Heuristics for Constrained Role Mining in the Post-Processing Framework

Carlo Blundo
DISA-MIS
Università di Salerno, Italy

Stelvio Cimato
Dipartimento di Informatica
Università degli studi di Milano, Italy

Luisa Siniscalchi
Concordium Blockchain Research Center
Aarhus University, Denmark
lsiniscalchi@cs.au.dk

May 10, 2021

Abstract

Role mining techniques are frequently used to derive a set of roles representing the current organization of a company following the RBAC model and simplifying the definition and the implementation of security policies. Constraints on the resulting roles can be defined to have valid roles, that can be efficiently managed, limiting for example the number of permissions included in a role or the users a role can be assigned to. Since the associated problems are NP hard, several heuristics have been developed to find sub-optimal solutions adopting the *concurrent* or the *post-processing approach*. In the first case, assignment matrices are obtained satisfying the given constraints during the computation, while in the second case, the intermediate solutions are obtained without considering the constraints, that are enforced successively.

In this paper we present two heuristics for the *Permission Usage* and *Role Usage* Cardinality Constraints in the post-processing approach: we consider constraints limiting the number of permissions that can be included in a role in the first case, and the number of roles that can include a permission in the second case, refining the roles produced by some other technique (not considering any constraint). For both heuristics we analyze their performance after their application to some standard datasets, showing the improved results obtained w.r.t. state of the art solutions.

1 Introduction

Role mining techniques aim to define a valid set of roles starting from the existing user-permissions assignment within a given organization and are usually executed as the first step for the implementation of Role Based Access Control (RBAC) framework. RBAC model is one of the most popular approaches to organize access to restricted resources, easing administration tasks and reducing costs in case of dynamic changes in the organizational assets of a company. Introduced at the end of the '90 [26], RBAC has been standardized by Ferrajolo et al. [9], where a reference model has been defined, describing also the functional requirements for the management of roles and relations and the support to the access control decision process.

Since roles are the key factor of the RBAC model, recently several frameworks have been enhanced with the possibility to define some constraints on the way roles are shaped or utilized. The idea is that to get effectively usable roles, they need to have some characteristics that the security manager can select in order to drive the role mining process. For example constraints can be imposed on the number of permissions a role can include, to avoid in the extreme cases trivial roles, containing just one or few permissions, or roles with many permissions, that could make difficult the management of the security policy. In literature, different approaches have considered constrained role mining, often using different terminologies and constraints, see [21] for a complete survey, while several works [12, 14, 4, 5] discuss their application during the role mining process.

There are basically two ways in which constraints can be included in the role mining process that are usually referred as the *concurrent* or the *post-processing* framework. In the first case, constraints are considered during each step of the computation, and they basically drive the way new roles and assignments among users and permissions are devised. In the second approach, one determines a set of roles regardless of any constraint. Successively, these mined roles are processed to meet the given constraints.

In this work we focus on two different kinds of cardinality constraints, namely *role-usage cardinality constraints* (RUCC) and *permission-usage cardinality constraints* (PUCC). For the RUCC case, constraints limit the number of roles that can be assigned to a user, while for the PUCC case, constraints restrict the number of permissions that can be included in a role. In both cases, starting from an initial set of roles and assignments, we operate in the post-processing framework where constraint satisfaction is imposed by correcting their violations.

We present two heuristics for the constrained role mining in the RUCC and PUCC scenario and evaluate them using real-world datasets [8] and considering standard metrics [16]. The results are then compared with the ones obtained applying state of the art heuristics available in literature. In particular, for the RUCC case, we consider the heuristics *Role-Priority-based Approach* and *Coverage-of-Permissions-based Approach* described in [14] and the heuristic *Fix Role Usage Constraint* proposed in [12]. For the PUCC case, we propose the first post-processing heuristic named `postPUCC`. In all cases, considering both the size of the role set and the execution time, our heuristics improve over the previous proposals.

The paper is organized as follows: In the next section, related works on constraint role mining are discussed. Section 3 introduces the role mining framework, reporting the basic definitions and the associated problems. In Section 4 we describe the heuristics we propose for the PUCC and the RUCC cases, showing some simple application examples, while in Section 5 we report the experimental evaluation including the results obtained after the execution of the heuristics to standard real-world datasets. Finally, in section 6, we draw some conclusions.

2 Related Work

Several different variants and extensions have been proposed in the literature for the basic Role Mining model. Heuristics were also defined by resorting to mapping Role Mining to known problems, as in the case of graph-based strategies [33], matrix decomposition [19], or formal concept analysis [23]. We refer to [21] for a complete survey of these different approaches.

As regards constraints, they were firstly introduced in role mining in [26], where the $RBAC_2$ model considers different types of constraints, including mutually exclusive roles, used to enforce separation of duty policies, and cardinality constraints, limiting some parameters of the resulting set of roles. In particular, four kinds of cardinality constraints can be defined considering: the maximum number of roles that can be assigned to a user; the maximum number of users that can be assigned to a role; the maximum number of permissions that can be included in a role; the maximum number of roles that can include a given permission.

The first class of constraints is usually referred to as *role-usage* constraint, and it has been considered in [14, 17, 18]. Lu et al [17, 18], consider two versions of the problem, one giving an exact solution, and the other including a given number of errors, and provide two heuristics in the concurrent framework, for both the correct and the approximate version. In [14], the authors propose two algorithms operating in the post-processing framework, named Role-Priority-based Approach (`RPA`) and Coverage-of-Permissions-based Approach (`CPA`). In [12], the heuristic *Fix Role Usage Constraint* (`FixRUC`), corresponding to Algorithm 1 in Section 3.1 of [12], was described for the post-processing framework. Such an heuristic unassigns some roles to the users violating the RUCC constraint and substitutes them with another role in such a way that, at the end of the procedure, all users will possess at most the maximum number of permitted roles. The second class of constraints is named *role-distribution* constraint, and it has been introduced in [13], where three heuristics are presented, based on the minimum biclique covering approach, firstly proposed in [8]. The third and fourth class of constraints are one the dual of the other and are usually referred to as *permission-usage* constraint and *permission-distribution* constraint, respectively. In [4], the authors propose a framework that can be easily adapted to each class of constraints, specializing a general approach to role mining.

Works [15] and [2] propose some solutions considering the restrictions on the number of permissions a role can include. In [2], two heuristics have been proposed, t-SMAR and t-SMAC. The symbol t refers to the constraint value, that is to the maximum number of permissions each role can contain. Both heuristics form a role selecting permissions from the users-to-permissions assignment matrix describing the permissions assigned to users (a permission grants system access to authorized users). The heuristics in [2] differ in how they select the permissions from the users-to-permissions assignment matrix: the first one chooses a *minimum-weight* row (i.e., a row containing the minimum number of permissions), while the second one chooses a minimum-weightcolumn. Kumar et al. [15] propose a technique named Constrained Role Miner (CRM), where first roles are created by grouping similar permission assignments of one or more users, and then roles satisfying the cardinality constraint are mined.

Some approaches considering multiple constraints holding on the final role-set, have also been discussed. Indeed. in [20] a role mining technique has been designed to provide a set of roles where both role-distribution and role-cardinality constraints are satisfied. The combination of role-usage and permission-usage constraints has been analyzed also in [3, 5].

3 Role Mining

In this section we recall the basic definitions for the RBAC model, the computational complexity of the related problems, and the two alternative frameworks for role mining.

The notation we use is based on the NIST standard for *Core Role-Based Access Control* (Core RBAC, or RBAC 0), see [27] and [9]. We denote with $\mathcal{U} = \{u_1, \ldots, u_n\}$ the set of users, $\mathcal{P} = \{p_1, \ldots, p_m\}$ the set of permissions, and $\mathcal{R} = \{r_1, \ldots, r_k\}$ the set of roles. The following assignment relations are defined:

- $\mathcal{UA} \subseteq \mathcal{U} \times \mathcal{R}$ is a many-to-many mapping *user-to-role* assignment relation.
- $\mathcal{PA} \subseteq \mathcal{R} \times \mathcal{P}$ is a many-to-many mapping *role-to-permission* assignment relation.
- $\mathcal{UPA} \subseteq \mathcal{U} \times \mathcal{P}$ is a many-to-many mapping *user-to-permission* assignment relation.

Obviously, we can represent the assignment relations by binary matrices. For instance, by `UA` we denote the $\mathcal{UA}$'s matrix representation. The binary matrix `UA` satisfies $\mathtt{UA}[i][j] = 1$ if and only if $(u_i, r_j) \in \mathcal{UA}$. This means that user u_i is assigned role r_j. In a similar way, we define the matrices `PA`, and `UPA`. Moreover, we define the following functions:

- $\mathsf{AssignedRoles_U} : \mathcal{U} \to 2^{\mathcal{R}}$. This function returns the set of roles assigned to a given user and any $u \in \mathcal{U}$, is defined as $\mathsf{AssignedRoles_U}(u) = \{r : (u, r) \in \mathcal{UA}\}$.
- $\mathsf{AssignedRoles_P} : \mathcal{P} \to 2^{\mathcal{R}}$. This function returns the set of roles assigned to a given permission and, for any $p \in \mathcal{P}$, is defined as $\mathsf{AssignedRoles_P}(p) = \{r : (r, p) \in \mathcal{PA}\}$.
- $\mathsf{AssignedUsers} : \mathcal{R} \to 2^{\mathcal{U}}$. This function returns the set of users assigned to a given role and, for any $r \in \mathcal{R}$, is defined as $\mathsf{AssignedUsers}(r) = \{u : (u, r) \in \mathcal{UA}\}$.
- $\mathsf{AssignedPrms_R} : \mathcal{R} \to 2^{\mathcal{P}}$. This function returns the set of permissions assigned to a given role and, for any $r \in \mathcal{R}$, is defined as $\mathsf{AssignedPrms_R}(r) = \{p : (r, p) \in \mathcal{PA}\}$.
- $\mathsf{AssignedPrms_U} : \mathcal{U} \to 2^{\mathcal{P}}$. This function returns the set of permissions assigned to a given user and, for any $u \in \mathcal{U}$, is defined as $\mathsf{AssignedPrms_U}(u) = \{p : (u, p) \in \mathcal{UPA}\}$.

By denoting with $[\ell]$ the set of positive integers up to ℓ included (i.e., $[\ell] = \{1, 2, \ldots, \ell\}$), we can define the above functions also as

- $\mathsf{AssignedRoles_U}(u_i) = \{r_j : j \in [k] \text{ and } \mathtt{UA}[i][j] = 1\}$.
- $\mathsf{AssignedRoles_P}(p_i) = \{r_j : j \in [k] \text{ and } \mathtt{PA}[j][i] = 1\}$.
- $\mathsf{AssignedUsers}(r_j) = \{u_i : i \in [n] \text{ and } \mathtt{UA}[i][j] = 1\}$.
- $\mathsf{AssignedPrms_R}(r_i) = \{p_j : j \in [m] \text{ and } \mathtt{PA}[i][j] = 1\}$.
- $\mathsf{AssignedPrms_U}(u_i) = \{p_j : j \in [m] \text{ and } \mathtt{UPA}[i][j] = 1\}$.

