Composition of Access Control Schemes via Co-Existence: Three Case-Studies

[Work in Progress Paper]

Indrani Ray
University of Waterloo
Waterloo, Canada
indrani.ray@uwaterloo.ca

Mahesh Tripunitara
University of Waterloo
Waterloo, Canada
tripunit@uwaterloo.ca

Kami Vaniea
University of Waterloo
Waterloo, Canada
kami.vaniea@uwaterloo.ca

Abstract

We address a particular kind of composition of access control schemes which include both a model to encode authorizations, and an administrative model to encode the manner in which such authorizations can change. The kind of composition we address is co-existence: two schemes working side-by-side in the same system in protection of the same set of resources with the same principals who seek access. We observe that practice appears to be ahead of research in this regard, and discuss case-studies we have carried out of three real-world compositions. We then articulate takeaways from the case-studies.

CCS Concepts

• **Security and privacy → Access control**.

Keywords

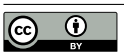
Composition, Co-existence

ACM Reference Format:
Indrani Ray, Mahesh Tripunitara, and Kami Vaniea. 2026. Composition of Access Control Schemes via Co-Existence: Three Case-Studies: [Work in Progress Paper]. In *Proceedings of the 31st ACM Symposium on Access Control Models and Technologies (SACMAT '26), July 08–10, 2026, Waterloo, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3750555.3811908

1 Introduction

The design of security subsystems can be a significant challenge. We address an aspect of this: the co-existence of different access control schemes.

Access control determines whether a principal, such as a user, is able to exercise certain actions, such as read or write, on a resource, such as a file. It is widely acknowledged to be an essential aspect of security. Access is granted if the principal is authorized; the set of all authorizations is called the authorization state. An authorization state is typically encoded in some syntax, that has traditionally been called a *model*; the access matrix [16] and Role-Based Access Control (RBAC) [11] are examples. Different models can be seen merely as different syntaxes to encode authorizations.



SACMAT '26, Waterloo, ON, Canada

ACM ISBN 979-8-4007-2107-6/2026/07
https://doi.org/10.1145/3750555.3811908

Consequently, a debate in research has been whether one should reconcile several different models, or focus on a single model, or "meta model" [4]. We do not challenge the validity of such a debate; however, we observe that in practice, i.e., in real systems, the debate appears to be settled: different models are supported, and that too, simultaneously within a single system (see our case-studies in Section 3).

A question that then arises is: in what manner should two access control systems that are based on different models yet protect the same set of resources and have the same set of principals, co-exist? What are different design choices that are meaningful to consider in such a co-existence? Rather than perceiving this as an exercise in articulating properties that are then proven, we see this as an exercise in good design, with choices that can be made either way.

There are answers in prior research to the broader question as to how two access control systems *compose*. We can categorize them into two approaches: "top-down" and "bottom-up". By top-down we refer to work such as those of Bonatti et al. [8] and Bertino et al. [6] that propose a framework within which to express different models and thereby compose systems that are instances of each. By bottom-up we refer to work such as that of Rao et al. [19] and Li et al. [17] that begin with real-world models and consider compositions of systems (or policies, in those two pieces of work) that are instances of those. There are different types of compositions that seem to be possible; accordingly, prior work has used terms such as "combining", "integration", and "composition". Our focus is the specific kind of composition via co-existence, and our approach is bottom-up under the above categorization. We discuss prior work further, and distinguish each from our work, in Section 2.

Schemes. Beyond models that are used to encode authorization states, there is the issue that authorization states can change over time. A specification for the manner in which authorization states may change has been called *administration* in prior work [20], and a syntax for such specifications is called an administrative model. An example, for the access matrix model is the command-syntax of Harrison et al. [16]. Real-world systems, presumably out of necessity, realize such administrative models along with their access control models. Consequently, the problem of composition needs to reconcile access control *schemes* [23], which include both an access control model, which encodes authorization states, and an administrative model, which encodes the manner in which a state may change to another.

There is little prior research on composing access control schemes, and these consider limited kinds of state-changes (see, for example, the work of Becker et al. [5]), and not full administrative models

such as the command syntax of Harrison et al. [16] or the ARBAC family of models of Sandhu et al. [20].

