MultiPol: Towards a Multi-policy Authorization Framework for RESTful Interfaces in the Cloud

Yang Luo[1], Tian Puyang[1], Wu Luo[1], Qingni Shen[1], Anbang Ruan[2], and Zhonghai Wu[1](✉)

[1] Peking University, Beijing, China
{luoyang,puyangsky,lwyeluo,qingnishen,wuzh}@pku.edu.cn
[2] University of Oxford, Oxford, UK
anbang.ruan@cs.ox.ac.uk

Abstract. Recently a large number of existing cloud systems adopt representational state transfer (REST) as the interface of their services. The end users or even components inside the cloud invoke RESTful calls to perform various actions. The authorization mechanisms of the existing clouds fail to supply two key elements: unified access control and flexible support for different policies. Moreover, different clouds usually provide distinct access control concepts and policy languages. This might cause confusion for customers whose business is distributed in multiple clouds. In this paper, we propose a multi-policy authorization framework called MultiPol to support various access control policies for OpenStack. The end users can customize or even integrate different policies together to form a single decision via logical connectors. This paper presents the design and implementation of MultiPol, including a new service called Policy Service and an attachment module called Request Filter. Experiments on OpenStack show that MultiPol has improved the flexibility and security of policy management without affecting other services. Meantime, the average performance overhead is as low as 7.8%, which is acceptable for practical use. Since MultiPol is built on REST, it is also adaptive to other clouds which also provide RESTful interfaces.

Keywords: Representational state transfer · Access control · OpenStack · Multi-policy

1 Introduction

Cloud computing has become a revolutionary force for enterprises to reduce their costs by using on-demand computational infrastructures [1–3]. Although being the most widely-adopted open sourced cloud, OpenStack failed to provide adequate consideration on security, especially the access control area. Presently, 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 mechanism of OpenStack is still unable to provide strong security. Locally stored policy is used to enforce access control for almost all OpenStack services, and

K.-Y. Lam et al. (Eds.): ICICS 2016, LNCS 9977, pp. 214–226, 2016.
DOI: 10.1007/978-3-319-50011-9_17

the policy is built in, which disallows anyone to modify it except cloud service provider itself. From the perspective of access control, attribute-based access control (ABAC) [4,5] is readily adopted by OpenStack and role-based access control (RBAC) [6] is partially supported as well. However, these models are hard-coded into the cloud and lack adequate flexibility to be customized. The above implementation has several limitations as an approach for providing cloud access control. First, the policy is currently decentralized. All access controls are restricted within the scope of a physical host. A policy supervisor has to suffer preparing a policy for each service and manually deploying them to all cloud nodes, the process of which can be quite unfriendly and error-prone. Second, the security hook code is a headache for developers. A part of the permission checks are even hard-coded into the platform. If a cloud customer is unwilling to use the built-in security policy and desires to customize his own, there will be no way for him to achieve this.

To solve the above-mentioned issues, in this paper, we propose an authorization framework called multiple-policy framework (MultiPol). It includes a standalone service called `Policy Service`, and an attachment module called `Request Filter` 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 highly loose-coupled with other services). With the MultiPol 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 `Request Filter` first. `Request Filter` then validates it by sending its security contexts to `Policy Service`, which will make a proper decision based on policy enforcement. If `Policy Service` says yes to this access, the request will be permitted by `Request Filter` to reach the demanded service. Otherwise `Request Filter` will reject it by returning an error. The MultiPol 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. And the security settings, especially policy configurations can be modified through the unified `Policy Service` interface. Policy Service manages multiple policies provided by both cloud providers and consumers. Several calls are provided by `Policy Service` and `Request Filter` as a part of the framework. We have implemented the security framework based on the latest OpenStack Mitaka [8]. Despite the fact that OpenStack employs REST as its primary interface, the MultiPol framework is general enough to be applied to other kind of interfaces, like simple object access protocol (SOAP), etc.

The remainder of this paper is organized as follows. Section 2 elaborates on the background. Section 3 presents the design of MultiPol framework. Section 4 brings the implementation. Section 5 describes experimental results. Section 6 concludes this paper.

