Assisted Authoring, Analysis and Enforcement of Access Control Policies in the Cloud

Umberto Morelli, Silvio Ranise

Assisted Authoring, Analysis and Enforcement of Access Control Policies in the Cloud

Umberto Morelli(✉) and Silvio Ranise

Fondazione Bruno Kessler, via Sommarive 18, 38123 Trento, Italy
{umorelli,ranise}@fbk.eu

Abstract. The heterogeneity of cloud computing platforms hinders the proper exploitation of cloud technologies since it prevents interoperability, promotes vendor lock-in and makes it very difficult to exploit the well-engineered security mechanisms made available by cloud providers. In this paper, we introduce a technique to help developers to specify and enforce access control policies in cloud applications. The main idea is twofold. First, use a high-level specification language with a formal semantics that allows to answer access requests abstracting from an access control mechanism available in a particular cloud platform. Second, exploit an automated translation mechanism to compute (equivalent) policies that can be enforced in two of the most widely used cloud platforms: AWS and Openstack. We illustrate the technique on a running example and report our experience with a prototype implementation.

Keywords: Policy Translation and Validation, Attribute-based Access Control, Amazon AWS, OpenStack

1 Introduction

Cloud computing platforms offer companies the opportunity to create applications that have global reach and can scale rapidly to meet sudden spikes in demand without requiring massive investments by adopting a pay-as-you-go approach. The cost of this extra flexibility is a loss of control over the software components deployed in the cloud and the data manipulated by applications, since part of the responsibility and control is transferred from Cloud Customers (CCs) to Cloud Providers (CPs); consider, e.g., the "Amazon Web Service Shared Responsibility Model"[1]. When using a cloud platform, it is crucial for CCs to understand and distinguish between security measures implemented and operated by CPs (called, security *of* the cloud) and those offered by them (called, security *in* the cloud), for which the CPs are accountable. Failure to understand the boundaries of this separation of concerns and responsibilities may lead to leave sensitive assets unprotected with the potential of disclosing sensitive information, thereby incurring in extra costs and potential loss of business, and eliminating many of the benefits of cloud computing. Even if the separation of

[1] https://aws.amazon.com/compliance/shared-responsibility-model

responsibilities is clear, CCs may find it difficult to use effectively the large array of security mechanisms provided by the specific CP. This hinders one the most important opportunities offered by CPs to CCs, namely the exploitation of the cornucopia of well-engineered security mechanisms made available by CPs.

In order to alleviate this situation, we propose a technique capable of assisting CCs in designing and deploying access control systems in two of the most widely popular cloud platforms: AWS[2] and OpenStack[3]. Access Control (AC) is one of the most important security mechanisms for the protection of data and services against unauthorized disclosure (confidentiality) and intentional or accidental unauthorized changes (integrity), while ensuring their accessibility by authorized users whenever needed (availability). The development of an AC system requires the definition of the regulations according to which access is to be controlled and their implementation as functions executable by a computer system. This development process is usually carried out with a multi-phase approach based on the concepts of policy, model, and enforcement mechanism [4]. A policy defines the (high-level) rules according to which access control must be regulated. A model provides a formal representation of the AC policy and its working. The formalization allows the proof of properties on the security provided by the AC system being designed. An enforcement mechanism defines the low level functions that implement the controls imposed by the policy and formally stated in the model. In a cloud computing platform, several enforcement mechanisms are available, ranging from access control lists to those based on roles [4]. Many of these enforcement mechanisms are of a low level nature or are variant of the standards as they are tightly coupled with the resources and operations that services made available by the CP support. For application developers, it is not easy to grasp how all the different enforcement mechanisms work and how they can be used to mediate access to the data and services that the application under development is using. In many cases, even security experts may have difficulties in expressing high-level AC constraints related to an application (e.g., Separation of Duties) by means of the enforcement mechanisms available by CPs.

The main contribution of the paper is a technique that allows application developers and security experts to design the application and the AC policies by using an abstract model of a cloud platform without committing to a particular cloud solution. Since the language in which the AC policies are written has a formal semantics, it is possible to re-use automated tools for the security analysis of policies to understand whether the written policies correspond to the designer expectations. When this is the case, the tool automatically translates the high-level AC rule into concrete policies that can be enforced by the mechanisms available in AWS and Openstack.

