It is difficult to evaluate the recall on this dataset because there are no ground-truth labels as discussed before. Due to the limitation of time and human resources, we evaluate the recall score on the RV dataset by checking how many vulnerable contracts are detected out of all the reported contracts.

Table 4 reports the results on the *RV* dataset. The results show that PrettySmart outperforms other existing tools in detecting PRD vulnerabilities in real-world attacks. As evidence, only PrettySmart identifies TransitSwap and Bancor vulnerabilities. The vulnerabilities in their code are exploited by manipulation of sensitive parameters of privileged functions and PrettySmart is the first tool for detecting such vulnerabilities. Similar to other detectors, PrettySmart failed to detect MEVBot. The main reason is that one of the contracts in MEVBot cannot be successfully decompiled by the decompiler for preprocessing. Meanwhile, we note in Table 4 that the above 3 reported vulnerabilities can only be triggered in the cross-contract. Thus, the detection of these vulnerabilities relies on *effective cross-contract analysis*. Although PRD-related cross-contracts do not occupy a large portion, such an analysis enables PrettySmart to conduct sound detection for PRD vulnerabilities.

Table 4: Results of detecting permission re-delegation bugs

Contracts	Slither	Maian	Securify	Mythril	SPCon	AChecker	Ours
TransitSwap†	△	△	△	△	△	△	▲
MEVBot†	△	△	△	△	△	△	△
Bancor†	△	△	△	△	△	△	▲
CVE-2018-10666	△	△	△	△	▲	▲	▲
CVE-2018-10705	△	△	△	△	▲	▲	▲
CVE-2018-11329	△	△	△	△	▲	▲	△
CVE-2018-19830	△	△	△	△	△	▲	▲
CVE-2018-19831	△	▲	△	△	▲	▲	▲
CVE-2018-19832	△	▲	△	△	▲	▲	▲
CVE-2018-19833	△	△	△	△	△	▲	▲
CVE-2018-19834	△	△	△	△	△	▲	▲
CVE-2019-15078	△	▲	△	▲	▲	▲	▲
CVE-2019-15079	△	△	△	△	▲	△	▲
CVE-2019-15080	△	△	△	△	▲	▲	▲
CVE-2020-35962	△	△	△	△	△	△	▲
CVE-2021-34272	△	△	△	△	▲	▲	▲
CVE-2021-34273	△	△	△	△	△	▲	▲
recall (%)	0	18	0	0	53	71	**88**

† Vulnerabilities in these contracts can only be triggered in the cross-contract context.

Some of the existing methods (e.g., Slither, Mythril, Securify, and Maian) have relatively weak performance in detecting PRD vulnerabilities. They adopted permission bug detection into their frameworks and proposed corresponding rules. Although most of these rules are straightforward, such as finding whether a state variable can be freely written, it is hard to apply them in a more complex scenario. Moreover, only SPCon and AChecker achieve a close performance to PrettySmart though they failed in TransitSwap and Bancor. According to the evaluation results, PrettySmart shows the *best performance* in detecting 15 of the 17 vulnerabilities and achieving the highest recall rate of 88%.

We also evaluated PrettySmart in detecting generic permission bugs on the SmartBugs-wild dataset. Table 5 shows the evaluation results. The first column reports the number of identified vulnerabilities by each tool. For Slither, Maian, Securify, and Mythril, their reported vulnerabilities can be found in the SmartBugs dataset. The reported vulnerabilities of SPCon and AChecker can be found in its `github` repository. As discussed in § 5.1, this dataset contains no ground truth labels and it is too expensive to manually go through all the results reported by the compared tools. For contracts reported by our analyzer, we went through all 673 reports to calculate the precise number of TPs and precision of PrettySmart. Similar to previous studies [19, 28], we leveraged statistical methods for calculating the precision of compared tools except for SPCon and Maian (since they report 44 and 45 positives). Specifically, we sampled a subset from the reported vulnerabilities with a 95% confidence level and a confidence interval of 5% for each tool. We then manually confirmed the precision of the sampled subset. In total, we checked 331, 237, 284, and 254 smart contracts for Slither, Securify, Mythril, and AChecker. We observe that PrettySmart achieves the highest precision (90%). 611 out of the 673 detection results are TPs, where 302 of them are also PRD vulnerabilities. Further, only SPCon and AChecker achieve close precision to ours (80% and 81%).

