

Model Checking Access Control Policies: A Case Study using Google Cloud IAM

Antonios Gouglidis^a, Anna Kagia^a, Vincent C. Hu^b

^a*School of Computing and Communications, Lancaster University, Lancaster, LA1 4WA Lancashire, U.K.*

^b*Computer Security Division, NIST, 100 Bureau Drive, Gaithersburg, 2089 Maryland, U.S.A.*

Abstract

Authoring access control policies is challenging and prone to misconfigurations. Access control policies must be conflict-free. Hence, administrators should identify discrepancies between policy specifications and their intended function to avoid violating security principles. This paper aims to demonstrate how to formally verify access control policies. Model checking is used to verify access control properties against policies supported by an access control model. The authors consider Google’s Cloud Identity and Access Management (IAM) as a case study and follow NIST’s guidelines to verify access control policies automatically. Automated verification using model checking can serve as a valuable tool and assist administrators in assessing the correctness of access control policies. This enables checking violations against security principles and performing security assessments of policies for compliance purposes. The authors demonstrate how to define Google’s IAM underlying role-based access control (RBAC) model, specify its supported policies, and formally verify a set of properties through three examples.

Keywords: Role-based access control, access control, authorization, policy verification, temporal logic, NuSMV

1. Introduction

The objective of an access control system is to control and limit the actions or operations in a system that an authorized user or process can perform on a set of resources [1, 2]. Access control is the process that checks all requests to a system and takes a decision to grant or deny access based on a set

of rules. This makes it an essential component in all computing systems. In recent years, Cloud services have rapidly grown, rendering Cloud computing a popular computing paradigm. It changed the way organizations obtain IT resources and reduced costs significantly. As a result, Cloud computing has received considerable attention from academia as well as industry. Access control in the Cloud poses significant security challenges, e.g., secure inter-operation [3], and supporting security assessment of policies [4].

Access control policies dictate who has what access to which resource and thus it is important that these policies are error-free throughout their life-cycle. However, in practice, policies often do not satisfy the desired security requirements, and flaws in their specification can remain hidden and cause observable harm when exploited. Indeed, [5] states that misconfigurations in access control policies are one of the main reasons for security and privacy breaches due to potential inconsistencies. To eliminate unwanted access control discrepancies, verifying and rigorously testing access control policies before enforcing them in an operational environment is necessary. Nevertheless, the correct specification of access control policies is challenging since it is difficult to identify discrepancies between policy rule specifications and their intended functions for ensuring no violation of access control security principles [6].

Although the integrated tools provided by Cloud providers can check policies for errors, Cloud administrators have little control over the specification of security requirements that can be formally verified in access control policies. We anticipate that having an automated technique to verify the correctness of access control policies against a set of desired security requirements would serve as a valuable tool for Cloud administrators. This may assist in promptly identifying issues in the existing policies and provide information on how to exploit them. In this paper, we use an existing Identity and Access Management (IAM) system (i.e., Google’s Cloud IAM) as a case study to elaborate on how policies can be modeled and subsequently verified against a set of user-defined properties.

The main contributions of this paper are:

- Demonstrate how we formally define the RBAC model of IAM based on the limited publicly available information.
- Specify a transition system for the RBAC model and demonstrate how to specify access control policies and properties in temporal logic.

- Verify user-defined properties in policy examples provided by Google through the above methods.

In the rest of this paper, we review some of the related work in Section 2, define Google’s Cloud IAM RBAC model in Section 3, specify a transition system for the defined RBAC model and relevant properties in Section 4, verify example policies in Section 5, and present concluding remarks in Section 6.

2. Related Work

Zhang et al., [7] described the main Cloud access control models for OpenStack, AWS, and Microsoft Azure Cloud platforms. They provided a formal specification of these access control models and extended them to include the capability of handling information and resource sharing across tenants. Power et al., [8] presented two formal models of the access policy language used within the AWS Cloud computing infrastructure. They followed a hybrid approach by using both the Z specification language and the Alloy modeling language to test multiple policy properties and generate and test candidate policies. Evangelidis et al., [9] proposed a probabilistic verification scheme based on performance modeling and formal verification of Cloud-based auto-scaling policies. To demonstrate the applicability of their method, they used a validation process on Amazon EC2 and Microsoft Azure, considering two different Cloud service models, i.e., IaaS and PaaS. Others focused on the challenges faced by the Cloud computing growth and conducted comparison studies between popular Cloud service providers, e.g., [10] compared Amazon EC2 and Microsoft Azure regarding how they deal with the challenges of availability, resource scaling, data deletion, data lock-in, and data security. Tajadod et al., [11] compared the same platforms looking at the security of architecture and the application levels.

A number of papers address verification of access control policies and several techniques have been reported in [5, 12, 13, 14, 15, 16, 17, 18]. Their objectives are to look at methods that can check the correctness of policies. In this paper, we demonstrate the application of a generic technique, following NIST’s guidelines [5], which can verify access control properties against policies supported by an access control model.

In addition to the aforementioned approaches, a few access control verification tools were developed [12, 19, 20, 21, 22] to facilitate policy-testing,

with Access Control Policy Tool (ACPT) [19] and Security Policy Tool (SPT) [23] as representative examples. The NIST Computer Security Division developed ACPT in collaboration with the North Carolina State University and University of Arkansas[24], and it is an implementation of the verification method in [5]. Through a graphical user interface (GUI), it provides templates for composing access control policies and properties and verifying them using a symbolic model verification (SMV) checker, NuSMV [25]. Moreover, it provides a complete test suite generated by NIST’s combinatorial testing tool ACTS [26] and generates XACML policy outputs of the verified model. SPT provides the same fundamental functions as ACPT with an extension of adding advanced features as a commercial product [23].

3. The $RBAC_{GCP}$ Model

Cloud IAM is part of the Google Cloud Platform (GCP), allowing Cloud administrators to control users’ access to resources. Hence, when enforcing a policy, an organization can meet its regulatory and business objectives [27]. “*Cloud IAM manages access control by defining who (identity) has what access (role) for which resource*” [28]. A high-level description of the RBAC model used in Google’s Cloud IAM is available. Although its formal definition is not provided, Google documents its main entities, relations, and main operations. We formally define an access control model for Cloud IAM by following publicly available information and specify it based on the ANSI INCITS 359-2012 RBAC [29], which provides a solid foundation for defining role-based models. The following sections provide formal definitions of the main elements and functionalities of the model. Henceforth, we refer to the GCP RBAC model as $RBAC_{GCP}$.

3.1. Model Description

The $RBAC_{GCP}$ model consists of eight elements: MEMBERS, ROLES, PERMISSIONS, RESOURCES, SERVICES, VERBS, POLICIES, and CONDITIONS. It binds MEMBERS to ROLES and ROLES to PERMISSIONS instead of assigning PERMISSIONS directly to MEMBERS [28]. Figure 1 illustrates the relation of $RBAC_{GCP}$ elements. A MEMBER representing a human user or autonomous entity can access RESOURCES through a ROLE representing a job function described by a collection of PERMISSIONS. PERMISSIONS determine what VERBS (i.e., operations) are allowed on a system’s RESOURCE (e.g., Compute Engine instances, Cloud Storage buckets).