Plan of the paper. Sec. 2 introduces a scenario that illustrates the main problems underlying the development of secure applications in the cloud. Sec. 3 describes our high-level policy specification language and its formal semantics by using a logical framework. Sec. 4 explains how the policies written in the high-level

[2] `https://aws.amazon.com`
[3] `https://www.openstack.org`

language can be translated to the policies that can be enforced by the AC mechanisms available in AWS and Openstack. Sec. 5 shows how a prototype implementation of our techniques (called SecurePG) solves the problems arising in the running example of Sec. 2. Sec. 5.2 presents some concluding remarks and a short comparison with related work.

2 A Running Example

The ACME shipping company wants to develop a cloud application to support a customer loyalty program (SpecialDiscounts). The idea is to reward e-payments made via a mobile application (PromoApp) with virtual credits that can be spent for additional ACME services or discounts on selected products offered by ACME Partners. To this end, ACME wants to grant the partners of the loyalty program access to a restricted set of information through the application, while maintaining control over customers' data; thereby configuring two different domains in the data storage services available in the cloud.

Figure 1 shows the ACME and the cloud domains, together with three groups of users: ACME Customers, ACME Employees and ACME Partners. ACME Employees, using a system in the ACME domain, can list customers profiles (label L), extract the information they contain (label G), add new profiles (label P) or delete existing ones (label D); those operations are represented by the labelled solid arrow from ACME Employees to the ellipse named 'Full ACME Customer Profiles'. ACME Customers, by using PromoApp, can get, add or delete the information stored in the partial customer profiles (labels G, P and D linked to the arrow connecting ACME Customers to the ellipse named 'Partial ACME Customer Profiles'). The same operations are performed on their full profiles by using a system in the ACME domain. ACME Partners, using the SpecialDiscounts application, can list the partial ACME customer profiles (label L on the arrow connecting ACME Partners to the ellipse named 'Partial ACME customer profiles') and can get or add information to the profile (labels G and P linked to the arrow connecting ACME Partners to the ellipse named 'Partial ACME Customer Profiles'). Since the full and partial ACME customer profiles can be updated independently (by using the cloud application or the system in the ACME domain), it should be possible to synchronize the information stored in both profiles (double arrow named 'Synch') so as to keep them up-to-date. The goal is to deploy the two applications, SpecialDiscounts and PromoApp, on a cloud computing platform while guaranteeing that the members of the various groups can perform only the actions discussed above. It may be also important to consider public or private cloud solutions, depending on the fact that sensitive information in the customer profiles must be stored also in the partial profiles managed by the cloud applications. For instance, it must be possible to deploy the applications on a public cloud—such as an Amazon AWS Platform-as-a-Service (PaaS) implementation, using the Simple Storage S3 service to manage customer profiles—or on a private cloud—such as an OpenStack Infrastructure-as-a-Service (IaaS) installation within ACME, that uses Swift as

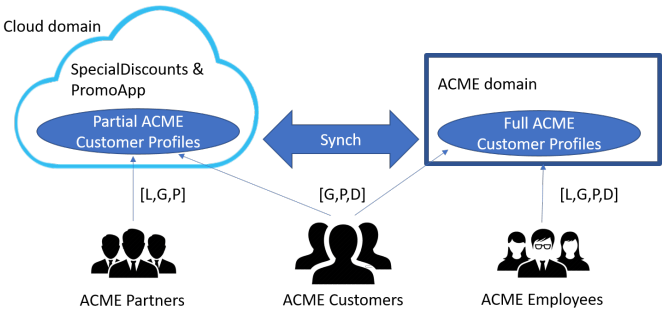

Fig. 1. Simplified architecture of the SpecialDiscounts and PromoApp cloud apps

the data container. The main requirement is to do all this by supporting a cloud provider-agnostic specification of the access control policies that permit the automated verification of basic security properties (e.g., a member of a certain group can/cannot perform a certain action on a given resource) and their automatic instantiation to the access control mechanisms available in a particular cloud solution. In this way, it is possible to manage the heterogeneity of the multitude of IaaS and PaaS solutions currently available by increasing interoperability and avoiding vendor lock-in while exploiting to the full the well-engineered security mechanisms available in different cloud solutions. While there exist approaches in the literature that allow to manage cloud applications across different platforms (see, e.g., [6]), none of these address security issues (and in particular access control policies) as we do in this paper. For this reason, in the following, we discuss only the issues related to access control while we point the interested reader to, e.g., [6] for an approach abstracting away from the functionalities and storage capabilities of a particular cloud provider.

