# Semantically Rich, Context Aware Access Control for Openstack

Vishal Rathod, Sandeep Nair Narayanan, Sudip Mittal and Anupam Joshi
University of Maryland, Baltimore County, Baltimore, MD 21250, USA
Email: {vishalr1, sand7, smittal1, joshi}@umbc.edu

***Abstract*— In an open source cloud computing platform such as OpenStack, operators use Role-Based Access Control (RBAC) model to grant access to cloud resources. However, these user-level role-based access control techniques fail to include comprehensive user context. We believe a situational aware framework will improve security by bringing in user's context to such cloud systems. In this paper, we create a semantically rich context-sensitive access control system for OpenStack by incorporating the user's current context attributes like location, time, etc. In a proof-of-concept implementation, we integrate a knowledge graph with our own access control system to express and enforce the contextual-situation policies in OpenStack. The proposed system provides enhanced, flexible access control while minimizing the overhead of altering the existing access control framework. We also discuss various use cases, to highlight the benefits of our system and show enforcement results.**

***Index Terms*— Cybersecurity, Access Control, Knowledge Graphs, OpenStack, Contextual Attributes**

## I. Introduction

Cloud computing environments allow developers, to create accessible Internet-based ubiquitous services. Securing these environments from unauthorized users formulates the need for *Access Control*. Access control systems allow an application to verify the identity of a user or another application. Once authenticated they can create, read, write, or append any resource or object. Generally, security administrators appropriate roles, and these roles are used to authorize users and applications via access control policies. These policies play a vital role in regulating the behavior and functionality of the system.

Policies control access rights within an organization, they also dictate an organization's operational behavior. For example, a policy may require that an organization has to encrypt its data as per some required minimum standard, say, AES 256 while communicating with the outside world. It may further restrict the movement of data by designating specific intermediate routers. A challenge for both policies and descriptive access control models is to be able to represent fine-grained context, this has led to the development of various access control frameworks. Some such frameworks are: Mandatory Access Control (MAC) [10], Discretionary Access Control (DAC) [19], Role-Based Access Control (RBAC) [6] and Rule-Based Access Control [16]. Among these Role Based Access Control (RBAC) model is the most widely used access control model [20]. Many applications and platforms use a custom form of RBAC as per their needs and requirements. Major cloud computing platforms such as *OpenStack* [21] utilize RBAC as their authorization system.

ABAC, on the other hand, combines various attribute information to provide greater flexibility. It allows a system to express fine-grained control policies in a simple and more powerful way [8]. The attribute information can be based on user, subject, and resource properties [7]. This inherent flexibility permits the creation of complex real-world applications and systems.

So as to achieve this, we can add *contextual attributes* to role centric system. A role-centric system can be augmented with the user, environmental, application-oriented, or a combination of these contextual attributes [5]. The system-state machine can be further differentiated based on these contextual attributes. Typically, the user context comprises of a user's location, profile, rights, the current authorization, time, actions, etc. Similarly, the environmental context can contain infrastructure, physical properties, and restrictions. This can include, the system's current time. Application context defines the system flow to fulfill various application-specific needs of a user. An example of such a scenario can be, permitting communication only on an AES 256 encrypted connection.

In this work, we develop a *role-centric attribute-based access control model for Openstack*. We extend the existing RBAC in OpenStack [22] with contextual attributes, by adding them to core cloud access control systems. We create policy representations where context-dependent information is included in the access control models. In our system, we can capture many of the contextual attributes like location, time, etc.

The system utilizes a knowledge graph, which replaces the existing access control module of OpenStack. It acts as an authorization component by getting information and permissions from the knowledge graph and then evaluates authorization decisions. The OpenStack sub-modules then enforce the decisions taken by our system. Our system is versatile enough, so as to enforce different types of access control policies. To facilitate communication between OpenStack and our system, we implement a RESTful service. This service takes an input API request from OpenStack, along with the contextual information, and returns the access control decision.

The rest of the paper is organized as follows – Section II discusses our related work. We present our system's architecture and explain different components in Section III. Various use-cases in Section IV. We evaluate our system in Section IV-C. We discuss our future work and conclusion in Section V.