Given the $n \times m$ users-to-permissions assignment matrix UPA, the *role mining problem* (see [28], [8], and [10]) consists in finding a binary decomposition of UPA, that is an $n \times k$ binary matrix UA and a $k \times m$ binary matrix PA such that,

$$\mathtt{UPA} = \mathtt{UA} \otimes \mathtt{PA}, \tag{1}$$

where, the operator $\otimes$ is such that, for $i \in [n]$ and $j \in [m]$,

$$\mathtt{UPA}[i][j] = \bigvee_{h=1}^{k} (\mathtt{UA}[i][h] \wedge \mathtt{PA}[h][j]). \tag{2}$$

Therefore, in solving a role mining problem (see [28] and [8]), we are looking for a factorization of the matrix UPA. Notice that, there are several matrices UA and PA satisfying (1). For instance, the two extreme cases are: *i*) we set a role for each user, hence UA is the $n \times n$ identity matrix and PA = UPA; *ii*) we set a role for each permission, hence UA = UPA and PA is the $m \times m$ identity matrix. In particular, the role mining problem consists in finding a user-to-role assignment $\mathcal{UA}$ and a role-to-permission assignment $\mathcal{PA}$ such that the matrices UA and PA satisfy (1) and the number of columns (rows) of UA (PA) is minimized. The smallest value k for which UPA can be factorized as $\mathtt{UA} \otimes \mathtt{PA}$ is referred to as the *binary rank* of UPA.
A *candidate* role consists of a set of permissions along with a user-to-role assignment. Hence, it can be described by a row of the matrix PA and a column of the matrix UA. The union of the candidate roles is referred to as *candidate role-set* and can be described by matrices UA and PA. A candidate role-set is *complete* if the permissions described by any UPA's row can be exaclty *covered* by the union of some candidate roles. In other words, a candidate role-set is complete if and only if it is a *solution* of the *equation* $\mathtt{UPA} = \mathtt{UA} \otimes \mathtt{PA}$. Hence, equivalently, the role mining problem consists in finding a complete candidate role-set having minimum cardinality. One could consider an *incomplete* role-set as well, where the matrices UA and PA do not cover all permissions represented by the matrix UPA. Such *uncovered* permissions should be handled separately, so they are directly assigned to users defining a *direct user-permission* assignment relation $\mathcal{DUPA} \subseteq \mathcal{U} \times \mathcal{P}$. We can represent such a relations by the binary matrix DUPA satisfying $\mathtt{DUPA}[i][j] = 1$ if and only if $(u_i, p_j) \in \mathcal{DUPA}$. Notice that, $\mathcal{DUPA}$ is not considered in standard RBAC models [27], but this approach is more general and can handle anomalous situation where an assignement of a permission to a user cannot be explained by a role (or, in other words, it does not make sense to introduce for a user a role having a single permission).

The NIST RBAC Reference Model [27] comprises four *model components*. Core RBAC is the one considered ad the beginnig of this section, the other three model are Hierarchical RBAC, Static Separation of Duty Relations, and Dynamic Separation of Duty Relations. Hierarchical RBAC (or RBAC 1, see [26]) adds to Core RBAC a role hierarchy relation $\mathcal{RH} \subseteq \mathcal{R} \times \mathcal{R}$ called *inheritance* relation and denoted by $\succeq$. One has that $r_1 \succeq r_2$ (i.e, role r_1 *inherits* role r_2) if and only if all permissions assigned to r_2 are also assigned to r_1 and all users assigned to r_1 are also assigned to r_2. Formally,

$$\mathsf{AssignedPrms_R}(r_2) \subset \mathsf{AssignedPrms_R}(r_1) \text{ and}$$

$$\mathsf{AssignedUsers}(r_1) \subseteq \mathsf{AssignedUsers}(r_2).$$

We can represent the role hierarchy relation by the binary matrix RH satisfying $\mathtt{RH}[i][j] = 1$ if and only if $r_1 \succeq r_2$.

Following [22], we refer to the tuple $\rho = \langle \mathcal{U}, \mathcal{P}, \mathcal{UPA} \rangle$ as *configuration* of an RBAC instance. As we have previously mentioned, the goal of role mining is to find a suitable decomposition of the matrix UPA, but, depending on the scenario, role mining algorithms could output, as well, an inheritance and a direct user-permission assignment relations. Therefore, in general, given a configuration ρ one wants to find an RBAC state $\gamma = \langle \mathcal{R}, \mathcal{UA}, \mathcal{PA}, \mathcal{RH}, \mathcal{DUPA} \rangle$ that is *consistent* with ρ. The RBAC state γ is consistent with ρ if every user in $\mathcal{U}$ has the same set of permissions in the RBAC state as in $\mathcal{UPA}$. In the case of Core RBAC model, any role mining algorithm will output a configuration γ with both $\mathcal{RH} = \emptyset$ and $\mathcal{DUPA} = \emptyset$, while in the case of Hierarchical RBAC model any algorithm will output $\mathcal{DUPA} = \emptyset$.

3.1 Constrained Role Mining

We recall here the definition of *constrained* role mining problems, that is when a number of constraints may be enforced on different characteristics of the roleset, limiting sometimes the size or the usage of the included roles [14, 12, 4, 3].

We consider here two different kinds of constraints: 1) we limit the number of roles that can be assigned to each user, defining the Role-Usage Cardinality Constraint Role Mining problem (RUCC); 2) we fix an upper bound on the number of permission that can be assigned to each role, defining the Permission-Usage Cardinality Constraint Role Mining problem (PUCC). More formally, we define the *constrained* role mining problems in the following way:

Problem 1. (RUCC) Given a set of user $\mathcal{U}$, a set of permission $\mathcal{P}$ and a user-permission assignement matrix UPA, find a decomposition (UA, PA) for which the following conditions hold: 1) $\texttt{UPA} = \texttt{UA} \otimes \texttt{PA}$; 2) the role-set $\mathcal{R}$ cardinality is minimized; 3) for all users $u \in \mathcal{U}$, it holds that $|\mathsf{AssignedRoles_U}(u)| \leq mru$, where $mru > 1$.

Problem 2. (PUCC) Given a set of user $\mathcal{U}$, a set of permission $\mathcal{P}$ and a user-permission assignement matrix UPA, find a decomposition (UA, PA) for which the following conditions hold: 1) $\texttt{UPA} = \texttt{UA} \otimes \texttt{PA}$; 2) the role-set $\mathcal{R}$ cardinality is minimized; 3) for all roles $r \in \mathcal{R}$, it holds that $|\mathsf{AssignedPrms_R}(r)| \leq mpr$, where $mpr > 1$.

In [14, 12], the previous problems have been proved to be NP-Hard. The computational complexity of the Role Mining problem (and of some of its variants) has been also considered in several papers (see, for instance, [7, 8, 29, 28]). Other related problems considering similar constraints have been defined in [12, 15, 13, 4].

4 Heuristics

Since finding an optimal solution to the constrained role mining problem is NP-hard, we have to resort to some heuristics to get a sub-optimal solution. Our heuristics fall within the post-processing framework where roles are first mined irrespectively of the constraint, using any other known role mining algorithm. Then, the post-processing heuristic takes as input a decomposition of UPA into UA and PA and *manages* to fix the cases that violate the constraints by deleting (unassigning) roles and/or adding (assigning) new ones. In the following sections we present post-processing heuristics to mine roles satisfying the *PUCC* and *RUCC* constraints.

4.1 Post-processing PUCC

In the following we present a post-processing heuristic, referred to as postPUCC, for the Permission-Usage Cardinality Constraint scenario. As required by the post-processing framework, our heuristic takes as input a decomposition of UPA into UA and PA *mined* irrespectively of the constraint and *adjusts* the roles violating the constraint by substituting them with roles (new or existing ones) having less than *mpr* permissions.

To simplify the description of the procedure postPUCC we introduce some data structures (namely, ARU, APR, and CR) representing, in a compact way UA, PA, and all mined roles possessing at most *mpr* permissions contained in a given role of *size* bigger than *mpr*. The data structure ARU represents the roles assigned to users, more precisely, ARU[i] contains the indices of the roles assigned to user u_i; APR represents the permissions assigned to roles (i.e., APR[j] contains the indices of the permissions assigned to role r_j); while CR represents the roles of *size* at most *mpr* contained in a given role possessing more than *mpr* permissions (i.e., if $|\texttt{APR}[j]| > mpr$, then CR[j] contains all the indices j' such that $|\texttt{APR}[j']| \leq mpr$ and $\texttt{APR}[j'] \subset \texttt{APR}[j]$). Previous data structures are filled in by the simple procedure `extractInfo` by exploring the entries of UA and PA. We report it in the following for reader's convenience, but we will not comment on it as it is self-explanatory.

ALGORITHM 1: `extractInfo`

```
input  : A decomposition (UA, PA) of the n × m matrix UPA and the constraint value mpr
output: The data structures ARU, APR, and CR
k = Number of rows in PA
foreach i in [n] do ARU[i] = {ℓ : UA[i][j] = 1}                  // User u_i has role r_j
foreach j in [k] do APR[j] = {ℓ : PA[j][ℓ] = 1}                  // Role r_j has permission p_ℓ
foreach (i, j) in [k] × [k] do                                    // For all pairs of roles
    if |APR[j]| ≤ mpr < |APR[i]| and APR[j] ⊂ APR[i] then        // r_i contains r_j
        CR[i] = CR[i] ∪ {j}
return (ARU, APR, CR)
```

The procedure `postPUCC` described below, starting from a decomposition of UPA into UA and PA, constructs two new matrices `newUA` and `newPA` having, respectively, n rows (one for each user) and m columns (one for each permission) as the corresponding matrices UA and PA. The new matrices will satisfy the permission-usage cardinality constraint as `postPUCC` directily re-assigns the roles having at most *mpr* permissions to each user holding them (i.e., it does reuse the UA assignement). If a user, say u, possesses a role r having more than *mpr* permissions, then our heuristic re-distributes the permissions in r into smaller roles (i.e., roles of dimension at most *mpr*) that are assigned to user u. At the end of the procedure, the matrices `newUA` and `newPA` will represent a complete role-set covering UPA.

ALGORITHM 2: `postPUCC`

```
   input  : A decomposition (UA, PA) of the n × m matrix UPA and the constraint value mpr
   output: A new decomposition (newUA, newPA) of the matrix UPA satisfying the PUCC constraint
1  newUA = [n][·], newPA = [·][m]
2  (ARU, APR, CR) = extractInfo(UA, PA, mpr)
3  foreach i in [n] do                                             // For any user u_i
4      foreach j in ARU[i] do                                      // For all roles r_j assigned to u_i
5          if |APR[j]| ≤ mpr then
6              (newUA, newPA) = update(newUA, newPA, i, APR[j])
7          else
8              tmpAP = APR[j]
9              foreach rc in CR[j] do
10                 (newUA, newPA) = update(newUA, newPA, i, APR[cr])
11                 tmpAP = tmpAP\APR[cr]
12                 if tmpAP == ∅ then break
13             nr = ∅
14             foreach p in tmpAP do
15                 nr = nr ∪ {p}
16                 tmpAP = tmpAP\{p}
17                 if tmpAP == ∅ or |nr| == mpr then
18                     (newUA, newPA) = update(newUA, newPA, i, nr)
19                     nr = ∅
20 return (newUA, newPA)
```

More in detail, in line 2, the procedure `extractInfo` returns the data structures ARU, APR, and CR previuosly described. Then (see lines 3 and 4), procedure `postPUCC` examines all roles assigned to each user according to the decomposition (UA, PA). If the role r_j assigned to user u_i has at most *mpr* permissions (see line 5), then postPUCC, through the procedure update (described below) will update the matrices newUA and newPA by re-assigning r_j to u_i. On the other hand, if role r_j has more than *mpr* permissions, then the procedure postPUCC (see lines 8-12) updates the matrices newUA and newPA by assigning to user u_i all roles described by PA that are *contained* in r_j and have less than *mpr* permissions (i.e., it assigns to u_i the roles represented by CR[j]). If a subset of the roles represented by CR[j] covers all r_j's permissions (line 12), then we have done. Otherwise, we have to reallocate the remaining uncovered permissions in one or more new roles assigning them to user u_i (see lines 13-19). The remaining uncovered permissions (represented by *tmpAP*) are distributed into new roles, each containing at most *mpr* permissions (see lines 13-15). Each new role, represented by the variable *nr*, is then assigned in line 18 to user u_i updating, through the procedure update, the matrices newUA and newPA.

The following procedure update, on input a *partial* decomposition (newUA, newPA) of the matrix UPA, a user u, and a role r, assigns role r to user u by properly modifying newUA and newPA. The matrices newUA and newPA have, respectively, n rows (one for each user) and m columns (one for each permission) as the corresponding matrices UA and PA. In procedure update, the symbol k indcates the number of roles *mined* so far, that is, the number of columns (resp., rows) of the matrix UA (resp., PA).

