J Grid Computing (2022) 20: 18
https://doi.org/10.1007/s10723-022-09606-1

A Formal Approach for the Identification of Authorization Policy Conflicts within Multi-Cloud Environments

Ehtesham Zahoor · Asim Ikram · Sabina Akhtar · Olivier Perrin

Received: 31 August 2021 / Accepted: 17 April 2022 / Published online: 20 May 2022
© The Author(s), under exclusive licence to Springer Nature B.V. 2022

Abstract The use of the Cloud computing has been constantly on the rise. The flexible billing model coupled with elastic resource provisioning make the Cloud appealing to consumers. However there are still many challenges associated with the Cloud limiting its adoption, such as vendor lock-in and security concerns. One way to address some of these challenges is to use services from more than one Cloud providers. This may help in avoiding the case of vendor lock-in and will also allow for the use of multiple resources available at multiple Clouds. The use of multi-cloud environments can also assist in the case of Cloud bursting where a workload in a private cloud bursts into a public cloud when the need arises. However, the security concerns in such an environment are amplified when compared to a single Cloud. In this paper we address the specification and consistency management of authorization policies in Multi-Cloud environments. The problem being address is significant as an erroneous authorization policy can have severe consequences on the security of the system being protected. In a Multi-Cloud environment, it is difficult to ensure consistency with different Clouds having authorization models, different implementations of the same authorization model and different access control policies. To this end, we have proposed a formal Event-Calculus based model to model the aggregated authorization policies from multiple Cloud providers. The translated Event-Calculus models are then reasoned upon to identify the policy conflicts. We have applied our approach on authorization policies from AWS, GCP and Microsoft Azure. Further, we have provided tool support to automate the complete verification process and provided detailed performance evaluation results to justify the practicality and scalability of the proposed approach.

Keywords Cloud · Authorization · Access control · Event-calculus · Verification · Multi-cloud

Ehtesham Zahoor (✉) · Asim Ikram
FAST, NUCES, Islamabad, Pakistan
e-mail: ehtesham.zahoor@nu.edu.pk

Asim Ikram
e-mail: i161022@nu.edu.pk

Sabina Akhtar
Bahria University, Islamabad, Pakistan
e-mail: sabina.buic@bahria.edu.pk

Olivier Perrin
Université de Lorraine, LORIA, 54506, Vandoeuvre-lès-Nancy Cedex, France
e-mail: olivier.perrin@loria.fr

1 Introduction

The need for information security has always been there. It has however amplified in last few decades thanks to our increasing reliance and widespread adoption of information systems. At one end, the need for computing resources exposed the scalability limitations of legacy systems and resulted in the wide

spread adoption of distributed computing architectures such as Cluster, Grid and the Cloud computing. In this context, the use of the Cloud computing has been constantly on the rise. The flexible billing model coupled with elastic resource provisioning make the Cloud appealing to consumers. Major providers include Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure and IBM Bluemix. However, reliance on a particular Cloud provider may result in vendor lock-in making it difficult to port information systems from one Cloud provider to the other. Further, an organization may require services from more than one Cloud provider and this led to the inception of multi-Cloud environments such as *RightScale* Multi Cloud Platform, a cloud broker that provides multi-Cloud solutions.

The increase in scale has put enormous pressure to improve our security capabilities needed to secure these systems. Information security research has thus been in the mainstream of computing and has been widely studied research direction for the past few decades. Security has been one of the most critical factors limiting Cloud adoption and all major providers, have well-defined and thorough security capabilities. However, the challenges are further amplified in the case of multi-Cloud environments as there is no agreed-upon consistent approach to implement security principles and it becomes challenging to aggregate different implementations. There are many facets and levels to implement security principles in an organization. One such aspect is the authorization process which allows for controlling access to the resources. Authorization process is handled by authorization policies which specify access rules specifying which subjects can perform specified actions on the specified objects. As similar to other services, all major Cloud providers provide authorization process as a service. These include Identity and Access Management (IAM) service provided by the Amazon Web Services (AWS) and Google Cloud Identity & Access Management (IAM) service provided by Google Cloud Platform (GCP).

A Multi-Cloud environment with multiple Cloud providers may result in the use of different authorization models and different implementations of the same authorization model by different Cloud providers. As an example we consider the case of authorization services provided by major Cloud providers. The Google Cloud Identity & Access Management (IAM) is the access control service provided by Google Cloud Platform (GCP). The service uses the Role Based Access Control (RBAC) model and it allows to define and assign the required roles to the users. An example policy is shown in Fig. 1-b. Further, AWS provides an identity and access management service (IAM) for managing the authorization and authentication process. These policies can be either directly assigned to IAM Users, the UBAC policy model, or to IAM Groups and AWS thus supports both UBAC and RBAC. Microsoft Azure, like GCP, also uses an RBAC model for authorization specification. Azure policies consist of actions, not actions, and assignable scopes [4]. An example policy is shown in Fig. 1-a. In a Multi-Cloud environment policies

```
{
  "id": "/subscriptions/28d7b..."
  "properties": {
    "roleName": "ExampleRole",
    "description": "An example.",
    "assignableScopres": [
        "/"],
    "permissions": [
     {
       "actions": [ "*/read" ],
       "notActions": [],
       "dataActions": [],
       "notDataActions": []
     }]}
```

a) An example Microsoft Azure Policy

```
{
  "bindings":[
   {
     "members": [
     "serviceAccount:alice-27@alice.
       iam.gserviceaccount.com"
     ],
     "role" : "roles/iam.serviceAdmin"
   }
  ],
  "etag": "BwXEbmYO-ek",
  "version": 1
}
```

b) An example GCP Policy

```
{
  "Statement":[
   {
     "Sid": "Stmt01",
     "Effect": "Allow",
     "Action": [
        "aws-portal:ViewBilling"
     ],
     "Resource":[
        "*"
     ]
   }
  ]
}
```

c) An example AWS Policy

Fig. 1 GCP, Microsoft Azure and AWS Policy Examples

from multiple Clouds need to be aggregated and this becomes challenging as their capabilities, expressiveness and implementations differ. For instance, AWS supports UBAC, a model not supported by other Cloud providers. For the RBAC model, supported by all the Cloud providers, the implementation of the model differ such as the support of explicit *deny* in case of AWS IAM. Policies aggregation may lead to conflicting access control policies and it becomes challenging to ensure consistency after aggregation.

In this paper we address the specification and consistency management of authorization policies in Multi-Cloud environments. To this end, we have proposed a formal Event-Calculus [13] based model to model the aggregated authorization policies from multiple Cloud providers. The translated Event-Calculus models are then reasoned upon to identify the policy conflicts. We have applied our approach on authorization policies from AWS, GCP and Microsoft Azure. Further, we have provided tool support to automate the complete verification process and provided detailed performance evaluation results to justify the practicality and scalability of the proposed approach. This paper builds on our approach presented in [29] and in this work we have throughly improved content, implementation details and the performance evaluation results. We have also modified and improved the Event-Calculus models for more expressiveness and performance. A detailed discussion about changes to the EC models can be found in Section 5.3 and a detailed performance evaluation comparison can be found in Section 8.

The rest of the paper is organized as follows: Section 2 discusses the traditional approaches for authorization polcicies management and consistency verification. Further, in Section 3 we briefly present how major cloud providers like AWS, GCP and Microsoft Azure handle the authorization process. We also introduce a motivating example to highlight the problem being addressed in this work and our contributions. In Section 4 we provide an overview about the proposed approach and detail the proposed approach for policies aggregation in 5. Section 6 discusses the authorization conflicts in Multi-Cloud environments categorizing them as syntactic and semantic conflicts. Further, in Section 7 we present the implementation details and the performance evaluation results in Section 8.

2 Related Work

The authorization policies and different policy models are used to handle the authorization process. In Role Based Access Control (RBAC) model, users are assigned roles and are grouped together. Each role has its defined authorization policies. On the other hand, in User based access control (UBAC) there is no concept of roles. The authorization policies are applied on the individual users. RBAC is the most commonly used policy model but as the number of roles increase, it suffers from scalability issues. The number of roles can surpass the number of users [9] because different roles are created to address the needs of diverse permissions. This scenario is called role explosion problem. Attribute based Access Control (ABAC) policy model [11] associates some attributes with the resources, subjects and environment. Authorization policy using ABAC is then considered as a boolean function on these attributes. ABAC model provides more flexibility as compared to RBAC models. It can incorporate other policy models as the *user* and the *role* (as needed for UBAC and RBAC respectively) can be considered as an attribute in an ABAC model.

A lot of research has been conducted on authorization in the Cloud Computing domain. Data storage on the Cloud based services requires research in the area of security that includes Attribute-Based Encryption (ABE) proposed in [26]. In Attribute-Based Encryption (ABE), data encrypted based on attributes and only users with the same set of attributes can decrypt the data. Similarly, the authors in [24] have proposed a model that uses Ciphertext Policy based ABE (CP-ABE) to achieve access control in multi-authority clouds. The authors in [1] have modified the CP-ABE algorithm to make it more secure and have extended their approach to multi-cloud authorities as well. The model proposed by [6] was also concerned with the improvement of CP-ABE and put forth a model to revoke permissions by dividing the data after uploading it to the cloud. The approaches used in [25, 31] address the attribute hierarchies and revocation while the approach proposed in [21, 22] deals with the keyword search.

