Finding Permission Bugs in Smart Contracts with Role Mining

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

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

Shang-Wei Lin
shang-wei.lin@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 SPCon. The experimental evaluation on labeled smart contract role mining benchmark demonstrates that SPCon 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 SPCon 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.

ACM Reference Format:
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 (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534372

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07...$15.00
https://doi.org/10.1145/3533767.3534372

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, SPCon, 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

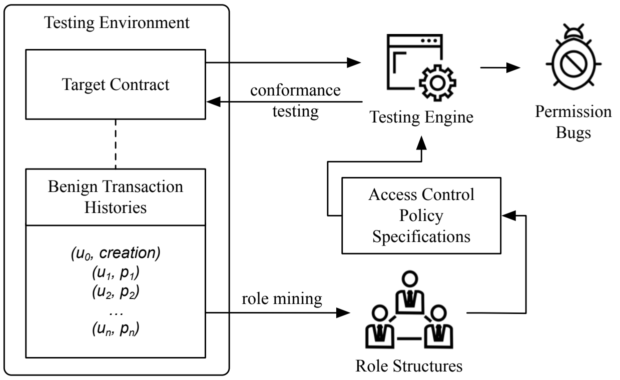

Figure 1: Role mining-based permission bug identification.

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, SB^{wild} [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 SB^{wild}. The results show that SPCon achieves the highest accuracy of permission bug detection and finds 11 previously unknown permission bugs in SB^{wild}. 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 "`notIntialized`" (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 || operator == msg.sender); _;}
  modifier onlyReserveFund() { require(reserveFund == msg.sender || operator == msg.sender); _;}
  modifier notInitialized() { require(!initialized); _;}
  /* ======= Initialize function ======== */
  function initialize(
      address _stakeToken,
      address _liquidityToken,
      address _reserveFund) public notInitialized {
    stakeToken = _stakeToken;
    liquidityToken = _liquidityToken;
    reserveFund = _reserveFund;
    operator = msg.sender;
    setRewardPool(liquityToken);
+   initialized = true; // bug-fix
  }
  /* ==== Other functions (omitted) ====
   onlyOperator: setOperator
   onlyExchangeProxy: setExchangeProxy, depositFor
   onlyReserveFund: setReserveFund, allocateMoreRewards
   Permissionless: deposit, withdraw, claimReward */
}
```

Figure 2: The simplified code of *ProfitSharingRewardPool*.

user could successfully re-initialize the contract. 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. More specifically, pattern-based approaches may fail because, (1) the "initialize" function is already protected by a modifier, therefore, a static analysis tool searching for the "unrestricted write" pattern [3, 56] would not raise an alarm, and (2) one has to invalidate the protecting modifier by realizing that the condition "initialized == false" still holds after the initialize function is called. Yet, this kind of reasoning is beyond the scope of existing bug patterns.

In general, bug patterns targeting common low-level coding errors may miss design flaws and fail to reveal many permission bugs. To address these challenges, we propose to first derive access control policy specifications through role mining, which can then be used to perform conformance testing to detect design-level permission bugs.

3 BACKGROUND

In this section, we review role-based access control (RBAC) [52], the classic role mining problem, and other relevant preliminaries required for the rest of the paper.

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 }

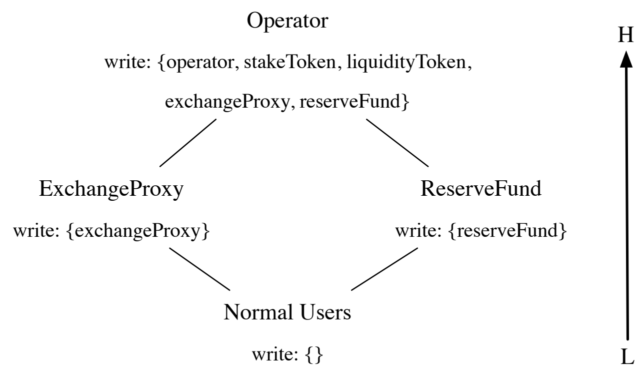

Figure 3: The *ProfitSharingRewardPool*'s security lattice.

