Finding Permission Bugs in Smart Contracts with Role Mining

Ye Liu
li0003ye@e.ntu.edu.sg
Nanyang Technological University
Singapore

Shang-Wei Lin
shang-wei.lin@ntu.edu.sg
Nanyang Technological University
Singapore

Yi Li
yi_li@ntu.edu.sg
Nanyang Technological University
Singapore

Cyrille Artho
artho@kth.se
KTH Royal Institute of Technology
Sweden

ABSTRACT

Smart contracts deployed on permissionless blockchains, such as Ethereum, are accessible to any user in a trustless environment. Therefore, most smart contract applications implement access control policies to protect their valuable assets from unauthorized accesses. A difficulty in validating the conformance to such policies, i.e., whether the contract implementation adheres to the expected behaviors, is the lack of policy specifications. In this paper, we mine past transactions of a contract to recover a likely access control model, which can then be checked against various information flow policies and identify potential bugs related to user permissions. We implement our role mining and security policy validation in tool SPFCon. The experimental evaluation on labeled smart contract role mining benchmark demonstrates that SPFCon effectively mines more accurate user roles compared to the state-of-the-art role mining tools. Moreover, the experimental evaluation on real-world smart contract benchmark and access control CVEs indicates SPFCon effectively detects potential permission bugs while having better scalability and lower false-positive rate compared to the state-of-the-art security tools, finding 11 previously unknown bugs and detecting six CVEs that no other tool can find.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging;
• Security and privacy → Access control.

KEYWORDS

Smart contract, access control, role mining, information flow policy.

1 INTRODUCTION

Smart contracts are computer programs that run on a blockchain platform and manage large sums of money, carry out transactions of assets, and govern the transfer of digital rights between multiple parties. Ethereum [70] and EOS [31] are among the most popular blockchain platforms which support smart contracts and have them applied in many areas, such as finance, supply chain, identity management, games, etc. As of January 20, 2022, there are over 48 million smart contracts deployed on Ethereum, which is a 2.3-fold increase from just two years ago [5]. These smart contracts have enabled 3,886 decentralized applications (DApps) serving about 180.13k daily active users [10].

The security of smart contracts has been at the forefront of attention, ever since their adoption in the management of massive monetary transactions. A large class of smart contract security issues occurred due to low-level coding errors, such as reentrancy [55], integer overflow/underflow [15], incorrect exception handling [7], and gas-related issues [19]. While many of these bugs caused devastating monetary losses and made the headlines [55], they have been widely studied [21] and are relatively easy to address. Such low-level bugs can often be captured by common vulnerability patterns [13] and avoided by adopting the suggested coding practices [57, 69]. However, another class of security issues stems from flaws in high-level security policy design and enforcement; such flaws are more subtle to discover and more difficult to address.

One difficulty in validating the conformance to such policies, i.e., whether the contract implementation adheres to the expected behaviors, is the "test oracle problem" [14]—there is a lack of policy specifications. The current practice is to implement intended access control policies with ad-hoc Solidity [57] (the programming languages used to develop Ethereum smart contracts) idioms, such as the "require" statements, to check if the address of a user is within a predefined whitelist. Many permission bugs [53] are results from this ad-hoc approach. In particular, when the number of roles and the complexity of the access control patterns increase, it is difficult for developers to avoid mistakes, giving rise to vulnerabilities.

In this paper, we address this problem by mining past transactions of a contract to recover a likely access control policy specification, which is then used to validate the contract implementation and identify potential bugs related to user permissions. To this end, we implemented a security policy validator, SPFCon, based on role mining from past transaction histories. Because of the transparency and immutability of blockchain transactions, the transaction histories of a smart contract application from its initial deployment are...
always available. These historical transactions contain benign user interactions, assuming the contract has not yet been attacked by any malicious party. As is shown in Fig. 1, the key idea behind SPCon is that we first perform role mining on the transaction histories to reverse engineer role structures which describe the different user groups, the permissions entitled to each group, and potential hierarchies among them. Then, access-control policy specifications are constructed based on the role structures and relevant information flow policies, such as integrity or separation of duty [48]. Through conformance testing, SPCon validates the actual contract implementation against the policy specifications. Since historical transactions only under-approximate the behaviors allowed by the contract implementation, any discrepancy discovered indicates potential policy violations that may occur in the future.

There are two usage scenarios of SPCon in practice: (1) detecting bugs in deployed on-chain smart contracts based on existing histories—buggy contracts are to be safely destroyed to prevent potential money loss, and (2) during internal testing before the deployment, such as a user acceptance testing—such testing exhibits typical usage patterns that are not attacks, allowing to fix vulnerabilities before deployment and detect attacks in production.

The main challenge for SPCon is to obtain high-quality underlying role structures from contracts’ historical transactions, which may not provide full information on every user’s access patterns. We formulate this partial-observation role mining task as an optimization problem, in which we optimize over two quality metrics, namely, the role-similarity error and the role-consistency error. Given the complexity of the optimization problem, we use a genetic algorithm (GA) to find an approximate solution. To improve the performance of the GA, we augment the optimization model with additional constraints: (1) the mined role model must subsume all observed user permissions from the historical transactions, and (2) users that share similar permission patterns can be represented by the same role. Empirically, the GA improves the role structure quality within a reasonable amount of time. The experimental results on labeled smart contract role mining benchmark demonstrates that SPCon is able to mine role structures from contracts’ historical transactions with better accuracy than existing state-of-the-art role mining approaches [23, 41, 54, 72]. Moreover, the experimental evaluation on real-world smart contract benchmark, namely, SBwild [22] and access control CVEs, indicates that SPCon effectively detects potential permission bugs while having better scalability and lower false-positive rate compared to the state-of-the-art security tools [1–4, 8, 16, 56, 60].

Contribution. Our main contributions are summarized as below.