The policy languages used for specification of authorization policies is another area of research on authorization in Cloud Computing. Some approaches

have considered providing formal semantics [7, 12, 16, 23] in XACML (eXtensible Access Control Markup Language) as it is based on the Attribute Based Access Control model. An extended version of XACML is proposed in [14] that handles specifying an integration policy at its own end before merging its policies with another entity. In [5], authors have used a formative work that uses an algebra for policy composition from multiple domains and it supports heterogeneous and unknown policies as well. A method to integrate policies from one cloud to another cloud is presented in [17] . However their approach is limited as it only supports homogeneous platforms and it only supports OpenStack. An algebra for fine-grained integration of policies focusing on generating XACML policies is proposed in [19]. A service brokering scheme is introduced by authors in [18] that satisfies the user requests. In [2], a fine grained access control mechanism for data sharing in cloud federations has been proposed. The authors in [8] have proposed a model to identify policy changes and an XACML model for policy evaluation. The authors in [20] have proposed an interesting method concerned with the conversion of XACML access control policies to Answer Set Programming and have evaluated their approach on the Clingo tool.

For distributed environments, the literature mainly discusses about the formal semantics of XACML and the problem of policy aggregation. According to our research and knowledge about the literature, there exists no formal approach (other than our previous work [29]) that handles the policy aggregation and conflicts identification amongst access control policies in Multi-Cloud environments, focusing on *actual* Cloud providers and Multi-Cloud solutions. It is difficult to ensure consistency in an environment where we can have multiple Clouds with different authorization models, different implementations of the same authorization model and different access control policies. Our work in this paper focuses on an approach for policy aggregation and have also categorized and proposed an approach to identify policy-conflicts in such environments.

This paper further builds on our approach presented in [29] in which we have applied our technique on actual policies from AWS, GCP and Microsoft Azure. We have also provided tool support to automatically fetch policies from the cloud providers and convert them to Event-Calculus [13] models. Our work explains the design of the Event-Calculus models in further detail and the proposed models perform better as compared to the existing work. It also improves on the implementation details of our work on Authorization policies. We have also refined the results for our implementation and provided a comparison of our approach and tool with another work in the literature. A detailed discussion about changes to the EC models can be found in Section 5.3 and a detailed performance evaluation comparison can be found in Section 8. In [27] the authors have proposed verification of Intra and Inter policy conflicts for AWS IAM policies, but this work focuses on Multi-Cloud environments. In our work, we present optimized Event-Calculus models that allow for both design time verification and for policy evaluation based on actual request.

3 Authorization in the Cloud

In this section we will briefly discuss how major cloud providers like AWS, GCP and Microsoft Azure handle the authorization using their models and services including the case of RightScale Multi-Cloud service. We will also discuss our motivating example that will highlight our contributions even further.

3.1 IAM Services by Major Providers

The Google Cloud Identity & Access Management (IAM) is the access control service provided by Google Cloud Platform (GCP) that is based on RBAC authorization model. For different services provided by GCP, it allows to add the users and assign them the required roles. You can create new roles by selecting the permissions or you can also inherit from existing roles.

An example policy is shown in Fig. 1-b. The components of a GCP policy are described below [10].

- *Bindings:* Associate a list of members with roles
- *Roles:* Specifies the roles that the users are assigned. Roles have predefined permissions to resources.
- *Members:* Defines the users that are granted the specified role. Members can be specified by an email ID, a service account ID, all authorized users, all users (anyone on the internet), domain, or a user group.

Amazon provides Amazon Web Services (AWS) as a Cloud solution for its users. To handle authentication and authorization of AWS users, it supports an identity and access management service (IAM). Policies in IAM are high level descriptions explicitly listing the permissions and are stored in a JSON format. Each *policy* has a set of *statements* and each statement contains a *Resource*, *Action* and an *Effect*. An *Effect* refers to the access to resource whether the resource is allowed or denied. These policies can be either directly assigned to IAM Users, the UBAC policy model, or to IAM Groups. AWS supports both UBAC and RBAC, but RBAC suffers from role explosion. Therefore, AWS imposes some limitations (for example number of policies attached to a role) but that is not a convincing solution. An example policy is shown in Fig. 1-c. The components of an AWS policy are described below [3].

- *Effect:* Specifies whether access to resources is allowed or denied.
- *Principal:* Specifies the user/account the permissions are granted to.
- *Action:* Specifies the actions that are allowed on the resources (* specifies all permissions).
- *Resources:* Specifies the components on which the effect is applied.

Microsoft Azure, like GCP, also uses an RBAC model for authorization specification. Azure policies consist of actions, not actions, and assignable scopes [4]. An example policy is shown in Figure 1-a.

- *Actions:* Define the permissions allowed on a resource.
- *Not actions:* Defines actions not allowed on resources that are included in actions.
- *Assignable scopes:* Defines the scopes to which the role can be assigned to.

RightScale Multi Cloud Platform (*RightScaleMC*) is a cloud broker that provides Multi-Cloud solutions based on RBAC. They support two types of accounts that are *RightScaleAccounts* and *CloudAccounts*. *RightScale* dashboard and other services are accessed using *RightScaleAccounts*. On the other hand, the *CloudAccounts* are linked to the cloud provider account, for instance an AWS account. The roles provided by the *RightScaleMC* include *admin*, *actor* and *observer* and other roles.

3.2 Motivating Example

As a motivating example, consider the case where a user needs to use resources such as storage and computation from multiple cloud providers (AWS, GCP, and Azure). Let us consider that the user is named *Alice* and she is working on a project *MCproject* that is using resources from multiple cloud providers as shown in Fig. 2. To use resource from multiple cloud

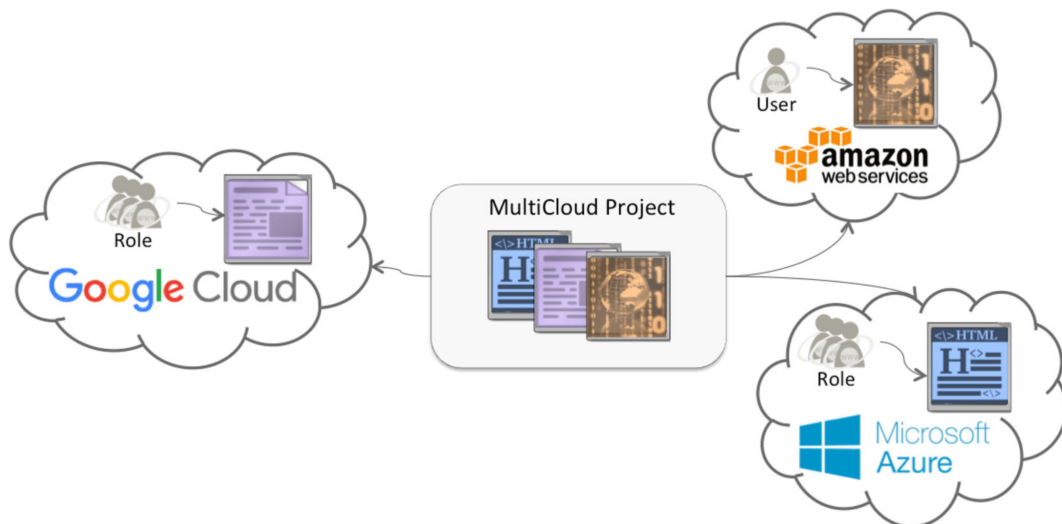

Fig. 2 A Multi-Cloud Scenario

providers, the user needs access to those resources. One case would be to use a cloud broker, such as *RightScale Multi Cloud*. RightScaleMC allows users to get authorization access to multiple cloud providers using RBAC. For our scenario, the user would first have to create a project at RightScaleMC (let us call it *mcProject*). We would then have to create multiple cloud accounts that give us access to other cloud providers. Before we can actually add people to work on the project, we need to create roles. For this purpose, we can create an *actor* role and add members, including Alice, to the project.

From the example given for *RightScaleMC*, we can achieve the purpose of using resources from multiple cloud providers. However, if we examine the authorization policies provided by *RightScaleMC*, we find that they are not fine-grained. Alice might be allowed to perform actions that would not be allowed to her. This could be due to different policy specifications on the cloud providers (RBAC vs. UBAC) or even different naming conventions on the cloud providers. To combat this, we could add more roles that would handle these conflicts but we believe that increasing the number of roles is not sufficient to handle this issue. This is because increasing the number of roles to match those of the cloud providers and to further handle users/roles at the multi-cloud level would lead to role explosion. The problem of role explosion is especially amplified in the case of multi-clouds since roles from a number of clouds need to be aggregated in the correct manner (which in some cases would lead to the creation of new roles) to provide a correct authorization model. From this, we can conclude that using RBAC to handle conflicts in such a diverse environment as a multi-cloud is not a viable solution and that we need to propose a model other than RBAC when aggregating policies from multiple cloud service providers and identifying conflicts between the policies. An attribute based access control (ABAC) model would be preferable in such a scenario as ABAC subsumes both UBAC and RBAC by considering role as an attribute.

