OpenStack Security Modules: a Least-Invasive Access Control Framework for the Cloud

Yang Luo*, Wu Luo*, Tian Puyang*, Qingni Shen*, Anbang Ruan†, Zhonghai Wu*‡
*Peking University, China
Email: {luoyang, lwyeluo, puyangsky, qingnishen, wuzh}@pku.edu.cn
†University of Oxford, United Kingdom
Email: anbang.ruan@cs.ox.ac.uk
‡Corresponding author

***Abstract*—The access control mechanisms of existing cloud systems, mainly OpenStack, fail to provide two key factors: i) centralized access mediation and ii) flexible policy customization. This situation prevents cloud administrators and end customers from enhancing their security. Furthermore, a variety of clouds have implemented their access control systems and policies in separated ways. This might confuse the customers whose businesses are built on multiple clouds, as they have to take efforts to accommodate their policies for different platforms. The OpenStack Security Modules (OSM) project has developed a least-invasive access control framework for OpenStack to enable different access control models to be implemented as loadable modules. This framework can be a good replacement of the existing permission checks in OpenStack and other platforms. We also propose an integration mechanism for multiple policies to form a single decision. This paper presents the design and implementation of OSM, including a new service called patron and an attachment module called access endpoint middleware (AEM). Experiments on the tempest benchmark indicate that OSM has improved the flexibility and security of policy management without affecting other services. Meantime, the average performance overhead remains as low as 7.3%, which is acceptable for practical use.**

***Keywords*-cloud service; OpenStack; access control; security hook; multi-policy mechanism;**

I. Introduction

Cloud computing has become a revolutionary power for enterprises to reduce their costs by using on-demand computational infrastructures [1], [2], [3]. Although being the most widely adopted cloud in open-source world, OpenStack failed to give due weight to the security, especially access control. At present, the critical role of single sign-on in providing cloud security has been well demonstrated by keystone, a standalone authentication service, yet the access control mechanisms of existing mainstream OpenStack are still inadequate to provide strong security. Locally stored policy is used to enforce access controls for almost all OpenStack services, and the policy is implemented as built-in, which allows no one to modify it except cloud service providers themselves. From the model perspective, attribute-based access control (ABAC) [4], [5] is readily adopted by OpenStack and a part of role-based access control (RBAC) [6] (the RBAC0 model) is supported as well. The cloud provider is able to specify authorization rules based on the properties of subjects and objects. However, these models are hard-coded into the cloud and lack the flexibility to be altered. The recent implementation has several limitations as a method for providing cloud access controls. First, the policy is currently decentralized. All access controls are restricted within the scope of a physical host. A policy supervisor has to spend his time on preparing a policy for each service and manually deploys them onto all nodes of the cloud, the process of which can be quite unfriendly and error-prone. Second, hooking code is a headache for developers. For a general-purpose system, access controls are typically enforced by hooking the internal functions of it. We have found that nova has implemented at least 454 security hooks in its code. This fact indicates that all functional developers of OpenStack have to pay adequate attention to insert proper security hooks into the proper code lines by themselves. However, not all of them are security experts, and they probably make mistakes. For example, in the `show` operation of `security groups`, an identical hook has been planted twice, and in `list` action of `instance type`, the security check is not even positioned. This might be expected sometimes, but there is no guarantee that an access will always get mediated. If the security hook is mistakenly absent from a critical action's execution path, this action can then be performed by an unauthorized user. Additionally, a portion of the permission check is even hard-coded into the platform. If a cloud customer is unwilling to use the default security policy and desires to customize his own, there will be no way to achieve this.

To solve the above-mentioned issues, in this paper, we propose a security framework called OpenStack Security Modules. It includes a stand-alone service called patron, and an attachment module called AEM which is required to be attached to other OpenStack services to filter the requests sent to them (we state it as "attached" not "patched" because it is totally decoupled with other services). With the OSM framework, each representational state transfer (REST) [7] request towards the cloud (including requests sent by not only outside consumers but also the cloud infrastructure itself) is filtered by AEM first. AEM then validates it by sending its security contexts to patron, which will make a proper decision based on policy enforcement. If patron says yes to this access, the request will be permitted by AEM to reach the demanded service. Otherwise AEM will reject it by returning an error. The OSM framework highlights the achievement of decoupling access controls from functionalities of cloud in code level, which facilitates both cloud service providers and tenant administrators to have a global view on their security perimeters: the entire cloud or an individual tenant.

