IAM Role Diet
A Scalable Approach to Detecting RBAC Data Inefficiencies

Moratore, Roberto; Barbaro, Eduardo; Zhauniarovich, Yury

DOI
10.1109/DSN-S65789.2025.00052

Licence
Dutch Copyright Act (Article 25fa)

Publication date
2025

Document Version
Final published version

Published in
Proceedings - 2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume, DSN-S 2025

2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S) | 979-8-3315-1203-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/DSN-S65789.2025.00052

IAM Role Diet: A Scalable Approach to Detecting RBAC Data Inefficiencies

Roberto Moratore
ING Bank

Eduardo Barbaro
ING Bank & TU Delft

Yury Zhauniarovich
TU Delft

***Abstract*—More than three decades after its introduction, Role-Based Access Control (RBAC) continues to be one of the most widely used access control models in organizations. This popularity stems from its simplicity, the reduced risk of errors, and its clear alignment with business processes. However, the primarily manual nature of data management in RBAC systems, coupled with a lack of oversight, can lead to various inefficiencies over time. These may include roles that are not assigned to any users or roles that have identical sets of permissions. Such issues can slow down systems that rely on these data and, more critically, complicate auditing processes, increasing the risk of security gaps and compliance violations.**

In this paper, we present a taxonomy of inefficiencies that can arise in RBAC data over time and propose a framework for detecting these inefficiencies. We specifically focus on the most resource-intensive inefficiencies, namely roles that share the same or similar users or permissions. To address these issues, we propose three detection methods, including a custom algorithm we developed. We evaluate these methods using synthetic datasets, demonstrating that our algorithm significantly outperforms baseline approaches. Its efficiency allows us to identify these inefficiencies even on a standard laptop used by large organizations. Furthermore, we applied our framework to real RBAC data from a large organization with over 60,000 employees and uncovered a substantial number of inefficiencies, highlighting its practical value in real-world scenarios.

***Index Terms*—RBAC, inefficiencies, detection.**

I. Introduction

Role-Based Access Control (RBAC) is a method of access control that restricts system access to authorized users based on their roles within an organization. In RBAC, permissions (or entitlements) are assigned to roles rather than individual users. Users are then assigned roles based on their job responsibilities or other criteria. This allows for more efficient access control management since permissions can be assigned to roles rather than to each user. First introduced by Ferraiolo and Kuhn [1], the RBAC model has become widely adopted in various systems due to its simplicity, reduced risk of errors, and clear connection with the business processes [2], [3]. After three decades, the market of RBAC systems is considerable and continues to grow [4]. Moreover, researchers continue improving the model and propose new extensions [5], [6].

Although RBAC considerably simplifies access control management, it still does not remove all concerns related to this process. Particularly, developed policies may contain various issues, such as redundant roles or contradicting permissions [7]. Researchers have been consistently proposing methods to fix these issues [8] and extensions, such as Administrative Role-Based Access Control (ARBAC) [9], [10], albeit with limited industry application. In addition, several approaches have been developed to eliminate them by suggesting what roles should be created [11] given different objective functions, e.g., minimal number of roles [12], similar permissions in groups [13], or semantically meaningful roles [14].

While all these approaches are available, most organizations still manage their access policies using simpler out-of-the-box RBAC commercial tooling [10]. However, those access policies tend to be siloed per department, causing several inefficiencies, such as repeated roles having the same permissions, e.g. in different departments or countries. That is especially important for global enterprises, as those inefficiencies slow down their operations because authorization checks persist throughout the year. In addition, in practice, most RBAC commercial platforms tend to have their performance hampered by these repeated, empty, or ill-defined roles, making the management and – critically – auditing those roles a complex and prone-to-error process. Therefore, it is highly beneficial for enterprises to minimize such inefficiencies.

Modern large organizations employ tens of thousands of people, likely having several hundred complex departments worldwide, each typically with their structure and set of roles, making identifying RBAC inefficiencies manually a nearly impossible task [6], [15]. For instance, according to [16], Dresdner Bank has over 50 thousand employees with over 1,300 roles created manually. The resulting access policy might contain millions of entries, making its management inefficient.

