Design and Implementation of Access Control as a Service for IaaS Cloud

Ruoyu Wu
Arizona State University
Email: ruoyu.wu@asu.edu

Xinwen Zhang
Huawei Research Center
Email: xinwen.zhang@huawei.com

Gail-Joon Ahn
Arizona State University
Email: gahn@asu.edu

Hadi Sharifi
Arizona State University
Email: hsharif1@asu.edu

Haiyong Xie
USTC & Huawei Research Center
Email: haiyong.xie@ustc.edu

ABSTRACT

Organizations and enterprises have been outsourcing their computation, storage, and workflows to Infrastructure-as-a-Service (IaaS) based cloud platforms. The heterogeneity and high diversity of IaaS cloud environment demand a comprehensive and fine-grained access control mechanism, in order to meet dynamic, extensible, and highly configurable security requirements of these cloud consumers. However, existing security mechanisms provided by IaaS cloud providers do not satisfy these requirements. To address such an emergent demand, we propose a new cloud service called *access control as a service* (ACaaS), a service-oriented architecture in cloud to support multiple access control models, with the spirit of pluggable access control modules in modern operating systems. As a proof-of-concept reference prototype, we design and implement ACaaS$_{RBAC}$ to provide role-based access control (RBAC) for Amazon Web Services (AWS), where cloud customers can easily integrate the service into enterprise applications in order to extend RBAC policy enforcement in AWS. We describe challenges and lessons in implementing ACaaS$_{RBAC}$, demonstrate how this service can be seamlessly integrated with enterprise cloud applications, and discuss evaluation results.

I INTRODUCTION

Although cloud computing brings many benefits, security issues have impacted its wide adoption for enterprises and organizations. In this paper, we focus on addressing access control issues in public Infrastructure-as-a-Service (IaaS) cloud. There are several challenges for controlling resource accesses in such a cloud environment, compared with the problem in legacy systems within an organization. First, a multi-tenant computing environment in clouds demands strong isolation between virtualized resource usages among multi-tenants on the same physical resources, while in a legacy enterprise environment, the single domain owns all computing resources. Secondly, since cloud computing is a service-oriented computing model, the access control mechanism of a *cloud provider* should be configurable in very flexible way such that it satisfies many different customers' organizational security policies, such as role-based access control for enterprises and multi-level security for government agencies. Current public cloud provider lacks such an important flexibility. For example, Amazon Web Services (AWS), the leading IaaS provider, only supports identity-based authorization for cloud customers with its Identity and Access Management Services (IAM) [2]. Thirdly, completely delegating access control to a cloud provider–including policy management, storage, and security enforcement–requires strong trust relationship and implementation dependency between a cloud customer and the cloud provider. With the separation of computing resource ownership and usage, we believe separating security policies and their enforcement reduces these dependencies.

In light of service-oriented computing model, we propose a new cloud service for access control called *access control as a service* (ACaaS). The core idea of ACaaS is to outsource access control policy management and storage to service providers, which provides value-added functions for organizations with security expertise. A cloud customer (e.g., an enterprise or its security administrators) specifies and manages security policies and configurations with interfaces provided by ACaaS service providers. These high level security policies are then converted to low level and enforceable policies for individual cloud providers. The separation of this service-oriented security management and customized enforcement in different cloud providers not only reduces the trust management cost of cloud providers for enterprise customers, but also offers great flexibility for cloud customers to develop their own security policies based on organizational requirements, without worrying about their enforcement mechanisms in a concrete cloud environment. Furthermore, ACaaS enables a cloud customer to choose different cloud providers for security reason without a permanent lock-in.

We propose a modular architecture for ACaaS for public IaaS cloud, where variant security modules can be plugged in for different cloud customers, e.g., to support role-based access control (RBAC) policies, multi-level security policies, Chinese Wall security policy, and so on. Also, our architecture flexibly supports many public cloud infrastructures with web services based APIs. As a case study and reference implementation, we design and implement $ACaaS_{RBAC}$ for AWS, an ACaaS module that configures RBAC policies and converts to AWS IAM policies such that the access requests to AWS resources from a customer's user (e.g., the employees of an enterprise that uses AWS) are controlled based on the enterprise's security policies. Specifically, in this paper:

- We propose a new modular architecture for access control called *access control as a service* (ACaaS) in cloud computing environments, which configures and manages multiple access control policy models for variant cloud customers' security requirements, and converts to enforceable security policies in public cloud providers. That is, ACaaS enables securely and efficiently outsourcing access control management of an organization in clouds (Section II);

- We identify the limitations of the existing access control mechanism of AWS IAM and design ACaaS$_{RBAC}$, a reference ACaaS architecture that supports RBAC policies to address those limitations. We articulate the design challenges as well as formalize a corresponding ACaaS$_{RBAC}$ model for AWS (Section II, III, and IV);

- We implement a prototype system, and provide web-based administrative tool and web services APIs for third party applications' integration. We demonstrate the practicality, efficiency, and scalability of our system through a case study and performance evaluation (Section V).

II ACaaS FOR CLOUDS

In this section, we first present two motivating scenarios. We then discuss the overview of ACaaS for cloud computing followed by its motivations for AWS customers.

1 MOTIVATING SCENARIOS

IT sandbox in clouds IT sandbox is an isolated computing environment which can be used for software development, security testing, pre-production testbeds, training and IT labs, among others. In many organizations, IT sandbox environments are provisioned for dynamic workloads but not well compatible with capital intensive in-house data center resources. Therefore, cloud computing environments have been considered for addressing such a limitation. Various types of sandboxes can be created to meet different project needs in software development life cycle, such as development sandbox, integration sandbox, demo sandbox, testing sandbox, and production sandbox. Those sandboxes are dynamically created and terminated using scalable cloud resources. For instance, a development sandbox is created with pre-configured tools (e.g., IDEs, SSH, and SVN) and it consumes cloud services (e.g., EC2, S3, and RDS) for computing and storage. There are various functional roles in each project such as project manager, developer, tester, and security administrator. Users with different roles have different access privileges on sandbox resources. In such a dynamic environment, it is critical to ensure that cloud resources associated with a certain sandbox can be accessed only by authorized users with a low administrative cost.

Government cloud platform According to a recent study by KPMG International [1], 28 % of overall IT expenditures in government agencies would be in clouds by the end of 2012. US NIST has designed its cloud computing program to support accelerated US government adoption. Along with this broad shift, security concerns are the most significant barriers to the adoption of cloud computing by government sectors. Many government agencies must comply with regulatory statutes, such as the Health Insurance Portability and Accountability Act (HIPAA) [10], the Sarbanes-Oxley Act of 2002 (SOX) [20], and the Federal Information Security Management Act (FISMA) [9] in US. Security and regulatory compliance needs vary in terms of different levels of government sectors such as federal, state, and local governments and different administrative dimensions such as academia, civil, and military. Therefore, diverse security requirements should be accommodated in the government cloud platform, while a single cloud provider usually lacks the capability to support all of these requirements.

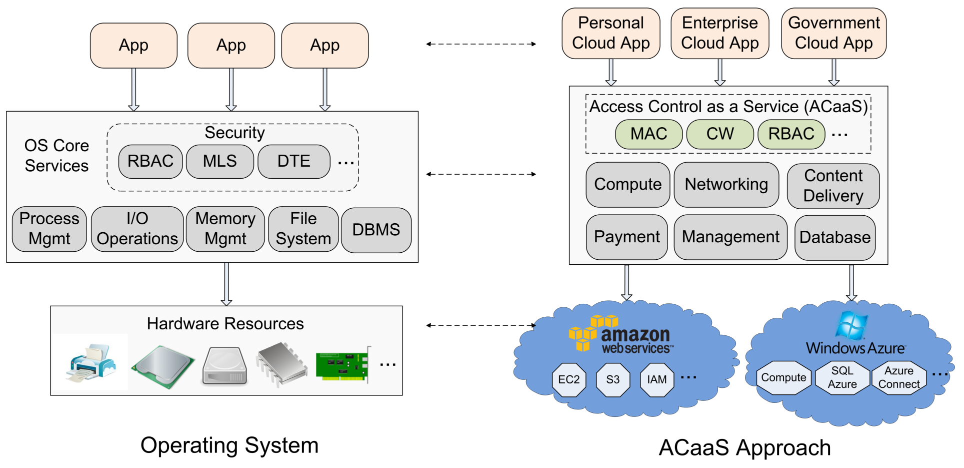


Figure 1: ACaaS vs. security modules in operating system.