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.15

And the security settings, especially policy configurations can be modified through the unified patron interface. In patron, we design a mechanism to manage multiple policies provided by both cloud providers and consumers. Several calls are provided by patron and AEM as a part of the framework. We have implemented the security framework based on the latest OpenStack Liberty [8]. Despite the fact that OpenStack employs REST as its primary interface, the OSM framework is general enough to be applied to other kind of interfaces, like simple object access protocol (SOAP), etc. Our source code is released on GitHub: *https://github.com/openstack-patron/patron*. The next move would be encouraging the community to adopt our scheme.

The remainder of this paper is organized as follows. Section II describes related work. Section III presents the design. Section IV shows the implementation. Section V describes the evaluations, including testing, performance overhead, and code invasiveness so far. Section VI presents our conclusions.

II. Related Work

In the past few years a strong interest has been dedicated to the investigation of cloud security. A wide variety of industrial and academic efforts have been done to provide a more secure cloud. Amazon Web Services (AWS) [9], currently the leading proprietary cloud, has supported identity-based authorization for cloud customers with its Identity and Access Management (IAM). Basic elements like users, groups and permissions are provided for customers to restrict the access to their own resources. Other clouds like Microsoft Azure [10], Google App Engine [11], etc. have highlighted the access controls by constructing a single sign-on mechanism to support their policies. However, their policies are based on their own systems and lack adequate flexibility for users to customize them unlimitedly.

Access control as a service (ACaaS), proposed in [12] by Wu et al., provided a new cloud service to provide comprehensive and fine-grained access control. It is claimed to support multiple access control models, whereas there is no evidence that this approach applies to the models except RBAC. And this work is highly based on IAM provided by AWS, which makes it difficult to apply for other clouds.

OpenStack access control (OSAC), proposed in [13] by Tang et al., has presented a formalized description for conceptions in keystone, such as domains and projects in addition to roles. It further proposed a domain trust extension for OSAC to facilitate secure cross-domain authorization. This work is orthogonal to ours, since it mainly focuses on the enhancement of keystone. The domain trust decision made by OSAC can be used as a policy condition in patron, which increases the granularity of access controls. So our work can be well integrated together.

The work proposed in [14] by Jin et al., has defined a formal ABAC specification suitable for infrastructure as a service (IaaS) and implemented it in OpenStack. It includes two models: the operational model $IaaS_{op}$ and the administrative model $IaaS_{ad}$, which provide fine-grained access control for tenants. However, This work only focuses on isolated tenants and cross-tenant access control is not supported. Moreover, their model is bundled with ABAC, which lacks the flexibility for cloud users to design a policy which is based on a customized model.

Numerous efforts of providing an access control framework have been done in the operating system field, including Linux Security Modules (LSM) [15], [16], Flask [17], and GFAC [18]. All of those frameworks were implemented in Linux, and LSM has even become a part of mainstream Linux since kernel 2.6. Like these prior projects, patron seeks to provide general support for access controls of the cloud. However, its goal slightly differs from these projects by emphasizing least invasiveness as the most important value, and this design is made possible by unified public interfaces provided by the cloud, which is nearly impossible for a general operating system.

The very idea of OSM originates from the practical and technical concerns of the various OpenStack basic projects, which are required to enforce access controls by their own. As a complex system, OpenStack has necessity to decouple its code by isolating its access controls from other parts and providing them as services just like other core functionalities. The naming of OpenStack Security Modules is inspired by LSM [15], [16] for Linux operating system, as they both provide policy-independent framework of general access controls. However, what makes them different is that LSM focuses on inserting general hooks into Linux kernel to enforce mandatory access control, yet OSM is trying to remove them. Besides academic efforts, OpenStack Security Modules project as well maintains the code repository and is willing to convince the developer community into integrating this work. We want this security framework to be:

- truly decoupled. Other projects are liberated from maintaining a couple of security hooks in their code;
- conceptually simple, minimally invasive, and efficient;
- customizable for consumers and manageable for administrators at the same time.

The core functionality for most of system security frameworks like LSM was access control, which was conventionally implemented by numerous security hooks. Unlike those frameworks, OSM merged all security hooks into one "huge hook" at the entrance of cloud to avoid invasiveness. Actually, the existing keystone has already been such a hook to provide authentication service for other functionalities. As authentication and access controls are always regarded as two equally key factors of security, it is highly logical to let access controls be a standalone part of the cloud as well.

III. Design

This section covers the design details of OSM framework. The overview of the OSM architecture will be described in Section III-A. The multi-policy mechanism will be described in Section III-B.

A. Overview of OSM

The architecture of OSM framework is illustrated in Figure 1. The newly added patron is provided as a policy service. AEM is provided as a request manager placed between customers and cloud services. Access control services are provided by the framework to support policy enforcement of other components, including compute, image, network, etc.