4 Proposed Methodology

The proposed approach relies on the formal Event-Calculus (EC) models for the identification of conflicts in multi-cloud environments. Figure 3 presents an overview of the proposed approach. The policies are written by policy designers using services provided by the Cloud, for instance using Identity and Access Management (IAM) service of AWS. These policies may be both syntactically and semantically

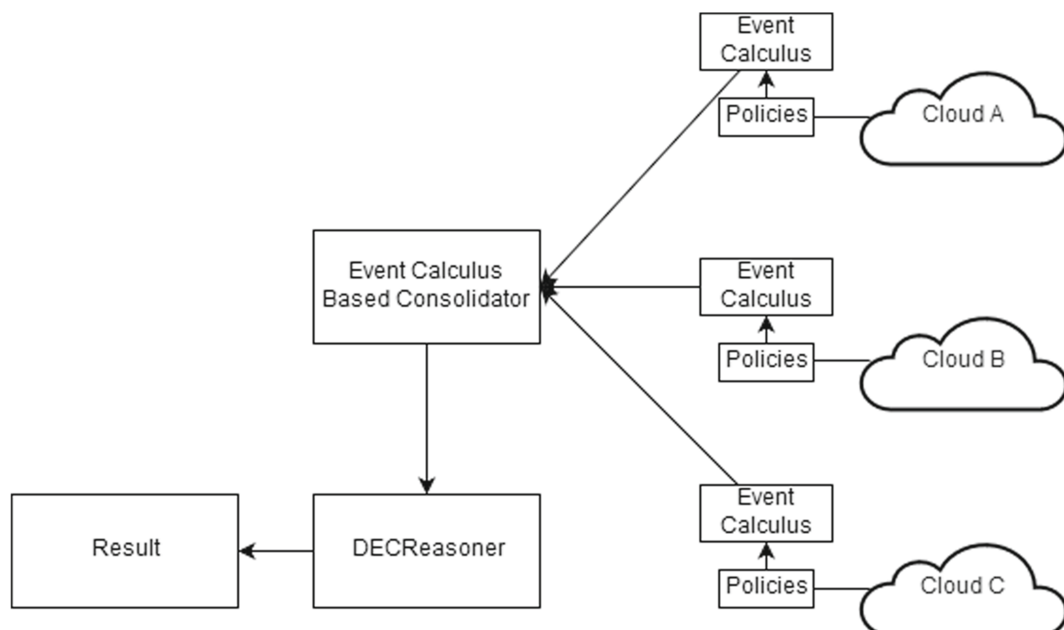

Fig. 3 Proposed Framework

different and they are converted to the generic Event-Calculus models to be reasoned upon and to identify conflicts. Different models from different Cloud providers are then aggregated and passed on to the *DECReasoner*, which is a tool initially developed by IBM to reason about Event-Calculus models. The results provided by the *DECReasoner* can help identify the presence (or absence) of the policy conflicts.

Event-Calculus (EC) is a logical language proposed by Robert Kowalski and Marek Sergot [13] to model and reason about actions over time as we discuss in the Section 5. The models are difficult to design and understand for a policy designer and so is the overall reasoning process by the *DECReasoner*. We have intentionally designed the models to be highly generic to model multiple related aspects and this has further allowed us to automate the reasoning process requiring no knowledge of EC and the reasoning process. The proposed methodology consists of fetching the rules from AWS, GCP, and Azure through a web application. The web application then converts the rules to EC based models automatically and aggregates them to a combined/composite policy. The composite policy can then be passed to DECReasoner for evaluation. This process is illustrated in Fig. 3. The details of the EC models and web application are given in Sections 5, 6, and 7.

5 Policies Aggregation in Multi-Cloud Environments

There is no standard way of representing authorization policies in the Multi-Cloud environments. Each cloud provider uses its own authorization model similarly it will have different implementations of the model and policy specifications as well. Along with these restrictions an organization might also have its own policies, thus complicating the process of policy aggregation even further. Our approach proposed in this paper is formal and based on ABAC model as it subsume RBAC, UBAC as well as other access control models and it can also handle role explosion.

5.1 Event-Calculus

Event-Calculus (EC) is a logical language introduced by Robert Kowalski and Marek Sergot [13] to model and reason about actions over time. In EC, events represent the actions, $\mathcal{A}$, and for time-varying properties called fluents, $\mathcal{F}$, the events trigger the change in state. In EC, fluents are like variables whose value can change over time, for example *AccessGranted* can be a fluent that may *hold* or may not hold at certain time during execution. In this work, we would use discrete EC [15] as it limits time-points to integer values and we would use its associated reasoner, called *DECReasoner*.

We have several motivations for choosing Event-Calculus for specification of our models. It is highly expressive as it allows a detailed model to be created. It can also handle the events which are context-sensitive or have indirect effects. For any given model, various types of commonsense reasoning can be performed. It has an explicit time structure and can be used to give temporal representations. Finally, it is flexible as the same logical model can be used for verification at both design time and runtime. Due to limited space, we would only emphasize on core concepts[1] and we have used universally quantified variables, unless explicitly specified, and we have shortened their names as well.

5.2 Rules Specification

We start our discussion by first presenting the EC models for the *rules* construct, which can be used to specify an access rule. Each rule has a *Target* and an *Effect*. The rule target specifies if the rule applies to a particular context i.e. is it applicable for a given request. To model the rule target in EC, we first define some sorts, such as *rule*, *subject*, *object* and *action*. When specifying an actual rule, we will instantiate these sorts and they can thus be regarded as types. The state of a rule can be either it is permitted, denied or not applicable (if the rule target does not hold) and these states can be modeled using EC fluents, whose value can change over time. For instance, a rule is neither permitted or denied at the start and this can change over time. We have thus defined fluents *RuleIsPermitted/Denied/NotApplicable* to represent rule state.

The state of a fluent changes on the occurrence of events and we have also defined some events, such as *Approve/DenyRule*, whose occurrence would change

[1]Complete models can be found at https://www.icloud.com/iclouddrive/0cwdSlable8lHOX_NyBRhf5SA#nordsec.zip

the state of fluents. Further, EC *Initiates/Terminates axioms* link an event with fluent state and specify how the fluent state is changed on the occurrence of the event. In our model, Initiate axiom states that if the event *ApproveRule* happens at time *t*, the fluent *RuleIsPermitted* would hold at *t+1*. Further, we have defined some constraints to restrict events occurrence and the events can only occur (happen) once specific condition hold (or does not hold). We then specify the initial states for the fluents, they do not hold at time 0, and the goal for the reasoner to find a solution to reach the goal state.

Rules Model 1 (Meta-model for IAM Rules)

```
;Sorts in EC are like types
;Their instances represent individual rules and subjects/objects and actions
sort rule, subject, object, action

;Fluents change with time and we define the following
fluent RuleTargetHolds(rule), RuleConditionHolds(rule)
fluent RuleEffectIsPermit(rule),
fluent RuleIsPermitted/RuleIsDenied/RuleIsNotApplicable(rule)
;Events occur at specific time-points and we define the following
event Match(rule), Mismatch(rule),
Approve/DenyRule(rule), RuleDoesntApply(rule)

;These axioms link fluents with events
;Initiates makes fluent hold and Terminates does otherwise
Initiates (Match(rule), RuleTargetHolds(rule), time).
Terminates (Mismatch(rule), RuleTargetHolds(rule), time).
Initiates(Approve/DenyRule(rule), RuleIsPermitted/Denied(rule), time).
Initiates(RuleDoesntApply(rule), RuleIsNotApplicable(rule), time).

;The conditions below are to restrict even occurrence
;The events can only occur once specific condition hold (or does not hold)
Happens(ApproveRule(rule), time) → HoldsAt(RuleTargetHolds(rule), time) &
& HoldsAt(RuleEffectIsPermit(rule), time).
Happens(RuleDsntApply(rule), time) → !HoldsAt(RuleTargetHolds(rule), time).

;These are the initial states for the Fluents, they do not hold at time 0
!HoldsAt(RuleIsPermitted/Denied/NotApplicable(rule),0).
;The goal for the reasoner, it tries to find a solution reach this state
HoldsAt(RuleTargetHolds(rule),1) | !HoldsAt(RuleTargetHolds(rule),1).
HoldsAt(RuleIsPermitted/Denied/NotApplicable(rule),2).
```

In terms of control flow, we need to first check whether the specific rule being defined is applicable, i.e. rule target holds or not. We include *Match/Mismatch* events in the model as they take the decision of a fluent *RuleTargetHolds* that whether it holds or does not hold. If the fluent holds, this means the decision is based on rule effect to be permitted or denied and if the fluent does not hold then rule is considered to be not applicable, *RuleIsNotApplicable*. We have made this model as a generic one so that it can be considered as a meta-model to be included for the specification. Below, we show an example of using a generic model for the UBAC based AWS rule and RBAC based GCP rule.

Rules Model 2 (AWS IAM rule specification)

```
load includes/rules/... ;generic model files

load includes/input.e
subject Alice
object AWSresource
action AnyAction
;Attributes would be specified in input.e based on the actual request

rule RuleAWS
;Specifying when the rule target holds
[time] Happens(Match(RuleAWS),time) ->
{subject, object, action} subject = Alice & object = AWSresource & action = AnyAction.

time, subject, object, action Happens(Mismatch(RuleAWS), time) ->
subject != Alice | object != AWSresource | action != AnyAction.
HoldsAt(RuleEffectIsPermit(RuleAWS),0).
```

In the model above, we first include the generic meta-model files and then include a file named *input.e*. This file contains the attribute name value pairs matching the actual authorization request. In our case *Alice* is trying to perform *AnyAction* on the *AWSresource*.

For the rule specification, we name the rule as *RuleAWS* and define a conditional axiom that the event *Match* can only happen if the attribute name value pairs match. The variables in curly brackets, such as {subject} are existentially quantified and the first axiom essentially says that the event Match can only happen if there exists a *subject* named *Alice*, *object* named *AWSresource* and an *action* named *AnyAction*. We define another axiom stating that the event *Mismatch* happens otherwise.

The model is for the UBAC approach as supported by AWS. However, the proposed approach is generic and is based on ABAC so we can also model RBAC, as provided by GCP. This is shown below:

Rules Model 3 (GCP rule specification)

```
...
rule RuleGCP
[time] Happens(Match(RuleGCP),time) ->
{subject, object, action} subject = ProjMembers & object = GCPresource & action = Read.

time, subject, object, action Happens(Mismatch(RuleGCP), time) ->
subject != ProjMembers | object != GCPresource | action != Read.

HoldsAt(RuleEffectIsPermit(RuleGCP),0).
```

In order to reason about the models, we can use the *DECReasoner*[2] which attempts to find a solution (sequence of events) that leads from initial fluents state to the goal. The solution shows which events happen and what fluents hold true (shown with a plus(+) sign) at specific time-points. When we invoke DECReasoner on the above AWS model, it shows an output as shown in evaluation 1. DECReasoner first encodes the problem into a satisfiability (SAT) problem and then invokes a SAT solver to reason about the problem. The solution shows the events that occur and the fluents that hold true or false at certain time points.