In this paper, we identify five common types of inefficiencies observed in RBAC data – (i) standalone nodes, (ii) roles that are not connected to users or permissions, (iii) roles connected to only a single user or permission, (iv) roles sharing the same users or permissions, and (v) roles sharing similar users or permissions – and propose a framework for their detection. We test three distinct clustering approaches – exact, approximate, and our algorithm – to address the issue identified in (iv) and (v), where roles have the same or similar users or permissions, as those are the most time-consuming inefficiencies. We evaluate these approaches on synthetic data, and the results show that our custom algorithm significantly outperforms baseline methods that rely on exact or approximate clustering techniques. For example, our algorithm takes only 2.27 seconds to detect roles sharing the same users in a dataset with 10,000 roles and 1,000 users. In contrast, the same task requires 496.41 seconds with exact clustering and 327.85 seconds with approximate clustering.

We also applied our framework to a real-life dataset from a large international organization with over 60,000 employees. The framework detected a large number of inefficiencies. For instance, it showed that only by consolidating the roles sharing the same users or permissions, it is possible to remove about 10% of all roles.

Thus, we make the following contributions in this paper:

- Identified the most common inefficiencies in RBAC data, proposing a taxonomy of their types;
- Developed a framework to detect all identified types of inefficiencies;
- Proposed three different methods, including our own algorithm, to identify the most resources-consuming inefficiency types – roles sharing same or similar users or permissions – and showed significant time savings offered by our algorithm;
- Applied the framework to a real RBAC dataset, demonstrating its viability in real-world scenarios.

II. Related Work

Reducing or ideally eliminating unnecessary privileges, i.e. the least privilege principle, is the primary goal of RBAC [7], [13]. In recent years, extensive research has been done on addressing the issue of policy redundancies and inconsistencies [6], [8], [9], [16], [17]. Stoller et.al. [9] showed that understanding a system's policies and their interactions is critical to its security. Despite its importance and the fact that the individual rules are – in isolation – simple, it is often impossible to capture all their relations and interactions. To solve that, they propose creating classes of policies instead of analyzing them individually. They also demonstrate that role bloat remains challenging even analytically, typically yielding complex algorithms [13], [14]. Interestingly, [14] proposed using formal concept analysis to find meaningful roles. They discouraged utilizing any type of clustering technique, e.g., bi-clustering or co-clustering. To mine roles, i.e., find roles from existing permissions, they developed a hierarchical miner achieving good results, albeit on artificial datasets. In contrast, Vaidya et.al. [13] proposed an unsupervised approach (role miner) that mines roles from existing user-permission assignments using clustering techniques. The key difference from traditional clustering methods is that roles have overlapping permissions, making it a unique problem. Here, we are not proposing a top-down or a bottom-up algorithm to create new roles, but rather a simple algorithm that combines existing roles (without granting extra permissions), yielding an exact solution that can be scaled to real-world enterprises with hundreds of thousands of employees. Li and Tripunitara [15] argued that the only way to maintain a large and complex system of permissions and users is to combine security analysis techniques to uphold basic properties with the decentralization and delegation of privileges.

Focusing on reducing permissions, D'Antoni et. al. [18] discussed the concept of automatic least-permissive RBAC, focusing on policies and access logs. They showed that refining current policies is better (or at least as effective) than generating new ones from scratch. We utilize a similar concept applied for roles and claim it is possible to obtain a much-reduced set of roles and permissions starting from an initial (potentially incorrect) set of roles/permissions. In a similar fashion, Huang et.al. [8] proposed an algorithm to check for redundancies and inconsistencies in roles based on a given policy definition. While a successful approach, the algorithm's scalability remained a challenge, and its use in real-world large-scale corporations requires further investigation.

Jayaraman et.al. [10] explored one of the most common – but potentially catastrophic – side effects of role bloat, which happens when an incorrect policy grants users privileges they should not have. They showed that an effective access-control policy should be accomplished by various techniques, where more traditional verification is combined with automatic access-control techniques. In our research, we aim to reduce the role bloat by using analytics to combine roles effectively, thus minimizing the risk of incorrect policy assignment as fewer roles are available.

Parkinson and Khan [7] highlighted several interesting points in their survey study, most notably that synthetically generated datasets might not represent real-world conditions both in terms of size and complexity. Oftentimes, "issues" are artificially introduced a-posteriori following a pre-determined pattern. Therefore, in our work, we also utilize a real-world dataset from a large international organization with over 60,000 employees.

