Baaz: A System for Detecting Access Control Misconfigurations

Tathagata Das
Microsoft Research India
tathadas@microsoft.com

Ranjita Bhagwan
Microsoft Research India
bhagwan@microsoft.com

Prasad Naldurg
Microsoft Research India
prasadn@microsoft.com

Abstract

Maintaining correct access control to shared resources such as file servers, wikis, and databases is an important part of enterprise network management. A combination of many factors, including high rates of churn in organizational roles, policy changes, and dynamic information-sharing scenarios, can trigger frequent updates to user permissions, leading to potential inconsistencies. With Baaz, we present a distributed system that monitors updates to access control metadata, analyzes this information to alert administrators about potential security and accessibility issues, and recommends suitable changes. Baaz detects misconfigurations that manifest as small inconsistencies in user permissions that are different from what their peers are entitled to, and prevents integrity and confidentiality vulnerabilities that could lead to insider attacks. In a deployment of our system on an organizational file server that stored confidential data, we found 10 high level security issues that impacted 1639 out of 105682 directories. These were promptly rectified.

1 Introduction

In present-day enterprise networks, shared resources such as file servers, web-based services such as wikis, and federated computing resources are becoming increasingly prevalent. Managing such shared resources requires not only timely availability of data, but also correct enforcement of enterprise security policies.

Ideally, all access should be managed through a perfectly engineered role-based access control (RBAC) system. Individuals in an organization should have well-defined and precise roles, and access control to all resources should be based purely on these roles. When a user changes her role, her access rights to all shared resources should automatically change according to the new role with immediate effect.

In reality though, several organizations use disjoint access control mechanisms which are not kept consistent. Often, access is granted to individual users rather than to appropriate roles. To make matters worse, administrators and resource owners manually provide and revoke access on an as-needed and sometimes ad-hoc basis. As access requirements and rights of individuals in the enterprise change over time, it is widely recognized [19, 12, 5] that maintaining consistent permissions to shared resources in compliance with organizational policy is a significant operational challenge.

Incorrect access permissions, or *access control misconfigurations*, can lead to both *security* and *accessibility* issues. Security misconfigurations arise when a user who should not have access to a certain resource according to organizational policy, does indeed have access. According to a recent report [12], 50 to 90% of the employees in 4 large financial organizations had permissions in excess to what was entitled to their organizational role, opening a window of opportunity for insider attacks that can lead to disclosure of confidential information for profit, data theft, or data integrity violations. The 2007 Price Waterhouse Cooper survey on the global state of information security found that 69% of database breaches were by insiders [24]. On the other hand, accessibility misconfigurations arise when a user who should legitimately have access to an object, does not. Such misconfigurations, in addition to being annoyances, impact user productivity.

Security and accessibility misconfigurations occur due to several reasons. One contributing factor is the high rate of churn in organizations, and in organizational roles among existing employees, which necessitate changes in access permissions. In the same report [12], it was estimated that in one business group of 3000 people, 1000 organizational changes were observed over a period of few months. Another factor is the dynamic nature of information sharing workflows, where employees work together across organizational groups on short-term collaborations. When permissions are granted to shared resources for such collaborations, they are rarely revoked. In longer time-scales, organizations also update their policies in response to changing protection needs. Very often, these policies are not explicitly written down and system administrators, who have an operational view of security, may not have a global view of organizational needs, and may not be able to make these changes in a timely manner.

To make matters worse, very often, no *complete* high-level manifests exist, which correctly assign access permissions for a resource according to organizational policy. Consequently, given the large numbers of shared resources, different access control mechanisms and enterprise churn, it is difficult for administrators to manually

manage access control.

To address these limitations of existing access control management systems, we present Baaz, a system that monitors access control metadata of various shared resources across an enterprise, finds security and accessibility misconfigurations using fast and efficient algorithms, and suggests suitable changes.

To our knowledge, Baaz is the first system that helps an administrator audit access control mechanisms and discover critical security and accessibility vulnerabilities in access control without using a high-level policy manifest. To do this, Baaz uses two novel algorithms: **Group Mapping**, which correlates two different access control or group membership datasets to find discrepancies, and **Object Clustering**, which uses statistical techniques to find slight differences in access control between users in the same dataset.

We do not claim that techniques we use in Baaz will find all misconfigurations, as the notion of policy itself is not defined in most of our deployment settings. Also, given that access permissions change very organically over time and several of these changes are linked to ad-hoc and one-off access requirements, it is very difficult for an automated system to deduce the exact and complete list of all misconfigurations. However, our deployment experiences with real datasets have shown Baaz to be very effective at flagging high-value security and accessibility misconfigurations.

The operational context and main characteristics of Baaz are:

- **No assumption of well-defined policy:** Baaz does not require a high-level policy manifest, though it can exploit one if it exists. Rather than checking for "correct" access control, it checks for "consistent" access control by comparing users' access permissions and memberships across different resources.
- **Proactive vs Reactive:** Baaz takes as input static permissions, such as access control lists, rather than access logs. This approach helps fix misconfigurations *before* they can be exploited, reducing chances of insider attacks. However, the system can be easily augmented to process access logs if required.
- **Timeliness:** Baaz continuously monitors access control, so it can be configured to detect and report misconfigurations on sensitive data items as they occur, or just present periodic reports for less sensitive data.

We present results from Baaz deployments on three heterogeneous resources across two organizations, We interacted with system administrators of both organizations to validate the reports and found a number of high-value security and accessibility misconfigurations, some of which were fixed immediately by the respective system administrators. In all these organizations, no policy manifest was readily available. Before we deployed Baaz, these administrators had to examine thousands of individual or group permissions to validate whether these permissions were intended. The utility of Baaz can be gauged to some extent from some comments we received from administrators:

> *"This report is very useful. I didn't even know these folks had access!"*
>
> *"This output tells me how many issues there are. Now I HAVE to figure out what to do in the future to handle access control better."*
>
> *"I did not realize that our policy change had not been implemented!"*

Our Baaz deployment in one organization found 10 security and 8 accessibility misconfigurations in confidential data stored on a shared file server. The security misconfigurations were providing 7 users unwarranted access to 1639 directories.

The rest of the paper is organized as follows: Section 2 describes our problem scope and assumptions. Section 3 presents the system architecture of Baaz, as well as an overview of our algorithm workflow. Section 4 explains our Matrix Reduction procedure for generating summary statements and reference groups, followed by Sections 5 and 6, in which we present our Group Mapping and Object Clustering algorithms. In Section 7, we outline more detailed issues we encountered while designing the system, and in Section 8, we describe our implementation, deployment and evaluation of the Baaz prototype. Related work is presented in Section 9, and Section 10 summarizes the paper.

2 System Assumptions

The main goal of Baaz is to find misconfigurations in access control permissions (as in ACLs) typically caused by inadvertent misconfigurations, which are difficult for an administrator to detect and rectify manually. We do not detect misconfigurations of access permissions caused by manipulation by active adversaries. We assume that the inputs to our tool, such as the ACLs and well-known user groups, are not tampered. In many organizations, only administrators or resource owners will be able to view and modify these metadata in the first place, so this assumption is reasonable.

