Extending OpenStack Access Control with Domain Trust

Bo Tang and Ravi Sandhu

Institute for Cyber Security and Department of Computer Science
University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249, US
xyp368@my.utsa.edu, ravi.sandhu@utsa.edu

Abstract. OpenStack has been rapidly established as the most popular open-source platform for cloud Infrastrusture-as-a-Service in this fast moving industry. In response to increasing access control requirements from its users, the OpenStack identity service Keystone has introduced several entities, such as domains and projects in addition to roles, resulting in a rather complex and somewhat obscure authorization model. In this paper, we present a formalized description of the core OpenStack access control (OSAC). We further propose a domain trust extension for OSAC to facilitate secure cross-domain authorization. We have implemented a proof-of-concept prototype of this trust extension based on Keystone. The authorization delay introduced by the domain trusts is 0.7 percent on average in our experiments.

Keywords: Distributed Access Control; Identity Management; Security in Cloud and Grid Systems; Trust Management.

1 Introduction

Cloud computing is widely anticipated as the next generation computing infrastructure although it is still in its infancy. The concept of cloud computing is attracting attention from both business and technology perspectives. Its pay-on-the-go business model tremendously minimizes on-premise investment on IT infrastructures for organizations and individuals. The multi-layered service-oriented architecture (SOA) design enables cloud service providers (CSPs) to serve their consumers with centralized software and data centers in a multi-tenant fashion. However, concerns of security with respect to data location and control, as well as availability create resistance to cloud adoption. One of the key issues in this regard is access control.

Multi-tenancy is one of the crucial characteristics of cloud services. We define a tenant from the perspective of a CSP, as an independent customer of the CSP responsible for paying for services used by that tenant.[1] A principal responsibility

[1] Payment is the norm in a public cloud while in a community cloud there often will be other methods for a tenant to obtain services. From the perspective of the tenant, a tenant could be a private individual, an organization big or small, a department within a larger organization, an ad hoc collaboration, and so on. This aspect of a tenant is typically not visible to the CSP in a public cloud.

of the CSP is to maintain isolation across tenants, so that the tenant's users can only access resources within that tenant's scope. Over time it has been recognized that controlled cross-tenant access is desirable and some models for that purpose have been proposed [10,20,21,22]. These models establish cross-tenant trust on a bilateral basis so as to enable appropriate cross-tenant access. Some notion of a trust relationship of this nature has been prevalent in prior work on distributed systems when crossing administrative boundaries. This is exemplified by the well-known mechanisms in Windows Active Directory (AD) [3] and the Grid [6,11].

Our central goal in this paper is to investigate the addition of cross-tenant access in the popular open-source OpenStack [4] platform for cloud infrastructure-as-a-service (IaaS). Specifically, with respect to the Havana release. The general concept of a tenant in a cloud maps to the concept of domain in the Havana release of OpenStack.[2] The identity service in OpenStack, called Keystone, is used to manage users as globally available resources. More specifically, the administrator of a domain can view all the user information and assign any user to roles controlled by that domain. Each user, as created, belongs to a single domain and the domain owner or administrator can only see and manage users within the domain. So far, the use cases of cross-domain access has not been carefully addressed in OpenStack. This lack is the main motivation for this paper.

In this paper, we propose a domain trust model addressing cross-domain access control in OpenStack and provide a proof-of-concept implementation by extending KeyStone. In response to increasing access control requirements from its users, Keystone has introduced several entities, such as domains, projects and groups in addition to roles, resulting in a rather complex and somewhat obscure authorization model. Before we can add cross-domain trust to this model it is necessary to cast the core OpenStack access control (OSAC) model in a formal way which is consistent with familiar terminology from the access control research literature. Development of this formal rigorous statement of OSAC is in itself an important contribution of this paper.

The rest of this paper is organized as follows. Section 2 gives the motivating use case of cross-domain authorization and some of the existing approaches. The formalized OpenStack Access Control (OSAC) model is presented in Section 3, followed by the extended domain trust model in Section 4. To demonstrate the feasibility of the novel domain trust model in OpenStack, we develop a proof-of-concept prototype based on Keystone. This is described in Section 5, along with evaluation results. Related work in the cross-domain trust arena is discussed in Section 6. Finally, we conclude the paper in Section 7.

2 Background and Motivation

In this section, we use a DevOps [1] example to explain why we need cross-domain accesses in the cloud and what potential problems we have in the latest

[2] Previous releases of OpenStack employed the term tenant for what has now come to be called project in OpenStack. The term tenant is no longer used in OpenStack. In this paper we use the term tenant as a generic concept in cloud computing, while domain is specific to OpenStack as its realization of a tenant.

OpenStack solution. Also, we discuss the pros and cons of existing cross-domain authorization solutions. The scope and assumptions of our work are given.