3 An Abstract Access Control Model for the Cloud

Since Attribute-Based Access Control (ABAC) [7] offers a powerful and unifying extension to several access control models in the literature (see [8] for a thorough discussion about the expressive power of ABAC with respect to other models), we have chosen it as the framework in which to develop our policy language and access control model for cloud applications. In ABAC, requesters are permitted or denied access to a resource based on the properties, called *attributes*, that may be associated to subjects, resources, and contextual information. Suitably defined attributes can represent identities, access control lists, or roles; in this sense, ABAC supplements rather than supplant traditional access control models [8]. Abstractly, policies in ABAC are conditions on the attribute values of the entities involved in an access decision. Our policy language for the cloud is based on the following construct:

Listing 1.1. Abstract policy specification construct

```
Grant|Deny SUBJECTS [ATTRIBUTES] the permission to ACTIONS on RESOURCES
    [ATTRIBUTES] if CONDITIONS;
```

where the parts in black are mandatory and those in gray are optional. Intuitively, the meaning is to grant or deny to a subject (described by an identifier plus, optionally, some simple conditions on its attributes) the permission to perform some action on a resource (also described by an identifier plus some simple conditions on its attributes) provided that all the complex authorization conditions on the attributes of subjects, resources, or the context are satisfied (the meaning of simple and complex conditions will be made precise below).

A subject, a resource, or an action are identified by a unique name. The attributes of a subject, a resource, or the context are also identified by a unique name and each one is associated to a type, such as the Booleans, the Integers, or an enumerated data type. Formally, a subject, a resource, or the context can be seen as records whose (typed) fields are the attributes; a type defines a set of values plus some functions and predicates (including at least the equality $=$ operator) that can be applied to the values. A *simple condition* on a subject or on a resource can be expressed as a list (intended conjunctively) of equalities of the form $att = val$ where att is an attribute and val is one of its possible values. A *complex condition* is a Boolean combination of atomic expressions containing attributes of subjects, resources, or the context together with values and operators of the appropriate type (notice that we forbid quantifiers in complex conditions; this restriction in expressiveness was never a hindrance to specify policies in our experience). An example of a policy specification relevant to the example of Section 2 is provided below:

Listing 1.2. Example of an abstract rule that translates to a role policy

```
Grant ACME_employees the permission to add user to group and remove user
    from group on ACME_customers
```

This grants the ACME employees (identified by the role ID `ACME_employees`) the right to add and remove users from the group of ACME customers (`ACME_customers`). As another example, consider the following user policy:

```
Grant ACME_user_1 the permission to get object on ACME_user_1_profile if
    access time greater than 1451606400 and if access time less than
    1451779200
```

that grants the ACME user identified by the ID `ACME_user_1` the permission to access his profile (`ACME_user_1_profile`), e.g. to check the number of virtual credits, provided that this is done in a given period of (Unix) time, namely from the first to the third of January 2016.

Following [1], it is possible to formalize the meaning of the policy constructs in Listing 1.1 by using first-order logic formulas. For this, we preliminary introduce the notion of query as a tuple (sl, a, rl, cl) where a is an action and sl, rl, cl are simple conditions involving the attributes and values of a subject, a resource,

and the context, respectively. We write $\langle sl \rangle$, $\langle rl \rangle$, $\langle cl \rangle$ to denote the conjunction of equalities in the simple conditions sl, rl, and cl, respectively. A complex authorization condition in a policy construct can be considered as a first-order formula in which quantifiers does not occur. We assume the availability of the attributes *sid* and *rid* of subjects and resources, respectively, that range over their sets of identifiers.

Given a finite set Π of policy constructs of the form 1.1 and a theory T formalizing the types of the attributes in P (it is well-known how to do this, we point the interested reader to [1] for details), we say that a query $(sl, \mathtt{a}, rl, cl)$ is granted (with respect to Π) iff there exists a policy construct

```
Grant s [sA] the permission to a on r [rA] if C
```

in Π such that the formula $sid = \mathtt{s} \wedge \langle \mathtt{sA} \rangle \wedge rid = \mathtt{r} \wedge \langle \mathtt{rA} \rangle \wedge \mathtt{C}$ in conjunction with $\langle sl \rangle \wedge \langle rl \rangle \wedge \langle cl \rangle$ is satisfiable in T (i.e. there exists a first-order structure which is a model of T and satisfies both formulae) and there is no policy construct

```
Deny s' [sA'] the permission to a on r' [rA'] if C'
```

in Π such that the formula $sid = \mathtt{s'} \wedge \langle \mathtt{sA'} \rangle \wedge rid = \mathtt{r'} \wedge \langle \mathtt{rA'} \rangle \wedge \mathtt{C'}$ in conjunction with $\langle sl \rangle \wedge \langle rl \rangle \wedge \langle cl \rangle$ is satisfiable in T. Otherwise, we say that the query is denied.