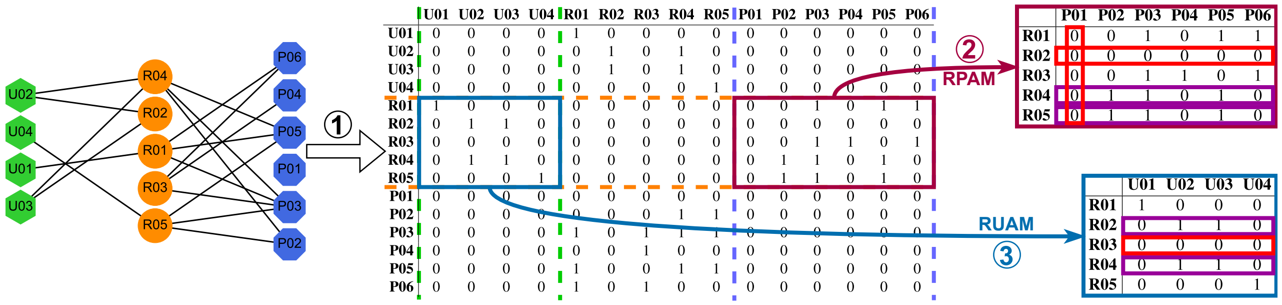
III. Our Approach

In its simplest form, the RBAC data can be represented as a *tripartite graph*. The network in the left part of Figure 1 shows an example of such graph. There are three sets of nodes that correspond to *users* (marked as U01...U04), *roles* (R01...R05), and *permissions* (P01...P06). The edges in this graph represent the assignment of users to roles and permissions to roles.

A. Identified Inefficiencies

Our analysis identified *five* groups of inefficiencies that may appear in the RBAC data due to poor management (we show examples of each case in Figure 1):

1) **Standalone nodes.** The stand-alone nodes appear due to the removal of the corresponding edges. For instance, a user who is not longer working in an organization should not be assigned to any role, and the corresponding entry in the system should be removed. The P01 permission is an example of such a node.
2) **Roles not connected to users/permissions.** The RBAC data may contain roles not connected to users/permissions. So, as these roles do not have any links with permissions/users, they and the corresponding edges to users/permissions can probably be removed. For instance,



	U01	U02	U03	U04	R01	R02	R03	R04	R05	P01	P02	P03	P04	P05	P06
U01	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
U02	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
U03	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
U04	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
R01	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1
R02	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
R03	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
R04	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0
R05	0	0	0	1	0	0	0	0	0	0	1	1	0	1	0
P01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P02	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
P03	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0
P04	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
P05	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0
P06	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

	P01	P02	P03	P04	P05	P06
R01	0	0	1	0	1	1
R02	0	0	0	0	0	0
R03	0	0	1	1	0	1
R04	0	1	1	0	1	0
R05	0	1	1	0	1	0

	U01	U02	U03	U04
R01	1	0	0	0
R02	0	1	1	0
R03	0	0	0	0
R04	0	1	1	0
R05	0	0	0	1

Fig. 1: Approach High-Level Overview (Users are Marked as U01...U04, Roles as R01...R05, and Permissions P1...P6. Edges Represent Assignments of Users to Roles and Permissions to Roles)

role R02 is not connected to any permission node, and role R03 is not linked to any user node.

3) **Roles connected to a single user/permission.** While this might be a legitimate case, it is likely a sign of inefficiency. For instance, the R01 and R05 roles have a single user assigned.
4) **Roles sharing the same users/permissions.** Roles that share the same set of users/permissions are probably redundant and could be combined. For instance, roles R04 and R05, sharing the same set of permissions, might be alike, as well as roles R02 and R04, connected to identical users.
5) **Roles sharing a similar set of users/permissions.** Similarly to the previous case, it may be worth considering roles that share a similar set of users/permissions. For instance, an administrator might decide to consider and merge the roles that have all but one identical permissions. Note that the approach should be generalizable over an arbitrary number of differences set by an administrator.

These inefficiencies must not be fixed automatically as they may correspond to legitimate corner cases. For instance, a role might be connected only to a single user if it is assigned to the CEO. Therefore, the administrator must carefully consider and approve every instance of inefficiency detected.

B. *Idea Description*

To find the instances of the inefficiencies described in Section III-A, we represent (Step 1 in Figure 1) our tripartite graph as an *adjacency matrix*. An adjacency matrix is a square matrix where the rows and columns of the matrix correspond to the vertices of the graph, and the entries of the matrix indicate whether there is an edge between two vertices. For instance, if an edge exists between vertices i and j, the (i, j) entry is 1. Otherwise, the entry is 0.