Table 5: Permission bug detection results on SmartBugs-wild

Tool	# Reported	# Sampled	Precision	# Overlap
Slither	2,361	331	21%	79
Maian	44	44	64%	21
Securify	614	237	22%	9
Mythril	1076	284	39%	305
SPCon	45	45	80%	21
AChecker	624	254	81%	228
PrettySmart	673	673	90%	-

Among the 673 smart contracts, 163 of them are uniquely detected by our PrettySmart. We further inspected these newly detected smart contracts and confirmed that 118 of them are TPs. The distribution is as follows: 1) In 109 contracts, adversaries can manipulate the parameters of privileged functions and increase their token balances in the end. These vulnerabilities are newly detected because PrettySmart is the first to consider how accessible functions can be manipulated for malicious use. 2) In 30 contracts, adversaries have abnormal accessibility to state variables but other tools encountered timeout or other errors. These contracts may contain more than 1 thousand lines of code, resulting in timeout for tools with complex computational overhead such as symbolic execution. 3) In 15 contracts, there is at least one cross-contract call with a determined address. In these cases, the vulnerability in one of the smart contracts is triggered in another smart contract. Meanwhile, we evaluated the overlaps of the detection results between our method and the compared tools. The number of overlaps is shown in the last column of Table 4. As an illustration, fewer than 50% of our detected results can be covered by another individual tool.

```
1  address winner;
2  uint256 timeLock;
3  function () payable external {
4      require(msg.value >= 0.1 ether);
5      timeLock = now + 6 hours;
6      winner = msg.sender;
7  }
8  function claim() public {
9      require(msg.sender == winner);
10     require(now >= timeLock);
11     msg.sender.transfer(address(this).balance);
12 }
```

Figure 11: A false positive caused by underestimation of permission constraint for sensitive function.

False Positives (FPs) are mainly caused by two reasons: 1) *Underestimation of the PCs.* PrettySmart proposes PCI to detect sensitive operations that lack proper permission protection although there are situations that the PCI method cannot cover. In § 5.2, we have discussed the impact of keyword and threshold selection. We take `if-then` conditions as a case study to show our ability to extend to PCs based on a conditional jump with `msg.sender`. However, PrettySmart cannot cover PCs that do not restrict the address of the contract caller. For example, Figure 11 shows a game contract example of an underestimation scenario of PCI, in which winner address of the game contract has the privilege to take all the balances (line 11) after a certain timeLock (line 10). In this case, the sensitive operation `transfer` is guarded by two protections: a permission check for the caller's address and a permission check for time. PrettySmart successfully extracted the address restriction check but failed to extract the time check. This constitutes a *threat to validation* since PrettySmart aims to infer the address-based PCs; 2) *Imprecision of the decompilation result.* PrettySmart is built upon the decompilation result of Gigahorse [20]. There are occasions that the storage slot for `SSTORE` operation is imprecisely inferred, thereby causing imprecise recognition of taint sink in § 4.2.

Answer to RQ2: PrettySmart finds more vulnerabilities while achieving better precision (90%) than state-of-the-art methods.

5.4 Effectiveness of the Permission Constraint Inference and bypass recognition

A main concern of PrettySmart's effectiveness lies in whether the identified PCs in § 4.2 can cover most of the real-world cases. We evaluate the effectiveness of PCI by manually checking its success rate on the RV dataset. Table 6 reports the results. Specifically, we manually look up the public functions of each smart contract and check the number of PCs adopted by these functions. Further, we apply PCI to these smart contracts and record the results. As shown in Table 6, PrettySmart successfully inferred the PCs adopted by smart contracts in most cases, where the results marked with '-' represent no source codes available. Our analyzer precisely inferred all the PCs used in 12 out of the 15 smart contracts and missed only 6 of the 75 PCs overall. For those contracts with only bytecode available, it is hard to obtain the ground-truth (GT) number of PCs. Moreover, the inferred result also shows the distinction between business and personal contracts. For example, business contracts, such as TransitSwap and token contracts in CVEs, usually provide users with various interfaces. Thus, they tend to put fewer PCs on many public functions. On the other hand, most of the public functions of personal contracts (e.g., MEVBot) are restricted since they are developed for individual use.

We also evaluate bypass recognition by manually comparing the inferred result with contract source codes. The results are shown in the last two columns of Table 6, where "Inferred" denotes # of inferred PCs and the bypassed PCs. As shown in Table 6, PrettySmart achieves high accuracy in recognizing bypassed PCs. Only one bypassed constraint in CVE-2018-19831 is missed. We also observe that the DApps in reported hacks barely involve bypassed PCs. This is because these DApp vulnerabilities are not triggered by abnormal function accesses but by malicious exploitation of publicly accessible functions. The observation also confirms our motivation for detecting publicly-accessible functions to be exploited and manipulated to compromise the security of smart contracts.