A POLICY is a collection of ROLE bindings, which bind one or more MEMBERS to individual ROLES. CONDITIONS assigned on ROLE bindings are logical expressions based on Google’s Common Expression Language (CEL) and assigned on ROLE bindings.

Typically, in Cloud IAM, MEMBERS can be of the following type: Google account, Service account, Google group, G Suite domain, or Cloud Identity domain [28]. ROLES can be Primitives, Predefined, or Custom. Primitives are the three concentric roles that have always existed in the GCP console: the Owner, Editor, and Viewer ROLES. The Owner ROLE contains the PERMISSIONS of the Editor, and the Editor ROLE includes the Viewer’s PERMISSIONS. Google creates and maintains predefined roles and can provide granular access to specific GCP resources. Each product in the Google Cloud platform has its predefined role since different types of operations apply to different resources. A particular kind of role in Cloud IAM is Custom, which allow administrators to combine one or more PERMISSIONS and create unique ROLES that satisfy their organizations’ needs when predefined ROLES are insufficient. Custom roles can only be granted within the Organization and cannot be used to grant PERMISSIONS on RESOURCES owned by a different Organization. Maintaining custom roles poses a challenge for administrators in creating potential security risks despite their flexibility. These ROLES are user-defined, therefore, not maintained by Google. Also, they are not automatically updated when an administrator adds new permissions, features, or services to the GCP [30]. Consequently, administrators must always keep up with the changes and ensure that any new functionality

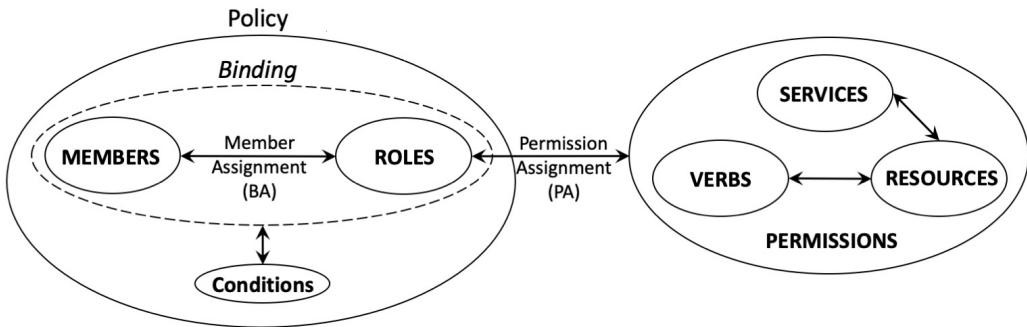


Figure 1: The $RBAC_{GCP}$ model.

is consistent with the existing access control policies so as not to violate the security principles of the Organization. This task is challenging and can be highly complex and time-consuming [31].

PERMISSIONS in Cloud IAM are tuples $\langle service \rangle, \langle resource \rangle, \langle verb \rangle$ that describe using VERBS what OPERATIONS are allowed on a RESOURCE. A PERMISSION is defined per SERVICE and RESOURCE since every RESOURCE enables different OPERATIONS [28]. For example, the PERMISSION `"storage.buckets.create"` indicates creating a bucket in Cloud Storage is permitted for the storage service. RESOURCES are the fundamental components that comprise the GCP services, the Compute Engine instances (i.e., virtual machines), the App Engine services, the Cloud Storage buckets, and the Cloud Pub/Sub topics [32]. RESOURCES in Cloud IAM are hierarchical, as shown in Figure 2. Projects are the children of the Folders, which are children of Organization, and the Resources are the descendants of Projects at the lowest level. Folders is an optional grouping mechanism.

POLICIES of Cloud IAM manage access to RESOURCES. A POLICY is a collection of statements that define the BINDING of ROLES and MEM-

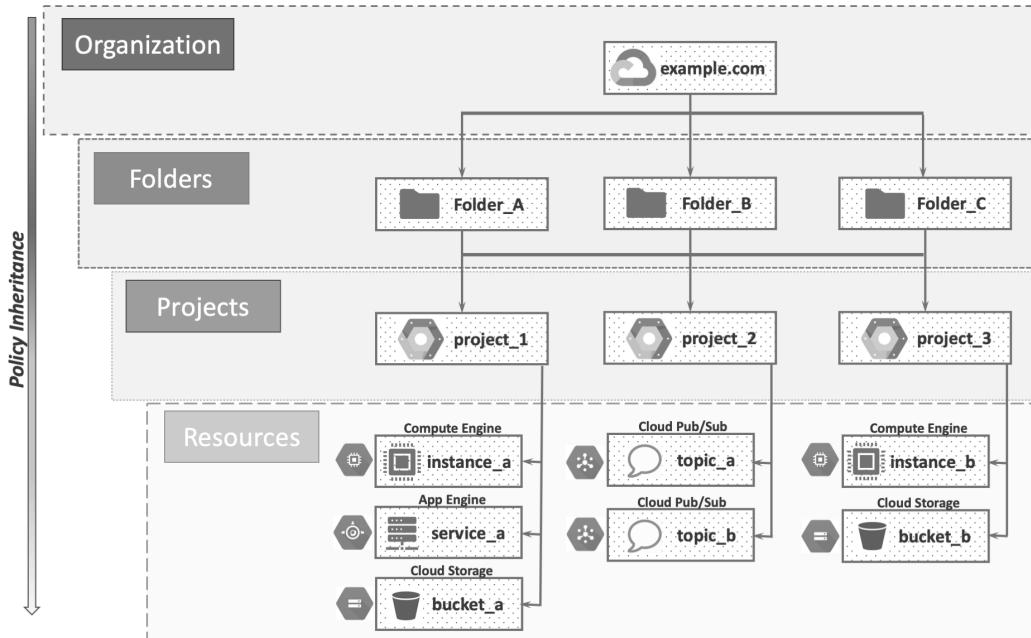


Figure 2: The Cloud IAM resource hierarchy (based on [33]).

BERS, as illustrated in Figure 3 [28]. BINDINGS can contain a CONDITION, an expression that includes one or multiple logic statements that evaluate various conditional attributes, which is optional, and each role BINDING may have only one. A BINDING without a CONDITION will always grant the ROLE to the specified MEMBERS. A BINDING is valid if a CONDITION is evaluated to TRUE. CONDITIONS provide constraints based either on the availability of a requested RESOURCE or on the situation of the access request. Examples for the former are the RESOURCE type and the RESOURCE name, and for the latter, the date/time of the request, the expected URL path, and the destination IP address. The enforcement of CONDITIONS can support attribute-based access control (ABAC) [34] to enhance the $RBAC_{GCP}$ model, allowing administrators to create more flexible and efficient access control policies. For instance, they can grant access to MEMBERS only during specified working hours and only for a specific RESOURCE type with the desired access level [35].