In our target environment, a definition of correct policy is not explicitly available. Therefore, rather than checking for correct access control, which we believe is difficult, the system checks for consistent access control. Essentially, Baaz finds relatively *small inconsistencies* in

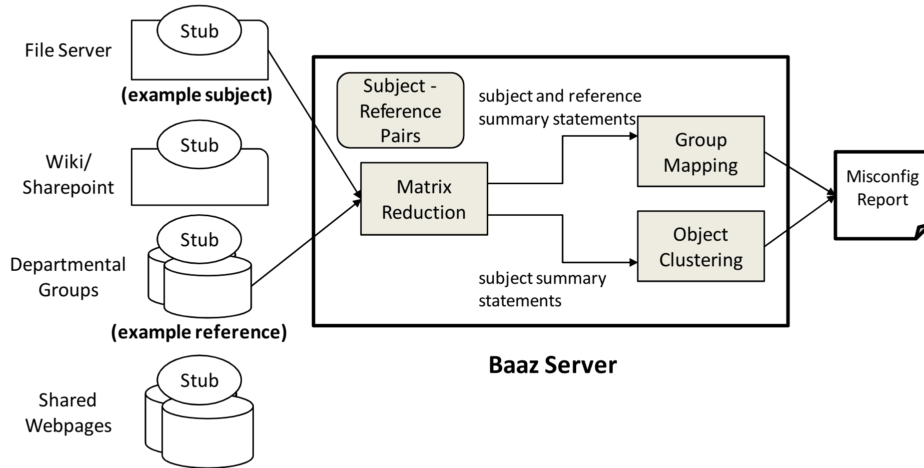

Figure 1: Baaz System Architecture

user permissions by comparing different sets of access control lists, or by comparing user permissions within the same access control list. We assume that *large differences* in access control are not indicative of misconfigurations. Clearly, our definition of small inconsistencies and large differences (provided in Sections 5 and 6) will govern the set of misconfigurations we find. It is possible that this may lead to the system missing some genuine problems which is an inherent limitation. In fact, as described in Section 8.2, our deployment of Baaz missed detecting some valid misconfigurations. However, administrators can tune these parameters to keep the output concise and useful.

3 System Overview

In this section, we present an overview of the system components of Baaz. At the heart of our system, as shown in Figure 1, is a central server that collects access permission and membership change events from distributed *stubs* attached to shared resources. This server runs the misconfiguration detection algorithm when it receives these change events, and generates a report. An administrator/resource owner can decides whether each misconfiguration tuple that Baaz reports is valid, invalid, or an intentional exception. Administrators/owners will need to fix the valid misconfigurations manually. We now provide an overview of the client stubs and server functions.

3.1 Baaz Client Stubs

Baaz stubs continuously monitor access control permissions on shared resources such as file servers, wikis, version-control systems, and databases, and they monitor updates to memberships in departmental groups, email lists, etc. Each stub translates the access permissions for a shared resource into a binary **relation matrix**, an example of which is shown in Figure 2. Each such matrix captures relations specific to the resource that the stub runs on. For example, a file server stub captures the user-file access relationship, relating which users can access given files. On a database that stores organizational hierarchy, the Baaz stubs capture the user-group membership relation, relating which users are members of given groups. We shall refer to an element in the relation matrix M as $M_{i,j}$. A “1” in the ith row and the jth column of M indicates the relation holds between the entity at row i with the entity at column j, e.g., user i can read file j, or user i belongs to group j, whereas a “0” indicates that the relation does not hold.

Each Baaz stub sends $M_{i,j}$ to the Baaz server either periodically, or in response to a change in the relationship. Section 7.2 further describes various issues that we need to consider while designing and implementing stubs.

3.2 Baaz Server

At initial setup, an administrator registers pairs of **subject datasets** and **reference datasets** with the server, which form inputs to the server’s misconfiguration detection algorithm. The subject dataset is the access control dataset which an administrator wants to inspect for misconfigurations. A reference dataset is a separate access control or group membership dataset that Baaz treats as a baseline against which it compares the subject. In a sense, one can view the subject dataset as the implementation, and the reference dataset as an approximate policy, and the process of misconfiguration detection compares the implementation with the approximate policy.

Figure 2 shows an example subject dataset relation matrix of ten users (labeled as A to J) and 16 objects (labeled as 1 to 16), and Figure 3 shows an example reference dataset relation matrix of the same set of users

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A									1	1	1	1	1			
B									1	1	1	1	1			
C	1	1	1	1	1	1	1		1	1	1	1	1		1	1
D	1	1	1	1	1	1	1		1	1	1	1	1		1	1
E	1	1	1	1	1	1	1									
F	1	1	1	1	1	1	1									
G	1	1	1	1	1	1	1									
H	1	1	1	1	1											
I													1			
J																

Figure 2: Example subject dataset's relation matrix

	W	*X*	*Y*	*Z*
A	1		1	
B	1		1	
C	1	1	1	1
D		1		1
E		1		
F		1		
G		1		
H		1		
I				
J		1		

Figure 3: Example reference dataset's relation matrix

and 4 groups (labeled as W to Z). We will use these example inputs to illustrate our misconfiguration detection algorithm.

Administrators can register multiple subject-reference pairs with the server, and each pair is processed independently, with the server periodically generating one misconfiguration report for each. If any changes are detected in matrices corresponding to a registered subject-reference pair, the server runs the misconfiguration detection algorithm, which has three steps:

Matrix Reduction: In the first step, the server reduces the subject and reference datasets' relation matrices to *summary statements* that capture sets of users that have similar access permissions and group memberships. Each summary statement can be thought of as a high-level statement of policy intent, gleaned entirely from the low-level relation matrices. We explain this procedure in Section 4.

Group Mapping: In this step, our goal is to uncover access permissions in the subject dataset that seem inconsistent with patterns in the reference dataset. Consider an example where the subject is a file server, and a reference is a list of departmental groups, as shown in Figure 1. Say a directory hierarchy on the file server can be accessed by all members in the human resources department in an organization, and by only one member of the facilities department. This has a high likelihood of being a security misconfiguration. Section 5 explains this procedure.

Object Clustering: Finally, in the Object Clustering phase, Baaz finds potential inconsistencies in the subject dataset by comparing summary statements for the subject that are "similar", but not the same. The main idea is that a user whose access permissions differ only slightly from that of a larger set of users could potentially be a misconfiguration. For example, if 10 users in the subject dataset can access a given set of 100 files, but say an 11th user can access only 99 of these files, Baaz flags a candidate accessibility misconfiguration. We describe this in Section 6.

The system reports security candidates as *"A user set U MAY NOT need access to object set O"* . Accessibility candidates are of the form *"A user set U MAY need access to object set O"* At this point, the administrator will need to identify reported misconfiguration candidates as "valid", "invalid", or "intentional exceptions", which are defined as follows.

Valid: The misconfiguration candidate is correct, and the administrator needs to make the recommended changes.

Invalid: The misconfiguration candidate is incorrect, and the administrator should not make the recommended changes.

Intentional Exception: The administrator should not make the recommended changes, but the candidate provides useful information to the administrator.

The *intentional exception* category captures all reported misconfigurations that correspond to exceptions which appear out of the ordinary but are legitimate. Administrators found these exceptions to be useful as they help check compliance and may, over time, become valid misconfigurations. An example of an intentional exception is a user who has just changed roles. To help with the transition, he still has access to some documents related to his previous role. Hence while his access should not be revoked at the current time, it should probably be in the near future.

The server archives candidates marked as invalid, and does not explicitly display them in future reports. The reports will, however, display intentional exceptions. Section 7.1 describes more specific issues related to server design and evaluation.

One of the important properties of our algorithms is that the misconfiguration candidates converge to a steady state. That is, if we run our Group Mapping and Object Clustering algorithms repeatedly starting from a given raw configuration, and if we resolve our misconfigurations as suggested, we will eventually (and fairly quickly) reach a state where no new candidates appear. This guarantee is what we call *internal consistency*. We will illustrate this through our examples in Sections 4 and

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A									1	1	1	1	1			
B									1	1	1	1	1			
C	1	1	1	1	1	1	1		1	1	1	1	1		1	1
D	1	1	1	1	1	1	1		1	1	1	1	1		1	1
E	1	1	1	1	1	1	1									
F	1	1	1	1	1	1	1									
G	1	1	1	1	1	1	1									
H	1	1	1	1	1											
I													1			
J																

Subject Dataset Summary Statements

1. $\{C, D\} \rightarrow \{15, 16\}$
2. $\{C, D, E, F, G\} \rightarrow \{6, 7\}$
3. $\{A, B, C, D\} \rightarrow \{9, 10, 11, 12\}$
4. $\{A, B, C, D, I\} \rightarrow \{13\}$
5. $\{C, D, E, F, G, H\} \rightarrow \{1, 2, 3, 4, 5\}$

Figure 4: The result of the matrix reduction step on our example subject dataset's matrix.