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, $PA \subseteq P \times R$ is the permission-to-role assignment relation, and $UA \subseteq U \times R$ is the user-to-role assignment relation.*

A role is properly 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, permissions can be enforced at both the function- and statement-level.

Table 1 shows the access control model of *ProfitSharingRewardPool*. There are four roles, namely, "Operator", "ExchangeProxy", "ReservFund", and "Normal Users". Each role is granted a set of permissions: 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 information 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 $\mathcal{SC}$ with a partially ordered dominance relation $\succeq$ 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., $r_i \succeq r_j$ if and only if $write(r_i) \supseteq write(r_j)$, where $write(r_i)$ 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 "intialize" 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 ith user has the jth 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 δ-Consistency Role Mining Problem (δ-RMP) was first defined by Vaidya et al. [63].

Definition 3 (δ-RMP [63]). *Given a set of users U, a set of permissions P, and a user-permission assignment matrix UPA, the problem of δ-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 δ-RMP allows a limited number of mismatches below the given threshold δ. 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. HierarchicalMiner (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 δ, 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 not equal, without loss of generality, assume that R_1 is the smallest set, then we have,*

$$M = \left\{\arg\max_{r_j \in R_2} \text{jaccard}(r_i, r_j) \,\middle|\, \forall r_i \in R_1\right\}, \tag{1}$$

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,

$$\overline{M} = \left\{r_j \,\middle|\, \exists r_j \in (R_2 \setminus M) \cdot \max_{r_i \in R_1} \text{jaccard}(r_i, r_j) > t\right\}, \tag{2}$$

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

$$\begin{aligned} Sim(R_1, R_2) = \Bigg(&\sum_{r_i \in R_1} \max_{r_j \in M} \text{jaccard}(r_i, r_j) \\ &+ \sum_{r_j \in \overline{M}} \max_{r_i \in R_1} \text{jaccard}(r_i, r_j)\Bigg) \Bigg/ \left(||R_1|| + ||\overline{M}||\right). \end{aligned} \tag{3}$$

When $t = 1$, 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 $t = 0$, 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 $\boldsymbol{x} \in \mathbb{R}^{|P|}$, where $x_i = n(r, p_i)/|r|$.*

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:*

$$SimErr = \max_{r_i, r_j \in R} \cos\left(AFV(r_i), AFV(r_j)\right). \tag{4}$$

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 δ value of δ-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:*

$$DeltaErr = \frac{||UA \otimes PA - UPA||_1}{||UA \otimes PA||_1}, \tag{5}$$

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

PORM as an Optimization Problem. The RMP and δ-RMP are both NP-Complete problems [62, 63]. The complexity of the PORM is at least as hard as that of the δ-RMP, and we would like to ensure the generalizability of roles with *SimErr*, while maintaining good consistency with *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,

$$\begin{aligned} \min \quad & \alpha \cdot SimErr + \beta \cdot DeltaErr, \\ \text{s.t.} \quad & UA \otimes PA \supseteq UPA, \end{aligned} \tag{6}$$

where α and β 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 δ, 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* $Chr[\boldsymbol{x}]$ in the following form: $Chr[\boldsymbol{x}] = \langle x_0, \ldots, x_i, \ldots, x_{k-1} \rangle$, 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.