## II. Related Work & Background

In this section, we discuss some related work on access control and give a brief background on OpenStack.

### A. Access Control

Attribute based access control in cloud systems has been studied with different traditional access control models like DAC [19], MAC [10], and RBAC [6]. These models are identity-centric, and identification is usually done through role assignment [23]. These models, however, are suitable for static systems, with a limited set of users and services [14]. These systems are mostly centralized [9]. Policies in attribute-based access control (ABAC) models are semantically more expressive than the RBAC model [8]. Jin et. al. [10] created a combined attribute-based access control model that can be configured with DAC, MAC, and RBAC [11]. They also created a proof of concept using an OpenStack implementation, where they replaced the native RBAC [22].

Our approach is designed to incorporate contextual attributes in OpenStack's existing RBAC framework. Other approaches to include attributes in OpenStack for cloud federation and identity management have been discussed by Chadwick et al. [4] and by Lee et al. [15].

Pustchi et al. [18] discussed an application of attribute-based access control to enable collaboration between tenants in a cloud IaaS platform. In our technique, we focus on authorization within a single tenant. A formal role-centric attribute-based access control (RABAC) model has been proposed by Jin et al. [12], along with XACML profiles. XACML is a general-purpose access control policy language for managing access to resources. Joshi et al. [13], have proposed a semantically rich access control system that evaluates access decisions based on rules generated using the organization's confidentiality policies. Their proposed system analyzes the multi-valued attributes of the user and the requested document, before making an access decision.

### B. OpenStack

OpenStack is an open-source cloud middleware that helps orchestrate a cloud environment and provides various services by virtualizing network, storage, and compute resources. It also keeps the different offered services decoupled, and provides high scalability and availability.

In this paper, we use OpenStack's *Nova* component to demonstrate our access control extension. Nova is an OpenStack service that provides custom compute instances. An operator can access multiple servers and OS services by making API requests to the compute service. In the current implementation, the compute service is integrated with the OSLO policy engine [1], [17], as shown in Figure 1, to define access control policies and to compute various access decisions. Whenever a user wants to access a specific resource, it submits a request to the OSLO policy engine. The OSLO engine, using different policies defined, makes a decision to allow or deny access.

Each OpenStack service defines the access policies for its resources (for example, API access, the ability to attach to a volume, ability to fire up instances, etc.) in an associated policy JSON file. The policy file stores various rules for which some roles have access to certain resources. Each API call requires a corresponding permission structure to be present in the policy file and dictates what level of access is allowed. The syntax for the same is

$$api\_name : rule\_statement$$

. For example, a sample expression

$$\text{“}identity : create\_server\text{”} : \text{“}role : admin\_or\_owner\text{”}$$

translates to: you must have admin or owner role for creating a new server through the compute service. Administrators will build access grant rules using default deny policy architecture[3] where the destination will be a set of critical API operations of OpenStack and action results will be set to “Grant Access”. This procedure is analogous to a firewall policy where everything is dismissed unless explicitly approved.

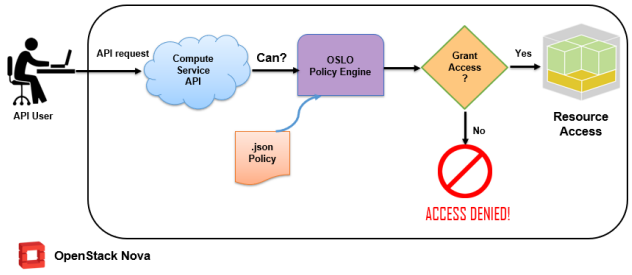


Fig. 1. OpenStack Policy Engine Architecture

However, these access policies are rigid and do not capture many of the contextual attributes like location, time, etc. In this paper, we provide an extension for this existing access control in OpenStack by incorporating contextual attributes and reasoning capabilities.