The decidability and NP-completeness of the satisfiability checks with respect to the theory T follow from results in [1] when the types of the attributes are Booleans, Integers or enumerated data types. We do not elaborate the details here for lack of space; we just observe that complex conditions with arbitrary Boolean structure makes the problem already NP-hard because the Boolean satisfiability problem is subsumed. Indeed, NP-completeness of the induced satisfiability problems implies that also answering queries is NP-complete. This should not be seen as a hindrance to the usability of our approach. SMT engines solving the generated satisfiability problems guarantees the practical viability of the technique at policy design-time with queries solved in few seconds. One reason for the good practical performances is the relative simplicity of the Boolean structure of complex conditions.

4 From Abstract to Enforceable Policies in the Cloud

Cloud providers are not able to fully support the complexity of the ABAC model and the granularity required for handling an arbitrary list of attributes: the ability to scale while maintaining data integrity and authorizations evaluation performance allows for simple AC policies based only on the identity of subjects (that request a cloud resource). Those uses a basic set of user attributes, i.e. his name, role or the group he belongs and, if supported, further restrict requester permissions with a set of environment conditions.

Using the model introduced in Section 3 it is possible to extend the identity-centric approach using generic conditions and providing attributes for the subjects and the resources. Table 1 shows the authorization patterns to explicitly

suggest our prototype implementation the entity types (using their identifier and the type attribute) and the cloud attributes that uniquely identify them in Amazon and OpenStack: a subject identifier (ID), the URL or the Amazon Resource Name (ARN) code for the former and the ID or email of the subject for the latter. The current version supports three types of subjects (users, groups, roles) and two types of resources (objects and folders). Other components refer to the ability to authorize a service (handled as a role), support the identity federation features or create special policies (*type* keys or trust). If the end-user provides only the entity name, its type is retrieved querying a database; if the type is not supported (missing cases in the table) the information is instead ignored. Similarly, the attributes of subjects and resources, together with the environment conditions, are identified (using their name) and processed only if supported by the specific CP. This process, although not expanding the AC model of the supported CPs, greatly simplifies the task of writing AC policies. Moreover it allows the tool to create valid AC rules and easily supports the pure RBAC model of OpenStack and the RBAC-oriented implementation of AWS, with the possibility of future developments when more complex AC models will be made available by CPs.

Table 1. Authorization patterns and cloud attributes that identify the entity types

Authorization patterns		AWS	OpenStack
Subject component	UserID [type = user_subject]	ID and ARN	ID and email
	GroupID [type= group]	ARN	ID
	RoleID [type = role]	ID and ARN	ID and email
	ServiceID [type = service]	URL	missing
	FederatedID [type = federated]	URL or ARN	missing
Resource component	ObjectID[type = object]	ARN	ID
	FolderID[type = folder]	ARN	ID
	ResourceID[type = keys]	ARN	missing
	ResourceID[type = trust]	ARN	missing

4.1 Reconstruction of the Amazon and OpenStack AC Model

Figure 2 links the elements associated to the subjects in Amazon and OpenStack AC models, highlighting with dashes those that belong only to OpenStack and with the crosses those of AWS; solid lines represent instead common components and are among the ones supported by our prototype implementation.

The picture shows that both solutions present an administrative boundary, called *domain* in OpenStack and *root account* in AWS, that contains all the supported entities: Users, Groups, Roles and, exclusively for OpenStack, *projects* and *tokens*. Users may belong to a group (*User-Group* assignment) or be linked to a role (*User-Role* assignment), that AWS considers as a separate complex entity (with its own set of permissions) while OpenStack as a mandatory simple

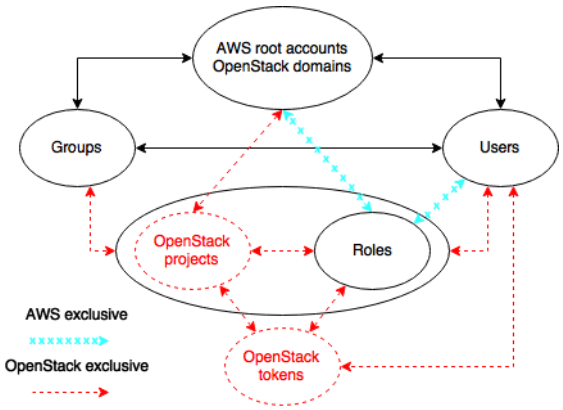

Fig. 2. Amazon AWS and OpenStack AC models. Adapted from [13, 12]

property of U. OpenStack requires the user to interact with a set of resources with specific roles (*User-Project* assignment) and providing a valid authentication *token*; the same process allows the interaction for all the entities that belongs to a group (*Group-Project* assignment).

