ACaaS: Access Control as a Service for IaaS Cloud

Ruoyu Wu*, Xinwen Zhang†, Gail-Joon Ahn*, Hadi Sharifi* and Haiyong Xie†‡
*Arizona State University, Tempe, AZ 85287, USA
Email: {ruoyu.wu, gahn, hsharif1}@asu.edu
†Huawei Research Center, Santa Clara, CA 95050, USA
Email: {xinwen.zhang, haiyong.xie}@huawei.com
‡University of Science and Technology of China, Hefei, China

***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.**

***Keywords*—security, access control, cloud computing**

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 and most importantly, 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 or commercial 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 as cloud platform) 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 (Section II, III);

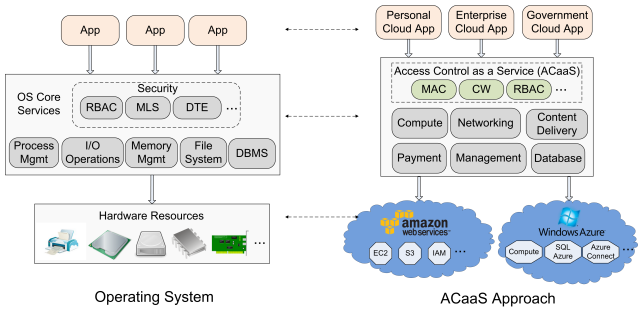

Fig. 1: ACaaS vs. security modules in operating system.

- We implement a prototype system, and provide web-based an administrative tool and web services APIs for third party applications' integration (Section IV).

II. ACaaS for Clouds

A. 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) [11], the Chinese Wall security policy (CW) [3], and role based access control [12]. This plug & play fashion enables parallel evolution of cloud customer's own policy specifications and a cloud provider's security enforcement mechanisms.

B. 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 - a service called Identity and Access Management

```
{"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/*"}]
}
```

Fig. 2: Example AWS IAM policy.

(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 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*.

Based on the above-mentioned IAM 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 2. 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 [1]. 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. Design of ACaaS$_{RBAC}$ for AWS

In this section, we present the design of ACaaS$_{RBAC}$, a reference architecture of ACaaS that supports RBAC for Amazon AWS cloud platform. 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 [13], 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.

A. 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.

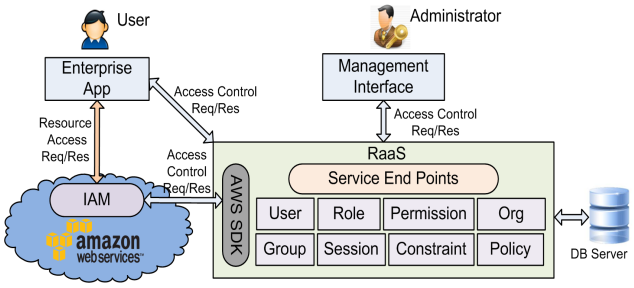

Fig. 3: ACaaS$_{RBAC}$ system architecture for AWS.

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.

B. System Design

Our design of ACaaS$_{RBAC}$ addresses all identified challenges III-A with a service-oriented RBAC for AWS cloud resources. Figure 3 shows the system architecture of ACaaS$_{RBAC}$. In current AWS platform, enterprise 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 [7] 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].

[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.

RaaS provides browser interfaces for enterprise administrators and users to configure RBAC policies. In our prototype (cf. Section IV), RBAC policies are implemented as relational database entries. 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.

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 provisioned 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.

1) 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.

2) 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.

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, P_all ← ∅;
2  r_is ← getImmediateSeniorRole(r_a);
3  if hasRole(u, r_is) = TRUE AND active(u, r_is) = TRUE then
4      return ∅;
5  else
6      ComputeP(u, r_a);
7      foreach p ∈ Permissions(r_a) do
8          if p ∉ P_all then
9              add p into P;
10         end
11     end
12     return P;
13 end
14 ComputeP(User u, Role r_a)
15 begin
16     R ← getSiblingRoles(r_a) ⋃ getImmediateJuniorRoles(r_a);
17     if R = ∅ then return;
18     else foreach r ∈ R do
19         if active(u, r) = TRUE then
20             foreach p ∈ Permissions(r) do
21                 if p ∉ P_all then
22                     add p into P_all;
23                 end
24             end
25         else
26             ComputeP(u, r);
27         end
28     end
29 end
```

Towards efficient role-related operations, we adopt the Nested Set Model [9] 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. 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 [7]. As our future work, we would extend our implementation towards more general role hierarchy management.

3) 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.

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.

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.

4) 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 II-B 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 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.

5) 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.

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
P, P_all ← ∅;
R_senior ← getSeniorRoles(r_d);
if R_senior ≠ ∅ then
    foreach r ∈ R_senior do
        if active(u, r) = TRUE then
            return ∅;
        end
    end
end
R_sibling ← getActivatedSiblingRoles(r_d);
if R_sibling = ∅ then
    return Permissions(r_d);
else
    foreach r ∈ R_sibling do
        if active(u, r) = TRUE then
            foreach p ∈ Permissions(r) do
                if p ∉ P_all then
                    add p into P_all;
                end
            end
        end
    end
    foreach p ∈ Permissions(r_d) do
        if p ∉ P_all then
            add p into P;
        end
    end
    return P;
end
```

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.

IV. Implementation

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$_{RBAC}$ 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.

In our future work, we will conduct case studies to illustrate how services provided by ACaaS$_{RBAC}$ can be utilized by existing cloud applications, and evaluated our ACaaS$_{RBAC}$ system in terms metrics such as *efficiency* and *scalability*.

V. 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.* [8] 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. 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.* [5] presented an effective solution for de-coupling access control from services that provide content by leveraging attribute based encryption (ABE). CloudSeal [15] 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 [1]. 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 [14] only provides 6 pre-established roles with capabilities to modify permissions on them but no functionality to add or delete roles. OpenStack [10] realizes the role notion by using user assigned tokens. Eucalyptus [6] 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.

VI. 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. 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 multi-level 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] Aws document, working with roles, http://docs.amazonwebservices.com/IAM/latest/UserGuide/WorkingWithRoles.html.

[2] Aws identity and access management, http://docs.amazonwebservices.com/IAM/latest/UserGuide/Welcome.html.

[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] 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.

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

[7] 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.

[8] 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.

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

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

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

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

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

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

[15] 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.