ALGORITHM 3: update

```
   input  : A partial decomposition (newUA, newPA) of UPA, a user u, and a role r
   output: The modified decomposition (newUA, newPA) of the matrix UPA
1  k = number of rows in newPA
2  flag = True
3  foreach i in [k] do                              // Check whether r already appears in newPA
4      r_i = {j : newPA[i][j] = 1}
5      if r == r_i then
6          newUA[u][i] = 1                          // Assign found role to u
7          flag = False
8          break
9  if flag then                                     // r does not already appear in newPA
10     foreach j in r do                            // Add r to newPA
11         newPA[k + 1][j] = 1
12     newUA[u][k + 1] = 1                          // Assign r to u
13 return (newUA, newPA)
```

The procedure update first checks (see lines 3-5) whether the role described by r already is comprised in newPA. If so, it assigns its row index (i.e., i) to user represented by u. Otherwise (i.e., *flag* is equal to True), the procedure update adds the new role to matrix newPA (lines 10 and 11) and assigns it to user represented by u (line 12). Both procedures extractInfo and update wiil also be used by our heuristic for the RUCC scenario described in the next section.

Illustrative Example for postPUCC. In the following we provide an illustrative example of the execution of our heuristic postPUCC assuming that $mpr = 2$. The procedure starts having in input the matrices UA and PA reported on the right-hand side of Figure 1 that have been computed by running $\mathtt{SMAU}_R$ on the UPA matrix depicted on the left-hand side of Figure 7.

The heuristic postPUCC, for each user $u \in \{u_1, u_2, u_3, u_4, u_5\}$, re-assigns the roles having at most *mpr* permissions, while, in case the role violates the constraint, it re-distributes the included permissions to smaller roles. It is immediate to see that the role $r_1 = \{p_3, p_4, p_5\}$ violates the constraint, and then it is decomposed into two roles, one including permissions $\{p_3, p_4\}$ and the other only $\{p_5\}$ that are assigned to u_1. The new temporary matrices newUA and newPA are as follows.

User u_2 has role r_3 and r_4. Since r_4 does not violate the constraint, it is re-assigned to u_2 (as r'_3), while role r_3 is split into two roles, one with permissions $\{p_1, p_4\}$ and the other with permission $\{p_5\}$ (role already

	p_1	p_2	p_3	p_4	p_5
u_1	0	0	1	1	1
u_2	1	1	0	1	1
u_3	1	1	0	0	1
u_4	0	1	1	1	0
u_5	1	0	0	1	1

	r_1	r_2	r_3	r_4	r_5
u_1	1	0	0	0	0
u_2	0	0	1	1	0
u_3	0	0	0	1	1
u_4	0	1	0	0	0
u_5	0	0	1	0	0

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	1
r_2	0	1	1	1	0
r_3	1	0	0	1	1
r_4	0	1	0	0	0
r_5	1	0	0	0	1

Figure 1: `UPA` matrix (left) and `UA` and `PA` matrices (right)

	r'_1	r'_2
u_1	1	1

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0
r'_2	0	0	0	0	1

Figure 2: `newUA` matrix (left) and `newPA` matrix (right)

present in `newPA` as r'_2; the updated matrices `newUA` and `newPA` are reported in Figure 3

	r'_1	r'_2	r'_3	r'_4
u_1	1	1	0	0
u_2	0	1	1	1

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0
r'_2	0	0	0	0	1
r'_3	0	1	0	0	0
r'_4	1	0	0	1	0

Figure 3: `newUA` matrix (left) and `newPA` matrix (right)

Both roles assigned to u_3 include less than *mpr* permissions, and can be reassigned. A new role r'_5 is included is `newPA` and assigned to u_3 together with the already existing role r'_3. After these updates, the matrices `newUA` and `newPA` are in Figure 4.
Since the role assigned to user u_4 violates the constraint, it is split into two new roles, both are already present in `newPA` (they correspond to r'_1 and r'_3) that remains unchanged, while `newUA` contains the new role assignment for u_4 as reported in Figure 5.
Also for user u_5, the assigned role includes more than *mpr* permissions and for this reason it is split into two roles, one with permissions $\{p_1, p_4\}$), and the other with permission $\{p_5\}$). Both roles have already been defined in `newPA` and correspond to roles $\{r'_4\}$ and $\{r'_2\}$. The matrix `newUA` includes the new assignment for u_5, while `newPA` is not modified as depicted in Figure 6. Since no more users are left to be examined these matrices are the ones returned by heuristic `postPUCC`.

4.2 Post-processing RUCC

In the following we present an heuristics for the post-processing framework referred to `postRUCC`. Such an heuristic takes as input a decomposition of `UPA` into `UA` and `PA` mined irrespectively of the constraint and re-assigns roles to each user regardless of the constraint's violation trying to reduce the overall number of roles assigned to each user. In short, it tries to covers all the permissions of each user by using the minimum number of roles described by `PA`. Our heuristic, first sort users in decreasing order with respect to the number of assigned permissions, the it cover them. In effect, for each user, we compute an approximation of the minimum covering, as cover user's permissions using the minimum number of roles is an NP-Hard problem. Indeed, it is easy to see that such problem corresponds to the Set-Covering Problem (for its decisional version, see SP25 in [11]). If the number of required roles exceeds the threshold *mru*, then we select the *first* $mru - 1$ roles, we *transfer* the remaining uncovered permissions to a new role, and we assign the $mru - 1$ selected roles and the new one to the user. We decided to select the *first* up to $mru - 1$ roles, but any strategy could be used. Indeed, we could have chosen any random $mru - 1$ roles (we experimentally saw that there is no such a great difference and sometime the random choice produced worse results) or any up to $mru - 1$ roles

	r'_1	r'_2	r'_3	r'_4	r'_5
u_1	1	1	0	0	0
u_2	0	1	1	1	0
u_3	0	0	1	0	1

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0
r'_2	0	0	0	0	1
r'_3	0	1	0	0	0
r'_4	1	0	0	1	0
r'_5	1	0	0	0	1

Figure 4: `newUA` matrix (left) and `newPA` matrix (right)

	r'_1	r'_2	r'_3	r'_4	r'_5
u_1	1	1	0	0	0
u_2	0	1	1	1	0
u_3	0	0	1	0	1
u_4	1	0	1	0	0

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0
r'_2	0	0	0	0	1
r'_3	0	1	0	0	0
r'_4	1	0	0	1	0
r'_5	1	0	0	0	1

Figure 5: `newUA` matrix (left) and `newPA` matrix (right)

belonging to the greatest number of users (this will cover a larger part of `UPA`, but to determine such $mru-1$ roles could take a prohibitively large amount of time).

ALGORITHM 4: `postRUCC`

input : The $n \times m$ matrix `UPA`, its decomposition into `UA` and `PA`, and the threshold mru
output: A new decomposition `newUA` and `newPA` of `UPA` satisfying the RUCC constraint

1 Sort `UA`'s rows in decreasing order with respect to the number of permissions in them
2 `newUA` $= [n][\cdot]$, `newPA` $= [\cdot][m]$
3 $k =$ *number of rows in* `PA`
4 (`ARU`, `APR`, `CR`) = `extractInfo`(`UA`, `PA`, 0)
5 **for** $i = 1$ **to** n **do**
6     $perms = \{j : \texttt{UPA}[i][j] = 1\}$
7   `COVER` = `approxCover`($perms$, `APR`)
8   **if** $|\texttt{COVER}| \leq mru$ **then** $\ell = |\texttt{COVER}|$ **else** $\ell = mru - 1$
9   **for** $j = 1$ **to** ℓ **do**
10         $r = \texttt{COVER}[j]$ `// j-th role in the cover`
11   (`newUA`, `newPA`) = `update`(`newUA`, `newPA`, i, `APR`[r])
12         $perms = perms \setminus \texttt{APR}[r]$
13   **if** $perms \neq \emptyset$ **then**
14   (`newUA`, `newPA`) = `update`(`newUA`, `newPA`, i, $perms$)
15   **if** $perms \notin \texttt{APR}$ **then**
16             $k = k + 1$
17             $\texttt{APR}[k] = perms$ `// Add the new role to the role-set`
18 **return** (`newUA`, `newPA`)

In line 4, Heuristic `postRUCC`, using procedure `extractInfo` described in Section 4.1, computes a compact representation of the matrices `UA` and `PA`. Since, `postRUCC` does not need the data structure `CR`, the value mpr is set equal to 0. In lines 5-17, `postRUCC` re-assigns roles to users so that at most mru roles will be distributed to each user. More specifically, in line 7, our heuristic tries to cover all permissions of user u_i by using the mined roles represented by `APR`. Since to to cover u_i's permissions using the the minimum number of roles is an NP-Hard problem, `postRUCC` uses the *classical* greedy approximation algorithm solving the Set-Covering Problem [32]. Once a covering has been obtained, the `postRUCC` checks whether it contains at most mru roles (see line 8). If so, all such roles are assigned to user u_i (see lines 9-12); otherwise, only the *first* $mru - 1$ roles returned by `approxCover` are assigned to u_i (again, see lines 9-12). Such an assignment is done (see line 11) by the procedure `update` described in Section 4.1. The procedure `update` assigns the

	r'_1	r'_2	r'_3	r'_4	r'_5
u_1	1	1	0	0	0
u_2	0	1	1	1	0
u_3	0	0	1	0	1
u_4	1	0	1	0	0
u_5	0	1	0	0	1

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0
r'_2	0	0	0	0	1
r'_3	0	1	0	0	0
r'_4	1	0	0	1	0
r'_5	1	0	0	0	1

Figure 6: `newUA` matrix (left) and `newPA` matrix (right)

role `APR`$[r]$ to user u_i by appropriately modifying the matrix `newUA` and also adds it to `newPA` if not present. In line 12, `postRUCC`, by means of the variable *perms* keeps track of u_i's uncovered permissions. If, after executing the lines 9-12, there still are uncovered permissions (see line 13), then they are *packed* into a role and assigned to user u_i using the procedure `update` (see line 14). Notice that, the test in line 13 will be satisfied (i.e., there are uncovered permissions) when the procedure `approxCover` returns more than *mru* roles (see line 8). If the role role induced by *perms* is a new one (i.e., the role represented by *perms* does not belong to the role-set represented by `APR`, see line 15), then, in line 17, it will added to the data structure `APR`.

ALGORITHM 5: `approxCover`

input : A set of permissions *perms* and the set of roles `APR`
output: The set of roles $coveringRoles \subseteq$ `APR` covering the permissions *perms*

1 $origPerms = perms$
2 $coveringRoles = \emptyset$
3 $k =$ *number of roles in* `APR`
4 **while** $perms \neq \emptyset$ **do**
5     $max = idx = 0$
6   `// Select a role covering the maximum number of uncovered permission`
7   **foreach** r **in** $[k]$ **do**
8   **if** `APR`$[r] \subseteq origPerms$ **and** $|perms \cap$ `APR`$[r]| > max$ **then**
9             $max = |perms \cap$ `APR`$[r]|$
10             $idx = r$
11     $perms = perms \setminus$`APR`$[idx]$
12     $coveringRoles = coveringRoles \cup \{idx\}$
13 **return** *coveringRoles*

The procedure `approxCover` returns a covering of user's u_i permissions using the roles in `APR`. We will not comment on such a procedure as it implements the *classical* greedy approximation algorithm solving the Set-Covering Problem [32]. Notice that in lines 7-10 we could have used any strategy to select the roles to add to the covering. For instance, we could have chosen first the roles that have been assigned to the maximum number of users. In the procedure `approxCover` we preferred to choose the roles according to the *classical* strategy as, by experimental analysis, we have noticed that other strategies do not improve the quality of the computed role-set.

We conclude this section by pointing out that heuristic `postRUCC` could have assigned to a user u_i the roles returned by `approxCover` only if $|$`COVER`$| < |$`UA`$[i]| \leq mru$. That is, `postRUCC` uses the roles in `COVER` only if they are fewer than the roles originally assigned to u_i. We implemented both `postRUCC` and the previously described variant and observed that in only six out of 12.859 tests the proposed variant returns a role-set smaller (by one) than the one computed by `postRUCC`; while in nine tests `postRUCC` returns a smaller role-set than that computed by the variant. Hence, to keep `postRUCC`'s description simple, we preferred not to add this variant to `postRUCC`.

