APPROVAL SHEET

Title of Thesis: Semantically Rich, Context-Aware, Attribute based Access Control Model for Cloud Systems

Name of Candidate: Vishal Rathod
MS Computer Science
August 2018

Thesis and Abstract Approved: ______________________
Dr. Anupam Joshi
Oros Family Professor and Chair
Department of Computer Science and
Electrical Engineering

Date Approved: ____________________________

ABSTRACT

Title of Thesis: Semantically Rich, Context-Aware, Attribute based Access Control Model for Cloud Systems

Vishal Rathod, MS Computer Science, 2018

Thesis directed by: Dr. Anupam Joshi,
Oros Family Professor and Chair
Department of Computer Science and Electrical Engineering

Resource access control is an important research topic in cloud systems security. Much of the work has been focused on context-sensitive access control and rule representation. In an open source cloud computing platform such as OpenStack, operators use Role-Based Access Control (RBAC) model to grant user-access to cloud resources. However, these user level role-based access control technique fails to include a comprehensive user context. A situational aware framework will provide hardened access security by bringing in users context in such cloud systems.

In this work, we propose a semantically rich context-sensitive access control system for OpenStack by incorporating the user's current context attributes like location, time, etc. We integrate our own knowledge graph dependent attribute-based policy system with OpenStack policy engine to demonstrate our approach. 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, while keeping OpenStack's current RBAC architecture in place. The proposed system provides enhanced, flexible access control while minimizing the overhead of altering the existing access control framework. We present use cases to highlight the benefits of our system and show enforcement results. The study also investigates the flexibility of integrating different policy frameworks in OpenStack in order to enhance the access control.

Semantically Rich, Context-Aware, Attribute Based Access Control Model for Cloud Systems

by

Vishal Rathod

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2018

I dedicate this work to my mom, dad, and sister.

ACKNOWLEDGMENTS

I would like to first express my deepest gratitude to my thesis advisor, Dr. Anupam Joshi for supporting me through my masters study and research. I am gratefully indebted to his invaluable guidance, understanding, patience and motivation for this thesis. I am also grateful to Dr. Tim Finin, Dr. Karuna Joshi and entire ebiquity team, for providing me valuable feedback throughout my thesis and research work. I would like to thank my senior friend, Piyush Nimbalkar, who encouraged and advised me through it. Finally, I thank all my friends here at UMBC for being such amazing people and making this journey a memorable one. This accomplishment would not have been possible without them.

Thank You!

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Cloud computing is an emerging technology, it represents a paradigm shift in terms of system deployment. Such computing environment allows developers, to create accessible Internet-based ubiquitous services. Securing this environment 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.

In general, access control policies are quite complex. 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. As access control policies are expressed in natural language, the current researches focus on expressing these policies in the machine-understandable format (Anderson *et al.* 2003, Kagal, Finin, & Joshi 2003, Tonti *et al.* 2003).

In order to specify organizational policies efficiently, we also need a clearly defined access control model. These access control models provide a systematic way to represent an organization's behavioral posture in realistic scenarios.

A challenge for both policies and the descriptive models is to be able to represent fine-grained context. Industry wide general access control models involve development of traditional access control frameworks. Such frameworks mainly categories into Mandatory Access Control (MAC) (Jin, Krishnan, & Sandhu 2012), Discretionary Access Control (DAC) (Sandhu & Munawer 1998), Role-Based Access Control (RBAC) (Ferraiolo *et al.* 2001) and Rule-Based Access Control (Li *et al.* 2005). Among these Role Based Access Control (RBAC) model is the most widely used access control model in the industry (Sandhu *et al.* 1996).

Many applications and platforms use a custom form of RBAC as per their needs and requirements. Major cloud computing platforms such as OpenStack (Sefraoui, Aissaoui, & Eleuldj 2012), AWS (aws), and Microsoft Azure (Azu) utilize RBAC as their authorization system. The RBAC system has many advantages and limitations. Some well known limitations are role explosion and role-permission explosion (Punithasurya & Jeba Priya 2012). In the RBAC model, the access control policies can be defined on the basis of roles, which curbs the creation of complex access control rules and incorporation of fine-grained context (Ferraiolo *et al.* 2001, Fuchs, Pernul, & Sandhu 2011).

Access Control framework, 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 (Hu, Kuhn, & Ferraiolo 2015). The attribute information can be based on user, subject, and resource properties (Hu *et al.* 2013). This inherent flexibility permits creation of complex real-world applications and systems.

Nonetheless, it is difficult for existing systems to adapt to attribute-based access control policies, as they require a definite and a robust attribute and access control management system (Jin, Krishnan, & Sandhu 2012). We believe that the appendage of RBAC with attribute information has to be a gradual, but definite next step, so as to develop the future industry standard.

So as to achieve this, we can add *contextual attributes* to role centric system. An role-centric system can be augmented with user, environmental, application oriented, or a combination of these contextual attributes (Das, Joshi, & Finin 2017). The system-state machine can be further differentiated based on these contextual attributes. Typically, the user context comprises of a users 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 for such a scenario can be, permitting communication only on an AES 256 encrypted connection.

Consider Access Control on a smart-phone, which provides permission to an installed messaging application. The user may want context augmented policies that state: ‘the messaging application can be used between 8 AM to 9 PM except when in a meeting’. The access control model in this case needs to extend the traditional Attribute Based Access Control (ABAC) models with user and environmental context.

In this work, we propose a *role-centric attribute based access control model* for cloud systems by adding contextual attributes to core cloud access control systems. We create policy representations where context dependent information is included in the access control models.

We developed a simulation of OpenStack Access Control Model using a knowledge graph and added the provisioning of configurable contextual attributes to the same. Additionally, we built an attribute-based policy system which uses above knowledge graph to evaluate access decision for an OpenStack API request. This newly built policy system is integrated with the existing OpenStack policy engine, which provides a unifying framework to define, administer and enforce commonly known access control policies as well as new context-aware access control policies. It enables expression and enforcement of complex access control policies which are difficult to demonstrate in languages like XML. In the end, we bring advantage to develop flexible, more secure and controlled access control for the cloud system. Also, this allows us to extend existing RBAC in OpenStack with contextual attributes in order to combine the advantages of both models.

As a proof of concept, we extend the existing RBAC in *OpenStack* (Tang & Sandhu 2014) with contextual attributes. Our attribute-based policy system, is a general purpose semantically rich, context-aware, attribute-based access control framework. 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 2 discusses our related work.We present our system's architecture and explain different components in Section 4. We evaluate our system in Section 6. We discuss our future work and conclusion in Section 7.