2 OVERVIEW

Securely maintaining valuable digital assets in clouds is critical for both cloud service providers and customers. The diversity of cloud services across a wide range of organizations and domains requires various security requirements. Accordingly, a comprehensive and adaptive access control mechanism needs to be in place to support various security policy models for the diverse security needs. However, current cloud computing platforms such as AWS, Windows Azure, Google App Engine, and Eucalyptus all fail to meet such identified needs. Towards this, we propose the concept of access control as a service (ACaaS) with the spirit of pluggable access control modules in modern operating systems. As shown in Figure 1, we draw an analogy between computing, storage, network, and other resources provided by IaaS providers and hardware resources in physical machines such as CPU, disk, and network stack. Cloud provider offerings can be mapped to operating system services such as process management, memory management, scheduling, I/O operations, and networking. For instance, process management conducts basic tasks including starting and suspending processes, CPU allocation, and scheduling for multiple processes. Similarly, computing services in a cloud handle booting and terminating virtual machines instances, allocating resources, and scheduling computing tasks and workflows. For security purposes, authorization modules and policies in traditional operating systems (e.g., Linux) can be dynamically loaded (e.g., SELinux modules), and every access to underlying resources from processes and applications is then be controlled. Similarly, ACaaS can load different access control modules and support various security policy models for different cloud customers, such as mandatory access control (MAC) [15], the Chinese Wall security policy (CW) [3], and role based access control [17]. This plug & play fashion enables parallel evolution of cloud customer's own policy specifications and a cloud provider's security enforcement mechanisms.

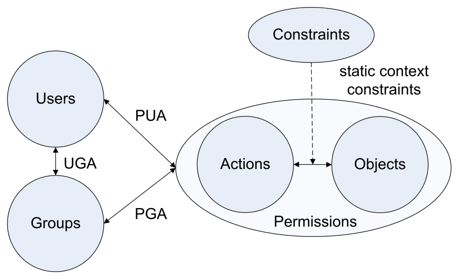


Figure 2: Security model of AWS IAM.

3 AWS ACCESS CONTROL SERVICE AND ITS LIMITATIONS

To motivate the design and development of ACaaS, we analyze the access control mechanism in Amazon AWS cloud platform. AWS is a leading cloud platform with a suite of IaaS services, such as elastic compute cloud (EC2), simple storage service (S3) and elastic block store (EBS), SQL and NoSQL database (SimpleDB and DynamoDB), simple workflow service (SWF), and content distribution services (CloudFront). Among various services AWS provides, we particularly analyze its access control service called Identity and Access Management (IAM) [2] which enables an organization to securely control users' access to the AWS services and resources subscribed by the organization. IAM defines security policies with a set of pre-defined components. Figure 2 shows an abstract representation of IAM which consists of following components: *Users*, *Groups*, *Actions*, *Objects*, *Permissions*, *Constraints*, *User-Group-Assignment* (UGA), *Permission-User-Assignment* (PUA), and *Permission-Group-Assignment* (PGA). *Permissions* are defined in the form of *Actions* on *Objects* under certain *Constraints*. Note that since some AWS action APIs are not necessarily bounded with objects such as `CreateKeyPair`, which is an action API of EC2 service to create key pair for instances, permissions can also contain only actions in that case. *Objects* are identified using Amazon resource name (ARN). *Constraints* are imposed on permissions in the form of key-value pairs. Each key-value pair can be one of following types including `String`, `Numeric`, `Date and Time`, `Boolean`, and `IP address`. For example, the statement `‘‘DateLessThan":{‘‘aws:CurrentTime": ‘‘2012-06-01T00:00:00Z"}` uses the `Date and Time` type `DateLessThan` constraint with the `aws:CurrentTime` key to specify that the request must be received before June 1, 2012. Permissions can be directly assigned to users or groups. Users explicitly own permissions assigned to groups they belong to.

```
{"Statement":[{
  "Effect":"Allow",
  "Action":["iam:CreateAccessKey","iam:ListAccessKeys"],
  "Resource":"arn:aws:iam::123456789012:user/*",
  "Condition":{
    "DateGreaterThan":{
      "aws:CurrentTime":"2010-07-01T00:00Z"}}},
  {"Effect":"Allow",
  "Action":["sdb:CreateDomain","sdb:DeleteDomain"],
  "Resource":"arn:aws:sdb:*:123456789012:domain/*"}]
}
```

Figure 3: Example AWS IAM policy.

Based on the above-mentioned model components, an IAM policy statement can be formally defined as a 4-tuple `P = (user, permission, constraint, effect)`, where `effect` can be `Allow` or `Deny`. An IAM policy can contain several IAM policy statements. For example, an IAM policy in JSON format with two policy statements is shown in Figure 3. The user or group that the policy is attached to is not explicitly shown in the policy statements, who can be any user or group within the root AWS account with ID 123456789012. The root AWS account user can explicitly assign this policy to his users or groups. This policy authorizes users to perform the following tasks: (i) Create and list the access keys for any user in the AWS account, starting on July 1, 2010; and (ii) Create and delete Amazon SimpleDB domains in the 123456789012 AWS account for any region.

AWS IAM enables cloud customers to manage users and user permissions to secure their resources in clouds. However, we identify several limitations of IAM for enterprise cloud customers as follows:

1. IAM directly assigns permissions to users. With increasing outsourcing computing infrastructures to IaaS, the number of users and permissions can be quite dynamic and in a very large scale. The management cost for the mapping between users and permissions can be extremely high.

2. IAM supports groups to categorize users and let users explicitly obtain permissions assigned to groups they belong to. However, groups are organized in a flat structure, which cannot reflect the hierarchical structures of organizations. For example, a global sales department of a multinational company should have all the permissions of its regional sales departments. Besides, if an IAM policy is removed from a group, the permission associated with the policy is revoked as well, which is not necessary in many cases.

3. IAM allows to specify static constraints on permissions. However, it lacks a systematic support for many other important constraints such as separation of duty (SoD), a well-known principle for preventing the potential fraud. SoD divides the responsibility of a critical task into different people. When many financial and governmental systems are shifting into cloud platforms, SoD issues become even more critical. If permissions with conflict-of-interest issues are assigned to the same user, many valuable assets in clouds can be jeopardized.

4. Session management is missing in IAM such that all permissions of users are effective all the times, which conflicts with the principle of least privilege. Users should be able to manage their sessions for performing tasks. Besides, without session management, dynamic SoD cannot be enforced.

5. IAM does not distinguish administrators and regular users clearly. The root user with the AWS account has both administrative and regular permissions, which also conflicts with the principle of least privilege. Ideally, permissions associated with an AWS account should be split into multiple units.

AWS recently released a new IAM role feature, which enables an EC2 instance running with a predefined IAM role to securely access other AWS service APIs [5]. However, this feature is still too preliminary and coarse-grained to address these limitations. First, it only supports assigning IAM roles to an EC2 instance level but not to user level. Hence, all applications running in an EC2 instance assume the same set of permissions. This violates the least privilege principle, since it is very general that different applications in an EC2 instance run on behalf of different users to have different permissions. Second, an IAM role does not support session management such that an EC2 instance can only run with a single IAM role. Without re-launching the EC2 instance, it is impossible to switch between different IAM roles during the runtime. Furthermore, EC2 role does not support other important RBAC features such as role hierarchy and delegation. Fundamentally, we claim that the EC2 role is more similar to the traditional concept of "group" in access control and there is no RBAC model formulated in AWS.