Our work. We discuss three case-studies we have carried out of composition via co-existence in real-world systems. Two of the systems we study are cloud systems: Amazon Web Services (AWS) [1], and Google Cloud [14]. The other is a database system: Oracle [18]. Our specific focus is to understand design choices that have been made in these systems. We then articulate takeaways from the case-studies which we argue can be adopted as design criteria for such compositions.

Organization. The remainder of this paper is organized as follows. In the next section, we discuss related work. In Section 3, we discuss our three case-studies; at the end of that section, we list key takeaways. We conclude with Section 4 in which we discuss future work as well.

2 Related Work

As we mention in Section 1, prior work can largely be categorized into "top-down" and "bottom-up". By the former, we mean work that proposes frameworks or highly expressive models within which to encode and thereby compose other models. Such work includes those of Bai et al. [3], Schneider [22], Bonatti et al. [8, 9], Bertino et al. [6], Ferraiolo et al. [12], Wijesekera and Jajodia [24], Becker et al. [5] and Bertolissi et al. [7]. We now discuss those pieces of work in turn. The work of Bonatti et al. [8, 9] proposes an algebra for authorizations with the specific intent of modeling compositions of models within it. The algebra focuses on what it calls policies, rather than more general purpose models such as the access matrix or RBAC. Also, it does not address administrative models, nor does it reconcile design criteria from real-world systems.

Bai et al. [3] propose a way to specify state transformations in authorization policies by defining a policy base comprised of a finite set of closed first-order formulae and a Herbrand model of the set. They define transformations of the models and propose algorithms to discover transformed models based on the principle of minimal change. Schneider [22] presents an automata-based formalism for specifying security policies and provides techniques on how to enforce them. To compose security automata, that work suggests a conjunction of security policies. Neither addresses co-existence, nor real world systems as our work does.

The work of Bertino et al. [6] proposes a logic for encoding different access control models, and reasoning about properties such as consistency, and comparing the expressive power of those models. The explicit intent does not appear to be composition; however, the logic can be adapted as a framework for that purpose. The work does not address administration, nor composition via co-existence.

The work of Ferraiolo et al. [12] proposes the Policy Machine, whose intent is a single syntax in which several different models can be encoded simultaneously, and thereby composed. While the Policy Machine exhibits high expressive power for models, it does not address administration, nor does it address design criteria from real-world systems as our work does. The work of Wijesekera and Jajodia [24] also proposes a logic for modeling different policies. Its main feature is to model what it calls controlled nondeterminism that itself models administration, i.e., state-changes, in particular ways. Another intent is to abstract away details of models so that one can reason about properties such as consistency (whether a principal is both allowed and denied access, for example). It is unclear that the logic can capture full administrative models such as those of Harrison et al. [16] and Sandhu et al. [20]. Also, it does not consider composition via co-existence, nor real-world systems.

The work of Becker et al. [5] also proposes a logic for authorization models (or policies, as it calls them). Its explicit intent is to be able to capture what it calls state-modifying behaviour. These are not full administrative models, but rather restricted kinds of state-changes that result from access-actions, such as in the Chinese Wall model [10]. The work of Bertolissi et al. [7] appeals to category theory to develop a broadly expressive model, and administrative model, that can then express compositions in a single syntax. Neither pieces of work considers co-existence of schemes, nor reconcile real-world systems as we do.

As to "bottom-up" work on compositions, we are aware of the work of Rao et al. [19] and Li et al. [17]. Both address the composition of XACML policies. The focus in the former is the specification of several different types of constraints in such compositions. The focus of the latter is the provision of what it calls a language for such compositions that allows for enforcement disciplines such as weak and strong majority, and reconciles issues such as policy evaluation errors. As such, these pieces of work are focused on XACML policies. They do not address administrative models, nor syntaxes other than XACML.

3 Three Case-Studies

In this section, we discuss three case-studies of composition of access control schemes via co-existence. In the constituent Sections 3.1, 3.2 and 3.3, we discuss schemes in Google Cloud [14], Amazon Web Services (AWS) [1] and Oracle [18], respectively. We do not consider schemes in their entirety as realized by these systems; rather we focus on portions of the schemes that serve our purpose of understanding the manner in which composition via co-existence is realized in those systems. Our description of each is organized as follows:

- We first state the resources that are protected, and any dependencies between them to the extent that those impact access control. We state also the principals who seek to access those resources. This delimits also the kinds of accesses (e.g., "download an object") we consider.
- We describe one of the access control models, and then the administrative model that goes with it. We specify also the manner in which access is enforced in the former, i.e., the conditions under which a pricipal is allowed and denied access in an authorization state of the model.
- We describe the other access control model, the manner in which access is enforced, and the administrative model.
- We describe the manner in which the two schemes co-exist.