1.0.1 Contributions

We created a context-aware access control model which integrates attributes to Role-Based Access Control (RBAC) policies. These policies permit access decisions based on contextual attributes information of users and resources being accessed. Access decisions are evaluated using a context augmented knowledge graph. We integrated our policy system with the OpenStack's access control engine which provides a unifying framework to define, administer, and enforce commonly known access control policies as well as new context-aware access control policies. Additionally, we compared both the systems i.e first with RBAC policy in OpenStack without any modifications, and second, our context enhanced role-centric policy. This allows us to compare and combine the advantages provided by both models.

Chapter 2

RELATED WORK & BACKGROUND

2.0.1 Access Control

Attribute based access control in cloud systems and in OpenStack has been studied with different traditional access control models like DAC (Sandhu & Munawer 1998), MAC (Jin, Krishnan, & Sandhu 2012), RBAC (Ferraiolo *et al.* 2001). These models are identity-based, and the identification is usually done through roles assignment (Yuan & Tong 2005). These models however, are suitable for static systems, with limited set of users and services (Khan 2012). These systems are mostly centralized (Jiang & Zhang 2012). Policies in attribute based access control models are semantically more expressive than the RBAC model (Hu, Kuhn, & Ferraiolo 2015). Jin et. al. (Jin, Krishnan, & Sandhu 2012) created a combined attribute-based access control model that can be configured to with DAC, MAC, and RBAC (Jin, Krishnan, & Sandhu 2014). They also created a proof of concept using an OpenStack implementation, where they replaced the native RBAC (Tang & Sandhu 2014).

Our approach is designed to incorporate contextual attributes in OpenStack's existing RBAC framework. Approaches to include attributes in OpenStack for cloud federation and identity management have been discussed by Chadwick et al. (Chadwick *et al.* 2014) and by Lee et al. (Lee & Desai 2014).

Pustchi et al. (Pustchi & Sandhu 2015) have discussed the 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. (Jin, Sandhu, & Krishnan 2012), along with XACML profiles. XACML is a general-purpose access control policy language for managing access to resources. Joshi et al. (Joshi *et al.* 2017), have proposed a semantically rich access control system that evaluates access decisions based on rules generated using the organizations confidentiality policies. Their proposed system analyzes the multi-valued attributes of the user and the requested document, before making an access decision.

2.0.2 OpenStack

A private cloud can be installed in an organization only with the help of a private cloud middleware. A cloud middleware is a tool that helps setup the cloud environment, which acts in the middle of cloud resources and the client. It then provisions resource according to the client's needs. There are two types of private cloud middlewares available - proprietary and open-source.

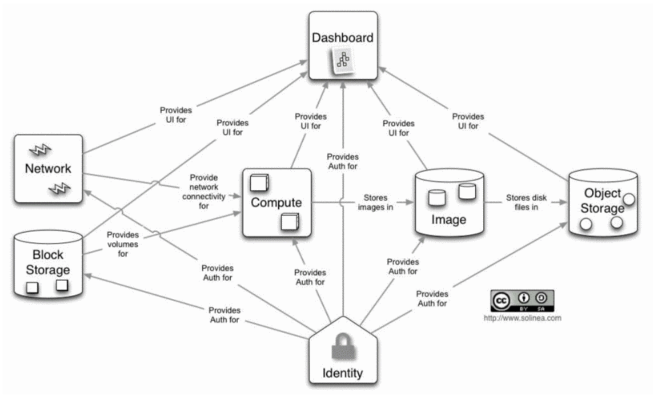


FIG. 2.1. OpenStack Architecture

Enterprise private cloud providers like, Amazon (aws), Microsoft (Azu), VMWare (vmw), IBM (IBM), etc. offer solutions to build an organizational private cloud. However, their offerings remain proprietary, this restricts complex user specific modifications. On the other hand, open-source private cloud middlewares, like, OpenStack (Sefraoui, Aissaoui, & Eleuldj 2012) allow modifications.

OpenStack is an open source, distributed cloud system. It keeps its services decoupled, and is designed to provide scalability. It is a collection of software pieces that aim to provide cloud services by virtualizing network, storage, and compute resources. Figure 3.1 shows the OpenStack architecture. There are three modules - Compute node for virtualization, Network node for networking, and Controller node for Controlling the environment. OpenStack community has also developed DevStack (dev) - for easy installation of OpenStack on a single node. In this work, we have used DevStack for single node installations of OpenStack.

Access control for OpenStack is maintained through the identity service, Keystone. It has several entities, such as domains and projects in addition to user roles. Keystone is able to formulate moderately complex authorization models. These authorization models are built to control access to all cloud resources which include computing, storage, and network.

Tang et al. (Tang & Sandhu 2014) created a core OpenStack Access Control model based on the OpenStack Identity API v3. For simplicity, this model was developed by removing Groups and Domains. In this model, Keystone, like most OpenStack projects, defines policy rules based on a RBAC approach. The policy file stores various rules for which some roles have access to certain activities. Each API call requires a corresponding permission structure to be present in the policy file. This dictates what level of access is allowed. The syntax for the same is:

API_NAME: RULE_STATEMENT

eg.

“identity:create_server”: “role:admin_or_owner”

This translates to: you must have admin or owner role for creating a new server through the compute service.

Open Stack Core Components and Terminology OpenStack access control model comprises of users, projects, roles, services, object types, operations, and tokens. Groups and domains are generally not included in a simplified model since, groups are defined to be a small collection of users. All users in OpenStack belong to a single domain or tenant. Only the domain owner or administrator can see, and manage users. Users are individuals authenticated to access cloud resources. Projects are resource containers, through which users get access to specific cloud resources such as virtual machines (VMs), storage, etc. Roles are global entities used to associate users with projects inside a domain. Permissions are assigned to role-project pairs and are utilized to designate access levels. A cloud administrator defines the Role-permission assignments. Object types are different resources in cloud services such as virtual machines (VMs), images, etc. Operations are access methods on these object types owned by cloud services.

Each authenticated user receives a token from the identity service, Keystone. This token defines the scope of resources that the user is allowed to access. A token is equivalent to a subject and has information about the user, its roles, and its associated projects. To generate the token it is necessary to send a payload containing credentials in a request. If the request succeeds, the server returns an authentication token that identifies a specific user, in a particular project. Then, on executing API requests and including the valid token in the X-Auth-Token header, you can gain access to a service. If the Unauthorized (401) error occurs, we need to request another token. This is the process followed by the RBAC model. To overcome the limitations of the RBAC model, we extend the simplified current access control model by adding user attributes. In Section 4 we discuss the extended model in detail.