## III. An Attribute-based Extension for OpenStack

Integrating access control models and policy specification language like Web Ontology Language (OWL) gives us a powerful architecture to formulate complex policies. The associated reasoning framework can then be used to make intricate access control decisions. Our system replaces the existing policy engine in OpenStack and uses various user attributes like time, location, etc. It then reasons over a knowledge graph to deliver access decisions to different OpenStack services. Figure 2, describes how our proposed extension alters the existing OpenStack framework to make better access control decisions using contextual attributes. Our system bypasses the existing OSLO policy engine and processes access control requests using two modules; the Request Preprocessing module and QueryEngine module. The Request Preprocessing module combines users request with contextual attributes and converts it to an RDF format. The access control decision will be then taken by QueryEngine module by reasoning over the input attributes and a carefully crafted knowledge graph.

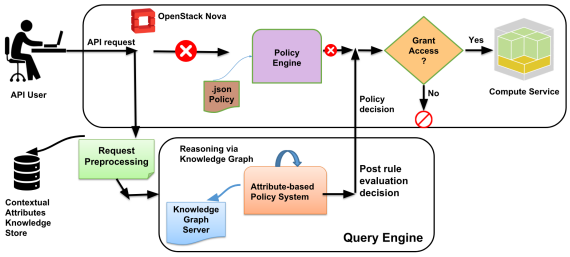


Fig. 2. Extended System Architecture

### A. Request Preprocessing Module

The input to Request Preprocessing module is the request from the user for granting access to a specific resource. In our implementation, we bypass the input to the OSLO policy engine to this module. This Module serves mainly two purposes, first, it will aggregate the requesting user's contextual information using the knowledge store attached to it, as shown in Figure 2. The information retrieved includes contextual information about the user like current location, time, host IP, etc. As an initial step, this knowledge store needs to be populated with contextual attributes. This store can be populated in different ways. It can be generated from the request itself (for example location can be obtained from IP addresses geo-location) or the information can be retrieved from other OpenStack projects. For example, if the current system is configured to use with OpenStack services like *keystone*[1] by implementing the OpenStack Identity API[2], user-related information can be fetched from them. The second function of this module is to convert the request and the information gathered from the knowledge store to RDF format for further processing. The generated RDF will then be passed on to the Query Engine module, where the access control policies are evaluated and decisions are generated.

### B. QueryEngine Module

Query Engine module reasons over a knowledge graph and the input from the Request Preprocessing module and generates access control decisions. The inferred decisions are then returned back to the caller in our architecture. In this architecture, we utilize the Query Engine as a centralized policy administration module that returns a set of user permissions on objects based on the rules definitions. Query Engine initially verifies the project in the target and operation to be performed. Further, it makes the access control decisions by calling different program functions of Query engine utilizing contextual attribute extension for OpenStack. First, it populates the knowledge graphs from the RDF inputs received from the preprocessing module. Then a standard reasoner like Pellet[3] is used to reason over the knowledge graph to determine access control decisions. Our implementation also allows to specify rules using SWRL[4].

[1]https://docs.openstack.org/keystone/pike/
[2]https://developer.openstack.org/api-ref/identity/index.html
[3]https://www.w3.org/2001/sw/wiki/Pellet
[4]https://www.w3.org/Submission/SWRL/

The core component of this module is the knowledge graph which abstracts the domain information. Figure 3 represents a sample knowledge graph which we generated for our proof of concept implementation. The knowledge graph represents many classes. One class *Role* class which specifies different roles in an organization like *admin*, *staff*, etc. This is similar to the OpenStack's RBAC roles. Other user context related classes include *Location* class and *Time* class which define the location and time of the entity. Yet another important class defines the *Operations* class which defines all the operations available in the system.

Each of these classes is arranged in carefully crafted sub-classes. For example, Work location, and NonWork locations are sub-classes of the main *Location* class and the sub-classes of *Operations* class include *Critical* and *Normal*. Such carefully designed sub-classes can improve the expressiveness of the policies. For example, some operations can be allowed only from work locations (e.g. Printing a confidential document, reboot the physical server, etc.). With these subclasses defined, administrators can easily define rules like *Critical Operations are allowed only if the requesting user's location is an instance of a Work location* using the SWRL specification. The presence of knowledge graph also helps to use the same knowledge graph with different OpenStack services because of the commands are specific to each service in OpenStack such as Nova, Glance, Cinder, etc. Other standard policies, typically listed in OpenStack policy files can also be expressed using similar rules.

