A Lightweight Policy Enforcement System for Resource Protection and Management in the SDN-based Cloud ⋆

Xue Leng[a], Kaiyu Hou[b], Yan Chen[a,b,*], Kai Bu[a], Libin Song[c], You Li[b]

[a] College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China
[b] Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA
[c] TuSimple, San Diego, CA 92122, USA

Abstract

SDN-based cloud adopts Software-defined Networking (SDN) to provide network services to the cloud, which allows more flexibility in network management. Meanwhile, the SDN controller provides users and administrators with various APIs to access and manage network resources. However, unauthorized requests, which are either sent from unregistered users or containing malicious operations, cannot be completely defended. Moreover, the correctness of network configuration in the SDN-based cloud cannot be guaranteed. In this paper, we propose SDNKeeper, a generic and fine-grained policy enforcement system for the SDN-based cloud, which can defend against unauthorized attacks and avoid network resource misconfiguration. Besides, a policy language is designed for administrators to define policies based on the attributes of the requester, resource, and environment. These policies will take effect when there are requests accessing the SDN controller via Northbound Interface (NBI). Specifically, SDNKeeper can block unauthorized network access requests outside the controller to protect the resources inside. Compared to other traditional policy-based access control systems, SDNKeeper is application-transparent and lightweight, which makes it easy to implement, deploy, and reconfigure at runtime. Based on the correctness proof of system design and the prototype implementation and evaluation, we conclude that SDNKeeper achieves accurate and efficient access control with insignificant throughput degradation and computational overhead.

Keywords: Software-defined Networking, SDN-based Cloud, Network Management, Access Control

1. Introduction

Combining the programmability of Software-defined Networking (SDN) [2] and elasticity of the cloud, SDN-based cloud as a new paradigm, provides a more flexible and convenient way to control and manage network resources. These advantages make SDN-based cloud have a broad application prospects. The global cloud service providers like Microsoft Azure [3, 4], IBM [5] and Google [6] are all leading to use SDN in their cloud network architectures. What's more, the Cloud Data Center and Carrier Networks, such as CloudFabric [7] developed by Huawei and NovoDC [8] developed by China Mobile, also adopt SDN-based Cloud. The Synergy Research Group [9] shows the global cloud infrastructure service revenue has reached $12 billion in the third quarter of 2017. And the International Data Corporation [10] predicts that the worldwide market of the SDN-based Cloud Data Center will reach $12.5 billion by 2020.

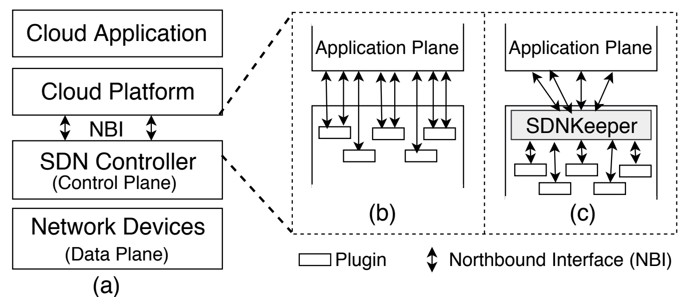

Figure 1: Architecture of SDN-based cloud, as well as the application scenario of *SDNKeeper*.

The abstract architecture of SDN-based cloud is depicted in Fig. 1 (a). The SDN controller is the core component, providing network services for upper layer applications and managing the fundamental network resources. Due to the combination of SDN and the cloud, SDN-based cloud can serve more users with a wealth of services than

⋆This work is supported in part by National Key R&D Program of China (2017YFB0801703), and in part by the Key Research and Development Program of Zhejiang Province (2018C01088). A preliminary version of this manuscript has been published in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, (Banff, Alberta, Canada, June 4-6, 2018) as a regular paper [1].

*Corresponding author
Email addresses: lengxue_2015@outlook.com (Xue Leng), kyhou@u.northwestern.edu (Kaiyu Hou), ychen@northwestern.edu (Yan Chen), kaibu@zju.edu.cn (Kai Bu), slbthu@gmail.com (Libin Song), you.li@u.northwestern.edu (You Li)

SDN, and also manage network resources more flexibly than cloud. Therefore, there are two aspects that need to be considered, one is effectively protecting network resources from users' malicious requests, and the other is flexibly managing various network resources.

Since the advent of SDN, there has been a lot of excellent research focusing on the security of SDN from different angles. To protect the SDN controller, works [11, 12] adopt user authentication to block unregistered users. But these methods would be useless if the legal user is compromised to execute malicious operations with his verified identity. Works [13, 14, 15, 16] perform access control on either the southbound interface or the plugins inside the controller, both of which can not obtain the intention of users intuitively. Works [17, 18] redesign the northbound interface (NBI) to secure the SDN controller, but cannot control user behavior at the attribute level. Based on the current research, securing SDN network at a fine-grained level and managing resources in a uniform manner still have not been solved well.

To address the remaining problems mentioned above, in this paper, we propose a fine-grained policy enforcement system, **SDNKeeper**, to protect and manage network resources in the SDN-based cloud. According to the architecture in Fig. 1 (c), SDNKeeper is at the top of the SDN controller to filter out the malicious requests at the NBI before they invade the controller resources and network resources. Performing access control on NBI can not only reserve the high-level abstraction information from users and upper layer applications, but also block the illegal access requests outside the controller. Hence, the precious controller resources can be used to process the necessary requests.

In order to make better use of SDN, various plugins[1] are developed. These plugins provide a large number of APIs for upper layer applications. Thus, managing these APIs in a unified manner, as well as effectively verifying the legitimacy of access requests is challenging. What's more, to reduce the occurrence of human errors, it's necessary to provide a convenient way for administrators to manage the resources. While designing a friendly and effective way to interact is also a challenge. In the process of designing and implementing SDNKeeper, we overcome the challenges mentioned above.

SDNKeeper performs access control on NBI based on the policies defined by administrators. In order to provide administrators with a convenient way to express their intentions to protect and manage resources, we design a policy language with a readable and operable format, which can narrow down to any specific attribute of the requests and resources. A *policy interpreter* is designed to parse these policies into a controller-processable format and then issue them to the data store. After a request has arrived at the controller, *permission engine* will check its legitimacy against the policies issued previously. The benign requests will continue to be processed by the controller, while the illegal ones will be rejected by the policies and then blocked outside the controller. In summary, all access control strategies for protecting and managing resources can be expressed in our policy language and take effect in our system.

For this work, we made the following contributions.