	W	X	Y	Z
A	1		1	
B	1		1	
C	1	1	1	1
D		1		1
E		1		
F		1		
G		1		
H		1		
I				
J		1		

Reference Dataset Summary Statements

1. $\mathbf{G_1}$: $\{C, D, E, F, G, H, J\} \rightarrow \{X\}$
2. $\mathbf{G_2}$: $\{A, B, C\} \rightarrow \{W, Y\}$
3. $\mathbf{G_3}$: $\{C, D\} \rightarrow \{Z\}$

Figure 5: The result of the matrix reduction step on our example reference dataset's matrix.

5. The detailed proof is available on our webpage [1]. In the next three sections, we describe the server algorithm in detail.

4 Matrix Reduction

We apply the matrix reduction procedure on the relation matrices of both the subject and reference datasets. The goal of this step, in the context of the subject dataset, is to find summary statements relating sets of users (*user-sets*) that can access the same sets of objects (*object-sets*). Given a relation matrix, different kinds of summaries can be generated. Role mining algorithms [22, 25, 18, 28, 10], for example, try to find minimal overlapping sets of users and objects that have common permissions. In contrast, we find user-sets that have access to disjoint object-sets, as required by our misconfiguration detection algorithms. For the reference dataset, we find group membership summaries in a similar manner.

4.1 Subject Dataset

Our algorithm takes the relation matrix for the subject dataset as input, and examines each column, grouping together all objects that have identical column vectors. Essentially, it groups all objects that are accessible to an identical set of users.

Figure 4 shows the summary statements that our Matrix Reduction algorithm finds for the example shown earlier in Figure 2. Each greyscale coloring within the matrix represents a distinct summary statement. The list of summary statements that our algorithm yields is also shown in the figure. The first statement arises from users C and D having identical access rights, since they both have access to objects 15 and 16, and to no other object. We therefore interpret this in the following way: Users C and D have *exclusive* access to objects 15 and 16, i.e. no other user has access to these objects.

The Baaz server finds all such summary statements to completely capture the matrix. Next, it explicitly filters out all summary statements that involve only one user since our algorithm only looks for misconfigurations involving objects that are shared between more than one user. Figure 6 presents this algorithm in detail.

Complexity: Since the algorithm simply involves one sweep through the subject's relation matrix, grouping together identical columns, it runs in $O(nm)$ time, where n is the number of users in the matrix and m is the number of objects.

EXTRACT SUMMARY STATEMENTS

Input: M {binary relation matrix of all users $\mathcal{U}$ and all objects $\mathcal{O}$}
Output: $\mathcal{S}$ {set of summary statements $[U_k \rightarrow O_k]$}
Uses: H {hashtable, indexed by sets of users, stores sets of objects}

```
1:  S = φ, H = φ
2:  for all o ∈ O do
3:      U = Get User Set(M, o) // gets the set of users who can access o
4:      if H.contains(U) then
5:          O_U = H.get(U)
6:          H.put(U, O_U ⋃ {o})
7:      else
8:          H.put(U, {o})
9:      end if
10: end for
11: for all U_k ∈ H.keys do
12:     O_k = H.get(U_k)
13:     S = S ⋃ {[U_k → O_k]}
14: end for
15: return S
```

Figure 6: Algorithm to extract summary statements given the users and the access control matrix

[1] http://research.microsoft.com/baaz

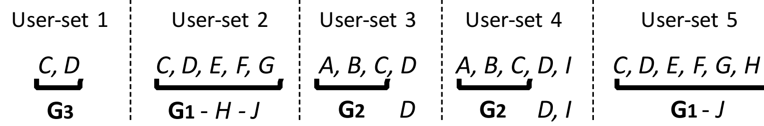

Figure 7: The result of the Group Mapping algorithm on the example subject matrix.

4.2 Reference Dataset

We apply the same process on the matrix for the reference dataset. The summary statements that our algorithm finds for the reference dataset relation matrix are shown in Figure 5. We call the user-set in each summary statement obtained from the reference dataset a *reference group*. The reference groups for our example are:

$\mathbf{G_1} = \{C, D, E, F, G, H, J\}$
$\mathbf{G_2} = \{A, B, C\}$
$\mathbf{G_3} = \{C, D\}$

The objects W, X, Y, Z are merely used to find the reference groups, and are not used by future phases of our algorithm.

5 Group Mapping

In this section, we describe the Group Mapping algorithm, that takes as input the user-sets representing the subject dataset, and the reference groups discovered from the reference dataset, and finds the best mapping from the each user-set to the reference groups. The server uses these maps to flag outliers (users) as misconfiguration candidates. We first explain why Group Mapping is a useful step in finding misconfigurations. Next, we explain how Group Mapping works on our example data, and then we present the algorithm in detail.

5.1 Algorithm

Now we describe the Group Mapping algorithm in more detail. Table 1 summarizes the list of symbols and variables we use here, and in the description of the Object Clustering algorithm.

5.2 Intuition and Definitions

The Group Mapping algorithm for finding misconfigurations relies on the following two assumptions:

1. Users in the same reference group should have same access permissions.

2. Given a set of reference groups that have the same access permissions, any user who is not a member of these reference groups should not have the same access permissions as users within these reference groups.

Based on these two assumptions, we define misconfiguration candidates for the algorithm to find as follows:

Accessibility (based on Assumption 1): If a majority of the members of a reference group all have access to a set of objects, and a minority do not have access to the same set of objects, then we flag the users without access as accessibility misconfiguration candidates.

Security (based on Assumption 2): Of all users in a user-set, if a majority of them form one or more reference groups, and a minority of users do not form any reference groups, we flag the minority of users as security misconfiguration candidates.

Following these definitions, the first thing to do is to find a mapping from user-sets to reference groups. However, since we are looking for outliers, we do not restrict the algorithm to finding an exact and complete mapping. Our goal is to find the "best-effort" mapping from user-sets to reference groups. In this process, some users in the user-sets may not map to any reference group, or a user-set may map to a reference group that has some extraneous users, who are not part of the user-set.

To illustrate with our running example, our Group Mapping algorithm maps the five user-sets in the summary statements we found in Figure 4 to the reference groups found in the Section 4.2 as shown in Figure 7. For the user-set of summary statement 1, the mapping is exact. For the user-set for statement 2, the best map is $\mathbf{G_1}$, which covers all users but also includes users H and J who are not in the user-set. For the user-set in summary statement 4, the best map is $\mathbf{G_2}$, while users D and I remain unmapped.

From this mapping, using the assumptions and definitions stated above, we infer the following misconfiguration candidates:

1. *From summary statement 2, users H and J MAY need access to objects 6, 7.*

2. *From summary statement 3, user D MAY NOT need access to objects 9, 10, 11, and 12.*

3. *From summary statement 4, users D and I MAY NOT need access to object 13.*

4. *From summary statement 5, user J MAY need access to objects 1, 2, 3, 4, and 5.*

Symbol	Definition
n	number of users
m	number of objects
l	number of summary statements/user-sets from subject dataset
g	number of reference groups from reference dataset
$U_i \rightarrow O_i$	i^{th} summary statement for subject, with U_i being the user-set and O_i being the object-set
$\mathbf{G_j}$	j^{th} reference group
C_i	set of groups used to cover user-set U_i
T_i	list of uncovered users in user-set U_i after covering it by C_i
$\Delta\mathbf{G_j}$	list of users in $\mathbf{G_j}$ but not in user-set U_i, where $\mathbf{G_j} \in C_i$

Table 1: Table summarizing all symbols used to explain Group Mapping and Object Clustering

The second and third are security misconfiguration candidates, while the first and fourth are accessibility misconfiguration candidates. User-set 1 does not generate a misconfiguration candidate because the mapping is exact.

Fixing these misconfigurations will improve the mapping from user-sets to reference groups in future runs of the algorithm. For example, if the administrator removes user D's access to objects 9, 10, 11 and 12, the next time the algorithm runs, the summary statement 3 will reduce to $\{A, B, C\} \rightarrow \{9, 10, 11, 12\}$. Group mapping will exactly map the new user-set to $\mathbf{G_2}$, and hence the number of misconfiguration candidates will reduce. This is what we mean by our algorithm reaching an internally consistent state, as mentioned in Section 3.2.