Due to the properties of the tripartite graph, i.e., edges can exist only between different sets of nodes, storing the full adjacency matrix is not required. Instead, we can rebuild the whole matrix using two sub-matrices: *Role-Permission Assignment Matrix (RPAM)* and *Role-User Assignment Matrix (RUAM)*. RPAM, highlighted with a brown box, is built by selecting the rows matching role nodes and columns corresponding to permission nodes (see Step 2 in Figure 1). RUAM, highlighted with a blue box, is created by selecting the rows related to role nodes and columns matching to user nodes (see Step 3 in Figure 1). Thus, if the total number of roles is r, permissions p and users u, we will need $r * (p + u)$ instead of $(r + p + u)^2$ memory space to store the data. Note that it is possible to further optimize required memory space by using a sparse matrix representation; however, the type of sparse matrix should be chosen considering other factors, such as conversion time, based on the experimental evaluation.

Standalone nodes. As it is clear from the definition, if a node is not connected to any of the rest, then all values in the row or column corresponding to it will have 0's. Since there are no edges between the permission and user nodes, then to identify standalone permission and user nodes, it is enough to find the columns containing all 0's in RPAM and RUAM correspondingly. Thus, we can sum all the values in the RPAM and RUAM columns and find the ones whose sum is equal to 0. For instance, in (see Figure 1), the RPAM column corresponding to the P01 node (outlined with a red vertical rectangle) exemplifies this case, showing that the corresponding permission is not connected to any role. At the same time, identifying standalone role nodes is trickier. To achieve this goal, we need to sum all values in both RPAM and RUAM matrices and find the row in both matrices corresponding to the same role node where both sums are equal to 0.

Roles connected only to users/permissions. During the previous step, we calculated the sum of row values in RPAM and RUAM. It is possible to re-use the data obtained during this step to identify the roles not connected to users/permissions – the rows with 0-sum in RPAM and RUAM will correspond to roles not connected to any permission or user, correspondingly (see rows in RPAM and RUAM matrices, outlined with red horizontal rectangular boxes).

Roles connected to a single user/permission. The same sum values can also be used to identify the roles connected to a single user/permission. In this case, though, we will need to find the rows where the sum is equal to 1. For instance, in RUAM, the rows for the R01 and R05 roles fall into this category.

Roles sharing the same users/permissions. It is clear that roles that share the same users/permissions have the same values in the corresponding cells, i.e., the vectors (row values) corresponding to the roles sharing the same users or permissions should be equal in RUAM or RPAM matrices, correspondingly. For instance, in Figure 1, the roles R04 and R05 rows are equal in RPAM (outlined with a pink horizontal rectangle), as the R02 and R04 roles rows in RUAM. A naïve approach would be to check all pairs of role vectors in RUAM and RPAM matrices and check if they are equal. However, this approach is largely inefficient and does not scale. Also, generally, there could be multiple roles with the same vector. Therefore, we need to find efficient approaches that allow one to find all groups of the same vectors.

Roles sharing a similar set of users/permission. If two roles share a similar set of users (or permissions), the corresponding row vectors in the RUAM (or RPAM) will have identical values of 1 in the columns associated with those users (or permissions). At the same time, they will have different values in the columns corresponding to distinctive users (or permissions). The number of columns with different values will correspond to the number of distinct users (or permissions). Thus, we need to find an efficient approach that allows one to find all groups of similar vectors.

C. Approaches to Find Role Groups

Among all these inefficiencies, the most complex ones to detect, in terms of computational complexity, are those related to finding roles sharing the same or similar permissions/users. All the rest of the inefficiencies can be found in linear time. In this section, we propose three different approaches to finding role groups: *exact spatial clustering*, *approximate nearest neighbors*, and *our custom algorithm*. In the rest of the paper, we consider only one matrix – RUAM – for finding groups of roles sharing the same or similar users. However, the exact same approach can be used to find groups of roles sharing the same or similar permissions by feeding RPAM instead of RUAM into them.

Exact Clustering. From the description of the problem, it is apparent that we can apply a clustering algorithm to identify groups of roles. Indeed, each role vector can be represented as a point in a multidimensional space. Therefore, to find the groups of roles, we need to find the clusters of these points in space. We can apply spatial clustering approaches to achieve this goal and find the groups of roles sharing the same or similar users by considering the points in the identified clusters.