POLICIES are hierarchical and follow the same path as the RESOURCE

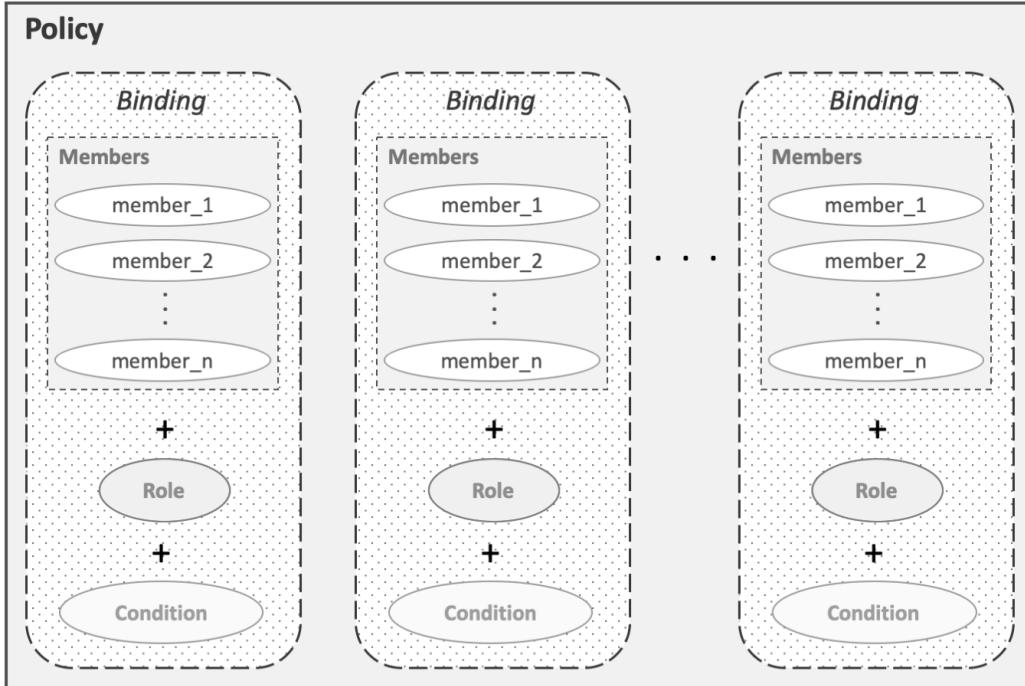


Figure 3: The Cloud IAM bindings (based on [28]).

hierarchy in Figure 2. That means that the administrator can set a policy at any level in the RESOURCE hierarchy (e.g., Organization, Folder, Project, Resource level), and the children’s resources of that level can automatically inherit it. RESOURCES always inherit the POLICIES of the parent RESOURCE, and the inheritance is transitive through the hierarchical path. Therefore, RESOURCES inherit the POLICIES of the Project; Projects inherit the POLICIES of the Folder, and Folders inherit the Organization’s POLICIES. At each level, the effective policies (i.e., in the presence of a hierarchy) are equal to the union of policies directly applied at the level and POLICIES inherited from its ancestors. For instance, a POLICY used in a Folder will also apply to Projects and RESOURCES under that Folder. Note that the POLICY hierarchy will change if the RESOURCE hierarchy is changed such that the PERMISSIONS that a child node inherited from its original parent will be lost and replaced by the PERMISSIONS set at the destination parent. $RBAC_{GCP}$ has no sessions. Instead, a ROLE remains dormant and not grantable if the respective SERVICE is not enabled. An administrator can use custom ROLES to enforce the principle of least privilege [36].

3.2. Model Definition

Following the notation used in the ANSI INCITS 359-2012 standard, we formally define the core $RBAC_{GCP}$ model as:

- $MEMBERS, ROLES, SERVICES, RESOURCES, VERBS, CONDITIONS$ are sets of members, roles, services, resources, verbs, and conditions, respectively;
- $BINDING$ is a binding, such as $BINDING \subseteq MEMBERS \times ROLES \times CONDITIONS$ is a many-to-many mapping relation of $MEMBERS, ROLES$ and $CONDITIONS$ assignment. $CONDITIONS$ are optional;
- $PERMISSIONS = 2^{(SERVICES \times RESOURCES \times VERBS)}$ is a set of permissions;
- $PA \subseteq PERMISSIONS \times ROLES$ is a many-to-many mapping of $PERMISSIONS$ to $ROLES$ assignment;
- $POLICIES \subseteq 2^{BINDING}$ is the set of policies, i.e., a single policy is a set of bindings.

4. Model and Properties Specification

This section elaborates on the model checking technique for verifying $RBAC_{GCP}$ policies. The process is compliant with NIST's guidelines [5]. Specifically, we define the $RBAC_{GCP}$ model using a transition system (TS). And verify example policies using temporal logic specifications for demonstration purposes.

4.1. A Transition System for $RBAC_{GCP}$

Model checking is a formal verification technique that can be applied to verify the correctness of models and detect faults in model specifications. It takes a finite-state model and checks it against specified properties expressed using temporal modalities, linear temporal operators, and path quantifiers. To achieve this, we define access control rules in a transition system for the $RBAC_{GCP}$, as follows.

Definition 1. An access control rule is an implication of $c \rightarrow d$, where constraint c is a predicate expression of the form:

$$(\bigvee MEMBER = mbrs) \wedge (\bigvee ROLE = role) \wedge (\bigvee PERMISSION = prms)$$

$\wedge (\bigvee RESOURCE = rscs)$ which when *true* implies the access control decision d , i.e., $decision = Grant$ or $decision = Deny$, where $mbrs \in MEMBERS$, $role \in ROLES$, $prms \in PERMISSIONS$, and $rscs \in RESOURCES$. The symbol of \bigvee denotes that more than one formula may be present, e.g., $\bigvee MEMBER = mbrs$ could be $MEMBER = mbrs_1 \vee MEMBER = mbrs_2 \vee \dots \vee MEMBER = mbrs_n$, where $mbrs_1, \dots, mbrs_n \in MEMBERS$.

Definition 2. An $RBAC_{GCP}$ access control property $prop$ is an implication formula of $\forall \square(c \rightarrow \forall \diamond d)$, where c is the cause and d is the effect (response property pattern). Both \square and \diamond are elementary temporal modalities for "always" and "eventually", respectively, and \forall means "for all paths" (Computation Tree Logic (CTL) semantics) [37].

Definition 3. The transition system TS for the $RBAC_{GCP}$ model is expressed as a tuple (S, Act, δ, i_0) where:

- S is a set of system states, $S = \{Grant, Deny\}$;
- Act is a set of actions, where $Act = \{((\bigvee MEMBER = mbrs) \wedge (\bigvee ROLE = role) \wedge (\bigvee PERMISSION = prms) \wedge (\bigvee RESOURCE = rscs) \rightarrow decision = Grant), \dots\}$
- δ is a transition relation, where $\delta : S \times Act \rightarrow S$;

- i_0 is the initial state, $i_0 = \{Deny\}$.

Access control rules define the system’s behavior, which functions as the transition relation δ in TS . In other words, a transition system specifies how a system can evolve from one state to another when the transition relation is applied, i.e., an action Act is performed on a state S to bring the system to the next state of S . To verify $RBAC_{GCP}$ access control properties using a temporal logic formula, we say that model TS satisfies $prop$ by $TS \models prop$ i.e., $TS \models \forall \Box(c \rightarrow \forall \Diamond d)$ from Definition 2.