## IV. Use-case Scenarios & Evaluation

In this section, we describe two scenarios; access control using OSLO policy engine (Section IV-A) and access control using our attribute-based extension (Section IV-B). Later in Section IV-C, we evaluate our implementation by comparing the time taken in both these scenarios. Our results show that both scenarios take comparable time in these simple scenarios. While we acknowledge that our results only shows simple scenarios, it should be noted in actual systems, there would be number of simple scenarios.

### A. Simplified RBAC Policy Model with OSLO Engine

In this scenario, we use a subset of roles and resources from OpenStack. The policies used include two roles (Admin and Staff) and four Nova commands: compute extension-create, compute extension-delete, compute extension-reboot, and compute extension-show.

The permissions are determined by the OSLO policy engine (which use RBAC policies) based on the user's role assigned for a specific project. The rules for each command, for a generic user $u$, are given below.

$$Roles : \{Admin, Owner\}$$

Authorization rules for any user u:

- Compute extension-list:

$$Role(u) = Admin$$

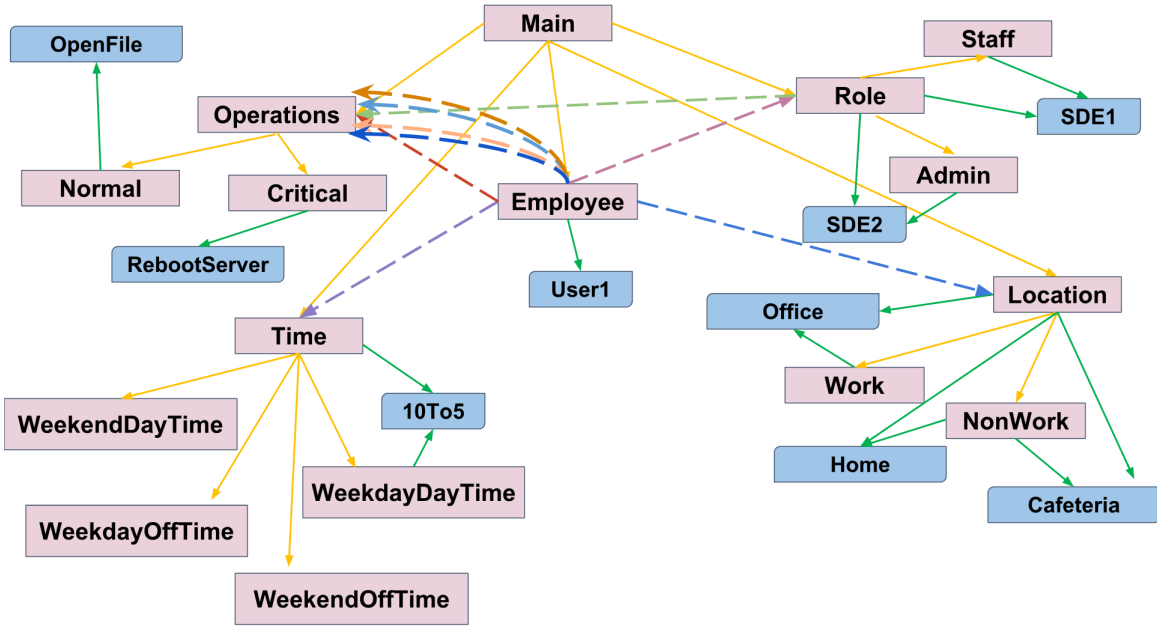


Fig. 3. A Sample Knowledge-Graph utilized in our access control model. We also show an instance 'User1' asserted in the knowledge graph.

- Compute extension-create:

$$(Role(u) = Admin \wedge Role(u) = Owner)$$

- compute extension-delete:

$$Role(u) = Admin \wedge Role(u) = Owner$$

The above rules assert that a user must have an $Admin$ role to list existing servers, whereas to perform the compute extension-create and compute extension-delete server operations the user is required to be an $Admin$ or $Owner$. To enforce the authorization policy, we define a similar policy to approve each of these commands in our system. There are two roles defined an $Admin$ and an $Owner$ along with the relationship between commands and roles via operation sets.