Solution 1 (AWS Rule Evaluation Using DECReasoner (Match))

```
40 variables and 89 clauses
relsat solver
1 model
—
model 1:
0
RuleEffectIsPermit(RuleAWS).
Happens(Match(RuleAWS), 0).
1
+RuleTargetHolds(RuleAWS).
Happens(ApproveRule(RuleAWS), 1).
2
+RuleIsPermitted(RuleAWS).
3
```

Evaluation 1 shows the results obtained when DECReasoner was run on an AWS rule. In this case, the attribute values in the input file, were intentionally kept the same for easier illustration of results. In practice, the values would be populated based on actual requests. Since the values were initialized for the illustration of a *match* event, the execution of DECReasoner results in a *Match* event and the rule gets permitted. When the attribute values do not match, a *Mismatch* event occurs and the rule is not applicable as shown in evaluation 2.

[2]http://decreasoner.sourceforge.net/

Solution 2 (AWS Rule Evaluation Using DECReasoner (Mismatch))

```
64 variables and 218 clauses
relsat solver
1 model
—
model 1:
0
RuleEffectIsPermit(RuleAWS).
Happens(Mismatch(RuleAWS), 0).
1
Happens(RuleDoesntApply(RuleAWS), 1).
2
+RuleIsNotApplicable(RuleAWS).
3
```

5.3 Comparison to the existing models

For the specification of rules, the core model for identification of rule target (if the rule applies and if the event *Match* or *Mismatch* should happen) as presented in [29] is shown below:

Rules Model 4 (Meta-model for IAM Rules [29])

```
;Sorts for attributes name/values
sort rule, atname, atvalue        predicate AtHasValue (atname, atvalue)
;Fluents for Rules evaluation
fluent RuleTargetHolds(rule), RuleConditionHolds(rule)
fluent RuleEffectIsPermit(rule), RuleIsPermitted/Denied/NotApplicable(rule)
;Rest of the model follows ...
```

We first define some sorts, such as *rule*, *atname* and *atvalue*, which can be considered as types and which represent individual rules, attribute names and values respectively. The predicate *AtHasValue* links attribute name-value pairs. Then when we instantiate the model, we specify attribute names/values and link them using a predicate *AtHasValue* as shown below.

Rules Model 5 (AWS IAM rule specification [29])

```
load includes/rules/... ;generic model files

load includes/input.e
atname Subject, Object, Action
atvalue Alice, AWSresource, AnyAction
AtHasValue(Subject,Alice). AtHasValue(Object,AWSresource)...
;Attributes would be specified in input.e based on the actual request

rule RuleAWS
;Specifying when the rule target holds
Happens(Match(rule),time) &
AtHasValue(Subject, atvalue1) & AtHasValue(Object, atvalue2) & AtHasValue(Action,
atvalue3) -> atvalue1 = Alice & atvalue2 = AWSresource & atvalue3 = AnyAction.

HoldsAt(RuleEffectIsPermit(RuleAWS),0).
```

We have identified some limitations of the models as presented in [29] and [27]. Our model does not make use of attribute names and values and can specify policy relationships. In addition, our model improves performance as universally quantifying multiple instances of sorts *atname* and *atvalue*, as shown

in the model above, makes the models complex and less efficient.

We presented EC models for policy shadowing in [28]. The models for the identification of shadow policies are different and it is essentially a design time process. In terms of EC models, however some comparison is still possible. In this context, even for the specification of rules the proposed approach is more expressive. For instance, it can handle the case when a subject has multiple identities, i.e. belongs to multiple groups. In this case the policies need to be evaluated to cater for any role to which a user belongs and any matching policy would result in access being granted (or denied if so is the rule effect). Let us consider the *RuleGCP* presented earlier which allows *ProjMembers* to have *read* access on some resource named *GCPresource*. Let us consider a user named *Alice* which belongs to the role named *ProjMembers* and also to some other role named *Users*. In this case the *input* request needs to be updated to have all the identities belonging to a user and then a request needs to be made as shown in the model below.

Rules Model 6 (The contents of input.e file for a user having multiple identities)

```
subject Alice,ProjMembers,Users
object GCPresource
action Read
```

Invoking the DECreasoner after this change would still grant access to the user Alice, as she belongs to the *ProjMembers* group. Building further on this example however reveals another interesting case where there is a conflict in the roles assigned to a user. In the example above, consider the case where the members of role *Users* are not allowed to access that specific resource. These conflicts are not in the scope of this work but can easily be identified by extending the proposed models.

5.4 Authorization Composition

In a multi-cloud environment, rules from multiple clouds may need to be aggregated to provide authorization. Aggregation of rules can cause issues since each cloud provider has its own implementation of authorization rules. For example, the rule specification of AWS is more high level than that of GCP and Azure. In AWS, policies contain multiple rules based on resources and the actions that can be performed those resources. GCP policies can be considered more abstract than AWS. They consist of assigning roles to members. The members are then assigned permissions based on the roles. GCP does not have the ability to directly assign permissions to members, it has to do it in an indirect manner by using roles. Azure also has policies that are different than AWS and GCP. As an example, Azure has the ability to specify which actions are allowed and which actions are explicitly not allowed on a resource.

The method that we followed to handle the inconsistency in the implementation of the policies was to consider GCP and Azure rules as-is (permissions for each resource) and consider AWS policies as individual rules. This approach results in better performance than combining GCP and Azure rules into policies since this would involve more constructs.

The reason for the performance degradation in case of combining rules in policies is that we would first need to evaluate a rule, then evaluate the policy, and then the results of the evaluated policy could be passed to the composite policy. This would add an extra step to the evaluation process and as such would have a negative effect on the performance.

Due to disadvantage of combining GCP and Azure rules in a policy, we have unwrapped AWS policies as rules and used those rules in our model. The meta-model for policies is shown in policy model 1.

Policy Model 1 (Meta-model for Policies)

```
sort policy        predicate PolicyHasRule(policy, rule)
;Fluents for Policy State/Evaluation
fluent PolicyIsPermitted(policy)
fluent PolicyIsDenied(policy)
;Initiates Axioms for Events/Fluents
event ApprovePolicy(policy)
Initiates(ApprovePolicy(policy), PolicyIsPermitted(policy), time).
event DenyPolicy(policy)
Initiates(DenyPolicy(policy), PolicyIsDenied(policy), time).

;permit if even one of the rule is permitted - permit overrides
Happens(ApprovePolicy(policy), time) -> {rule} PolicyHasRule(policy, rule) &
HoldsAt(RuleIsPermitted(rule), time).
;Deny if all are not applicable
Happens(DenyPolicy(policy), time) & PolicyHasRule(policy, rule) ->
HoldsAt(RuleIsDenied(rule), time) | HoldsAt(RuleIsNotApplicable(rule), time).

;For all events we don't them to happen again if the fluent is true
HoldsAt(PolicyIsPermitted(policy), time) -> !Happens(ApprovePolicy(policy), time).
HoldsAt(PolicyIsDenied(policy), time) -> !Happens(DenyPolicy(policy), time).

;Initial conditions for fluents
!HoldsAt(PolicyIsPermitted/Denied(policy),0).
```

First we define a fluent *policy* that defines a policy. Next, we define a predicate *PolicyHasRule* that defines which rules exist within a policy. We then define the fluents *PolicyIsPermitted*, *PolicyIsDenied* that identify whether a policy would be permitted or denied. Next, we define the events *ApprovePolicy* and *DenyPolicy*. These events are responsible for changing the state of the fluents. Then we define some axioms. *Initiates(ApprovePolicy(policy), PolicyIsPermitted(policy), time)* states that if event *ApprovePolicy* happens at time *t* then *PolicyIsPermitted* would hold true at time *t + 1*. *Initiates(DenyPolicy(policy), PolicyIsDenied(policy), time)* states that if *DenyPolicy* happens at time *t* then *PolicyIsDenied* would hold true at time *t + 1*.

To illustrate an example of the policy composition, we populate the policy meta-model with the rules of AWS, GCP, and Azure. The rules had been defined as described in the previous section. Policy model 2 shows the model for policy specification. In the model, we have included the already defined rules for AWS, GCP, and Azure.

Policy Model 2 (Policy Specification)

```
;Load generic models for rules/policies and instantiated rules
load includes/rules/...      load includes/policy/...
load includes/rules/defined/RuleAWS... /RuleGCP/RuleAzure/RuleOrg.e

policy CompositePolicy
PolicyHasRule(CompositePolicy, RuleAWS/RuleGCP/RuleAzure...).
;Goal: Decide if the policy is permitted/denied
HoldsAt(PolicyIsPermitted(policy),3) | HoldsAt(PolicyIsDenied(policy),3).
```

First, we include the rule specification for the cloud providers (in this case, AWS, GCP, and Azure). For illustration purposes, only one rule from each cloud service provider was used in the model above, but any number of rule specifications could be included in the policy model. Next, we define a sorts *CompositePolicy* that represents the rules that the combined policy could represent. We then initialize some predicates. *PolicyHasRule(CompositePolicy,RuleAws0)* has a rule from AWS in the composite policy, *PolicyHasRule(CompositePolicy,RuleGCP0)* adds a rule from GCP to the composite policy, and *PolicyHasRule(CompositePolicy,RuleAzure0)* adds a rule from Azure to the composite policy. Finally, we define the goal for the reasoner. It decides whether the composite policy is permitted, denied, or is not applicable.

The result from DECReasoner is shown in evaluation 3. Since the rules for AWS were intentionally

created to match with the request for an example, they result in a *Match* event. The rules for GCP and Azure result in a *Mismatch* event. As the combination algorithm was designed to permit the policy even when one rule holds, at time point 2 the *ApprovePolicy* event happens at the policy gets permitted.