Illustrative Example for `postRUCC`. In the following we provide an illustrative example of the execution of our heuristic `postRUCC` when $mru = 2$. We assume that `postRUCC` receives as input the matrices `UA` and

PA described on the right-hand side of Figure 7 that have been computed by running $\texttt{SMAU}_R$ on the UPA matrix depicted on the left-hand side of Figure 7.

	p_1	p_2	p_3	p_4	p_5
u_1	0	0	1	1	0
u_2	0	0	1	1	1
u_3	1	1	1	1	1
u_4	0	0	0	0	1
u_5	1	0	1	1	1

	r_1	r_2	r_3	r_4
u_1	0	1	0	0
u_2	1	1	0	0
u_3	1	1	1	1
u_4	1	0	0	0
u_5	1	1	1	0

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	0	0	1
r_2	0	0	1	1	0
r_3	1	0	0	0	0
r_4	0	1	0	0	0

Figure 7: UPA matrix (left) and UA and PA matrices (right)

The heuristic postRUCC, for each user $u \in \{u_1, u_2, u_3, u_4, u_5\}$, invokes the function approxCover to cover u's permissions using the roles in ARU, while the function update *build* the new user-to-role and role-to-permission matrices newUA and newPA. The role-set ARU is initialized with the roles mined by $\texttt{SMAU}_R$ (i.e., $\texttt{ARU} = \{r_1, r_2, r_3, r_4\}$. It is immediate to see that the role $r_2 = \{p_3, p_4\}$ covers all permissions of user u_1. Hence, the function approxCover returns role r_2 (its index 2) and the new temporary matrices newUA and newPA are as follows.

	r'_1
u_1	1

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0

Figure 8: newUA matrix (left) and newPA matrix (right)

For user u_2, the function approxCover returns the roles r_2 and r_1 in ARU (to be precise, approxCover returns roles' indices 2 and 1). Both roles are assigned to u_2 in newUA, role r_1 already appears in newPA as r'_1, while r_2 is added, as role r'_2, to newPA. Hence, the matrices newUA and newPA are as follows.

	r'_1	r'_2
u_1	1	0
u_2	1	1

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0
r'_2	0	0	0	0	1

Figure 9: newUA matrix (left) and newPA matrix (right)

Considering the user u_3's permissions (i.e., $\{p_1, p_2, p_3, p_4, p_5\}$), the function approxCover, returns the roles r_2, r_1, r_3, and r_4. Since, the number of returned roles is larger than the constraint's value $mru = 2$, the heuristic postRUCC assigns $r_2 = \{p_3, p_4\}$ (named r'_1 in newUA) to u_3. Then, a new role r'_3, containing the permissions $\{p_1, p_2, p_5\}$, is formed and added both to newUA and ARU. After these updates, the matrices newUA and newPA are in Figure 10.

	r'_1	r'_2	r'_3
u_1	1	0	0
u_2	1	1	0
u_3	1	0	1

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0
r'_2	0	0	0	0	1
r'_3	1	1	0	0	1

Figure 10: newUA matrix (left) and newPA matrix (right)

User u_4 has assigned only the permission p_5 that is the unique permission in role r_1 of the original PA matrix (this role corresponds to role r'_2 of the newPA matrix). In this case, the function approxCover returns the role r_1 (i.e., the *new* role r'_2) that is assigned to u_4. Finally, for user u_5, possessing permissions $\{p_1, p_3, p_4, p_5\}$), the function approxCover returns the role r_2, r_1, and r_3. The number of returned roles exceed the maximum number of roles (e.g., 2) that can be assigned to any user in this example. Therefore, among the roles returned by approxCover only the role $r_2 = \{p_3, p_4\}$, corresponding to r'_1 in newPA, is assigned to u_5. Then, the new

	r'_1	r'_2	r'_3	r'_4
u_1	1	0	0	0
u_2	1	1	0	0
u_3	1	0	1	0
u_4	0	1	0	0
u_5	1	0	0	1

	p_1	p_2	p_3	p_4	p_5
r'_1	0	0	1	1	0
r'_2	0	0	0	0	1
r'_3	1	1	0	0	1
r'_4	1	0	0	0	1

Figure 11: `newUA` matrix (left) and `newPA` matrix (right)

role $r'_4 = \{p_1, p_5\}$ is formed and assigned to u_5. The matrices `newUA` and `newPA` are modified accordingly and depicted in the following Figure 11.
No more users are left to be examined and the heuristic `postRUCC` returns the user-to-role and role-to-permission matrices `newUA` and `newPA` described in Figure 11.

5 Experimental Evaluation

In this section, we compare our heuristics with state of the art ones. Since all heuristics have almost the same running time, heuristics will not be evaluated by this means. Indeed, we run a set of experiments to assess heuristics' performance measuring by the *quality* of the RBAC state returned by them. In particular, we consider the size of the role-set and the *Weighted Structural Complexity* (WSC). The Weighted Structural Complexity measures the *size* of a Core RBAC state $\gamma = \langle \mathcal{R}, \mathcal{UA}, \mathcal{PA}, \mathcal{RH}, \mathcal{DUPA} \rangle$ that is consistent with a given configuration $\rho = \langle \mathcal{U}, \mathcal{P}, \mathcal{UPA} \rangle$ of a Core RBAC instance. Given a role hierarchy relation $\mathcal{RH}$, its transitive reduction $t_{reduce}(\mathcal{RH})$ is the minimum relation having the same transitive closure as $\mathcal{RH}$. For instance, $\{(r_1, r_2), (r_2, r_3)\}$ is the transitive reduction of $\{(r_1, r_2), (r_2, r_3), (r_1, r_3)\}$. According to [16, 22] the Weighted Structural Complexity is defined as follows.

Definition 5.1 *Given $W = \langle w_r, w_u, w_p, w_h, w_d \rangle$, where $w_r, w_u, w_p, w_h, w_d \in \mathbb{Q}^+ \cup \{\infty\}$, the* Weighted Structural Complexity *(WSC) of an RBAC state γ, denoted by $wsc(\gamma, W)$, is computed as follow.*

$$wsc(\gamma, W) = w_r \cdot |\mathcal{R}| + w_u \cdot |\mathcal{UA}| + w_p \cdot |\mathcal{PA}| + w_h \cdot |t_{reduce}(\mathcal{RH})| + w_d \cdot |\mathcal{DUPA}|$$

where $|\cdot|$ denotes the size of the set or relation.

Given a weight vector $W = \langle w_r, w_u, w_p, w_h, w_d \rangle$, one would like to find an RBAC state having the smallest Weighted Structural Complexity. Hence, different weight vectors encode different mining objective and minimization goals. For example, by setting $W = \langle 1, 0, 0, \infty, \infty \rangle$ one wants to minimize the number of role forbidding role hierarchy and direct user-permission assignment; while, setting $W = \langle 0, 1, 1, \infty, \infty \rangle$ one wants to minimize the number of assignments user-roles and role-permissions (this problem was referred to as *min-edge role mining* in [19]). In our case we set $W = \langle 1, 1, 1, 0, \infty \rangle$, because we want to compare heuristics that generate RBAC states exhibiting a complete role-set (i.e., we do not allow direct user-permission assignment) and we stick to the Core RBAC model, where hierarchy relations do not come into play (since our heuristics and the ones we compare with, do not generate roles hierarchies).

5.1 Test-bed

All heuristics have been implemented in Python 3.9 and tested on a MacBook Pro running OS X 11.11.2 on a 2.3 GHz Intel Core i9 8 core CPU having 16 GB 2667 MHz DDR4 RAM. In the evaluation, we use nine real-world datasets that have been widely used in literature for analyzing the performances of various role mining heuristics (see, for instance, [8, 24, 15, 12, 14]). The parameters of the real-world datasets are summarized in Table 1 where, for each dataset, we report the number of users $|\mathcal{U}|$, the number of permissions $|\mathcal{P}|$, the number of user-to-permission assignments $|\mathcal{UPA}|$, the minimum and the maximum number of permissions assigned to a user (respectively, min#P and max#P), and the minimum and the maximum number of users that have the same permission (respectively, min#U and max#U)[1]. The last column of Table 1 contains the density of the `UPA` matrix, that is the number of entries equal to one with respect its size.

Dataset	$\|\mathcal{U}\|$	$\|\mathcal{P}\|$	$\|\mathcal{UPA}\|$	min#P	max#P	min#U	max#U	Density
Americas Large	3485	10127	185294	1	733	1	2812	0.53%
Americas Small	3477	1587	105205	1	310	1	2866	1.61%
Apj	2044	1164	6841	1	58	1	291	0.29%
Customer	10021	277	45427	1	25	1	4184	1.64%
Domino	79	231	730	1	209	1	52	4.00%
Emea	35	3046	7220	9	554	1	32	6.77%
Firewall 1	365	709	31951	1	617	1	251	12.35%
Firewall 2	325	590	36428	6	590	46	298	19.00%
Healthcare	46	46	1486	7	46	3	45	70.23%

Table 1: Characteristics of the real-world datasets considered in this paper

The datasets *Americas small* and *Americas large* were obtained from Cisco firewalls granting access to the HP network to authenticated users (users' access depends on their profiles). Similar datasets are *Apj* and *Emea*. The *Healthcare* dataset was received from the US Veteran's Administration; the *Domino* data was from a Lotus Domino server; *Customer* is based on the access control graph obtained from the IT department of an HP customer. Finally, the *Firewall 1* and *Firewall 2* datasets are results of running an analysis algorithm on Checkpoint firewalls. Such real-world datasets were publicly available on the web page at HP Labs of one of the authors of [8]. With the exception of the dataset *Customer*, the optimal decompositions (i.e., a representation of a minimum size role-set along with a *user-to-role* assignment relation) were available as well. From such optimal decompositions, we derived the information listed in Table 2. For the dataset *Customer*, since an optimal decomposition was not publicly available, we extrapolated data (listed in boldface) from the *user-to-permission* assignment relation.

Dataset	$\|\mathcal{R}\|$	$\overset{min}{ppr}$	$\overset{max}{ppr}$	$\overset{min}{rpu}$	$\overset{max}{rpu}$
Americas large	398	1	733	1	4
Americas small	178	1	263	1	12
Apj	453	1	52	1	8
Customer	276	**1**	**25**	**1**	**25**
Domino	20	1	201	1	9
Emea	34	9	554	1	1
Firewall 1	64	1	395	1	9
Firewall 2	10	2	307	1	3
Healthcare	14	1	32	1	6

Table 2: Characteristics of optimal decomposition (in boldface *not optimal* data)

In Table 2, the columns indexed by $\overset{min}{ppr}$ and $\overset{max}{ppr}$ contain, respectively, the minimum and the maximum number of permissions assigned to roles in the optimal decompositions; while, the columns indexed by $\overset{min}{rpu}$ and $\overset{max}{rpu}$ represent, respectively, the minimum and maximum number of roles assigned to users. For the dataset *Customer*, the values $\overset{max}{ppr}$ and $\overset{max}{rpu}$ were substituted by their upper bound max#P given in the fifth column of Table 1.

In the post-processing framework, any heuristic starts from a complete decomposition of `UPA` into two matrices `UA` and matrix `PA` such that $\mathtt{UPA} = \mathtt{UA} \otimes \mathtt{PA}$. Such matrices can be computed in several ways. Our experiments will consider decompositions obtained applying the techniques in [30], [8], [1], and [19], as well

[1]Formally, min#P is defined as $\min\{|\mathsf{AssignedPrms}_{\mathsf{U}}(u)| : u \in \mathcal{U}\}$, we can define max#P, min#U, and max#U analogously.

as, the optimal decomposition available from HP Labs. The heuristics used to get the decompositions used in this paper are summarized in Figure 12.