4.2. Specification of Properties

The transition system describes the system’s behavior, which can be used for verifying properties [38]. The verification shows if the access control policy is correctly specified and according to the security requirements. Specifically, model checking performs exhaustive testing of all behaviors of the model. It can verify if the defined properties hold or not throughout the model’s behaviors (i.e., system states). In the $RBAC_{GCP}$ model, properties are expressed as (based on Definition 2; conditions are optional):

$$\begin{aligned} & \forall \Box((MEMBER = m \wedge ROLE = r \wedge \\ & \quad PERMISSION = prms \wedge \\ & \quad RESOURCE = rsrc \wedge \\ & \quad CONDITION = value) \rightarrow \\ & \quad \forall \Diamond(decision = Grant \vee Deny)) \end{aligned}$$

Different specifications can be expressed depending on the values used in the predicates forming the property above. Consequently, we can define several different logical representations of the response pattern property using the same CTL formula.

5. Verification of Example Policies

This section demonstrates using examples from Google’s Cloud IAM website [33] how to verify $RBAC_{GCP}$ policies. The examples show how the POLICY inheritance works in the Cloud IAM platform. We use these examples for their diversity in terms of used RESOURCES, MEMBER types, structural complexity, number of PERMISSIONS per ROLE, and level of a hierarchy of access control policy rules. The NuSMV code of all three examples are available on GitHub [39].

We assign values $m, r, prms, rsrc, value$ to the parameters $MEMBER, ROLE, PERMISSION, RESOURCE,$ and $CONDITION$, respectively, following the CTL formula in Section 4.2 to specify properties. $CONDITION$ is optional and not used in the examples. The ANY value is introduced for all variables as a wild card. The response property is written as: $AG(c \rightarrow AF(d))$, where G is an equivalent symbol used instead of \square , and F instead of \diamond . A represents the universal path quantifier \forall . So, we can rewrite access control properties in NuSMV as:

$$\begin{aligned} AG((& MEMBER = m \& ROLE = r \& \\ & PERMISSION = prms \& \\ & RESOURCE = rsrc \& \\ & CONDITION = value) \rightarrow \\ & AF(decision = Grant \mid Deny)). \end{aligned}$$

The model checker creates all system model states and evaluates whether the policy model satisfies the specified properties. If it does, there are no errors from the output of NuSMV. Otherwise, a counterexample is generated, which details why the model fails to satisfy a property.

5.1. Example 1: Cloud Pub/Sub

The first example [33] uses Cloud Pub/Sub RESOURCES, which are topics under a Project. As illustrated in Figure 4, *topic_a* resides in *project_a*. The Cloud IAM platform manages two Google accounts, i.e., *bob@gmail.com* and *alice@gmail.com*. We assume that the POLICY pl_1 is set on *project_a* to assign the ROLE of Editor (*roles/pubsub.editor*) to *bob@gmail.com* and POLICY pl_2 is set on *topic_a* to assign the ROLE of Publisher (*roles/pubsub.publisher*) to *alice@gmail.com*. Hence the two POLICIES that contain the rules are (based on Definition 1):

POLICY for *bob@gmail.com*:

$$\begin{aligned} pl_1 : MEMBER = "bob@gmail.com" \& \\ ROLE = "roles/pubsub.editor" \& \\ (PERMISSION = prms_1 \mid \dots \mid prms_n) \& \\ RESOURCE = "project_a" \rightarrow \\ (decision = Grant) \end{aligned}$$

POLICY for *alice@gmail.com*:

Table 1: Example 1 - Authorized roles per member and resource.

Resource	Authorized role	
	bob@gmail.com	alice@gmail.com
project_a	Editor	No access
topic_a	Editor	Publisher

$$\begin{aligned}
 pl_2 : MEMBER = "alice@gmail.com" \& \\
 ROLE = "roles/pubsub.publisher" \& \\
 (PERMISSION = prms_1 | \dots | prms_n) \& \\
 RESOURCE = "topic_a" \rightarrow \\
 (decision = Grant)
 \end{aligned}$$

As RESOURCES always inherit the POLICIES of the parent RESOURCE, *topic_a* inherits the POLICY from *project_a*. Hence, we introduce an additional POLICY pl'_1 for *topic_a* to assign the Editor ROLE *roles/pubsub.editor* to *bob@gmail.com*, as follows:

$$\begin{aligned}
 pl'_1 : MEMBER = "bob@gmail.com" \& ROLE = "roles/pubsub.editor" \& \\
 (PERMISSION = prms_1 | \dots | prms_n) \& RESOURCE = "topic_a" \rightarrow \\
 (decision = Grant)
 \end{aligned}$$

Ultimately the effective policy for *topic_a* will be the union of the POLICIES directly applied to *topic_a* and the POLICIES inherited from its ancestors.

The respective NuSMV code for the POLICIES and the transition system is available in Example 1 on GitHub [39]. As a result, the ROLE assignments for each MEMBER per RESOURCE are shown in Table 1.

After expressing POLICIES and the $RBAC_{GCP} TS$, the policy properties should be specified for verification in the model checker. When a specification is evaluated to be TRUE, there is no error to report, i.e., the specified property is satisfied by the model. On the other hand, when the specified property is not satisfied and evaluated to be FALSE, the model checker provides a counterexample to justify the result. For example, the NuSMV specification to check if *alice@gmail.com* has the publisher ROLE for *project_a* at the *Projects* hierarchy level is:

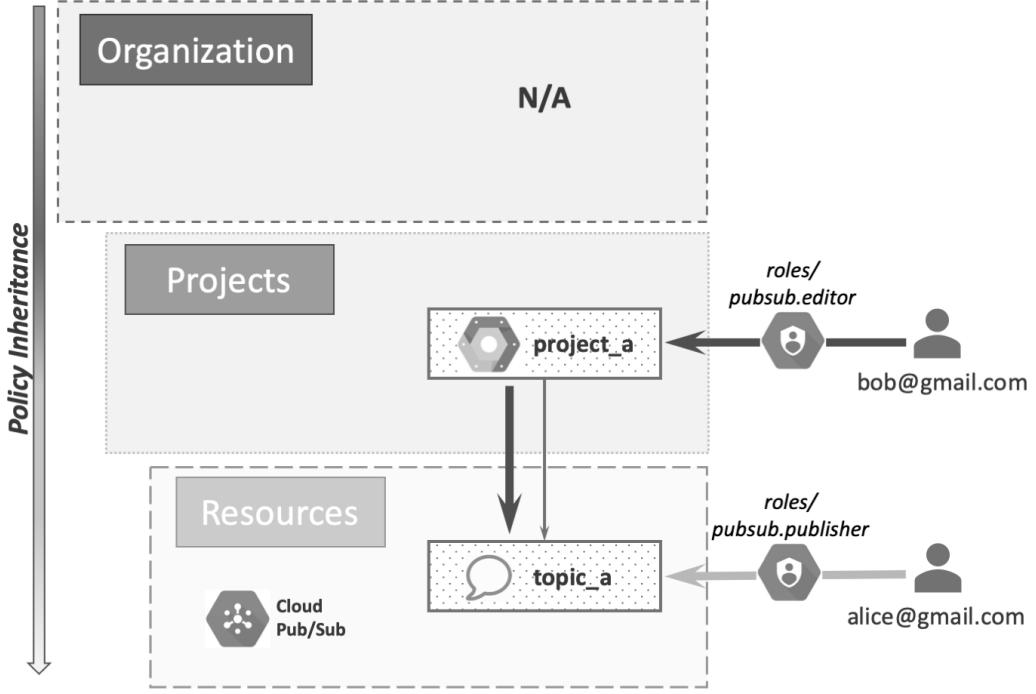


Figure 4: Example 1 - Cloud Pub/Sub (based on [33]).

```
SPEC AG ((MEMBER = "alice@gmail.com") &
(ROLE = "roles.pubsub.publisher") &
(PERMISSION = ANY) & (RESOURCE = "project_a") -> AF decision = Grant)
```

The above will be evaluated to be FALSE since *alice@gmail.com* is assigned to ROLE *Publisher* on *topic_a*, and according to $RBAC_{GCP}$ POLICY, she cannot access *project_a* because it resides in a higher level.