Chapter 3

ATTRIBUTE BASED EXTENSION FOR OPENSTACK

In this section, we describe our contextual attribute based role-centric access control model for OpenStack by extending it's current simplified access control model.

3.0.1 Access control knowledge representation

Coupling Access control models and policy specification language like Web Ontology Language (OWL) gives us the means to formulate complex policies. This helps in creating access control models that are well understood by the security community. We leverage the power of reasoning to make access control decisions.

In our work we create a knowledge graph and add rules that capture the Attribute Based Access Control model, thus enabling fine-grained, context-dependent rules. Real-world entities and their relationships are captured using OWL.

Our system is a general-purpose attribute based access control framework which can express and query the arbitrary, organization specific, attribute-based access control policy rules. Our own attribute-base policy framework allows defining access control policies in terms of a standardized and a generic set of relations and functions that can be reused. The foremost objective of the system is to provide a unifying framework to support a wide range of attribute-based policies or complex rules combinations.

The system has five core entities: users, operations, user attributes, operation attributes, access right classes. The system has a default deny policy architecture (Alicherry, Keromytis, & Stavrou 2009). This procedure is analogous to a firewall policy where everything is dismissed unless explicitly approved. Administrator will build access grant rules using default deny policy architecture where the destination will be a set of critical API operations of OpenStack and the inferred action results will be set to "Grant Access".

In our current implementation, we utilize only two types of relations assignment for specifying policies rules which are relations between users to their current attributes, and user attributes to operation attributes. The general architecture of our attribute-based policy system is shown in Figure 3.1. It contains the system server and the knowledge graph.

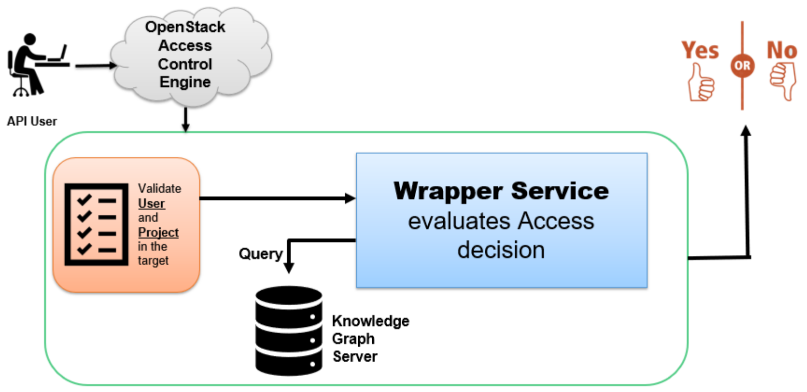

OpenStack Access Control Engine
API User
Yes
OR
No
Validate User and Project in the target
Wrapper Service evaluates Access decision
Query
Knowledge Graph Server

FIG. 3.1. Attribute-Base Policy System Architecture

The system supports a rich set of capabilities for establishing customized access control policies. It allows to specify access allowed relations and apply constraints. Attribute-based policy system allows user to define access control policies by creating policy classes, users, user attributes, setting operations sets and permissions between user attributes and operations. All this data, information and relations are stored in the form of a knowledge graph. Users request access to objects through our own policy system which in turn queries to knowledge graph ontology for access control decisions and enforce these decisions on host systems.

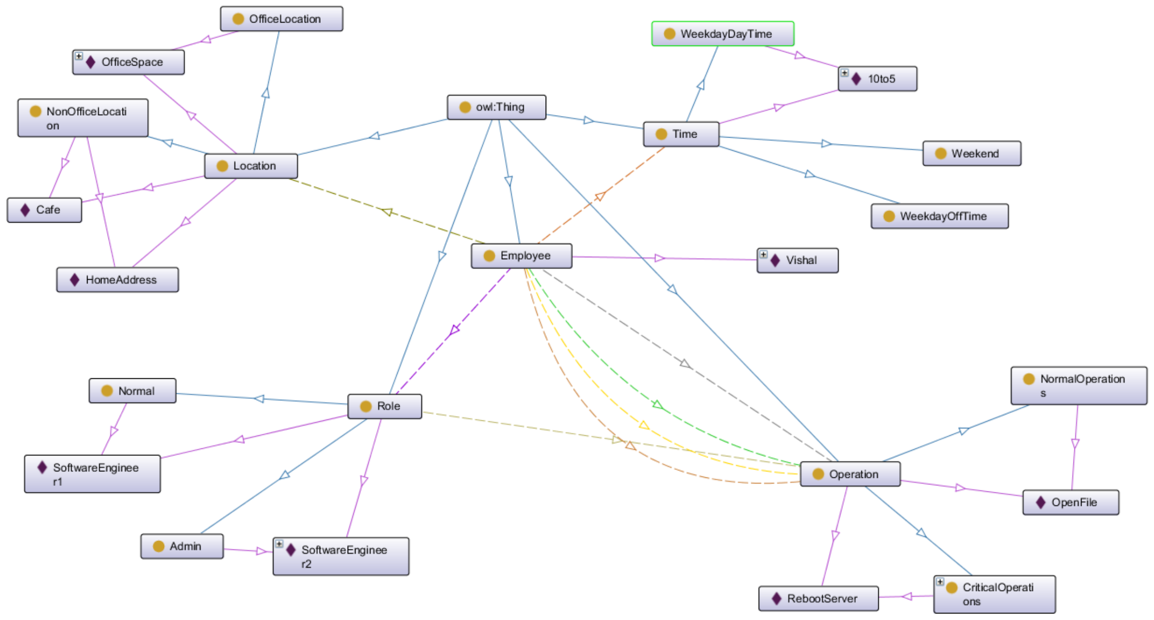


FIG. 3.2. Knowledge-Graph

In the attribute enhanced OpenStack access control model, we require that each user attribute is *atomic*. Every user is associated with a finite set of user attribute functions whose values are assigned and modified by the security administrators. For any user, all the associated user attributes are always active during their lifetime in any defined policy. As shown in knowledge graph Figure 3.2, we have created Employee, Location, Operation, Role and Time classes. Property assertion for user-attribute assignments includes different relations of a user with distinct instances of role, location and time.

```
Rule: 'User' -> {hasLocation == officeSpace ^ makeRequestAt ==
   WeekDayTime ^ hasRole == Admin ^ isExecuting == RebootServer
   ^ RebootServer == CriticalOperation } ->
   hasAccess(RebootServer)
```

That is, if a User $hasLocation$ ‘OfficeSpace’ which is an instance of Location class and $makeRequestAt$ ‘10to5_Weekday’ which is an instance Time class and $hasRole$ ‘Admin’ which is an instance of Role class and executing an operation which is critical operation then user has access to execute that operation.