### *B. An attribute based Role-Centric Access Control*

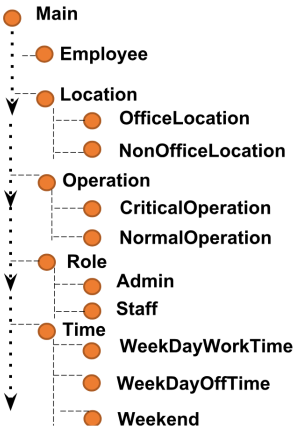


Fig. 4. Some context Attributes used in the sample scenario

This section describes how our extension can be used to generate policies integrating contextual attributes like time and location with exiting role-based policies. In this scenario, we use the same roles and commands as in Section IV-A. Besides roles and commands, we use the contextual attributes location, operation severity, and time while making access control decisions. For simplicity we have considered location attributes within the range of values $\{OfficeLocation, NonOfficeLocation\}$, time attribute in the range of $\{WeekdayWorkTime, WeekdayOffTime, Weekend\}$, and the Operation-severity attribute having the values in range $\{CriticalOperation, NormalOperation\}$, as shown in Figure 4. Contextual attributes are atomic valued, unlike roles which imply that a user can have multiple roles assigned but it can have only one location and time value. For any user, accesses are defined based on their roles and their associated user and operation attributes. The SWRL rules which are used in the query engine are enumerated below.

```
Rule 1: CriticalOperations(?opr),
    Employee(?emp), hasRole(?emp, ?rle),
    Admin(?rle) -> hasRoleBasedAccess(?emp,
    ?opr)

Rule 2: CriticalOperations(?opr),
    Employee(?emp), hasLocation(?emp, ?loc),
    OfficeLocation(?loc) ->
    hasLocationBasedAccess(?emp, ?opr)

Rule 3: WeekdayDayTime(?time), hasTime(?emp,
    ?time), CriticalOperations(?opr),
    Employee(?emp) ->
    hasTimeBasedAccess(?emp, ?opr)

Rule 4: hasRoleBasedAccess(?emp, ?opr),
```

```
    Operation(?opr),
    hasTimeBasedAccess(?emp, ?opr),
    Employee(?emp),
    hasLocationBasedAccess(?emp, ?opr) ->
    hasAccess(?emp, ?opr)
.
```

The first three rules specify sample role-based, location-based, and time-based rules. For instance, Rule 2 is an example of a location-based policy in which an employee '?emp' is authorized to all the instance of $CriticalOperation$ class only if the employee has the location which is the instance of an $OfficeLocation$ class. Similarly, many complex role-based, time-based access, and location-based rules can be evaluated using the reasoning framework. Rule 4 is then used for making an overall decision i.e., an employee is allowed to execute the operation if he has the location, time, and role-based accesses are granted. The QueryEngine in our extension will take in the inputs from the preprocessing engine and reasons over the knowledge graph and rules to the final generate access grants.

In figure 5, we have constructed a sample username: '$User1$' and assigned few property values to it signifying its current contextual attributes. This structure will be represented in RDF format by the preprocessing module.

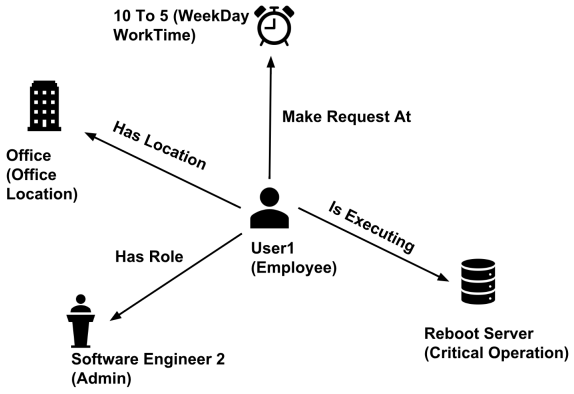


Fig. 5. RDF for an instance of Class Employee

Figure 5 implies that User '$User1$' $hasLocation$ '$OfficeSpace$' (an instance of Location class), $makeRequestAt$ '$10to5\_Weekday$' (an instance Time class), and $hasRole$ '$SoftwareEngineer2$' (an instance of Admin class which is a subclass of Class Role). Now assume that $User1$ request access to operation $RebootServer$. The first level of reasoning infers that $User1$ is an $admin$ user, since $SoftwareEngineer2$ is it instance and $RebootServer$ is a critical critical operation. The final inference is to grant access to $User1$ and it is depicted in Figure 6. Our system can be extended by mentioning more users, roles, and operations available in OpenStack.