Motivation DevOps is a newly emerged software development methodology that stresses collaboration among software development, quality assurance (QA) and operations. Numerous companies are actively practicing DevOps since it aims to help organizations rapidly produce software products and services [1]. When DevOps for an organization comes into play in an OpenStack cloud, cross-domain accesses become inevitable and requires suitable control. Figure 1 shows the authorization related components in OpenStack giving cross-domain accesses for a DevOps use case. The token information is managed by the centralized identity service. The policy rules are administered and checked against access requests along with user tokens in each distributed cloud service.

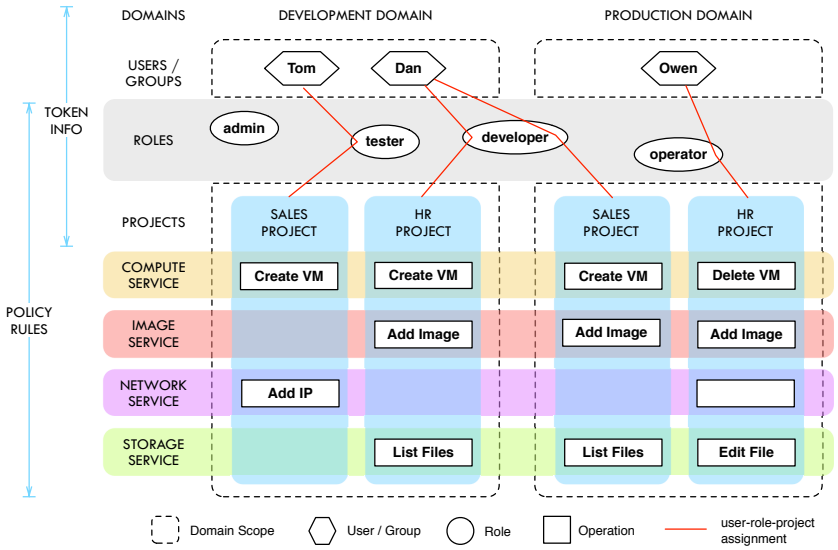


Fig. 1: An DevOps use case of cross-domain accesses.

Suppose the organization has two domains in the cloud: *Production* and *Development*. *Production* hosts live applications supporting the organization's daily business requiring strict controls on changes. Meanwhile, *Development* consists of development and testing environments, basically a sandbox, where developers and testers can freely conduct experiments with the cloud resources. The isolation of the two domains is mandatory for best practice and compliance reasons. Each domain contains its own set of users, groups, projects and controlled access to the full-spectrum of cloud services, such as compute, image and network. As shown in Figure 1, Owen is an operator in *Production*. Dan and Tom are a developer and a tester respectively in *Development*. Each domain has two independent projects: *Sales* and *HR*. The users are assigned necessary permissions to access projects in their owning domains to accomplish their daily jobs (each

user in OpenStack has a single domain which "owns" the user). In order to process a DevOps case in which a live application in *Sales.Production*[3] needs to be updated by a developer, Dan needs authorization to access the application. The *Production* administrator may prefer not to create another user account for Dan in *Production* but to assign Dan as a developer to *Sales.Production* instead for the following reasons.

- Dan does not have to switch between user accounts in different domains.
- Intra-domain and cross-domain assignments can be distinguished.
- After the DevOps case is completed, *Production* administrator can avoid removing the temporary user and correlated assignments by simply revoking the cross-domain user assignments.

This example is a typical use case for organizations using either public or community clouds. By design, OpenStack supports cross-domain assignments however they are treated much the same as intra-domain assignments. For example, *Production*'s administrator can assign any user from other domains to roles in *Production*'s projects. This approach may cause a series of problems.

a) *Production* administrator should be able to see *Development* users and their assigned roles in all *Development* projects, at least during authorization time, to issue proper cross-domain assignments.
b) *Development* administrator cannot control the cross-domain authorization.
c) Since DevOps jobs are usually temporary, the management of cross-domain authorization should be flexible, convenient and rapid.
d) No option to specify additional controls upon cross-domain accesses.

On the one hand, the visibility of a user's roles inside its owning domain provides crucial information for other domain administrators to authorize access of the user since the users in OpenStack are not global but identifiable inside each domain. On the other hand, the user owner needs to monitor or constrain the roles assigned to its users in other domains in order to prevent violations of security in multi-domain interoperation [7,18]. In this setting, the collaborating domains should both have control over the cross-domain access instead of only one of them.

Letting the cloud administrator take charge of all cross-domain assignments can solve the visibility issue in Problem a) but the administrative overhead may become overwhelming. Moreover, it is inappropriate for the cloud administrator to be so closely involved in the management within individual domains. To address Problem b), mechanisms that involve both domain administrators should be introduced. For Problem c) we need a rapid means to enable or disable cross-domain assignments for better efficiency. As Problem d) refers, collaborative cross-domain accesses rather than intra-domain accesses need more control related to the authorization.

[3] We use "." to represent the ownership relation between projects and domains. For example, *Sales.Production* refers to the *Sales* project in *Production* domain.