Chapter 4

ENFORCEMENT AND IMPLEMENTATION

This section discusses our enforcement and implementation details of enhanced access control model by emphasizing contextual attributes in OpenStack employing the system service.

OpenStack is an emerging opensource cloud platform that presents an architecture to utilize or enhance its services as per the specifications. In this work, we are focusing on one of the services of OpenStack namely Nova, a project that provides a way to provision compute instances. Nova supports creating virtual machines, bare-metal servers, and has limited support for system containers. As shown in Figure 5.1, OpenStack Access control policy engine relies on access control and authorization framework which is *OSLO* policy engine (Martinelli, Nash, & Topol 2015, OSL) to estimate the access decisions of cloud resources.

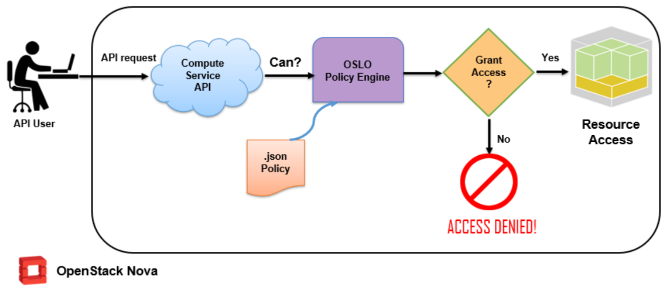


FIG. 4.1. OpenStack Policy Engine Architecture

An operator can access the servers and OS services by making API requests to the compute service. Each OpenStack service defines the access policies for its resources in an associated policy file. In the current implementation, the compute service is integrated with the OSLO policy engine (OSL) to control the access of a resource by either utilizing the code based implementation of access control policy or *policy.json* file. For example, a resource could be API access, the ability to attach to a volume, or to fire up instances. Since OpenStack is open-source service, it provides the capability to integrate our own attribute-based policy system in its authorization framework. To enforce user's contextual attributes, OpenStack is using an organization knowledge store. This organizational knowledge store has all the contextual information of any user like current location, time, host IP, etc. This store can be populated through the system's current location and time or retrieved from other OpenStack projects which store user's details. The enforcement framework uses OpenStack Ocata (Oca) release with Identity API version 2 and our own attribute-based policy system.

4.0.1 Enforcement Architecture

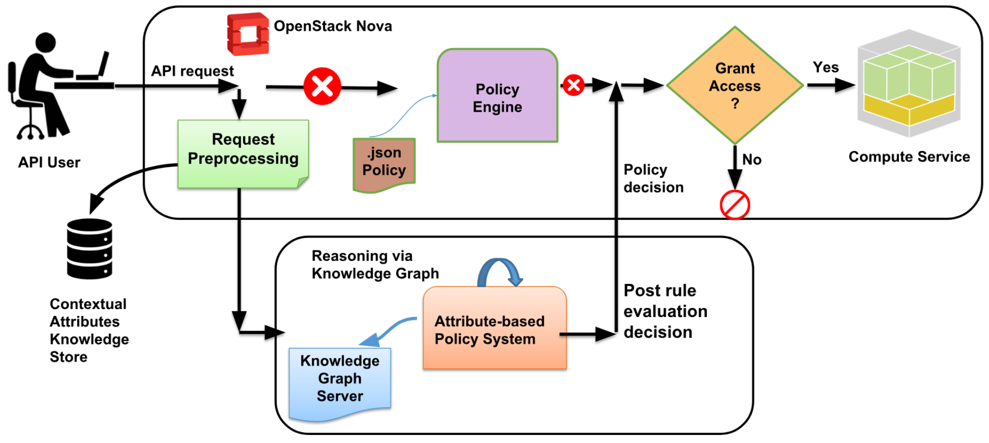


FIG. 4.2. Extended System Architecture

In this architecture, we utilize the attribute-based policy system as a centralized policy administration module that returns a set of user permissions on objects based on the rules definitions. The OpenStack compute service is connected to an active directory, an organizational knowledge store having all the users and their associated user attributes. We also assume that OpenStack uses system services or OpenStack projects as its user identity back-end to store all users related information, including attributes. As shown in Figure 3.2, we have organized a knowledge graph of user's permission which has all the relations of a user with their current contextual attributes and allowed compute service API operation based those relations. Due to the dynamic nature of cloud objects, we acknowledge OpenStack commands as objects in the policy defined in the knowledge graph. The commands are specific to each service in OpenStack such as Nova, Glance, Cinder, etc. The policy definitions, typically listed in OpenStack policy file have been defined in our own attribute-based policy system in an Operation class as shown in Figure 4.3.

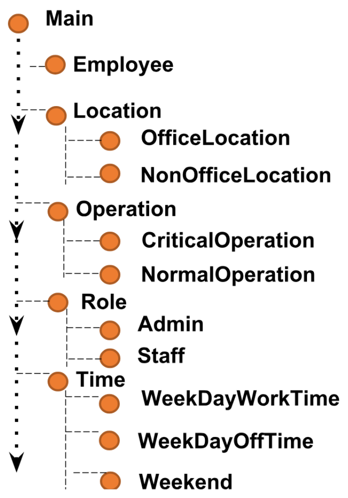


FIG. 4.3. Class hierarchy

One instance of our operation class, from “Objects/Attributes” view, is shown in Figure 4.3. For the current implementation, we have examined a sample policy specified for

Nova API commands in OpenStack. The user attributes and objects in knowledge graph are associated with a specific operation set, for example, the operation set with “allowed access” permissions defined between user attributeadmin, and objectcompute extension-rebootServer. The user attributes and objects in knowledge graph are associated with a specific operation set, for example, the operation set with “allowed access” permissions defined between user attributeadmin, and objectcompute extension-rebootServer. That means the attribute value admin has allowed access permissions on computer extension-reboot server command so, the user with any of these roles is authorized to do this operation in OpenStack. OpenStack services communicate through a RESTful API to our system which then interfaces with the knowledge graph's inferred results server for making authorization decisions.

4.0.2 Attribute-Based Policy System

our system is an interface for conversation between inferred policy rules and different OpenStack services. It is written in Python using a RESTful API and acts as a RESTful server for OpenStack services. The OpenStack service requires the user to query the knowledge store and fetch the user's current attributes to perform any operation. Then the request is forwarded to the developed policy system which initially verifies the project in the target and operation to be performed. Next, it makes the access control decisions by calling different program functions utilizing contextual attribute extension for OpenStack.