`Optimal`, `Biclique` [8] $\texttt{SMA}_R$, $\texttt{SMAU}_R$, $\texttt{SMA}_C$, $\texttt{SMAU}_C$ [1] `FastMiner`[30] `OBMD` [19]

Figure 12: Heuristics used to compute the starting `UPA` decompositions

In [30], the heuristic `FastMiner` computes a complete role-set starting from an *initial* role-set formed by grouping the users having the same set of permissions and forming a role for each set of such common permissions. Then, roles formed by the permissions in the intersections between pairs of initial roles are added to the initial role-set. Notice that `FastMiner` could generate *redundant* roles. For a user i, a role r is redundant if the permissions associated to r are a subset of the permissions associated to other roles assigned to user i. The heuristic in [19] try reduce such redundancy. In [19] (see also [31]), the heuristic for computing an `UPA`'s decomposition uses a greedy strategy that, starting from a role-set obtained by running `FastMiner`, selects a subset of such roles that cover all ones in `UPA`. More precisely, the heuristic selects a role that can be assigned to as many users as possible without violating relation (2) of Section 3. This process is repeated until all ones in `UPA` are covered. In this paper, the heuristic in [19] will be referred to as `OBMD` (Optimal Boolean Matrix Decomposition). In [8], the user-to-permission assignment relation $\mathcal{UPA}$ is represented as a bipartite graph G. Any biclique in G (i.e., a complete bipartite subgraph of G) identifies a role (i.e., users assigned to the role along with the permissions included in the role itself). The heuristic `Biclique` [8] aims at finding a biclique cover of all the edges of the bipartite graph G. Finally, in [1], the heuristic $\texttt{SMA}_R$ generates the role-set by covering the matrix `UPA` using its rows. That is, first $\texttt{SMA}_R$ forms a role r by considering the permissions in a row with the smallest number of ones in it. Then, it assign r to any user possessing the permissions in r. Another heuristic in [1], referred to as $\texttt{SMA}_C$, forms roles by considering `UPA`'s columns (i.e., it is a sort of $\texttt{SMA}_R$ run on `UPA`'s transpose by interchanging the *functions* of users and permissions). To form a role, as stressed in [5], the permissions can be picked out either from the ones in `UPA` (as done in the heuristics $\texttt{SMA}_R$ and $\texttt{SMA}_C$) or from the permissions left uncovered during the mining steps. Hence, from the heuristics $\texttt{SMA}_R$ and $\texttt{SMA}_C$, another two heuristics can be derived, namely $\texttt{SMAU}_R$ and $\texttt{SMAU}_C$. Such heuristics, generate a role by considering, each time, a reduced instance of the problem (i.e., a user-to-permission assignment matrix containing only uncovered permissions).

To get a complete decomposition of `UPA` used to test our heuristics, we ran the heuristics $\texttt{SMA}_R$, $\texttt{SMAU}_R$, $\texttt{SMA}_C$, $\texttt{SMAU}_C$, `FastMiner`, `OBMD`, and `Biclique` on the real-world datasets listed in Table 1. We report the role-set sizes and the WSC values obtained running these heuristics in Table 3 where the second column, except for the *Customer* dataset, contains $|\mathcal{R}|$ and WSC of the optimal decompositions given in [8].

5.2 Experiments

To run the experiments, for the $PUCC$ and $RUCC$ scenario, we have to fix the constraint values. Except that for the *Customer* dataset, we know the optimal decomposition for the real-world datasets in Table 1. Hence, to choose the constraint values used in our tests, we consider the characteristics of the optimal solutions summarized in Table 2. For each combination of dataset, heuristic, and *starting* decomposition, we run a test changing the constraint's value. In particular, the constraint values for the $PUCC$ scenario will be set to the 10%, 30%,50%, 80%, and 100% of $\overset{max}{ppr}$. In the last test, setting mpr equal to $\overset{max}{ppr}$ allows us to compare the heuristics against the optimal solution. From Table 2, one can see that the optimal solutions distribute few roles to each user. Hence, for the $RUCC$ scenario, we adopt a slightly different method to select the mru values used in our tests. For all datasets, except *Emea*, the first value mru will take is 2, the last but one is $\overset{max}{rpu}$, and the last value is 20% bigger than $\overset{max}{rpu}$. Moreover, if possible, we add at most another two equally spaced values between the first and the last but one. For the dataset *Emea*, the optimal decomposition, for the unconstrained case, distributes one role to each user. Hence, in our tests mru will takes value in $(1, 2, 3, 4, 5)$. We summarize in Table 4 the constraint values mpr and mru.

Dataset	`Optimal`	`SMA`$_R$	`SMAU`$_R$	`SMA`$_C$	`SMAU`$_C$	`FastMiner`	`OBMD`	`Biclique`	
Americas Large	398	430	415	612	416	6528	564	423	$\|\mathcal{R}\|$
	95407	107624	93138	91237	95176	1017743	126433	101494	WSC
Americas Small	178	225	207	204	198	1778	202	213	$\|\mathcal{R}\|$
	11217	22950	11656	15251	15978	168086	20947	22173	WSC
Apj	453	475	455	465	453	781	466	456	$\|\mathcal{R}\|$
	4867	6391	5115	5524	5271	12807	6373	5770	WSC
Customer	-	1154	276	276	276	40616	297	276	$\|\mathcal{R}\|$
	-	55184	45978	45845	45893	819509	48652	45978	WSC
Domino	20	20	20	22	20	64	21	20	$\|\mathcal{R}\|$
	754	789	761	775	758	1845	899	762	WSC
Emea	34	34	34	40	34	242	43	34	$\|\mathcal{R}\|$
	7280	7280	7280	7595	7280	23026	9086	7280	WSC
Firewall 1	66	71	68	74	65	266	66	69	$\|\mathcal{R}\|$
	2019	6517	3273	5020	5231	26680	5445	5531	WSC
Firewall 2	10	10	10	10	10	20	10	10	$\|\mathcal{R}\|$
	1120	1965	1564	1469	1466	3147	1977	1772	WSC
Healthcare	14	16	14	14	14	29	14	15	$\|\mathcal{R}\|$
	268	797	369	425	542	1314	685	444	WSC

Table 3: $|\mathcal{R}|$ and WSC of state-of-the-art heuristics (unconstrained scenario) for real-world datasets

Dataset	mpr values	mru values
Americas Large	73, 220, 367, 586, 733	2, 3, 4, 5
Americas Small	26, 79, 132, 210, 263	2, 6, 10, 12, 14
Apj	5, 16, 26, 42, 52	2, 4, 6, 8, 10
Customer	3, 8, 13, 20, 25	2, 4, 6, 8, 10
Domino	20, 60, 101, 161, 201	2, 4, 7, 9, 11
Emea	55, 166, 277, 443, 554	1, 2, 3, 4, 5
Firewall 1	40, 119, 198, 316, 395	2, 4, 7, 9, 11
Firewall 2	31, 92, 154, 246, 307	2, 3, 4
Healthcare	3, 10, 16, 26, 32	2, 4, 6, 7

Table 4: mpr and mru values used in the experiments

PUCC Scenario. To the best of our knowledge, beside the heuristic `postPUCC` presented in Section 4.1, in the current literature there are no other heuristics for the $PUCC$ scenario in the post-processing framework. Hence, in the following we report some of the results of the applications of our heuristics to the decompositions summarized in Table 3. As an example, in Tables 5 and 6, we report the results for the datasets *Apj* and *Healtcare* when executing the heuristic `postPUCC` on the decompositions summarized in Figure 12 for the mpr values given in Table 4.

We selected these two datasets as *Apj* has a low density (i.e., 0.29%), while *Healthcare* has a high density (i.e., 70.23%). The experiments on the other datasets are available online in the supplemental material [6]. Considering the *Apj* dataset, we notice that for small values of mpr (i.e., $mpr \in \{5, 16\}$), our heuristic computes the smaller role-set, when starting from the `SMAU`$_R$ decomposition, while, for larger values of mpr, it generates a smaller role-set when starting from the `Optimal` decomposition. Anyway, except for the the case $mpr = 5$, the size of the role-sets computed starting either from the `Optimal` decomposition or from the `SMAU`$_R$ one differs just by at most two units. If we consider the WSC value, from Table 5 we see that, regardless of the constraint value, our heuristic returns a solution with smaller WSC when starting from an `Optimal` decomposition. The second best WSC value is attained when `postPUCC` starts from the `SMAU`$_R$ decomposition. From Table 5, we also notice that, independently of the decomposition given as input to `postPUCC`, when the value assigned to mpr increases, the number of generated roles decreases, in some cases to a large extent. This reduction was someway expected, as larger roles, usually, can cover larger parts of

Decomposition	10%	30%	50%	80%	100%	
`Optimal`	564	467	458	454	453	$\|\mathcal{R}\|$
	5233	4898	4878	4870	4867	WSC
$\texttt{SMA}_R$	644	518	489	478	476	$\|\mathcal{R}\|$
	6407	6365	6395	6398	6394	WSC
$\texttt{SMAU}_R$	537	467	459	455	455	$\|\mathcal{R}\|$
	5329	5141	5124	5115	5115	WSC
$\texttt{SMA}_C$	618	506	479	468	466	$\|\mathcal{R}\|$
	5629	5556	5554	5531	5527	WSC
$\texttt{SMAU}_C$	604	492	467	456	454	$\|\mathcal{R}\|$
	5350	5282	5276	5278	5274	WSC
`FastMiner`	1026	821	795	784	782	$\|\mathcal{R}\|$
	12239	12242	12785	12814	12810	WSC
`OBMD`	619	509	480	469	467	$\|\mathcal{R}\|$
	6686	6383	6378	6380	6376	WSC
`Biclique`	600	492	469	459	457	$\|\mathcal{R}\|$
	5795	5783	5773	5777	5773	WSC

Table 5: $|\mathcal{R}|$ and WSC for the *Apj* dataset

the `UPA` matrix. Therefore, less roles have to be generated.

Decomposition	10%	30%	50%	80%	100%	
`Optimal`	37	20	17	15	14	$\|\mathcal{R}\|$
	674	378	320	289	268	WSC
$\texttt{SMA}_R$	56	31	24	19	18	$\|\mathcal{R}\|$
	1579	974	844	780	803	WSC
$\texttt{SMAU}_R$	24	16	15	14	14	$\|\mathcal{R}\|$
	706	461	415	369	369	WSC
$\texttt{SMA}_C$	52	29	22	17	16	$\|\mathcal{R}\|$
	728	427	388	376	414	WSC
$\texttt{SMAU}_C$	48	27	21	17	15	$\|\mathcal{R}\|$
	1239	759	650	577	561	WSC
`FastMiner`	66	44	38	32	32	$\|\mathcal{R}\|$
	1854	1328	1256	1197	1308	WSC
`OBMD`	53	29	22	17	16	$\|\mathcal{R}\|$
	1512	922	780	668	691	WSC
`Biclique`	52	28	21	18	16	$\|\mathcal{R}\|$
	792	482	442	454	446	WSC

Table 6: $|\mathcal{R}|$ and WSC for the dataset *Healthcare*

In Table 6, we report the results of our experiments on the high density dataset *Healthcare*. In this case, the smallest role-set is obtained starting from the $\texttt{SMAU}_R$ decomposition, and, as in the previous case, our heuristics generate a solution with the smallest WSC value when it receives as input the `Optimal` decomposition. We notice the same pattern, as well as, for the dataset *Firewall 2* having a density equal to 19%. Such a behaviour might depend on the density of the `UPA` matrix.

If the experiments results are described as in Tables 5 and 6, then, to reduce either the role-set size or the WSC value, it could be difficult to deduce what decomposition is preferable over the others as input of the heuristic `postPUCC`. Therefore, we rank the decomposition using the method used in [24] and [5]. Since there are eight possible decomposition (seven in the case of the dataset *Customer*), we rank them from 1 to 8 (from 1 to 7, for the dataset *Customer*). More precisely, considering the Table 6, for each fixed column and a given evaluation criterion (i.e., $|\mathcal{R}|$ or WSC), we assign a rank from 1 to 8 to each decomposition. A lower rank is better. If two or more decompositions produce a tie, they will be given the same ranking such that the sum of the ranking of all eight decompositions remains constant and equal to 36 as $1+2++8=36$.