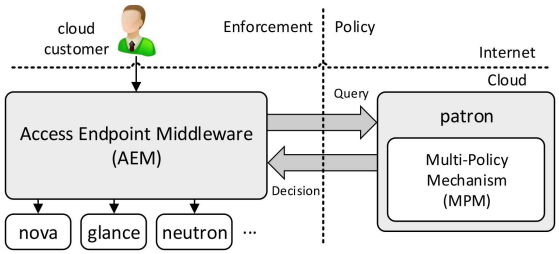

Figure 1. Architecture of the OSM Framework

AEM is an attachment module that performs request filtering and mediating on behalf of the target service. AEM is required to examine all incoming requests to determine the operation and the information of the target going to be operated, then send all these data to patron for a ruling result. Patron determines whether to approve this access based on the policy.

As a service of providing centralized access controls for the whole cloud, the main functions of patron include access verification, policy storage, policy update, etc. As shown in Figure 1, when a user request reaches AEM, AEM sends a `verify` request to patron for permission authorization. The latter will then determine whether this access is okay based on the policy and related security context which is a part of the request. The security context includes three elements:

- `caller context`: the requesting body, usually a cloud user.
- `object context`: the requested resource, such as a virtual machine (VM) instance or a mountable disk.
- `op`: the operation the subject is performing on the object (like start or stop an instance).

These elements are previously used to enforce the original machine-wide policy and we have taken them unchangingly as the parameters of the new `verify` request to remain compatible with original code. So far, the most simplified model would be that patron still employs these elements to enforce the policy in the same manner as like other services do for the local policy. The only difference is that the previous policy rules distributed in multiple services have been centralized into a single node, thus making it fit well for large-scale deployment.

Patron is assumed to be positioned similarly in the overall OpenStack architecture to keystone, as a result of a shared communicating manner. e.g., for a multi-regional cloud, those two services can be both positioned at the boundaries of the regions. In practice, patron can even be physically collocated with keystone to minimize hardware cost.

In OpenStack, any user with an `admin` role is regarded as the cloud administrator by default. This assumption is too coarse-grained as it fails to achieve the differentiation between tenant administrators and cloud administrators. Thus in the OSM framework, we did a slight bit of modification to the original model: let users with `admin` role in `admin` tenant be the cloud administrators, and users with `admin` role in other tenants (not `admin`) be the tenant administrators. Cloud administrators are privileged to control the whole cloud, while tenant administrators can only manage their own tenants. This modification can be done by an additional tenant check on the provided credential.

Despite the fact that the communications between AEM and patron happen in the intranet of the cloud, there is a chance that an insider attacker compromises all the access controls by tampering the network traffic containing patron's decisions. Cryptographic measures can be taken to secure their communicating channel. However, such protection may also impact the efficiency of the system. This is a trade-off required to be made by the cloud provider. In this paper, we assume that proper protection on communication channels is offered.

A crucial premise of the OSM framework is whether AEM can enforce all permission validations as comprehensive and fine-grained as the original security hooks deeply embedded in the code? We hold a positive opinion because of the following two arguments: **First**, OpenStack basically have all main functions defined as API calls, all of which are REST based and can be invoked through web server gateway interface (WSGI). By intercepting this interface, AEM is supposed to be capable of filtering all incoming requests. This guarantees that all accesses towards the cloud are mediated and restricted. **Second**, all critical operations embedded inside the cloud that are not reflected as REST calls but as well require to be access controlled are well-handled. In fact, we have found these kinds of operations. Like `nova show demo-instance1`, when nova is handling this call, it not only checks against the `compute:get` rule, but also enforces several additional rules, including `compute_extension:security_groups`, `compute_extension:keypairs`, etc. Fortunately, we found that only one of these rules, `compute:get` in this example is the "main" rule. Without the approval of the main rule, the whole action would fail. Based on this pattern, the OSM framework only needs to check the main rule to achieve the same effect as the original hooks. For brevity, in the following sections, we will assume that there is only one rule (op) for each function call.

B. Multi-Policy Mechanism

The original policy for OpenStack is a single JSON file called `policy.json` locally stored on the service node. A number of rules are provided in that file and are supposed to be globally enforced for every cloud user, including cloud administrators. A user cannot specify his own policy rules or make changes to the original policy enforcement mechanism. This structure was poorly designed and failed to meet policy customization demands from both consumer and cloud provider's perspectives. Therefore, this paper proposes a model called multi-policy mechanism (MPM). In MPM, two types of policy are designed as below:

Global policy. This type of policy is provided by the cloud provider. It can be enforced on all tenants across the whole cloud, and only cloud administrator can modify it. A global policy is public to be viewed and used by all cloud users, just like the built-in types for a VM instance.