The system replaces the existing policy engine in OpenStack and is responsible for evaluating the access rules defined in knowledge graph and delivers the access decisions to OpenStack services. OpenStack Services are the policy enforcement modules that drive the access control verdicts returned by the attribute-based policy system and responds to the users with appropriate results. A sequence diagram exhibiting authorization in OpenStack utilizing knowledge store and our own policy system is shown in Figure 5.2. It explains the course of actions involved in the authorization process. Currently, the system operates with two types of policies, an RBAC policy (same as OpenStack's current access control policy), and a role-centric attribute-based policy with user's contextual attributes taken into account and it can be easily extended to implement other sorts of access control policies as expected.

This implementation is a proof-of-concept. It centers to demonstrate the applicability and feasibility of our proposed model in the OpenStack cloud platform. However, it has not optimized for performance. The attribute-based policy system can be devised to be a universally independent component which can be used with several policy-configuration tools that contribute additional benefits like dynamically resolving policy rules conflict, and be applicable to cloud platforms besides OpenStack.

4.0.3 Alternate Approaches

A reasoner is a key component for working with OWL ontologies. In fact, all querying of an OWL ontology should be done using a reasoner. This is because knowledge in an ontology might not be explicit and a reasoner is required to deduce implicit knowledge so that the correct query results are obtained. In order to access a reasoner via the API, a reasoner implementation is needed. The OWL API includes the OWLReasoner interface for accessing OWL reasoners. JFact, HermiT, Pellet are few general purpose reasoners which provide implementations of the OWL API OWLReasoner interface.

OpenStack access control system has the capability to integrate with external access control engine. There are three possible ways to implement these modifications. We can integrate general-purpose reasoner, SPARQL query engine or special purpose programmable custom reasoner with the current implementation of OpenStack access control engine. For our proof of concept, we have experimented with these set of reasoner and analyzed their results and performances.

Initially, we experimented with a Pellet reasoner. Pellet is an open-source Java based OWL 2 reasoner. We have loaded the knowledge graph of our ontology to the reasoner and created property and resources for query the reasoner. It's possible to directly load a file that contains SWRL rules into Pellet and rules will be parsed and processed. There are two options either to use the OWLAPI or Jena interface; they both handle SWRL. We have used OWLAPI to perform the reasoning over our ontology and extracted the inferred results from this reasoner into a separate result file. Later, we exposed those results through a Python service which is integrated with the OpenStack Policy engine. The mentioned service results allowed operations that are permitted to a user based on its current contextual attributes provided as an input.

We also experimented with Apache Jena, it's an open-source framework for building semantic web applications. Fuseki is a component of Jena that lets you query an RDF data model using the SPARQL query language. Fuseki is a SPARQL server which provides REST-style SPARQL HTTP Update, SPARQL Query, and SPARQL Update using the SPARQL protocol over HTTP. We installed the latest version of Fuseki server into our local machine and instead of extracting the results in a file we stored the results in Fuseki server. After this, we execute SPARQL queries using a wrapper around a SPARQL service, called as SPARQLWrapper. This wrapper service helps in creating the query URI and, possibly, convert the result into a more manageable format. Based on the results obtained from the SPARQL query, we forwarded the access decision back to OpenStack access control engine.

In order to test with special purpose reasoner, we created our own simplified custom procedural reasoning engine. This was a programmatic approach which involved, creation of a small set of classes and their relations. We have created functional rules using nested conditional checks. Then we implemented a service which uses a couple of functions to do the reasoning and verifying the results.

This overall attempt was just a preliminary attempt to verify the system results and flexibility along with analyzing the performance of with different approaches. We noticed that there is no major significant performance difference while experimenting with general purpose and special purpose reasoner. Although we tried with special small amounts of instances and rules with special reasoner, the performance evaluation matrix shows a minor difference in overall request-response time compared to other approaches. With our own programmatic special purpose reasoner we obtained expected results and slight performance improvement but for a variety of large use case and dataset, you can use the general purpose reasoner as these are production level reasoner.

As the policy system is flexible to integrate with any reasoner, it's just required to modify the endpoint in policy system. While experimenting with the general purpose reasoner we used RDF formate instead of JSON because it helped to clearly define the contextual attributes of users and resources. But JSON could be a great alternative if you consider the scalability aspect of the system.

Chapter 5

USE CASES

We explain two use cases - first, with only roles and second with roles and user attributes. These use cases illustrate, how a simplified existing access control model and an attribute-based enhanced access control model can be enforced in OpenStack.

5.0.1 A Simplified RBAC Policy Model with Query Engine

Initially, we use a sample RBAC policy, equivalent to OpenStack access control policy with two roles Admin and Staff and four Nova commands: compute extension-create, compute extension-delete, compute extension-reboot, and compute extension-show. Preceding samples command preferred for the use case. Additionally, we can also specify authorization policies for other commands in various services of OpenStack. The permissions are determined based on the user's role assigned for a specific project. The Nova API commands are used to generate ssh keys for a user which are utilized to authenticate the API call. The authorization rules for each command, for a generic user u, are given below.

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

Authorization rules for any user u:

- Compute extension-list:

$$Role(u) = Admin$$

- 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.

5.0.2 An attribute based Role-Centric Access Control

Now we discuss an attribute-based role-centric access control model. The attributes are determined based on the contextual information of a user along with attributes of an operation to be performed. This is an example of our contextual attribute-based access control policy of OpenStack. Besides roles and commands, there are user attributes Location, Time, Role and Operation - severity attribute that can be assigned only one value from its range. For simplicity we have considered Location within range:

$$\{OfficeLocation, NonOfficeLocation\}$$

Similarly for Time range:

$$\{WeekdayWorkTime, WeekdayOffTime, Weekend\}$$

Also, the Operation-severity attribute can have one of the values of:

$$\{CriticalOperation, NormalOperation\}$$

User 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. In an organization, there are multiple roles and contextual attributes, and permissions are assigned based on these parameters.

Give below is the authorization policy based on roles and contextual attributes. As shown in figure 5.1, we have constructed a sample username: '$Vishal$' and assigned few property values to it signifying its current contextual attributes.