Existing Approaches We have found similar problems in Microsoft Windows Active Directory (AD). An AD, comparable with the identity service in OpenStack, maintains various types of trust relations to allow users in one domain to access resources in another [3]. But they are not directly applicable in the cloud environment since AD is designed to manage identities for a centralized authority but not decentralized ones like in the cloud.

Currently, OpenStack Keystone supports delegation for users. In particular, a user can delegate a part of his or her permissions to another user through a trust relation. The trustee can impersonate the trustor to perform a subset of the permissions that the trustor has been authorized. Issuing a trust relation does not need involvement of the domain administrators so that Problem b) still exists. In addition, it requires a single user to have all the permissions that the requesting user needs in the target project and limits the capability of cross-domain collaborations.

Amazon Web Service (AWS) allows delegating access across accounts (accounts are comparable to OpenStack domains). By creating a trust relation and associating an assumed role with it, the trustor account authorizes the users from the trustee account to access permissions associated with the assumed role in the trustor account. In this way, cross-account accesses are enabled. However, the trust relation cannot support customized control other than the assumed role or be constrained.

Scope and Assumptions In this paper we assume that cross-domain authorization only happens in a single cloud. Nevertheless, the model we propose may be extended to federated cloud scenarios. We assume the users in our models are properly authenticated as supported by Keystone. Our discussion and implementation are based on the Havana release of OpenStack [4].

3 OpenStack Access Control Model

In this section, we present the core OpenStack Access Control (OSAC) model based on the OpenStack Identity API v3 [5] which is relatively the latest stable version. Since OpenStack is a rapidly changing system solving practical problems, we feel it impossible and unnecessary to model every feature in OpenStack identity service. Instead, we keep only the core components in the model and formally present how they interplay with each other in the authorization and administration processes. Hence, the term core OpenStack Access Control model. For simplicity we will often omit the core prefix.

Core OSAC Core OSAC extends the traditional RBAC model [12] to support multi-tenancy. The model elements and relations are defined in Figure 2. OSAC contains eight core entity components: Users (U), Groups (G), Projects (P), Domains (D), Roles (R), Services (S), Operations (O) and Tokens (T). Other entities in the OpenStack Identity API are regarded implementation specific such as credentials, regions and endpoints. Each of the entities has a globally unique

resource identifier provided by the identity service. The Domain Trust (DT) relationship is shown in dashed lines since it is not currently part of OpenStack but is proposed as an extension in this paper.

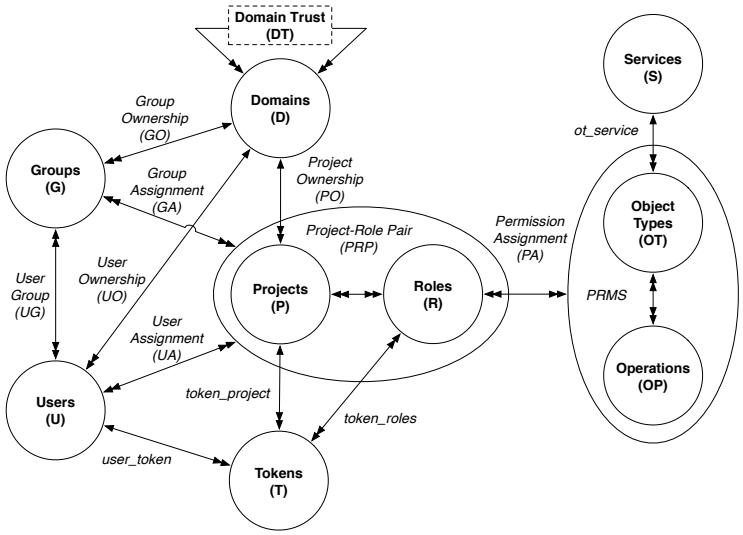


Fig. 2: Core OpenStack Access Control (OSAC) model with domain trust.

Users and Groups. A user represents an individual who can authenticate and access cloud resources. In OpenStack, users are the only consumers of cloud resources. A group is simply a collection of users. Each user or group is owned by one and only one domain. Each group contains only users in its owning domain. Since groups share the nature of users, for convenience reason we understand "users" to mean "users or groups" in the rest of this paper.

Projects. A project is a scope and/or a container of cloud resources. A project manages multiple services and a service segregates its resources into multiple projects. Using a project and a service, we can locate a specific set of resources. For example, the compute service of *Sales.Production* project manages the virtual machine (VM) instances of production applications for the sales department. Each project is owned by one and only one domain.

Domains. A domain is an administrative boundary of users, groups and projects. Domains are mutually exclusive. Each user, group and project belongs to one and only one domain.

Roles. Roles are global names which are used to associate users with any of the projects. A user is assigned a role with respect to a project, in other words to a project-role pair.[4] Users can be authorized permissions only through roles. The functions of roles may vary drastically in different services depending on the nature of the service.

[4] Users can also be assigned a domain-role pair. This is for administrative usage only and will be discussed in Section 3.