For this study, we choose the *Density-Based Spatial Clustering of Applications with Noise (DBSCAN)* algorithm for spatial clustering. DBSCAN [19] is a popular exact clustering algorithm that groups points that are closely packed together, marking as outliers points that lie alone in low-density regions. It is particularly useful for identifying clusters of arbitrary shape and handling noise in the data. What is more important in our case: it does not require specifying the number of clusters.

This clustering approach requires one to specify the *minimum number of points in a cluster*, what *distance function* to use, and what is the *maximum distance between two samples*. It is clear that in our case, the minimum number of points in a cluster is equal to 2, because we want to find even two akin roles. As for the other two parameters, we should consider the cases of roles sharing the *same* and *similar* users separately. In the case of roles sharing the *same* users, the corresponding points have the same coordinates, i.e., they coincide. According to the definition of the distance function, a distance between them must be equal to 0 regardless of a chosen metric. Thus, we can use any distance metric with the maximum distance between points set to 0. At the same time, in the case of a similar set of users, we cannot use any metric and have to choose the right one for our purposes. The number of different users corresponds to the number of columns, where the values in the corresponding rows are distinct. So, as all values in a matrix are 0's and 1's (bit-size values), we can use the *Hamming distance* in this case. Thus, to find roles that share the same but one set of permissions, we need to find all pairs of vectors, the Hamming distance between which equals 1.

Approximate Clustering. The performance of exact spatial clustering methods like DBSCAN can be slow on large, high-dimensional datasets [20], [21]. Unfortunately, our case belongs exactly to this category. To address this issue, we consider employing approximate clustering approaches that trade-off recall for increased speed. Note that, given that the task of cleaning the RBAC database is expected to run periodically, not being able to identify all roles in a group does not hurt, as they will be identified during the next run. In this work, we employ the approximate nearest neighbor search approach described in the paper by Malkov and Yashunin [22].

Our Algorithm. Our custom algorithm leverages the specific properties of the RBAC data to efficiently identify clusters of roles sharing the same or similar (same users $\pm$ manually set threshold) users.

Let R^i be a role, and $|R^i|$ is its norm, i.e. the number of users assigned to it. Define the function:

$$g(R^i, R^j) = g^{ij} : \mathbb{R} \Rightarrow \mathbb{N}$$

which is the number of user co-occurrences between roles R^i and R^j. For our RUAM, the co-occurrences matrix C looks as such:

	R01	**R02**	**R03**	**R04**	**R05**
R01	1	0	0	0	0
R02	0	2	0	2	0
R03	0	0	0	0	0
R04	0	2	0	2	0
R05	0	0	0	0	1

Where we define the elements of the matrix as:

$$C^{ij} \;=\; \begin{cases} g^{ij}, i \neq j \\ |R^i|, i = j \end{cases}$$

For our purposes, we define that roles R^i and R^j can be combined if they contain the same users. For this, we define an indicator function:

$$\mathbb{I}^{ij} := \begin{cases} 1 & \text{iff } |R^i| = g^{ij} = |R^j|,\ i \neq j \\ 0 & \text{otherwise} \end{cases}$$

Where the groups, we are interested in, are:

$$\mathbb{I}^{ij} = 1\ \forall\ i, j$$

D. Implementation

We implemented all the approaches for identifying role groups, as described in Section III-C. To promote transparency and reproducibility, we provide open access[1] to our notebook, which includes implementation and detailed documentation of the developed code. These resources enable researchers and practitioners to validate our methods, replicate our experiments, and build upon our work.

For the DBSCAN approach, we employed the implementation from the *scikit-learn* [23] library. We set all its parameters as described in Section III-C; however, we add a small ϵ value to the maximum distance parameter to account for floating-point comparison inaccuracies. The scikit-learn DBSCAN `fit_predict` method returns a vector of integer labels, with a size equal to the number of roles. The indices in this vector, where the labels are equal, determine which roles belong to the same group (the -1 special value is reserved for noise cases). Thus, iterating over this vector, we can list all the groups of roles.