- We propose a generic policy enforcement system on SDN controller to protect and manage network resources in SDN-based cloud by monitoring the access requests.
- We design a fine-grained policy language for administrators to define management policies, realizing centralized protection and management of resources.
- We adopt formal methods to prove the design correctness of SDNKeeper by constructing the system model and checking the critical properties.
- We implement SDNKeeper with the feature of hot-update, to be specific, it means policy hot-update. Administrators can update policies on the fly and the modified policies will take effect to subsequent requests soon after.
- We evaluate the performance of SDNKeeper with three metrics: effectiveness, latency and throughput, and the results show that SDNKeeper can accurately intercept illegal access requests with minor computational overhead.

The rest of paper is structured as follows. First, research background (Section 2.1) and related work (Section 2.2) are presented. Then, an overview of SDNKeeper is described in Section 3, including application scenarios and the architecture of SDNKeeper. In Section 4 and Section 5, we illustrate the design details of policies and permission engine. Then Section 6 proves the correctness of the system design. Finally, we implement and evaluate the SDNKeeper prototype in Section 7, and conclude the paper in Section 8.

2. Background and Related Work

2.1. Background

Since the birth of SDN, academia and industry have invested a lot of energy in research. Fortunately, by virtue of unique advantages of programmability and centralized control, SDN has been widely used in various scenarios, such as home networking ([19, 20]), enterprise networking ([21]), telecommunication networking ([22, 23]) and data center/cloud networking ([24, 25, 26]). The emergence of SDN has brought new ideas to solve the inherent problems in these scenarios, such as increased management complexity, complex deployment of solutions and high cost for new feature insertion.

[1]A plugin is a project that is located in controller and provides specific functions for upper level applications.

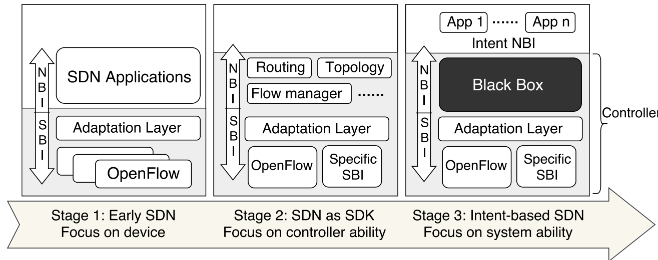

Figure 2: The evolution of Northbound Interface.

In large-scale networking, the resource management as well as the network creation and configuration completely rely on network administrators, so that human errors are inevitable. When the administrator configures the network, he needs to always keep in mind the commands and parameters which are configured previously. With a slight negligence, networks that need to be isolated could communicate with each other. Sometimes these man-made configuration errors are hard to be aware until alerts are received from physical network devices or users. What's more, locating the root cause of network misbehavior is also a challenging work. Under this circumstance, a tool for assisting in managing network resources becomes important and necessary for network administrators.

Besides these inherent drawbacks, when adopting SDN to provide fundamental network services in large-scale scenarios such as cloud, a series of new issues will arise. First, the registered user is compromised to issue malicious requests to the SDN controller. Second, the requests are sent correctly, but the content could have been tampered with during transmission. Finally, attackers can send malicious requests to the SDN controller by invoking the NBI obtained by other means. More details of these issues will be demonstrated in Section 3.2. Without access control and verification mechanism, SDN controller may execute harmful requests and the network will be paralyzed at worst. Since access requests are passed from cloud to the SDN controller via NBI, the best way to protect resources is to perform access control on the NBI. As depicted in Fig. 2, the evolution of NBI has gone through three stages [27]. As the focus moves up, more and more specific plugins are integrated into the controller and eventually become black boxes. Due to lack of internal details, performing access control inside these plugins is not advisable. Thus, performing access control on NBI is a suitable design, which adapts to the evolution of NBI and makes the system compatible.

2.2. Related Work

We divide the related work into the following categories, traditional access control, access control in SDN, access control in cloud, policy-based work, and policy language, and then compare our work to these work from above five aspects.

Traditional Access Control. Many mature traditional AAA (Authentication, Authorization, and Accounting) protocols are widely applied to perform the access control on the users' requests. For example, RADIUS (Remote Authentication Dial-In User Service) [28] is a networking protocol, which provides centralized AAA management, and Diameter protocol [29] extends the RADIUS by adding new commands and attributes. However, RADIUS does not support attribute addressed policies. Similarly, although Diameter provides the ability of attribute-based control, most of Diameter commands and attributes are predefined, which makes Diameter do not meet the requirement of resource management in the SDN-based cloud.

Access Control in SDN. Most of research are carried out around the architecture of SDN. Works [13, 14, 15] perform access control on the southbound interface and the data plane. Another work SDNShield [16] performs access control on the plugins inside the controller by modifying plugins' codes. However, due to lack of the intention of users, these work can not control user behavior. Works [30, 31, 17, 18] design new controller architectures and APIs to prevent malicious operations, which weaken the applicability and flexibility of the controller. Another related concept is Access Control List (ACL) in network field, PGA[32], FlowGuard[33] are designed based on ACL. These works are able to control packet routing in the existing network, but can not control the network configuration in the SDN-based cloud, as well as the requests accessing the controller via the NBI.

There are a few works focusing on the northbound of SDN. AAA [11], a project of OpenDaylight controller [34], can realize basic user authentication and authorization. The prototype of SDNKeeper is also implemented in the OpenDaylight controller. Comparing to SDNKeeper, *Fluorine*, the latest version of AAA, can filter access requests at the coarse user granularity, while unable to make decisions according to the attributes of the requester, resource, and environment. For instance, AAA can allow a user Bob to create a resource "network" based on his identity but cannot make the decision in a finer attribute granularity, like allowing the user Bob to create a network of type VLAN before December 31, 2018. Similarly, in order to perform role based access control, works [12, 17, 18] develop a more secure authentication mechanism to verify user's identity. But these works will loss effectiveness if the registered users are compromised to perform malicious operations.

Access Control in Cloud. Prior access control mechanisms in cloud [35, 36, 37, 38, 39, 40] are in allusion to protect data security and user privacy, but do not provide effective protection for SDN. However, works [41, 42] are based on SDN, which can control the network and detect attacks within and between clouds, but they do not consider security issues from the perspective of the SDN controller.

Policy-based Work. Policy-based methods are widely

applied, and some of them are proposed for traditional network [43, 44, 45], which are not suitable for SDN, since the managed resources and the application scenarios are different. However, after the emergence of SDN, many policy-based frameworks are designed to manage and secure the SDN network ([46, 47, 48]). They either (1) defend against covert channel attack by resolving rule conflicts and preventing the conflicts installed in the SDN data plane [46], or (2) design security policies to secure end hosts and defend against attacks related to the data flow and path in the data plane [48], or (3) control the access time to the network services and control the access to the switches based on users' roles [47]. Compared to these papers, our work focuses on protecting and managing resources in the controller, and performs fine-grained access control on the access requests based on the attributes of the requester, resource, and environment.