Solution 3 (Policy evaluation result by DECReasoner)

```
0
RuleEffectIsPermit(RuleAWS/RuleAzure/RuleGCP...).
Happens(Match(RuleAWS), 0).
Happens(Mismatch(RuleAzure/RuleGCP/RuleOrg), 0).
1
+RuleTargetHolds(RuleAWS).
Happens(ApproveRule(RuleAWS), 1).
Happens(RuleDoesntApply(RuleAzure/RuleGCP/RuleOrg), 1).
2
+RuleIsPermitted(RuleAWS).
+RuleIsNotApplicable(RuleAzure/RuleGCP/RuleOrg).
Happens(ApprovePolicy(CompositePolicy), 2).
3
+PolicyIsPermitted(CompositePolicy).
```

5.5 Synthesis

As discussed earlier, in Multi-Cloud environments the authorization policies can constitute of different formats. Due to which, the policy aggregation process becomes difficult and can also result in conflicting policies. Our approach focuses on the use of ABAC model that allows us to easily aggregate the policies written in RBAC, UBAC or any other access control models. ABAC model is more adaptable as compare to other models as it can subsume RBAC and UBAC models. We can easily represent a role or a user as an attribute in a policy. For example, *Alice* as a user or as a role *ProjMembers* refer to the same attribute that represents a *Subject*.

The Algorithm 1 explains how a policy of GCP, AWS or Azure that can be of UBAC or RBAC model, be mapped to ABAC model. The algorithm is generic and can incorporate policies by any cloud provider. It takes a policy P to be mapped to ABAC model, P_{ABAC}, and the type T of cloud provider. If policy belongs to GCP or Microsoft Azure, then it follows RBAC model and the role defines the action that can be performed on a resource. Then, the role has a binding with the user that is represented as a member in GCP or Microsoft Azure policy. Similarly, AWS provides UBAC model for the policies where *principal* represents the user that can be mapped to the *subject* in ABAC model. The resource will be mapped to the object of P_{ABAC} and action that is allowed on the resource will be mapped to action of P_{ABAC}. This generic algorithm can be extended to map policies by any cloud provider.

Algorithm 1 *mapping* function for converting an access control policy to ABAC.

Require: P is the policy that should be mapped to an ABAC model
Require: T is the type of cloud provider
1: **procedure** MAPPING($P \rightarrow P_{ABAC}$)
2: **if** $T = GCP \vee T = AZURE$ **then**
3: $P_{ABAC}.subject = P.member$
4: $P_{ABAC}.object = P.role.resource$
5: $P_{ABAC}.action = P.role.action$
6: **else if** $T = AWS$ **then**
7: $P_{ABAC}.subject = P.principal$
8: $P_{ABAC}.object = P.resource$
9: $P_{ABAC}.action = P.action$
10: **end if**
11: **return** P_{ABAC}
12: **end procedure**

As we have discussed the IAM services of major cloud providers in the Section 3.1, they all have basic components that help us map their policies to our generic model. For example, if *Alice* is a user who has an access to *read* a *resource* is mapped to a *sub-*

ject performing an *action* on an *object* as shown in our examples above. Alice can also be represented as a member of a role and this relationship is clearly reflected in the *input.e* file. Therefore, our generic model ensures the semantic correctness of the policies.

6 Authorization Conflicts in Multi-Cloud Environments

The authorization conflicts that can arise in multi-cloud environments can be categorized into syntactic and semantic conflicts.

6.1 Syntactic Conflicts

Syntactic conflicts consist of namespace conflicts and full and partial request/ response. Namespace conflicts can be handled by syntactically matching policies from different cloud providers. For example, in the case of AWS, all resources are prepended with *arn:*. This prefix identifies that the resource originates from AWS. Intra-cloud namespace conflicts are highly unlikely since each cloud controls its own policy specification. Conflicts with full and partial request/response can be handled ABAC as it has the ability to handle unforeseen circumstances.

6.2 Semantic Conflicts

Semantic conflicts consist of policy conflicts, policy redundancy, and different authorization models. In this work we will focus on semantic conflicts.

6.3 Policy Conflicts

Policy conflicts can arise when the decision returned by a policy is ambiguous, that is, one policy returns permit and the other returns deny. In the case of each individual cloud it is unlikely for these conflicts to occur since they have full control of their resources. This can be handled by inter and intra-policy conflict identification as discussed in [27] for AWS IAM. In the case of multiple clouds however, policy aggregation may lead to conflicts. A method to handle this is to alert when policies are conflicting. This can be handled with the model shown in 3. We create an event *InvalidatePolicy* and a fluent *PolicyIsInvalid*. We then specify that the event *InvalidatePolicy* occurs when one rule is permitted and the other is denied.

Policy Model 3 (Updated meta-model for policy conflicts)

```
...
fluent PolicyIsInvalid(policy) event InvalidatePolicy(policy)
Initiates(InvalidatePolicy(policy), PolicyIsInvalid(policy), time).

;Policy is invalid if the rule outcome is conflicting
Happens(InvalidatePolicy(policy), time) -> {rule1, rule2}
PolicyHasRule(policy, rule1) & PolicyHasRule(policy, rule2)
& HoldsAt(RuleIsPermitted(rule1), time) & HoldsAt(RuleIsDenied(rule2), time)
...
```

In the model above, first some fluents and events are defined. The main logic of the model lies in the conditions below these. The condition states that an *InvalidatePolicy* event occurs when a policy has multiple rules but one rule is permitted while the other is denied. Since this will check all possible pairs of the rules, any two conflicting rules will result in an invalid policy.

Alternatively, we can assign precedence to the policies, so that whenever a conflict arises between the policies, a higher priority policy would be applied due to precedence. For example, the RuleOrg should be given preference.

Policy Model 4 (Updated instantiated model for policy conflicts)

```
...
;Policy is permitted iff the rule from the organization says so
Happens(ApprovePolicy(CompositePolicy), time) & PolicyHasRule
(CompositePolicy, RuleOrg) -> HoldsAt(RuleIsPermitted(RuleOrg), time)
...
```

6.4 Policy Relationships

In policy redundancy, we can have conflicts that are based on syntax or semantics of the policies. The rules in the policy having same set of subject, object, effect and contextual information can be checked syntactically for redundancy and can easily be removed. However, when the redundancy is associated with the semantic attributes, in such a case syntactic comparison would not be useful. Therefore, a rule that does not change the consequence of a policy decision can be marked as redundant rule.

For example if a composite policy has rule that gives *Alice* the permission to write on resource *R* and there is another rule in the same policy stating that the role (eg. GlobalAdmin) of Alice provides the user with the write access on resource *R*. This means that the first rule can be considered redundant. This scenario identifies that the authorization rule had a similarity between the subject of the two rules that resulted in redundancy. We can also include actions, objects and even context to extend the criteria. Identification and removal of redundant policies is a complicated process and it is out of scope of our work.

6.5 Authorization Model Conflicts

Heterogeneity in policy models can result in inconsistencies. The cloud providers use different types of authorization models that result in conflicts. For example, Azure and GCP only provides RBAC model while AWS supports both UBAC and RBAC. To address this conflict we have used ABAC model as it subsumes both UBAC and RBAC. Our ABAC model is explained in the previous section. Another reason for the policy aggregation conflicts can be different implementations of authorization models by different cloud providers. For instance, Azure, AWS and GCP all provide RBAC but their semantics and implementations are different. For example, the concept of Policy is inconsistent in different cloud providers. In Azure, the concept of Policy is related to resources while in AWS, a Policy contains authorization rules that is assigned either to a User or a Role. On the other hand, in GCP it is just a binding of users (members) to the roles.

Furthermore, in AWS a decision to a rule can be explicitly specified to either Permit or Deny while in GCP and Azure it can only be Permit and in case of Deny, the rule must not be present. To further complicate rule aggregation, a rule in Azure contains nonactions that specifies what actions are excluded. Due to this, our EC models need to be updated such that if a policy are evaluated as *NotApplicable*, the policy is considered as *denied*.

Policy Model 5 (Updated Rule Combining Algorithm)

```
;Deny if all the rules are either Denied or NotApplicable
Happens(DenyPolicy(policy), time) & PolicyHasRule(policy, rule) ->
HoldsAt(RuleIsDenied(rule), time) | HoldsAt(RuleIsNotApplicable(rule), time).
```

7 Implementation

Browsing and collecting the policies from Cloud service providers is a monotonous and error-prone task since they can be as large as 500 rules for a GCP viewer role. To automatically process the policies of Cloud providers, we have developed a Web application[3] that performs the process of fetching the policies. It has been developed using Django, python 3.6.2 and Bootstrap 4.0.0.

A cloud service provider is selected at the start of the Web application. At this step, Add Provider button is available for the user to add a provider. On the click of this button, a popup dialog box appears where a dropdown menu provides different options for the available cloud service providers as shown in Fig. 4. Once the user selects the required cloud service provider, the web application dynamically loads the next step as follows:

[3]Source code and implementation details are available at https://www.icloud.com/iclouddrive/0cwdSlable8lHOX_NyBRhf5SA#nordsec.zip

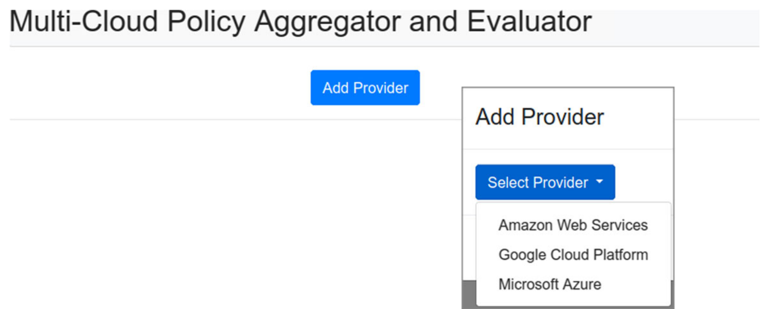

Fig. 4 Selecting a cloud service provider

- AWS: Web application requires access key and secret key from the user.
- GCP: Resource ID and API key are required from the user.
- Azure: Tenant ID, client ID, key, and subscription ID are required from the user.