Note that in flagging these candidates, we may have missed some misconfigurations. For example, it is certainly possible that users C and D (forming group $\mathbf{G_3}$) *should not* have access to objects 15 and 16. But given that there is no definition of correct policy, a complete and correct list of misconfigurations cannot be expected. However, Baaz does ensure that the permissions are consistent across user-sets and the reference groups they map to.

Baaz can use role mining algorithms in the Matrix Reduction step to find possibly a larger number of summary statements. However, our definitions of misconfiguration and our algorithms hinge on the property of object-sets being *disjoint*, without which the system may find conflicting misconfiguration candidates. For example, if summary statement 3 included object 15, i.e. $\{A, B, C, D\} \rightarrow \{9, 10, 11, 12, 15\}$, the object 15 would be common to the object-sets of summary statements 1 and 3. Then, from summary statement 3, Group Mapping would suggest that D should not have access to object 15, but the exact Group Map for summary statement 1 indicates that D should have access to object 15. Hence, while Baaz could use role mining algorithms, and leverage richer and larger numbers of user-sets, it would need to include more logic to resolve such conflicts. Instead, we go with the approach of using the simple Matrix Reduction algorithm that provides object-disjoint user-sets.

In spite of its procedural limitations, administrators and resource owners in various domains have found Baaz's techniques very useful in finding genuine high-value misconfigurations. We show this through our evaluation in Section 8,

Say the Matrix Reduction step from Section 4 outputs a total of l summary statements and g reference groups. The input to the Group Mapping step is the set of user-sets $U = \{U_1, U_2, \cdots, U_l\}$ from the summary statements, and the set of reference groups $\mathcal{G} = \{\mathbf{G_1}, \mathbf{G_2}, \cdots, \mathbf{G_g}\}$. Our objective can now be expressed in terms of finding a *set cover* for each user-set U_i using a subset of the groups in $\mathcal{G}$. A set cover, in its usual sense, implies that the union of the covering subsets is exactly equal to the set to be covered. But, we are interested in finding an *approximate set cover*, where the cover need not be exhaustive, and reference groups could include members that are not in the user-set. The idea is to find a *maximal* overlap between the subject dataset user-sets and the reference groups. This *approximate set cover* C_i may contain a group $\mathbf{G_j}$ such that some users in $\mathbf{G_j}$ are absent in U_i, as shown in Figure 7 with user-sets 2 and 5. Also, it is not necessary that C_i covers *every* user in U_i, as shown with user-sets 3 and 4. We refer to the set of uncovered users in U_i as T_i, i.e., is $T_i = U_i - \bigcup_{\forall \mathbf{G_j} \in C_i} \mathbf{G_j}$.

We choose an approximate set cover based on the *minimum description length (MDL)* principle [11], which ensures that the overlap is large, while the leftover set of uncovered users is small. In other words, $|C_i| + |T_i|$ is minimum over all possible *approximate set covers*. The *minimum* set cover problem is known to be NP-Hard, as it can take running time that is exponential on the set of users. By the same logic, the problem of finding *approximate set cover with minimum description length* is also NP-Hard. In practice, we have found that if the number of reference groups is less than 20, then it is feasible to solve it exactly on our testbed computers. For larger reference datasets, we use a well-known greedy approximation algorithm [16], which picks the set that has the maximal overlap, removes it from the reference set, and repeats the process. This is known to work within $O(log\ m)$ of optimal, where m is the number of

GROUP MAPPING

```
Input: S {summary statements}, G {reference groups}
Output:
GAM {accessibility misconfigs [users,objects]},
GSM {security misconfigs [users,objects]}
 1: GAM = φ ; GSM = φ
 2: U = all user-sets in the extracted summary statements S
 3: for all U_i ∈ U do
 4:     (C_i, T_i) = Map Groups (U_i, G)
 5:     for all G_j ∈ C_i do
 6:         if |G_j − U_i| / |U_i| < 0.5 then
 7:             GAM = GAM ⋃ {[G_j − U_i, O_i]}
 8:         end if
 9:     end for
10:     if |T_i| / |U_i| < 0.5 then
11:         GSM = GSM ⋃ {[T_i, O_i]}
12:     end if
13: end for
14: return GAM, GSM
```

MAP GROUPS (APPROXIMATE)

```
Input: U_i {set to be covered}, G {Groups}
Output: C_i {cover from G}, T_i {uncovered users in U_i}
 1: C_i = φ ; T_i = φ ; G' = φ ; U'_i = U_i
 2: for all G ∈ G do
 3:     if |G_j − U_i| / |U_i| < 0.5 then
 4:         G' = G' ∪ {G}
 5:     end if
 6: end for
 7: repeat
 8:     MDL_min = MDL(U_i, C_i) ; G_min = φ
 9:     for all G ∈ G' do
10:         if MDL(U_i, C_i ∪ {G}) < MDL_min then
11:             G_min = G
12:             MDL_min = MDL(U_i, C_i ∪ {G_min})
13:         end if
14:     end for
15:     if G_min = φ then
16:         return C_i, U'_i
17:     end if
18:     C_i = C_i ⋃ {G_min} ; U'_i = U'_i − G_min
19: until U'_i = φ
20: return C_i, φ
```

Figure 8: Group Mapping Algorithm.

users in the user set, for the original minimum set cover problem. We modify this algorithm suitably to generate the *approximate set cover with minimum description length*.

Figure 8 shows the pseudocode for our Group Mapping algorithm. The main steps of the algorithm for a given list of user-sets $\{U_1, U_2, \cdots, U_l\}$ can be summarized as follows:

Step 1: For each user-set, first eliminate all groups in which more than half of the users are not members of the user-set (lines 2–6 in MAP GROUPS, Figure 8). Since less than half of the users in these reference group intersect with the user-set, this reference group will not figure in either security or accessibility misconfiguration candidates as defined in Section 5.2.

Step 2: When the number of groups in $\mathcal{G}$ is less than 20, we exhaustively search for all set covers and use the minimum. For larger $\mathcal{G}$, we use a modified version of the greedy set-cover algorithm to do the matching, as shown in Figure 8. For each user-set U_i, we pick a group $\mathbf{G}$ that overlaps maximally with U_i (pick any one in case of ties). To apply the *minimum description length* principle, we define the description length for U_i in terms of $\mathbf{G}$ as $|U_i - \mathbf{G}| + |\mathbf{G} - U_i|$. For example, in user-set 2, two potential mappings are $\mathbf{G_1}$ as shown in the example, or $\mathbf{G_3}$, which contains users C and D. In the former case, $|U_2 - \mathbf{G_1}|$ is 0, and $|\mathbf{G_1} - U_2|$ is 2, since $\mathbf{G_1}$ contains two extraneous users, H and J. In the latter mapping, $|U_2 - \mathbf{G_3}|$ is 3, since $\mathbf{G_3}$ covers C and D and does not include E, F and G. Also, $|\mathbf{G_3} - U_2|$ is 0. Therefore the MDL metric for the former cover is 2, while in the latter case it is 3. Hence our algorithm picks $\mathbf{G_1}$ as the cover. Refer to lines 8–14 in MAP GROUPS, Figure 8.

Add this selected group to the cover C_i. Remove the covered users from U_i to get U_i' and repeat until all users are covered, and the ones that cannot be covered by any group are output as T_i. Refer to lines 15–19 in MAP GROUPS, Figure 8.