### C. Evaluation

In this section, we evaluate the time required for authorization in the existing OpenStack access control framework and a contextual attribute-based policy extended OpenStack

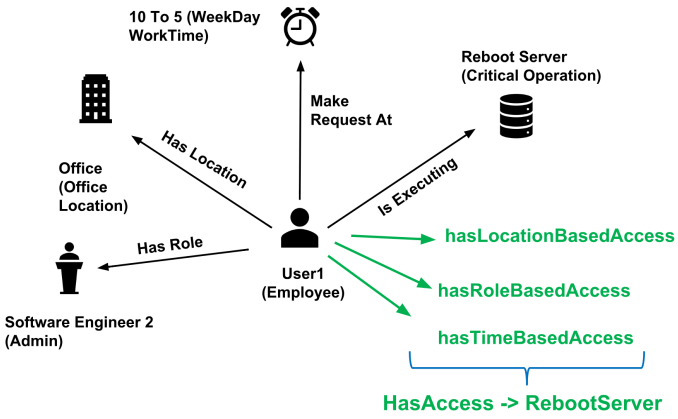


Fig. 6. Inferred results on reasoning policy rules

access control framework by utilizing the Knowledge Graph and our Policy System. For such a comparison, we take the two use-case scenarios mentioned in Section IV-A and Section IV-B.

To compare the performance, we developed a python script which requests for a specific resource $n$ number of times. First, we ran the script against our first scenario, using OSLO policy engine, and collected the time taken for a number of requests ranging from 10 through 100 with an interval of 10. The experiment is then repeated for the second scenario, with our proposed extended query engine, for the same range of requests. A graph is then plotted, as shown in Figure 7 using the results from both the experiments with the number of requests on the x-axis and time taken in seconds on the y-axis

We develop a Python script to examine the performance of Nova API commands and ascertain the time needed for the set of requests (compute API service operation). The graph shown below represents the overall time for the set of requests executed by an OpenStack user. Our main aspiration is to evaluate the time needed for authorization in existing OpenStack access control framework and a contextual attribute-based policy extended OpenStack access control framework by utilizing the Knowledge Graph and our Policy System.

These graphs comprise of two curves, first for an RBAC policy in OpenStack without any modifications (OS RBAC), and second for our contextual-attribute enhanced role-centric policy in OpenStack with Knowledge Graph and our policy system. The overall request response time for these two cases are quite similar since the access grant decision check time for each of these cases are not significantly different from each other. As we are using upgraded knowledge graph and policy system features, there is not much latency in policy evaluation time compared to old platforms like 'cwm engine' of REIN policy framework [2].

## V. Conclusion

In this paper, we create a context-aware access control system for OpenStack. We created a proof of concept, utilizing a knowledge graph and a policy engine. Including contextual attributes while evaluating access decision makes the system

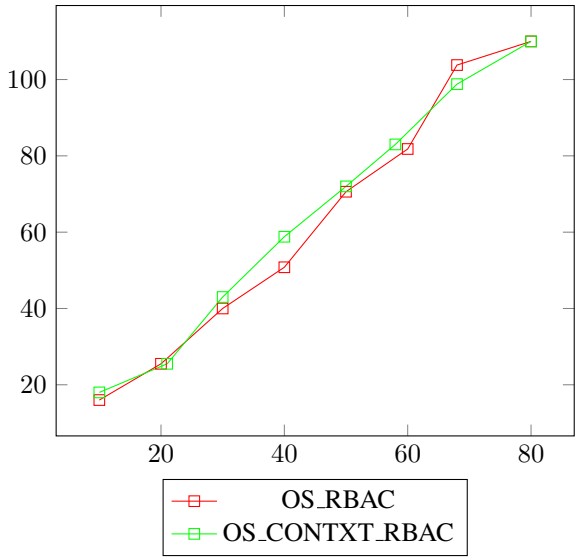


Fig. 7. Time Comparison for Authorization Requests

secure and also provides flexibility in writing complex access control policies. We also discuss some use cases, where we show the benefits of our policy system extension. Our work is an initial attempt towards applying a combination of context and role-centric access control models for OpenStack. In the future, we will like to extend our model so as to resolve access control policy conflicts in collaborative cloud systems. Attribute and Role hierarchy aware systems are one of the possible extensions to our model.