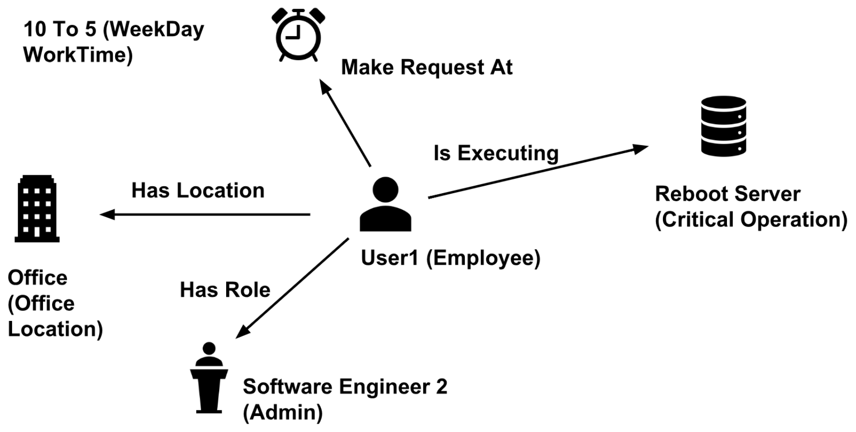

10 To 5 (WeekDay WorkTime)
Make Request At
Is Executing
Reboot Server (Critical Operation)
Has Location
User1 (Employee)
Office (Office Location)
Has Role
Software Engineer 2 (Admin)

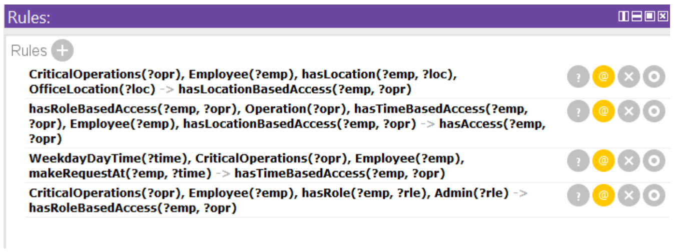
FIG. 5.1. RDF for an instance of Class Employee

That is User '$Vishal$' $hasLocation$ '$OfficeSpace$' which is an instance of Location class and $makeRequestAt$ '$10to5\_Weekday$' which is an instance Time class and $hasRole$ '$SoftwareEngineer2$' which is an instance of Admin class which is a subclass of Class Role. Additionally, an access grant is requested for the Role-Operation relations, which covers relations between role instance and operation:

$$Admin \rightarrow \{isExecuting \rightarrow RebootServer\}$$

That is User with having role '$Admin$' is executing Operation '$RebootServer$' which is an instance of CriticalOperations class which is the subclass of Class Operations.

And we have created a set of policy rules which make access grant decisions based on user attributes and prominence of an operation to be executed. These set of policy rules looks like:



```
CriticalOperations(?opr), Employee(?emp), hasLocation(?emp, ?loc),
OfficeLocation(?loc) -> hasLocationBasedAccess(?emp, ?opr)
hasRoleBasedAccess(?emp, ?opr), Operation(?opr), hasTimeBasedAccess(?emp,
?opr), Employee(?emp), hasLocationBasedAccess(?emp, ?opr) -> hasAccess(?emp,
?opr)
WeekdayDayTime(?time), CriticalOperations(?opr), Employee(?emp),
makeRequestAt(?emp, ?time) -> hasTimeBasedAccess(?emp, ?opr)
CriticalOperations(?opr), Employee(?emp), hasRole(?emp, ?rle), Admin(?rle) ->
hasRoleBasedAccess(?emp, ?opr)
```

FIG. 5.2. Policy Rules Comprising Contextual attributes

```
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)
```

As per the first rule, an employee 'emp' is authorized to all the instance of Critical-Operation class only if the employee has the location which is the instance of an OfficeLocation class. This access right is an intermediate right which we have defined as Location-based access. Similarly, we have created Role-based and Time-Based access which is evaluated based on an employee's assigned role and current local time. These intermediate rules are evaluated for making an overall decision in Rule 4, i.e., an employee is allowed to execute the operation if he has the location, time, and role-based access allotted. All of these requirements need to be accurate otherwise, an employee without the specified role and contextual attribute values will be denied. Since this is a role-centric policy, the roles are checked first and then, the user attribute values. If a role check fails then the user-attribute value is not checked and access is denied.

Inferred results of different rules and configuring valid attribute information generates the access decision which looks like:

That is employee 'Vishal' has granted an access to listed critical operations. A system screenshot of results in OpenStack is presented in Figure 5.3.

Figure 5.4 shows access grant decisions observed on providing authorized and unauthorized contextual policy information while making API requests. The figure, signifies requests and response of current authorization API for servers, images, and flavors of Open-

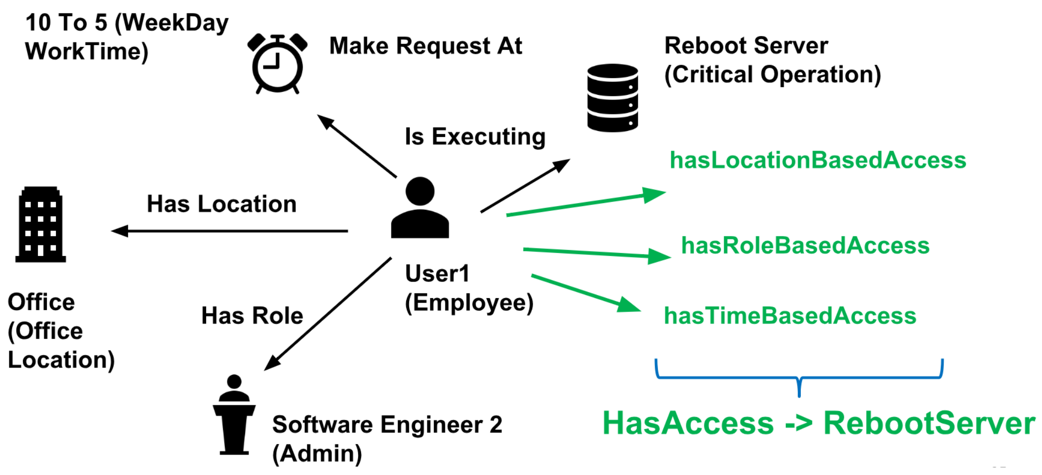
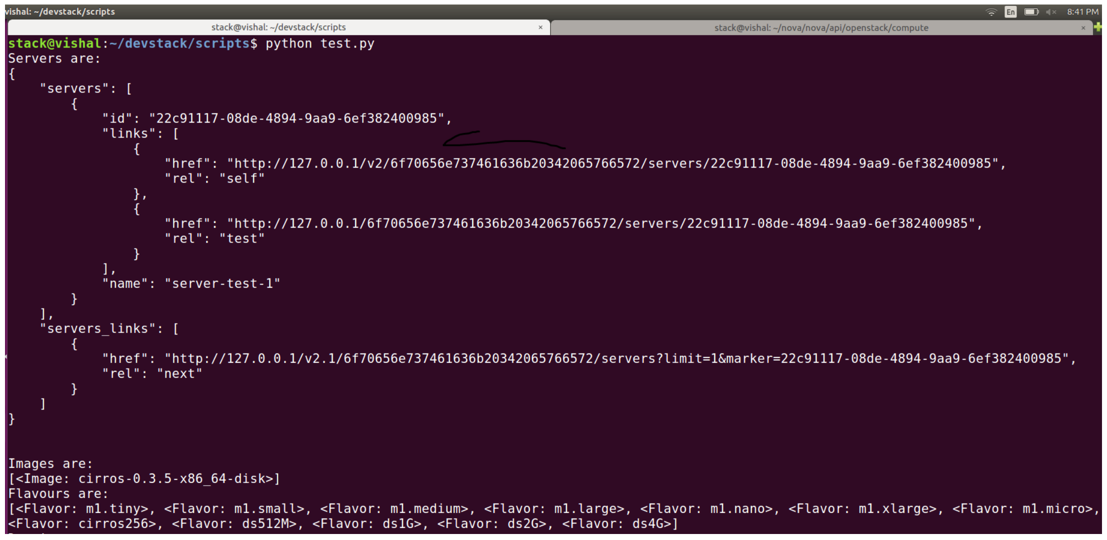