- We propose SPCon, a tool targeting permission bugs in smart contracts. It mines role structures from historical transactions, and thus enables conformance testing without specifications.
- We define the partial-observation role mining problem, where generalizable role structures need to be inferred without full user access information. We pose it as an optimization problem and design an effective solution based on genetic algorithms.
- We collect and label a smart contract role mining benchmark. We implement SPCon and evaluate it on the sampled 50 smart contracts within the role mining benchmark. The results show that SPCon can largely reduce the number of mined roles and mine the role structures with better accuracy compared to the existing role mining tools.
- The policy specifications produced by SPCon can be used to enhance existing testing tools. We evaluate SPCon’s bug detection capability on real-world smart contract benchmark SBwild. The results show that SPCon achieves the highest accuracy of permission bug detection and finds 11 previously unknown permission bugs in SBwild. Moreover, SPCon detects six more previously confirmed CVEs that cannot be found by existing tools. The dataset, raw results and prototype used are available online: https://doi.org/10.21979/N9/MBHBCI.

Outline. This rest of this paper is organized as follows. Section 2 motivates our work with a recent real-world permission bug. Section 3 presents necessary background on role mining. We then provide details on the SPCon framework in Sect. 4, and present evaluation results in Section 5. Finally, we discuss related works in Sect. 6 and conclude this work in Sect. 7.

2 MOTIVATING EXAMPLE

On May 7, 2021, a smart contract ProfitSharingRewardPool, used by a Decentralized Finance (DeFi) platform named ValueDefi, was hacked due to missing a line of code and lost around six million dollars [11]. ProfitSharingRewardPool is written in Solidity [57]; Fig. 2 shows its simplified source code. ValueDefi used this pool contract for profit sharing with its users. The contract defines several modifiers to restrict user access, including “onlyOperator” (Line 9) and “notInitialized” (Line 12). Unlike most other contracts, ProfitSharingRewardPool requires an explicit initialization after the contract deployment, because it does not provide a specialized constructor. Before the initialization, the “initialized” flag remains false, while other fields remain uninitialized (Lines 3 to 7).

To properly initialize the contract, the contract owner should invoke the “initialize” function (Lines 14 to 24), which comes with the “notInitialized” modifier to restrict other users’ access after the initialization is performed (Line 12). Apart from configuring the staked and liquidity tokens (Lines 18 to 19), the contract owner may configure administrator roles—“reserveFund” and “operator”—during the initialization stage. Yet, since the statement “initialized = true” (Line 23) was missing, a malicious
pragma solidity ^0.6.12;

contract ProfitSharingRewardPool {
    address public operator;
    address public reserveFund;
    address public exchangeProxy;

    bool public initialized = false;
    address public liquidityToken, stakeToken;

    /********** Modifiers ********** /

    modifier onlyOperator () {
        require(operator == msg.sender);
        _;
    }

    modifier onlyExchangeProxy () {
        require(exchangeProxy == msg.sender);
        _;
    }

    modifier onlyReserveFund () {
        require(reserveFund == msg.sender);
        _;
    }

    modifier notInitialized () {
        require(!initialized);
        _;
    }

    function initialize () public {
        // validate the protecting modifier by realizing that the condition
        // still holds after the function is already protected by a modifier,
        // and that we care about the most are contract state variables,
        // recording critical information such as the balance values, role assignments,
        // and other relevant preliminaries required for the rest of the paper.
        // For now, let’s assume that the attacker can modify the token configurations
        // and perform privilege escalation by setting himself as the “operator”.
        // Existing smart contract security analysis tools mostly rely on
        // generic bug patterns; thus, they may face challenges in detecting
        // this permission bug without an accurate contract specification.

        // Figure 2: The simplified code of ProfitSharingRewardPool.

        // TABLE 1: Role structures of ProfitSharingRewardPool.

        Users (UA) | Permissions (PA)
        --- | ---
        { Operator } | { initialize, setOperator, setExchangeProxy, setReservedFund, depositFor, allocateMoreRewards }
        { ExchangeProxy } | { setExchangeProxy, depositFor }
        { ReserveFund } | { setReservedFund, allocateMoreRewards }
        { Normal Users } | { deposit, withdraw, claimReward }

        Figure 3: The ProfitSharingRewardPool’s security lattice.

        3.1 Role-Based Access Control Model

        RBAC has been well studied in the last twenty years since the
        establishment of the NIST RBAC standard in 1995 [50]. We borrow
        the standard definition as follows.

        Definition 1 (RBAC [52]). An RBAC model M can be defined
        as a tuple (U, R, P, PA, UA) with the following components:
        U is a set of users, R is a set of roles, P is a set of permissions,
        PA ⊆ P × R is the permission-to-role assignment relation, and
        UA ⊆ U × P is the user-to-role assignment relation.

        A role is typically viewed as a semantic construct around which
        access control policy is formulated [52]. Notice that RBAC is policy-
        neutral and can be used to implement various types of security
        policies. This owes to the flexible granularity of the permission
        concept, which could either be a coarse-grained job function or a
        fine-grained data read/write. In the smart contract context, permis-
        sions can be enforced at both the function- and statement-level.

        Table 1 shows the access control model of ProfitSharingReward-
        Pool. There are four roles, namely, “Operator”, “ExchangeProxy”,
        “ReserveFund”, and “Normal Users”. Each role is granted a set of per-
        missions: for example, “Normal Users” can only call permissionless
        functions such as “deposit” and “withdraw”, while “Operator” can
        initialize the contract, set users for other roles, etc.

        3.2 Information Flow Policy

        Information flow policies are concerned with the flow of informa-
        tion from one object to another [51]. In smart contracts, the objects
        that we care about the most are contract state variables, recording
critical information such as the balance values, role assignments,
and asset prices. An important class of information flow policies can be defined on top of a security lattice.