Services. A service represents a distributed cloud service. Since OpenStack and most of other cloud systems are designed following service-oriented architecture (SOA) model, cloud applications and resources are delivered to the customers as services. The core service types in OpenStack include compute, image, identity, volume and network.

Object Types and Operations[5]. An object type represents a kind of cloud resources such as VM or image. Each service may provide multiple object types. For example, within the network service, IP and port are different object types. An operation is an access method to the object types. General operations are create, read, update and delete (CRUD) interacting with object types.For example, a typical permission "Delete VM" is a combination of delete operation and VM object type. Note that in cloud environments we cannot specify a particular object in the policy since the objects are created "on-demand". Thus, the finest-grained access control unit is a collection of objects identified by a specific object type and a specific project.

As a role-based authorization model, the central part of OSAC is the assignments related to roles: user assignment (UA), group assignment (GA) and permission assignment (PA) as illustrated in Figure 2. Both groups and users are assigned to project-role pairs but permissions are assigned to roles. As a result, the permissions assigned to a role populates across all the projects. For example, if Dan is assigned a developer role in both *Sales.Development* and *HR.Development*, the permissions available to Dan through the developer role in both projects are identical. This arrangement embodies the multi-tenant nature of cloud resources and provides great flexibility for the assignments as long as the definition of roles is consistent within each service. It is worth noting that user assignments and group assignments are managed centrally in the identity service which permission assignments are distributed into each service.

Tokens. A token represents a subject acting on behalf of a user. A token is issued by the identity service for an authenticated user and then validated by other services whenever the user requests cloud resource accesses. A token may be expired or revoked during its lifetime. The content of a token is encrypted with public key infrastructure (PKI) so that it cannot be altered during transportation. It reveals all the information needed to authorize the access including the accessing user, the target project[6], all the assigned roles in the project[7] and service catalogs. The function *user_tokens* returns the set of tokens that are associated with a user, the function *token_project* returns the target project and the function *token_roles* returns the roles assigned to the user in the target project. Typically a user is issued one token for each project. Thus, in a partic-

[5] For clarity, we introduce object types and operations as components of permissions to the OSAC model. There is no specification of these two concepts in the identity service API.

[6] The accessing scope may be project, domain, or even unscoped. For ordinary accesses, a token is scoped to a project

[7] Currently, OpenStack does not support activating an arbitrary subset of roles assigned to a user in the project.

ular project, the permissions available to the user are the permissions assigned to the roles revealed in the correlated token.

We summarize the above in the following definition.

Definition 1. *Core OSAC model has the following components.*

- U, G, P, D, R, S, OT, OP *and* T *are finite sets of users, groups, projects, domains, roles, services, object types, operations and tokens respectively.*
- $user\_owner : U \rightarrow D$, *a function mapping a user to its owning domain. Equivalently viewed as a many-to-one relation* $UO \subseteq U \times D$.
- $group\_owner : G \rightarrow D$, *a function mapping a group to its owning domain. Equivalently viewed as a many-to-one relation* $GO \subseteq G \times D$.
- $project\_owner : P \rightarrow D$, *a function mapping a project to its owning domain. Equivalently viewed as a many-to-one relation* $PO \subseteq P \times D$.
- $UG \subseteq U \times G$, *a many-to-many relation assigning users to groups where the user and group must be owned by the same domain.*
- $PRP = P \times R$, *the set of project-role pairs.*
- $PERMS = OT \times OP$, *the set of permissions.*
- $ot\_service : OT \rightarrow S$, *a function mapping an object type to its associated service.*
- $PA \subseteq PERMS \times R$, *a many-to-many permission to role assignment relation.*
- $UA \subseteq U \times PRP$, *a many-to-many user to project-role assignment relation.*
- $GA \subseteq G \times PRP$, *a many-to-many group to project-role assignment relation.*
- $user\_tokens : U \rightarrow 2^T$, *a function mapping a user to a set of tokens; correspondingly,* $token\_user : T \rightarrow U$, *mapping of a token to its owning user.*
- $token\_project : T \rightarrow P$, *a function mapping a token to its target project.*
- $token\_roles : T \rightarrow 2^R$, *a function mapping token to its set of roles. Formally,* $token\_roles(t) = \{r \in R | (token\_user(t), (token\_project(t), r)) \in UA\} \cup (\bigcup_{g \in user\_groups(token\_user(t))} \{r \in R | (g, (token\_project(t), r)) \in GA\})$.
- $avail\_token\_perms : T \rightarrow 2^{PERMS}$, *the permissions available to a user through a token, Formally,* $avail\_token\_perms(t) = \bigcup_{r \in token\_roles(t)} \{perm \in PERMS | (perms, r) \in PA\}$.

Role hierarchy (RH) is not supported in OSAC but it could be a reasonable extension for convenience. Depending on operation needs, the hierarchy relation may be added upon roles or to project-role pairs. Both approaches allow specification of role-hierarchy assignments in the centralized identity service while the former also supports distributed assignment since different service may build different structures of role hierarchy as needed. Consideration of these extensions is beyond the scope of this paper.