A NuSMV specification to check if *alice@gmail.com* has the *pubsub.topics.publish* PERMISSION on *project_a* at *Projects* hierarchy level can be written:

```
SPEC AG ((MEMBER = "alice@gmail.com") &
(ROLE = ANY) &
(PERMISSION = "pubsub.topics.publish") &
(RESOURCE = "project_a") -> AF decision = Grant)
```

The above will be evaluated to be FALSE since *alice@gmail.com* has the PERMISSION *pubsub.topics.publish*, for her *Publisher* ROLE only on *topic_a*, but not on *project_a* that resides on a higher level.

Lastly, to check if *alice@gmail.com* has the PERMISSION *pubsub.topics.delete* on *topic_a* at *Resources* hierarchy level, we write:

```
SPEC AG ((MEMBER = "alice@gmail.com") &
(ROLE = ANY) &
(PERMISSION = "pubsub.topics.delete") &
(RESOURCE = "topic_a") -> AF decision = Grant)
```

Although *alice@gmail.com* has the ROLE *Publisher* on *topic_a* she does not have the PERMISSION *pubsub.topics.delete* since an assignment is missing between that ROLE and the PERMISSION; hence, it is evaluated to be FALSE.

In all three specifications, the result of the verification is *RBAC.decision = Deny* without a next state, which indicates that they can never be satisfied, according to the *RBAC_{GCP}* POLICIES. The model checker could not find any system state where the property verified to be TRUE for the access permission *Grant* to happen.

5.2. Example 2: Cloud Storage

The second example [33] uses Cloud Storage RESOURCES called buckets. The bucket *upload_here* belongs to the Project *project_a* of the Organization *example.com* and is used to store files uploaded from GCP users (see Figure 5). Many users can use the same bucket to upload files; thus, it requires that no user can delete any of the files uploaded by other users. However, the data processing expert should be able to gain or delete anyone's files.

We assume that *alice@example.com* is the Google account of the data processing expert and *data_uploaders@example.com* is the group account of users who upload files to the bucket. The group has three MEMBERS: *jane@example.com*, *harry@example.com*, and *bob@example.com*. To achieve the security requirements, a POLICY is set on *project_a* to assign the Storage Object Admin ROLE (*roles/storage.objectAdmin*) to *alice@example.com*, and a second POLICY is set on *project_a* to assign the Storage Object Creator ROLE (*roles/storage.objectCreator*) to *data_uploaders@example.com*. These ROLES should allow *alice@example.com* to upload or delete any object in any bucket in *project_a*, while the MEMBERS of *data_uploaders@example.com* should be allowed to upload files. The two POLICIES will look as follows:

POLICY for *alice@example.com*:

$$\begin{aligned}
pl_1 : MEMBER = "alice@example.com" \& \\
ROLE = "roles/storage.objectAdmin" \& \\
(PERMISSION = prms_1 | \dots | prms_n) \& RESOURCE = "project_a" \rightarrow \\
(decision = Grant)
\end{aligned}$$

POLICY for data $data_uploaders@example.com$:

$$\begin{aligned}
pl_2 : (MEMBER = "data_uploaders@example.com" | "jane@example.com" | \\
"bob@example.com" | "harry@example.com") \& \\
ROLE = "roles/storage.objectCreator" \& \\
(PERMISSION = prms_1 | \dots | prms_n) \& RESOURCE = "project_a" \rightarrow \\
(decision = Grant)
\end{aligned}$$

POLICY pl_2 applies to $project_a$ for every group MEMBER, which assigns the Storage Object Creator ROLE to $jane@example.com$, $harry@example.com$ and $bob@example.com$, as well.

Bucket $upload_here$ inherits POLICIES from its parent RESOURCE $project_a$. POLICIES pl_1 and pl_2 will then be defined and populated to the transition system of the $RBAC_{GCP}$ model. Although the bucket has no defined POLICIES, these two POLICIES will apply on $upload_here$ (due to hierarchy) such that the Storage Object Admin ROLE is assigned to $alice@example.com$ on $upload_here$, and the Storage Object Creator ROLE is assigned to $data_uploaders@example.com$ for $upload_here$, as follows:

$$\begin{aligned}
pl'_1 : MEMBER = "alice@example.com" \& \\
ROLE = "roles/storage.objectAdmin" \& \\
(PERMISSION = prms_1 | \dots | prms_n) \& \\
RESOURCE = "upload_here" \rightarrow \\
(decision = Grant)
\end{aligned}$$