Answer to RQ3: The proposed PCI methods can effectively detect the PCs and bypassed PCs.

6 RELATED WORK

Smart contract analysis has received increasing attention recently. Many static analysis methods [7, 11, 23, 29, 36] and dynamic methods [12, 22, 32, 35] have been proposed to perform security

Table 6: Success rate of PC inference and bypass analysis

Contracts	P.C. Inference		Bypass Detection	
	GT	Inferred (Acc.)	GT	Inferred (Acc.)
TransitS.1	-	7	-	0
TransitS.2	-	8	-	0
TransitS.3	-	7	-	0
MEVBot1	-	14	-	0
MEVBot2	-	-	-	-
Bancor	0	0 (100%)	0	0 (100%)
CVE-2018-10666	7	7 (100%)	5	5 (100%)
CVE-2018-10705	6	6 (100%)	5	5 (100%)
CVE-2018-11329	2	2 (100%)	0	0 (100%)
CVE-2018-19830	6	3 (50%)	3	3 (100%)
CVE-2018-19831	12	12 (100%)	12	11 (92%)
CVE-2018-19832	7	5 (71%)	5	5 (100%)
CVE-2018-19833	4	4 (100%)	3	3 (100%)
CVE-2018-19834	5	5 (100%)	3	3 (100%)
CVE-2019-15078	9	9 (100%)	5	5 (100%)
CVE-2019-15079	0	0 (100%)	0	0 (100%)
CVE-2019-15080	4	4 (100%)	4	4 (100%)
CVE-2020-35962	5	5 (100%)	0	0 (100%)
CVE-2021-34272	5	5 (100%)	5	5 (100%)
CVE-2021-34273	3	2 (66%)	1	1 (100%)
Total	75	69 (92%)	51	50 (98%)

analysis for smart contracts. Among these existing methods, taint analysis has been widely used to detect various vulnerabilities such as reentrancy. Sereum [34] and Slither [16] were the early tools that applied taint tracking to analyze the data dependency of smart contract executions. Although most of these tools focus on intra-contract analysis, a set of analyzers have recently been proposed for cross-contract analysis, including Clairvoyance [39], SmartDagger [27], and SAILFISH [7]. Clairvoyance empirically studied typical FPs of prior methods and leveraged inter-procedural taint analysis to reduce FPs. SmartDagger worked on bytecode and proposed variable semantic recovery to improve the effectiveness of cross-contract vulnerability detection. Symbolic execution methods, such as Oyente [29], Mythril [31], and ETHBMC [18] have been proposed to explore the execution paths based on constraint solving. To improve the scalability of symbolic execution, Park [41] has been proposed as a parallel symbolic execution framework with better efficiency and coverage. Dynamic methods [12, 22, 32, 35] have also been proposed for vulnerability detection with an aim at generating good test cases and synthesizing effective test oracles.

Permission bug detection has been studied for many years. Most of these studies focus on the security of permission systems for Android and Linux systems. Felt et. al. [17] first investigated the permission re-delegation problem for the Android system. Several following studies further investigated this problem for Android applications [15, 24, 40] and frameworks [25, 38]. Under the context of smart contracts, most of the prior studies [16, 29, 31, 36] include permission bugs in their frameworks and find whether a state variable can be modified by any user or whether there is an unrestricted `DELEGATECALL`. SPCon [28] further detects the flaws of high-level permission policy by recovering permission policy from transaction records and finding abnormal accessibility from conformance testing. AChecker [19] leverages taint analysis to detect vulnerabilities for access control policies of the address restriction type. However, existing tools mainly focus on detecting abnormal accessibility of functions and state variables. None of them detect whether an accessible function can be manipulated by adversaries.

7 CONCLUSION

This paper proposes PrettySmart, a bytecode-level static analyzer for detecting PRD vulnerabilities for token behaviors in smart contracts. Specifically, we first conduct an empirical study based on 0.43 million open-source smart contracts and find five types of widely-used permission constraints, which dominate 98% of studied contracts. These permission constraints are further divided into bypassed and not-bypassed ones through taint analyzing for their dependent state variables. As a result, PrettySmart detects the PRD vulnerability by taint analysis of the parameters of sensitive functions or privileged state variables in smart contracts. PrettySmart is evaluated on real-world smart contracts collected from reported vulnerabilities and a large-scale dataset of real-world smart contracts. The experimental results show that PrettySmart outperforms existing methods by achieving the highest precision and finding 118 new vulnerabilities.