Policy Language. Due to the expansion of scale and increased management complexity, a variety of policy-driven languages expressing the intention of administrators are proposed to manage and secure the network [49]. These languages focus on either network management or security management, for instance, PCIM [50] as an object-oriented information model can specify various policies for traditional networks in general, such as configuration policies, installation policies, error and event policies, and security policies, etc. XACML [51] is also a general attribute-based access control system for evaluating access requests. Compared to these traditional works, in the scenario of the SDN-based cloud, we simplify the specification of policies, which is easy to grasp and use for administrators, and design a policy language with considering the specific properties of SDN, such as centralized resource management and programmability of the network. Besides, SDNKeeper can work as a plugin inside the SDN controller, which makes it easy to implement and deploy, as well as compatible with other modules.

3. SDNKeeper Overview

In this section, we first describe two application scenarios of SDNKeeper, following by a bird's-eye view of the whole system and a macroscopic description of the core components. To better understand the mechanism of SDNKeeper, we will demonstrate the workflow within the architecture shown in Fig. 3.

3.1. Threat Model

In our threat model, the attacker is a malicious user with authenticated identity. We assume the attacker has two ways to access the SDN controller. One is sending requests to the SDN controller via applications. The other is accessing the SDN controller directly through the URI, as shown in Fig.3. Besides, there is another type of attacker who can hijack and tamper with requests sent from applications to the SDN controller. The goal of such an attacker is to fetch the information beyond his scope to infer

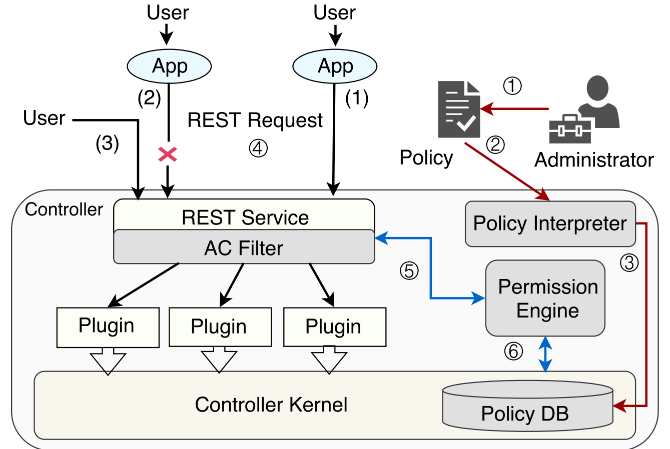

Figure 3: Architecture and workflow of *SDNKeeper*.

the global status based on this information and attempt to modify or delete the resources outside his scope.

To concentrate on demonstrating the core design of SDNKeeper and to clarify our work, we make two assumptions here: **1) SDN controller**, which performs permission checking, **is safe**. That means the controller is secure enough and running correctly with no bugs. We are confident because a large body of research work and troubleshooting techniques ([52, 53, 54]) make the SDN controller safer and more robust. Therefore, our work mainly focuses on the legitimacy of the access requests and the security of resources. **2) The administrator**, who specifies policies, **is credible**. The administrator has the highest authority such that he can configure the whole network by making policies and executing commands. Hence, attacking network by hijacking administrators is not within the scope of this paper.

3.2. Application Scenario

Based on the threat model, we summarize two application scenarios of SDNKeeper as follows. Here we regard resources as all plugins inside the controller, as well as network resources managed by the controller, including flow tables, statistics and devices, etc.

Scenario 1: Protecting resources. The requests accessing resources can be divided into three categories as described in Section 2.1, also shown in Fig. 3. Request (1) comes from the registered user and application, which carries illegal information. Request (2) is also sent from benign user and application, but it was tampered with halfway. And request (3) is sent directly to the controller by the registered user. All these requests are dangerous to the controller and put the resources at risk of being tampered with. SDNKeeper is designed to intercept these malicious requests.

Scenario 2: Managing resources. There are various plugins inside the controller providing a large number of APIs for upper layer applications, as depicted in Fig. 1

(b). SDNKeeper provides a unified entrance to manage resources, such as controlling which resource can be deleted or which resource can be queried in a fine-grained manner. Administrators just need to insert specific policies in SDNKeeper to achieve the goal of managing resources in the controller and the data plane.

3.3. Architecture of SDNKeeper

In the SDN-based cloud, the fundamental network service is provided by SDN. Taking a typical application scenario as an example, as depicted in Fig. 1, cloud communicates with SDN controller through REST API (i.e. NBI) provided by plugins inside the controller. Meanwhile, REST Service, as the unique northbound channel in the SDN controller, processes all requests sent from cloud. Hence, our key idea of protecting and managing network resources in the SDN-based cloud is to perform access control at the NBI level.

SDNKeeper is an attribute-based access control system, which can perform access control based on the attributes of the requester, resource, and environment. For instance, attributes can be user role, user name, resource name, resource type, requested action, and legal operating time, etc. Since SDNKeeper runs at the northbound of the controller, all illegal requests from northbound will be blocked outside the controller. However, it cannot be applied to prevent malicious messages from the southbound.

In general, SDNKeeper as a fine-grained policy enforcement system provides real-time protection and permission checking for the SDN controller. Specifically, SDNKeeper allows administrators to design policies based on the global view of the whole network. No matter which application the access request comes from, it will be rejected if it violates the policies.

In our design, SDNKeeper mainly consists of two parts, policy interpreter and permission engine. After the administrator defines policies in policy language based on the global view and security demands, these policies will be issued to the controller. Then, **policy interpreter** will parse and transform the semantic policies into tree-structured data and store them in the data store of the controller.

Permission engine is the core component, which enforces permission checking based on the policies defined by the administrator. SDNKeeper can be regarded as a filter between the SDN controller and upper applications. During the lifetime of the controller, permission engine keeps mediating all access requests at the NBI level continuously. Permission engine also supports runtime policy modification, providing the flexibility of access control.

The complete access control workflow of SDNKeeper is shown in Fig. 3 and described as follows.

1. The administrator first defines the policies according to current global view and security demands (step ①), and then issues these policies to the controller (step ②).
2. The policy interpreter parses and transforms the semantic policies into formalized structural data, which are controller-identifiable and SDNKeeper-processable. Parsed policies are stored in the data store (step ③).
3. When the controller receives a REST Request (step ④), the filter in REST Service will intercept and send this request to permission engine (step ⑤).
4. Permission engine checks the required operation with policies stored in the data store (step ⑥). If the request violates the policy, permission engine will reject it along with response massages.