III OVERVIEW OF ACaaS$_{RBAC}$ FOR AWS

In this section, we present the overview of ACaaS$_{RBAC}$, a reference architecture of ACaaS that supports RBAC for Amazon AWS cloud platform. We first give an introduction of RBAC, then articulate challenges on supporting RBAC for AWS cloud platform, followed by presenting a formal ACaaS$_{RBAC}$ model for AWS. There are several reasons that we choose to support RBAC for AWS cloud platform. First, RBAC has been widely adopted in enterprise applications. When those applications are moving to clouds, RBAC should be naturally supported in cloud environments. According to a recent cloud market overview [21], RBAC is one of the key criteria to evaluate cloud computing solutions. Second, RBAC is a very generic access control model, and we believe tackling RBAC in clouds requires to address many challenges that will be identically addressed for supporting other access control models with ACaaS. Third, prior research and implementations [8, 18] have demonstrated that RBAC can address all the identified limitations of AWS IAM service.

1 ROLE-BASED ACCESS CONTROL

RBAC has been a widely accepted as an alternative to traditional mandatory access control (MAC) and discretionary access control (DAC) [18], and is an enabling technology for managing and enforcing security in large-scale and enterprise-wide systems. Among many RBAC models proposed in the past decades, NIST has defined a standard RBAC model, which has been widely accepted and implemented. The base model presents the central notion that permissions are associated with roles, users are assigned to appropriate roles, and users acquire permissions by being members of roles. Users can be easily reassigned to from one role to another as needed. Similarly, roles can have new permissions granted and existing permissions revoked as an organization acquires new applications and systems. Users establish sessions during which they may activate a subset of the roles they belong to. Each session maps one user to possibly many roles and each user can establish zero or multiple sessions. NIST RBAC also introduces role hierarchies which are natural means for structuring roles to reflect an organization's lines of authority and responsibility. Senior roles explicitly inherit permissions of their junior roles. Constraints are defined for laying out higher level organizational policies such as SoD.

2 CHALLENGES OF SUPPORTING RBAC FOR AWS

In order to provide RBAC as a service for AWS cloud platform, there are several critical challenges:

Challenge 1: Efficient role hierarchy management. In cloud computing environments, due to dynamic business needs and scalable resource provisioning, the number of roles in an organization could be very large and fluctuated frequently. Accordingly, role hierarchies in the organization could be complex and need to be updated. It is crucial to have an efficient way to manage role hierarchies in terms of both maintenance and update/change management.

Challenge 2: Session management. Session management should be supported to track users' interactions and meet the least privilege principle. Users should be able to activate or deactivate their roles for performing certain tasks in their sessions. With the nature of highly distributed service-oriented computing infrastructure, ACaaS$_{RBAC}$ has to seamlessly support session management without compromising the security property of RBAC.

Challenge 3: SoD support and management of privileged account. SoD constraints should be specified by administrators and each cloud should be able to enforce those constraints for avoiding permission abuse and unexpected fraud. The super user of an AWS account has all privileges upon cloud resources, which should be split into different units or roles in ACaaS$_{RBAC}$.

Challenge 4: System integration and minimal overhead. To leverage the pluggable capabilities provided by ACaaS, ACaaS$_{RBAC}$ services should be easily integrated with customers' applications. Besides, this integration should introduce acceptable performance and network traffic overheads between customers' applications and AWS cloud platform.

Our design of ACaaS$_{RBAC}$ addresses all these challenges with a service-oriented RBAC for AWS cloud resources, which is detailedly explained in next section. In the rest of this section we define a formal model for ACaaS$_{RBAC}$.

3 ACaaS$_{RBAC}$ MODEL FOR AWS

Beyond the existing RBAC96 [17] and NIST RBAC models [8], ACaaS$_{RBAC}$ integrates the group concept of IAM into RBAC. Furthermore, ACaaS$_{RBAC}$ introduces the concept of administrative scope towards a very comprehensive administration model. In this section, we formally define the core model and administration model, while postpone their security assessment and safety analysis [13] to our future work.

3.1 CORE MODEL

Similar to NIST RBAC standard, our core ACaaS$_{RBAC}$ distinguishes regular users and administrative users. Beyond this, our model introduces the concept of group so that we can manage user sets and relatively static permission sets to capture the user group in AWS IAM. Group can be a good counterpart to role for better managing user sets and relatively static permission sets. Correspondingly, our administrative model supports managing user groups.

Definition 1. [**Core Model**] We define the components of ACaaS$_{RBAC}$ core model as follows:

- U, AU, G, R, AR, P, AP, S are sets of regular users, administrative users, groups, roles, administrative roles, user permissions, administrative permissions, and sessions, respectively.
- UR $\subseteq$ U $\times$ R is many-to-many regular user to role assignment relation.
- PR $\subseteq$ P $\times$ R is many-to-many user permission to role assignment relation.
- RH $\subseteq$ R $\times$ R is a partially ordered role hierarchy. $r_i \geq r_j$ means a role r_i is senior to a role r_j and inherits the permissions of r_j.
- UG $\subseteq$ U $\times$ G is many-to-many regular user to group assignment relation.
- PG $\subseteq$ P $\times$ G is many-to-many user permission to group assignment relation. Users explicitly own permissions assigned to groups they belong to and those permissions are activated by default.
- AUAR $\subseteq$ AU $\times$ AR is many-to-many administrative user to administrative role assignment relation.
- APAR $\subseteq$ AP $\times$ AR is many-to-many administrative permission to administrative role assignment relation.
- Sessions: $U \rightarrow 2^S$ is a function that maps a regular user to a set of sessions.
- Roles: $S \rightarrow 2^R$ is a function that maps a session to a set of roles.
- Permissions: $S \rightarrow 2^P$ is a function that derived from PR $\cup$ PG mapping each session to a set of user permissions.

3.2 ROLE-BASED ADMINISTRATION MODEL

Based on the components in the core model, we define a role-based administration model for ACaaS$_{RBAC}$, which consists of four components: *administrative resources*, *administrative actions*, *administrative scopes*, and *administrative policies*.

Definition 2. [**Administrative Resource**] The set of administrative resources RES$_A$ $\subseteq$ U $\cup$ G $\cup$ P $\cup$ R.

Definition 3. [**Administrative Action**] The set of administrative actions ACT$_A$ = {Create, Delete, Assign, Revoke}, where action *Create* and *Delete* correspond to administrative resources including U, R, G and P, and action *Assign* and *Revoke* correspond to relations among UR, UG, PR, and RH.

An administrative action can be performed on administrative resources by an administrative user. For instances, $Create(u)$ creates a regular user $u \in U$, and $Assign(u, r)$ assigns a regular user $u \in U$ to a role $r \in R$, which further introduces a regular user to role assignment relation, i.e., $(u,\ r) \in UR$.

Extended from existing *Range* notion in the ARBAC97 model [16], which is only defined on roles

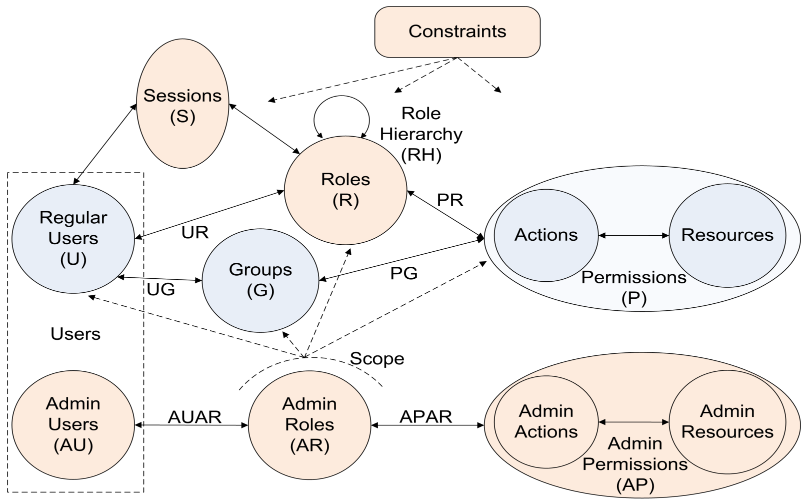


Figure 4: ACaaS$_{RBAC}$ for AWS.

to control user-role assignment, we introduce the concept of *administrative scope* to accommodate the user and group concepts in current AWS IAM policy. Specifically, an administrative scope is a property of an administrative role, which can be specified on particular users, groups, permissions, and roles. An administrative user can perform administrative actions on administrative resources within the scopes associated with her administrative roles.