Using this mapping, we can find both security and accessibility misconfigurations for each user set U_i extracted from the summary statements ($U_i \rightarrow O_i$), as shown in lines 4–14 GROUP MAPPING, Figure 8. The summary statement can be rewritten as:

$$\{\mathbf{G_1'} \cup \cdots \cup \mathbf{G_c'} \cup T_i\} \rightarrow O_i.$$

where $\mathbf{G_j'} = \mathbf{G_j} \cap U_i, \forall \mathbf{G_j} \in C_i$. Let $\Delta\mathbf{G_j}$ be the users in $\mathbf{G_j}$ who are not in U_i. Note that Step 1 ensures that $\frac{|\Delta\mathbf{G_j}|}{|\mathbf{G_j}|} < 0.5$, that is $\Delta\mathbf{G_j}$ is a minority in $\mathbf{G_j}$. Based on the intuition provided in the previous section, we infer that users in $\Delta\mathbf{G_j}$ (if any) may require access to the objects O_i. Hence, the intended access should be $\{\mathbf{G_1} \cup \cdots \cup \mathbf{G_c} \cup T_i\} \rightarrow O_i$ and for each $\mathbf{G_j} \in C_i$ such that corresponding $\Delta\mathbf{G_j} \neq \phi$, the system reports accessibility misconfiguration candidate as:

users in $\Delta\mathbf{G_j}$ MAY need access to O_i

Finding security misconfiguration candidates is a slightly different process. Again, for a given user-set U_i, the users in T_i are those that do not match any of the reference groups but still have access to O_i. If these users form a minority of the users in the user-set U_i, that is

$\frac{|T_i|}{|U_i|} < 0.5$ and $T_i \neq \phi$, then the system infers that the intended access should be $\{\mathbf{G_1} \cup \cdots \cup \mathbf{G_c}\} \rightarrow O_i$ and all users in T_i are reported to be security misconfiguration candidates as:

users in T_i MAY NOT need access to O_i

Note that while we use metrics based on simple majority and minority to detect misconfiguration candidates, our prototype implementation supports any threshold value between 0 and 1. A higher threshold may find more valid misconfigurations but may also increase the number of false alarms.

Complexity: The group mapping run time is bounded as $O(k^2 lg)$, where k is the maximum number of users in a reference group, g is the number of reference groups and l is the number of summary statements.

5.3 Misconfiguration Prioritization

When Baaz presents the misconfiguration report to the administrator, it lists the candidates in a priority order. Prioritization of candidate misconfigurations is important because administrators may not have the time to validate all misconfiguration candidates that Baaz outputs, as in Dataset 2 in Section 8. In such cases, a ranking function helps them focus their attention on the high-value candidates.

The main intuition behind our ranking function is that when the mismatches between a user-set and its covering reference group is smaller, the possibility of the misconfiguration candidate being a valid issue is higher. The formula used for prioritization of both accessibility and security candidates capture this measure of difference in similarity between a user-set and its cover.

For accessibility misconfigurations, for a given U_i, the system computes a priority over each reference group G_j in C_i, as:

$$\mathcal{P}(accessibility\, misconfig) = 1 - \frac{\sum_{j=1}^{c} |\Delta \mathbf{G_j}|}{|U_i|}$$

For security misconfiguration candidates, we use the fraction of potentially unauthorized users to prioritize as follows. The smaller the fraction of uncovered users, the higher the priority.

$$\mathcal{P}(security\, misconfig) = 1 - \frac{|T_i|}{|U_i|}$$

6 Object Clustering

Our second technique for finding misconfiguration candidates is Object Clustering. This procedure uses only the summary statements as input and is therefore particularly useful in the absence of suitable reference groups.

Summ St 5: C, D, E, F, G, H ⟹ 1, 2, 3, 4, 5
Summ St 2: C, D, E, F, G ⟹ 6,7
H ⟹ 6,7

Summ St 3: A, B, C, D ⟹ 9, 10, 11, 12
Summ St 4: A, B, C, D, I ⟹ 13
I ⇏ 13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A									1	1	1	1	1			
B									1	1	1	1	1			
C	1	1	1	1	1	1	1		1	1	1	1	1		1	1
D	1	1	1	1	1	1	1		1	1	1	1	1		1	1
E	1	1	1	1	1	1	1									
F	1	1	1	1	1	1	1									
G	1	1	1	1	1	1	1									
H	1	1	1	1	1	1	1									
I													1			
J																

Figure 9: The result of the Object Clustering algorithm on the example subject matrix.

6.1 Intuition

We first present the intuition behind our Object Clustering algorithm. When the access permissions for a small user-set is only slightly different from the access control for a much larger user-set, this may indicate a misconfiguration.

Figure 9 explains this intuition using our example. Observe that the user-sets for summary statements 3 and 4 differ in one user – I – because I has access to object 13, but does not have access to any of 9, 10, 11 and 12. On the other hand, users A, B, C and D have access to objects 9, 10, 11, 12 and 13. Therefore, Baaz suggests a security misconfiguration candidate:

user I MAY NOT need access to object 13.

Similarly, summary statements 5 and 2 differ in only one user – H – because H does not have access to objects 6 and 7. Users C, D, E, F and G have access to 1, 2, 3, 4, 5, 6 and 7. Therefore, as shown in the figure, Baaz suggests an accessibility misconfiguration candidate:

user H MAY need access to objects 6 and 7.

The matrix in Figure 9 shows that if an administrator or resource owner determines that these are indeed valid misconfigurations and fixes them, the matrix becomes more uniform. A future iteration of matrix reduction will output fewer summary statements. In this example, C, D, E, F, G and H now have identical access and hence the reduction will remove summary statement 2. Similarly, since user I will no longer have access to object 13, statement 4 will not be found in future iterations. This will lead to our algorithms finding the same number, or fewer misconfiguration candidates in the future, if no changes are made to the input matrices. This supports our claim of internal consistency in Section 3.2.

Object Clustering

Input: $\mathcal{S}$ {summary statements}
Output: OAM {accessibility misconfigurations [users, objects]}, OSM {security misconfigurations [users, objects]}

1: $OAM = \phi$; $OSM = \phi$
2: **for all** pairs of summary statements in $\mathcal{S}$, $[U1, O1]$ & $[U_2, O_2]$ **do**
3:     **if** $\frac{|U_1 - U_2|}{|U_1|} < 0.5$ and $\frac{|U_2 - U_1|}{|U_1|} < 0.5$ and $\frac{|O_2|}{|O_1|} < 0.5$ **then**
4:         **if** $U_1 - U_2 \neq \phi$ **then**
5:             $OAM = OAM \bigcup \{[U_1 - U_2, O_2]\}$
6:         **end if**
7:         **if** $U_2 - U_1 \neq \phi$ **then**
8:             $OSM = OSM \bigcup \{[U_2 - U_1, O_2]\}$
9:         **end if**
10:     **end if**
11: **end for**
12: **return** OAM, OSM

Figure 10: Object Clustering algorithm.

The Group Mapping and Object Clustering phases do not find disjoint sets of misconfigurations. For example, both the above misconfigurations were also flagged by Group Mapping. We intend to use Object Clustering as a fallback in situations where there do not exist suitable reference groups to flag misconfiguration candidates through Group Mapping.

6.2 Algorithm

We now describe the Object Clustering algorithm in detail. We first look for pairs of summary statements with the following template:

$$U_1 \rightarrow O_1 \text{ and } U_2 \rightarrow O_2 \text{ such that } \frac{|U_1 - U_2|}{|U_1|} < 0.5, \frac{|U_2 - U_1|}{|U_1|} < 0.5, \text{ and } \frac{|O_2|}{|O_1|} < 0.5$$

Now, our definition of an object misconfiguration is as follows: For the two summary statements, $U_1 \rightarrow O_1$ and $U_2 \rightarrow O_2$ that match the template, say $|U_1 - U_2|/|U_1|$ and $|U_2 - U_1|/|U_1|$ are both smaller than 0.5 (a majority of users in U_1 are in U_2 and vice-versa), and $|O_2|/|O_1|$ is smaller than 0.5 (O_2 is less than half the size of O_1). We characterize a security misconfiguration candidate as:

$$U_2 - U_1 \textit{ MAY NOT need access to } O_2.$$

and an accessibility misconfiguration candidate is given as:

$$U_1 - U_2 \textit{ MAY need access to } O_2.$$

Complexity: Given that there are l summary statements, n users, and m objects, the Object Clustering algorithm runs in $O(l^2(n + m))$ time.

6.3 Misconfiguration Prioritization

In the report, as in the case of Group Mapping, the Baaz server prioritizes these misconfigurations using the intuition that the more similar the user-sets U_1 and U_2, and the smaller the size of O_2, the higher the probability that the candidate is a genuine misconfiguration. The metric we use is the harmonic mean:

$$\mathcal{P}(misconfig) = 0.5 * \left((1 - \frac{|\Delta U|}{|U_1|}) + (1 - \frac{|O_2|}{|O_1|})\right)$$

Here ΔU corresponds to $U_2 - U_1$ or $U_1 - U_2$ depending on whether it is a security or an accessibility misconfiguration.