Generally, in the original SDN system, all requests sent from various applications are directly loaded into the controller without checking for the legitimacy and correctness of the requests at the attribute level. Thus, malicious requests can strike the system without any obstruction. Although several security inspection techniques ([55, 56, 57]) are presented for the safety of applications, they can only work offline and are unable to ensure the legality of the requests sent to a running plugin. In SDNKeeper, the controller can not only avoid infringement caused by malicious requests, but also save precious resources to efficiently process benign requests and provide real-time protection for the system. We will describe the design of policies, and the details of policy interpreter and permission engine in Section 4 and Section 5, respectively.

4. Policy Language and Policy Design

In this section, we will expound what the policy is, how to manage policies in SDNKeeper and how to write a policy for the administrator.

4.1. REST Request

REST API[2] is the most common NBI for users to access network resources in SDN. Almost all SDN controllers, like OpenDaylight [34], Floodlight [58], ONOS [59] and Ryu [60], support REST API, and recommend or require using REST Request to access network resources at the northbound of SDN. A REST Request consists of four main parts as shown below.

1. `Method` defines the HTTP verbs a requester intends to perform. The most common HTTP verbs are `POST`, `GET`, `PUT` and `DELETE`, which correspond to create, read, update and delete operations, respectively.

2. `URI` identifies the network resource provided by the controller. Typically, plugins register their `URIs` in the REST Service. Taking the *RESTConf* [61], the REST Service in the OpenDaylight controller, for

[2] REST API: **RE**presentational **S**tate **T**ransfer **A**pplication **P**rogramming **I**nterface, which allows the requester to access and manipulate resources using a uniform, stateless operation over HTTP.

an example, the plugin should first resister the URIs, which are used to identify its resources, to the *REST-Conf*. Then, users can query their resources by combining the GET verb and the URI registered previously.

3. Headers carry a list of information in the HTTP request, such as the content type of this request and the authorization token of the requester.
4. If a requester requests to create (POST), update (PUT) or delete (DELETE) a resource, a JSON body with detailed attributes of this resource should be included.

With the information carried in the REST request, a policy can be created to perform fine-grained access control on the requests, which are sent by users to access network resources in the controller and the data plane. The following is a typical REST Request example.

A Typical REST Request Example

```
1) Method: POST  (POST/GET/PUT/DELETE)
2) URI: https://<controller-ip>:<port>/networks/
3) Headers: {
     Content-Type : application/json,
     Authorization : {
        Username : Alice, Password : *** },
     ... }
4) Body (optional): {
     network : {
        name : alice-network,
        tenant_id : 9bacb3c5d39d41a7951...,
        subnets : [],
        network_type : vlan,
        ... }}
```

4.2. Policy

A policy in SDNKeeper is designed to determine whether to approve or decline a REST request. In order to defend against unauthorized requests, the policy needs to clearly describe the details of the requester, requested resource and environment. Thus, we formulate a resource access control policy (P) into three terms: Subject, Object and Environment, which can cover all the information contained in a REST Request.

```
P(S,O,E) := (ATTR(S) op ATTR(O) op ATTR(E))
```

- Subject (S) is a requester, usually means a user who issues access requests to the controller (*Headers: Authorization*). Its attributes (ATTR) are the information related to the users, like username and role type.
- Object (O) is the requested resource provided by the controller, such as networks, firewalls and routers, etc. (URI). All the context in the Body part of the REST request are the attributes of this Object.
- The system Environment (E) is also an important aspect we should consider. For example, date is a crucial environment attribute in the lease of a network resource. A user cannot use resources after the lease expires.

We predefined a data structure to fetch the attributes of Subject, Object and Environment:

```
predefined
  :  'subject.' ('role' | 'user')
  |  'action.'  ('uri'  | 'query' | 'method' )
  |  'environment.' ('date' | 'time' | 'week')
```

For instance, Subject's attributes such as role and username can be obtained in the format like *subject.role* and *subject.user*. For Object, action attributes can be fetched in the format like *action.uri* and *action.method*. Similarly, query string for GET verb can be obtained by using *action.query*. As the same, *environment* data structure represents the system date and time in the controller.

In addition, we can refer to JsonPath syntax to fetch the attributes in the Body part:

```
jsonpath : '$.' string ('.' string)*
```

For example, *$.network.type* can get the type of the network. Therefore, with our predefined data structure, network administrators can get any information from the REST request and customize arbitrary access control policies according to our predefined data structure.

Each policy is a set of assertion expressions combined with the iteration of *if-statement*s and *AND/OR* operations, and will eventually return a value of *ACCEPT* or *REJECT*:

```
policy    : policy_name '{' statement '}'
statement : 'ACCEPT' |  'REJECT' | if_state
if_state  : 'if (' expr ')' statement
            ('else' statement)?
```

Below, we show an example of a policy which follows the policy language syntax. This policy is called "*Bob_can_post_vlan*". With the first *if-statement*, a REST request from user Bob will hit this policy. Under the assertions in the second *if-statement*, Bob can create a *network*, if the type of the *network* is *VLAN*.

A Policy

```
1 Bob_can_post_vlan{
2   if (subject.user == 'Bob') {
3     if (action.uri REG '/networks/' &&
4         action.method == 'POST' &&
5         $.network.type == 'vlan') {
6       ACCEPT }}}
```

4.3. Policy Hierarchy

SDNKeeper classifies the policies into two categories, global policy and local policy:

```
policySet   : globalSet? localSet? ;
globalSet   : 'GLOBAL_POLICY {' policy* '}'
localSet    : 'LOCAL_POLICY  {' localPolicy* '}'
localPolicy : role. (user)? '{' policy* '}'
```

- **Global policies** are intended for all requests. When a request comes in, it will be checked against all the global policies.
- **Local policies** are only intended for individual user group and user, which have user-related attributes: *role* and *username*. When a request from a certain user comes in, only the relevant local policies with the matching *role* and *username* will be checked.

There are two reasons for designing these two separated policy sets. One is *for performance*. Permission engine only needs to check global policies and relevant local polices. This will greatly reduce the policy checking burden when the policy set is large. And the other more important reason is *for expressiveness and simplicity*. Administrators can make group policies to manage requests in batches according to specific requirements, as well as make individual policies for particular users to control their resources.

A Policy File

```
1 GLOBAL_POLICY {
2   system_update {
3     if (environment.time > 1am &&
4         environment.time <  6am ) {
5       REJECT }}}
6 LOCAL_POLICY {
7   user {
8     user_can_get_on_monday {
9       if (action.method == 'GET') {
10        if (environment.weekday == 'mon') {
11          ACCEPT }}}}
12  user.Alice {
13    alice_cannot_delete_firewall {
14      if (action.uri REG '/firewalls/') {
15        if (action.method == 'DELETE') {
16          REJECT }
17        else {
18          ACCEPT }}}
19      ... }}
```

In order to have an intuitive understanding, we provide an example of a policy file as shown above, including global policies and local policies. For user "Alice" with "user" role, her REST requests will be processed by global policy *system_update*, local policy *user_can_get_on_monday* and local policy *alice_cannot_delete_firewall*. However, for user "Bob" with "user" role, his requests will be checked with only two policies, global policy *system_update* and local policy *user_can_ get_on_monday*.