Definition 4. [**Administrative Scope**] SCO $\subseteq 2^U \times 2^G \times 2^P \times 2^R$ is the set of administrative scopes SCO in ACaaS$_{RBAC}$.

- Get_U: SCO $\rightarrow 2^U$ is a function that maps an administrative scope to a set of regular users specified in that scope.
- Get_G: SCO $\rightarrow 2^G$ is a function that maps an administrative scope to a set of groups specified in that scope.
- Get_P: SCO $\rightarrow 2^P$ is a function that maps an administrative scope to a set of permissions specified in that scope.
- Get_R: SCO $\rightarrow 2^R$ is a function that maps an administrative scope to a set of roles specified in that scope.
- Get_RES$_A$: SCO $\rightarrow 2^{RES_A}$ is a function that maps an administrative scope to a set of administrative resource; Get_RES$_A$(sco) = Get_U(sco) $\cup$ Get_G(sco) $\cup$ Get_P(sco) $\cup$ Get_R(sco).
- Scopes: AR $\rightarrow 2^{SCO}$ is a function that maps an administrative role to a set of administrative scopes.
- Scope_U: AR $\rightarrow 2^U$ is a function that maps an administrative role to a set of regular users; Scope_U(ar) = $\bigcup_{sco \in Scopes(ar)}$ Get_U(sco).
- Scope_G: AR $\rightarrow 2^G$ is a function that maps an administrative role to a set of groups; Scope_G(ar) = $\bigcup_{sco \in Scopes(ar)}$ Get_G(sco).
- Scope_P: AR $\rightarrow 2^P$ is a function that maps an administrative role to a set of user permissions; Scope_P(ar) = $\bigcup_{sco \in Scopes(ar)}$ Get_P(sco).
- Scope_R: AR $\rightarrow 2^R$ is a function that maps an administrative role to a set of roles; Scope_R(ar) = $\bigcup_{sco \in Scopes(ar)}$ {r | ∀r‘ ∈ Get_R(sco), r‘ ≥ r}.

Table 1 shows administrative actions of an administrative user of a role $ar \in AR$, and its pre- and post-conditions. To authorize administrative actions, we introduce a formal definition of administrative policies as follows:

Definition 5. [**Administrative Policy**] An administrative policy is a 4-tuple p_a = <ar, act, res_a, sco>, where ar ∈ AR, act ∈ ACT$_A$, res_a ⊆ Get_RES$_A$(sco), and sco ∈ Scopes(ar), which authorizes an administrative user with the administrative role *ar* to perform the administrative action *act* on a set of administrative resources res_a within the administrative scope *sco* corresponding to *ar*.

IV SYSTEM DESIGN

1 OVERVIEW

Figure 5 shows the system architecture of ACaaS$_{RBAC}$. In current AWS platform, enterprise

Table 1: Administrative Actions

Action	Preconditions	Postconditions
Create(u)	N/A	u ∈ U, u ∈ Scope_U(ar)
Delete(u)	u ∈ Scope_U(ar)	u ∉ U, u ∉ Scope_U(ar)
Create(g)	N/A	g ∈ G, g ∈ Scope_G(ar)
Delete(g)	g ∈ Scope_G(ar)	g ∉ G, g ∉ Scope_G(ar)
Create(p)	N/A	p ∈ P, p ∈ Scope_P(ar)
Delete(p)	p ∈ Scope_P(ar)	p ∉ P, p ∉ Scope_P(ar)
Create(r)	N/A	r ∈ R, r ∈ Scope_R(ar)
Delete(r)	r ∈ Scope_R(ar)	r ∉ R, r ∉ Scope_R(ar)
Assign(r_i, r_j)	r_i, r_j ∈ Scope_R(ar)	(r_i, r_j) ∈ RH
Revoke(r_i, r_j)	r_i, r_j ∈ Scope_R(ar)	(r_i, r_j) ∉ RH
Assign(u, g)	u ∈ Scope_U(ar), g ∈ Scope_G(ar)	(u, g) ∈ UG
Revoke(u, g)	u ∈ Scope_U(ar), g ∈ Scope_G(ar)	(u, g) ∉ UG
Assign(u, r)	u ∈ Scope_U(ar), r ∈ Scope_R(ar)	(u, r) ∈ UR
Revoke(u, r)	u ∈ Scope_U(ar), r ∈ Scope_R(ar)	(u, r) ∉ UR
Assign(p, r)	p ∈ Scope_P(ar), r ∈ Scope_R(ar)	(p, r) ∈ PR
Revoke(p, r)	p ∈ Scope_P(ar), r ∈ Scope_R(ar)	(p, r) ∉ PR
Assign(p, g)	p ∈ Scope_P(ar), g ∈ Scope_G(ar)	(p, g) ∈ PG
Revoke(p, g)	p ∈ Scope_P(ar), g ∈ Scope_G(ar)	(p, g) ∉ PG

applications are able to access cloud resources on behalf of enterprise users, where the AWS IAM enforces security policies defined by enterprise administrators. ACaaS$_{RBAC}$ introduces *RBAC as a service (RaaS)*, which is an RBAC module designed based on NIST RBAC standard [8] and can be hosted by AWS or any third party service provider. This module supports and enforces RBAC configurations by leveraging Amazon IAM service for enterprise administrators. It also provides session capability for enterprise users, e.g., a user or an application can activate and deactivate roles within a single session when accessing resources in AWS [1].

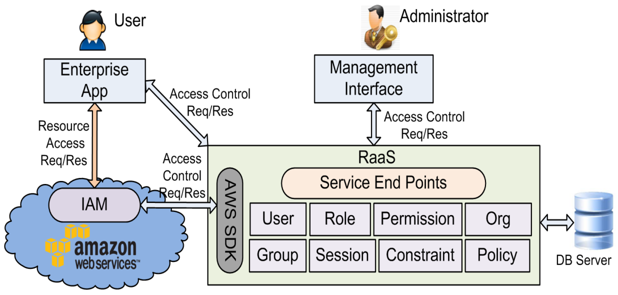


Figure 5: ACaaS$_{RBAC}$ system architecture for AWS.

RaaS provides browser interfaces for enterprise administrators and users to configure RBAC policies. In our prototype (cf. Section V), RBAC policies are implemented as a relational database which stores the relationships according to our ACaaS$_{RBAC}$ model introduced in Section III. RaaS also provides web services APIs such that operations can be integrated into administrative tools or applications from the enterprise side. The results of these configurations are IAM policies that are pushed back to AWS, so that any further access from enterprise applications will be controlled by these policies. Since IAM does not support RBAC, RaaS transforms all role-based policies of a user into AWS permission based policies which can be understandable and enforceable by IAM. The transformation process is to generate direct relationships between users and permissions by removing the role notion between them in the role-based policies. For an enterprise that already has AWS resources in active use, RaaS provisions the information of users, groups, permissions, objects, and actions from AWS via IAM APIs.