**Definition 2 (Security Lattice [49]).** There is a finite lattice of security labels $SC$ with a partially ordered dominance relation $\geq$ and a least upper bound operator.

As RBAC is policy neutral, it can be used to articulate a wide range of information flow policies [49]. We may recover a security lattice from the role structures, such that the security labels correspond to roles ($R$) and the dominance relation is defined over the write sets of roles: i.e., $r_i \geq r_j$ if and only if $\text{write}(r_i) \supseteq \text{write}(r_j)$, where $\text{write}(r_i)$ denotes the set of variables that $r_i$ can write.

For example, Fig. 3 shows a security lattice based on the roles in ProfitSharingRewardPool, where “Operator” and “Normal Users” are the top and bottom roles, respectively. If an integrity policy is to be enforced among them, i.e., lower roles should not be able to write data owned by higher roles, then normal users should not write to “stakeToken”, etc. Yet, due to the buggy “initialize” function, normal users are able to re-initialize the contract, thus violating the integrity policy.

### 3.3 Role Mining Problem (RMP)

The purpose of RMP is to utilize the observed user access information captured by the user permission assignment matrix $UPA$, to infer the decomposed user roles ($UA, PA$), such that the decomposition exactly describes the $UPA$ and the number of roles are minimized. Let $a_{ij}$ denote the entry ($i, j$) of $UPA$. Then $a_{ij} = 1$ indicates that the $i$th user has the $j$th permission. Typically, users sharing the same set of permissions should be classified into the same role. A more generalized version of the RMP allows noises in the decomposition. More formally, the $\delta$-Consistency Role Mining Problem ($\delta$-RMP) was first defined by Vaidya et al. [63].

**Definition 3 ($\delta$-RMP [63]).** Given a set of users $U$, a set of permissions $P$, and a user-permission assignment matrix $UPA$, the problem of $\delta$-RMP is to find a set of roles $R$, a user-to-role assignment matrix $UA$, and a role-to-permission assignment matrix $PA$ such that the number of roles $|R|$ is minimized, and the following inequality is satisfied.

$$||UA \otimes PA - UPA||_1 \leq \delta,$$

where $|| \cdot ||_1$ denotes the $L^1$ norm and $\otimes$ refers to Boolean matrix multiplication.

A solution to $\delta$-RMP allows a limited number of mismatches below the given threshold $\delta$. Here, we introduce four well-known role mining approaches [42] that assume a fully-observed user permission assignment and find solutions satisfying $\delta = 0$. ORCA [54] clusters users and permissions hierarchically according to the maximal overlap among the users, to form roles without minimizing the number of roles. The HP Labs proposed a role minimization approach (HPr) [23] for finding minimal number of roles to cover the user-permission assignment. The Graph Optimization (GO) algorithm [72] views the role mining problem as a graph optimization problem whose objective is to minimize the number of roles and the number of graph edges. Hierarchical Miner (HM) [41] is based on the ontology of formal concept analysis to mine roles with hierarchy, whose objective is to minimize a predefined complexity metric to create optimal roles.

Our proposed role mining approach aims to reverse engineer high-quality roles based on the limited partial observations from the historical transactions. Therefore, we allow a non-zero $\delta$, and at the same time, optimize over quality metrics (more details are discussed in Sect. 4).

### 3.4 Role-Mining Evaluation Metrics

Evaluating the quality of a role-mining algorithm involves the comparison between the mined roles and the ground-truth roles. In general, a role can be viewed as a set of permissions. To compare the similarity between two sets of roles, one needs to decide a role-to-role mapping before the similarity between a pair of roles can be evaluated using, for example, the Jaccard Coefficient [32]. Vaidya et al. [64] define the similarity between two role sets as the average over the maximum similarity between each role in the smaller role set and any matched role in the larger role set. However, this similarity definition neglects the contribution of unmatched roles and thus usually prefers solutions that yield a larger number of mined roles [59]. To mitigate this issue, Takabi et al. [59] extended it by taking unmatched roles into consideration: they define the similarity between two role sets as follows.

**Definition 4 (Role-Set-Role-Set Similarity [59]).** Given two role sets $R_1$ and $R_2$, the similarity between them is determined in three steps. First, if the sizes of the two sets are equal, without loss of generality, assume that $R_1$ is the smallest set, then we have,

$$M = \left\{ \arg \max_{r_j \in R_2} \text{jaccard}(r_i, r_j) \right\} \forall r_i \in R_1,$$

which computes for each role in $R_1$ the similarity with its ideal match in $R_2$. Second, for the remaining $R_2$ that are not the ideal match with $R_1$, keep only those elements that match to some degree,

$$\tilde{M} = \left\{ r_j \mid \exists r_j \in (R_2 \setminus M) \cdot \max_{r_i \in R_1} \text{jaccard}(r_i, r_j) > \tau \right\},$$

where $\tau$ is the similarity threshold to take unmatched roles into consideration. Finally, the similarity between $R_1$ and $R_2$ is given as,

$$\text{Sim}(R_1, R_2) = \left( \sum_{r_i \in R_1} \max_{r_j \in \tilde{M}} \text{jaccard}(r_i, r_j) \right) + \sum_{r_j \in \tilde{M}} \max_{r_i \in R_1} \text{jaccard}(r_i, r_j) \left/ \left(||R_1|| + ||\tilde{M}||\right) \right..$$

When $\tau = 1$, role-set-role-set similarity is equivalent to the definition used by Vaidya et al. [64], which only averages similarities of the ideal matches. When $\tau = 0$, role-set-role-set similarity averages the similarity of all possible role pairs where the Jaccard Coefficient of each role pair is larger than zero.

## 4 Framework

In this section, we present SPCON, a framework for permission bug identification in smart contracts through role mining.