For approximate clustering, we employed the Hierarchical Navigable Small World (HNSW) algorithm implemented in the *datasketch* [24] library. HNSW is among the fastest and most accurate algorithms for approximate nearest neighbor (ANN) search, which is currently widely used in many vector databases. Using this library, we constructed an `HNSW` index with the RUAM data, relying on default parameter settings and using Manhattan distance as the distance metric. Then, we queried the index to identify roles similar to the provided one. It is worth noting that more efficient implementations of HNSW and other ANN algorithms in general are available and could be used to further optimize performance, e.g., implemented in native code. However, our focus in this work is to illustrate the general trend.

Finally, for our custom algorithm, we employ standard data science libraries, namely *pandas* [25] and *numpy* [26].

IV. Evaluation

From the description of our framework for detecting the inefficiencies (see Section III), it is evident that the most time-consuming type of inefficiency to detect is identifying roles with the same or similar sets of users or permissions. Therefore, in this section, we concentrate on evaluating the efficiency of the three proposed approaches for this task. Notably, the process of finding roles that share the same or similar users is equivalent for the case of permissions, as both cases utilize the same detection method. The only difference lies in the input: for the former, we use RUAM, while for the latter, we use RPAM.

We executed all our measurements on an Apple Macbook Pro with a M1 Pro chip, with 32GB of integrated memory, on macOS Sequoia 15.3.2, and Python 3.12.6.

A. Execution Time

To perform an execution time evaluation, we developed a generator function that creates a matrix resembling RUAM/RPAM with predefined properties. Specifically, the generator depends on several key parameters, including the number of roles (rows in the matrix), the number of users (columns in the matrix), the proportion of the number of roles in clusters relative to the total number of roles, and the maximum number of identical roles within a cluster.

We ran each experiment five times, recording the execution duration, and calculated the average and standard deviation of the measured variable. We fixed the proportion of the number of roles in clusters relative to the total number of roles to 0.2 and the maximum number of identical roles within a cluster to 10. For DBSCAN and HNSW clustering algorithms, we used the default parameter values besides the ones described in Section III.

During the first experiment, we fixed the number of roles to 1,000 (number of rows in the matrix) and varied the number of users between 1,000 and 10,000 (number of columns in the matrix). The experiment results can be found in Figure 2. We observe that the time required to run the clustering process is nearly constant. The approximate clustering approach takes the longest time to complete the task, with the duration increasing from approximately 16.36 seconds for 1,000 users to 23.07 seconds for 10,000 users. Building the index requires significant time, and this method does not demonstrate its advantages with a smaller number of entries, such as 1,000 rows. In contrast, the exact clustering approach is considerably faster, taking around 0.47 seconds for 10,000 users. Our algorithm, however, exhibits the shortest processing time, detecting roles shared by the same users in about 0.13 seconds.

In the second experiment, we fixed the number of users at 1,000 (i.e., the number of columns), changing the number of roles from 1,000 to 10,000 (the number of rows in the matrix), as shown in Figure 3.

Several conclusions can be drawn from this experiment. First, for all approaches, execution time increases as the number of roles grows. Therefore, reducing the number of roles should be a primary optimization goal. Second, the execution time for exact clustering increases at a faster rate than for approximate clustering. In experiments with fewer entries, a significant portion of the execution time for approximate clustering is spent on building the index, which allows the exact clustering approach to perform better initially. However, at around 7,000 rows, approximate clustering begins to outperform exact clustering in terms of execution time. It is important to note that approximate clustering may miss

[1]https://github.com/neo2478/rbac_reduction_paper

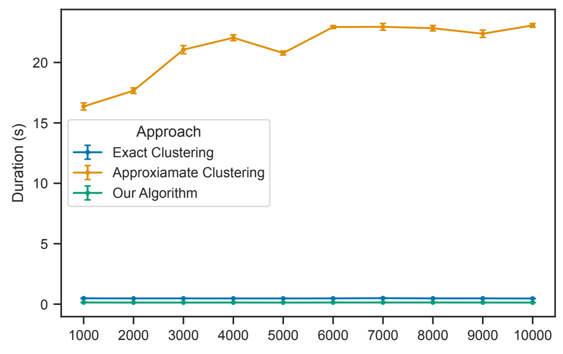


Fig. 2: Duration of the Analysis Depending on User Number (Number of Roles is Equal to 1000)