2 Background

For the past few years, there has been a considerable interest in environments that support multiple and complex access control policies [9–13]. Access control as a service (ACaaS), proposed in [14] by Wu et al., provided a new cloud service to enforce a 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 [15] 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 MultiPol, which increases the granularity of access controls. So our work can be well integrated together. The work proposed in [16] 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, 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. The existing solutions usually follow a path by trying to progress in terms of expressiveness and functionality. However, a vast majority of them remain merely academic and lack practical acceptance owing to their complexity in usage or computation. So instead of making a universal policy model applicable for all scenarios, we try to narrow down the scope to the most popular cloud systems. And we found that recently, REST [7] has become the most widely-accepted interface standard for the clouds.

Systems that conform to the constraints of REST can be called RESTful. RESTful systems typically, but not always, communicate over hypertext transfer protocol (HTTP) with methods like `GET`, `POST`, `PUT` and `DELETE` that web browsers use to retrieve web pages and send data to remote servers.

As being an abstract architectural style instead of a strictly defined protocol, REST does not seek to specify the accurate syntax on the request form. This fairly results in the current numerous implementations respectively from different service providers. However, based on the analysis upon the interface design from mainstream web services in the marketplace, we can extract the representative elements for a typical RESTful request. A standard request form (SRF) is defined from those elements to ensure that the vast majority of the RESTful requests would fit in it. The MultiPol framework accepts the input of SRF requests to ensure it can be seamlessly applied to a control system using the SRF. Now we present the definition of SRF as below:

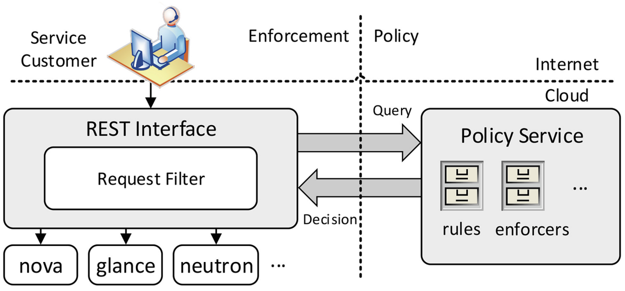


Fig. 1. Architecture of multiPol authorization framework

```
Verb
{http|https}://DomainName[/Version]/Object/
```

`Verb`: the HTTP method, possible values are `GET`, `POST`, `PUT` and `DELETE`.

`DomainName`: the domain name of the cloud service provider.

`Version`: the version of the REST interface this request is asking for. Offering version numbers in REST calls is usually a good design for maintaining multiple API versions simultaneously.

`Object`: the part of the path in the URI, which identifies the resource. It is generally an ordered list of strings separated by `/`.

It is notable that as an essential part for an access behavior, the `subject` does not show up in the SRF. This is due to the fact that the `subject` is usually implicitly embedded in the HTTP header, or represented by an access token retrieved in the previous authentication step. Therefore, this entity needs to be obtained according to the specific implementation. An example of a policy rule for RESTful interfaces is shown below.

```
Subject, Object, Verb -> Effect
```

In the above expression, `Effect` is the authorization decision that the policy declares to enforce. To support both positive and negative authorizations, `Effect` can be assigned to either `Allow` or `Deny`. Finally, the entire hierarchy of the RESTful policy syntax can be described in backus-naur form like below:

```
Policy ::= [Version,] Rules
Rules ::= {Rule}
Rule ::= Subject, Object, Verb -> Effect
Verb ::= GET | POST | PUT | DELETE
Effect ::= Allow | Deny
```

3 MultiPol Design

The architecture of the MultiPol authorization framework is shown in Fig. 1. We divide it into two parts: `Request Filter` and `Policy Service`. `Request`

`Filter` is responsible to intercept all the calls to OpenStack's REST interface. `Policy Service` is the container of policy rules. Each of the REST requests towards the cloud (including requests sent by not only the outside consumers but also the cloud infrastructure itself) is filtered by `Request Filter` first. `Request Filter` then validates it by sending its security contexts (`Subject`, `Object`, `Verb`, etc.) to `Policy Service`, which will make a proper decision based on policy enforcement. If `Policy Service` says yes to this access, the request will be permitted by `Request Filter` to reach the target service.

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, the MultiPol framework intends to solve this issue. In the MultiPol framework, 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.

The MultiPol framework 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 MultiPol in a depth-first manner. Policy is composed of a number of fields including name, type, enforcer, version and rules, which are described in the Table 1.