$$\text{fitness}(\boldsymbol{x}) = (\alpha \cdot SimErr + \beta \cdot DeltaErr)^{-1}, \tag{7}$$

where *SimErr* and *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

Algorithm 1 Access control policy conformance testing

Require: (F, V, S_0) ▷ Function set, state variables, initial concrete state
Require: (UA, PA) ▷ Role structures
Require: k ▷ Maximum length of test sequences
1: **procedure** ConformanceTest(F, V, S_0, PA, UA, k)
2: **for all** $\langle f_0, \ldots, f_{k-1} \rangle \in F^k$ **do** ▷ Function sequences
3: $S, \Pi \leftarrow S_0$, true ▷ Initialize state S and path Π
4: **for** $i \in [0, k-1]$ **do**
5: $\Pi \leftarrow \Pi \wedge \bigvee_{\pi \in \textsc{SymbExe}(f_i)} \pi[v \mapsto S(v) \mid v \in V]$
6: **if** Π is not satisfiable **then break**
7: **if** PolicyCheck(Π, V, PA) **then**
8: **return** $\langle f_0, \cdots, f_i \rangle$ ▷ Exploit (error trace)
9: $S \leftarrow S[v \mapsto f_i(v) \mid v \in V]$ ▷ Update S w.r.t. f_i
10: **return** $\langle\rangle$ ▷ Bug not found
Require: λ ▷ An information flow policy defined on lattice
11: **procedure** PolicyCheck(Π, V, PA)
12: $R \leftarrow \emptyset$ ▷ Initialize role set
13: **for all** $(r, P_r) \in PA$ **do** ▷ Role-permission pairs
14: $R \leftarrow R \cup \{r\}$ ▷ Update R
15: $write(r) \leftarrow \bigcup_{p \in P_r} (p.write \cap V)$ ▷ Update write set
16: $L \leftarrow$ BuildLattice(R, $write$) ▷ Build lattice L
17: **if** $\Pi \not\models \lambda(L)$ **then** ▷ λ is violated on Π
18: **return** true ▷ Bug found
19: **else**
20: **return** false ▷ Bug not found

of our GA is randomly generated. Meanwhile, we also keep one best individual from the current generation to carry over to the next as the elitist selection [24] to guarantee that the solution quality obtained by the GA will not decrease. The detailed illustration of the GA role mining process can be found in the supplemental materials: https://sites.google.com/view/spcon/.

4.4 Policy Validation via Conformance Testing

With the mined role structures, we are able to derive a set of access control policy specifications and perform conformance testing on the contract implementation. Since the mined role structures are based on benign user behaviors, any discrepancy between the policy specification and the actual allowed behaviors indicates a potential permission bug. The combination of role mining and testing solves the problem of missing specifications and provides a rich set of test oracles specialized for permission violation bugs.

Algorithm 1 details the conformance testing process. The inputs to Alg. 1 include the contract functions (F), state variables (V), a snapshot of the initial concrete contract state (S_0), the mined role structures (UA, PA), and a user-defined bound for the test sequence length (k). The output is either a test sequence that exploits a permission bug, where access control policies are bypassed, or an empty sequence indicating no bug is found within the given bound.

The conformance testing procedure iterates through each test sequence, consisting of k functions from F (Line 2). In each iteration, we initialize the contract state S using S_0 and set the initial symbolic path condition Π to *true* (Line 3). The algorithm progressively constructs symbolic paths with increasing lengths, and checks against the information flow policy defined over the mined role structure (Lines 4 to 9). On Line 5, we construct a global path Π by concatenating function-level symbolic paths generated from symbolic execution (i. e., SymbExe in Alg. 1). We also substitute the state variables ($v \in V$) with their updated values captured in S, which is either concrete values inherited from S_0 or symbolic expressions as a result of previous function (symbolic) executions. If Π is not satisfiable, we skip the current test and start a new test sequence. Otherwise, we check Π against the given information security policy λ defined on the security lattice (Line 7).

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, ts_1 and ts_2, 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 ts_3 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 ts_4, the buggy "initialize" function is executed first, where the value of "stakeToken" modified in S, thus breaking the integrity policy. Therefore, SPCon reports the shortest attack test sequence—"⟨*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 (S_0 in Alg. 1) faithfully. Step 2, the function-level symbolic execution, is implemented based on Manticore [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

Test Sequences	**Path check**	**Changes in S**	**Policy check**
$ts_1 = setExchangeProxy(\_) \rightarrow depositFor(\_)$	*false*	–	–
$ts_2 = setReservedFund(\_) \rightarrow allocateMoreRewards(\_)$	*false*	–	–
$ts_3 = deposit(X) \rightarrow withdraw(Y)$	*true*	$tokenBalance[attacker] : 0 \rightarrow X \rightarrow (X - Y)$	*false*
$ts_4 = initialize(Z, \_, \_) \rightarrow setExchangeProxy(\_)$	*true*	$stakeToken : \_ \rightarrow Z \rightarrow Z$	*true*

Figure 4: Illustration of the discovery of the *ProfitSharingRewardPool* attack by SPCon.