FIG. 5.3. Inferred results on reasoning policy rules

Stack server.

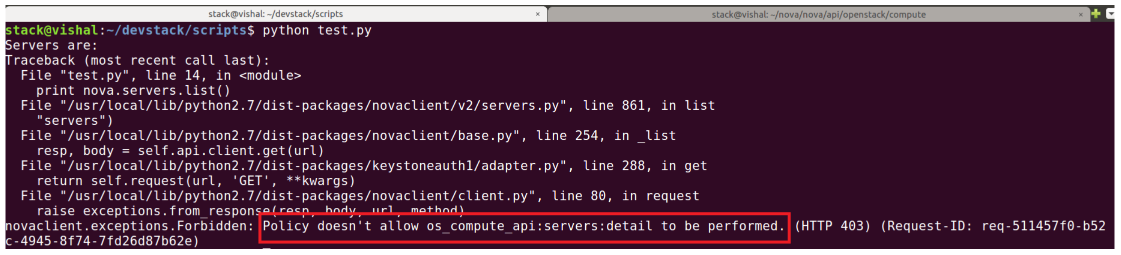
In Figure 5.5, the highlighted text depicts the unauthorized request exception. The values of contextual attributes determine the access grant.


vishal: ~/devstack/scripts
8:41 PM
stack@vishal: ~/devstack/scripts
stack@vishal: ~/nova/nova/api/openstack/compute

```
stack@vishal:~/devstack/scripts$ python test.py
Servers are:
{
    "servers": [
        {
            "id": "22c91117-08de-4894-9aa9-6ef382400985",
            "links": [
                {
                    "href": "http://127.0.0.1/v2/6f70656e737461636b20342065766572/servers/22c91117-08de-4894-9aa9-6ef382400985",
                    "rel": "self"
                },
                {
                    "href": "http://127.0.0.1/6f70656e737461636b20342065766572/servers/22c91117-08de-4894-9aa9-6ef382400985",
                    "rel": "test"
                }
            ],
            "name": "server-test-1"
        }
    ],
    "servers_links": [
        {
            "href": "http://127.0.0.1/v2.1/6f70656e737461636b20342065766572/servers?limit=1&marker=22c91117-08de-4894-9aa9-6ef382400985",
            "rel": "next"
        }
    ]
}


Images are:
[<Image: cirros-0.3.5-x86_64-disk>]
Flavours are:
[<Flavor: m1.tiny>, <Flavor: m1.small>, <Flavor: m1.medium>, <Flavor: m1.large>, <Flavor: m1.nano>, <Flavor: m1.xlarge>, <Flavor: m1.micro>,
<Flavor: cirros256>, <Flavor: ds512M>, <Flavor: ds1G>, <Flavor: ds2G>, <Flavor: ds4G>]
```

FIG. 5.4. Authorized Policy Requests


stack@vishal: ~/devstack/scripts
stack@vishal: ~/nova/nova/api/openstack/compute

```
stack@vishal:~/devstack/scripts$ python test.py
Servers are:
Traceback (most recent call last):
  File "test.py", line 14, in <module>
    print nova.servers.list()
  File "/usr/local/lib/python2.7/dist-packages/novaclient/v2/servers.py", line 861, in list
    "servers")
  File "/usr/local/lib/python2.7/dist-packages/novaclient/base.py", line 254, in _list
    resp, body = self.api.client.get(url)
  File "/usr/local/lib/python2.7/dist-packages/keystoneauth1/adapter.py", line 288, in get
    return self.request(url, 'GET', **kwargs)
  File "/usr/local/lib/python2.7/dist-packages/novaclient/client.py", line 80, in request
    raise exceptions.from_response(resp, body, url, method)
novaclient.exceptions.Forbidden: Policy doesn't allow os_compute_api:servers:detail to be performed. (HTTP 403) (Request-ID: req-511457f0-b52
c-4945-8f74-7fd26d87b62e)
```

FIG. 5.5. Unauthorized Policy Requests

Chapter 6

EVALUATIONS

This section describes the results and details of our performance evaluation. Further, we analyze the applicability of our modified access control system with possible performance enrichments. The experiments were performed on two policies discussed in the use case section.

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

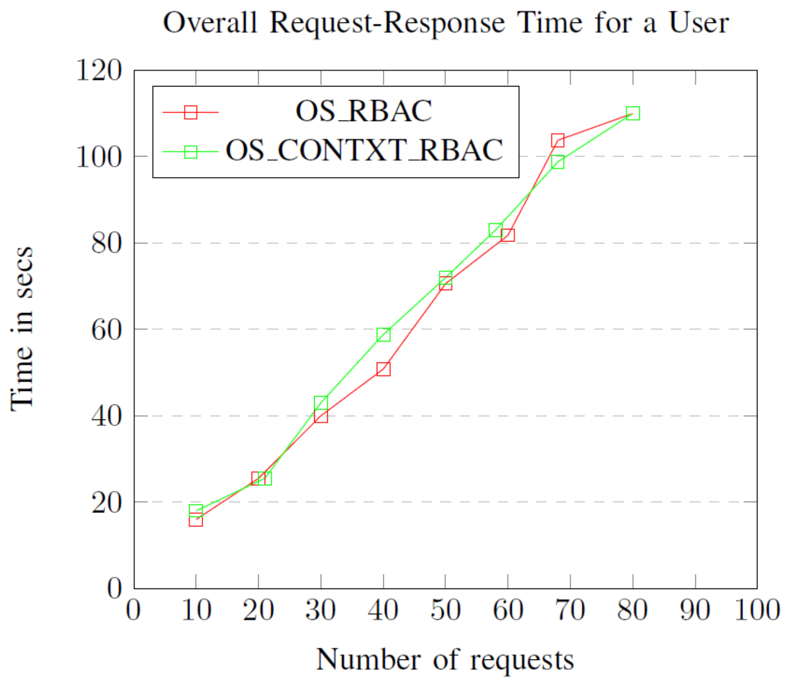


FIG. 6.1. Overall Time Taken in Requests-Response

in policy evaluation time compared to old platforms like ‘cwm engine’ of REIN policy framework (Rei).