Dataset	$\|\mathcal{R}\|$						WSC					
	10%	30%	50%	80%	100%	*avg*	10%	30%	50%	80%	100%	*avg*
`Optimal`	2.0	2.0	2.0	2.0	1.5	1.9	1.0	1.0	1.0	1.0	1.0	1.0
$\texttt{SMA}_R$	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0
$\texttt{SMAU}_R$	1.0	1.0	1.0	1.0	1.5	1.1	2.0	3.0	3.0	2.0	2.0	2.4
$\texttt{SMA}_C$	4.5	5.5	5.5	4.0	5.0	4.9	3.0	2.0	2.0	3.0	3.0	2.6
$\texttt{SMAU}_C$	3.0	3.0	3.5	4.0	3.0	3.3	5.0	5.0	5.0	5.0	5.0	5.0
`FastMiner`	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0
`OBMD`	6.0	5.5	5.5	4.0	5.0	5.2	6.0	6.0	6.0	6.0	6.0	6.0
`Biclique`	4.5	4.0	3.5	6.0	5.0	4.6	4.0	4.0	4.0	4.0	4.0	4.0

Table 7: Rank for the *Healthcare* dataset

Consider, for instance, the column of Table 6 with label 10%. Both the decompositions $\texttt{SMA}_C$ and `Biclique`, used as input of `postPUCC`, determine a role-set of size 52. Using the decompositions `Optimal`, $\texttt{SMAU}_R$, and $\texttt{SMAU}_C$ one gets smaller role-sets and, using the remaining decompositions, the resulting role-sets will be larger. The decompositions $\texttt{SMA}_C$ and `Biclique` are tied both for fourth place. Hence, the rank assigned to them is $4.5 = (4+5)/2$. As another example, if four decompositions are tied for third place, then they will all be given the rank $4.5 = (3+4+5+6)/4$. In Table 7, we report the ranking of all decompositions for the dataset *Healthcare*. For each decomposition, the seventh and thirteenth columns report the average ranking over the five experiments of the role-set size and the Weighted Structural Complexity, respectively.

Dataset	`Optimal`	$\texttt{SMA}_R$	$\texttt{SMAU}_R$	$\texttt{SMA}_C$	$\texttt{SMAU}_C$	`FastMiner`	`OBMD`	`Biclique`
Americas Large	**1.4**	5.2	1.6	6.2	3.0	8.0	6.6	4.0
Americas Small	**1.2**	7.0	2.8	3.8	3.0	8.0	4.2	6.0
Apj	**1.3**	7.0	1.9	5.0	3.1	8.0	6.0	3.7
Customer	-	6.0	**1.8**	2.7	3.7	7.0	5.0	**1.8**
Domino	3.0	4.2	**2.2**	5.6	4.2	8.0	6.6	**2.2**
Emea	**3.0**	**3.0**	**3.0**	6.0	**3.0**	8.0	7.0	**3.0**
Firewall 1	**1.6**	6.4	2.6	6.4	2.3	8.0	3.5	5.2
Firewall 2	1.7	5.3	**1.3**	4.9	5.3	8.0	5.3	4.2
Healthcare	1.9	7.0	**1.1**	4.9	3.3	8.0	5.2	4.6

Table 8: Average rank for the real-world datasets (role-set size)

Dataset	`Optimal`	$\texttt{SMA}_R$	$\texttt{SMAU}_R$	$\texttt{SMA}_C$	$\texttt{SMAU}_C$	`FastMiner`	`OBMD`	`Biclique`
Americas Large	4.0	6.0	1.8	**1.2**	3.0	8.0	7.0	5.0
Americas Small	**1.2**	6.6	1.8	3.0	4.0	8.0	5.4	6.0
Apj	**1.0**	6.6	2.0	4.0	3.0	8.0	6.4	5.0
Customer	-	6.0	3.1	**1.4**	2.4	7.0	5.0	3.1
Domino	**2.0**	6.0	3.0	2.6	3.4	8.0	7.0	4.0
Emea	**3.0**	**3.0**	**3.0**	6.0	**3.0**	8.0	7.0	**3.0**
Firewall 1	**1.0**	7.0	2.0	3.2	4.2	8.0	5.4	5.2
Firewall 2	**1.0**	6.0	3.2	3.2	2.6	8.0	7.0	5.0
Healthcare	**1.0**	7.0	2.4	2.6	5.0	8.0	6.0	4.0

Table 9: Average rank for the real-world datasets (WSC)

The details of the experiments on the datasets described in Table 1 using the `UPA` decompositions compute using the heuristics of Figure 12 can be found online in Section 2.10 of [6]. In Tables 8 and 9, we report the average rank of the role-set size and WSC for such experiments. For each dataset, we denote in boldface the smallest rank. From Table 8, we see that for four out of nine datasets, using the `Optimal` decomposition

as input to postPUCC gives rise to smallest role-set and the second best decomposition is $\texttt{SMAU}_R$. For other four out of nine datasets it is better use the $\texttt{SMAU}_R$ decomposition; while, for the dataset *Emea*, the two decompositions are equivalent. This is a positive finding as, in general, given a user-to-permission assignment matrix UPA, to compute the smallest role-set covering it is an NP-hard problem [28] (even the problem of computing the minimal role-set cannot be approximated within any constant factor in polynomial time unless $P = NP$, see [4]). So, it is not always feasible to compute an optimal decomposition of an UPA matrix and starting from $\texttt{SMAU}_R$ decomposition (computable in polynomial time), our heuristic generates solutions quite similar to the ones computed from an Optimal decomposition.
Considering the WSC measure, from Table 9, we see that for two datasets (i.e., *Americas Large* and *Customer*), the best solution is obtained using the decomposition $\texttt{SMA}_C$. For the dataset *Emea*, most decompositions allow to compute a solution with the lowest WSC and, as well, the smaller role-set. This is due to the structure of the dataset *Emea* all users are assigned a different subset of permissions and the heuristics Optimal, $\texttt{SMA}_R$, $\texttt{SMAU}_R$, $\texttt{SMAU}_C$, and Biclique return the same decomposition. For the remaining six datasets, using the Optimal decomposition allows to generate solutions with the lowest WSC value (recall that for the dataset *Customer*, an Optimal decomposition is not available). For these six datasets, the second best WSC value is attained when postPUCC starts from one of decompositions computed using the heuristics in [1]. Hence, we can conclude that for generic UPA matrices, although an optimal decomposition is not available, one can use UPA decompositions obtained from the heuristics in [1] without worsening much the parameters of the computed solutions.

In the remaining part of this section, to stress our heuristic, we execute some experiments setting $mrp = 2$. A large role-set returned by our heuristic is not a surprise at all. It depends on the structure of the UPA matrix. Several users have much more than two permissions, so we need many roles to cover all of them. For instance, users in the dataset *Emea* have assigned 3046 distinct permissions. So, independently of the decomposition we use as input of our heuristic postPUCC, when mpr = 2, we need at least 1523 different roles to cover them.

Dataset	Optimal	$\texttt{SMA}_R$	$\texttt{SMAU}_R$	$\texttt{SMA}_C$	$\texttt{SMAU}_C$	FastMiner	OBMD	Biclique	
Americas Large	11343	11524	10956	11322	11274	20237	12352	11294	$\lvert\mathcal{R}\rvert$
	124029	129149	121079	145979	123729	305323	177006	131942	WSC
	28.5	26.8	26.4	18.5	27.1	3.1	21.9	26.7	gf $\lvert\mathcal{R}\rvert$
	1.3	1.2	1.3	1.6	1.3	0.3	1.4	1.3	gf WSC
Firewall 2	325	457	297	445	457	470	457	395	$\lvert\mathcal{R}\rvert$
	19376	27510	19394	19832	19791	29582	27876	19492	WSC
	32.5	45.7	29.7	44.5	45.7	23.5	45.7	39.5	gf $\lvert\mathcal{R}\rvert$
	17.3	14.0	12.4	13.5	13.5	9.4	14.1	11.0	gf WSC

Table 10: Computed solution vs unconstrained solution for $mpr = 2$

Table 10, for the datasets *Americas Large* and *Firewall 2*, summarizes the *growing factor* (denoted by gf) of the role-set size and the WSC computed by our heuristic when $mpr = 2$. In particular, it reports report the role-set size and the WSC value of the decomposition computed by postPUCC and ratio between the role-set size (resp,. WSC) of the initial *unconstrained*decompositions and the role-set size (resp., WSC) of the computed ones. The results for the remaining datasets can be found online in Section 2.10 of [6]. For the *Americas Large* dataset, we note that, considering the role-set size, the growing factor for all decompositions, except FastMiner is between 18 and 28. This factor reduces to about 3 when postPUCC receives as input the FastMiner decomposition. Anyway, the role-set computed using the FastMiner decomposition is much bigger than the role-sets computed using the other decompositions. Hence, the growing factor measure cannot be used to compare heuristics' behaviour. For a fixed starting decomposition, it can only be applied to see how the constraint impacts on the number of generated roles with respect to an unconstrained scenario. Similar arguments apply to the dataset *Firewall 2*, too. Finally, notice that for the dataset *Americas Large* the number of roles generated from any starting decomposition is bigger than the number $|\mathcal{P}|$ of permissions of the UPA matrix (10127 according to the data in Table 1). Recall that, in the *PUCC* scenario, independently of the *mpr* value, there always exists a decomposition consisting of $|\mathcal{P}|$ roles (each role containing a single permission) satisfying the constraint. Hence, the solutions generated for the *Americas Large* dataset, when $mpr = 2$ are worse than the *naive* solution. Considering the results in Section 2.10 of [6], we see that this phenomenon also emerges for some of the other decompositions. Nevertheless, for the other eight datasets, when `postPUCC` receives as input the SMAU$_R$ decomposition, the resulting role-set size is smaller than $|\mathcal{P}|$ confirming that SMAU$_R$ is the best decomposition to use in the post-processing framework for the *PUCC* scenario.

RUCC Scenario. In the following, our heuristic `postRUCC` described in Section 4.2 is compared with state-of-the-art ones. More specifically, `postRUCC` is compared against the heuristic *Fix Role Usage Cardinality Constraint* (referred to as `FixRUC`, see Algorithm 1 in [12]) and the heuristics *Role Priority based Algorithm* (referred to as `RPA`, see Algorithm 1 in [14]) and *Coverage of Permissions based Algorithm* (referred to as `CPA`, see Algorithm 2 in [14]). Similarly to the *PUCC* scenario, the real-world datasets listed in Table 1 are used to compare heuristics. Heuristics `postRUCC`, `FixRUC`, `CPA`, and `RPA` were tested on the decompositions summarized in Figure 12 for the *mru* values given in Table 4.

mru	decomposition	$\lvert\mathcal{R}\rvert$				WSC			
		postRUCC	FixRUC	CPA	RPA	postRUCC	FixRUC	CPA	RPA
2	Optimal	279	298	294	294	15166	21460	15328	15328
	SMA$_R$	230	272	248	248	21519	26387	21695	21695
	SMAU$_R$	276	364	280	280	16224	21470	15983	15983
	SMA$_C$	278	324	295	295	16164	25152	15498	15498
	SMAU$_C$	258	277	272	272	16446	22892	16442	16442
	FastMiner	259	1789	259	259	25488	115653	25488	25488
	OBMD	263	334	294	294	15074	31014	14773	14773
	Biclique	293	341	301	301	20171	26287	20227	20227
6	Optimal	187	202	196	211	10933	12868	10949	11268
	SMA$_R$	225	235	225	225	21479	23229	21479	21481
	SMAU$_R$	270	275	275	283	9821	14350	9958	10487
	SMA$_C$	219	262	229	230	13232	19260	13226	13517
	SMAU$_C$	214	238	220	222	13908	18079	13981	14061
	FastMiner	259	1802	259	259	25488	123830	25488	25488
	OBMD	196	264	198	200	14448	24633	14452	14428
	Biclique	249	257	246	257	21120	22789	21091	21414
10	Optimal	178	181	179	178	10905	11213	10907	11033
	SMA$_R$	225	229	225	225	21479	22904	21479	21481
	SMAU$_R$	246	252	243	242	9631	12627	9584	9832
	SMA$_C$	205	234	205	205	13416	16888	13410	13362
	SMAU$_C$	198	222	198	198	13946	16262	13946	14003
	FastMiner	259	1811	259	259	25488	126416	25488	25488
	OBMD	196	236	196	195	14448	22041	14448	14409
	Biclique	220	232	218	228	21105	22071	21081	21354
12	Optimal	178	178	178	178	10905	11217	10905	11033
	SMA$_R$	225	227	225	225	21479	22875	21479	21481
	SMAU$_R$	233	235	236	235	9785	11404	9660	9829
	SMA$_C$	204	228	204	204	13420	16387	13420	13420
	SMAU$_C$	198	213	198	198	13946	16011	13946	14003
	FastMiner	259	1817	259	259	25488	127695	25488	25488
	OBMD	196	224	196	195	14448	21576	14448	14409
	Biclique	211	225	211	213	21155	21876	21152	21314
14	Optimal	178	178	178	178	10905	11217	10905	11033
	SMA$_R$	225	226	225	225	21479	22961	21479	21481
	SMAU$_R$	228	229	224	221	9917	11105	9838	9814
	SMA$_C$	204	220	204	204	13420	15809	13420	13420
	SMAU$_C$	198	209	198	198	13946	15725	13946	14003
	FastMiner	259	1828	259	259	25488	129429	25488	25488
	OBMD	196	214	196	195	14448	18010	14448	14409
	Biclique	211	220	208	212	21207	21923	21193	21361