## Acknowledgement

We will like to thank the members of the Ebiquity research lab, specifically Dr. Tim Finin and Dr. Karuna Joshi for their insights.

## References

[1] OpenStack OSLO Policy Engine. `https://wiki.openstack.org/wiki/Oslo`. [Online].

[2] The Rein Policy Framework for the Semantic Web. `http://dig.csail.mit.edu/2006/06/rein/`. [Online].

[3] Mansoor Alicherry, Angelos D Keromytis, and Angelos Stavrou. Deny-by-default distributed security policy enforcement in mobile ad hoc networks. In *International Conference on Security and Privacy in Communication Systems*, pages 41–50. Springer, 2009.

[4] David W Chadwick, Kristy Siu, Craig Lee, Yann Fouillat, and Damien Germonville. Adding federated identity management to openstack. *Journal of Grid Computing*, 12(1):3–27, 2014.

[5] Prajit Kumar Das, Anupam Joshi, and Tim Finin. Personalizing context-aware access control on mobile platforms. In *Collaboration and Internet Computing (CIC), 2017 IEEE 3rd International Conference on*, pages 107–116. IEEE, 2017.

[6] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[7] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, 800(162), 2013.

[8] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *Computer*, 48(2):85–88, 2015.

[9] Huangqin Jiang and Hongqi Zhang. Access control model for composite web services. In *Communication Technology (ICCT), 2012 IEEE 14th International Conference on*, pages 684–688. IEEE, 2012.

[10] Xin Jin, Ram Krishnan, and Ravi Sandhu. A unified attribute-based access control model covering dac, mac and rbac. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 41–55. Springer, 2012.

[11] Xin Jin, Ram Krishnan, and Ravi Sandhu. Role and attribute based collaborative administration of intra-tenant cloud iaas. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2014 International Conference on*, pages 261–274. IEEE, 2014.

[12] Xin Jin, Ravi Sandhu, and Ram Krishnan. Rabac: role-centric attribute-based access control. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 84–96. Springer, 2012.

[13] Maithilee Joshi, Sudip Mittal, Karuna P Joshi, and Tim Finin. Semantically rich, oblivious access control using abac for secure cloud storage. In *Edge Computing (EDGE), 2017 IEEE International Conference on*, pages 142–149. IEEE, 2017.

[14] Abdul Raouf Khan. Access control in cloud computing environment. *ARPN Journal of Engineering and Applied Sciences*, 7(5):613–615, 2012.

[15] Craig A Lee and Nehal Desai. Approaches for virtual organization support in openstack. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 432–438. IEEE, 2014.

[16] Huiying Li, Xiang Zhang, Honghan Wu, and Yuzhong Qu. Design and application of rule based access control policies. In *Proc of the Semantic Web and Policy Workshop*, pages 34–41. Citeseer, 2005.

[17] Steve Martinelli, Henry Nash, and Brad Topol. *Identity, Authentication, and Access Management in OpenStack: Implementing and Deploying Keystone*. " O'Reilly Media, Inc.", 2015.

[18] Navid Pustchi and Ravi Sandhu. Mt-abac: A multi-tenant attribute-based access control model with tenant trust. In *International Conference on Network and System Security*, pages 206–220. Springer, 2015.

[19] Ravi Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *Proceedings of the third ACM workshop on Role-based access control*, pages 47–54. ACM, 1998.

[20] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[21] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[22] Bo Tang and Ravi Sandhu. Extending openstack access control with domain trust. In *International Conference on Network and System Security*, pages 54–69. Springer, 2014.

[23] Eric Yuan and Jin Tong. Attributed based access control (abac) for web services. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 2005.