According to ACaaS$_{RBAC}$ model, RaaS contains eight sub-modules: *Organization*, *User*, *Group*, *Role*, *Permission*, *Session*, *Constraint* and *Policy*, each of which is exposed as web services. The *Organization* sub-module manages (e.g., list, register, and delete) organizations to support the multi-tenant feature of ACaaS$_{RBAC}$. The *Group* sub-module manages user groups of a single organization for administrative users, where the user information is provi-

[1]We note that an RBAC session here is usually different from that in AWS services, e.g., a DynamoDB session, although they can be co-related in an implementation.

sioned from the organization's AWS account. The *Permission* sub-module manages the permissions of ACaaS$_{RBAC}$. There are two types of permissions: user permissions (P) which are inherited from existing AWS permissions, and administrative permissions (AP), which are effective for RaaS only. The *Permission* sub-module maintains both P and AP, and manages their assignment relations with roles. We elaborate the design of sub-modules in the rest of this section.

2 USER AND PERMISSION

This sub-module provides management on regular users (U), administrative users (AU), and permissions (P), and provides interfaces and APIs to create, delete, activate, and deactivate U, AU, and P, and manage their group memberships and role memberships. In RaaS design, both regular users and permissions are provisioned from AWS directly, with the credentials of the AWS account, which is usually the user who can create other users and policies in IAM. Therefore, RaaS does not store any information for U and P.

Administrative users can be further categorized into two types: root administrative users and regular administrative users. Root administrative users are able to add and delete regular administrative users, and manage their permissions. Regular administrative users are able to perform certain administrative actions (ACT_A) on administrative resources (RES_A) based on their administrative role memberships. By default, a root administrative user is the AWS user that owns the IAM account. With the interfaces provided by RaaS, this user can further create regular administrative users and roles, and their administrative scopes. By controlling the user-role assignments to regular users and administrative users, RaaS can support flexible policies such as splitting privileged accounts, and separation of duty constraints.

3 ROLE

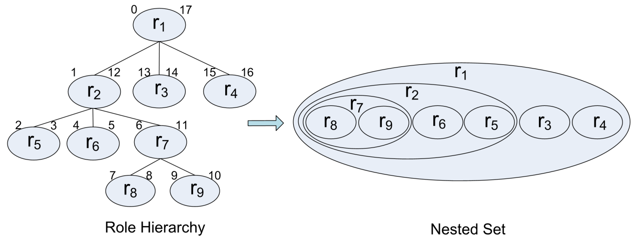
This sub-module creates and deletes roles (R) and administrative roles (AR), and most importantly, it manages role hierarchies (RH). RaaS distinguishes R and AR such that mandatory security policies can be enforced, e.g., by only assigning necessary regular roles to regular users. For least privilege purposes, RaaS may introduce many primitive regular roles, each of which is assigned with atomic permissions. This usually introduces large number of regular roles in a system, where role hierarchy becomes necessary.

Towards efficient role-related operations, we adopt the Nested Set Model [12] for role hierarchy in our implementation, which assigns left and right values to represent a scope of each role in a role hierarchy. If the scope of a role is inside the scope of another role, it means the former role is junior to the latter role. Figure 6(a) shows an example of role hierarchy, where left and right values are assigned to each role in the role hierarchy as well as the Nested Set Model representation of the role hierarchy. A big advantage of Nested Set Model for managing role hierarchies is that only one entry for each role needs to be maintained in the database. This avoids storing a lot of redundant role relationship information to maintain role hierarchies. It is also easy to update role hierarchies by updating associated roles' left and right values. One limitation of our current design is that, the Nested Set Model only supports to represent limited role hierarchies in a simple tree structure, i.e., one of two role hierarchies based on NIST RBAC standard [8]. As our future work, we would extend our implementation towards more general role hierarchy management.

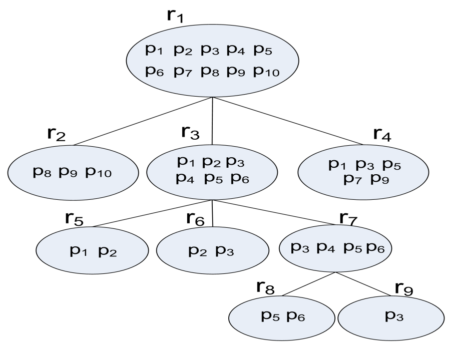
4 SESSION

This module provides session management including role activation and deactivation for regular users. Usually, a user can be assigned with several roles at the same time. In a session, to meet the least privilege principle, only some of those roles which are needed to perform a certain task should be activated. After finishing this task, relevant activated roles can be deactivated. When activating a role, permissions associated with that role should be effective by enforcing corresponding Amazon IAM policies such that users are able to access cloud resources with needed permissions.

An intuitive way for deactivating a role and activating another role is to remove all permissions of the original role for the user in IAM, and then create new permissions and policies, and push all new policies to IAM. This is the approach we adopt if these two roles have no overlapped permissions. With the existence of role hierarchy, it may not be necessary to generate and enforce Amazon IAM policies for all permissions of the new role since some permissions may have already been effective, e.g., these two roles have common inherited permissions from role hierarchy. For an example shown in Figure 6(b), suppose



(a) Role Hierarchy in Nested Set Model.



(b) Role Activation and Deactivation

Figure 6: Role hierarchy and usage in ACaaS$_{RBAC}$ system implementation.

only role r_2 and r_4 are activated and role r_1 needs to be activated. In this case, it is necessary to generate and enforce Amazon IAM policies only for permission p_2, p_4, and p_6, since r_1 has already been effective due to the activated role r_2 and r_4.

To improve the efficiency of role activation and minimize the communication overheads between RaaS module and AWS cloud platform, we implement an efficient role activation algorithm to compute a minimum permission set when activating a role, as shown in Algorithm 1. This algorithm works as follows: if a user u owns an immediate senior and activated role to role r_a which needs to be activated, an empty permission set is returned and no policy needs to be generated and enforced by Amazon IAM. Otherwise, For each sibling role and immediate junior role to role r_a, their senior-most and activated junior roles are identified recursively and a corresponding permission set P_{all} is constructed. Then for each permission associated with the role r_a, if it does not belong to P_{all}, then it will be added to the returned permission set. On the other hand, when deactivating roles, the deactivation should not affect functionalities of other activated roles when they have overlapped permissions. For example, suppose only role r_2 and r_4 in Figure 6(b) are activated and role r_2 needs to be deactivated. To avoid affecting the functionality of role r_4, only permission p_8 and p_{10} should be ineffective in AWS by enforcing corresponding Amazon IAM policies. Permission p_9 remains effective since role r_4 owns permission p_9.

Correspondingly, we implement a role deactivation algorithm, as shown in Algorithm 2. The algorithm works as follows: if any senior role to a role r_d which needs to be deactivated is activated, an empty permission set is returned and no policy needs to be generated and enforced by Amazon IAM. Otherwise if role r_d does not have any activated sibling roles, a permission set containing all permissions associated with the role r_d is returned. Otherwise, a permission set P_{all} containing all permissions associated with activated sibling roles of the role r_d is constructed. Then for each permission associated with the role r_d, if it does not belong to P_{all}, it is added into the returned permission set.

5 CONSTRAINT

This sub-module provides constraints management services including creating, deleting, updating static constraints as well as separation of duty constraints. Static constraints are specified on permissions in existing Amazon IAM constraints format discussed in Section 3 and enforced by IAM. Only when the static constraints are satisfied, users are able to perform corresponding permissions. When creating an SoD constraint, a set of potential conflicting roles and a cardinality value need to be specified. The cardinality value is a threshold of the total role occurrence in the potential conflicting role set. When it is reached, IAM security policy of the user will be updated and any corresponding request will be denied. SoD constraints are enforced by *Constraint* sub-module itself when administrative users assign users to roles, or users activate their roles. Those static constraints are then converted into IAM policies and pushed into AWS.