and

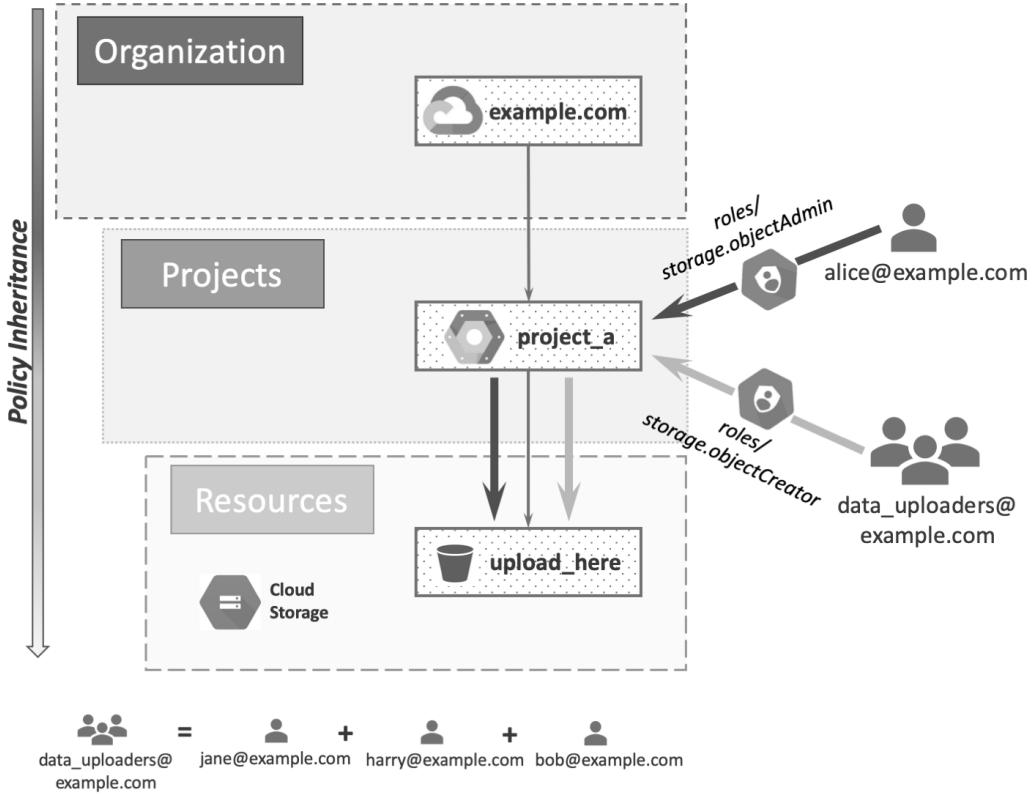


Figure 5: Example 2 - Cloud Storage (based on [33]).

```


$$\begin{aligned}
pl'_2 : (MEMBER = "data\_uploaders@example.com" \mid "jane@example.com" \mid \\
& "bob@example.com" \mid "harry@example.com") \& \\
ROLE = "roles/storage.objectCreator" \& \\
(PERMISION = prms_1 \mid \dots \mid prms_n) \& \\
RESOURCE = "upload\_here" \rightarrow \\
(decision = Grant)
\end{aligned}$$


```

Ultimately, the effective POLICIES at *project_a* and *upload_here* will be the union of the POLICIES directly applied to them and the POLICIES inherited from their ancestors.

Table 2 shows the ROLES assigned to each MEMBER per RESOURCE. The respective NuSMV code for the POLICIES and the transition system

Table 2: Example 2 - Authorized roles per member and resource.

Resource	Authorized role		
	alice@example.com	data_uploaders@example.com	
	jane	harry	bob
example.com	No access		No access
project_a	Storage Object Admin		Storage Object Creator
upload_here	Storage Object Admin		Storage Object Creator

is available on GitHub [39], under Example 2. After expressing POLICIES and the *TS* of the $RBAC_{GCP}$ in NuSMV, we specify the policy properties to be verified by the model checker. The following explains the evaluation of specifications.

Four of the example properties will be evaluated to be FALSE as follows.

```
SPEC AG ((MEMBER = "data_uploaders@example.com") &
          (ROLE = ANY) & (PERMISSION = "storage.objects.delete") &
          (RESOURCE = ANY) -> AF decision = Grant)
```

This property is FALSE since the group of *data_uploaders@example.com* does not have the permission *storage.objects.delete* on any RESOURCE.

```
SPEC AG ((MEMBER = "alice@example.com") &
          (ROLE = ANY) & (PERMISSION = ANY) &
          (RESOURCE = "example.com") -> AF decision = Grant)
```

This property was also evaluated to be FALSE since we assigned *alice@example.com* to the Storage Object Admin ROLE on *project_a*, and from the RESOURCE hierarchy, it has no access on *example.com* in a higher level.

```
SPEC AG ((MEMBER = ANY) & (ROLE = ANY) &
          (PERMISSION = "storage.objects.delete" |
           PERMISSION = "storage.objects.update" ) &
          (RESOURCE = "example.com") -> AF decision = Grant)
```

The above property is evaluated to FALSE since, according to $RBAC_{GCP}$ RESOURCE hierarchy, none of the MEMBERS have the PERMISSION *storage.objects.delete* or *storage.objects.update* on *example.com* because we assigned them to *project_a* that resides in a lower level.

```

SPEC AG ((MEMBER != "alice@example.com") &
          (ROLE = ANY) & (PERMISSION = "storage.objects.create") &
          (RESOURCE = ANY) -> AF decision = Deny)

```

This property is also FALSE since MEMBERS (different than *alice@example.com*) have the PERMISSION *storage.objects.create* on a RESOURCE at the RESOURCE hierarchy level. Group MEMBERS *data_uploaders@example.com* have PERMISSION for its assignment for the Storage Object Creator ROLE on *project_a* that resides at a higher level.

The verification of the first three specifications result is $RBAC.decision = Deny$ without a next state, which indicates that these properties can never be satisfied in the $RBAC_{GCP}$ model. The model checker NuSMV could not find any system state where the property would be evaluated to be TRUE so that it could eventually cause the access permission *Grant* to happen. Similarly, the verification of the fourth specification results in $RBAC.decision = Grant$; hence, it is invalidated too.

5.3. Example 3: Compute Engine

The third example [33] uses Compute Engine RESOURCES, which are virtual machines (VM) hosted on Google’s infrastructure. For this example, the organization *example.com*, owns two projects, *project_1* and *project_2*. And RESOURCES *instance_a* and *instance_b* belong to each project respectively, as illustrated in Figure 6. Assuming that *bob@example.com* is a MEMBER of the administrator’s team that manages the network and security RESOURCES of the Organization, and *alice@example.com* is a MEMBER of the development team. *bob@example.com* is capable of making changes to all network RESOURCES and any project under it, and *alice@example.com* should be allowed to launch instances and carry out other actions related to instances related to her project. Such security requirements are implemented by the POLICY on *example.com* that assigns the Compute Network Admin ROLE (*roles/compute.networkAdmin*) to *bob@example.com* and a second POLICY on *project_2* that assigns the Compute Instance Admin ROLE (*roles/compute.instanceAdmin*) to *alice@example.com*. The two POLICIES are:

POLICY for *bob@example.com*:

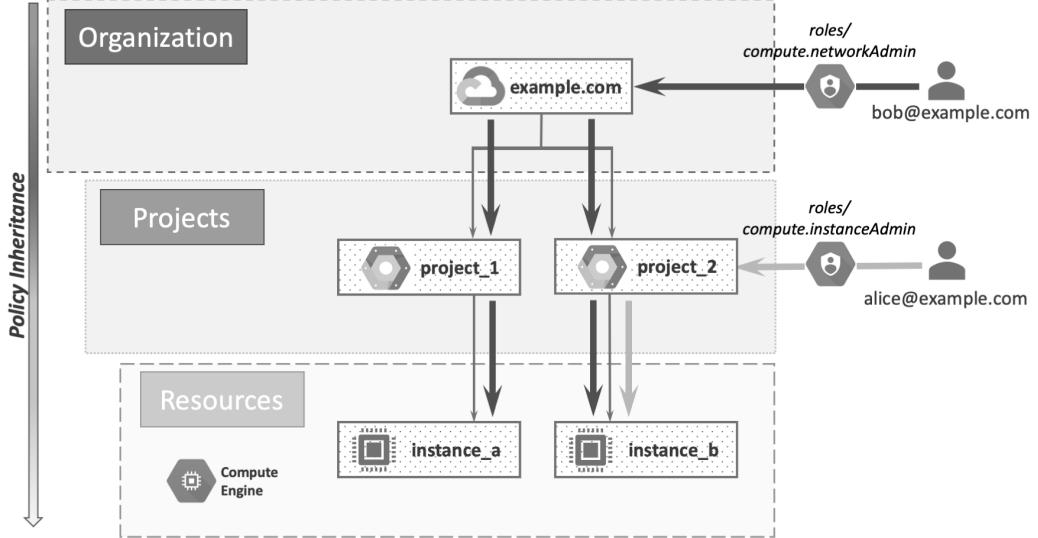


Figure 6: Example 3 - Compute Engine (based on [33]).