4.2 Policy Support in Amazon and OpenStack

Our AC solution for the cloud supports six categories of authorization policies: assigned to the subject (User Policy or **UP**), to a users group or a role (respectively Group Policy or **GP** and Role Policy or **RoP**) and to the resources on which the subjects perform their actions (Resource Policy or **ReP**). The GP and RoP policies can also be specialized to apply only to a single user belonging to the specified group or role; in the following, these will be called special GP (**sGP**) and special RoP (**sRoP**). Two other types of permissions refer to the ability to offer the identity federation feature, which requires a trust relationship between users (Trust Policy or **TP**), and the possibility to assign permissions together with login credentials (Credential Policy or **CredP**). Those are supported only in Amazon and, when generating the OpenStack AC rules, are handled as UPs.

Table 2 provides an outline of our policy types and their support in the two CPs, including the required attributes or the services that enforce them. Amazon implement almost all types of authorization through the Identity and Access Management (IAM) service, while the Security Token Service (STS) is used for handling the CredP and three other services allow the user to specify ReP directly associated with the resource involved: S3 and the notification and queue services (respectively Simple NS and Simple QS). OpenStack instead manages permissions only through Keystone, using a set of rules related to the action performed (API actions) or the OpenStack service involved (such as *identity*, *network* or *compute*). The concept of TP and CredP is not supported in OpenStack and the only way to implement ReP, in the case of Swift, is to provide an

Table 2. Policy types supported in our model, Amazon AWS and OpenStack

Policy types	Amazon AWS implementation	OpenStack implementation
UP	IAM UP	API-Service:API UP
GP	IAM GP	API-Service:API GP
sGP	Restricted GP (AWS username or user_ID)	Restricted GP (OpenStack user_id)
RoP	IAM RoP	API-Service:API RoP
sRoP	Restricted RoP (AWS username or user_ID)	Restricted RoP (OpenStack user_id)
ReP	AWS ReP for the S3, SNS and SQS services	Swift ACL or temporary URLs
TP	IAM TP	missing
CredP	STS-AWS CredP	missing

Access Control List (ACL) or generate a URL that allows the owner temporary access; unlike the ACL, the latter can distinguish between a folder and an object. The special GP and RoP are supported in both cloud platforms using the ID associated to an user.

5 SecurePG

To implement our AC model for the cloud according to Section 3, we developed a prototype implementation called SecurePG (also referred to as *the tool*) that integrates a graphical user interface and a policy engine, both written in Java, and is supported by a MySQL database. The policy engine is responsible to analyse the authorization sentences using version 4.5.3 of the framework ANother Tool for Language Recognition (ANTLR) and identify, with the support of a general purpose grammar, the tuple < Policy Decision, Subjects, Actions, Resources, Conditions > according to our abstract policy language; it also investigates the absence of ambiguities such as the use of the same subject, resource or action in the positive and negative form or errors as the usage of a wrong operator or values when specifying the actions or the conditions names. The MySQL component contains a database schema that replicate the CPs data model and link cloud compliant names with the ones used for the authorization formulas, i.e. the subject name with its Amazon identifiers. By design, this component will interactively ask the user whether to continue if no value or more than one value are retrieved from the database tables. In the first case it also gives the possibility to generate a random value.