When we talk about a security policy, we do not quite distinguish between practical policy rules or just the enforcing logic for this sort of policy. MultiPol 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. MultiPol makes it customizable even for cloud users to offer maximum flexibility. In OpenStack, policy enforcement was originally implemented as a module called `policy.py`, which can be viewed as an enforcer for different policy languages including the original ABAC policy. We refactor it by extracting out the shared logic of a general enforcer into an inheritable base class, so MultiPol can support multiple policies. A policy language author for the MultiPol should write his own enforcer in accordance with the base class declarations. These declarations can be simplified as below:

- **Input**: request vector (subject, object, verb), policy rules
- **Output**: decision (`permit|deny`)

Table 1. The fields of a multiPol policy

Fild	Meaning	
Name	The identifier for the policy, must be unique within the metadata	
Type	A flag tells whether the policy is provided by cloud provider itself, possible values are:	
	`Global`	This policy is provided by the cloud provider itself, customer can just refer to it by name
	`Customer`	This is a user-customized policy, and the customer has to specify its content. This value is by default
Enforcer	The processing logic for this policy, output ips decision, possible values are:	
	`Default`	The original ABAC policy enforcer adopted by `policy.json`
	`op-and`	The intersection enforcer, meaning all sub policies will be composed in a deny-override manner
	`op-or`	The union enforcer, meaning all sub policy decisions will be composed in an allow-override manner
	`all-pass`	A special enforcer that permits all accesses
	`all-forbid`	A special enforcer that denies all accesses
	custom	A cloud user can customize a policy enforcer on his own
Version	A descriptive string that indicates the current version of the policy	
Rules	This field's meaning varies based on different situations:	
	When *enforcer* = `all-pass` or `all-forbid`	This field will be omitted
	When *enforcer* = `op-and` or `op-or`	This field will be a name list of its sub policies
	other conditions	The field will be the rules of the policy

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 examine measures for uploaded enforcer files to ensure there are no malicious code included.

4 Implementation

4.1 Request Filter

As a part of the MultiPol framework, `Request Filter` serves as an extension to target services to provide access controls for them. OpenStack typically provides its services based on web server gateway interface specification (WSGI). This standard specifies a structure called filter chain, which is an extension mechanism that supports the preprocessing and filtering of incoming requests before they arrive 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 `Request Filter` as a WSGI filter as well. It is called `multipol_enforce` and exactly located after `keystonecontext` (keystone's middleware). We believe this is an optimal place as access control always comes after authentication.

`Request Filter` attempts to reduce invasiveness to other services by limiting modification of `Request Filter` to a couple of lines of configuration, so no existing code change is involved. The following code fragment shows `Request Filter`'s modification in nova's configuration: */etc/nova/api-paste.ini*.

```
[composite:openstack_compute_api_v2]
keystone = compute_req_id faultwrap sizelimit authtoken keystonecontext
multipol_enforce ratelimit osapi_compute_app_v2
keystone_nolimit = compute_req_id faultwrap sizelimit authtoken
keystonecontext multipol_enforce osapi_compute_app_v2
[filter:multipol_enforce]
paste.filter_factory = multipol.rf.enforce:Multipol Enforce.factory
```

`Request Filter` normally queries for a security decision each time when needed. This manner will bring additional time overhead, mainly the network latency. To alleviate this situation, a cache module is provided in `Request Filter` for each decision from `Policy Service`. So that a new request can just use the cached result, instead of accessing `Policy Service` again. If cache misses, `Request Filter` then queries `Policy Service` for decision making, and newly obtained ruling result will be buffered in the cache.

4.2 Policy Service

`Policy Service` is the newly proposed service which plays a crucial part in the MultiPol enforcement framework. It provides access controls for all functions calls to the REST interfaces of the cloud. The structure of `Policy Service` is shown in Fig. 2. It primarily composes of three parts:

– `API Module`: serves as a REST interface of the `Policy Service`.
– `Verify Module`: consults access rules in the policy and determines an access ruling result in response to the `verify` query from `Request Filter`.
– `Update Module`: manages the storage of all policies in the `Policy Service`, controls the policy updates and also provides a functionality to send notifications to `Request Filter` for events like cache wiping.

Additionally, a database and a message queue are also required by `Policy Service` 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 among `API Module`, `Verify Module` and `Update Module`.