Dynamic separation of duty (DSoD) constraints are

```
Algorithm 1: ComputeActivatePermissions(u, r_a) → P
Input: A user $u$ wants to activate a role $r_a$
Output: A permission set $P$, of which corresponding IAM
        policies need to be enforced
1  $P \leftarrow \oslash$;
2  $P_{all} \leftarrow \oslash$;
3  $r_{is} \leftarrow$ getImmediateSeniorRole($r_a$);
4  if $hasRole(u, r_{is}) = TRUE\ AND\ active(u, r_{is}) = TRUE$
   then
5      return $\oslash$;
6  else
7      ComputeP(u, $r_a$);
8      foreach $p \in Permissions(r_a)$ do
9          if $p \notin P_{all}$ then
10             add $p$ into $P$;
11         end
12     end
13     return $P$;
14 end
15 ComputeP(User u, Role $r_a$)
16 begin
17     R ← getSiblingRoles($r_a$) $\bigcup$
       getImmediateJuniorRoles($r_a$);
18     if $R = \oslash$ then return;
19     else foreach $r \in R$ do
20         if $active(u, r) = TRUE$ then
21             foreach $p \in Permissions(r)$ do
22                 if $p \notin P_{all}$ then
23                     add $p$ into $P_{all}$;
24                 end
25             end
26         else
27             ComputeP(u, r);
28         end
29     end
30 end
```

```
Algorithm 2: ComputeDeactivatePermissions(u, r_d) →
P
Input: A user $u$ wants to deactivate a role $r_d$
Output: A permission set $P$, of which corresponding IAM
        policies need to be enforced
1  $P \leftarrow \oslash$;
2  $P_{all} \leftarrow \oslash$;
3  $R_{senior} \leftarrow$ getSeniorRoles($r_d$);
4  if $R_{senior} \neq \oslash$ then
5      foreach $r \in R_{senior}$ do
6          if $active(u, r) = TRUE$ then
7              return $\oslash$;
8          end
9      end
10 end
11 $R_{sibling} \leftarrow$ getActivatedSiblingRoles($r_d$);
12 if $R_{sibling} = \oslash$ then
13     return $Permissions(r_d)$;
14 else
15     foreach $r \in R_{sibling}$ do
16         if $active(u, r) = TRUE$ then
17             foreach $p \in Permissions(r)$ do
18                 if $p \notin P_{all}$ then
19                     add $p$ into $P_{all}$;
20                 end
21             end
22         end
23     end
24     foreach $p \in Permissions(r_d)$ do
25         if $p \notin P_{all}$ then
26             add $p$ into $P$;
27         end
28     end
29     return $P$;
30 end
```

usually enforced during runtime, e.g., conflicting roles cannot be activated in a single session. Similar constraints can be defined and checked by the session module.

6 POLICY

This sub-module provides Amazon IAM policy generation and pushing services to ensure RBAC configurations of an enterprise can be reflected in AWS cloud platform. For example, when a user activates or deactivates roles, corresponding Amazon IAM policies are generated and pushed to the Amazon IAM policy engine for the enforcement. More specifically, for each permission in the permission set computed by Algorithm 1 or Algorithm 2, a corresponding IAM policy is constructed and sent to IAM for the enforcement. Policy transformation and deployment are also triggered by other administration actions that change the regular permissions of a user.

We note that our policy transformation is both complete and sound. That is, each state of the RaaS (the ACaaS$_{RBAC}$ relationships stored in its local database) can be translated to a set of IAM policies, e.g., each regular user has an IAM policy. This is due to the fact that both the users and permissions are provisioned from AWS directly. For each RaaS state, the net result of its configuration is a set of permissions that are authorized for a user, after revolving the constraints such as SoD and DSoD. Similarly, each translation corresponds to a valid IAM policy since the permissions are defined with valid resource names and actions defined by AWS.

7 CONCURRENCY CONTROL

As shown in Figure 5, for performance and compatibility considerations, a AWS user or application can access AWS resources by directly calling AWS interfaces or APIs, while RaaS pushes IAM policies with another channel. Therefore, there are chances that a user's session of using AWS resources is ongoing, while any of required permissions is revoked, i.e., due to some administrative operations in RaaS by an administrator. This concurrency control issue has been discussed in [26] in context of XACML, where a lock manager is proposed to revoke active permission or delay the administrative operation. In our design, we take the conservative approach of delaying the revoking operation, since our RaaS cannot actively revoke an ongoing access session in AWS, due to the fact that it runs out of AWS platform.

V IMPLEMENTATION & EVALUATION

Based on our design, we have implemented a prototype system to provide RBAC services in AWS cloud platform through a web browser interface as well as web services. The core services of the system are implemented in Java based on AWS SDK 1.3.0 and exposed as SOAP-based web services using GlassFish Metro 2.2. A web-based management interface is developed by using JavaServer Pages (JSP) and MySQL Community Server 5.1. Both administrative users and normal users can log into the interface with their usernames and passwords. Administration tools can interact with the system by calling the SOAP-based web services APIs where the body of messages are signed with pre-shared secret keys.

All entities of the major components in ACaaS$_{RBAC}$ model (cf. Section 3) are stored in tables of a relational database, which jointly represents the state of the RaaS system. The name spaces of permissions, which are built on resources in AWS, are provisioned through AWS APIs. An administrative operation results in calling one or more AWS APIs, e.g., to create a user, a group, a permission, or add or remove an IAM policy in the root user's AWS account.

We evaluated our ACaaS$_{RBAC}$ system in terms of two metrics: *efficiency* and *scalability*. Specifically, a scenario-based case study is presented to illustrate how services provided by ACaaS$_{RBAC}$ can be utilized by existing cloud applications. We then show the performance overheads and network traffic overheads measured on role activation and deactivation services.

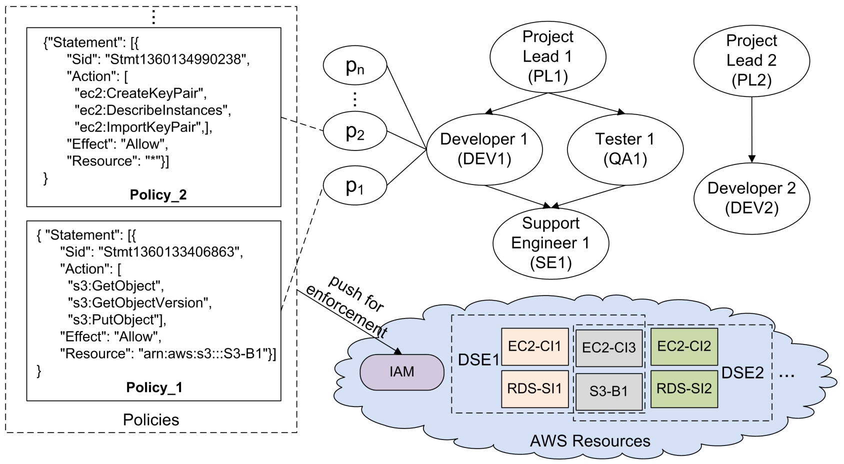
1 SCENARIO DESCRIPTION

IT sandboxes are isolated computing environments which can be used for software development and test. Consider a company called *VirtualSoft* which develops software systems using IT sandboxes and adopts RBAC for managing the access to its sandboxes' resources. Due to increasing customers' needs and cost efficiency, *VirtualSoft* decides to migrate its sandbox environments from its on-premise data center to AWS cloud platform. However, AWS cloud platform does not support RBAC for its authorization mechanism. To bridge this gap, *VirtualSoft* leverages services provided by ACaaS$_{RBAC}$ system for RBAC enforcement on AWS cloud platform. Consider two ongoing collaborative projects – Project 1 and Project 2, using two development sandbox environments – DSE1 and DSE2, respectively. These two sandboxes are built on two Amazon EC2 M1 large instances CI1 and CI2 for computing and two Amazon RDS instances SI1 and SI2 for database services, respectively. They share another Amazon M1 medium instance CI3 which runs Perforce for source code version control, and an Amazon S3 bucket B1 for storing collaborative documents and slides. The role hierarchies of both projects are shown in Figure 7: Project 1 has four roles of `PL1`, `DEV1`, `QA1`, and `SE1`, and Project 2 has two roles of `PL2` and `DEV2`. Each role is associated with certain permissions for accessing various sandbox resources built on AWS cloud. For instance, developers working on Project 1 are assigned to the role `Dev1`, which enables them to access any resource under DSE1 consisting of CI1, SI1, CI3, and B1. Similarly, Project 2 developers who are assigned to the role `Dev2` are able to access any resource under DSE2 consisting of CI2, SI2, CI3, and B1. Each permission is further corresponding to an IAM policy generated by the ACaaS$_{RBAC}$ *Policy* module (cf. Section 6). Taking an example shown in Figure 7, two IAM policies `Policy_1` and `Policy_2` are respectively in correspondence with two permissions `p1` and `p2` of the role `Dev1`. `Policy_1` is to enable the access to the Amazon S3 bucket `B1`, and `Policy_2` is to enable the access the Amazon EC2 instance `CI1`. Those two policies are pushed to Amazon IAM service for enforcement when a user activates the role `Dev1`.