SDNKeeper's policy language syntax is summarized in Appendix Appendix A.

4.4. Policy Generation

To make SDNKeeper work correctly in the SDN-based cloud, there are only two steps for generating policies. First, the REST APIs and related attributes of the requester and resources should be provided to the network administrator. Second, the administrator defines access control policies based on these attributes and following the description in Section 4. Since these policies are Json-based rules, it is easy to generate policies just follow the grammar of Json.

Before SDNKeeper is running, administrators first need to summarize the characteristics of attacks, security demands and system restrictions, such as pointing out the resources which can be accessed by users, specifying the resource attributes which need to be checked, and setting an available service time, etc. These characteristics will be further used to create global policies and local policies. When a new user joins, administrators only need to assign this user with a corresponding role (new or existed). The authority of this user will follow the policies described by the predefined global and local policies within this role. In addition, the administrator can also create specific policies for the particular user by adding a new local policy for this user on the fly.

5. Policy Interpreter and Permission Engine

In this section, details of policy interpreter and permission engine are introduced, as well as the mechanism of REST request processing and permission checking.

5.1. Policy Interpreter

The human-language based policies need to be translated into the computer-processable data structure. In the policy interpreter, abstract policies issued by the administrator are parsed into a semantic tree, which is loaded into the controller's memory. Intuitively, Fig. 4 shows the semantic tree of a global policy set. In the semantic tree, each leaf node represents an attribute or a comparing value and other nodes represent logical operators. Thus, each expression can be expressed by a subtree. After recursively evaluating the left children and right children, we can get the value of the root node, *i.e.*, the result of permission checking.

Matching in the semantic tree is very fast. Our evaluation in Section 7.2 shows that the matching time will not be significantly affected after we quadruple the total number of policies.

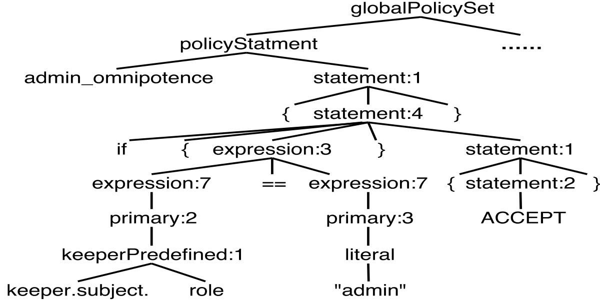

Figure 4: Semantic tree of global policy.

5.2. Permission Engine

Each request issued by users will be checked by the permission engine. Generally speaking, permission engine 1) extracts attributes of request, such as *user*, *uri* and *method*, 2) evaluates this request by checking it against policies in the data store, and 3) finally makes a decision on approving or declining this request. We highlight several issues in permission engine design as follows.

Policy Conflict. Because of the intersection of different policies, a REST request may be approved by one matched policy but rejected by another matched policy, which brings a policy conflict. As shown below, if user Alice requests to `GET` the network resource. Her demand will be approved by *all_can_get* policy in global policy set. However, Alice does not have the permission to access the network resource as described in local policy *net_reject_alice*. Therefore, it will be inaccurate if we return the decision once the policy is matched.

A Policy File with Policy Conflicts

```
1 GLOBAL_POLICY {
2   all_can_get {
3     if (action.method == 'GET') {
4       ACCEPT }}}
5 LOCAL_POLICY {
6   user.Alice {
7     net_reject_alice {
8       if (action.uri == '/networks/') {
9         REJECT }}}}
```

For the sake of security, we introduce full match strategy in the permission checking process. A REST request is checked in the order of global policies, group local policies, and user local policies. If a matched policy returns the checking result of "REJECT", permission engine will decline this request immediately. If the checking result of the matched policy is "ACCEPT", the permission checking process will go on until all policies have been checked or get the result of "REJECT". After all policies have been checked and matched an "ACCEPT" decision, this request should be approved. If no policy is matched by this request, this request will be declined. The complete permission checking process is illustrated in Algorithm 1.

Algorithm 1: Permission Checking

Input : *request*
Output: $ACCEPT$ or $REJECT$
1 $approved \leftarrow false$
2 $policy_set \leftarrow \{Global, Local[role], Local[role][user]\}$
3 **for** policy ***in*** policy_set **do**
4   **if** *request* **matches** policy **then**
5   **if** policy.eval(request) $== REJECT$ **then**
6   **return** $REJECT$
7   **else** $approved \leftarrow true$
8 **If** $approved == true$ **return** $ACCEPT$
9 **return** $REJECT$;

Filter Based. Permission engine acts as a filter between the application plane and the control plane. Therefore, illegal requests will be rejected before reaching the relevant modules inside the controller, which will not occupy the computational resources of the controller. Filter based design can also bring benefits to deployment. Typically, controllers have a REST Service module for receiving and distributing REST requests. It will only need a few code changes when adding a new REST filter to the REST Service module. In most mainstream SDN controllers, like OpenDaylight controller and ONOS controller, we can enable SDNKeeper in them by adding several dependencies to the configuration files.

Runtime Configuration. Since administrators may need to refine policies dynamically according to the security and management demands, runtime configuration is an important feature for permission engine. In SDNKeeper, administrators are allowed to access and update the policies in the data store, where a listener is registered, at any time. Once an insert/delete/update operation occurs, the listener will send a notification to the permission engine. And the permission engine will update the policies in the memory cache, so that subsequent requests will be checked by new policies.

6. Correctness Proof of SDNKeeper

SDNKeeper is the first protective barrier for network control and management, the correctness of its system design is critical and need to be proved and guaranteed. In the case that the system follows the designed workflow, returns the correct checking results, and is able to process requests continuously, we can confirm that the design of the system is correct. In this section, we will illustrate the correctness of the system design by modeling the whole system with Formal Methods (FM) and checking the necessary properties. In order to avoid missing some points which cannot be covered by the experiments, proving the correctness of the system design is necessary. On

the premise of ensuring the correctness of the design, we will show the implementation correctness by evaluating the whole system with sufficient experiments.

6.1. System Modeling

We first build a system model consisting of *Access Control Filter* and *Permission Engine* as shown in Fig. 5. Each sub-model is a finite state machine (FSM) with several states and transitions.

6.1.1. ***Access Control Filter (ACF)***

The *Access_Control_Filter* FSM has 5 states and 6 transitions. *ACF_Idle* state is the initial state and *ACF_Intercepted* state represents that there is a new request to access the controller. After sending the request to the *Permission Engine*, *Access_Control_Filter* FSM will go to *ACF_Waiting* state. Depending on the checking result received, the *Access_Control_Filter* FSM will reach the *ACF_Received_AC_Processing* state if the checking result received is `accept`, and will transfer to the *ACF_Received_REJ_Block* state if the checking result received is `reject`.