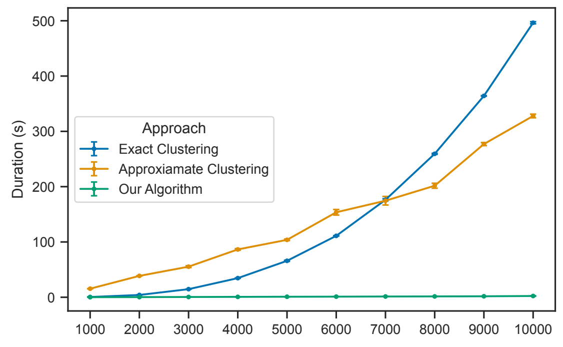


Fig. 3: Duration of the Analysis Depending on Role Number (Number of Users is Equal to 1000)

some entries within clusters. Nevertheless, we assume that the algorithm can be run periodically, enabling the results to converge gradually to the optimal solution over time. Lastly, our approach demonstrates superior performance compared to the other two methods. Specifically, the exact clustering approach takes approximately 0.47 seconds to process $1{,}000$ entries and around 496.41 seconds for $10{,}000$ entries. For approximate clustering, execution time ranges from 15.48 seconds for $1{,}000$ entries to 327.85 seconds for $10{,}000$ entries. In contrast, our algorithm completes the task in just 0.13 seconds for $1{,}000$ rows and 2.27 seconds for $10{,}000$ rows, *demonstrating significant performance gains over the baseline approaches*. Moreover, our algorithm is entirely deterministic, which means it consistently identifies all clusters without fail.

B. *Inefficiencies Detection in the Real Dataset*

We applied our framework for detecting inefficiencies to a real dataset from a large organization that employs more than $60{,}000$ employees. To maintain anonymity, we do not disclose exact figures but, instead, present results focused on a very similar order of magnitude. It is important to note that the same roles can be linked to multiple types of inefficiencies.

The analyzed version of the dataset, before any optimization, consists of approximately $90{,}000$ users and $350{,}000$ permissions assigned to around $50{,}000$ roles. Due to its large size, this dataset is an excellent test case for assessing the efficiency and scalability of our approach. We also tried applying both methods to these data, but their execution was halted after 24 hours, which is impractical for the real world. In contrast, our method was able to process the real data in just 2 minutes.

First, we identified standalone nodes, usually consisting of new users and permissions, or permissions linked to decommissioned assets that had not been cleaned up. We detected 500 users who were not assigned to any role. Even more strikingly, we found that nearly half of all permissions – approximately $180{,}000$ – were not linked to any role.

Next, we used the framework to identify disconnected roles, which included newly created roles or those associated with decommissioned assets. We discovered $12{,}000$ roles that had no users assigned to them; these roles were linked solely to permissions. Additionally, we found $1{,}000$ roles that were assigned only to users without any associated permissions.

For the third type of inefficiency – which consists of roles connected to a single user or a single permission – we identified approximately $4{,}000$ roles assigned to only one user and around $21{,}000$ roles linked to a single permission. This latter figure indicates a misuse of the role structures and highlights a significant opportunity for role consolidation.

While the approach for consolidating roles related to the previous inefficiency still needs to be developed, the fourth type of inefficiency – roles sharing the same users or permissions – is addressed here. This issue arises from a fragmented landscape of independent role owners and managers. We identified $8{,}000$ roles sharing the same users and $2{,}000$ roles sharing the same permissions. Even if each cluster contains only two roles, these findings suggest that the total number of roles can be reduced by approximately $4{,}000$ and $1{,}000$ respectively, accounting for about 10% of all roles.

Finally, we ran the framework to identify roles that share all but one user or permission. We detected $6{,}000$ roles that share the same users, except for one, and $4{,}000$ roles that share the same permissions, except for one.

The data indicate a significant opportunity to reduce inefficiencies, which could result in notable enhancements to role-based access control data management.

V. Conclusion

In this work, we examined the most common recurring issues in RBAC data, which appear due to the predominantly manual nature of data management and lack of oversight. We proposed a taxonomy of inefficiencies along with a framework for their detection. We introduced three detection methods for the most resource-consuming inefficiency types, including our custom algorithm, which outperformed baseline approaches on several synthetic datasets of increasing complexity. Finally, we also tested our framework on a real dataset from a large organization, which outperformed the baseline approaches, identified many inefficiencies and demonstrated its practical value in real-world applications.