3.1 IAM and ACLs in Google Cloud

In this section, we discuss our use-case of the Google Cloud Storage service [14]. A principal in this system is a Google Cloud user, bound to an email address, who is authenticated. There are two

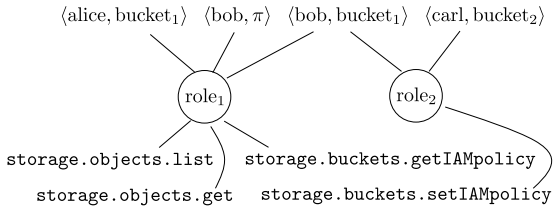


Figure 1: An example IAM authorization state in Google Cloud.

	b_1	o_1	b_2
alice	OWNER	OWNER	
bob	OWNER		WRITER
carl		READER	

Figure 2: An example ACL authorization state in Google Cloud rendered as an access matrix. The set of rights = {WRITER, OWNER, READER}. It is possible for more than one user to be in the OWNER group for a bucket or object. Not shown is that $containedIn(o_1) = b_1$.

types of resources that are protected by each of the schemes we consider: *buckets*, a set we denote as B, and *objects*, the set O. A bucket contains zero or more objects, and an object is a chunk of data. Each object is contained in exactly one bucket, i.e., there exists a function, $containedIn: O \to B$. Neither a bucket nor an object is of the other type, i.e., $B \cap O = \emptyset$.

The two access control schemes we consider are *Identity and Access Management* (IAM), which is role-based, and *Access Control Lists* (ACL), which is a variant of the access matrix.

3.1.1 IAM. In addition to the users, U who are the principals, and resources B, O we mention above, that exist, an authorization state in the Google Cloud IAM comprises the following:

- π: a Google *project* which we represent as a static string which is neither a bucket nor object ($\pi \notin B \cup O$). The reason we incorporate a project in our considerations is that authorizations on the project may have implications to authorizations on buckets and/or objects. Google does allow multiple projects to exist and authorizations related to those; we restrict our considerations to a single project only that does not change.
- P: a set of *permissions*; this is a somewhat large set that is fixed [15].
- L: a set of *roles*.
- $ur: L \to 2^{U \times (B \cup \{\pi\})}$, a function which associates pairs of ⟨user, bucket/π⟩ with a role l.
- $lp: L \to 2^P$, a function which associates a set of permissions with a role.

We point out that *ur* and *pr* above are different from the customary manner in which one maps users, resources and access-actions to a role-based model — see Section **??**. However, our exposition here more accurately captures this scheme. An example of an authorization state is in Figure 1.

Access enforcement. An access-action a user may want to exercise, e.g., "download object", is associated with a set of permissions that the user must be authorized to, either for the project π or the bucket b. In the case that the action pertains to an object o, these permissions are for the bucket b where $containedIn(o) = b$. A user $u \in U$ is authorized to a permission $p \in P$ for (the project) π if and only if there exists a role $l \in L$ such that $\langle u, \pi \rangle \in ur(l)$ and $p \in lp(l)$. A user $u \in U$ is authorized to a permission $p \in P$ for a bucket $b \in B$ if and only if there exists a role $l \in L$ such that $\langle u, b \rangle \in ur(l)$ or $\langle u, \pi \rangle \in ur(l)$, and $p \in lp(l)$.

A user is authorized to exercise an action if he is assigned to roles that collectively contain permissions that are sufficient for that action. That is, suppose the permissions $\{p_1, p_2, \ldots, p_n\}$ are sufficient to exercise an action that pertains to a bucket b. For example, the action "download object" is associated with the two permissions `storage.objects.list` and `storage.objects.get`, which the user must possess for the bucket that contains the object. Then it must be the case that there exists a set of roles, $L_1 \subseteq L$ such that:

- For each $l \in L_1$, $\{\langle u, b \rangle, \langle u, \pi \rangle\} \cap ur(l) \neq \emptyset$, and,
- $\{p_1, p_2, \ldots, p_n\} \subseteq \bigcup_{l \in L_1} lp(l)$.