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^{wild} [22], a public data set consisting of 47,518 contracts from the Ethereum blockchain. Specifically, SB^{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.

Table 2: Benchmark used for the research questions.

Research Questions	Benchmarks
RQ1	Labeled smart contracts for role mining
RQ2	(1) SB^{wild}; (2) access control CVE.

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 α and β, 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 HPr, 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: *Num_Ratio* and *Sim(mined_roles, groundtruth_roles)*. *Num_Ratio* is the ratio of the number of the mined roles to the number of roles in the ground truth. *Sim(mined_roles, groundtruth_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 *Num_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.

Table 3: Evaluation result on the 50 sampled smart contracts of the role mining benchmark.

Approach	Run time (s)	Number of roles	Number of mined roles per roles in ground truth	Similarity				Similarity (privileged roles)			
				t=1	t=0.5	t=0.25	t=0	t=1	t=0.5	t=0.25	t=0
HPr	0.21	8.28	2.69	0.545	0.544	0.515	0.319	0.807	0.806	0.771	0.732
ORCA	5.08	21.96	7.17	0.713	0.700	0.525	0.320	0.885	0.838	0.615	0.535
HM	49.54	19.04	6.37	0.572	0.566	0.448	0.323	0.824	0.827	0.763	0.714
GO	191.72	15.34	4.86	0.570	0.565	0.464	0.350	0.805	0.805	0.755	0.719
SPCon (0.4, 0.6)	31.69	7.00	2.27	0.556	0.556	0.541	0.356	0.832	0.832	0.808	0.753
SPCon (0.5, 0.5)	33.10	4.64	1.54	0.541	0.541	0.519	0.408	0.814	0.814	0.784	0.718
SPCon (0.6, 0.4)	34.55	3.52	1.11	0.580	0.580	0.561	0.426	0.778	0.778	0.742	0.665

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 α, 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 (α, β) 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^{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^{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^{wild}, while Slither reported the most, namely 2,356 contracts. However, Thomas et

Table 4: Permission bug detection results on SB^{wild}

Tool	Number of vulnerabilities	Agree (≥ 1): Num (%)	True-positive rate
Slither	2,356	568 (24 %)	24.2 %
Securify	614	93 (15 %)	32.8 %
SmartCheck	384	193 (50 %)	29.3 %
Mythril	1,076	460 (43 %)	39.0 %
Maian	44	29 (66 %)	61.4 %
Manticore	47	19 (40 %)	19.1 %
SPCon	44	33 (75 %)	81.8 %

Table 5: Evaluation results on the 17 permission CVEs.

CVE	Slither	Oyente	Maian	SmartCheck	Manticore	Mythril	Securify	Ethainter	SPCon
CVE-2018-10666	✗	✗	✗	✗	✗	✗	✗	✗	✓
CVE-2018-10705	✗	✗	✗	✗	✗	✗	✗	✗	✓
CVE-2018-11329	✗	✗	✗	✗	✗	✗	✗	✗	✓
CVE-2018-17111	✗	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2018-19830	✗	✗	✗	✗	✗	✗	✗	✗	N/A
CVE-2018-19831	✗	✗	✓	✗	✗	✗	✗	✗	✓
CVE-2018-19832	✗	✗	✓	✗	✗	✗	✗	✗	✓
CVE-2018-19833	✗	✗	✗	✗	✗	✗	✗	✗	N/A
CVE-2018-19834	✗	✗	✗	✗	✗	✗	✗	✗	N/A
CVE-2019-15078	✗	✗	✓	✗	✗	✓	✗	✗	✓
CVE-2019-15079	✗	✗	✗	✗	✗	✗	✗	✗	✓
CVE-2019-15080	✗	✗	✗	✗	✗	✗	✗	✗	✓
CVE-2020-17753	✓	✗	✓	✓	✗	✓	✗	✗	✗
CVE-2020-35962	✗	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2021-3006	✗	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2021-34272	✗	✗	✗	✗	✗	✗	✗	✗	✓
CVE-2021-34273	✗	✗	✗	✗	✗	✗	✗	✗	✗

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+44+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 SB^{wild}.