It is worth noting that the `Policy Service` can be deployed on multiple nodes just as other services do, so requests to `Policy Service` API will be load-balanced to gain performance and avoid single point failures.

Since `Policy Service` is also provided as a service, its interfaces also require to be access controlled by `Request Filter`. This means that all requests to functions provided by `Policy Service` will be mediated by `Policy Service` itself. This kind of manner might cause deadlock if not handled correctly:

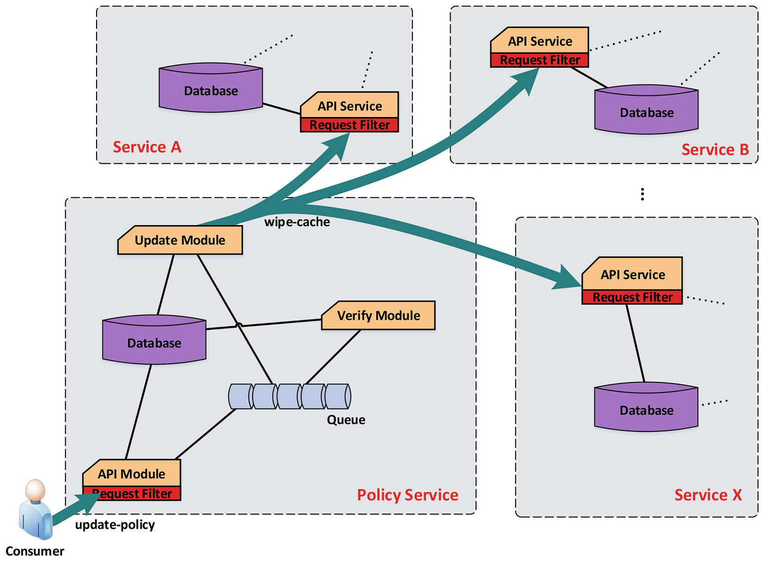


Fig. 2. The components of `Policy Service`

a function call to `Policy Service` asks for a permission from `Policy Service`. It is easy to see that the `verify` operation will cause an endless loop of calling and it requires to be specially handled to interrupt the loop. A rational solution is letting `Request Filter` act differently for this function: instead of querying `Policy Service` for decision, `Request Filter` 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.

`Update Module` is the handler for policy update matters, including managing policy storage, enforcing policy constraints and calling `Request Filter` for purpose like cache wiping.

Since multiple policies are supported in `Policy Service`, first we need to design the storage arrangement to hold these policies. A good implementation is storing the policies based on their types. Global policies are cloud-wide functioned and should be stored in a top-level path like `/etc/multipol/`. While customer policies are only restricted to tenants and should be stored in a tenant-specific path like `/etc/multipol/customer_policy/%TENANT_ID%/` (where `%TENANT_ID%` represents the tenant identifier). The filename of a policy is identical with its occurrence in the metadata, so the enforcer can easily find the related policy files by parsing the metadata. Since a policy name in a metadata is unique, there will not be a filename conflict in the storage stage. For convenience, we refer to policy, enforcer, metadata together as policy in this paper.

Next we will illustrate MultiPol's strategy about who can access the policy. It includes four constraints:

Constraint I. A cloud administrator should be privileged to read all tenants' policies through for maintenance convenience. Because it is helpful for troubleshooting potential errors of policy configuration. However, the administrator should be forbidden to set customer policies to avoid an insider attack. The second task for a cloud administrator is to manage global policies, so he is supposed to be fully authorized to read and modify them. MultiPol enforces this constraint straightly by posing global policies and cloud administrators in the same tenant, so that cloud administrators can modify global policies without additional settings.

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

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

Constraint IV. Only `Policy Service` is authorized to call any functions provided by `Request Filter`.

Based on the above constraints, we can deduce several global policies, which can be enforced by the MultiPol framework for all the cloud users:

– `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 `Policy Service`.
– `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 `Policy Service`.
– `self-protect`: the policy to ensure functions provided by `Request Filter` cannot be invoked by any other code except `Policy Service`. This policy is based on `Constraint IV`. It is notable that this policy is directly enforced by `Request Filter` to avoid the above mentioned deadlock.

5 Experiments

This section shows how we implement MultiPol enforcement framework in the OpenStack cloud and evaluate its performance and usability.