$pl_1 : MEMBER = "bob@example.com" \& ROLE = "compute.networkAdmin" \& (PERMISSION = prms_1 | \dots | prms_n) \& RESOURCE = "example.com" \rightarrow (decision = Grant)$

POLICY for $alice@example.com$:

$pl_2 : MEMBER = "alice@example.com" \& ROLE = "roles/compute.instanceAdmin" \& (PERMISSION = prms_1 | \dots | prms_n) \& RESOURCE = "project_2" \rightarrow (decision = Grant)$

Since $project_1$ and $project_2$ inherit the POLICIES of $example.com$, once we define POLICY pl_1 , we introduce the following POLICIES for the Compute Network Admin ROLE ($roles/compute.networkAdmin$) to be assigned to $bob@example.com$ on $project_1$ and $project_2$, as follows:

POLICY for $bob@example.com$ on $project_1$:

$$\begin{aligned}
pl_{1.1} : MEMBER = "bob@example.com" \& \\
ROLE = "roles/compute.networkAdmin" \& \\
(PERMISISON = prms_1 | \dots | prms_n) \& \\
RESOURCE = "project_1" \rightarrow \\
(decision = Grant)
\end{aligned}$$

POLICY for *bob@example.com* on *project_2*:

$$\begin{aligned}
pl_{1.2} : MEMBER = "bob@example.com" \& \\
ROLE = "roles/compute.networkAdmin" \& \\
(PERMISISON = prms_1 | \dots | prms_n) \& \\
RESOURCE = "project_2" \rightarrow \\
(decision = Grant)
\end{aligned}$$

RESOURCES *instance_a* and *instance_b* also inherit their parent resources' POLICY *project_1* and *project_2*, respectively. The Compute Network Admin ROLE (*roles/compute.networkAdmin*) is assigned to *bob@example.com* on *instance_a* and *instance_b*, and the Compute Instance Admin ROLE (*roles/compute.instanceAdmin*) is assigned to *alice@example.com* only on *instance_b*. The introduced POLICIES are:

POLICY for *bob@example.com* on *instance_a*:

$$\begin{aligned}
pl'_{1.1} : MEMBER = "bob@example.com" \& \\
ROLE = "roles/compute.networkAdmin" \& \\
(PERMISISON = prms_1 | \dots | prms_n) \& \\
RESOURCE = "instance_a" \rightarrow \\
(decision = Grant)
\end{aligned}$$

POLICY for *bob@example.com* on *instance_b*:

$$\begin{aligned}
pl'_{1.2} : MEMBER = "bob@example.com" \& \\
ROLE = "compute.networkAdmin" \& \\
(PERMISISON = prms_1 | \dots | prms_n) \& \\
RESOURCE = "instance_b" \rightarrow \\
(decision = Grant)
\end{aligned}$$

POLICY for *alice@example.com* on *instance_b*:

Table 3: Example 3 - Authorized roles per member and resource

Resource	Authorized role	
	bob@example.com	alice@example.com
example.com	Compute Network Admin	No access
<i>project_1</i>	Compute Network Admin	No access
<i>project_2</i>	Compute Network Admin	Compute Instance Admin
<i>instance_a</i>	Compute Network Admin	No access
<i>instance_b</i>	Compute Network Admin	Compute Instance Admin

$$\begin{aligned}
 pl'_2 : MEMBER = "alice@example.com" \& \\
 ROLE = "roles/compute.instanceAdmin" \& \\
 (PERMISSION = prms_1 | \dots | prms_n) \& \\
 RESOURCE = "instance_b" \rightarrow \\
 (decision = Grant)
 \end{aligned}$$

Ultimately, the effective POLICIES on every RESOURCE are the union of the POLICIES directly applied to the RESOURCE and the POLICIES inherited from its ancestors.

Table 3 shows the ROLES assigned to each MEMBER per RESOURCE.

The NuSMV code for the properties specification of this example is available on GitHub [39], under Example 3.

```

SPEC AG ((MEMBER = "alice@example.com") &
          (ROLE = ANY) &
          (PERMISSION = "compute.instances.create") &
          (RESOURCE = "project_1") -> AF decision = Grant)

```

This property will be evaluated to FALSE since *alice@example.com* has the PERMISSION *compute.instances.create* assigned to the Compute Instance Admin ROLE, but not on *project_1* that resides in a different branch of the RESOURCE hierarchy. ROLES do not affect peer RESOURCES.

```

SPEC AG ((MEMBER = ANY) &
          (ROLE = "roles/compute.instanceAdmin") &
          (PERMISSION = ANY) & (RESOURCE = "instance_a") -> AF decision = Grant)

```

This property is FALSE since the Compute Instance Admin ROLE is assigned to *instance_b*.

```

SPEC AG ((MEMBER = ANY) & (ROLE = ANY) &
(PERMISSION = "compute.instances.create") &
(RESOURCE = "project_1") -> AF decision = Grant)

```

We have that *bob@example.com* has access to *project_1*, but his ROLE (Compute Network Admin) does not contain that specific PERMISSION. And *alice@example.com* has this PERMISSION because of her assigned ROLE (Compute Instance Admin) on *project_2*, but not on *project_1* in a different branch of the RESOURCE hierarchy. Hence, the property is FALSE since no one has the PERMISSION *compute.instances.create* on *project_1*.

In all three specifications, the result of the verification is *RBAC.decision = Deny*, without a next state since the NuSMV model checker could not find any system state where the properties is TRUE.

5.4. Summary of Examples

The first example used Cloud Pub/Sub RESOURCES and presented a case of RESOURCE hierarchy between a Project and a topic. We considered two different POLICIES for two MEMBERS, one on each RESOURCE. This example demonstrates how the *TS* operates, and how properties are specified to check whether the hierarchy was implemented correctly. The second example used Cloud Storage RESOURCES to demonstrate the enforcement of two different POLICIES for two MEMBERS on the same RESOURCE. One of the MEMBERS is a Google group account that allowed us to investigate how the applied technique handles this type of a MEMBER. Google groups are a convenient way to apply organization access control policies and a best practice for role distribution [33]. The third example used Compute Engine RESOURCES, which allowed us to evaluate the security policies in a more complex configuration where the resource structure contains more branches and nodes. Various properties in each example were checked to satisfy specific security requirements in compliance with Google’s proposed best practices [33]. Overall, the applied technique successfully verified the properties in all three examples; hence, offering the capability of a tool for administrators to specify policies/properties and verify their correctness.

6. Conclusion

When defining policies in Cloud systems, it is imperative to understand the underlying access control model and supported policies to avoid configuration errors or even inconsistencies. Towards achieving this aim, we defined

$RBAC_{GCP}$ to provide a better understanding of the RBAC model and policies supported by the Google Cloud IAM platform. The RBAC access control model of Cloud IAM has a few differences compared to the ANSI standard model. Specifically, the former supports permission inheritance through RESOURCE hierarchies but not ROLE hierarchies. We applied model checking to formally verify supported access control policies. And we demonstrated the technique’s applicability through three examples described on the official Google Cloud IAM website. We anticipate this work to assist system administrators in ensuring the correctness of policy specification and checking violations against security requirement [6] and, even further, performing a security assessment of policies for compliance purposes [40].