Customer policy. This type of policy is configurable on the consumer side. It only applies within a tenant-wide

scope, and a tenant's administrator can modify it, while other tenants can neither view nor change it.

MPM is based on metadata, which is a file that describes the multiple policies and organizes them into a tree structure. Policies can be nested. The inner policy is called a "sub-policy" of the outer one. The outmost policy is the root of the policy tree and will be enforced by MPM in a depth-first manner. Policy is composed of a number of fields including name, type, enforcer, version and rules.

When we talked about a security policy, we did not quite distinguish between practical policy rules or just the enforcing logic for this sort of policy. MPM has clarified these conceptions by defining them as rules and enforcer. A security system that supports multiple policies typically allows its users to design their policy rules. However, the underlying enforcement logic of them is usually unchangeable. MPM makes it customizable even for cloud users to offer maximum flexibility, so this policy as well represents the "security module" in OSM. In OpenStack, policy enforcement was originally implemented as a module called `policy.py`, which can be viewed as an enforcer for ABAC policy in our terminology. We refactor it by extracting out the shared logic of a general enforcer into an inheritable base class. A security module's author for the OSM framework should write his own enforcer in accordance with the base class declarations. These declarations can be simplified as below:

- **Input**: request vector (caller context, target context, op), rules
- **Output**: decision (`permit|deny`)

Through this mechanism, a cloud user can customize their own security module by submitting his policy to the cloud. Since enforcer is essentially Python code, the cloud provider is required to provide some sorts of code examination measures for uploaded enforcer files to ensure there is no malicious code included. Since anti-injection has already proved to be an effective measure to prevent arbitrary code loading in web services, we believe the cloud provider can employ similar techniques to perform the policy code examination. Furthermore, some common enforcers like the original enforcer of OpenStack should be provided. And their policy should be viewable and editable in an user-friendly manner like a graphical user interface.

We also emphasize on the strategy about who can access the policy. It includes four constraints:

Constraint I. A cloud administrator should be privileged to access all tenants' policies through read-only functions such as `get-policy` for maintenance convenience, because this allows him to assist users in troubleshooting potential errors of policy configuration. However, he should be forbidden to set a customer tenant's policy to avoid an insider attack. The second task for a cloud administrator is to manage global policies, so he must have full authorization to access and modify them. We implemented it straightforwardly by posing global policies and cloud administrators in a single tenant `admin`, so that cloud administrators can modify global policies without additional settings.

Constraint II. A tenant administrator is approved to call all functions provided by patron for managing the policy of his own tenant.

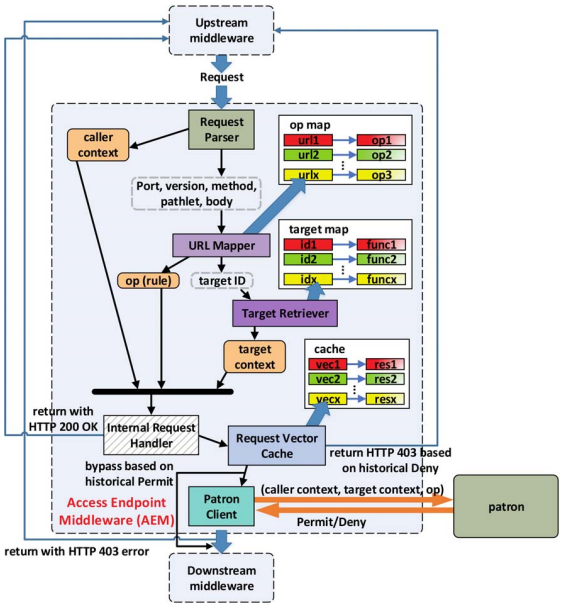

Figure 2. AEM Internal Design

Constraint III. An end-user is only approved to access resources in his own tenant by default.

Constraint IV. Only patron is authorized to call any functions provided by AEM.

IV. Implementation

In this section we show details about the implementation of the OSM framework.

A. AEM

As shown in Figure 2, AEM is designed to be composed of several components, including **Request Parser**, **URL Mapper**, **Target Retriever**, **Internal Request Handler**, **Request Vector Cache** and **Patron Client**. AEM first receives the request from its upstream WSGI middleware (aka `keystonecontext`). It then gathers useful information using Request Parser, URL Mapper and Target Retriever. If the request is dedicated to AEM itself, it will be handled in Internal Request Handler. Otherwise it will be looked up on the Request Vector Cache. If the cache is not hit, finally AEM will query for a decision by sending a validation request to patron via Patron Client. Based on patron's decision, AEM chooses to deliver the request to the next middleware or just deny the request by returning authorization failure (aka `HTTP 403 error`) to the prior middleware.