Administrative model. The administrative model in IAM is role-based, i.e., a user acquires a permission that pertains to a state-change via membership in a role. A state-change modifies one of U, B, O, L, ur or lp. We use the '$'$' notation below to indicate a new set or function that results from such a change. We have investigated 14 state-changing actions as part of this case-study. Each action is parameterized by an initiator $i \in U$ and users, resources, roles, and/or permissions.

An example state-change is: remove role from user on bucket (i, l, u, b), which alters ur as follows:

$$ur'(l) \leftarrow ur(l) \setminus \{\langle u, b \rangle\}$$

Similar to an access-action, the initiator $i \in U$ of such a state-change must be authorized to a set of permissions on the particular bucket b that pertains to the state-change, or the project π, for it to succeed. In the case of the above example, this is the two permissions `storage.buckets.getIAMpolicy` and `storage.buckets.setIAMpolicy`. More generally, if $\{p_1, \ldots, p_n\}$ is this set of permissions, it must be the case that there exists a set of roles $L_1 \subseteq L$ such that:

- For each $l \in L_1$, $\{\langle i, \pi \rangle, \langle i, b \rangle\} \cap ur(l) \neq \emptyset$, and,
- $\{p_1, \ldots, p_n\} \subseteq\in \bigcup_{l \in L_1} lp(l_i)$

3.1.2 ACL. In addition to the principals, who are the users U, and the resources B, O, an authorization state for the ACL scheme in Google Cloud Storage comprises:

- P: a set of *permissions*, {`OWNER`, `WRITER`, `READER`}. A user who is authorized to `OWNER` on a resource is authorized also to `WRITER`, and a user who is authorized to `WRITER` is authorized also to `READER`.
- $up: P \to 2^{U \times (B \cup O)}$ a function which associates pairs of ⟨user, bucket/object⟩ to a permission. This function is constrained in that: (i) a user is allowed to be authorized to at most one permission from P on a resource, and, (ii) `WRITER` is authorized to buckets only, and not objects.

Access enforcement. A user is authorized to exercise an action if they possesses the necessary permission over a corresponding resource. For example, to exercise the action download object on an object o where $o \in \mathit{containedIn}(o) = b$, the must exist a permission $p \in P$ such that $\langle u, b\rangle \in \mathit{up}(p)$, or $p \in P \setminus \{\texttt{WRITER}\}$ such that $\langle u, o\rangle \in \mathit{up}(p)$.

Administrative model. As with the IAM scheme, every state-change is performed by an initiator, $i \in U$. There are four state-changes we have investigated: create object, remove object, change permission on bucket and change permission on object. Each requires the initiator to possess a permission, which may be acquired either via roles in IAM, or "directly". For example, to exercise create object to create an object in a bucket b, the initiator must possess `storage.objects.create` to b via IAM, or `WRITER` on b via an ACL. A change permission action automatically revokes the current permission the target user possesses as a user is allowed exacly one permission on a resource only; change permission is parameterized in a particular way to entirely revoke a user's current permission.

3.1.3 Composition via co-existence. Let s be an authorization state in an IAM system which comprises the sets U_1, B_1, O_1, P_1, and L, the project π, and functions ur and lp. In the composed state, we have a set of users $U = U_1 \cup U_2$, a set of buckets $B = B_1 \cup B_2$ and of objects $O = O_1 \cup O_2$, a project π, a set of IAM permissions P_1, a set of ACL permissions P_2, a set of IAM roles L, functions ur, lp, and up. An access is allowed if and only if it is allowed via one of the schemes. A state-change from a composed state is a state-change in one of the schemes. Provided a state-change is possible in one of the schemes, that state-change is possible when the schemes co-exist.

However, not all possible states in the two systems are able to co-exist. Google Cloud simply prevent such a composed state from coming into being. Let t be an authorization state in an ACL system that comprises the sets U_2, B_2, O_2, and P_2, and the function up. Then, s and t compose only if it is never the case that there is both (i) an IAM role that grants access to a user on a bucket, and, (ii) an ACL permission that grants access to the user on the bucket or an object that is contained in the bucket. Otherwise, s and t compose, i.e., may co-exist. We adopt this observation that not every two states may compose as a key takeaway and design criterion — see Section 3.4.