5.1 Performance

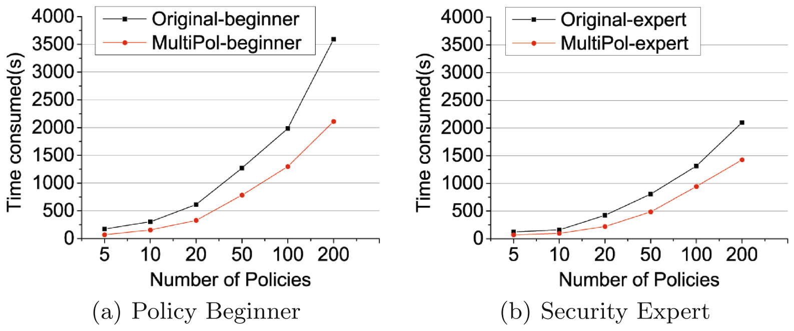
We used tempest [17] for benchmarking. We compared the standard OpenStack Mitaka cloud against an alternative with the MultiPol framework applied (for both `Request Filter`'s cache off and on). The results are shown in Table 2. The additional time introduced by the security framework is acceptable, as the average cost was 14.2% for cache disabled, and 7.8% for cache enabled.

Table 2. Tempest benchmarks, time in seconds

Service	Mitaka	MultiPol	%Overhead	MultiPol+Cache	%Overhead
Nova	643.85	709.31	10.2%	697.27	8.3%
Glance	246.34	288.71	17.2%	275.13	11.7%
Neutron	238.16	268.95	12.9%	253.34	6.4%
Cinder	157.22	199.33	26.8%	175.71	11.8%
Heat	324.35	371.28	14.5%	349.68	7.8%
Ceilometer	698.63	723.32	3.5%	706.14	1.1%

This overhead is primarily due to the communication delays between `Policy Service` and `Request Filter`. The worst case was 26.8% for cache disabled and 11.8% for cache enabled in cinder. 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 MultiPol to ceilometer is not obvious due to fewer operations needed to be access controlled. The average additional cost per function call is close to 121 ms, which is a fairly small figure compared with the delay across a large-scale public network like Internet.

Since the cache mechanism in `Request Filter` is a critical component, it is important to evaluate its memory usage. A record in the cache is stored in a format like a vector, which contains information of `Subject`, `Object` and `Verb`. The calculation shows that one record requires 150 Bytes on average. To be more intuitive, we use Tempest to test all commands of OpenStack nova. The size of used cache turns out to be 68.5 KBytes after running 403 tests. Let us assume that the usage of an individual user roughly equals to a Tempest test. If the size of cache is 1 GBytes, a cloud based on Openstack with one nova API service can support nearly 15000 users (1G/68.5K). This capacity is adequate for practical use. Moreover, we can utilize a caching algorithm like least recently used (LRU) to delete a couple of records when the cache size exceeds a preset threshold,



(a) Policy Beginner (b) Security Expert

Fig. 3. Comparison of time consumed on designing of two different policies

such as 85 %. Therefore, the size of cache required by `Request Filter` is affordable, which cannot affect the usage of the MultiPol enforcement framework.

5.2 Usability

We analyzed the total time consumed in the policy designing process conducted by end-users at different degrees (e.g., policy beginners and security experts). The results are shown in Fig. 3. MultiPol has mitigated almost 41.3% efforts for a beginner and 32.1% for security expert on the average. This is because MultiPol allows to use different policies, and we have provided some built-in policy enforcers like ABAC, RBAC, etc. So the users can just choose to write the policy they are familiar with. Moreover, MultiPol has provided a standard request form shared by all kinds of REST interfaces. No matter how many REST systems a customer needs to manage, he only needs to learn the basic elements of access control once, like subject, object, verb, etc. It saves a large number of efforts by reducing the learning time for different policies.

From the developers' perspective, We also give a rough estimate of the scale and complexity of adding centralized security enforcement to OpenStack. In summary, `Request Filter`'s code modification to a service was only limited to several Python code files, summing up to hundreds of LOC. And those components increased in size less than 1.5% (except `Policy Service`). Besides these code modifications, we also examined other types of changes involved, like configuration changes, data changes, etc. Since `Request Filter` is designed to be a WSGI attachment, this needs a couple of lines of modifications in the configuration of that service.