2 EXPERIMENT ENVIRONMENT AND RESULTS

In our experiment, ACaaS$_{RBAC}$ is hosted in a desktop machine with Intel Core2 Duo CPU 3.00 GHz 4 GB RAM running Ubuntu 11.10. Assume Alice is a member of the role `Dev1` working on Project 1 and Bob is a member of the role `Dev2` working on Project 2. In order to accomplish a collaborative task across projects, Alice needs to be assigned to the role `Dev2` to access the resources of DSE2. We measure the `Dev2` role activation for Bob when he logs in the sandbox system and the `Dev2` role activation for Alice when she is assigned to the role `Dev2` in terms of performance overhead (T) and network traffic overhead (NT).

Table 2 shows corresponding average results based on 10 measurements. Performance overhead can be



```
{"Statement": [{
     "Sid": "Stmt1360134990238",
     "Action": [
       "ec2:CreateKeyPair",
       "ec2:DescribeInstances",
       "ec2:ImportKeyPair",],
     "Effect": "Allow",
     "Resource": "*"}]
}
```

```
{ "Statement": [{
     "Sid": "Stmt1360133406863",
     "Action": [
       "s3:GetObject",
       "s3:GetObjectVersion",
       "s3:PutObject"],
     "Effect": "Allow",
     "Resource": "arn:aws:s3:::S3-B1"}]
}
```

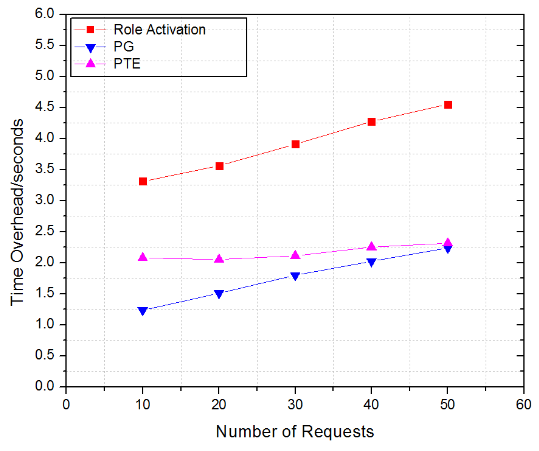
Figure 7: A Sandbox scenario for software test in AWS with ACaaS$_{RBAC}$.

divided into two sub parts: policy generation (PG) time and policy transmission and enforcement (PTE) time. The PTE introduced by AWS platform takes up above 95% of total overhead. Note that the same `Dev2` role is activated for Alice and Bob. However, both performance overhead and network traffic overhead for Alice are much less than that for Bob. This is because the role activation service efficiently identifies the permission overlap between the role `Dev1` and `Dev2` – both have the permissions to access *CI3* and *B1* – to minimize the communication cost between ACaaS$_{RBAC}$ and Amazon IAM as discussed in Section IV.

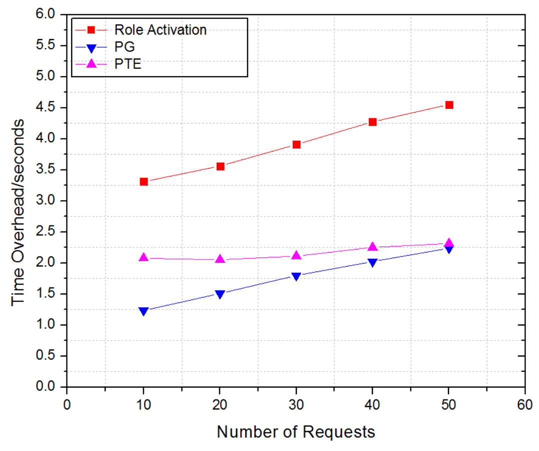
To evaluate the scalability of ACaaS$_{RBAC}$, we create multiple threads and measure average performance overhead and total network traffic overhead while increasing the numbers of simultaneous role activation and deactivation requests from different users. Some of the users are the members of role `Dev1` and some are members of role `Dev2`. Figures 8(a) and 8(b) respectively show the average performance overhead per request including PG overhead and PTE overhead, as the number of role activation and deactivation requests increases from 10 to 50. We observe that the average performance overhead for a role activation request increases smoothly as the number of simultaneous requests is increased. The PG overhead takes up a smaller portion of a role activation request overhead than the PTE overhead does, until the number of role activation requests reaches to 50. The PTE overhead contributes the most portion of the role deactivation request overhead. Figure 8(c) shows the total network traffic overhead as the number of role activation and deactivation requests increases from 10 to 50. The network traffic overhead for role activation requests is slightly larger than that of role deactivation requests as the number of requests increases. When the number of role activation requests is 10, the total network traffic overhead is around 150 kb. When the number of role activation requests increases to 50, the total network traffic overhead is just above 700 kb, which we believe is still manageable. To further enhance the efficiency and scalability of our implementation, our role activation service is extended to support activating multiple roles as a batch operation. Figure 8(d) shows the average performance overhead on activating one, two, and three roles per request while increasing the number of role activation requests simultaneously from 10 to 50. The average performance overhead on activating two roles as a batch operation per request is lower than two times of that on activating one role per request at the same number of requests. Similar findings can be observed when comparing the average performance overhead of three roles batch activation requests with that of one role.

VI RELATED WORK

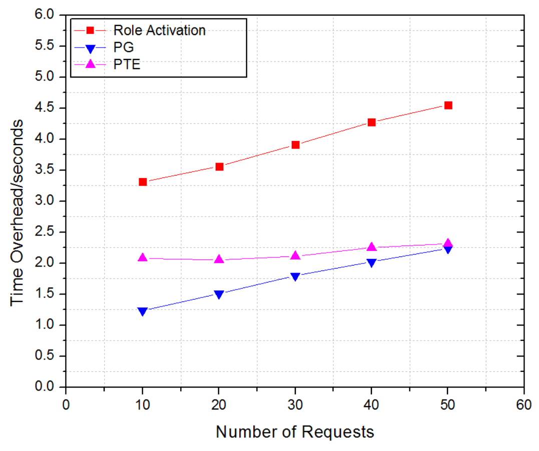
There are several authorization and access control solutions in cloud computing. Calero *et al.* [4] propose an authorization model that supports multi-tenancy, role-based access control, path-based object hierarchies, and federation. Hu *et al.* [11] introduce semantic web technologies to distributed role-based access control method and propose a new Semantic Access Control Policy Language (SACPL) for describing access control policies in cloud computing environments. Access Control Oriented Ontology System (ACOOS) is designed as the semantic basis of



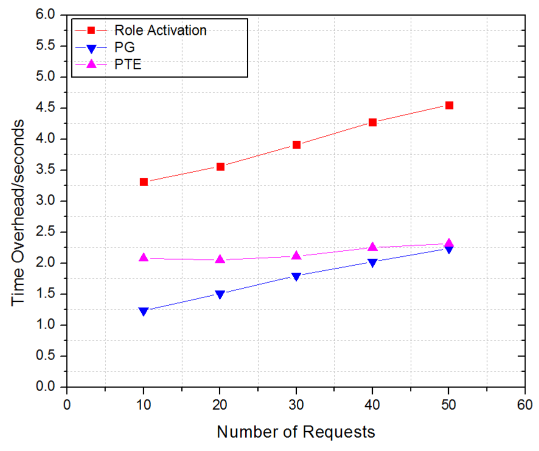
(a) Activation Time



(b) Deactivation Time



(c) Network Traffic



(d) Batch Activation Time

Figure 8: Overheads of activating and deactivating roles.