6.1.2. ***Permission Engine (PE)***

The *Permission_Engine* FSM has 4 states and 11 transitions. Similarly, *PE_Idle* state is the initial state. After receiving requests, the *Permission_Engine* FSM will go to *PE_Checking_Global_Policy* state to check global policy set. Following that, the *Permission_Engine* FSM will reach *PE_Checking_Local_Policy* state to check all local policies after all global policies have been checked with. In this progress, once there are matched policies with the decision of `reject`, the checking_result will be returned via the Channel_PA, at the same time the checking process in the PE is over. Otherwise, no matter the checking_result is `accept` or `unmatch`, which means the request and policy do not match, the checking process will continue until all policies have been examined and reach the *PE_Check_All_Policies* state.

Policy Interpreter is responsible for parsing high-level abstract policies into tree structure policies and pushing them into the data store. It plays a role of providing policies for permission checking in the whole system, and has no interactions with above two core components *Access Control Filter* and *Permission Engine*. So it does not impact the correctness analysis of the system design. Therefore, the model of *Policy Interpreter* can be omitted.

6.2. Model Checking

We choose NuSMV, a state-of-the-art symbolic model checker, to perform model checking. Its latest version, NuSMV2 [62], is based on the powerful satisfiability (SAT) engine to achieve good scalability and efficiency. Users can specify synchronous or asynchronous finite state models in an intuitive fashion. The properties to be checked can be described as linear temporal logic (LTL) specifications. Given a certain number of rounds, NuSMV model checker either provides a counterexample to the property, or concludes that the property is satisfied by the corresponding model. Table 1 lists the LTL operators we used in the model checking.

Table 1: LTL operators used in model checking.

Operator	Intuitive meaning
G	globally
F	eventually
O	once
X	next state
Y	previous state

In order to prove the correctness of the system design, we check all critical properties based on the model built above. The key point is the selection of properties. The main function of *Access Control Filter* is intercepting access requests and sending them to *Permission Engine*, then waiting to handle the request until receiving the checking result. Thus, the properties of *Access Control Filter* should satisfy this workflow and is able to make the system process requests uninterruptedly. Since the workflow of *Permission Engine* is more complex, its properties should not only respect its workflow, but also describe its key features to ensure the *Permission Engine* work smoothly. The detailed descriptions of each property are illustrated as follows and the corresponding formal statements are shown in Table 2.

Property 1. If ACF FSM is in the *ACF_Idle* state, it will eventually return to the *ACF_Idle* state.

This property can ensure the *Access Control Filter* can come back to its initial state, that is to say, it can process access requests continuously. It is important for the network in the case of a large number of requests accessing the controller.

Property 2. ACF FSM must go through *ACF_Intercepted* state and *ACF_Waiting* state if the ACF FSM gets the checking_result from PE.

Access Control Filter intercepts access requests first and is in a waiting state until receiving the checking result from *Permission Engine*. This is the normal processing flow indicating that the *Access Control Filter* is in a normal working state.

Property 3. If ACF FSM reaches *ACF_Waiting* state, *ACF_Intercepted* state must be its last state before that.

Similar to property 2, this property also describes the workflow of *Access Control Filter*, but is more precise. Fine-grained property checking is necessary because we need to ensure every step of this ACF FSM respects to the system design.

Property 4. If there is a request coming in, the ACF FSM must reach either *ACF_Received_AC_Processing* state or *ACF_Received_REJ_Block* state.

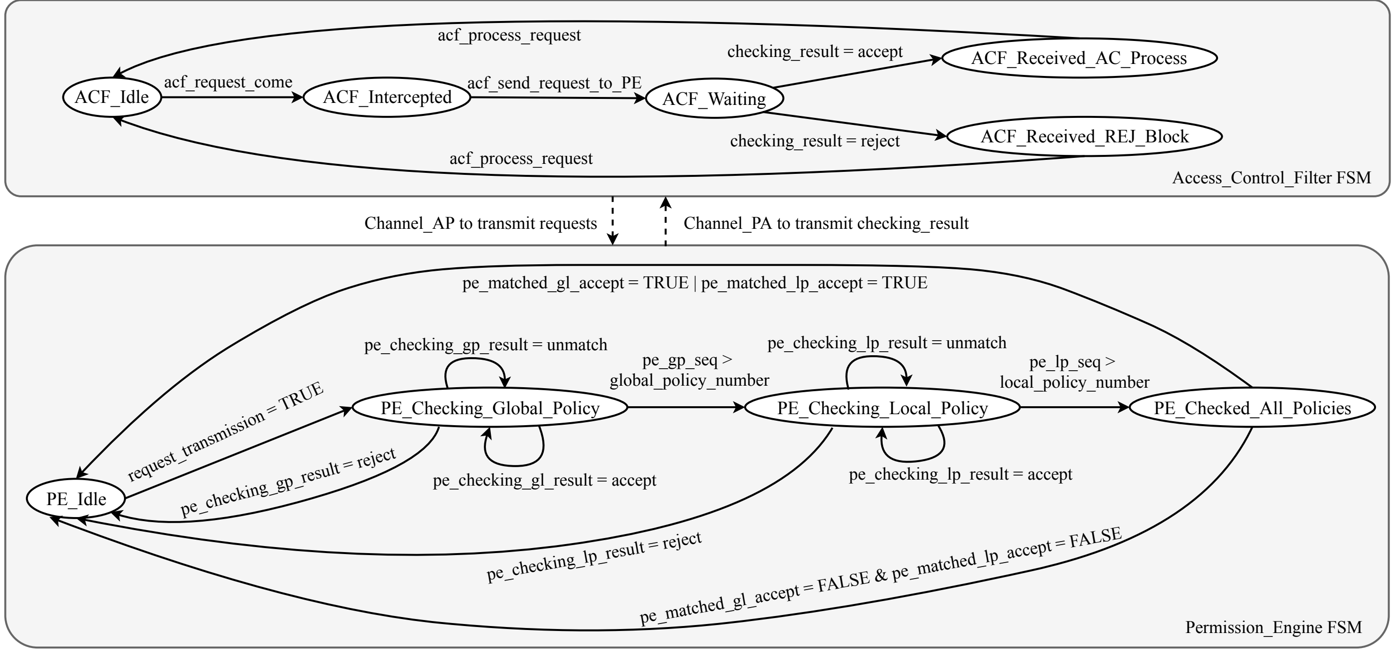

Figure 5: SDNKeeper system model consisting of *Access_Control_Filter* FSM and *Permission_Engine* FSM.