1) Setting up: As a part of the OSM framework, AEM serves as an extension to target services to provide access controls for them. OpenStack typically provides its services based on WSGI specification. This standard specifies a structure called filter chain, which is an extension mechanism that supports the preprocessing and filtering of incoming requests before their arrival to the application side. Keystone has utilized these filters to offer authentication for the cloud. Thus a natural way to think about it would be implementing AEM

as a WSGI filter as well. It is called `patron_verify` and exactly located after `keystonecontext` (keystone's middleware). We believe this is an optimal place as access control always comes after authentication.

2) Op and Target Map: As shown in Figure 2, there are two data structures required by AEM, op map and target map. As a cost of removing hooks, AEM does not naturally understand which operation a request is going to perform and which resource it is going to operate on without additional information. Actually, we found there exists a mapping between the request and its corresponding operation, which allows us to use a map (aka op map) to store this information beforehand. Thus when a request arrives, AEM can recognize its operation by looking up this map.

The key of op map is called path tuple. It is a 5-tuple:

- `port`: the port of target service.
- `version`: the API version of target service.
- `pathlet`: the path and query parts of the requested URL.
- `method`: the `method` field of the request's HTTP header.
- `inner action`: the `body` field of the request's HTTP header.

It is notable that inner action is optional and used by OpenStack to represent extra information like parameters or actions if pathlet has not specified them clearly. Take `nova show demo-instacne1` for instance. The path tuple is:

```
(8774, "/v2", "/servers/cb5e2885-ebf1-438a-89db-2628
4bdf75c1", "GET", "")
```

Since an entry from op map is supposed to match all operations of the same type, individual information like a specific UUID of an instance should be generalized before getting stored in op map. Like the pathlet below:

```
/servers/cb5e2885-ebf1-438a-89db-26284bdf75c1
```

It will be generalized to:

```
/servers/%UUID%
```

The value of op map is called operation name, abbreviated as op. It is originally defined in the mechanism of policy enforcement of OpenStack. By adding log printing code (print the path tuple and op together), we have collected op maps for a couple of services. An op map can compose of up to hundreds of entries, the size of which depends on amount of function calls provided by the target service.

So far, op map is provided to answer the first question we posed at the beginning of IV-A2. The next is how to retrieve the resource object the request is asking for. As we have observed, pathlet has already provided the identifier of the resource, what we call target ID. However, the original policy enforcement of OpenStack is based on target context (actually an attribute list describing details of a target resource). Fortunately, we found OpenStack has been designed in a paradigm of object-oriented model. An object retrieving function (ORF) is provided to obtain each type of objects, like VM instances, network interfaces, key pairs, disk volumes, etc. These functions translate the identifiers into the object contexts. Thus, AEM can leverage these functions to obtain the context of a resource. A good news is the built-in object types are still humanly enumerable (only up to 21 types for nova), which allows us to manually connect an object type with its ORF. Since Python is a dynamic scripting language, ORFs can be implemented using reflection mechanism provided by the language.

3) Data Collection: After the preparations of configuration and map data, this section will describe AEM's actual processing steps for a request. As a new request comes in, the first phase for AEM to do is collecting security related data. Request Parser, URL Mapper and Target Retriever are involved in this phase.

Request Parser is leveraged to translate the coming request into two parts: caller context and path tuple. As a part of the request, caller context is essentially an attribute list describing the credentials of the calling user, including his tenant identifier, user identifier, roles, etc. The other part of the output is path tuple, as we described in the previous section.

URL Mapper takes the path tuple provided by Request Parser as input and retrieves the operation by looking up op map provided in Section IV-A2. Meanwhile, the target ID is also parsed out to serve as the input of Target Retriever.

Target Retriever then obtains the ORF by looking up the object type in target map provided in Section IV-A2 which is generated by URL Mapper.

4) Decision Caching and Querying: AEM normally queries for a security decision each time when needed. This manner will bring additional time overhead, mainly the network latency between AEM and patron. To alleviate this situation, a cache module called Request Vector Cache is provided in AEM. Caching will be performed for each patron's decision, so that a new request can just use the cached result, instead of accessing patron again. If cache misses, AEM then queries patron for decision making, and newly obtained ruling result will be buffered in the cache.

Since the buffer space of the cache is limited, AEM must ensure as little as possible about the memory use of a cached entry. The structure of a cache entry is shown as follows:

```
(op, caller_tenant_id::caller_user_id, target_tenant
_id::target_id) -> action
```

`op` is the aforementioned operation name, while `caller_tenant_id` and `caller_user_id` together can identify a caller, similarly, `target_tenant_id` and `target_id` can be used to identify a resource. `action` is simply a boolean value to represent permitted or denied. It is notable that a cloud user under different tenants can possess distinct permissions. To differentiate them, the cache must take tenant identifiers into consideration as a part of the mapping key.