### 4.1 Overview

The overall workflow of SPCON consists of two major steps (see Fig. 1). First, during role mining, SPCON recovers role structures of the contract based on the observed user permission assignments.
(UPA) from the transaction histories. We assume that benign transaction histories are readily available for the deployed contract. For undeployed contracts, transactions can be generated through internal user acceptance testing. Second, SPCON performs conformance testing to validate contract implementation against the access control policy specification defined based on the mined role structures. We describe each step in detail in the rest of this section.

4.2 Partial-Observation Role Mining (PORM)

Most existing role mining techniques [40] assume a fully-observed user permission assignment, i.e., UPA contains all permissions assigned to each user. This assumption does not work well in the smart contract setting. In particular, a user is unlikely to access all functions within her permission, especially for permissionless functions, and different users of the same role (e.g., normal users) may access different subsets of their permissions. Treating transaction histories of smart contracts as a fully-observed permission assignment will likely result in more roles than necessary and incorrect role assignments.

In this paper, we propose the problem of partial-observation role mining (PORM), where the given permission assignment is assumed to contain only partial information. To derive a high-quality estimation of the role structures, we rely on a good balance between the two quality metrics, namely, the role-similarity error and the role-consistency error, to guide the role mining process.

In practice, users of the same role tend to share a similar access pattern, which can be captured quantitatively by the average frequency vector (AFV).

**Definition 5 (Average Frequency Vector).** Let \( r \in R \) be a role, \(|r|\) be the number of users of \( r \), and \( P \) be the permission set. Let \( n(r, p_i) \) denote the total number of times the permission \( p_i \in P \) gets exercised by the users of \( r \). The average frequency vector of \( r \), denoted by \( AFV(r) \), is a \(|P|\)-dimensional vector \( x \in \mathbb{R}^{|P|} \), where \( x_i = \frac{n(r, p_i)}{|r|} \).

The AFV of a role measures how frequently its users exercise different permissions, which serves as a signature of the role and should vary across different roles. The desired roles should distinguish users of different access patterns, therefore, resulting in low AFV similarities between different roles.

**Definition 6 (Role-Similarity Error: SimErr).** The role-similarity error of a mining result is defined as the maximum of the cosine similarities between any pair of roles’ AFVs:

\[
SimErr = \max_{r_i, r_j \in R} \cos(\text{AFV}(r_i), \text{AFV}(r_j)).
\]

Meanwhile, we aim to minimize the number of mismatches between the mined role structures \((UA, PA)\) and the given user permission assignment \(UPA\), namely the \(\delta\) value of \(\delta\)-RMP (see Def. 3). We define this type of error as follows.

**Definition 7 (Role-Consistency Error: DeltaErr).** Let \( UPA \) be the user permission assignment matrix and \((UA, PA)\) be the mined role structure. The role-consistency error is defined as:

\[
\text{DeltaErr} = \frac{||UA \otimes PA - UPA||_1}{||UA \otimes PA||_1},
\]

where \( || \cdot ||_1 \) denotes the \(L^1\) norm and \( \otimes \) refers to Boolean matrix multiplication.

**PORM as an Optimization Problem.** The RMP and \(\delta\)-RMP are both NP-Complete problems [62, 63]. The complexity of the PORM is at least as hard as that of the \(\delta\)-RMP, and we would like to ensure the generalizability of roles with \(SimErr\), while maintaining good consistency with \(\text{DeltaErr}\). Therefore, we pose the PORM as a multi-objective optimization problem to produce likely role structures achieving a good trade-off between the two error metrics.

Given a set of users \(U\) of size \(m\), a set of permissions \(P\) of size \(n\), and a user-permission assignment matrix \(UPA\) of size \(m \times n\), PORM is to infer the unknown RBAC configuration \((UA, PA)\), where \(R\) is a set of roles of size \(k\), \(UA\) is a user-role assignment matrix of size \(m \times k\), and \(PA\) is a permission-role assignment matrix of size \(k \times n\), which satisfies,

\[
\min \alpha \cdot SimErr + \beta \cdot \text{DeltaErr},
\]

s.t. \(UA \otimes PA \supseteq UPA\),

where \(\alpha\) and \(\beta\) are relative weights on the two error metrics. Equation (6) constrains that the mined roles \((UA \otimes PA)\) should at least include the permissions appearing in the partial observation \((UPA)\).

4.3 A Solution Based on a Genetic Algorithm

In this section, we introduce a genetic algorithm (GA) to find a good solution to the PORM. Our algorithm takes \(UPA\) as the main input, which can be obtained from the smart contract transaction histories. Since a valid solution needs to satisfy Eq. (6), we first group the largest possible number of users sharing the largest set of permissions together, as basic roles. This helps reduce the search space significantly. The basic roles guarantee to have \(\delta = 0\) and thus satisfy Eq. (6). Subsequently in the GA, basic roles are randomly merged to form larger roles, which only increases \(\delta\), and therefore Eq. (6) is never violated.

**Chromosome.** Given a set of basic roles \(\{r_0, \ldots, r_i, \ldots, r_{k-1}\}\), we encode the solution as a chromosome \(Chr[x]\) in the following form:

\[
Chr[x] = (x_0, x_1, \ldots, x_{k-1}),
\]

where \(x_i\) is the gene of \(r_i\). Each gene consists of \(\log_2 k\) bits, so there are at most \(k\) different gene values. Each gene value represents a specific cluster of basic roles. For example, when \(x_i = x_j\), the two basic roles \(r_i\) and \(r_j\) are to be merged to generate a final role.

**Fitness function.** We use the following fitness function to guide the search.

\[
\text{fitness}(x) = (\alpha \cdot \text{SimErr} + \beta \cdot \text{DeltaErr})^{-1},
\]

where \(\text{SimErr}\) and \(\text{DeltaErr}\) are defined in Eqs. (4) and (5), respectively.