Table 11: $|\mathcal{R}|$ and WSC for the dataset *Americas Small*

The experiments on the dataset *Americas Small* are reported in Table 11. According to the data in this table, for any fixed decomposition and in almost all experiments, the heuristic `postRUCC` returns a solution having a smaller role-set and a lower WSC value than the other state-of-the-art heuristics. Indeed, `postRUCC` generates a bigger role-set only in eight experiments out of forty. Since, from Table 11, it could be difficult to verify which combination of heuristic and variant is preferable over the others, the heuristics were ranked as described for the *PUCC* scenario. In particular, in Table 12, for any fixed *mru* value, the heuristics have been ranked considering the average rank over the *starting* decompositions. From Table 12, it results that `postRUCC` returns, on average, a smaller role-set in three cases out of five and that `FixRUC` is the worse heuristic. Considering the WSC measure, according to Table 12, heuristic `CPA` provides, on average, better results in four case out of five. Anyway, `postRUCC` is not that bad, as, according to Table 11, it generates solutions whose WSC is larger than the other heuristics in 15 cases out of 40 and in most of such cases the difference is negligible (the best WSC values are only few units apart from the ones computed by `postRUCC`).

mru	$\lvert\mathcal{R}\rvert$				WSC			
	postRUCC	FixRUC	CPA	RPA	postRUCC	FixRUC	CPA	RPA
2	1.12	4.0	2.44	2.44	2.12	4.0	1.94	1.94
6	1.38	3.62	1.94	3.06	1.56	4.0	1.81	2.62
10	2.12	4.0	2.06	1.81	1.94	4.0	1.69	2.38
12	1.94	3.62	2.31	2.12	1.88	4.0	1.62	2.5
14	2.25	3.81	2.0	1.94	2.0	4.0	1.75	2.25

Table 12: Ranking for the dataset *Americas Small*

In Table 13 the results of our experiments are ranked for any given decomposition. More precisely, the heuristics have been ranked considering the average rank over the *mru* values. In this way, one can see at a glance which *starting* decomposition allows to obtain a better solution. According to Table 13, heuristic `postRUCC`, for all decompositions except `OBMD`, compute on average the smallest role-set. For the WSC measure, the heuristic `CPA` returns solutions with smaller WSC in four cases (i.e., for the decompositions $\texttt{SMAU}_R$, $\texttt{SMA}_C$, $\texttt{SMAU}_C$, and `Biclique`), the heuristic `postRUCC` in two cases (i.e., `Optimal` and $\texttt{SMA}_R$), and the heuristic `RPA` in one case (i.e., `OBMD`). Using as input the decomposition `FastMiner`, heuristics `postRUCC`, `CPA`, and `RPA` return role-sets of equal size and the same WSC value. Heuristic `FixRUC` returns the worse solutions.

decomposition	$\mathcal{R}$				WSC			
	postRUCC	FixRUC	CPA	RPA	postRUCC	FixRUC	CPA	RPA
Optimal	1.7	3.2	2.5	2.6	1.2	4.0	1.9	2.9
$\texttt{SMA}_R$	1.8	4.0	2.1	2.1	1.4	4.0	1.7	2.9
$\texttt{SMAU}_R$	1.8	3.4	2.6	2.2	2.2	4.0	1.5	2.3
$\texttt{SMA}_C$	1.6	4.0	2.1	2.3	2.4	4.0	1.7	1.9
$\texttt{SMAU}_C$	1.6	4.0	2.1	2.3	1.7	4.0	1.6	2.7
FastMiner	2.0	4.0	2.0	2.0	2.0	4.0	2.0	2.0
OBMD	1.9	4.0	2.4	1.7	2.5	4.0	2.4	1.1
Biclique	1.7	3.9	1.4	3.0	1.8	4.0	1.3	2.9

Table 13: Ranking for the dataset *Americas Small*

The results of the experiments for the other datasets are available online in the supplemental material [6]. For each dataset and each heuristic, Table 14 summarizes the average ranking over all experiments (i.e., over all *mru* values given in Table 4 and all decomposition listed in Figure 12.

From Table 14 it results that, except for the datasets *Customer* and *Domino*, heuristic `postRUCC` returns, on average, a smaller role-set than the other heuristics. Considering the WSC measure, the heuristic `CPA` computes, in most cases, solutions having lower WSC values. A closer look to the online data [6] shows that heuristic `FixRUC` returns the worse solutions and the solutions computed by the remaining heuristics are quite similar.

Dataset	$\lvert\mathcal{R}\rvert$				WSC			
	postRUCC	FixRUC	CPA	RPA	postRUCC	FixRUC	CPA	RPA
Americas Large	1.67	3.23	2.52	2.58	2.13	3.63	1.81	2.44
Americas Small	1.76	3.81	2.15	2.27	1.90	4.00	1.76	2.34
Apj	1.89	3.00	2.51	2.60	1.99	3.15	2.30	2.56
Customer	2.29	3.11	2.17	2.43	2.72	3.14	1.74	2.40
Domino	2.33	3.10	2.29	2.29	2.30	3.18	2.26	2.26
Emea	2.30	3.06	2.30	2.34	2.30	3.06	2.26	2.38
Firewall 1	1.97	3.06	2.38	2.59	1.73	3.62	1.94	2.71
Firewall 2	1.98	3.64	2.18	2.18	2.02	3.50	2.17	2.31
Healthcare	1.94	3.39	2.33	2.35	1.80	3.78	2.14	2.28

Table 14: Average ranking for all datasets - *RUCC* Scenario

6 Conclusions

In the *post-processing* framework, constraints are evaluated after that a valid set of roles has been determined, so that the resulting roles do not violate the imposed restrictions. This constitutes a clear advantage, since the post processing phase can be executed on the results provided by any other role mining procedure.

In this work we have focused on two different kinds of cardinality constraints, namely *role-usage cardinality constraints* (RUCC) and *permission-usage cardinality constraints* (PUCC) and we have provided two heuristics. We have evaluated the behavior of the proposed heuristics by discussing their application to standard datasets and have compared the results to the ones returned by other procedures that have been previously presented in literature, registering an effective improvements in most of the cases. In the next future, we plan to extend the approaches to obtain some estimates of the distance from the optimal result as discussed in [24], and/or consider different kinds of approaches, derived from genetic programming and machine learning [25].

References

[1] C. Blundo and S. Cimato. A simple role mining algorithm. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 1958–1962, Sierre, Switzerland, March 22-26 2010. ACM, New York.

[2] C. Blundo and S. Cimato. Constrained role mining. In *Security and Trust Management - 8th International Workshop, STM 2012, Revised Selected Papers*, volume 7783 of *Lecture Notes in Computer Science*, pages 289–304, Pisa, Italy, September 13-14 2012. Springer.

[3] C. Blundo, S. Cimato, and L. Siniscalchi. PRUCC-RM: permission-role-usage cardinality constrained role mining. In *41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017*, pages 149–154, Volume 2, Turin, Italy, July 4-8 2017. IEEE Computer Society.

[4] C. Blundo, S. Cimato, and L. Siniscalchi. Managing constraints in role based access control. *IEEE Access*, 8:140497–140511, 2020.

[5] C. Blundo, S. Cimato, and L. Siniscalchi. Role Mining Heuristics for Permission-Role-Usage Cardinality Constraints. *The Computer Journal*, 02 2021. https://doi.org/10.1093/comjnl/bxaa186.

[6] C. Blundo, S. Cimato, and L. Siniscalchi. Supplemental material for: Heuristics for constrained role mining in the post-processing framework. `https://github.com/RoleMining/ConstrainedRM`, 2021. Accessed: May 6th, 2021.

[7] L. Chen and J. Crampton. Set covering problems in role-based access control. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, 2009. Proceedings*,

volume 5789 of *Lecture Notes in Computer Science*, pages 689–704, Saint-Malo, France, September 21-23 2009. Springer.

[8] A. Ene, W. G. Horne, N. Milosavljevic, P. Rao, R. Schreiber, and R. E. Tarjan. Fast exact and heuristic methods for role minimization problems. In *13th ACM Symposium on Access Control Models and Technologies, SACMAT 2008, Proceedings*, pages 1–10, Estes Park, CO, USA, June 11-13 2008. ACM.

[9] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transaction on Information System Security*, 4(3):224–274, 2001.

[10] M. Frank, D. A. Basin, and J. M. Buhmann. A class of probabilistic models for role engineering. In *ACM Conference on Computer and Communications Security*, pages 299–310. ACM, 2008.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[12] P. Harika, M. Nagajyothi, J. C. John, S. Sural, J. Vaidya, and V. Atluri. Meeting cardinality constraints in role mining. *IEEE Trans. Dependable Sec. Comput.*, 12(1):71–84, 2015.

[13] M. Hingankar and S. Sural. Towards role mining with restricted user-role assignment. In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*, pages 1–5, Chennai, India, 28 Feb.-3 March 2011. IEEE.

[14] J. C. John, S. Sural, V. Atluri, and J. Vaidya. Role mining under role-usage cardinality constraint. In *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012. Proceedings*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 150–161, Heraklion, Crete, Greece, June 4-6 2012. Springer.

[15] R. Kumar, S. Sural, and A. Gupta. Mining RBAC roles under cardinality constraint. In *Information Systems Security - 6th International Conference, ICISS 2010. Proceedings*, volume 6503 of *Lecture Notes in Computer Science*, pages 171–185, Gandhinagar, India, December, 17-19 2010. Springer.

[16] N. Li, I. Molloy, Q. Wang, E. Bertino, S. Calo, and J. Lobo. Role mining for engineering and optimizing role based access control systems. Technical report, Purdue University, Purdue University, 11 2007.

[17] H. Lu, Y. Hong, Y. Yang, L. Duan, and N. Badar. Towards user-oriented RBAC model. In *Data and Applications Security and Privacy XXVII - 27th Annual IFIP WG 11.3 Conference, DBSec 2013. Proceedings*, volume 7964 of *Lecture Notes in Computer Science*, pages 81–96, Newark, NJ, USA, July 15-17 2013. Springer.

[18] H. Lu, Y. Hong, Y. Yang, L. Duan, and N. Badar. Towards user-oriented RBAC model. *Journal of Computer Security*, 23(1):107–129, 2015.

[19] H. Lu, J. Vaidya, and V. Atluri. Optimal boolean matrix decomposition: Application to role engineering. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008.*, pages 297–306, Cancún, Mexico, April 7-12 2008. IEEE Computer Society.

[20] X. Ma, R. Li, H. Wang, and H. Li. Role mining based on permission cardinality constraint and user cardinality constraint. *Security and Communication Networks*, 8(13):2317–2328, 2015.

[21] B. Mitra, S. Sural, J. Vaidya, and V. Atluri. A survey of role mining. *ACM Comput. Surv.*, 48(4):50:1–50:37, Feb. 2016.