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, Ethainter, 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. Ethainter [16] is a newly proposed security analyzer for information flow vulnerabilities caused by access control bugs. Note that although Ethainter 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 *EDU* token contract.[1] It was used to empower an academic platform led by Open Source University [43]. *EDU* was launched

```
1 contract EDUToken is {
2   Certifier public certifier;
3   address public ownerAddress;
4   function EDUToken() public {
5     certifier = Certifier(0x1e2F058C...);
6     ownerAddress = msg.sender;
7   }
8   function updateCertifier(address _address) public{
9     certifier = Certifier(_address);
10  }
11  function() payable{
12     // Only certified addresses are allowed to join
13    if (!certifier.certified(msg.sender)) {
14      revert();
15    }
16    ...
17  }
18 }
```

Figure 5: The *EDUToken* Solidity code.

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 *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 Securify look for Ether withdrawal, which does not occur in *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, `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 `EDU` either, as it never uses `tx.origin` in its code.

We should expect that Securify can identify this issue in *EDU* as a vulnerability of an unrestricted write. However, Securify failed in this case. A possible reason for this could be that Securify needs to analyze the semantics of smart contracts. In function *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, Securify 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 *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

[1]Ethereum address 0x849c2ea2a8f0ed0fe6d28b17fa0f779d6a45dff1

```
1 contract VoipToken is ERC20 {
2   address owner = msg.sender;
3   bool public distributionFinished = false;
4   mapping (address => uint256) balances;
5   modifier canDistr() {
6     require(!distributionFinished);_;
7   }
8   modifier onlyOwner() {
9     require(msg.sender == owner);_;
10  }
11  function VoipTken () public {
12    owner = msg.sender;
13    distr(owner, totalDistributed);
14  }
15  function finishDistribution() onlyOwner canDistr
        public {
16    distributionFinished = true;
17  }
18  function distr(address _to, uint256 _amount) canDistr
         private {
19    balances[_to] = balances[_to].add(_amount);
20  }
21 }
```

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*.[2] 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.[3] 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 SB^{wild}. 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.

[2] Ethereum address 0x3da034753b42bda1bcfa682f29685e2fd6729016

[3] See its storage slot: https://api.etherscan.io/api?module=proxy&action=eth_getStorageAt&address=0x3da034753b42bda1bcfa682f29685e2fd6729016&position=0x7

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 not 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. Securify [56] infers semantic information by analyzing the control- and data-dependencies of contract code, which is then checked against several predefined security patterns. Meanwhile, SmartScopy [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 SB^{wild}. 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

[1] 2017. Maian. https://github.com/ivicanikolicsg/MAIAN. A Tool for Automatic Detection of Buggy Ethereum Smart Contracts of Three Different Types: Prodigal, Suicidal and Greedy.
[2] 2019. Manticore. https://github.com/trailofbits/manticore. Symbolic Execution Tool for Smart Contracts.
[3] 2019. Mythril. https://github.com/ConsenSys/mythril. A Security Analysis Tool for EVM Bytecode.
[4] 2019. Oyente. https://github.com/melonproject/oyente. An Analysis Tool for Smart Contracts.
[5] 2020. Etherscan. https://etherscan.io.
[6] 2020. OpenZeppelin. https://openzeppelin.com/.
[7] 2021. *Ethereum Attacks*. https://gist.github.com/ethanbennett/7396bf3f61dd985d3426f2ee184d8822
[8] 2021. Slither. https://github.com/crytic/slither. The Solidity Source Analyzer.
[9] 2021. Smart Contract Search. https://etherscan.io/searchcontract. Search Smart Contract source codes on Etherscan with keywords, addresses, txhash date, block number, and more.
[10] 2021. State of The DApps. https://www.stateofthedapps.com/zh/platforms/ethereum.
[11] 2021. *VALUE DEFI - REKT 2*. https://rekt.news/value-rekt2/
[12] 2022. Ethainter. https://contract-library.com/. Contract Library Watchdog Service.
[13] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on Ethereum smart contracts (SoK). In *International conference on principles of security and trust*. Springer, 164–186.