**Selector, Crossover, and Mutator.** The three GA operators, selector, crossover and mutator [39], are designed as follows. The selector adopts the Tournament Selection method [38] to select the winner of each tournament to perform later crossover. We use a single-point crossover to exchange two chromosomes of the parents to generate new offspring in a manner where the front and back of the crossover point for the two chromosomes is exchanged. Our mutator performs a gene-level mutation, which flips a bit of a gene or swaps two genes in a chromosome. The initial population
The conformance testing process. The inputs include the contract functions ($F$, $V$, $S_0$), state variables, initial concrete state ($UB$, $PA$), and a user-defined bound for the test sequence length ($k$). The output is either a test sequence that exploits a permission bug or no bug is found within the given bound.

The security lattice $L$ is constructed according to the mined roles $R$ and the partial order over the set of state variables written by each role. Each role is associated with a set of permissions (functions) given by $PA$; therefore, we can map roles to their write sets via data-dependency analysis on the corresponding functions (Line 15). Note that, the write set of a function contains the written state variables excluding the ones read. A permission bug is reported if the information flow policy is violated on the path $Π$ (Line 17). We then return the shortest test sequence leading to the bug (Line 8). Otherwise, $S$ is updated, to take into account the state changes introduced by the function $f_i$ (Line 9). The conformance testing process continues, till all the test sequences have been tested or a permission bug is found.

**Example.** Examples of the security policies checked by SPCON include integrity and separation of duty. Integrity prevents critical information flowing from a low-security role to a high-security role, while separation of duty ensures that the privileged information owned by a role cannot be modified by other incompatible roles.

Figure 4 illustrates the process how SPCON detects the “initialize” attack to ProfitSharingRewardPool, which violates the integrity of the security lattice (see Fig. 3). Following the common practice, attackers are assumed from the lowest-security role, namely “Normal User”. In Fig. 4, there are four test sequences where each consists of two functions. The deployed contract is assumed to have been initialized once by the owner. Therefore, the initial state $S_0$ records the current concrete values of the state variables, such as “stakeToken” and “tokenBalance”. The first and the second test sequences, namely, $t_{s1}$ and $t_{s2}$, are infeasible, and there is no change made to $S$. This is because the attacker’s address will be rejected by the permission checks for “exchangeProxy” and “reserveFund”, respectively. The sequence $t_{s3}$ is feasible, and the attacker’s token balance in $S$ is updated to “X” and “X – Y” after “deposit” and “withdraw”, respectively. Yet, this does not result in any policy violation yet. In $t_{s4}$, the buggy “initialize” function is executed first, where the value of “stakeToken” is modified in $S$, thus breaking the integrity policy. Therefore, SPCON reports the shortest attack test sequence as “(initialize)”—as an indication of the permission bug.

**Implementation.** Step 1 in SPCON sets up the testing environment with a mirrored contract deployment from a peer node using the developer APIs provided by Etherscan [5], where we can fetch the most up-to-date contract states (Line 1) faithfully. Step 2, the function-level symbolic execution, is implemented based on Manticoore [2]; we substitute symbolic state variables with their concrete values captured in $S_0$. This avoids false positives from spurious contract configurations. As the number of possible test
sequences grows exponentially with length \( k \), we perform partial-order reduction on the test sequences according to their control-flow dependencies, which further reduces unnecessary test cases.

5 EVALUATION

To explore the capability of SPCon, we evaluated it based on the following research questions, comparing its performance to the state of the art:

RQ1: How accurately and efficiently does SPCon learn the likely RBAC model?
RQ2: How does SPCon perform in detecting permission bugs?
RQ3: Why do existing tools fail to detect many permission bugs, and how does our approach improve on this?

5.1 Experiment Setup

The benchmarks used in the evaluation are listed in Table 2. To answer RQ1, we collected a role mining benchmark for smart contracts with ground truth. We realized that most smart contracts are poorly documented, and their role structures are often implicit. To mitigate this issue, we collected smart contracts which use the AccessControl template of OpenZeppelin [6], because these contracts have to explicitly define their roles following the template. We searched all smart contracts whose source code use the AccessControl template via the Etherscan source code search service [9] on December 9, 2021. This resulted in 4,719 smart contracts on Ethereum that use the AccessControl template in their source code.

These contracts use the “onlyRole” and “hasRole” modifiers to label each privileged function with the corresponding roles. Our ground truth is largely derived based on these labels. But to avoid potential errors and incompleteness, we manually reviewed the labels on a smaller set of contracts. In particular, we selected the most used contracts having at least 1,000 historical transactions and this resulted in 228 smart contracts. Two authors independently labeled the role structures of these contracts with the third one to resolve the divergence of views. Due to time constraints, we sampled 50 smart contracts from the labeled benchmark and evaluated SPCon with other tools based on these 50 contracts to draw our conclusion of RQ1.

For RQ2, we evaluated the accuracy of SPCon on real-world smart contract benchmark \( SB^{\text{wild}} \) [22], a public data set consisting of 47,518 contracts from the Ethereum blockchain. Specifically, \( SB^{\text{wild}} \) includes 3,801 contracts marked as having access control bugs. Those access control bugs were detected by symbolic execution-based analysis tools such as Maian [1], Manticore [2], and Oyente [4], and program analysis tools including Securify [56] (dataflow properties), Slither [8] (taint tracking) and SmartCheck [60] (AST-level rules), as well as a hybrid analysis tool, Mythril [3]. Moreover, as of December 23, 2021, there are 531 smart contract CVEs, 19 of which contain access-control-related CVEs. We used these 19 CVEs to evaluate the capability of SPCon, as the rest of them are mainly caused by integer overflow or underflow, which is out of the scope of this work.