7 System Experiences

In this section, we describe issues that impact the quality of the misconfiguration reports produced by Baaz, based on our experiences in implementing and evaluating the Baaz server and stubs for our prototype, and discuss how we address them in our system design.

7.1 Server Design Issues

Here, we discuss our choice of reference dataset in our deployment and how an administrator can tune report time.

Choosing reference datasets: An administrator needs to use domain knowledge to choose the right reference dataset for a given subject dataset. We observe that the output reports vary depending on how rich or rigid the reference groups are. Some reference datasets, such as organizational group-membership relations, are rigid and structured, and contain few reference groups, potentially generating many misconfiguration candidates in the Group Mapping step, several of which may be invalid. This is because fewer groups will yield more approximate covers.

On the other hand, if a reference dataset contains a large number of reference groups, such as a set of email distribution lists, the report will contain fewer candidates because the chances of finding exact covers increases. As a result, the algorithm may not detect some valid misconfigurations. An administrator can decide which reference dataset to use, based on the sensitivity of the subject dataset, trading manual effort of validation for caution. For example, if a subject dataset folder is marked confidential, the administrator may choose to compare it with the organizational hierarchy, whereas email lists may be a better choice for less sensitive information.

In our evaluation described in Section 8, we choose email distribution lists as a reference dataset for two datasets and organizational hierarchy as a reference for one dataset, and our results verify our observations above.

Tuning report time: Since change events trigger Baaz's misconfiguration detection algorithms, the server may generate reports even in transient states while administrators manually change permissions. To avoid such spurious reports, each pair of subject and reference datasets has an associated *report time* (T_r): Baaz

includes a candidate in its report only if it has existed for at least T_r time. The administrator can configure T_r to be short for subjects that store highly sensitive data, while it can be high for less important subjects. In our deployed prototype, we found that we could generate a report as fast as one second after a stub reports a change, or delay its reporting using T_r, as required.

7.2 Stub Design Issues

We identify two design issues that directly play a role in the quality of generated reports:

Modeling access control: The system's misconfiguration detection can only be as good as the data the stub provides. Access control mechanisms can be complicated [20], which sometimes makes capturing complete semantics in a stub quite hard. In our stub implementations, we have used a conservative approach towards modeling access control: if there is ambiguity of whether an individual or group has access to an object, we assume that they do indeed have access. This approach catches more security candidates albeit at the risk of increasing the number of false alarms. Previously proposed security monitoring systems have tackled this problem [6] using a similar strategy.

Stub customization: Access mechanisms of different kinds of resources will require custom stub implementations that can specifically understand the underlying access controls. Similarly, stubs may need to be customized to different data layouts containing group membership data. However, some stubs can be reused across resources. For example, in our prototype, we have implemented a stub that can run on any SMB-based Windows file share. We have also implemented customized stubs to capture organizational hierarchy and email lists within our enterprise, both of which reside on an Active Directory server [1] (an implementation of the Lightweight Directory Access Protocol, LDAP).

Access control permissions are not necessarily binary. For example, in a file share, "read-only" access or "full access" are only two of a number of different access types possible. Consequently, our stub implementations support various modes of operation. An administrator can choose what a "1" in the binary relation matrix captures: full access, read-only access, *any* kind of access, etc.

8 Evaluation

In this section, we first describe the implementation of Baaz system components (Section 8.1). Next, we describe the results we achieve through our prototype deployment (Section 8.2), followed by a description of the collection, analysis, and validation of misconfiguration reports from two other datasets (Section 8.3). Finally, we present performance evaluation microbenchmarks for demonstrating the scalability (Section 8.4) of the misconfiguration detection algorithms.

8.1 Implementation

We have implemented the Baaz server in C# using 2707 lines of code. We have also implemented Baaz stubs for an SMB-based Windows file server, for organizational groups in Active Directory [1], and for email distribution lists also stored in Active Directory. The Windows file server stub is entirely event-based: it traps changes in access control through the FileSystemWatcher [8] library and reports these changes immediately to the server. Currently, we only trap changes to access control for directories, but we can easily extend this to capture changes for individual files. The Active Directory stubs, on the other hand, poll the database every 8 minutes since we do not have the right permissions or mechanisms to build an event-based stub for Active Directory. The file server stub used 830 lines of C# code and the Active Directory stub, which used a common code base for both the organizational groups and email lists, was 1327 lines of C# code.

8.2 Evaluation Through Deployment

We have deployed Baaz within our organization, with stubs continuously monitoring two resources within our organization since August 19th, 2009. The stubs monitor read access permissions for directories on a Windows SMB file server that the employees use to share confidential data, and an Active Directory server storing email distribution lists relevant to the organization. Various groups within the organization actively use the file server to share documents, hence we found significant usage of access control capabilities on it.

The objective of our deployment was to see whether Baaz could help find valid access control misconfigurations on this file server. We therefore registered the file server as the subject dataset and the email distribution list as the reference dataset with the server. We decided to use email distribution lists as opposed to organizational hierarchy since our administrator observed that only organizational groups might not capture the various user sets that actively use the file server.

We show our results in three steps: first, we show how Baaz's first report in the deployment was effective in finding misconfigurations. Second, we show the utility of continuously monitoring changes in access control to find misconfigurations. Third, we compare our results with the ground-truth we established by manually inspecting directory permissions on the file server, to detect how many actual misconfigurations Baaz was able to flag.

First-time report: Row 1 in Table 2 provides details on this dataset, and row 1 in Table 3 gives the classifica-

Dataset	Subject	Reference	Users	Objects	Ref Groups	Summ Stmts
1	File Server	Email Lists	119	105682	237	39
2	Shared Web Pages	Email Lists	1794	1917	3385	307
3	Email Lists	Org Grps	115	243	11	205

Table 2: Datasets used to evaluate Baaz.

Set	Security								Accessibility							
	Group Mapping				Object Clustering				Group Mapping				Object Clustering			
	Tot.	Val.	Exc.	Inv.	Tot.	Val.	Exc.	Inv.	Tot.	Val.	Exc.	Inv.	Tot.	Val.	Exc.	Inv.
1	11	10	0	1	11	7	1	3	8	8	0	0	9	0	0	9
2	7	3	0	4	0	0	0	0	9	4	1	4	0	0	0	0
3	18	6	5	7	0	0	0	0	33	6	0	27	0	0	0	0

Table 3: Misconfiguration analysis for each report generated by Baaz.

tion of the *first-time* report that Baaz generated using the relation matrices that the stubs sent to the Baaz server initially. The total number of users in the organization is 149, the number of objects (directories) in the subject data set's relation matrix is 105682, and the total number of reference groups (or unique distribution lists) is 237. The matrix reduction phase on the subject dataset produced 39 summary statements.

Baaz flagged a total of 39 misconfiguration candidates. To validate these, we involved the system administrator and the respective resource owners of the directories in question.

Security: Of the 11 security candidates that Baaz found through Group Mapping, 10 were valid security issues which the administrator considered important enough to fix immediately. Object Clustering found 7 of these 10 security misconfigurations, showing that Baaz would have been helpful in flagging security issues even if reference groups were not available to it. However it is clear that Group Mapping works more effectively than Object Clustering when a suitable reference dataset is available.

Accessibility: Baaz found 8 accessibility candidates through Group Mapping, all of which were valid. All 9 accessibility issues that Object Clustering flagged were invalid, showing that, with this dataset, while Group Mapping worked well in bringing out both security and accessibility issues, Object Clustering did well only with the security misconfigurations. Object Clustering was not effective in flagging valid accessibility issues since the difference between the summary statements were unexpectedly large.

Baaz found a total of 18 valid misconfigurations. There were 10 security misconfigurations involving 7 users which, when corrected, fixed access permissions on 1639 out of 105682 directories on the file server. There were 8 accessibility misconfigurations that affected 6 users and 163 directories.

Our deployment also helped us understand some of the reasons for why misconfigurations occur in access control lists, which we summarize below.