On the click of Fetch and Convert Policy button, it fetches all the policies from the cloud service provider and displays them in their native format as shown in Fig. 5. This steps also converts the fetched policies to EC models and displays the translated polices in their respective text-boxes as shown in Fig. 6. As policies are made up of rules, the textboxes show the rules that the policy is comprised of (the structure of the policies is defined by the cloud service provider and is described in Section 3.1). The user can aggregate and view the aggregated policy by clicking the *Aggregate Policy* button. On the click of Invoke DECReasoner button, the aggregated policy is sent to DECReasoner and the result is displayed in a separate text box. These steps have been shown in Fig. 7. For illustration purposes, Fig. 7 shows the aggregation and results of only two policies from each cloud service provider.

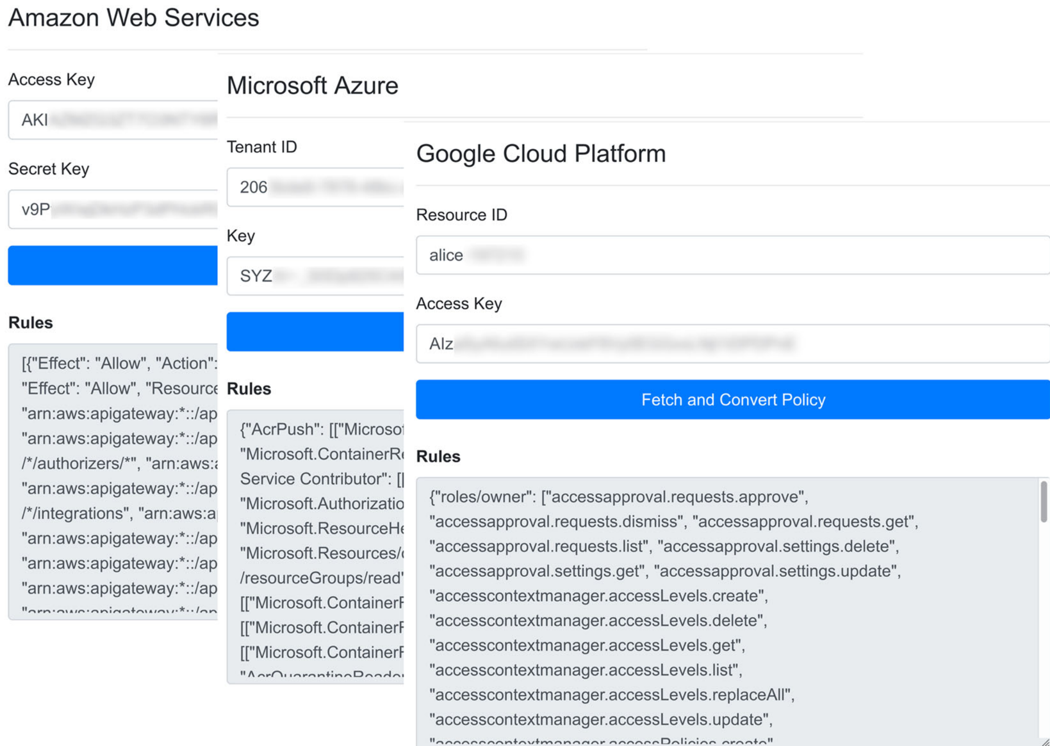

Fig. 5 Fetching rules from the cloud service providers

rule Azure10

[time] Happens
AZAcrImageSi
= Write.

[time, subject,
AZAcrImageSi
!= Write.

HoldsAt(RuleE

rule Gcp14

[time] Happens
object = Acces

[time, subject,
Alice | object !=

HoldsAt(RuleE

rule Aws37

[time] Happens(Match(Aws37),time) -> {subject, object, action} subject = Alice & object = ApplicationautoscalingdescribeScheduledActions & action = Allow.

[time, subject, object, action] Happens(Mismatch(Aws37), time) -> subject != Alice | object != ApplicationautoscalingdescribeScheduledActions | action != Allow.

HoldsAt(RuleEffectIsPermit(Aws37),0).

Fig. 6 Translated rules

8 Performance Evaluation

In order to measure performance and and scalability of our approach, we have created different test cases and evaluated them on Amazon EC2 c5.2xlarge instance having 8 vCPUs and 16 GiB memory. The Amazon Machine Image (AMI) used was *Ubuntu Server 20.04 LTS (HVM), EBS General Purpose (SSD) Volume Type*. We then setup the modified and improved *DECReasoner* version as proposed in [30] on the instance using *scp* to copy files over the ssh connection and using ssh to run the test cases. The proposed approach can be both used for design-time identification of the policy conflicts and for the access control decisions based on the actual requests, as evident by very encouraging performance evaluation results. For the test cases, we increased the number of rules and measured the time taken by the DECReasoner to

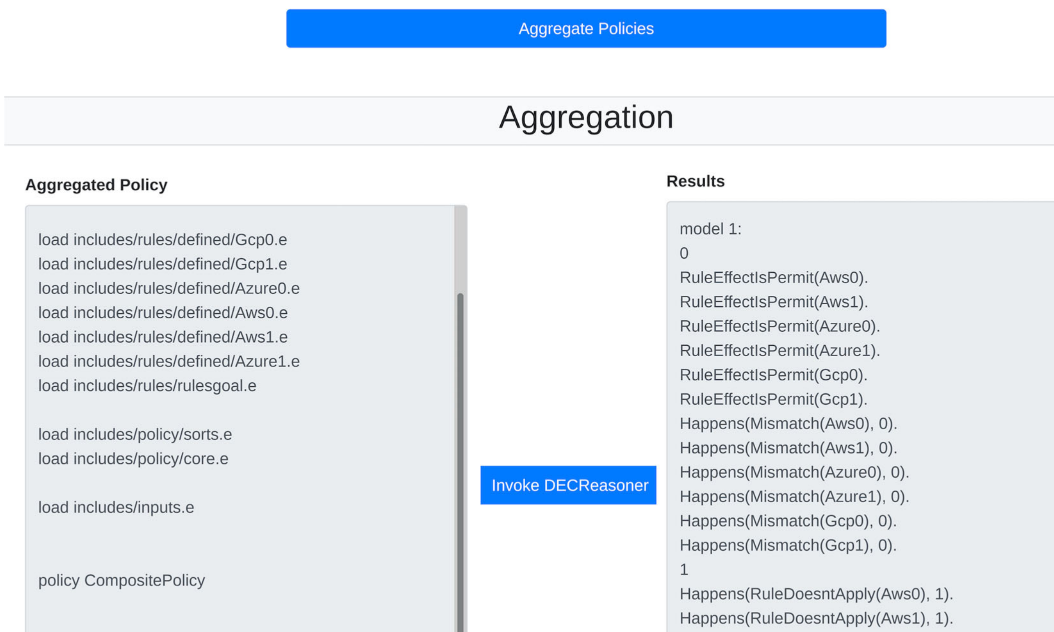

Fig. 7 Aggregating policies and invoking DECReasoner for the aggregated policies

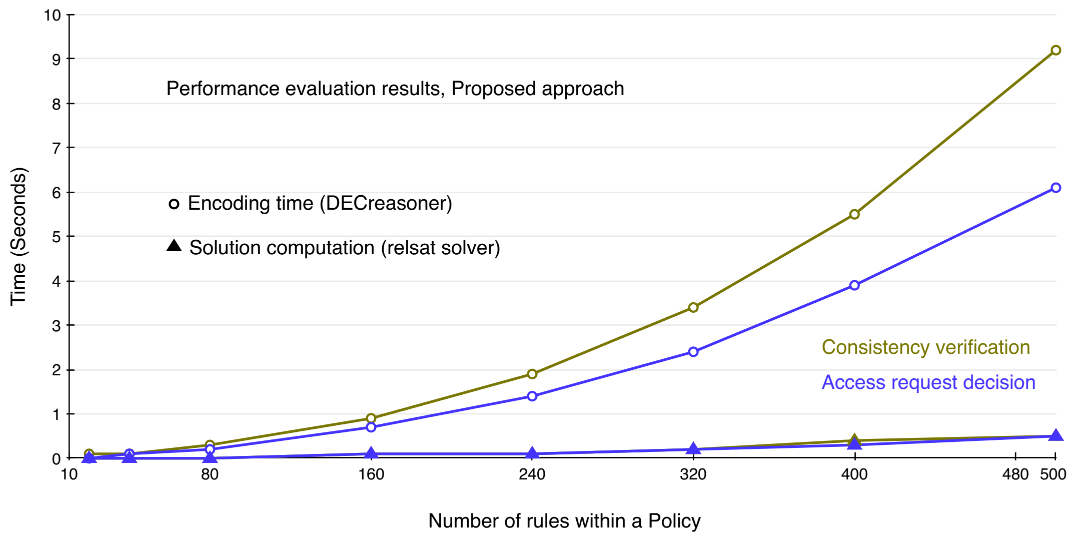

Fig. 8 Performance evaluation results - Proposed approach

encode the models in a SAT problem and then the time taken by the SAT solver for solution finding. We also noted the number of variables and clauses in the encoded SAT problem to compare the complexity of the models. The performance evaluation results are shown in Fig. 8 with x-axis showing the number of rules and the y-axis showing the time taken. The solution time is closer to zero for the simpler models and scales well. Similarly, the encoding time also shows promising results, increasing steadily with the increase in the number of rules within a policy.