3.2 Identity- and Resource-Based Policies in Amazon Web Services (AWS)

In this section, we address another use-case we have investigated: the Amazon Simple Storage Service (S3).

A principal is an authenticated user identified uniquely by their Amazon Resource Name (ARN). As in the use-case of the previous section, there are two types of resources: *buckets*, B, and *objects*, O, with a function $\mathit{containedIn}\colon O \to B$ and $B \cap O = \emptyset$.

The two access control schemes we consider are both variants of the access matrix: *identity-based* and *resource-based*. We consider a simplified version of these schemes which suffices for our purposes; we refer the reader to Backes et al.[2] for a more comprehensive discussion. There is a role-based version of these schemes which we do not consider; we consider the discretionary versions only, in which users are assigned privileges directly, and not via roles.

```
{ "Statement": [{
    "Effect": "Allow",
    "Action": "s3:ListBucket",
    "Resource": "arn:aws:s3:::myBucket" },
  { "Effect": "Deny",
    "Action": "s3:GetObject",
    "Resource": "arn:aws:s3:::yourBucket/*" }] }
```

Figure 3: An AWS identity-based policy that comprises two statements. The first grants a certain permission, and the second explicitly denies a permission.

3.2.1 Identity-based. Authorizations in this scheme comprise *policies*, each of which is attached to one or more users. A policy comprises *statements*, each of which specifies either an allowance or denial of access to a set of rources. Thus, in addition to the principals and resources we mention above, we can characterize an authorization state as comprising:

- P: a fixed of *permissions*. AWS calls these "actions", and the strings that correspond to these are verbs, e.g., "`s3:DeleteObject`". We call these permissions to distinguish them from access-actions of a user. Indeed, prior work has pointed out that there is not necessarily a one-to-one correspondence between a permission and a corresponding access-action [13]
- $E = \{\texttt{Allow}, \texttt{Deny}\}$: a fixed set of two possible *effects*.
- $ib\colon U \to 2^{(2^{B \cup O}) \times E \times 2^P}$ a function that maps a user to sets of statements, each containing a set of resources, an effect, and a set of permissions.

Figure 3 shows an example of an identity-based policy which comprises two statements.

Access enforcement. If a user has certain access, then this must be explicit, i.e., there must exist a statement in a policy attached to the user with the effect `Allow` for that access. Denial may be either explicit or implicit. The former corresponds to the existence of a statement with the effect `Deny`; the latter to the non-existence of a statement with `Allow`. This scheme adopts a deny-overrides discipline.

Administrative model. The administrative model is discretionary, i.e., there are specific permissions that pertain to state-changes, and if a user possesses such permissions, they may effect a state-change. An example of such a permission is `s3:DeleteBucketPolicy`, which allows an (administrative) user to remove a policy associate with a bucket.

3.2.2 Resource-based. Resource-based policies are similar to identity-based policies, except that they are attached to buckets. Thus, the resource to which the policy pertains is specified by the buckets to which it is attached. Rather than a function ib as we mention above for identity-based policies, an authorization state can be thought of including a function $rb : B \to 2^{2^U \times E \times 2^P}$ that maps buckets to sets of statements, each containing a set of users, an effect, and a set of permissions. Figure 4 shows an example of a resource-based policy that comprises a single statement.

```
{ "Statement": [{
    "Effect": "Allow",
    "Principal": {
        "AWS": "arn:aws:iam:000000000000:user/someUser"
    },
    "Action": "s3:ListBucket",
    "Resource": "arn:aws:s3:::myBucket/*" }] }
```

Figure 4: An AWS resource-based policy with one statement. It mentions the principal to whom the access pertains, and the resource to which the policy is bound.

	table_1	table_2
alice	$\langle \text{INSERT}, 1 \rangle$	
bob	$\langle \text{READ}, 0 \rangle$, $\langle \text{DELETE}, 0 \rangle$	$\langle \text{SELECT}, 1 \rangle$

Figure 5: An example discretionary authorization state in Oracle rendered as an access matrix. The user alice is allowed to further grant INSERT to table_1 to another user as their second component is 1. The user bob is not allowed to further grant SELECT to table_2.