- In most cases, the misconfigurations arise because of employees changing their roles or, as in some accessibility issues, from new employees joining the organization.
- One of the security misconfigurations was caused by a policy change within the organization, which had only been partially implemented. Certain older employees had greater degree of access than newer employees since the administrator had inadvertently applied the policy change only to employees who had joined after the change was announced.
- A resource owner misspelt the name of one of the users they wanted to provide access to, inadvertently providing access to a completely unrelated employee.

Real-time report: In our deployment, the stubs and the server are running continuously, monitoring access control and group membership changes and subsequently running the misconfiguration detection algorithm. On September 20th, 2009, an employee within the organization adopted a new role, which was reflected by his addition to certain email distribution lists. The Baaz stub reported these changes to the server, following which the server reported one new accessibility misconfiguration candidate within one second. The administrator considered this accessibility misconfiguration important enough to rectify promptly. This emphasizes the value of Baaz's continuous monitoring approach since it enables administrators to detect misconfigurations in a nearly real-time fashion, just after they occur.

Comparison to Ground-Truth: To understand how close Baaz was to finding all misconfigurations for this file server, we manually examined access permissions of all directories on the file server from the root down to three levels. Beyond the third level, we only examined directories whose access permissions differed from their

parent directories. We examined a total of 276 directories.

For each directory, we asked the directory owner two questions: If any user permissions to the directory should be revoked (security misconfiguration), and if anyone else should be provided access (accessibility misconfiguration). This procedure took two days to complete because of the manual effort involved. While we cannot claim that even this procedure would find all possible misconfigurations, we felt this exercise formed a good base-line to compare against Baaz.

We found that Baaz missed 4 security misconfigurations and 1 accessibility misconfiguration. Two security issues went undetected because an email list relevant to these issues was marked as private by the owner, and hence our Active Directory stub could not read the members. If we had the permission to run the stub with administrator privileges, Baaz would have flagged these issues. The other 3 issues (2 security and 1 accessibility) were genuinely missed by Baaz since there were no reference groups that matched the user-set, and the number of users involved in the misconfiguration (2) were more than half the size of the user-set (3).

Hence, while Baaz genuinely missed 3 misconfigurations, it did flag 18 valid misconfigurations which the administrator found very useful.

8.3 Snapshot Evaluation

We evaluated Baaz on two other subject and reference data pairs. We wrote stubs to gather snapshots of access control and group memberships from these datasets and generated a one-time report. Rows 2 and 3 of Table 2 describe the datasets and Table 3 summarize our findings. Dataset 2's subject is a server hosting shared internal web pages for projects and groups across an organization. The stub for this subject reads access permissions stored in an XML file in a custom format. The reference was, again, a set of email distribution lists created for this organization. This subject dataset comprised 1794 users and 1917 objects. For this dataset alone, the administrator decided to concentrate on misconfiguration candidates with priority more than 0.8.

In Dataset 3, the subject dataset is the set of email lists used as reference in Dataset 1, and the reference is the set of *organizational groups*. Here, each organizational group consists of a manager and all employees who report directly to the manager. As we have mentioned earlier, a reference dataset in Baaz may itself be inaccurate. Hence, this evaluation helps us check how stale the memberships to these email lists are. The number of users in this Dataset is 115 and the number of objects is 243. The slight discrepancy in the number of users in Datasets 1 and 3 is due to organizational churn in the period between when we ran the two experiments.

Baaz found many valid misconfigurations in all these datasets. Across all datasets, most security misconfigurations resulted due to role changes. Other security misconfigurations arose because an individual user, who had full permissions to an object, had inadvertently given access to another user who should not have had access. The causes of accessibility misconfigurations, similarly, were moves across organizations or inadvertent mistakes on the part of the individual manually assigning permissions.

We now summarize some other insights we acquired through this evaluation.

Administrator input: Baaz can only make recommendations. Only an administrator, or someone who has semantic knowledge about access requirements, needs to make the final decision of whether a misconfiguration is valid, an exception or invalid. For distributed access control systems such as Windows file servers, the validation will have to be through querying multiple people in the organization since objects involved in the misconfiguration can have different owners. This is not a simple task.

Despite this difficulty, overall, the administrators and resource owners found the system very useful since it found several valid security and accessibility misconfigurations. Moreover, what the administrators appreciated was that, instead of tracking down correct access for potentially thousands of objects, they needed to concentrate on a much smaller set of misconfiguration candidates that Baaz reports. For Datasets 1 and 3, the validation was mostly through conversation and email, and took approximately one hour. For Dataset 2, it took a total of three days turnaround time since we communicated only through email with resource owners who were at a remote site to complete the validation. Note that these are total turnaround times: it does not mean that an administrator spent three complete days just on the validation procedure.

Group Mapping vs Object Clustering: While Group Mapping is universally effective at finding misconfigurations, the Object Clustering approach is effective only in datasets which have a lot of statistical similarity. This is because Object Clustering relies on finding small deviations from a regular and often-repeated pattern of access control permissions. Datasets 2 and 3 do not have a regular pattern since most project web pages and email distribution lists had unique access permissions. Consequently, Object Clustering does not report any misconfigurations for these datasets. On the other hand, it does find misconfigurations for the file server (Dataset 1) since there were many directories on the file servers we evaluated with the same access permissions.

Invalid Misconfigurations: The number of invalid misconfigurations varies significantly across the different datasets. This is related to our discussion in Section 7.1.

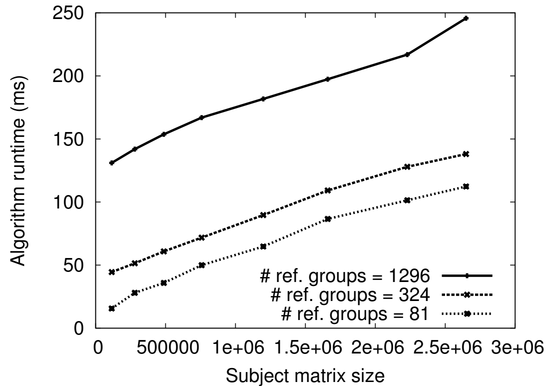

Figure 11: Scalability of the Baaz Algorithm

The organizational groups form a rigid reference dataset, so in Dataset 3, we see a large number of invalid misconfigurations. Across the datasets however, the number of invalid misconfigurations were small enough not to discourage an administrator in adopting our tool.

8.4 Algorithm Performance

In this Section, we concentrate on the performance and scalability of the server algorithm. We used Dataset 1 described in Table 2 for this experiment.

We ran the misconfiguration detection algorithm on the dataset while varying the subject relation matrix size, keeping the number of reference groups constant. To increase the matrix size, we increased the directory depth up to which we included objects into the subject's relation matrix, consequently increasing the number of objects, and therefore, the number of columns in the matrix.

Figure 11 shows the results of our experiments. Each line represents the algorithm's total run time which includes all three phases – Matrix Reduction, Group Mapping and Object Clustering – with different numbers of reference groups. We varied the number of reference groups by adding artificially created groups to the reference dataset while ensuring that the additional groups follow the same size distribution as the real reference groups. Every point in the graph is averaged across 20 runs. We ran all the experiments on a machine with a 3 GHz Intel Core 2 Duo CPU and 4 GB Memory, running a 64-bit version of Windows Server 2008.

With a matrix size of 2.7 million, and with 1296 reference groups, the misconfiguration detection takes a total of 246 ms to run. The increase in time is fairly linear in the matrix size because the Matrix Reduction step dominates the total run-time of the algorithm. For the same data point, where Matrix Reduction needs to inspect roughly 2.7 million cells in the subject's relation matrix, Group Mapping needed to process only 24 summary statements and 1296 reference groups, and Object Clustering processed ${}^{24}C_2 = 276$ summary statement pairs.

Projecting from this graph, for a subject dataset representing 100,000 employees and 100,000 objects, i.e., a matrix size of 10^{10}, and a reference dataset involving 1296 groups, the misconfiguration detection would take approximately 340 seconds to run. Our experiments indicate that the algorithm can scale to large datasets (much larger than encountered in our deployments as shown in Table 2), and run fast enough to provide prompt misconfiguration reports.