ACKNOWLEDGMENTS

The work described in this paper is partially supported by the Technology Program of Guangzhou, China (No. 202103050004), the National Natural Science Foundation of China (62032025), and the COMP Department Start-up Fund of Hong Kong Baptist University.

REFERENCES

[1] 2020. *Panoramix.* Retrieved January 10, 2024 from https://github.com/eveem-org/panoramix
[2] 2022. *Bancor Network Hack 2020.* Retrieved January 10, 2024 from https://medium.com/1inch-network/bancor-network-hack-2020-3c71444fd59d
[3] 2022. *Hack Analysis: 0xbaDc0dE MEV Bot.* Retrieved January 10, 2024 from https://medium.com/immunefi/0xbadc0de-mev-bot-hack-analysis-30b9031ff0ba
[4] 2022. *TRANSIT SWAP - REKT.* Retrieved January 10, 2024 from https://rekt.news/transit-swap-rekt/
[5] 2023. *ERC-20 TOKEN STANDARD.* Retrieved January 10, 2024 from https://ethereum.org/en/developers/docs/standards/tokens/erc-20/
[6] 2023. *Etherscan.* Retrieved January 10, 2024 from https://etherscan.io/
[7] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP).* IEEE, San Francisco, CA, USA, 161–178. https://doi.org/10.1109/SP46214.2022.9833721
[8] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 454–469. https://doi.org/10.1145/3385412.3385990
[9] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv:1809.03981 [cs.PL]
[10] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. https://doi.org/10.1109/TSE.2020.2989002
[11] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19).* Association for Computing Machinery, New York, NY, USA, 1503–1520. https://doi.org/10.1145/3319535.3345664
[12] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2022. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21).* IEEE Press, 227–239. https://doi.org/10.1109/ASE51524.2021.9678888
[13] J. Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37.

[14] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[15] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2379–2396. https://www.usenix.org/conference/usenixsecurity20/presentation/elsabagh

[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. 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, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[17] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *20th USENIX Security Symposium (USENIX Security 11)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/usenixsecurity11/permission-re-delegation-attacks-and-defenses

[18] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2757–2774. https://www.usenix.org/conference/usenixsecurity20/presentation/frank

[19] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 945–956. https://doi.org/10.1109/ICSE48619.2023.00087

[20] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

[21] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: Advanced Decompilation of Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 77 (apr 2022), 27 pages. https://doi.org/10.1145/3527321

[22] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 531–548. https://doi.org/10.1145/3319535.3363230

[23] Zheyuan He, Shuwei Song, Yang Bai, Xiapu Luo, Ting Chen, Wensheng Zhang, Peng He, Hongwei Li, Xiaodong Lin, and Xiaosong Zhang. 2023. TokenAware: Accurate and Efficient Bookkeeping Recognition for Token Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 26 (feb 2023), 35 pages. https://doi.org/10.1145/3560263

[24] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler. 2020. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 271–287. https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez

[25] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. 2022. FReD: Identifying File Re-Delegation in Android System Services. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1525–1542. https://www.usenix.org/conference/usenixsecurity22/presentation/gorski

[26] Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise Static Modeling of Ethereum "Memory". *Proc. ACM Program. Lang.* 4, OOPSLA, Article 190 (nov 2020), 26 pages. https://doi.org/10.1145/3428258

[27] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: A Bytecode-Based Static Analysis Approach for Detecting Cross-Contract Vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3533767.3534222

[28] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding Permission Bugs in Smart Contracts with Role Mining. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 716–727. https://doi.org/10.1145/3533767.3534372

[29] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[30] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2022. Pluto: Exposing Vulnerabilities in Inter-Contract Scenarios. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4380–4396. https://doi.org/10.1109/TSE.2021.3117966

[31] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. In *9th Annual HITB Security Conference (HITBSecConf)*, Vol. 54.

[32] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[33] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663. https://doi.org/10.1145/3274694.3274743

[34] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2018. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. arXiv:1812.05934 [cs.CR]

[35] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2023. 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* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/3551349.3560429

[36] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[37] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[38] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 623–634. https://doi.org/10.1145/2508859.2516728

[39] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2021. 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* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1029–1040. https://doi.org/10.1145/3324884.3416553

[40] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. 2018. Invetter: Locating Insecure Input Validations in Android Services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1165–1178. https://doi.org/10.1145/3243734.3243843

[41] Peilin Zheng, Zibin Zheng, and Xiapu Luo. 2022. Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 740–751. https://doi.org/10.1145/3533767.3534395