Access enforcement is similar to that for identity-based policies, i.e., access is allowed only if there is no explicit `Deny`, and there is an explicit `Allow`. Access may be denied either explicitly or implicitly; the discipline is deny-overrides. The administrative model is discretionary, as with identity-based policy. An administrative user may effect a state-change to the authorization state if they possess specific permissions that pertain to state-changes.

3.2.3 Composition via co-existence. Co-existence of the two schemes above is straightforward. Every possible pair of states across the two schemes may co-exist, unlike in the Google Cloud schemes (see Section 3.1). A state-change is enabled in the composition if and only if it is enabled in one of the constituent systems, i.e., a state-change corresponds to a change in either an identity-based policy or a resource-based policy.

3.3 Discretionary- and Role-Based Policies in the Oracle Database System

Our third and final case-study are portions of two schemes in the Oracle Database system [18].

A principal is a so-called "common user", who is bound to a unique username and is authenticated. The resources to which we limit our case-study are database *tables*, T.

The two access control schemes we consider are *discretionary* and *role-based*.

3.3.1 Discretionary. In addition to the principals and resources we mention above, an authorization state in the discretionary model comprises:

- P: a set of privileges; this is a fixed set specified by Oracle, e.g., "`SELECT`" and "`DELETE`".
- $tp \colon T \times U \to 2^{P \times \{0,1\}}$ a function mapping tables (or any table) and users to sets of $\langle \text{privilege}, 0/1 \rangle$ pairs. We use the bit 0 or 1 to indicate whether a privilege is possessed without "grant option", or with it. As we discuss below for state-changes, "with grant option" enables the possessor of the privilege to further grant it to other users.

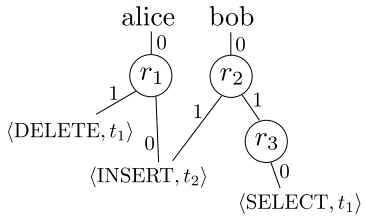


Figure 6: An example of a role-based authorization state in Oracle. Each user-role and role-role edge is anotated with 1 to indicate that that grant is with admin option, and 0 if it is not. Each role-privilege edge is annotated with 1 if it has been granted with grant option, and 0 if it has not. For example, the user bob is able to exercise INSERT to table t_2, and also further grant that privilege to other users.

Figure 5 shows an example of a discretionary state in Oracle.

Access enforcement. A user is authorized to privilege $p \in P$ on a table $t \in T$ if $\langle p, 0 \rangle \in tp(t, u)$ or $\langle p, 1 \rangle \in tp(t, u)$. A user is authorized to an access if they are authorized to a corresponding privilege. For example, $\langle \texttt{DELETE}, \cdot \rangle$ $intp(t, u)$, the the user u may delete (remove) rows from the table t; we use "$\cdot$" to indicate that it does not matter whether the bit is 0 or 1.

Administrative model. The administrative model in the discretionary scheme is itself discretionary. That is, there are specific privileges that authorize an administrative user to perform state-changes to the authorization state. For example, if a user a possesses the "`GRANT ANY OBJECT PRIVILEGE`", then a may grant the "`DELETE`" privilege on any table to another user. A user b may do this on a particular table t if, instead, $\langle \texttt{DELETE}, 1 \rangle \in tp(t, b)$, i.e., posssesses the `DELETE` privilege with grant option. There is also the notion of an owner for every table; the owner of a table may grant any privileges on it to another user.

3.3.2 Role-based. In addition to the users and tables that exist, an authorization state in the role-based model comprises:

- P: a set of privileges; this is the same set as in the discretionary model.
- R: a set of roles.
- $ur : U \to 2^{R \times \{0,1\}}$ a function which associates sets of $\langle \text{role}, 0/1 \rangle$ pairs to users. As with privileges in the discretionary model, the bit indicates whether the role has been granted with the "admin option" or not. This matter for state-changes only, and not for access. A role granted with the admin option enables the user to further grant the role to other users, and to even remove the role entirely. These are all state-, i.e., administrative, changes.
- $rr \colon R \to 2^{R \times \{0,1\}}$ a function which associates sets of $\langle \text{role}, 0/1 \rangle$ pairs to roles. This enables a role-hierarchy [21] to exist. In practice, Oracle allows the role-hierarchy to contain cycles; best practice is to keep it acyclic. We assume, to define the *unpac* below, that it is acyclic.