All experiments were conducted on an Ubuntu 20.04.1 LTS desktop equipped with an Intel Core i7 16-core processor and 32 GB of memory. The benchmark contracts and raw results are available at: https://sites.google.com/view/spcon/.

5.2 Results of the Experiments

We now discuss the results of the experiments in detail.

Results for RQ1. To answer RQ1, we evaluated three combinations of \( \alpha \) and \( \beta \), namely, \((0.4, 0.5), (0.5, 0.5)\) and \((0.6, 0.4)\), for the fitness function—see Eq. (7)—used by SPCon, since we believe the optimal solution should well balance SimErr and DeltaErr. As for the GA parameters, the population size is set to be 100, and the number of generations is 200. The mutation rate of the GA population is set to be 0.10 to avoid being stuck in a local optimum, and the crossover rate is 0.99. The role mining time budget is 20 minutes per contract for all the role mining tools. We compared SPCon with existing role mining tools HPi, ORCA, HM, and GO on the 50 sampled smart contracts of our role mining benchmark. We compared the mined roles and the ground truth roles using two metrics: \( \text{Num}_\text{Ratio} \) and \( \text{Sim}(\text{mined}\_\text{roles}, \text{groundtruth}\_\text{roles}) \). \( \text{Num}_\text{Ratio} \) is the ratio of the number of the mined roles to the number of roles in the ground truth. \( \text{Sim}(\text{mined}\_\text{roles}, \text{groundtruth}\_\text{roles}) \) measures the similarity between the roles of the mined result and the roles of the ground truth at a user-given threshold \( t \) (c. f. Def. 4).

Table 3 shows the evaluation on the role mining results. The first column is the role mining approach; we use the three aforementioned settings for SPCon. The next three columns show the average time cost, average number of roles, and average \( \text{Num}_\text{Ratio} \) per contract. The first four columns of the rest show the average similarity between the mined roles with respect to the ground truth at different given thresholds. Since privileged roles are critical to a security policy, the last four columns also present the similarity between mined privileged roles and the privileged roles of the ground truth, removing all the permissionless functions.
GO takes the longest time, around three minutes on average, while HPr is the fastest algorithm. ORCA generates the most roles (about 22 on average) and thus has the highest ratio of mined roles per actual role (7.17 on average). The reason is that ORCA uses a simple clustering analysis without any minimization goals. HM and GO also generate many more roles than the ground truth, with the ratio being 6.37 and 4.86, respectively. Although HPr has the best role similarity among the four existing tools, SPCon (0.4, 0.6) outperforms HPr in all metrics except for the runtime. This implies that SPCon (0.4, 0.6) mined more accurate roles than HPr, as SPCon (0.4, 0.6) also mined fewer unnecessary roles. SPCon (0.5, 0.5) generates fewer roles than SPCon (0.4, 0.6) with a similarity loss to some extent. We argue that SPCon (0.5, 0.5) is still better than HPr in the sense that SPCon (0.5, 0.5) reports much fewer roles than HPr, and the similarity of SPCon (0.5, 0.5) is comparable to that of HPr. SPCon (0.6, 0.4) has the lowest similarity for privileged roles, which downgrades the role mining accuracy. The reason is that, with higher weightage $\alpha$, SPCon (0.6, 0.4) could attempt to cluster different privileged roles to achieve a higher fitness score.

In summary, SPCon significantly reduces the number of excessive mined roles compared to other role mining approaches. SPCon outperforms the existing role mining approaches with respect to accuracy when we choose suitable ($\alpha$, $\beta$) combinations, such as (0.4, 0.6) or (0.5, 0.5). Moreover, SPCon runs efficiently, taking only half a minute on average for our examples. Due to the random nature of the GA, we performed 10 role mining experiments of SPCon (0.4, 0.6), and the mean and variance of the number of roles, ratio of the mined roles to the ground-truth roles, and similarity ($t=1$) are $\{6.776, 0.05\}$, $\{2.193, 0.004\}$, $\{0.553, 0.00008\}$, which implies that the results are robust.

**Answer to RQ1:** SPCon can accurately and efficiently reverse engineer likely RBAC models of smart contracts.

**Results for RQ2.** To answer RQ2, we evaluated SPCon on the contracts of $SB^{\text{wild}}$ with at least 50 transactions for the observation of a diversity of users behavior to mine high-quality roles for conformance testing. For SPCon, the length of test sequence $k$ is 2 which is same as the default setting of Manticore [2] and Mythril [3], and the test time budget for permission bug detection is set to 10 minutes per contract.

Table 4 shows the evaluation result on permission bug detection of $SB^{\text{wild}}$ of six pattern-based tools and SPCon. To avoid bias, we reused the original detection result of Slither, Securify, SmartCheck, Mythril, Maian and Manticore [21]. We elide Oyente, since it flags only two permission bugs. The columns in Table 4 show the different tool names, the number of reported permission bugs, the number of reported permission bugs agreed on by at least one other existing tool with the corresponding percentage, and the true-positive rate (confirmed by us).

SPCon reported 44 permission bugs among $SB^{\text{wild}}$, while Slither reported the most, namely 2, 356 contracts. However, Thomas et
al. [21] suggests a high number of false positives may exist in the detection results; its solution is to combine different tools to create a consensus to reduce the false positives. The agreement result shows that SPCON achieves the best precision. 75% of the results of SPCON are agreed upon by at least one other tool. Furthermore, we manually confirm those permission bugs to determine the true-positive rate for each tool.

As it is non-trivial to confirm all detection result of each tool, we manually confirmed all reported permission bugs by Maian, Manticore and SPCON. For Slither, Securify, SmartCheck and Mythril, we sampled 66, 61, 58, 64 reported permission bugs among their results to obtain a 90% confidence level and a margin of error of 10% on whether the sample is representative of all reports. For these 384 (66+61+58+64+47+44) contracts, two of the authors spent 5 minutes per contract to confirm the true positives, respectively. In case the verdict by two authors was not unanimous, a third author broke the tie. Via this confirmation process, we got the true positive rate of each tool. Although SPCON is neither sound nor complete, SPCON achieved the best in the result accuracy, namely, 81.8% of the detected permission bugs are true positives. Moreover, SPCON found 11 previously unknown permission bugs in $\text{SB}^{\text{old}}$.