Once there are requests accessing the controller and being intercepted by the *Access Control Filter*, the checking result must be returned, either `accept` or `reject`, which means the request is benign that can be processed further, and the request is illegal that need be blocked outside the controller respectively. This property can guarantee every request is checked by *Permission Engine*, then processed based on the checking result.

Property 5. If PE FSM is in the *PE_Idle* state, it will eventually return to the *PE_Idle* state.

In order to cooperate with *Access Control Filter*, *Permission Engine* should also be able to perform access control on the received requests continuously. No matter how many requests there are, it can return the checking result to *Access Control Filter* and come back to the initial state.

Property 6. For PE FSM, the *PE_Checking_Global_Policy* state must be reached before *PE_Checking_Local_Policy* state.

Benefit from the design of separated policy set, checking requests achieves high performance, since only global policies and user-related local policies need to be checked. What's more, global policies have higher priorities than local policies, thus, all global policies must be checked before any local policies.

Property 7. For PE FSM, if and only if all policies have been checked and either "pe_matched_gp_accept = TRUE" or "pe_matched_lp_accept = TRUE", the checking_result is set to `accept`.

To protect the controller effectively in case of policy conflict, we propose an elaborate design, which accepts a request if and only if all global policies and user-related policies have been checked with and the checking result is `accept`. This can ensure every request processed by the controller is benign without being impacted by policy conflict.

6.3. Discussion

We build system model consisting of two core components (*Access_Control_Filter* FSM and *Permission_Engine* FSM) and perform model checking on its critical properties, which are described formally. Based on the design principles of SDNKeeper, we propose and check all function-related properties above, which are important for making the access control system work correctly and smoothly. As depicted in Fig. 5, there are 17 transitions in the whole system model, and we run the model checker for 7 times, each time checks one property, which have covered all possible situations. The model checking results show that there are no counterexamples found, which means these properties are all satisfied. Based on the basic thought[3] of formal methods for proving the correctness of the system design, we can prove that the design of SDNKeeper is correct.

7. Implementation and Evaluation

There are two major components in the prototype of SDNKeeper: 1) **policy interpreter** parses semantic policies into semantic trees, and 2) **permission engine** performs permission checking on each coming request. In this

[3] After exhaustively checking all properties for enough times which cover all possible changes in all states and transitions, we can conclude that the system design is correct.

Table 2: The formal statement of properties for model checking.

<table>
<tr><th>FSM</th><th>Property</th><th>Formal Statement</th></tr>
<tr><td rowspan="4">ACF</td><td>Pro 1</td><td>G ((ACF.state = ACF_Idle & X (ACF.state != ACF_Idle)) -> F (ACF.state = ACF_Idle))</td></tr>
<tr><td>Pro 2</td><td>G ((checking_result = ACCEPT | checking_result = REJECT)
-> O (ACF.state = ACF_Intercepted & ACF.state = ACF_Waiting))</td></tr>
<tr><td>Pro 3</td><td>G (ACF.state = ACF_Waiting -> Y (ACF.state = ACF_Intercepted))</td></tr>
<tr><td>Pro 4</td><td>G (acf_request_come
-> F (ACF.state = ACF_Received_AC_Process | ACF.state = ACF_Received_REJ_Block))</td></tr>
<tr><td rowspan="3">PE</td><td>Pro 5</td><td>G ((PE.state = PE_Idle & X (PE.state != PE_Idle)) -> F (PE.state = PE_Idle))</td></tr>
<tr><td>Pro 6</td><td>G (PE.state = PE_Checking_Global_Policy
-> !O (PE.state = PE_Checking_Local_Policy))</td></tr>
<tr><td>Pro 7</td><td>G (check_result = ACCEPT
-> O (pe_matched_gp_accept | pe_matched_lp_accept)
& (pe_gp_seq > global_policy_number | pe_lp_seq > local_policy_number))</td></tr>
</table>

section, we first introduce the implementation of these two components, as well as the implementation of model construction and model checking. Then, we evaluate the performance of SDNKeeper from three metrics: **effectiveness**, **latency** and **throughput**, and briefly discuss the results in the end.

7.1. Implementation

We implement SDNKeeper as a plugin of the OpenDaylight [34] controller, which works as a filter to control any REST request from upper applications to the controller, and implement policy interpreter and permission engine as the controller-independent Java bundles. Currently, almost all mainstream controllers (*OpenDaylight, ONOS*) use REST API in northbound communication. This enables SDNKeeper to perform access control on these controllers. Benefit from the lightweight feature of SDNKeeper, the deployment of the whole system is low cost, which only needs to embed SDNKeeper into the controller as a feature and assign it with a high priority to filter REST requests first.

SDNKeeper is an attribute-based access control system, in which role is an important attribute of the requester for checking permission and making decisions. Since a user authentication module (AAA[11]) has already been developed, we liberate ourselves from repetitive work. Owing to the filter-based feature, SDNKeeper is compatible with other projects, so that the permission engine of SDNKeeper can be inserted behind AAA, then the checking progress is based on the authenticated result of AAA.

The two main components and system model checking are implemented as follows.

7.1.1. ***Policy Interpreter***

Policies defined by administrators are the JSON-based, human-readable rules. Policy Interpreter compiles these semantic policies into semantic trees. We implement a CLI command *SDNKeeper:load/reload* in Karaf console to load all semantic policies into the data store of the controller. In this progress, ANTLR [63], a language recognition tool, is responsible for reading and parsing semantic policies continuously, then a registered listener will insert the policy tree into the data store once a new one is loaded. Finally, all policies will be stored as a tree, so that the permission engine just needs to recursively traverse a tree to enforce a policy.

7.1.2. ***Permission Engine***

Permission Engine is the core component checking REST requests based on policies defined by administrators. In the real-world scenario, REST requests sent to the controller are usually highly concurrent. In order to adapt to this characteristic, we adopt Akka [64] to process multiple requests simultaneously by creating a certain number