- rp: $R \rightarrow 2^{2^{P \times \{0,1\}} \times T}$ a function which associates roles to sets of tuples, each containing a set of granted privileges and a table.
- $unpac$: $R \rightarrow 2^{2^{P \times \{0,1\}} \times T}$ a function which associates roles to sets of tuples, each containing a set of granted privileges and a table. We use this function to "traverse" the role-hierarchy to identify the privileges a user possesses in totality. That is, for any role $r \in R$, $unpac(r) = rp(r) \bigcup_{s \in rr(r)} unpac(s)$.

Figure 6 shows an example of a role-based authorization state in Oracle.

Access enforcement. A user is authorized to privileges via roles in a natural way. That is, suppose there exists a role r such that some $\langle X, t \rangle \in unpac(r)$, where $\langle p, \cdot \rangle \in X$. Suppose also that $r \in ur(r)$. Then, we say that u is authorized to the privilege p on the table t. (As before, we use "$\cdot$" to indicate that the value of that bit is consequential.) A user may exercise access-actions that correspond to privileges to which they are authorized.

Administrative model. A user may effect a state-change if they are authorized to a particular privilege that enables them. An example of such a privilege is `GRANT ANY ROLE`. This privilege may be acquired either via the discretionary system, or via a role. And there is the notion of an initiator (as administrative user) of state-change. (It is possible also that a state-change in the discretionary scheme is enabled by an authorization via roles.)

3.3.3 Composition via co-existence. Every possible pair of states from across the two schemes compose, unlike with Google Cloud (see Section 3.1). A state-change is enabled in the composition if it is enabled in one of the two schemes. Also, as we discuss above, a state-change to the role-based scheme may be made via authorization in the discretionary scheme, and vice versa. Thus, there is considerable "interchangeability" across the two schemes.

Another interesting aspect with this case-study is the possession of a privilege with grant, or admin, option. This further emphasizes discretionary state-changes because such a privilege enables a user to further grant the privilege (in the case of the grant option), or role (the admin option), to another user. Perhaps this is a consequence of the evolution of these schemes in Oracle — the discretionary scheme was implemented before the role-based scheme.

3.4 Some Takeaways from our Case-Studies

The intent of our case-studies is to learn design choices that have been made in those settings in practice with regards to composition via co-existence of access control schemes. In this section, we discuss some takeaways, which are design choices.

(1) Not all authorization states across the two schemes necessarily compose.
As an example from our case-studies, in Google Cloud (Section 3.1), not all states across the two schemes compose. Let S_1 be a state in the IAM-scheme with users U_1, buckets B_1, and roles L. Assume there is a user $u_1 \in U_1$ granted a role $l \in L$ on bucket $b \in B_1$ (i.e. $\langle u_1, b \rangle \in ur(l)$). Let S_2 be a state in the ACL-scheme with users U_2, buckets B_2, and permissions P. Assume there is a user $u_2 \in U_2$ granted a permission $p \in P$ on bucket $b \in B_2$ (i.e. $\langle u_2, b \rangle \in up(p)$). Given that $b \in B_1 \cap B_2$ and users are authorized to it in both states, S_1 and S_2 cannot compose — Google Cloud precludes them from existing simultaneously.

(2) If the instance of one of the schemes is in some authorization state, are the authorizations of that state *reflected* in, i.e., part of, the authorization state of the other scheme? In the IAM scheme of Google Cloud (Section 3.1), there is a concept of "legacy roles." These roles are mapped to fixed sets of IAM permissions and are equivalent to ACL permissions, i.e., enable the same access-actions. Thus, when a user is granted a permission on a bucket or object in the ACL scheme, there is an IAM state in which the user is granted a legacy role.

(3) There may exist constructs in one of the schemes in support of the other scheme. For example, in the resource-based scheme of AWS S3 (Section 3.2), a user can only set a bucket policy on a pre-existing bucket. To create a bucket, a user needs to be authorized to the `s3:CreateBucket` permission via an identity-based policy.