For a detailed comparison, we also evaluated SPCON and other tools on the detection of permission CVEs. There are 531 smart contract CVEs as of now, out of which 19 are permission bugs and most of the rest are integer overflow/underflow bugs. We ignored two permission CVEs, namely CVE-2021-39167 and CVE-2021-39168, since they do not target real-world smart contracts. Table 5 shows the detection results of Slither, Oyente, Maian, SmartCheck, Manticore, Mythril, Securify, Ethainer, and SPCON. SPCON does not apply (‘N/A’) to CVE-2018-19830, CVE-2018-19833, and CVE-2018-19833 since they have only one, four and three transactions, respectively. Ethainer [16] is a newly proposed security analyzer for information flow vulnerabilities caused by access control bugs. Note that although Ethainer is not open-source, it provides a public website [12] recording its analysis results on smart contracts; we used these published results in our comparison.

Table 5 shows that neither Oyente, Manticore, Securify, nor Ethainer could detect any permission CVEs. Slither and SmartCheck detected one permission CVE, which is due to the misuse of tx.origin, while Mythril and Maian found two and four, respectively. SPCON detected nine permission CVEs, which is more than all of the other tools combined and includes six CVEs that existing tools cannot find. This indicates that SPCON has an advantage over pattern-based approaches and can complement these to achieve better results.

**Answer to RQ2**: SPCON exceeded state-of-the-art vulnerability detection capabilities, showing higher accuracy on finding existing access-control bugs. It found 11 unknown permission bugs and six access-control CVEs that no other tool finds.

**Results for RQ3.** To understand the causes of previously unknown permission bugs, we performed a case study on a permission bug that was found only by SPCON. Figure 5 shows the simplified code of the $\text{EDU}$ token contract. It was used to empower an academic platform led by Open Source University [43]. $\text{EDU}$ was launched on November, 2017 and is currently deprecated; it has two sensitive variables, “ownerAddress” and “certifier” (Lines 2 to 3). The user having the address of “ownerAddress” is in charge of the contract; “certifier” is a contract instance which should implement the interface “Certifier”. The “certifier” instance is used to certify incoming participants via the fallback function (Lines 11 to 17). However, unauthorized users can reset “certifier” by calling the $\text{updateCertifier}$ function (Lines 8 to 10).

Existing vulnerability patterns used to detect permission bugs cannot capture this scenario, because they all capture behaviors that do not apply here. Some patterns used by Maian, Manticore, Mythril, Slither and Security look for Ether withdrawal, which does not occur in $\text{EDU}$. Similarly, Manticore, Mythril, and Slither also check if unauthorized users can taint delegatecall, but this instruction is not used in the contract. Another pattern, used by Maian, Manticore, Mythril and Slither, checks if unauthorized users can destroy a smart contract; however, $\text{EDU}$ contains no selfdestruct instruction. Finally, a pattern to check the misuse of tx.origin, used by Manticore, Mythril and SmartCheck, is not applicable in $\text{EDU}$ either, as it never uses tx.origin in its code.

We should expect that Security can identify this issue in $\text{EDU}$ as a vulnerability of an unrestricted write. However, Security failed in this case. A possible reason for this could be that Security needs to analyze the semantics of smart contracts. In function $\text{updateCertifier}$, certifier=Certifier(\_address) (Line 9) is a type conversion statement to covert \_address from an ordinary address type to “certifier” of the “Certifier” interface type. As the implementation of this interface is not available at compile time, Security cannot analyze it. SPCON solves this problem by using a role-based information security lattice as the expected behavior of smart contract. SPCON found that only high-security level roles can write to the “certifier” variable from the partial observation transaction history of $\text{EDU}$. With its conformance testing, SPCON confirmed that unauthorized users can write to “certifier”, which constitutes the permission bug.

SPCON only flags exploitable permission bugs. The bugs are exploitable in the sense that users can replay the attack on the on-chain smart contracts. Most of the existing pattern-based tools aim to detect a bug but without any guarantee if the bug can be
Figure 6: The VoipToken Solidity code.

exploited or not. However, vulnerable code does not imply that the contract is exploitable. Figure 6 shows the simplified code of VoipToken. This permission bug was reported by Maian but not by SPCOn. Notice that the constructor function name of the VoipToken contract has a typo, and it should be “VoipToken()” instead of “VoipTken()” (Lines 11 to 14). The latter can be used to modify the value of “owner” (Line 12). When attacker calls the function VoipTken, function distr (Lines 18 to 20) will be invoked to manage the distribution of tokens to users. The distr function is guarded by the modifier canDistr (Lines 5 to 7), which requires distributionFinished to be false (Line 6). For the on-chain smart contract VoipToken, however, the current value of distributionFinished is true. Therefore, an attacker cannot exploit the buggy constructor function VoipTken. To some extent, SPCOn can identify exploitable permission bugs for the on-chain smart contracts with higher accuracy and can alert the contract administrator before attacks cause a money loss.

Answer to RQ3: SPCOn complements existing pattern-based tools and finds previously unknown permission bugs while achieving a relatively high accuracy.

5.3 Threats to Validity

Internal validity. The ground truth on role structures of our OpenZeppelin smart contracts for role mining may not be fully reliable. To mitigate the issue, two of the authors independently labeled the role structures with the help of the third author to resolve a possible disagreement. Also, we lack a ground truth on permission bugs except for the previously confirmed CVEs. To mitigate this issue, we evaluated SPCOn on the previously well-studied real-world smart contract benchmark SBMod. False positives in the detection result could exist. We again have two or three authors confirm the reported permission bugs manually.