When compared to previous work [29], we have updated the EC models as discussed in Section 5.2. This both results in a more expressive model and

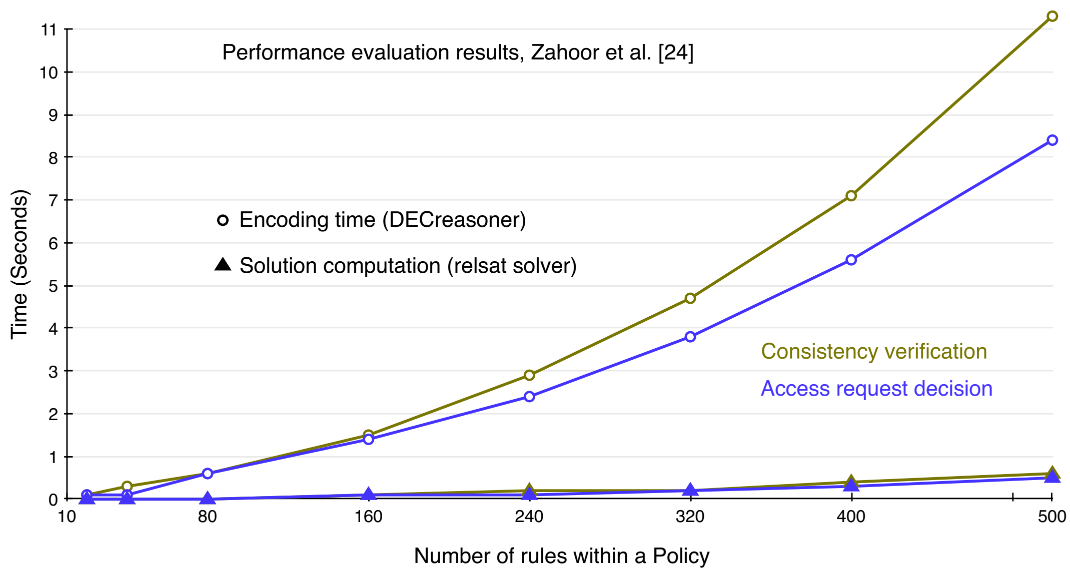

Fig. 9 Performance evaluation results [29]

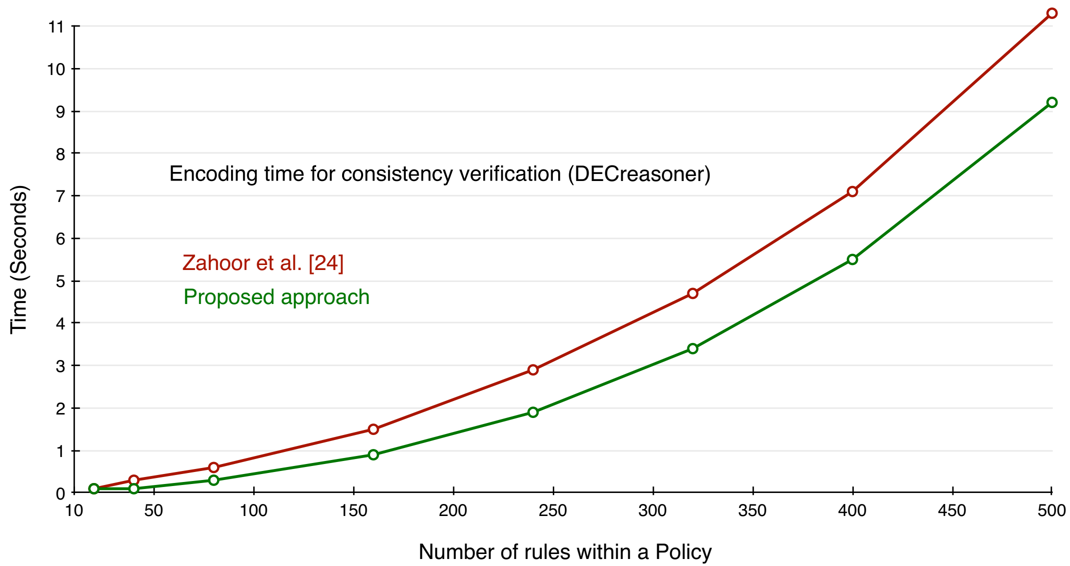

Fig. 10 Comparison of encoding time for consistency verification (DECreasoner)

significantly improves performance. The performance evaluation results for the same test cases on the same EC2 instance for the models presented in [29] are shown in Fig. 9, again with x-axis showing the number of rules and the y-axis showing the time taken.

As evident from the performance evaluation results, the proposed approach achieves performance improvement when compared to the existing work. This is further highlighted in Figs. 10 and 11 justifying that the proposed approach is more efficient when compared to [29].

Our work can also be compared to [20]. However the approach used in [20] used XACML policies and translated them to Answer Set Programming (ASP)

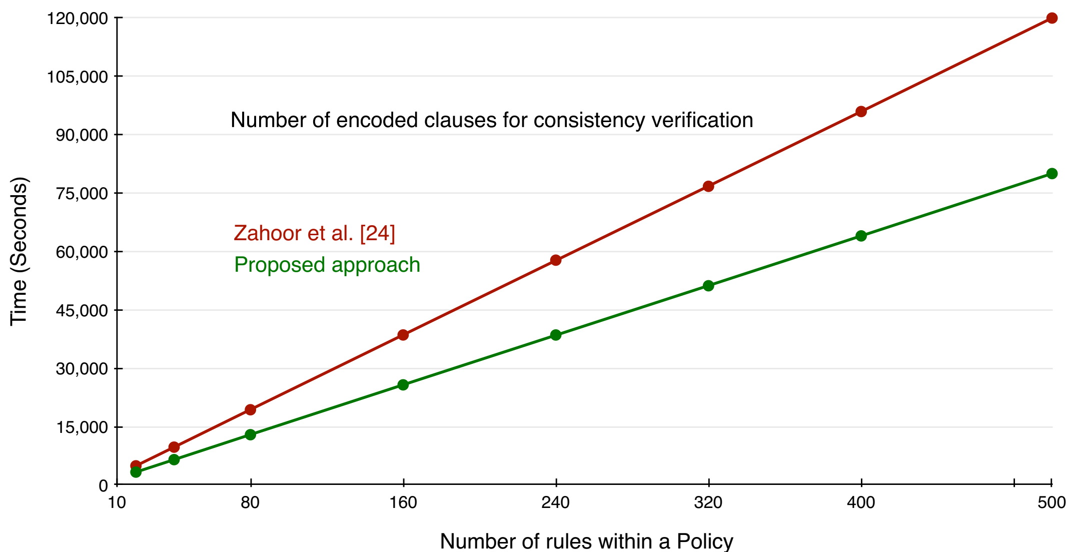

Fig. 11 Number of encoded clauses for consistency verification

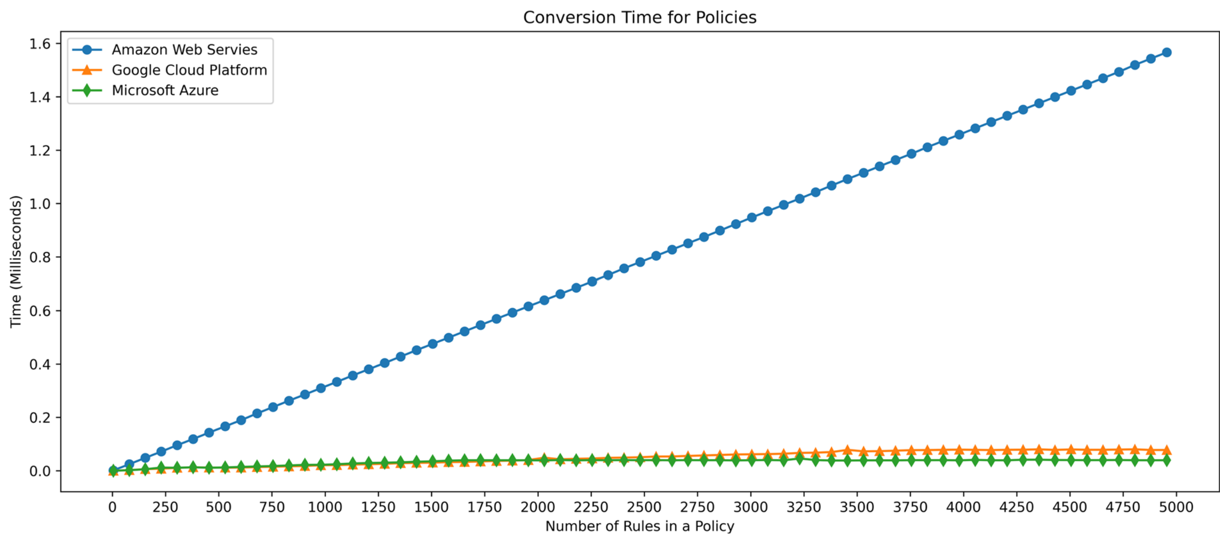

Fig. 12 Translation Time

problems. Our approach instead uses the actual policies from major Cloud provides and reasons about them.

In addition to the results described previously, we also evaluated the performance of our web application and calculated the overhead that was incurred for translation and the aggregation of policies. All tests were run 10 times and the time was calculated using the mean of the 10 runs. Figure 12 shows the time taken for the rules of AWS, GCP, and Azure to be translated to Event-Calculus models. Translation time is very reasonable since for all three cloud providers, the time taken for translation is in a few milliseconds. The increased time for AWS is caused due to the unpacking of policies to rules. Figure 13 shows the time taken to aggregate the modeled rules into a composite policy. From the graph, we can see that the aggregation time only takes a few microseconds and as such is reasonable. Aggregation mostly consists of file writing operations and the anomalous pattern can be explained by writing operations on the disk.