If the coming request is neither an internal request nor stored in the cache, decision querying will be performed by the Patron Client module. This module utilizes a client tool called `python-patronclient` built by us to call patron's functions. Finally, AEM will check patron's response. As a special case, if the querying process fails or timeouts, AEM will by default reject the request for security purpose.

5) Internal Requests: When policy on patron's side gets updated, corresponding cache in AEM also should be wiped

or it will be in an inconsistent state. Patron needs a way to inform AEM of the cache wiping event. Since AEM is implemented as an attachment to other services, it needs to provide its own calls for patron. We call those requests intended for AEM itself as internal requests, as they are supposed to be handled internally in its Internal Request Handler and will not go further into the service behind AEM. This manner effectively minimizes OSM's invasion to the target service. Currently, only one function `wipe-cache` has been exposed by AEM. However, it can be easily extended to provide new calls.

B. Patron

Patron is the newly proposed service which plays a crucial part in the OSM framework. It provides access controls for all functions calls to the REST interfaces of the cloud. It primarily composes of three parts:

- `patron-api`: serves as a REST interface of the patron service.
- `patron-verify`: consults access rules in the policy and determines an access ruling result in response to the query from AEM.
- `patron-update`: manages the storage of all patron's policies, controls the policy updates and also provides a functionality to send notifications to AEM for events like cache wiping.

Additionally, a database and a message queue are also required by patron just like other services. The database is currently only used for storing metadata about the policy. The practical policy rules are stored on disk. The message queue is used to provide communications between `patron-api`, `patron-verify` and `patron-update`. It is worth noting that patron can be deployed on multiple nodes just as other services do, so requests to patron API will be load-balanced to gain performance and avoid single point failures.

1) Provided Functions: As the name suggests, `patron-api` is mainly implemented to provide REST interfaces for patron by listening at port `8292` for connections. Since patron is also provided as a service, its interfaces also require to be access controlled by AEM. This means that all requests to functions provided by patron will be mediated by patron itself. This kind of manner might cause deadlock if not handled correctly: a function call to patron asks for a permission from patron. However, that mediation itself is also a function call to patron that demands another query to patron. It is not hard to see that the `verify` function call will cause an endless loop of calling and it requires to be specially handled to interrupt the loop. A rational solution is letting AEM act differently for this function: instead of querying patron for decision, AEM only performs a local check to ensure the caller is checking his own access rights. Since the caller context of a `verify` call just comes from the request context which has been authenticated by keystone, this prevents a malicious user to fake another user's identity and peep at other's access rights by testing against the `verify` function.

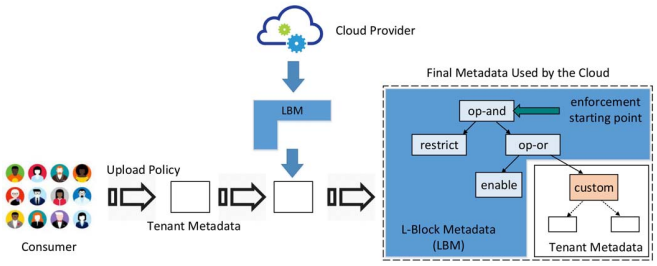

Figure 3. The Process of Wrapping Uploaded Metadata with LBM

2) Policy Enforcement: Based on the constraints in III-B, we can deduce several policies:

- `enable`: the policy enabling the rights for the administrators to configure their policies. This policy is based on `Constraint I` and `Constraint II` and enforced by patron.
- `restrict`: the policy to ensure users can only have rights to access resources belonging to their own tenants, so unauthorized access is restricted. This policy is based on `Constraint III` and enforced by patron.
- `self-protect`: the policy to ensure functions provided by AEM cannot be invoked by any others except patron. This policy is based on `Constraint IV`. It is notable that this policy is directly enforced by AEM to avoid the loop described in IV-B1.

The `enable` policy is shown below:

```
"is_admin": "role:admin",
"cloud_admin": "project_id:admin and rule:is_admin",
"project_admin": "project_id:%(project_id)s and
rule:is_ admin",
"cloud_or_project_admin": "rule:cloud_admin: or
rule:project_admin",
"access:get_policy": "rule:cloud_or_project_admin",
"access:set_policy": "rule:project_admin",
```

The `restrict` policy is shown below: (`*` means for all operations)

```
"*": "project_id:%(project_id)s"
```

The `self-protect` policy is shown below:

```
"access:verify": "project_id:service and user_id:pat
ron"
```