The changes required to implement the MultiPol framework did not involve any modifications to the existing Python code or RESTful calls. A couple of calls are provided by `Policy Service` to support policy management for both cloud consumers and service providers. Furthermore, an internal call is extended on `Request Filter` for cache wiping. Despite the fact that `Request Filter`'s call is sharing the same RESTful interface with the attached service, it does not actually increase invasiveness since they are straightly handled by `Request Filter` 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 MultiPol framework.

6 Conclusion

This paper proposes a multi-policy access control framework called MultiPol for clouds like OpenStack. It includes a new service called `Policy Service` and an attachment module for target services called `Request Filter`. MultiPol is especially designed for RESTful interfaces, so that it can be used in other clouds that support RESTful interfaces too. Meantime, Multipol can express different policies by providing the enforcer mechanism. It utilizes cache to minimize the performance overhead of remote permission checking. The experimental results on OpenStack Mitaka indicate the average enforcement overhead is 7.8%, which is an acceptable result.

Acknowledgment. We thank the reviewers for their help improving this paper. This work was supported by the National High Technology Research and Development Program ("863" Program) of China under Grant No. 2015AA016009, the National Natural Science Foundation of China under Grant No. 61232005, 61672062, and the Science and Technology Program of ShenZhen, China under Grant No. JSGG20140516162852628.

References

1. Crago, S., Dunn, K., Eads, P., Hochstein, L., Kang, D.-I., Kang, M., Modium, D., Singh, K., Suh, J., Walters, J.P.: Heterogeneous cloud computing. In: 2011 IEEE International Conference on Cluster Computing (CLUSTER), pp. 378–385. IEEE (2011)
2. Subashini, S., Kavitha, V.: A survey on security issues in service delivery models of cloud computing. J. Netw. Comput. Appl. **34**(1), 1–11 (2011)
3. Takabi, H., Joshi, J.B., Ahn, G.-J.: Security and privacy challenges in cloud computing environments. IEEE Secur. Privacy **6**, 24–31 (2010)
4. Yuan, E., Tong, J.: Attributed based access control (ABAC) for web services. In: Proceedings of 2005 IEEE International Conference on Web Services, ICWS 2005. IEEE (2005)
5. Hu, V.C., Kuhn, D.R., Ferraiolo, D.F.: Attribute-based access control. Computer **2**, 85–88 (2015)
6. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer **2**, 38–47 (1996)
7. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. dissertation, University of California, Irvine (2000)
8. OpenStack: Openstack mitaka (2016). https://www.openstack.org/software/mitaka
9. Ribeiro, C., Zúquete, A., Ferreira, P., Guedes, P.: SPL: an access control language for security policies with complex constraints. In: Network and Distributed System Security Symposium (NDSS01), pp. 89–107 (2001)
10. Bertino, E., Jajodia, S., Samarati, P.: Supporting multiple access control policies in database systems. In: Proceedings of 1996 IEEE Symposium on Security and Privacy, vol. 1996, pp. 94–107. IEEE (1996)
11. Carney, M., Loe, B.: A comparison of methods for implementing adaptive security policies. In: Proceedings of the Seventh USENIX Security Symposium, pp. 1–14 (1998)
12. Jajodia, S., Samarati, P., Subrahmanian, V., Bertino, E.: A unified framework for enforcing multiple access control policies. ACM Sigmod Record **26**(2), 474–485 (1997)
13. Minsky, N.H., Ungureanu, V.: Unified support for heterogeneous security policies in distributed systems. In: 7th USENIX Security Symposium, pp. 131–142 (1998)
14. Wu, R., Zhang, X., Ahn, G.-J., Sharifi, H., Xie, H.: Acaas: access control as a service for iaas cloud. In: 2013 International Conference on Social Computing (SocialCom), pp. 423–428. IEEE (2013)
15. Tang, B., Sandhu, R.: Extending openstack access control with domain trust. In: Au, M.H., Carminati, B., Kuo, C.-C.J. (eds.) NSS 2014. LNCS, vol. 8792, pp. 54–69. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11698-3_5

16. Jin, X., Krishnan, R., Sandhu, R.: Role and attribute based collaborative administration of intra-tenant cloud iaaS. In: 2014 International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), pp. 261–274. IEEE (2014)
17. OpenStack, Openstack tempest (2016). https://github.com/openstack/tempest