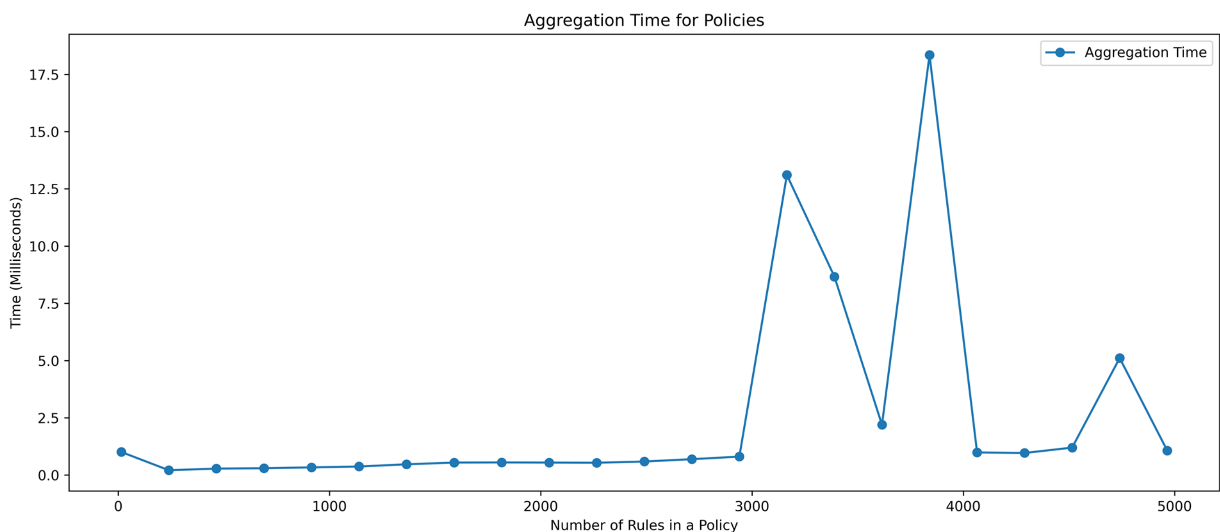

Fig. 13 Aggregation Time

9 Conclusion

Our work on authorization policies in Multi-Cloud environments focuses on the consistency management and their specification. The proposed approach provides aggregation of authorization policies for real-world cloud service providers having different authorization models or similar models with different implementations and different authorization policies having possible conflicts. The work in this paper presents a formal and generic approach based on ABAC model that can address different authorization models. It also identifies and categorizes the policy conflicts that include conflicts related to Authorization model, policy and policy relationships. During our emphasis on challenges and contributions, we consistently used different scenarios from AWS, GCP, Microsoft Azure and RightScale. To justify that our approach is scalable and practical, we have presented the performance evaluation results using our tool as well. The evaluation results are very encouraging and prove the scalability of our work. Furthermore, the performance evaluation of the web application have also been discussed.

Acknowledgments We would like to thank AWS educate for providing AWS Credits for carrying out this research.

Declarations Please consider following sub-title declarations as part of the submission process.

Availability of data and materials The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Competing interests The authors declare that they have no competing interests.

References

1. Al-Dahhan, R.: Efficient ciphertext-policy attribute based encryption for cloud-based access control. Ph.D. thesis, Liverpool John Moores University. https://doi.org/10.24377/LJMU.t.00011013. http://researchonline.ljmu.ac.uk/id/eprint/11013/ (2019)
2. Alansari, S., Paci, F., Sassone, V.: A Distributed Access Control System for Cloud Federations. In: International Conference on Distributed Computing Systems (2017)
3. AWS: http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html
4. Azure: https://docs.microsoft.com/en-us/azure/role-based-access-control/custom-roles
5. Bonatti, P.A., di Vimercati, S.D.C., Samarati, P.: An algebra for composing access control policies. ACM Trans. Inf. Syst. Secur. **5**(1), 1–35 (2002). https://doi.org/10.1145/504909.504910
6. Bouchaala, M., Ghazel, C., Saidane, L.A.: Toward Ciphertext Policy Attribute Based Encryption Model: a Revocable Access Control Solution in Cloud Computing. In: Kallel, S., Cuppens, F., Cuppens-Boulahia, N., Hadj Kacem, A. (eds.) Risks and Security of Internet and Systems, pp. 193-207. Springer International Publishing, Cham (2020)
7. Bryans, J.: Reasoning about Xacml Policies Using Csp. In: SWS, pp. 28–35 (2005)
8. Dang, T.K., Ha, X.S., Tran, L.K..: Xacs-dypol: Towards an xacml-based access control model for dynamic security policy (2020)
9. Elliott, A., Knight, S.: Role explosion: Acknowledging the problem. In: Proceedings of the 2010 International Conference on Software Engineering Research & Practice, SERP 2010, July 12-15, 2010, Las Vegas, Nevada, USA, 2 Volumes, pp. 349–355 (2010)
10. Google: https://cloud.google.com/iam/reference/rest/v1/Policy
11. Hu, V.C., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K.: Guide to attribute based access control (abac) definition and considerations. NIST Special Publication **800**, 162 (2014)
12. Kolovski, V., Hendler, J.A., Parsia, B.: Analyzing Web Access Control Policies. In: WWW, pp. 677–686 (2007)
13. Kowalski, R.A., Sergot, M.J.: A logic-based calculus of events. New Generation Comput. 4(1) (1986)
14. Mazzoleni, P., Crispo, B., Sivasubramanian, S., Bertino, E.: XACML policy integration algorithms. ACM Trans. Inf. Syst. Secur. **11**(1), 4:1–4:29 (2008). https://doi.org/10.1145/1330295.1330299
15. Mueller, E.T.: Commonsense reasoning. Morgan kaufmann publishers inc., CA USA (2006)
16. Nguyen, T.N., Thi, K.T.L., Dang, A.T., Van, H.D.S., Dang, T.K.: Towards a Flexible Framework to Support a Generalized Extension of Xacml for Spatio-Temporal Rbac Model with Reasoning Ability. In: ICCSA (5) (2013)
17. Pustchi, N., Krishnan, R., Sandhu, R.S.: Authorization federation in iaas multi cloud. In: Proceedings of the 3rd International Workshop on Security in Cloud Computing, SCC@ASIACCS '15, Singapore, Republic of Singapore, April 14, 2015, pp. 63–71 (2015). https://doi.org/10.1145/2732516.2732523
18. Ramya, P., Saraswathy, S., Sharmila, S.: Sivakumar, S.: T-Broker- a Trust-Aware Service Brokering Scheme for Multiple Cloud Collaborative Services. In: IEEE Transactions on Information Forensics and Security (2015)
19. Rao, P., Lin, D., Bertino, E., Li, N., Lobo, J.: An algebra for fine-grained integration of XACML policies. In: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, SACMAT 2009, Stresa, Italy, June 3-5, 2009, pp. 63–72 (2009). https://doi.org/10.1145/1542207.1542218
20. Rezvani, M., Rajaratnam, D., Ignjatovic, A., Pagnucco, M., Jha, S.: Analyzing XACML policies using answer set programming. Int. J. Inf. Sec. **18**(4), 465–479 (2019). https://doi.org/10.1007/s10207-018-0421-5

21. Sukmana, M.I.H., Torkura, K.A., Graupner, H., Cheng, F., Meinel, C.: Unified Cloud Access Control Model for Cloud Storage Broker. In: 2019 International Conference on Information Networking (ICOIN), Pp. 60–65 (2019). https://doi.org/10.1109/ICOIN.2019.8717982
22. Sun, W., Yu, S., Lou, W., Hou, Y.T., Li, H.: Protecting your right: Verifiable attribute-based keyword search with fine-grained owner-enforced search authorization in the cloud. IEEE Trans. Parallel Distrib. Syst. **27**(4), 1187–1198 (2016). https://doi.org/10.1109/TPDS.2014.2355202
23. Tsankov, P., Marinovic, S., Dashti, M.T., Basin, D.A.: Decentralized Composite Access Control. In: POST (2014)
24. Wei, J., Liu, W., Hu, X.: Secure and efficient attribute-based access control for multiauthority cloud storage. IEEE Syst. J. **12**(2), 1731–1742 (2018). https://doi.org/10.1109/JSYST.2016.2633559
25. Yang, K., Jia, X.: Expressive, efficient, and revocable data access control for multi-authority cloud storage. IEEE Trans. Parallel Distrib. Syst. **25**(7), 1735–1744 (2014). https://doi.org/10.1109/TPDS.2013.253
26. Yu, S., Wang, C., Ren, K., Lou, W.: Achieving secure, scalable, and fine-grained data access control in cloud computing. In: INFOCOM, pp. 534–542 (2010). https://doi.org/10.1109/INFCOM.2010.5462174
27. Zahoor, E., Asma, Z., Perrin, O.: A Formal Approach for the Verification of AWS IAM Access Control Policies. European Conference on Service-Oriented and Cloud Computing (2017)
28. Zahoor, E., Bibi, U., Perrin, O.: Shadowed authorization policies - A disaster waiting to happen? 11881, 341–355. https://doi.org/10.1007/978-3-030-34223-4_22 (2019)
29. Zahoor, E., Ikram, A., Akhtar, S., Perrin, O.: Authorization Policies Specification and Consistency Management within Multi-Cloud Environments. In: Gruschka, N. (ed.) Secure IT Systems - 23Rd Nordic Conference, Nordsec 2018, Oslo, Norway, November 28-30, 2018, Proceedings, Lecture Notes in Computer Science, Vol. 11252, pp. 272–288. Springer, Oslo, Norway (2018)
30. Zahoor, E., Perrin, O., Godart, C.: An Event-Based Reasoning Approach to Web Services Monitoring. In: ICWS (2011)
31. Zhu, Y., Huang, D., Hu, C., Wang, X.: From RBAC to ABAC: constructing flexible data access control for cloud storage services. IEEE Trans. Services Computing **8**(4), 601–616 (2015). https://doi.org/10.1109/TSC.2014.2363474

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.