At present, policy has already become customizable for consumers, including global policy and customer policy. Those policies are all optional and can be specified in a tenant's metadata. This has made MPM look likely to be a discretionary access control (DAC) design. However, a cloud provider also has requirement to enforce some mandatory access control (MAC) restrictions. For example, a tenant user can only access his own resources (the `restrict` policy), or a policy administrator in a tenant should always have the rights to configure the policy for his tenant (the `enable` policy), etc. These global restrictions are primarily for the majority of consumers and should not be violated in any way. Given that the policy for a tenant is controlled by the tenant administrator himself, a wrapping approach is proposed to encapsulate the tenant's metadata with a "L-Block Metadata" (LBM) as the final metadata to be enforced

for that tenant. LBM is essentially a metadata with an empty slot for another metadata to be "grafted" there as shown at the right side of Figure 3. The wrapping happens automatically when a cloud user uploads his metadata. So there would be no chance for a malicious user to bypass it. A typical processing of wrapping a metadata with LBM is shown in Figure 3. LBM gets its name as it has a shape of "L". The metadata of LBM is a combination of four global policies: `enable`, `restrict`, `op-and` and `op-or`. `enable` has already been described in this section earlier. It is connected with the customer-provided policy `custom` with an `op-or`, which represents granting the policy set permissions to administrators in any case. This is used to prevent user's misoperation because there is a chance that a tenant administrator incautiously locked down himself by revoking his own policy set privileges. This policy is connected with `restrict` by an `op-and`, because no more rights should be given beyond `restrict`. Otherwise illegal across-tenant access occurs. This mechanism of LBM ensures a cloud user could not modify global settings to breach cloud security by tamping with his metadata.

V. Evaluations

The experimental cloud is deployed following a four-node configuration: controller node, network node, compute node and block storage node. Patron is deployed on the controller.

A. Weakness of Hooks

Through analyzing the result of op map, we found a surprising fact that not all functions provided by OpenStack are currently under policy enforcement, as shown in Table I. 105 out of 371 operations are performed without any access control. 64 of them are intended to be used as open interface, which means the function designer does not want authorization check to occur. Examples like these are `instance type list`, `server availability zone list`, etc. However, the remaining 41 functions are critical and unprotected by the policy, like changing arbitrary machine or network states. Specifically, only heat and ceilometer have put all their functions under protection by the policy. The worst case is neutron for 19 out of 89 functions can be called without authorization. This manner greatly endangers the whole system as those functions can be called by any cloud user without any permission checking. The fundamental cause of this is the cloud provider can hardly have a precise understanding about the situation of security hooks. As a rapidly developing open-source project, OpenStack has involved a large foundation of developers in its development process. Numerous features can be added on a daily basis. It is difficult for anyone to investigate or supervise whether a function has been accompanied by an appropriate permission check. This should be viewed as a potential risk because the adversary has been given a chance to identify those unprotected calls and exploit them to compromise the cloud. The OSM framework offers the possibilities for a cloud provider to comprehend the whole security situation by simply examining the entries of op map. In the legacy hooking way, if a careless programmer forgets to add security check for his code, the function still runs. However, in the OSM framework any operations not registered in op map will be denied by default. This forces developers to register their operations and designing proper policy in patron before getting new features online, which ensures that all accesses will be controlled in the cloud.

Table I
Unprotected Ops of Op Map

Service	Op map entries	Empty ops		%Vulnerable rate
		Intended	Vulnerable	
nova	156	13	17	10.9%
glance	48	6	3	6.3%
neutron	89	40	19	21.3%
cinder	39	3	2	5.1%
heat	27	2	0	0%
ceilometer	12	0	0	0%

Table II
Tempest Benchmarks, Time in Seconds

Service	Liberty	OSM	Overhead	+Cache	Overhead
nova	651.68	714.44	9.6%	701.76	7.7%
glance	229.70	291.02	26.7%	253.16	10.2%
neutron	230.31	260.45	13.1%	248.27	7.8%
cinder	136.83	171.32	25.2%	148.78	8.7%
heat	292.62	345.34	18.0%	317.80	8.6%
ceilometer	618.79	628.46	1.6%	622.31	0.6%

B. Performance

We used tempest [19] for benchmarking. We compared the standard OpenStack Liberty cloud against an alternative with the OSM framework applied (for both AEM's cache off and on). The results are shown in Table II. The additional time introduced by the security framework is acceptable, as the average cost was 15.7% for cache disabled, and 7.3% for cache enabled. This overhead is primarily due to the communication delays between patron and AEM. The worst case was 26.7% for cache disabled and 10.2% for cache enabled in glance. This result is expected because of the relatively small amount of time consumed in each glance call compared to the execution of permission checking. The effect of OSM to ceilometer is not obvious due to fewer operations needed to be access controlled. The average additional cost per function call is close to 122ms, which is a fairly small figure compared with the delay across a large-scale public network like Internet.