Administrative OSAC Model As described previously, the identity information of all the entities including services, domains, users, groups, projects and roles are stored and managed by the Keystone identity service in OpenStack, as are the assignments associating users and groups with roles in domains or projects. It is worth to note that the permission assignments are separately

maintained by each cloud service provider in a policy file. The policy file for the identity service specifies the permissions to manage identities and assignments for administrator roles.

The administrative OSAC (AOSAC) model consists of three levels of administrative roles: *cloud_admin*, *domain_admin* and *project_admin*. As their names indicate, *cloud_admin* refers to top-level administrators with the CSP managing all the information in the identity service; *domain_admin* at the middle-level is able to conduct administrative tasks within the associated domain; and *project_admin* at the bottom-level take the responsibility of managing UA and GA assignments for the associated project. A user can only be assigned to *cloud_admin* role at the installation time of the cloud or by other users with the *cloud_admin* role afterwards. The *domain_admin* and *project_admin* roles are assigned to users by associating the users with the "admin" role in a specific domain or project respectively. Figure 3 illustrates an example administrative role hierarchy in AOSAC.

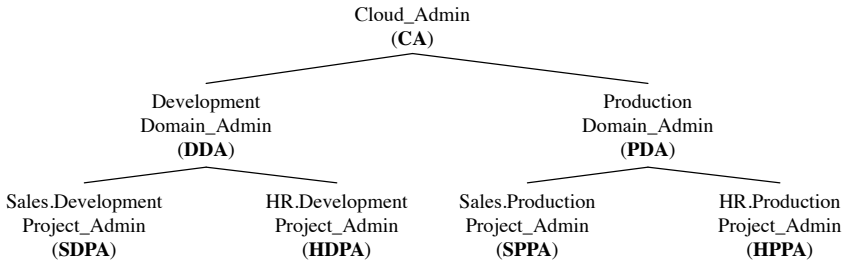


Fig. 3: An example administrative role hierarchy.

In the DevOps example described in Section 2, *Development domain_admin* (DDA) and *Production domain_admin* (PDA) roles are assigned to users owned by each domain respectively. A PDA can list and view users, groups and projects in *Production*. He or she can also assign roles, including the "admin" role, in a project of *Production* to a user. A *Sales.Production project_admin* ($SPPA$) can assign roles other than the "admin" role in *Sales.Production* to a user. Note that a PDA or a $SPPA$ can assign *Dennis@Development* to the "developer" role in *Sales.Production*. As a result, DevOps cross-domain accesses may be authorized. However, the administrative boundary of the two domains are intersected with each other. This may lead to unwanted authorization in cross-domain collaboration, such as the DevOps example.

4 Domain Trust Model

In order to achieve additional control for cross-domain accesses, we propose domain trust models integrating with the OSAC model. From the description in the previous sections, we observe that domains are introduced as administrative boundaries. Bridging domains using trust relations gives a controlled way to allow cross-boundary collaborations. For a user to have roles in a project, a

proper trust relation needs to be established between the owning domains of the user and the project.

Domain Trust Relation The definitions of trust relations vary in different application scenarios. In the field of access control, either explicit or implicit trust relation is essential to decentralized authorization [9]. Thus, in order to properly authorize cross-domain accesses, we have to specify what a trust relation means and how the trust relation interacts with the existing access control model.

Trust is a complicated concept and has been treated in different ways in the context of access control. The following is a list of characteristics related to domain trust relations. Figure 4 depicts the potential combinations.

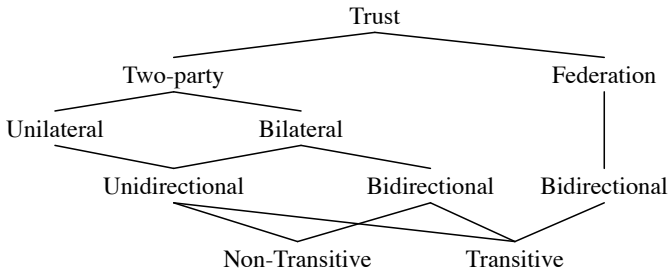


Fig. 4: A tree structure showing characteristics of domain trust relation.

Protocol (Two-party vs Federation). Two-party trust is established between two domains. A federation trust exists in an alliance or cooperative association in which a participant that is a domain trusts each other participants and it is also true in return.

Initiation (Bilateral vs Unilateral). When the trustor creates a trust relation, if the trustee is required to confirm, then the trust relation is regarded as bilateral otherwise unilateral. It is worth to note that transferring a unilateral trust relation to a bilateral one is much easier than doing the reverse.

Direction (Bidirectional vs Unidirectional). A bidirectional trust relation requires the actions enabled through the trust relation are equally available for the trustor and the trustee. Conversely, a unidirectional trust, as the name refers, requires availability of the actions only on one side.