Acknowledgement

The authors would like to thank Dr Andrew Sogokon at Lancaster University for his feedback. This research is supported in part by the Security Lancaster VERIFI Mini-Project under grant number IRL1025.

References

- [1] R. Sandhu, P. Samarati, Access Control: Principle and Practice, IEEE communications magazine 32 (9) (1994) 40–48.
- [2] D. Ferraiolo, D. R. Kuhn, R. Chandramouli, Role-based access control, Artech House, 2003.
- [3] V. Hu, M. Iorga, W. Bao, A. Li, Q. Li, A. Gouglidis, General Access Control Guidance for Cloud Systems, Tech. rep., National Institute of Standards and Technology (2020).
- [4] L. Hadarean, One-click formal methods, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 1–1. doi:10.1109/ICST46399.2020.00009.
- [5] V. C. Hu, R. Kuhn, D. Yaga, Verification and test methods for access control policies/models, NIST Special Publication 800 (2017) 192.
- [6] A. Gouglidis, I. Mavridis, V. C. Hu, Security policy verification for multi-domains in cloud systems, International Journal of Information Security 13 (2) (2014) 97–111.

- [7] Y. Zhang, R. Krishnan, F. Patwa, R. Sandhu, Access control in cloud IaaS, Security, Privacy, and Digital Forensics in the Cloud (2019) 81.
- [8] D. Power, M. Slaymaker, A. Simpson, On the modelling and analysis of amazon web services access policies, in: M. Frappier, U. Glässer, S. Khurshid, R. Laleau, S. Reeves (Eds.), Abstract State Machines, Alloy, B and Z, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 394–394.
- [9] A. Evangelidis, D. Parker, R. Bahsoon, Performance modelling and verification of cloud-based auto-scaling policies, Future Generation Computer Systems 87 (2018) 629–638.
- [10] A. Sleit, N. Misk, F. Badwan, T. Khalil, Cloud computing challenges with emphasis on Amazon EC2 and windows azure, International Journal of Computer Networks & Communications 5 (5) (2013) 35.
- [11] G. Tajadod, L. Batten, K. Govinda, Microsoft and Amazon: A comparison of approaches to cloud security, in: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, IEEE, 2012, pp. 539–544.
- [12] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, M. C. Tschantz, Verification and change-impact analysis of access-control policies, in: Proceedings of the 27th international conference on Software engineering, ACM, 2005, pp. 196–205.
- [13] V. C. Hu, D. R. Kuhn, T. Xie, Property verification for generic access control models, in: 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, Vol. 2, IEEE, 2008, pp. 243–250.
- [14] A. Li, Q. Li, V. C. Hu, J. Di, Evaluating the capability and performance of access control policy verification tools, in: MILCOM 2015-2015 IEEE Military Communications Conference, IEEE, 2015, pp. 366–371.
- [15] M. Aqib, R. A. Shaikh, Analysis and comparison of access control policies validation mechanisms, IJ Computer Network and Information Security 1 (1) (2015) 54–69.

- [16] V. C. Hu, D. Ferraiolo, D. R. Kuhn, Assessment of access control systems, US Department of Commerce, National Institute of Standards and Technology, 2006.
- [17] G. Hughes, T. Bultan, Automated verification of access control policies using a SAT solver, International journal on software tools for technology transfer 10 (6) (2008) 503–520.
- [18] T. S. Hoang, D. Basin, J.-R. Abrial, Specifying access control in event-b, Technical report 624 (2009).
- [19] J. Hwang, T. Xie, V. Hu, M. Altunay, ACPT: A tool for modeling and verifying access control policies, in: 2010 IEEE International Symposium on Policies for Distributed Systems and Networks, IEEE, 2010, pp. 40–43.
- [20] K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, S. Chapin, Automatic error finding in access-control policies, in: Proceedings of the 18th ACM conference on Computer and communications security, ACM, 2011, pp. 163–174.
- [21] D. Jackson, Alloy: a language & tool for relational models, <https://alloytools.org> (Accessed: June 2022).
- [22] E. Martin, J. Hwang, T. Xie, V. Hu, Assessing quality of policy properties in verification of access control policies, in: 2008 Annual Computer Security Applications Conference (ACSAC), IEEE, 2008, pp. 163–172.
- [23] InfoBeyond Technology LLC, Security Policy Tool: Access Control Security Policy Editing, Testing, Verification, and XACML Deployment, <https://securitypolicytool.com/Content/files/Technical-overview.pdf> (Accessed: June 2022).
- [24] NIST, Access control policy testing, <https://csrc.nist.gov/Projects/Access-Control-Policy-Tool/ACPT> (Accessed: June 2022).
- [25] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuXmv symbolic model checker, in: International Conference on Computer Aided Verification, Springer, 2014, pp. 334–342.

- [26] NIST, Combinatorial testing, <https://www.nist.gov/programs-projects/combinatorial-testing> (Accessed: June 2022).
- [27] Google, Cloud Cloud Platform - Security Products - Products and capabilities, <https://cloud.google.com/security/products/> (Accessed: January 2020).
- [28] Google, Cloud Cloud Platform - Security Products - Overview, <https://cloud.google.com/iam/docs/overview> (Accessed: January 2020).
- [29] A. INCITS, INCITS 359-2004. role-based access control, American National Standard for Information Technology (2017).
- [30] Google, Understanding IAM custom roles, <https://cloud.google.com/iam/docs/understanding-custom-roles> (Accessed: December 2019).
- [31] Google, Cloud Cloud Platform - Security Products - Understanding roles, <https://cloud.google.com/iam/docs/understanding-roles> (Accessed: January 2020).
- [32] Google, Cloud Cloud Platform - Security Products - Resource hierarchy, <https://cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy> (Accessed: January 2020).
- [33] Google, Cloud Cloud Platform - Security Products - Using resource hierarchy for access control, <https://cloud.google.com/iam/docs/resource-hierarchy-access-control> (Accessed: January 2020).
- [34] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, et al., Guide to attribute based access control (abac) definition and considerations (draft), NIST special publication 800 (162) (2013) 1–54.
- [35] Google, Cloud Cloud Platform - Security Products - Overview of Cloud IAM Conditions, <https://cloud.google.com/iam/docs/conditions-overview> (Accessed: January 2020).
- [36] Google, Cloud Cloud Platform - Security Products - Frequently asked questions, <https://cloud.google.com/iam/docs/faq> (Accessed: January 2020).

- [37] E. M. Clarke, B.-H. Schlingloff, Chapter 24 - model checking, in: A. Robinson, A. Voronkov (Eds.), *Handbook of Automated Reasoning*, *Handbook of Automated Reasoning*, North-Holland, Amsterdam, 2001, pp. 1635–1790. doi:<https://doi.org/10.1016/B978-044450813-3/50026-6>.
URL <https://www.sciencedirect.com/science/article/pii/B9780444508133500266>
- [38] C. Baier, J.-P. Katoen, *Principles of model checking*, MIT press, 2008.
- [39] A. Gouglidis, A. Kagia, Google Cloud IAM NuSMV examples, <https://github.com/agouglidis/GoogleIAM-NuSMV> (Accessed: May 2022).
- [40] M. Rochwell, Google cloud gets FedRAMP high tag, <https://fcw.com/articles/2019/12/06/google-cloud-fedramp-high.aspx> (Accessed: December 2019).