C. Invasiveness of OSM

In Table III we present data that give a rough estimate of the scale and complexity of adding centralized security enforcement to OpenStack. Overall, AEM's code modification to a service was only limited to three Python code files, summing up to hundreds of LOC. And those components increased in size less than 1.5% (except patron). Besides these code modifications, we also examined other types of changes involved, like configuration changes, data changes, etc. Since AEM is designed to be a WSGI attachment, this needs a couple of lines of modifications in the configuration

Table III
MODIFICATIONS OF OSM TO OPENSTACK, NUMBER IN LINES

Service	LOC without AEM	OSM's change					%Incr
		-Hooks	+AEM code	+AEM config	+AEM op map	+AEM target map	
nova	275801	454	767	4	156	21	0.34%
glance	69832	44	767	5	48	6	1.18%
neutron	118883	161	767	3	89	12	0.73%
cinder	186606	77	767	6	39	5	0.44%
heat	100990	70	767	4	27	6	0.80%
ceilometer	54732	19	767	3	12	1	1.43%
patron	8878	N/A	767	4	6	4	N/A

of that service. Additionally, supporting data like target and op maps are also represented in Table III. Since these data provide extra knowledge for function calls, their sizes keep proportional to the amount of calls in that service.

The changes required to implement the OSM framework did not involve any modifications to the existing Python code or REST calls. A couple of calls are provided by patron to support policy management for both cloud consumers and service providers. Furthermore, an internal call is extended on AEM for cache wiping. Despite the fact that AEM's call is sharing the same REST interface with the attached service, it does not actually increase invasiveness since they are straightly handled by AEM and have no involvement with the inner logic of attached service. All applications that run on the stock OpenStack cloud can be executed unchanged on a cloud equipped with OSM framework.

VI. CONCLUSION

The OpenStack project supports the classical ABAC security policies to be enforced locally as the primary way of access controls, which in many cases is not adequate. The combination of open source code and broad popularity has made OpenStack a popular target for enhanced cloud security frameworks. The OSM project exists to provide a least invasive security framework for access controls in OpenStack. It includes a new policy service called patron and an attachment module for target services called AEM. We presented the design and implementation of the OSM interface. OSM provides a multi-policy mechanism that is rich enough to enable a wide variety of security policies, while imposing minimal disturbance to the OpenStack source code, and minimal overhead on the performance. The average overhead is 7.3%, which is an acceptable result.

ACKNOWLEDGMENT

We thank the reviewers for their help improving this paper. This work was supported by National Natural Science Foundation of China under Grant No. 61232005, National 863 Program of China under Grant No. 2015AA016009 and Shenzhen Technology Program of China under Grant No. JSGG20140516162852628.

REFERENCES

[1] S. Crago, K. Dunn, P. Eads, L. Hochstein, D.-I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. P. Walters, "Heterogeneous cloud computing," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011, pp. 378–385.

[2] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications*, vol. 34, no. 1, pp. 1–11, 2011.

[3] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security & Privacy*, no. 6, pp. 24–31, 2010.

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

[5] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo, "Attribute-based access control," *Computer*, no. 2, pp. 85–88, 2015.

[6] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, no. 2, pp. 38–47, 1996.

[7] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[8] OpenStack, "Openstack liberty," https://www.openstack.org/software/liberty, 2016.

[9] Amazon, "Amazon web services," http://aws.amazon.com, 2016.

[10] Microsoft, "Microsoft azure," https://azure.microsoft.com, 2016.

[11] Google, "Google app engine," http://cloud.google.com/appengine, 2016.

[12] R. Wu, X. Zhang, G.-J. Ahn, H. Sharifi, and H. Xie, "Acaas: Access control as a service for iaas cloud," in *Social Computing (SocialCom), 2013 International Conference on*. IEEE, 2013, pp. 423–428.

[13] B. Tang and R. Sandhu, "Extending openstack access control with domain trust," in *Network and System Security*. Springer, 2014, pp. 54–69.

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

[15] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Foundations of Intrusion Tolerant Systems*. IEEE, 2003, p. 213.

[16] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security module framework," in *Ottawa Linux Symposium*, vol. 8032, 2002, p. 6.

[17] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The flask security architecture: System support for diverse security policies," 1999.

[18] M. D. Abrams, K. W. Eggers, L. LaPadula, and I. Olson, "A generalized framework for access control: An informal description," in *Proceedings of the 13th National Computer Security Conference*, 1990, pp. 135–143.

[19] OpenStack, "Openstack tempest," https://github.com/openstack/tempest, 2016.