Transitivity (Transitive vs Non-transitive). For domain A, B and C, if A trusts B and B trusts C it is implied that A trusts C, then the trust relation is transitive. Otherwise, the trust relation is non-transitive.

In this paper, the domain trust relation is specified as an two-party, unilateral, unidirectional and non-transitive relation. Moreover, it is reflexive meaning each domain trusts itself. It is functionally defined as the following.

Definition 2. *If and only if Domain A trusts Domain B, also written as "$A \trianglelefteq B$", A or B can perform unidirectional cross-domain authorization.*

The actions enabled through a domain trust relation depend on the trust types defined as follows.

Definition 3. *Based on collaborative access control needs, the domain trust relation described in Definition 2 can be categorized into three useful types.*

- ***Type-**α, requires visibility of the trustee's user information for the trustor to assign trustee's users to roles in trustor's projects, written as "*$\trianglelefteq_\alpha$*".*
- ***Type-**β, requires the trustor to expose its user information for the trustee to assign trustor's users to roles in trustee's projects, written as "*$\trianglelefteq_\beta$*".*
- ***Type-**γ, requires the trustor to expose its project information for the trustee to assign trustee's users to roles in trustor's projects, written as "*$\trianglelefteq_\gamma$*".*

Type-α Trust is used implicitly in current OpenStack since the trustor *domain_admin* and *project_admin* can see the users in all the domains and assign them to roles in the trustor's projects. Type-α Trust is only useful when user related information is not sensitive and available across domains by default. In contrast, Type-β Trust and Type-γ Trust protect user information as sensitive property of each domain. Both of them require dual control. In particular, the trustor manages the trust relation while the trustee manages cross-domain authorization. In this way, cross-domain accesses can be revoked by either end of the trust relation.

OSAC Domain Trust Since we have specifically defined the domain trust relation above, integrating it with OSAC becomes straightforward. The formal definition of the OSAC Domain Trust (OSAC-DT) model follows.

Definition 4. *The OSAC-DT model extends the OSAC model in Definition 1 with the following modifications.*

- $DT \subseteq D \times D$*, a many-to-many trust relation on* D*, also written as "*$\trianglelefteq$*".*
- UA *is modified to require that* $(u, (p, r)) \in UA$ *only if* $project\_owner(p) \equiv user\_owner(u) \vee project\_owner(p) \trianglelefteq_\alpha user\_owner(u) \vee user\_owner(u) \trianglelefteq_\beta project\_owner(p) \vee project\_owner(p) \trianglelefteq_\gamma user\_owner(u)$.
- GA *is modified to require that* $(g, (p, r)) \in GA$ *only if* $project\_owner(p) \equiv group\_owner(g) \vee project\_owner(p) \trianglelefteq_\alpha group\_owner(g) \vee group\_owner(g) \trianglelefteq_\beta project\_owner(p) \vee project\_owner(p) \trianglelefteq_\gamma group\_owner(g)$.

The modification focuses on the effect of the domain trust relation introduced. Particularly, the project owner has to trust the user owner for UA and GA to take effect. The trust relation is checked during both authorization time and accessing time so that if it is revoked the correlated cross-domain accesses may be automatically or manually revoked depending upon implementation.

OSAC-DT allows the three types of trust relations to coexist with each other. A specific cross-domain UA or GA is effective as long as the trust relation between the user or group and the project domains satisfy the condition described in Definition 4. In fact, combining Type-α and Type-β trusts we achieve a bilateral trust relation. For example, only if both $Production \trianglelefteq_\alpha Development$

and $Development \trianglelefteq_\beta Production$ exists, then cross-domain authorization by *Production* is enabled.

By introducing explicit domain trust relation, the following constraints may be enforced over cross-domain authorization.

Separation of Duties (SoD). Some of the collaborations among domains may have conflict of interests which should be addressed by additional constraint policy and lists of mutually exclusive domains.

Minimum Exposure. In collaboration, the over-exposure of user or project information increases security and privacy risks. An effective solution is limiting exposure of information based on each domain or each trust requirements.

Cardinality. A domain may limit the number of domains to be trusted. For example, some domains , such as *Production*, require high-level security and allow only one trusted domain at a time for temporary access if necessary.

The constraints listed above and a lot more are previously not available without domain trust relations.

Domain Trust Administration The administrative OSAC-DT (AOSAC-DT) model extends the AOSAC model by the administration of domain trust relations and their enabled actions. Since the trust relation is unilateral, only the *cloud_admin* the *domain_admin* of the trustor and have permission to create and revoke a specific domain trust relation. The trust relation enables the *project_admin* and *domain_admin* of the trustor, in case of Type-α trust, or the trustee, in case of Type-β trust or Type-γ trust, to view the user or project information necessary for them to make cross-domain authorization.