SACPL. In [19], Santos *et al.* propose the design of a trusted cloud computing platform (TCCP). TCCP provides the abstraction of a closed-box execution environment for a customer's VM, guaranteeing that no cloud provider's privileged administrator can inspect or tamper with its content. However, this approach requires the correctness of large and complex codebases, such as hypervisor, device drivers, or entire kernels. Cloud computing may face a scenario that an attacker shares the same physical resources with other tenants. Sharing resources could lead to information leakage due to known or unknown covert channels. Zhu *et al.* [28] design CloudPolice, a distributed access control mechanism implemented in hypervisors, to meet the access control needs of multi-tenancy, network-independence, and scalability in cloud computing environment. Their access control solution is designed from network perspective, which may fail to prevent diverse unauthorized accesses in applications. We note that these approaches attempt to address specific issues in access control, but do not consider a comprehensive approach that supports various policy models and can be embodied as service modules in clouds.

Another line of research work deal with access control issues by leveraging cryptographic techniques in cloud computing environments. Echeverria *et al.* [6] presented an effective solution for de-coupling access control from services that provide content by

Table 2: Access Control Overheads

	T(ms)	PG(ms)	PTE(ms)	NT(kb)
Dev2 Activation for Bob	1898.9	91.6	1807.2	15.95
Dev2 Activation for Alice	1200.2	47.4	1152.4	14.652

leveraging attribute based encryption (ABE). Yu *et al.* [27] combine attribute-based encryption, proxy re-encryption, and lazy re-encryption to delegate most of the computation tasks involved in user revocation to untrusted cloud service providers. Tassanaviboon and Gong [22] propose a new authorization scheme that combats untrusted cloud servers by adopting CP-ABE, ElGamal like masks, proxy re-encryption, and lazy re-encryption to achieve user-centric and end-to-end security. Wan *et al.* [23] introduce the HASBE scheme which incorporates a hierarchical structure of system users by applying a delegation algorithm to ASBE. CloudSeal [25] uses proxy re-encryption and broadcast revocation algorithms for flexible content access control in cloud. Even though cryptographic approach is effective for some specific requirements, they have no support for legacy security policies in enterprises such as role-based access control.

There are also industrial efforts on building RBAC support in cloud computing platforms. The latest AWS IAM enables EC2 instances to run with predefined IAM roles to securely access AWS service APIs [5]. However, it only allows assigning IAM roles to EC2 instances not to users. Hence, all applications running in an EC2 instance assume the same permission set of the IAM role. XenServer [24] only provides 6 pre-established roles with capabilities to modify permissions on them but no functionality to add or delete roles. OpenStack [14] realizes the role notion by using user assigned tokens. Eucalyptus [7] integrates existing Microsoft Active Directory or LDAP systems to capture the role notion for managing the access over cloud resources. In our analysis, all of above solutions fail to accommodate core RBAC functions such as role hierarchy, session management, and role-based administration and delegation; therefore, none of them provides a comprehensive built-in RBAC model.

VII CONCLUSION AND FUTURE WORK

We articulate the critical need of a comprehensive and fine-grained access control mechanism to meet dynamic, configurable, and extensible security requirements in public IaaS cloud computing environments. To accommodate this need, we propose a new modular architecture towards *access control as a service* (ACaaS) for supporting multiple access control models. As a reference implementation, we design and implement ACaaS$_{RBAC}$, a service architecture that supports configurations of RBAC as a service for Amazon Web Services. Our case study and system performance evaluation demonstrate the practicality and efficiency of our approach. For future work, we would evaluate our system with real-world datasets. In addition, we would enhance our system with flexible delegation mechanisms and accommodate revocation requirements, and design a more generic architecture to support other access control policies such as multilevel and general mandatory access control policies.

ACKNOWLEDGEMENT

The work of Ruoyu Wu, Gail-Joon Ahn and Hadi Sharifi was partially supported by the grants from National Science Foundation and Department of Energy. The work of Haiyong Xie was supported in part by NSFC Grant No. 61073192, by 973 Program Grant No. 2011CB302905, by NCET Program Grant No. NCET-09-0921, and by USTC Grant No. WK0110000014.

References

[1] Exploring the Cloud: A Global Study of Governments' Adoption of Cloud. Technical report, KPMG International Cooperative, 2012.

[2] AWS Identity and Access Management (IAM). http://docs.amazonwebservices.com/IAM/latest/UserGuide.

[3] D. Brewer and M. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214. IEEE, 1989.

[4] J. Calero, N. Edwards, J. Kirschnick, L. Wilcock, and M. Wray. Toward a multi-tenancy authorization system for cloud services. *Security & Privacy, IEEE*, 8(6):48–55, 2010.

[5] AWS Document, Working with Roles. http://docs.aws.amazon.com/IAM/latest/UserGuide/WorkingWithRoles.html.

[6] V. Echeverria, L. Liebrock, and D. Shin. Permission management system: Permission as a service in cloud computing. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, pages 371–375. IEEE, 2010.

[7] Cloud Computing Software from Eucalyptus — Leader in Cloud Software, 2012. http://www.eucalyptus.com/.

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

[9] Federal Information Security Management Act (FISMA). http://csrc.nist.gov/drivers/documents/FISMA-final.pdf.

[10] U.S. Department of Health and Human Services. The Health Insurance Portability and Accountability Act (HIPAA), 2011. http://www.hhs.gov/ocr/privacy/.

[11] L. Hu, S. Ying, X. Jia, and K. Zhao. Towards an approach of semantic access control for cloud computing. *Cloud Computing*, pages 145–156, 2009.

[12] M. J. Kamfonas. Recursive hierarchies. *The Relational Journal*, 1992.

[13] N. Li and M. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 9(4):391–420, 2006.

[14] OpenStack Compute Administration, 2012. http://docs.openstack.org/essex/openstack-compute/admin/content/.

[15] R. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[16] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

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

[18] R. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[19] N. Santos, K. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, page 3. USENIX Association, 2009.

[20] The Sarbanes-Oxley Act of 2002(SOX). http://sec.gov/about/laws/soa2002.pdf.

[21] J. Staten and L. E. Nelson. Market Overview: Private Cloud Solutions, Q2 2011. Technical report, Forrester Research, Inc, 2011.

[22] A. Tassanaviboon and G. Gong. Oauth and abe based authorization in semi-trusted cloud computing: aauth. In *Proceedings of the second international workshop on Data intensive computing in the clouds*, pages 41–50. ACM, 2011.

[23] Z. Wan, J. Liu, and R. Deng. Hasbe: A hierarchical attribute-based solution for flexible and scalable access control in cloud computing. *Information Forensics and Security, IEEE Transactions on*, (99):1–1, 2011.

[24] Available Role Based Access Control Permissions for XenServer, 2012. http://support.citrix.com/article/CTX126441.

[25] H. Xiong, X. Zhang, D. Yao, X. Wu, and Y. Wen. End-to-end content protection in cloud-based storage and delivery services. In *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[26] M. Xu, D. Wijesekera, and X. Zhang. Runtime administration of rbac profile for xacml. *IEEE Transactions on Services Computing (TSC)*, (4):286–299, 2011.

[27] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[28] Y. Zhu, H. Hu, G. Ahn, D. Huang, and S. Wang. Towards temporal access control in cloud computing. In *INFOCOM, IEEE*, 2012.