(4) Suppose we have a composed authorization state $S \circ S'$, and a transition T that is enabled in that state in the composed scheme. Note that T corresponds to one of the schemes in the composition, i.e., S if T corresponds to the first scheme, and S' if it corresponds to the second scheme. Then, if that scheme is considered in isolation, T is not necessarily enabled when we are in the state S or S'.
In the Google Cloud (Section 3.1) the action to revoke the role l from a user u on bucket b initiated by an administrative user i. In a composed state, the user i is authorized to exercise this action if, for example, he is assigned to the (ACL) permission of `OWNER` on the bucket. However, he cannot exercise this action in one of the component states, because there is no notion of a role in the ACL scheme.

4 Conclusions and Future Work

We have addressed the composition of access control schemes via co-existence, i.e., two schemes existing side-by-side in the same system, in protection of the same set of resources with the same principals. We have carried out a case-study of three real-world systems, and learned design criteria from those.

There is much scope for future work. While our work takes the mindset of adopting design criteria from practice, there is the question of what properties may be of interest commonly across most, if not all, instances of co-existence. For example, is there a meaningful notion of consistency, or completeness, that we can articulate, and then ask how we go about establishing such a property when two schemes co-exist? There is the issue also of more than two schemes co-existing in the same system. An example of this is the Oracle Database system [18]; while we have considered two schemes that co-exist, there is at least one more: a scheme that protects so-called stored procedures, which co-exists with the two schemes we have addressed in this work.

References

[1] Amazon, Inc. Amazon web services (AWS). https://aws.amazon.com/, last accessed: February, 2026.

[2] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for aws access policies using smt. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9, 2018.

[3] Yun Bai and V. Varadharajan. A logic for state transformations in authorization policies. In *Proceedings 10th Computer Security Foundations Workshop*, pages 173–182, 1997.

[4] Steve Barker. The next 700 access control models or a unifying meta-model? In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, page 187–196, New York, NY, USA, 2009. Association for Computing Machinery.

[5] Moritz Y. Becker and Sebastian Nanz. A logic for state-modifying authorization policies. *ACM Trans. Inf. Syst. Secur.*, 13(3), July 2010.

[6] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, February 2003.

[7] Clara Bertolissi, Maribel Fernández, and Bhavani Thuraisingham. Admin-cbac: An administration model for category-based access control. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, CODASPY '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Piero Bonatti, Sabrina de Capitani di Vimercati, and Pierangela Samarati. A modular approach to composing access control policies. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, page 164–173, New York, NY, USA, 2000. Association for Computing Machinery.

[9] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security*, 5(1):1–35, 2002.

[10] D.F.C. Brewer and M.J. Nash. The chinese wall security policy. In *Proceedings. 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[11] David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn. A role-based access control model and reference implementation within a corporate intranet. In *ACM Trans. Inf. Syst. Secur.*, volume 2, pages 34–64, New York, NY, USA, 1999. ACM.

[12] David F. Ferraiolo, Serban Gavrila, Vincent Hu, and D. Richard Kuhn. Composing and combining policies under the policy machine. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, SACMAT '05, page 11–20, New York, NY, USA, 2005. Association for Computing Machinery.

[13] Puneet Gill, Werner Dietl, and Mahesh Tripunitara. Least-privilege calls to amazon web services. *IEEE Transactions on Dependable and Secure Computing*, 20(3):2085–2096, 2023.

[14] Google, Inc. Google cloud. https://cloud.google.com/, last accessed: February, 2026.

[15] Google, Inc. IAM roles and permissions index. https://docs.cloud.google.com/iam/docs/roles-permissions, last accessed: February, 2026.

[16] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, aug 1976.

[17] Ninghui Li, Qihua Wang, Wahbeh Qardaji, Elisa Bertino, Prathima Rao, Jorge Lobo, and Dan Lin. Access control policy combining: theory meets practice. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, page 135–144, New York, NY, USA, 2009. Association for Computing Machinery.

[18] Oracle, Inc. AI database. https://www.oracle.com/database/, last accessed: February, 2026.

[19] Prathima Rao, Dan Lin, Elisa Bertino, Ninghui Li, and Jorge Lobo. An algebra for fine-grained integration of xacml policies. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, page 63–72, New York, NY, USA, 2009. Association for Computing Machinery.

[20] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The arbac97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, February 1999.

[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] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[23] Mahesh V. Tripunitara and Ninghui Li. Comparing the expressive power of access control models. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 62–71, New York, NY, USA, 2004. Association for Computing Machinery.

[24] Duminda Wijesekera and Sushil Jajodia. A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.*, 6(2):286–325, May 2003.