[14] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.
[15] Blockchain-Projects. 2020. Overflow Attack in Ethereum Smart Contracts. https://blockchain-projects.readthedocs.io/overflow.html.
[16] 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*. 454–469.
[17] Arnab Chatterjee, Yash Pitroda, and Manojkumar Parmar. 2020. Dynamic Role-Based Access Control for Decentralized Applications. In *International Conference on Blockchain*. Springer, 185–197.
[18] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-Optimized Smart Contracts Devour Your Money. In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*. 442–446. https://doi.org/10.1109/SANER.2017.7884650
[19] Consensys. 2021. *Gas Limit DoS on The Network via Block Stuffing*. https://consensys.github.io/smart-contract-best-practices/known_attacks/#dos-with-block-gas-limit
[20] Jason Paul Cruz, Yuichi Kaji, and Naoto Yanai. 2018. RBAC-SC: Role-Based Access Control Using Smart Contract. *Ieee Access* 6 (2018), 12240–12251.
[21] 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*. 530–541.
[22] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. *SmartBugs on Real World 47,587 Smart Contracts*. https://github.com/smartbugs/smartbugs-results
[23] Alina Ene, William Horne, Nikola Milosavljevic, Prasad Rao, Robert Schreiber, and Robert E Tarjan. 2008. Fast Exact and Heuristic Methods for Role Minimization Problems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*. 1–10.
[24] Larry J Eshelman. 1991. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In *Foundations of genetic algorithms*. Vol. 1. Elsevier, 265–283.

[25] 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.

[26] Yu Feng, Emina Torlak, and Rastislav Bodik. 2019. Precise Attack Synthesis for Smart Contracts. *arXiv preprint arXiv:1902.06067* (2019).

[27] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* (2018), 116.

[28] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[29] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. 2020. What are the actual flaws in important smart contracts (and how can we find them)?. In *International Conference on Financial Cryptography and Data Security*. Springer, 634–653.

[30] Yoichi Hirai. 2017. Defining The Ethereum Virtual Machine for Interactive Theorem Provers. In *International Conference on Financial Cryptography and Data Security*. Springer, 520–535.

[31] EOS IO. 2017. EOS. IO Technical White Paper. *EOS. IO (accessed 18 December 2017) https://github. com/EOSIO/Documentation* (2017).

[32] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[33] Bo Jiang, Ye Liu, and WK Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 259–269.

[34] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Ndss*. 1–12.

[35] Ye Liu, Yi Li, Shang-Wei Lin, and Qiang Yan. 2020. ModCon: A Model-Based Testing Platform for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1601–1605.

[36] Ye Liu, Yi Li, Shang-Wei Lin, and Rong Zhao. 2020. Towards Automated Verification of Smart Contract Fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 666–677.

[37] 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*. ACM, 254–269.

[38] Brad L Miller, David E Goldberg, et al. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems* 9, 3 (1995), 193–212.

[39] Melanie Mitchell. 1998. *An introduction to genetic algorithms*. MIT press.

[40] Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. 2016. A survey of role mining. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–37.

[41] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin Calo, and Jorge Lobo. 2008. Mining Roles with Semantic Meanings. In *Proceedings of the 13th ACM symposium on Access control models and technologies*. 21–30.

[42] Ian Molloy, Ninghui Li, Tiancheng Li, Ziqing Mao, Qihua Wang, and Jorge Lobo. 2009. Evaluating Role Mining Algorithms. In *Proceedings of the 14th ACM symposium on Access control models and technologies*. 95–104.

[43] Peshev Nikolay. 2017. *The First Block Chain Based Academical Platform is on the Way*. https://medium.com/@nikolaypeshev/the-first-block-chain-based-academical-platform-is-on-the-way-fe12d0a4e73d

[44] 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*. ACM, 653–663.

[45] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. 2017. Towards a Novel Privacy-Preserving Access Control Model Based on Blockchain Technology in IoT. In *Europe and MENA cooperation advances in information and communication technologies*. Springer, 523–533.