External validity. The type of OpenZeppelin smart contracts we used for role mining in this work may be limited. Our findings may not generalize to other cases. However, the OpenZeppelin access control library has empowered smart contracts across different domains. Therefore, we believe that other types of contracts are similar to the contracts we study in this work.

6 RELATED WORK

Our work is closely related to the security analysis of smart contracts and the access control models as well as role mining.

6.1 Smart Contract Security Analysis

The research landscape on smart contract security analysis can be broadly categorized according to the kinds of vulnerabilities addressed. After the DAO attack [55] in 2016, reentrancy has been recognized among the most serious vulnerabilities. In a reentrant contract, an external user is able to repeatedly call back to the contract within a single transaction. Arithmetic bugs exist in smart contracts, in the same way as in traditional programs [3], but often lead to worse damages. For example, “BEC” was attacked by creating a huge number of tokens by exploiting the integer overflow [15].

Other logical flaws are susceptible to different types of attacks: some smart contracts are suspected to have the “unchecked-send” bug, where the return value of a send function is not checked [28, 46]. This may lead to unwanted behaviors when send fails (e.g., due to insufficient gas) and no appropriate error handling code is present. Because the execution of a smart contract is independent from the blockchain environment, the improper use of environment variables (e.g., the block timestamp) puts smart contracts at the risk of dependence manipulation [67]. The transaction-ordering dependence (i.e., front running) problem exists if there are data races between contract functions. Similarly, attackers may conduct a denial-of-service attack by following some gas-consuming code patterns [18, 27] to make the gas costs of a certain function extremely high [19]. Missing permission checks on user accesses can make a smart contract prodigal and suicidal [44], and it may also enable arbitrary writes to critical variable [16, 56] or arbitrary code execution using the delegatecall [2, 3, 33], etc. Moreover, Groce et al. [29] analyzed many real-world audit reports and their findings show 42% of the access control bugs are of high severity.

The techniques used to address the security issues can be classified into static and dynamic analyses. The former can be further broken down to program analysis, program synthesis, symbolic execution, formal verification, and theorem proving. Slither [25] and Ethainter [16] perform taint analysis to find information flow vulnerabilities. SmartCheck [60] targets code issues by searching contracts’ AST against predefined rules. Securely [56] infers semantic information by analyzing the control- and data-dependencies of contract code, which is then checked against several predefined security patterns. Meanwhile, SmartScoppy [26] introduces a summary-based symbolic evaluation to synthesize attack programs for vulnerable contracts. Oyente [37] is one of the earliest symbolic execution engine for smart contracts, followed by Manticore [2] and Mythril [3]. Mythril [3] is an industrial security analysis tool which combines
symbolic execution and taint analysis to detect nearly 30 classes of vulnerabilities. Other formal verification [34, 36, 68] and theorem-proving [30, 47] tools were built aiming to check properties on the safety [34, 47], security, and fairness [36] of smart contracts. On the other hand, dynamic analyses find exploitable security bugs by directly executing the contracts in a testing environment. Test cases are usually generated through fuzzing or model-based testing [35]. ContractFuzzer [33] is one of the earliest black-box fuzzing frameworks for detecting smart contract vulnerabilities. It predefines several practical test oracles and analyzes the collected execution traces to check against the oracles. The gray-box fuzzing tools ContraMaster [65, 66], Harvey [71], and Echidna [61] employ a feedback-driven mechanism to guide the testing process.

SPCon distinguishes itself from these works in that it focuses on flaws in high-level security policy design and does not rely specific low-level bug patterns. This is only achievable through the reverse engineering of high-quality role structures.

6.2 RBAC in Smart Contracts and Role Mining
RBAC has been developed since 1995 [50]. It mitigates the management efforts of user access control, because permissions are assigned to roles, and the users of a role inherit its permissions. An advantage of RBAC is its policy-neutrality, which enables the implementation of a variety of access control policies. RBAC has been proposed as a solution to separate the execution of access control policies from the management of business logic in smart-contract-based decentralized applications [17]. On the other hand, smart contracts are also used as a tool to implement access control policies for off-chain applications, e.g., Internet of Things [73], data sharing systems [45, 58], and identity management in a trans-organizational setting [20].

Role engineering is a big challenge in applying RBAC: a top-down approach based on a specification of the roles is limited to cases where there exists a design specification of access control systems. However, for many real-world systems, such as smart contracts, these roles are not formally implemented or documented. Therefore, the role structures have to be first reverse engineered from past user access logs using a bottom-up approach, namely, role mining. We have discussed a number of well-known role mining algorithms [23, 41, 54, 72] in Sect. 3, and compared SPCon with them empirically in Sect. 5. Most existing role mining techniques assume a fully-observed user permission assignment, i.e., UPA contains all permissions assigned to each user [40]. This assumption does not work well in the smart contract setting.

SPCon distinguishes itself from the existing role mining approaches in that SPCon solves the partial-observation role mining problem to mine high-quality role structures.

7 CONCLUSION
In this paper, we presented a testing tool, SPCon, specialized for smart contract permission bugs. SPCon relies on historical transactions and a novel partial-observation role mining technique to solve the test oracle problem. The evaluation results on multiple datasets indicate that SPCon is able to mine high-quality role structures and discover exploitable permission bugs more accurately than existing tools. In particular, SPCon detects 11 previously unknown permission bugs from the well-studied vulnerability dataset SBwild. Our approach can also be used to improve the understanding of role-based security policies of deployed contracts, allowing potential permission attacks to be detected before causing real losses.

ACKNOWLEDGMENTS
This research is supported by the Singapore National Research Foundation under the National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001).

REFERENCES