Figure 3 provides the architecture of the tool in the default use-case scenario. Using the subjects, actions and resources (referred as the triple $< S, A, R >$) supported by the tool and agreed with the SA, the application developers can easily deploy cloud resource and features on the CP that best meets the requirements. To demonstrate the use of the tool on the reference scenario, we analyse the processing of two authorizations: the sRoP obtained by replacing the subject pattern

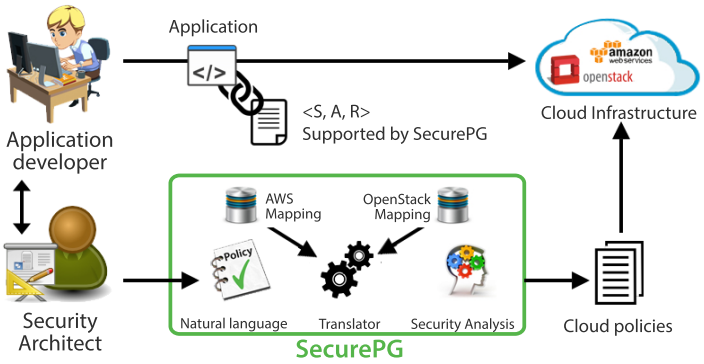

Fig. 3. SecurePG architecture

in Listing 1.2 with "ACME_employee_1 [role = ACME_employees]", reported in Table 3, and the CredP listed below. The latter allows the subject identified by the ID *ACME_user_1* to assume the role with ID *ACME_Customers* to access his profile and is structured in compliance with the Amazon AC model: one role policy (referred as the Role-component in Listing 1.3) assigned to the role *ACME_Customers* and one policy (referred as the Credential-component in Listing 1.3) provided to the *ACME_user_1* subject when authenticating. This policy, uniquely associated to the user's credentials, cannot define new permissions or extend pre-existing authorizations; i.e. the sRoP associated to *ACME_Customers*.

Listing 1.3. Example of an abstract rule that translates to a CredP

```
Role-component: Grant ACME_user_1 [role = ACME_customers] the permission
    to get object and put object on ACME/User_profiles/* [type = keys];

Credential-component: Deny ACME_user_1 [role = ACME_customers] the
    permission to get object and put object on not ACME_user_1_profile
    [type = keys];
```

5.1 Policy Generator Engine

Processing the sRoP and the CredP authorizations, SecurePG is able to determine the correct policy types and suggest the creation of all the necessary entities. For the sRoP, the tool may suggest the creation of one AWS root account, one IAM user and one IAM role (both belonging to the same root account); regarding OpenStack, it recommends the use of a domain and the KeyStone user *ACME_employee_1* (created providing the KeyStone role *ACME_employees*). In both cases SecurePG reports the skipping of the components not supported/recognized within the specific CP. Although some information may be stored, for example as metadata if the interaction refers to a Swift or S3 Resource, it can not be indicated as part of the AC rule.

When specifying a Swift Resource, as in the CredP, the user can choose between two solutions: a generated URL that allows the owner temporary access

to the Resource or a Swift ACL for the Keystone user that is assigned with a role on a project associated to the resource. In the example, the *user_id* 111 of *ACME_user_1* must be linked with the *role_id* 222 of *ACME_customers* and the *project_id* 333 of *P_ACME_user_1*. To implement the first solution, the system requires (or randomly generate) a duration, the resource path and a cluster key *sig* (that acts as a signature), according to the following URL template:

```
https://{host}/{path}?temp_url_sig={sig}&temp_url_expires={expires}
```

Regarding the Swift ACL instead, the SA needs to manually create a *User-Project* assignment using the Keystone interface (in the example, the triple <111, 222, 333>). Since OpenStack does not support the CredP type, SecurePG will ignore the resource attribute *keys* and create a user policy; lacking the support of negative ACLs either, the tool will be able to generate only an authorization associated to the permit component.

Table 3. Example of a sRoP processing

Auth. components	**AWS policy**

Policy decision:
< Grant >
Subjects:
< ACME_employee_1 >
Actions:
< add user
to group, true>
< remove user
from group, true>
Resources:
< ACME_customers,
true>
	[{"Role Policy": {
	"ACME_employee": {
	"Version": "2012-10-17",
	"Statement": {
	"Sid": "1",
	"Effect": "Allow",
	"Action": ["iam:AddUserToGroup,
	iam:RemoveUserFromGroup"],
	"Resource":
	["arn:aws:iam::xx:group/ACME_customers"]
	"Condition": {
	"StringEqualsIgnoreCase": {
	"aws:userid": "AIDAIYHF5BVYLMF36IKZY[4]" }
	}}}}}]

	OpenStack policy
	"identity:add_user_to_group": "role:ACME_employees and 'ACME_customers':%(target.group.name)s and user_id:123"

5.2 Abstract Policy Analysis Engine

To allow the validation of the AC policies before the enforcement, we integrated the support of the Java Content-Based Protection and Release Language tool (JCPRL)[1] to analyse the AC rules provided by the SA. This required a bridge component to translate from the language in Section 3 to first-order logic formulas taken in input by the JCPRL tool. A CPRL document is then created