[46] Daniel Perez and Benjamin Livshits. 2019. Smart Contract Vulnerabilities: Does Anyone Care? (feb 2019). arXiv:1902.06710 http://arxiv.org/abs/1902.06710

[47] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1661–1677.

[48] Pierangela Samarati and Sabrina Capitani de Vimercati. 2000. Access Control: Policies, Models, and Mechanisms. In *International School on Foundations of Security Analysis and Design*. Springer, 137–196.

[49] Ravi Sandhu. 1996. Role hierarchies and constraints for lattice-based access controls. In *European Symposium on Research in Computer Security*. Springer, 65–79.

[50] Ravi Sandhu, David Ferraiolo, Richard Kuhn, et al. 2000. The NIST Model for Role-Based Access control: Towards a Unified Standard. In *ACM workshop on Role-based access control*, Vol. 10.

[51] Ravi S. Sandhu. 1993. Lattice-based access control models. *Computer* 26, 11 (1993), 9–19.

[52] Ravi S Sandhu. 1998. Role-Based Access Control. In *Advances in computers*. Vol. 46. Elsevier, 237–286.

[53] Palladino Santiago. 2017. *The Parity Wallet Hack Explained*. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

[54] Jürgen Schlegelmilch and Ulrike Steffens. 2005. Role Mining with ORCA. In *Proceedings of the tenth ACM symposium on Access control models and technologies*. 168–176.

[55] David Siegel. 2016. *Understanding The DAO Attack*. https://www.coindesk.com/understanding-dao-hack-journalists

[56] Sofware Reliability Lab 2019. *Securify*. Sofware Reliability Lab. https://securify.ch/

[57] Solidity 2018. Solidity. https://solidity.readthedocs.io/en/v0.5.1/.

[58] Tanzeela Sultana, Ahmad Almogren, Mariam Akbar, Mansour Zuair, Ibrar Ullah, and Nadeem Javaid. 2020. Data Sharing System Integrating Access Control Mechanism Using Blockchain-Based Smart Contracts for IoT Devices. *Applied Sciences* 10, 2 (2020), 488.

[59] Hassan Takabi and James BD Joshi. 2010. StateMiner: an efficient similarity-based approach for optimal mining of role hierarchy. In *Proceedings of the 15th ACM symposium on Access control models and technologies*. 55–64.

[60] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

[61] Trail of Bits 2019. *Echidna*. Trail of Bits. https://github.com/trailofbits/echidna

[62] Emre Uzun, Vijayalakshmi Atluri, Haibing Lu, and Jaideep Vaidya. 2011. An optimization model for the extended role mining problem. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 76–89.

[63] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. 2007. The Role Mining Problem: Finding a Minimal Descriptive Set of Roles. In *Proceedings of the 12th ACM symposium on Access control models and technologies*. 175–184.

[64] Jaideep Vaidya, Vijayalakshmi Atluri, Qi Guo, and Nabil Adam. 2008. Migrating to optimal RBAC with minimal perturbation. In *Proceedings of the 13th ACM symposium on Access control models and technologies*. 11–20.

[65] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. Vultron: catching vulnerable smart contracts once and for all. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. IEEE Press, 1–4.

[66] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2020. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. *IEEE Transactions on Dependable and Secure Computing* (2020).

[67] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 189 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360615

[68] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2018. Formal Specification and Verification of Smart Contracts for Azure Blockchain. *arXiv preprint arXiv:1812.08829* (2018).

[69] Maximilian Wohrer and Uwe Zdun. 2018. Smart Contracts: Security Patterns in The Ethereum Ecosystem and Solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2–8.

[70] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum project yellow paper* 151 (2014), 1–32.

[71] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1398–1409.

[72] Dana Zhang, Kotagiri Ramamohanarao, and Tim Ebringer. 2007. Role Engineering Using Graph Optimisation. In *Proceedings of the 12th ACM symposium on Access control models and technologies*. 139–144.

[73] Yuanyu Zhang, Shoji Kasahara, Yulong Shen, Xiaohong Jiang, and Jianxiong Wan. 2018. Smart Contract-Based Access Control for The Internet of Things. *IEEE Internet of Things Journal* 6, 2 (2018), 1594–1605.