9 Related Work

In this section, we discuss our work in the context of related research.

Recent work by Baker et al. in detecting policy misconfigurations [4] uses data mining to infer association-rules between groups of resources that can be accessed by common sets of users, based on an off-line analysis of access attempts in log files. The authors use the profile and frequency of granted requests to predict and fix operational *accessibility* issues. For example, if a user belonging to such a common set inadvertently does not have access to a particular resource, their tool will flag this as a misconfiguration, and refer this to an appropriate resource owner.

Baaz on the other hand operates on access permissions. Consequently, in most cases, Baaz can flag and suggest fixes for misconfigurations before they can be exercised operationally. While access log analysis is an extremely useful mechanism in detecting security and accessibility issues, the approach is inherently complementary to the approach of analyzing access control permissions. Ideally, the two should be used in tandem.

Also, Baaz primarily uses a different technique, Group Mapping, whereby the system compares subject and reference datasets: several of the misconfigurations that the Group Mapping algorithm found in our evaluation could not have been found using association rules alone. These include the examples presented in Section 8.2 where users change roles, or new employees join an organization, and have not accessed any resources yet. In addition, Baaz finds both security and accessibility issues whereas Baker et al. concentrate only on accessibility issues.

Finally, the goal of their misconfiguration detection is similar in intent to Baaz's Object Clustering algorithm. While Baaz focuses on identifying sets of users that can access disjoint sets of objects, they identify all possible sets of users who have common access permissions to (possibly overlapping) sets of objects. In Baaz, we chose to focus on disjoint object-sets for reasons explained earlier.

Network intrusion prevention and detection systems also have a similar operational view of misconfigura-

tions [15, 14]. An attempt is made to characterize normal behavior, as opposed to anomalous behavior, and any deviation from this characterization is flagged as a potential vulnerability. In contrast, research on automatically discovering attack graphs [2, 23], by correlating information across lists of known software-vulnerabilities, improper access controls, and network misconfiguration issues, have a forensic flavor. This aspect is further explored in more recent work such as HeatRay [6], which explores identity-snowball attacks based on over-entitled user privileges across a networked enterprise. The HeatRay tool outputs suggestions to administrators to prune privilege-lists on particular machines, maximizing security versus availability tradeoffs, using machine learning and combinatorial optimization techniques. A system such as Baaz can help an administrator decide whether to remove access permissions as suggested by HeatRay.

Other related work on policy anomaly detection includes the work on access control spaces [13] where the authors describe a policy-authoring tool called Gokyo that can help discover policy coverage issues.While Gokyo assumes a high-level policy manifest exists, Baaz works in scenarios where such manifests are not available.

Role-based access control (RBAC) [21] is widely cited as a useful management tool to control access permissions by separating out the user-role and role-permission relationships. However, RBAC is known to be difficult to implement in practice [5, 12]. The problem of role mining [22, 25, 18, 28, 10] is related to Baaz's matrix reduction step (Section 4), where we find related user and object groups. In role-mining, the user-object access matrix is analyzed to find maximal overlapping groupings of users and objects that have the same permissions. In contrast, in Baaz, we are interested in misconfigurations on shared-object permissions, as opposed to discovering common patterns of access across user groups. Nevertheless, like organizational groups, email groups, and distribution lists, the output of a role-mining algorithm, specifically the user-role mappings, can be used as an input to our group mapping phase. We believe that even if organizations adopt some flavor of RBAC, a system like Baaz is useful in discovering misconfigurations caused by exceptions and role changes. There is also a wealth of related work on the topic of clustering in general, and a summary of this is outside the scope of this work.

Policy anomaly detection is also a popular subject of study in the firewall and network configuration space. Here, existing tools [27] explore the semantics of different filtering rules and firewall policies. Testing and static analysis techniques [26, 17, 3] have been proposed to explore and understand how policy configurations satisfy properties such as redundancy and contradiction. However, all of these techniques are specific to firewall configurations and are inherently different from Baaz which uses comparison across ACL datasets and within the same dataset to find misconfigurations.

Several network security scanning tools are actively used by network administrators to find vulnerabilities such as open ports, vulnerable applications and poor passwords [7, 9]. Baaz's purpose and techniques target a different problem – finding access control misconfigurations – and are therefore complementary to the intent of these tools. In fact, a number of such tools and systems should be used in tandem to ensure a high level of security for all enterprise resources.

10 Conclusion

In this paper, we have described the design, implementation and evaluation of Baaz, a system used to detect access control misconfigurations in shared resources. Baaz continuously monitors access permissions and group memberships, and through the use of two techniques – Group Mapping and Object Clustering – finds candidate misconfigurations in the access permissions. Our evaluation shows that Baaz is very effective at finding real security and accessibility misconfigurations, which are useful to administrators.

Acknowledgments

We would like to thank our shepherd, Somesh Jha, for his valuable comments and suggestions. We would also like to thank Ohil Manyam for testing and optimizing the prototype Baaz system, Rashmi K. Y, Geoffry Nordlund, and Chuck Needham for help with evaluating Baaz, and Geoffrey Voelker, Venkat Padmanabhan and Vishnu Navda for providing insightful comments that improved earlier drafts of this paper.

References

[1] Active Directory. http://www.microsoft.com/windowsserver2003/technologies/directory/activedirectory/.

[2] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[3] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[4] L. Bauer, S. Garriss, and M. K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proc. SACMAT '08*, pages 185–194, New York, NY, USA, 2008. ACM.

[5] Bruce Schneier, Real-World Access Control. http://www.schneier.com/crypto-gram-0909.html.

[6] J. Dunagan, A. X. Zheng, and D. R. Simon. Heat-ray: Combating identity snowball attacks using machine learning, combinatorial optimization and attack graphs. *SIGOPS Oper. Syst. Rev.*, 2009.

[7] D. Farmer and E. H. Spafford. The COPS security checker system. In *Proceedings of the Summer Usenix Conference*, 1990.

[8] File System Watcher Class. http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.aspx.

[9] S. S. A. T. for Analyzing Networks. http://www.porcupine.org/satan.

[10] M. Frank, D. Basin, and J. M. Buchmann. A class of probabilistic models for role engineering. In *CCS '08*. ACM, 2008.

[11] P. D. Grunwald. *The Minimum Description Length Principle*. The MIT Press, 2007.

[12] Information Risk in the Professional Services-Field Study Results from Financial Institutions and a Roadmap for Research. http://mba.tuck.dartmouth.edu/digital/Research/ResearchProjects/DataFinancial.pdf.

[13] T. Jaeger, X. Zhang, and A. Edwards. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, 2003.

[14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. *SIGOPS Oper. Syst. Rev.*, 39(5):91–104, 2005.

[15] S. T. King and P. M. Chen. Backtracking intrusions. *SIGOPS Oper. Syst. Rev.*, 37(5):223–236, 2003.

[16] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994.

[17] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 177, Washington, DC, USA, 2000. IEEE Computer Society.

[18] 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*, 2008.

[19] Privileged Password Management: combating the insider threat and meeting compliance regulations for the enterprise. http://www.cyber-ark.com/constants/white-papers.asp?dload=IDC_White_Paper.pdf.

[20] M. Russinovich, D. Solomon, and A. Ionescu. *Windows Internals, 5th Edition*. Microsoft Press, 2009.

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

[22] J. Schlegelmilch and U. Steffens. Role mining with orca. In *Proc. SACMAT '05*, pages 168–176, 2005.

[23] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[24] The insider threat: automated identity and access controls can help organizations mitigate risks to important data. http://findarticles.com/p/articles/mi_m4153/is_2_65/ai_n25449309.

[25] J. Vaidya, V. Atluri, and J. Warner. Roleminer: mining roles using subset enumeration. In *CCS '06*, pages 144–153. ACM, 2006.

[26] A. Wool. Architecting the lumeta firewall analyzer. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2001. USENIX Association.

[27] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.

[28] D. Zhang, K. Ramamohanarao, and T. Ebringer. Role engineering using graph optimisation. In *SACMAT '07*, pages 139–144. ACM, 2007.