[4] Value retrieved from the database using the subject name ACME_employee_1

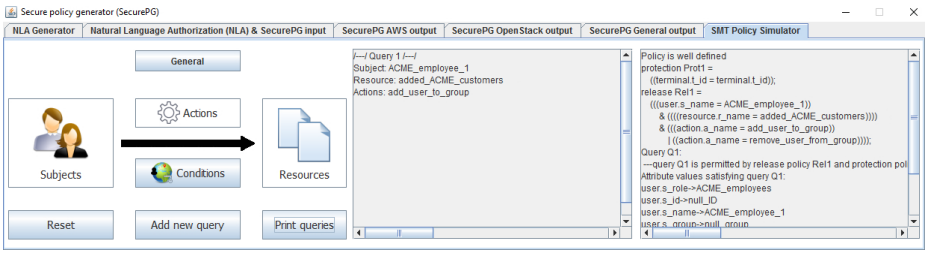

Fig. 4. SecurePG query output using JCPRL

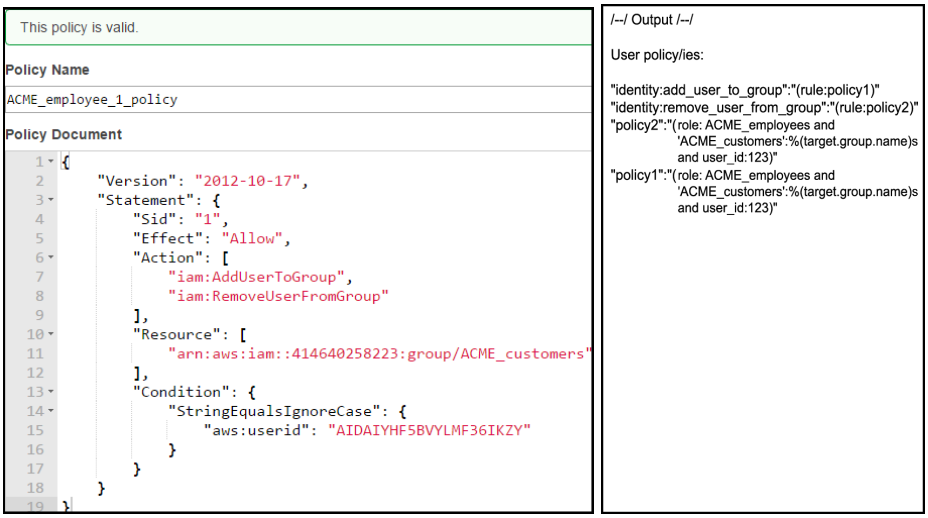

Fig. 5. SecurePG AWS output from the Amazon web UI and OpenStack output

retrieving all the necessary data from a MySQL database and the output is analysed with an SMT solver.

Figure 4 shows the output after the authorization query with the GUI (frame list on the left), while Figure 5 presents the AWS output from the official Amazon web interface when manually adding the rule and the OpenStack output that will be saved in the Keystone *policy.json* file. The former prevents the Amazon java SDK from showing the error message message: *"User: arn:aws:iam::414640258223:user/ACME_employee_1 is not authorized to perform: iam:AddUserToGroup on resource: group ACME_customers ..Error Code: AccessDenied."*.

sectionDiscussion

We have presented a technique supporting the authoring of high-level AC policies for cloud applications, the capability of answering queries independently of a particular cloud platform, and the automated translation of the high-level authorizations to policies that can be enforced in two of the most widely adopted cloud platforms, namely AWS and Openstack. We have also reported our expe-

rience with a prototype tool, called SecurePG, of the technique on a typical cloud application scenario.

The development of a language able to represent, share and facilitate the evaluation of different types of AC policies has received a lot of attention. The eXtensible Access Control Markup Language (XACML) [3] is the *de facto* standard. As discussed in [10], the precision with which this standard works is both a weakness and a strength. The solution proposed in [10] to overcome the difficulties of writing XACML is a graphical tool to display and edit the rules. Our approach is to avoid the difficulties of using XACML by employing a simple and abstract (but expressive) specification language to enforce user requirements in the cloud and easily exploit the JCPRL tool proposed in [1]. This allows to validate the AC policies before their deployment on a particular cloud platform by using a logical semantics and automated reasoning tools to mechanize the authorization query answering process. Similarly to [10], our prototype tool employs a graphical user interface to guide the generation and definition of the AC requirements hiding the complexities of the particular AC model and enforcement mechanism adopted by the cloud platform.