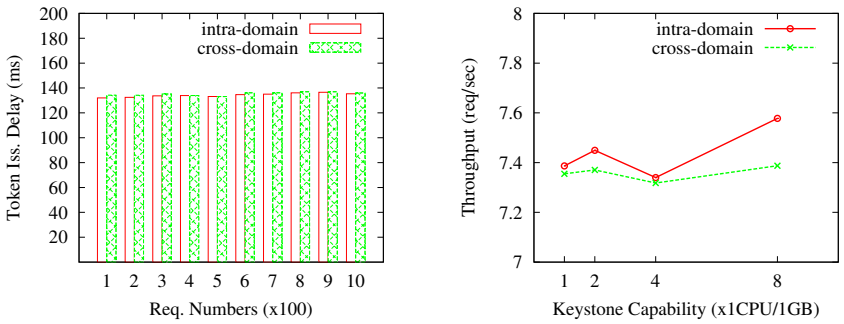
5 Prototype and Evaluation in OpenStack

To further explore the feasibility of our OSAC-DT model, we implement a prototype system based on the Havana release of Keystone source code [4]. Furthermore, we conduct experiments on the prototype system in terms of performance and scalability. The results turn out to be convincing that the integrated domain trust introduces minimum authorization overhead.

Implementation Overview The architecture of our prototype follows the Keystone design. The domain trust verification process intercepts the authentication process. Before Keystone issues the token for a requesting user, the domain trust relations stored in the MySQL database are checked. Only if the requesting user's owning domain is trusted by the target project's owning domain, then the token issuing process can go through. Otherwise, an "unauthorized" response will be returned. For proof of concept purpose, we implement only Type-γ trust in the prototype system. It is straightforward to extend the implementation other types of domain trust relations discussed in Section 4 and similar evaluation results are predicted because the domain trust verification processes are similar.

Evalutation The implementation and experiments are conducted in experimental Devstack [2] deployments in a private cloud. The core OpenStack services, including Keystone, are running on a single VM. The requesting clients

are from the same data center network of the private cloud. Since only Keystone code is modified, the experiments focus on evaluating the token issuing process including sequential processes of authentication, domain trust verification, token composition and network transmission, etc.



(a) performance (b) scalability

Fig. 5: Performance and scalability evaluation results

The experiments simulate sequential token requests for one-hundred users and projects owned by ten independent domains. Each user is associated with both intra-domain and cross-domain assignments through ten different roles. As Figure 5a shows, the x-axis represents the requests per user and the y-axis indicates the latency between request time and response time from the client end, also known as the token issuing delay. Comparing the token issuing delay of intra-domain and cross-domain access requests, the domain trust verification process costs 0.96 ms on average or 0.7% performance overhead which is acceptable.

Figure 5b presents the results for scalability tests on our prototype system. The x-axis represents the capability of the VM running Devstack in the unit of "1CPU/1GB RAM". The y-axis is the calculated throughput for the token issuing process. The plotted diagram shows that with ten requests per user, the throughput increase of the prototype system is proportional to the increase of the capability of Keystone servers from 1 unit to 8 units so that the system is scalable and adding the domain trust does not cause scalability problem.

6 Related Work

Role-Based Access Control (RBAC) [12] is a dominant model in single organization scenarios. ROBAC [24], one of the RBAC extensions, is able to manage authorization for multiple organizations, comparable to domains, but collaboration among organizations are not allowed. GB-RBAC [15] supports collaboration among groups. Yet, the group admin can not manage the users inside the group which is different with domains in OpenStack. More importantly, most of RBAC extensions need a centralized authority to administer each collaboration. In the cloud environment, the centralized authority is the cloud service provider (CSP) who is inappropriate and incapable to manage all the collaborations.

Role-based delegation models [8,13,14,23] are designed to solve collaboration problems. However, the chained delegation relations are not flexible enough. Since the entities in the cloud are created on-demand and deleted afterwards, any node of the chain may disappear resulting in void delegation. Comparing to OSAC-DT, the trust relation is always created and maintained by the trustor and non-transitive so that the management of trust relations is simple.

Secure multi-domain interoperation solutions [19,18] leverage role mapping techniques between domains to facilitate interoperation. Role mapping is a form of role hierarchy linking roles across domains. But it cannot be integrated with OpenStack since role hierarchy is not supported. Even if role hierarchy can be established, as we discussed in the OSAC model, role-mapping will not function until the PA becomes project-specific or domain-specific. In OSAC-DT, the trust relations is on domains and only affects UA and GA which are project-specific. Authorization services [6,11,16] in the Grid leverage decentralized trust management [9]. In order to establish collaboration, maintaining credentials between nodes becomes a huge performance overhead which could be avoided in the Cloud by the centralized identity service.

OSAC-DT is closely related to the models like MTAS [10,22], MT-RBAC [20] and CTTM [21]. OSAC-DT differs in its compatibility with the OpenStack Identity API v3. Further, the administrative model also merges with the OpenStack Keystone management. Ray et al [17] propose a formal trust-based delegation model solving similar problems in mobile cloud environment. However, the calculated trust relation is inappropriate and unnecessary between domains in OpenStack since domains have the authority to assign trust relations with each other.