Chapter 7

CONCLUSION AND FUTURE WORK

In this work, we implemented a contextual attribute-based access control system for OpenStack and imposed it utilizing the Knowledge Graph and Policy System. Adding contextual attribute while deciding access grant decision makes the access control system more secure and also adds flexibility for writing complex access control models. We presented use cases to depict the applicability and benefits of our Policy System extension with user and resource attributes. We also investigated the cost and feasibility of this model and its enforcement architecture. This is an initial attempt towards applying a combination of attribute-based access control model and role-centric access control model in widely used cloud systems.

We believe this work will facilitate the transition towards contextual attribute incorporation and will open prospective avenues to apply attribute-based access control in real-world applications using our Policy system. There are many interesting capabilities can be explored for Policy system as extensions to our model such as applying a combination of different access control policies defined in the current system, resolving access control policy conflicts in collaborative cloud systems or incorporating deny relations and constraints in the policies. This add-on might lead to a self contradictory policy and it will require a serious policy upgrade. Attribute and Role hierarchy are one of the possible extensions that can be a great addition in our model. Besides these, the proposed implementation is a proof-of-concept and hasn't been optimized for a real production environment. Additionally, we can maintain the reasoning result history which will help to improve the reasoning over complex policies or use those while integrating with other systems. But this might lead to a problem where storing these result will consume lot of memory which eventually needs a purging logic to deal with the scalability aspect.

The future work includes applying further performance enhancements to the enforcement framework and making it flexible for integration with all public and private cloud systems.

REFERENCES

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

[2] Anderson, A.; Nadalin, A.; Parducci, B.; Engovatov, D.; Lockhart, H.; Kudo, M.; Humenn, P.; Godik, S.; Anderson, S.; Crocker, S.; et al. 2003. extensible access control markup language (xacml) version 1.0. *OASIS*.

[3] Amazon Web Services (AWS) - Cloud Computing Services. `https://aws.amazon.com`. [Online].

[4] Microsoft Azure. `https://azure.microsoft.com`. [Online].

[5] Chadwick, D. W.; Siu, K.; Lee, C.; Fouillat, Y.; and Germonville, D. 2014. Adding federated identity management to openstack. *Journal of Grid Computing* 12(1):3–27.

[6] Das, P. K.; Joshi, A.; and Finin, T. 2017. Personalizing context-aware access control on mobile platforms. In *Collaboration and Internet Computing (CIC), 2017 IEEE 3rd International Conference on*, 107–116. IEEE.

[7] Devstack. `https://docs.openstack.org/devstack/latest/`. [Online].

[8] Ferraiolo, D. F.; Sandhu, R.; Gavrila, S.; Kuhn, D. R.; and Chandramouli, R. 2001. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)* 4(3):224–274.

[9] Fuchs, L.; Pernul, G.; and Sandhu, R. 2011. Roles in information security–a survey and classification of the research area. *computers & security* 30(8):748–769.

[10] Hu, V. C.; Ferraiolo, D.; Kuhn, R.; Friedman, A. R.; Lang, A. J.; Cogdell, M. M.; Schnitzer, A.; Sandlin, K.; Miller, R.; Scarfone, K.; et al. 2013. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication* 800(162).

[11] Hu, V. C.; Kuhn, D. R.; and Ferraiolo, D. F. 2015. Attribute-based access control. *Computer* 48(2):85–88.

[12] IBM Cloud. `https://www.ibm.com/cloud/`. [Online].

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

[14] Jin, X.; Krishnan, R.; and Sandhu, R. 2012. A unified attribute-based access control model covering dac, mac and rbac. In *IFIP Annual Conference on Data and Applications Security and Privacy*, 41–55. Springer.

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

[16] Jin, X.; Sandhu, R.; and Krishnan, R. 2012. Rabac: role-centric attribute-based access control. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, 84–96. Springer.

[17] Joshi, M.; Mittal, S.; Joshi, K. P.; and Finin, T. 2017. Semantically rich, oblivious access control using abac for secure cloud storage. In *Edge Computing (EDGE), 2017 IEEE International Conference on*, 142–149. IEEE.

[18] Kagal, L.; Finin, T.; and Joshi, A. 2003. A policy language for a pervasive computing environment. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, 63–74. IEEE.

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

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

[21] Li, H.; Zhang, X.; Wu, H.; and Qu, Y. 2005. Design and application of rule based access control policies. In *Proc of the Semantic Web and Policy Workshop*, 34–41. Citeseer.

[22] Martinelli, S.; Nash, H.; and Topol, B. 2015. *Identity, Authentication, and Access Management in OpenStack: Implementing and Deploying Keystone.* " O'Reilly Media, Inc.".

[23] OpenStack Relase Ocata. `https://www.openstack.org/software/ocata/`. [Online].

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

[25] Punithasurya, K., and Jeba Priya, S. 2012. Analysis of different access control mechanism in cloud. *International Journal of Applied Information Systems (IJAIS), Foundation of Computer Science FCS* 4(2).

[26] Pustchi, N., and Sandhu, R. 2015. Mt-abac: A multi-tenant attribute-based access

control model with tenant trust. In *International Conference on Network and System Security*, 206–220. Springer.

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

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

[29] Sandhu, R. S.; Coyne, E. J.; Feinstein, H. L.; and Youman, C. E. 1996. Role-based access control models. *Computer* 29(2):38–47.

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

[31] Tang, B., and Sandhu, R. 2014. Extending openstack access control with domain trust. In *International Conference on Network and System Security*, 54–69. Springer.

[32] Tonti, G.; Bradshaw, J. M.; Jeffers, R.; Montanari, R.; Suri, N.; and Uszok, A. 2003. Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In *International Semantic Web Conference*, 419–437. Springer.

[33] VMware Cloud. `https://cloud.vmware.com/`. [Online].

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