The abstract language we propose is similar to a structured natural language albeit simplified by using syntactic constructs to express ABAC authorization conditions in a way similar to Java Boolean conditions. A lot of work on the use of structured natural language has been done to express AC policies. For instance, the work in [9] analyzed the possibility to express user requirements using a domain dependent grammar and a restricted vocabulary of English sentences. We believe that this constitutes an interesting line of future work that can be integrated in our approach to make our abstract language even more intuitive, expressive, and friendly. Another interesting extension is to integrate a component that allows to import XACML policies in SecurePG, and therefore to compare our tool with other policy generators; following the ABAC policy mining process presented in [11], we tested (although not integrated in SecurePG) a component that processes the Amazon RBAC policies to generate expressions compliant with our abstract policy language. This allows to load pre-existing AWS policies (in the native JSON format) and generate, when supported by the OpenStack AC model, equivalent OpenStack authorization rules.

To the best of our knowledge, our technique is the first that is capable of generating enforceable policies in AWS and Openstack from a high-level description of the AC requirements. Our research is now trying to expand the support of our prototype tool to other cloud platforms; such as Microsoft Azure or Google Cloud in order to gain further experience with the automatic translation of high-level policies. We also intend to enrich the abstract policy languages with more constructs to express, for instance, the purpose of access—a feature which is becoming of paramount importance to ensure the privacy of the processing. This will also require the automatic synthesis of monitors to guarantee the satisfaction of purpose constraints. To this end, we envisage to integrate the approaches in, e.g., [5, 2].

References

1. Armando, A., Ranise, S., Traverso, R., Wrona, K.: Smt-based enforcement and analysis of nato content-based protection and release policies. In: Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control. pp. 35–46. ACM (2016)
2. Bertolissi, C., Dos Santos, D.R., Ranise, S.: Automated synthesis of run-time monitors to enforce authorization policies in business processes. In: Proc. of ASIACCS. pp. 297–308. ACM (2015)
3. Committee, O.X.T., et al.: eXtensible Access Control Markup Language (XACML) Version 3.0. Oasis standard, OASIS (2013)
4. De Capitani Di Vimercati, S., Foresti, S., Samarati, P., Jajodia, S.: Access control policies and languages. International Journal of Computational Science and Engineering 3(2), 94–102 (2007)
5. De Masellis, R., Ghidini, C., Ranise, S.: A declarative framework for specifying and enforcing purpose-aware policies. In: International Workshop on Security and Trust Management. pp. 55–71. Springer (2015)
6. Ferry, N., Song, H., Rossini, A., Chauvel, F., Solberg, A.: Cloudmf: applying mde to tame the complexity of managing multi-cloud applications. In: Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on. pp. 269–277. IEEE (2014)
7. Hu, V.C., Ferraiolo, D., Kuhn, R., Friedman, A.R., Lang, A.J., Cogdell, M.M., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K., et al.: Guide to attribute based access control (abac) definition and considerations (draft). NIST special publication 800(162) (2013)
8. Jin, X., Krishnan, R., Sandhu, R.: A unified attribute-based access control model covering dac, mac and rbac. In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 41–55. Springer (2012)
9. Perry, J., Arkoudas, K., Chiang, J., Chadha, R., Apgar, D., Whittaker, K.: Modular natural language interfaces to logic-based policy frameworks. Computer Standards & Interfaces 35(5), 417–427 (2013)
10. Stepien, B., Felty, A., Matwin, S.: A non-technical user-oriented display notation for xacml conditions. In: International Conference on E-Technologies. pp. 53–64. Springer (2009)
11. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies from rbac policies. In: Emerging Technologies for a Smarter World (CEWIT), 2013 10th International Conference and Expo on. pp. 1–6. IEEE (2013)
12. Zhang, Y., Patwa, F., Sandhu, R.: Community-based secure information and resource sharing in aws public cloud. In: Collaboration and Internet Computing (CIC), 2015 IEEE Conference on. pp. 46–53. IEEE (2015)
13. Zhang, Y., Patwa, F., Sandhu, R., Tang, B.: Hierarchical secure information and resource sharing in openstack community cloud. In: Information Reuse and Integration (IRI), 2015 IEEE International Conference on. pp. 419–426. IEEE (2015)