REFERENCES

[1] D. Ferraiolo and R. Kuhn, “Role-based access controls,” in *National Computer Security Conference*, ser. NCSC, 1992, pp. 554–563. [Online]. Available: https://csrc.nist.gov/pubs/conference/1992/10/13/rolebased-access-controls/final

[2] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, “Role-based access control models,” *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[3] C. Dilmegani and M. Palazoğlu, “6 real-life RBAC examples in 2025,” AI Multiple, January 2025. [Online]. Available: https://research.aimultiple.com/rbac-examples/

[4] MarketsAndMarkets, “Role-based access control market,” February 2023. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/role-based-access-control-market-46615680.html

[5] N. Kashmar, M. Adda, and M. Atieh, “From access control models to access control metamodels: A survey,” in *Advances in Information and Communication*, K. Arai and R. Bhatia, Eds. Cham: Springer International Publishing, 2020, pp. 892–911.

[6] M. P. Singh, S. Sural, J. Vaidya, and V. Atluri, “A role-based administrative model for administration of heterogeneous access control policies and its security analysis,” *Information Systems Frontiers*, 2021.

[7] S. Parkinson and S. Khan, “A survey on empirical security analysis of access-control systems: A real-world perspective,” *ACM Computing Surveys*, vol. 55, no. 6, Dec. 2022.

[8] C. Huang, J. Sun, X. Wang, and Y. Si, “Security policy management for systems employing role based access control model,” *Information Technology Journal*, vol. 8, no. 5, pp. 726–734, Jun. 2009.

[9] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman, “Efficient policy analysis for administrative role based access control,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS ’07, 2007, p. 445–455.

[10] K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin, “Automatic error finding in access-control policies,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS ’11. Association for Computing Machinery, 2011, p. 163–174.

[11] B. Mitra, S. Sural, J. Vaidya, and V. Atluri, “A survey of role mining,” *ACM Computing Surveys*, vol. 48, no. 4, Feb. 2016.

[12] M. Tripunitara, “Minimizing the number of roles in bottom-up role-mining using maximal biclique enumeration,” 2024. [Online]. Available: https://arxiv.org/abs/2407.15278

[13] 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*, ser. CCS ’06, 2006, p. 144–153.

[14] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo, “Mining roles with semantic meanings,” in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT ’08. Association for Computing Machinery, 2008, p. 21–30.

[15] N. Li and M. V. Tripunitara, “Security analysis in role-based access control,” *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 4, p. 391–420, Nov. 2006.

[16] A. Schaad, J. Moffett, and J. Jacob, “The role-based access control system of a European bank: a case study and discussion,” in *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, ser. SACMAT ’01, 2001, p. 3–9.

[17] N. Farhadighalati, L. A. Estrada-Jimenez, S. Nikghadam-Hojjati, and J. Barata, “A systematic review of access control models: Background, existing research, and challenges,” *IEEE Access*, vol. 13, p. 17777–17806, 2025.

[18] L. D’Antoni, S. Ding, A. Goel, M. Ramesh, N. Rungta, and C. Sung, “Automatically reducing privilege for access control policies,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024.

[19] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *KDD*, vol. 96, no. 34, 1996, pp. 226–231.

[20] Y. Wang, Y. Gu, and J. Shun, “Theoretically-efficient and practical parallel dbscan,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20. Association for Computing Machinery, 2020, p. 2555–2571.

[21] M. de Berg, A. Gunawan, and M. Roeloffzen, “Faster db-scan and hdb-scan in low-dimensional euclidean spaces,” 2017. [Online]. Available: https://arxiv.org/abs/1702.08607

[22] Y. A. Malkov and D. A. Yashunin, “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs,” 2018. [Online]. Available: https://arxiv.org/abs/1603.09320

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[24] E. Zhu, V. Markovtsev, A. Astafiev, A. Khan, C. Ha, W. Łukasiewicz, A. Foster, Sinusoidal36, S. Thakur, S. Ortolani, Titusz, V. Letal, Z. Bentley, fpug, hguhlich, long2ice, oisincar, R. Assa, S. Ibraimoski, R. Kumar, Q. TianHuan, M. J. Rosenthal, K. Joshi, K. Mann, JonR, J. Halliwell, and A. Oriekhov, “ekzhu/datasketch: v1.6.5,” Jun. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11462182

[25] The pandas development team, “pandas-dev/pandas: Pandas,” Sep. 2024.

[26] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.