[22] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. B. Calo, and J. Lobo. Mining roles with semantic meanings. In *13th ACM Symposium on Access Control Models and Technologies, SACMAT, 2008, Proceedings*, pages 21–30, Estes Park, CO, USA, June 11-13 2008. ACM.

[23] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. B. Calo, and J. Lobo. Mining roles with multiple objectives. *ACM Trans. Inf. Syst. Secur.*, 13(4):36:1–36:35, 2010.

[24] I. Molloy, N. Li, T. Li, Z. Mao, Q. Wang, and J. Lobo. Evaluating role mining algorithms. In *14th ACM Symposium on Access Control Models and Technologies, SACMAT 2009, Proceedings*, pages 95–104, Stresa, Italy, June 3-5 2009. ACM.

[25] I. Saenko and I. V. Kotenko. Genetic algorithms for role mining problem. In *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2011*, pages 646–650, Ayia Napa, Cyprus, 9-11 February 2011. IEEE Computer Society.

[26] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb. 1996.

[27] R. S. Sandhu, D. F. Ferraiolo, and D. R. Kuhn. The NIST model for role-based access control: towards a unified standard. In *Fifth ACM Workshop on Role-Based Access Control, RBAC 2000*, pages 47–63, Berlin, Germany, July 26-27 2000. ACM.

[28] J. Vaidya, V. Atluri, and Q. Guo. The role mining problem: finding a minimal descriptive set of roles. In *12th ACM Symposium on Access Control Models and Technologies, SACMAT 2007, Proceedings*, pages 175–184, Sophia Antipolis, France, June 20-22 2007. ACM.

[29] J. Vaidya, V. Atluri, and Q. Guo. The role mining problem: A formal perspective. *ACM Trans. Inf. Syst. Secur.*, 13(3), 2010.

[30] J. Vaidya, V. Atluri, and J. Warner. Roleminer: mining roles using subset enumeration. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006*, pages 144–153, Alexandria, VA, USA, October 30 - November 3 2006. ACM.

[31] J. Vaidya, V. Atluri, J. Warner, and Q. Guo. Role engineering via prioritized subset enumeration. *IEEE Trans. Dependable Sec. Comput.*, 7(3):300–314, 2010.

[32] N. E. Young. Greedy Set-Cover Algorithms. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 886–889. Springer, 2016.

[33] D. Zhang, K. Ramamohanarao, and T. Ebringer. Role engineering using graph optimisation. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 139–144, Sophia Antipolis France, June 2007. ACM.

7 Illustrative Example for postPUCC

In this section we will provide an illustrative example of the execution of our heuristics postPUCC w.r.t. the matrices UPA (represented in Table 17). In the example below described, the heuristics are executed considering $mpr = 2$.

postPUCC is executed on input the matrices UA and PA represented in tables 16, 15 (these matrices are obtained computing $\texttt{SMAU}_R$); the algorithm proceeds as follows.

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	1
r_2	0	1	1	1	0
r_3	1	0	0	1	1
r_4	0	1	0	0	0
r_5	1	0	0	0	1

Table 15: Matrix PA

	r_1	r_2	r_3	r_4	r_5
u_1	1	0	0	0	0
u_2	0	0	1	1	0
u_3	0	0	0	1	1
u_4	0	1	0	0	0
u_5	0	0	1	0	0

Table 16: Matrix UA

	p_1	p_2	p_3	p_4	p_5
u_1	0	0	1	1	1
u_2	1	1	0	1	1
u_3	1	1	0	0	1
u_4	0	1	1	1	0
u_5	1	0	0	1	1

Table 17: Matrix UPA

The first step of the procedure is running extractInfo which returns data structures ARU, APR, and CR expressed in tables 18, 19, 20.

postPUCC proceeds then analysing each user. We now describe the steps of postPUCC for each of them.

1. User u_1 has just one role r_1 that has three permissions, but this role does not satisfy the constrain ($mpr = 2$). To substitute role r_1 postPUCC proceeds as follow. The procedure postPUCC concludes that there is not existing role in CR that could be assigned to u_1, therefore postPUCC creates two roles one with permissions p_3, p_4 and one with permission p_5 to assign to u_1. The matrices newUA and newPA given in output by update are described in tables 21, 22.

2. User u_2 has two roles r_3, r_4, the role r_4 is maintained since it has cardinality one while r_3 does not satisfy the constrain ($mpr = 2$). To substitute role r_3 postPUCC proceeds as follows. The procedure postPUCC concludes that there is not existing role in CR that could be assigned to u_2, therefore postPUCC creates two roles one with permissions p_1, p_4 and one with permission p_5 to assign to u_2 (which corresponds to role r_2 in newPA). The matrices newUA and newPA given in output by update are described in tables 23, 24.

3. User u_3 has roles r_4 (which corresponds to role r_3 in newPA) and r_5 which satisfy the constrain so they remain unchanged, the matrices newUA and newPA given in output by update are described in tables 25, 26.

4. User u_4 has just one role r_2 that does not satisfy the constrain ($mpr = 2$). To substitute role r_2 postPUCC proceeds as follow. The procedure postPUCC using CR concludes that r_4 (which corresponds to r_3 in newPA) could be assigned to u_4, then postPUCC creates another role with permissions p_3, p_4 to assign to u_4 (which corresponds to r_1 in newPA). The matrices newUA and newPA given in output by update are described in tables 27, 28.

5. User u_5 has just one role r_3 that has three permissions, but this role does not satisfy the constrain ($mpr = 2$). To substitute role r_3 `postPUCC` proceeds as follow. The procedure `postPUCC` concludes that there is not existing role in `CR` that could be assigned to u_3, therefore `postPUCC` creates another two roles one with permissions p_1, p_4 (which corresponds to r_4 in `newPA`) and one with permission p_5 to assign to u_5 (which corresponds to role r_2 in `newPA`). The matrices `newUA` and `newPA` given in output by `update` are described in tables 29, 30 and they are the final output of the algorithm.

r_1	p_3	p_4	p_5
r_2	p_2	p_3	p_4
r_3	p_1	p_4	p_5
r_4	p_2		
r_5	p_1	p_5	

Table 18: `APR`

u_1	r_1	
u_2	r_3	r_4
u_3	r_4	r_5
u_4	r_2	
u_5	r_3	

Table 19: `ARU`

r_2	r_4

Table 20: `CR`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1

Table 21: `newPA`

	r_1	r_2
u_1	1	1

Table 22: `newUA`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1
r_3	0	1	0	0	0
r_4	1	0	0	1	0

Table 23: `newPA`

	r_1	r_2	r_3	r_4
u_1	1	1	0	0
u_2	0	1	1	1

Table 24: `newUA`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1
r_3	0	1	0	0	0
r_4	1	0	0	1	0
r_5	1	0	0	0	1

Table 25: `newPA`

	r_1	r_2	r_3	r_4	r_5
u_1	1	1	0	0	0
u_2	0	1	1	1	0
u_3	0	0	1	0	1

Table 26: `newUA`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1
r_3	0	1	0	0	0
r_4	1	0	0	1	0
r_5	1	0	0	0	1

Table 27: `newPA`

	r_1	r_2	r_3	r_4	r_5
u_1	1	1	0	0	0
u_2	0	1	1	1	0
u_3	0	0	1	0	1
u_4	1	0	1	0	0

Table 28: `newUA`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1
r_3	0	1	0	0	0
r_4	1	0	0	1	0
r_5	1	0	0	0	1

Table 29: `newPA`

	r_1	r_2	r_3	r_4	r_5
u_1	1	1	0	0	0
u_2	0	1	1	1	0
u_3	0	0	1	0	1
u_4	1	0	1	0	0
u_5	0	1	0	1	0

Table 30: `newUA`

8 Illustrative Example for postRUCC

In this section we will provide an illustrative example of the execution of our heuristics postRUCC w.r.t. the matrices UPA (represented in Table 33). In the example below described, the heuristics are executed considering $mru = 2$.

postRUCC is executed on input the matrices UA and PA represented in tables 32 (these matrices are obtained computing $\texttt{SMAU}_R$), 31; the algorithm proceeds as follows.

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	0	0	1
r_2	0	0	1	1	0
r_3	1	0	0	0	0
r_4	0	1	0	0	0

Table 31: Matrix PA

	r_1	r_2	r_3	r_4
u_1	0	1	0	0
u_2	1	1	0	0
u_3	1	1	1	1
u_4	1	0	0	0
u_5	1	1	1	0

Table 32: Matrix UA

	p_1	p_2	p_3	p_4	p_5
u_1	0	0	1	1	0
u_2	0	0	1	1	1
u_3	1	1	1	1	1
u_4	0	0	0	0	1
u_5	1	0	1	1	1

Table 33: Matrix UPA

The first step of the procedure is running extractInfo which returns data structures ARU, APRexpressed in tables 34, 35.

postRUCC proceeds then analysing each user. We now describe the steps of postRUCC for each of them.

1. User u_1 has associate roles r_2 which is also identified by approxCover as the existing role in APR that covers more (all) permissions of u_1, therefore r_2 is associated to u_1 and no further actions are required. The updated matrices newUA and newPA are described in tables 36, 37.

2. User u_2 has associate roles r_1, r_2 which is also identified by approxCover as the existing role in APR that covers more (all) permissions of u_2, therefore r_3 is associated to u_2 and no further actions are required. The updated matrices newUA and newPA are described in tables 38, 39.

3. User u_3 has associate roles r_1, r_2, r_3, r_4 which violates the constrain since $mru = 2$. The first step of postRUCC is to invoke the sub-procedure approxCover which returns the role r_1 in APR. The remaining permissions of u_3, namely $\{p_1, p_2, p_5\}$, are forming a new role that is added in APR (the update value of APR is described in table 42). The matrices newUA and newPA given in output by update are described in tables 40, 41.

4. User u_4 has associate roles r_1 which is also identified by approxCover as the existing role in APR that covers more (all) permissions of u_4, therefore r_1 is associated to u_4 and no further actions are required. The updated matrices newUA and newPA are described in tables 43, 44.

5. User u_5 has three roles r_1, r_2, r_3 which violates the constrain since $mru = 2$. The first step of postRUCC is to invoke the sub-procedure approxCover which returns the role r_1 in APR. The remaining permissions of u_5, namely $\{p_1, p_5\}$, are forming a new role that is added in APR (the update value of APR is described in table 47). The matrices newUA and newPA given in output by update are described in tables 45, 46 and they are the final output of the algorithm.

r_1	p_5		
r_2	p_3	p_4	
r_3	p_1		
r_4	p_2		

Table 34: `APR`

u_1	r_1			
u_2	r_1	r_2		
u_3	r_1	r_2	r_3	r_4
u_4	r_1	r_2	r_3	

Table 35: `ARU`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0

Table 36: `newPA`

	r_1
u_1	1

Table 37: `newUA`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1

Table 38: `newPA`

	r_1	r_2
u_1	1	0
u_2	1	1

Table 39: `newUA`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1
r_3	1	1	0	0	1

Table 40: `newPA`

	r_1	r_2	r_3
u_1	1	0	0
u_2	1	1	0
u_3	0	1	1

Table 41: `newUA`

r_1	p_5		
r_2	p_3	p_4	
r_3	p_1		
r_4	p_2		
r_5	p_1	p_2	p_5

Table 42: `APR`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1
r_3	1	1	0	0	1

Table 43: `newPA`

	r_1	r_2	r_3
u_1	1	0	0
u_2	1	1	0
u_3	0	1	1
u_4	0	1	0

Table 44: `newUA`

	p_1	p_2	p_3	p_4	p_5
r_1	0	0	1	1	0
r_2	0	0	0	0	1
r_3	1	1	0	0	1
r_4	1	0	0	0	1

Table 45: `newPA`

	r_1	r_2	r_3	r_4
u_1	1	0	0	0
u_2	1	1	0	0
u_3	0	1	1	0
u_4	0	1	0	0
u_5	1	0	0	1

Table 46: `newUA`

r_1	p_5		
r_2	p_3	p_4	
r_3	p_1		
r_4	p_2		
r_5	p_1	p_2	p_5
r_6	p_1	p_5	

Table 47: `APR`