7 Conclusion

In this paper, we present a formalized OpenStack access control model from which we propose a domain-trust extension to better facilitate the decentralized authorization for cross-domain collaborations. There are three useful types of OpenStack-specific domain trust relations, intuitive trust, user-aware trust and project-aware trust, applicable to various collaboration needs. Further, we implement a proof of concept prototype system with project-aware trust based on Keystone Havana release source code. The experiment results show that the integrated domain trust model is acceptable in both performance and scalability.

Acknowledgement

Sincere gratitude is hereby extended to the following. Farhan Patwa, director of ICS, for his patient help on the OpenStack implementation and active connections with the OpenStack community. Dolph Mathews, PTL of Keystone, for his reviews on the preliminary OSAC model and the domain trust blueprint. Dr. Jaehong Park, research associate professor of ICS, for his insights and comments leading to improvement of the OSAC model. This work is partially supported by grants from the National Science Foundation and AFOSR MURI program.

References

1. DevOps. http://en.wikipedia.org/wiki/DevOps
2. Devstack. http://www.devstack.org
3. Microsoft windows active directory. http://en.wikipedia.org/wiki/Active_Directory
4. OpenStack Havana Release. http://www.openstack.org/software/havana
5. Openstack identity service api v3 (stable). http://developer.openstack.org/api-ref-identity-v3.html
6. Alfieri, R., Cecchini, R., et al: From gridmap-file to VOMS: managing authorization in a grid environment. Future Generation Computer Systems 21(4), 549–558 (2005)
7. Baracaldo, N., Masoumzadeh, A., Joshi, J.: A secure, constraint-aware role-based access control interoperation framework. In: Proc. of the 5th International Conference on Network and System Security (NSS). pp. 200–207. IEEE (2011)
8. Barka, E., Sandhu, R.: Framework for role-based delegation models. In: Proc. of the Annual Conf. on Comp. Sec. Applications (ACSAC). pp. 168–176. IEEE (2000)
9. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: Proc. of the 1996 IEEE Symp. on Security and Privacy. pp. 164–173. IEEE (1996)
10. Calero, J.M.A., Edwards, N., et al: Toward a multi-tenancy authorization system for cloud services. IEEE Security & Privacy Nov/Dec 2010, 48–55 (2010)
11. Chadwick, D.W., Otenko, A.: The PERMIS X. 509 role based privilege management infrastructure. vol. 19, pp. 277–289. Elsevier (2003)
12. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. TISSEC 4(3), 224–274 (Aug 2001)
13. Freudenthal, E., Pesin, T., et al: dRBAC: distributed role-based access control for dynamic coalition environments. In: Proc. of ICDCS. pp. 411–420. IEEE (2002)
14. Li, N., Mitchell, J.C., et al: Design of a role-based trust-management framework. In: Proc. of IEEE Symp. on Sec. and Privacy. pp. 114–130. IEEE (2002)
15. Li, Q., Zhang, X., Xu, M., Wu, J.: Towards secure dynamic collaborations with group-based RBAC model. Computers & Security 28(5), 260–275 (2009)
16. Pearlman, L., Welch, V., Foster, I., et al: A community authorization service for group collaboration. In: Proc. of intl. POLICY. pp. 50–59. IEEE (2002)
17. Ray, I., Mulamba, D., Ray, I., Han, K.J.: A model for trust-based access control and delegation in mobile clouds. In: IFIP DBSec, pp. 242–257 (2013)
18. Shafiq, B., Joshi, J.B., Bertino, E., Ghafoor, A.: Secure interoperation in a multidomain environment employing RBAC policies. IEEE Transactions on Knowledge and Data Engineering 17(11), 1557–1577 (2005)
19. Shehab, M., Bertino, E., Ghafoor, A.: SERAT: SEcure role mApping technique for decentralized secure interoperability. In: Proc. of SACMAT. pp. 159–167 (2005)
20. Tang, B., Li, Q., Sandhu, R.: A multi-tenant RBAC model for collaborative cloud services. In: Proc. of IEEE Conf. on Privacy, Security and Trust (PST) (2013)
21. Tang, B., Sandhu, R.: Cross-tenant trust models in cloud computing. In: Proc. of IEEE Conf. on Information Reuse and Integration (IRI) (2013)
22. Tang, B., Sandhu, R., Li, Q.: Multi-tenancy authorization models for collaborative cloud services. In: Proc. of Intl. Conf. on Collab. Tech. and Sys. (CTS) (2013)
23. Zhang, X., Oh, S., Sandhu, R.: PBDM: a flexible delegation model in RBAC. In: Proc. of SACMAT. pp. 149–157. ACM (2003)
24. Zhang, Z., Zhang, X., Sandhu, R.: ROBAC: Scalable role and organization based access control models. In: Proc. of CollaborateCom. pp. 1–9. IEEE (2006)