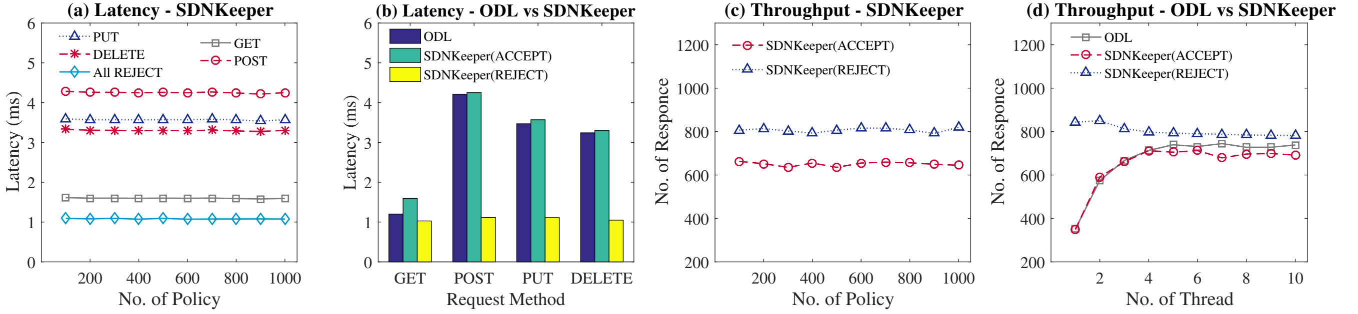

Figure 6: *Evaluation Result*: ***(a)*** latency of SDNKeeper with different numbers of policies, ***(b)*** latency between original and SDNKeeper-enabled OpenDaylight with 1,000 policies, ***(c)*** throughput of SDNKeeper per second with different numbers of policies under 2 threads, ***(d)*** throughput between original and SDNKeeper-enabled OpenDaylight with different numbers of threads.

of *Actors*. Making full use of controller's computing resources helps us achieve high system performance, i.e., low processing latency and high processing throughput.

What's more, *request queue* and *response queue* are designed for caching access requests and check results respectively to mitigate the congestion of requests. *Policy cache* is designed for accelerating the process of policy matching. With the *Policy Data Store Listener* in permission engine, the policy cache can be updated at runtime once there are policy changes in the data store. And the new policies will take effect on subsequent requests. In order to assist the administrator to checkout whether the new policies are effective, we implement a CLI command *SDNKeeper:cache*, which can be executed in the Karaf console to get the policies in the cache.

7.1.3. ***Model Checking***

To perform model checking on the whole system, we first construct system model consisting of two finite state machines, *Access_Control_Filter* FSM and *Permission_Engine* FSM. The model has a total of 9 states and 17 transitions and is built in XML with 500+ lines of code referred to LTEInspector [65]. Then, we express specifications in Linear Temporal Logic (LTL), which characterizes each linear path induced by the finite state machine, and check the system model against 7 properties with the aid of NuSMV [62].

7.2. Evaluation

7.2.1. ***Methodology***

We establish the testbed of SDNKeeper on the mainstream SDN controller OpenDaylight (Intel i7-7700 8x3.6 GHz, 16 GB Memory, 4 CPU cores), and choose Neutron Northbound [66], a component enabling communication between Opendaylight and Openstack [67], as our test application. Neutron Northbound provides 30 kinds of REST APIs, ranging from networking, firewall, QoS to load balance, with 185 kinds of requests (GET, POST, PUT, DELETE) and 664 related attributes, which are enough to evaluate the effectiveness of SDNKeeper.

In our evaluation, users send REST requests to OpenDaylight through REST API. SDNKeeper performs access control on these requests at the NBI level. We examine the check results of those requests in the controller and the response received by users to evaluate the performance of SDNKeeper.

We first evaluate the effectiveness of SDNKeeper, i.e., whether SDNKeeper can reject unauthorized requests correctly. Then, we measure the extra processing latency introduced by SDNKeeper and REST request throughput. Finally, we evaluate the computational overhead of SDNKeeper with different numbers of requests. Both measurements are conducted in the controllers with and without SDNKeeper. If an illegal request is rejected by SDNKeeper, the processing time and resources occupancy would be largely reduced. Hence, for the sake of fairness, we evaluate the performance of SDNKeeper in cases where all decisions are “ACCEPT” and all decisions are “REJECT”.

Table 3: REST API provided in Neutron Northbound

Type	# of APIs	# of Attributes
Networking	6	220
Firewall	3	83
Security	2	24
VPN	4	104
SFC	4	60
Meter	2	13
QoS	2	31
Load Balance	4	81
BGP VPN	1	22
L2 Gateway	2	26

7.2.2. ***Effectiveness Evaluation***

In order to evaluate the effectiveness of SDNKeeper, we design test cases corresponding to the three types of illegal requests mentioned in Section 3.2. Since these illegal requests have the same format, we simulate these requests by sending REST requests uniformly. Table 3 lists all types of REST APIs provided by Neutron. These APIs are representative to show the correctness of SDNKeeper in rejecting the unauthorized access requests in the SDN-based cloud.

When verifying the effectiveness of intercepting unauthorized requests, we send two kinds of illegal requests: 1) requests sent to access resources not belonging to the current user; 2) requests sent to perform extra operations on the resources owned by the current user. We create 2789 policies in 3 granularities: 30 policies for all kinds of APIs, 185 policies for all kinds of actions in the API, 664 policies for all kinds of attributes, and 1910 policies for all possible combinations of two attributes. Based on these policies, we create benign and illegal access requests. The benign requests are generated to make the requests able to pass the permission checking. And illegal requests are generated by setting incorrect values in some fields to make them violate one or more policies which need to be checked with. Thus, when these requests are sent to access the controller, they will be checked with the generated policies. If the request violates one or more policies, it will be rejected and returned to the requester, while if it passes all policies, it will be further processed by the controller.The results show that all these illegal requests are rejected by SDNKeeper.

7.2.3. ***Latency Evaluation***

In SDNKeeper, matching policies and checking permissions may introduce extra delay to the controller when processing a REST request. We evaluate this delay by measuring the latency in users from sending a request to receiving the corresponding response. Three experiments are performed in this part: 1) latency with different numbers of policies shown in Fig. 6 (a), 2) latency between controllers with and without SDNKeeper shown in Fig. 6 (b), 3) latency with different numbers of requests shown in Fig. 7 (a). For first two experiments, each test is executed 5 times with 30000 requests.

Fig. 6 (a) illustrates the processing latency with different numbers of policies. As we can see, in all of those four request categories, almost no latency increase is introduced when we increase the number of policies. The insignificant computation overhead mainly benefits from our design of storing policies in the semantic tree. What's more, the matching time will not be significantly affected after increasing the number of policies because of the design of policy hierarchy, only policies under specific users will be checked with.

Under the same scenario with 1,000 policies, we compare the latency between SDNKeeper-enabled OpenDaylight controller and original OpenDaylight controller. According to the access control mechanism, the policy decision will affect request processing time. Due to the processing time of the controller, decision "REJECT" will make the processing time shorter than original, while decision "ACCEPT" will introduce a little computational overhead. As shown in Fig. 6 (b), SDNKeeper with decision "ACCEPT" only introduces about 0.15 ms extra delay on average. And when request is "REJECT" by SDNKeeper, the latency is largely reduced about 0.17 ms, 3.10 ms, 2.36 ms and 2.19 ms in GET, POST, PUT, DELETE requests respectively. In practice, decision "ACCEPT" and "REJECT" are mixed to construct a blameless policy set, thus the extra delay which is introduced by SDNKeeper will be further reduced.

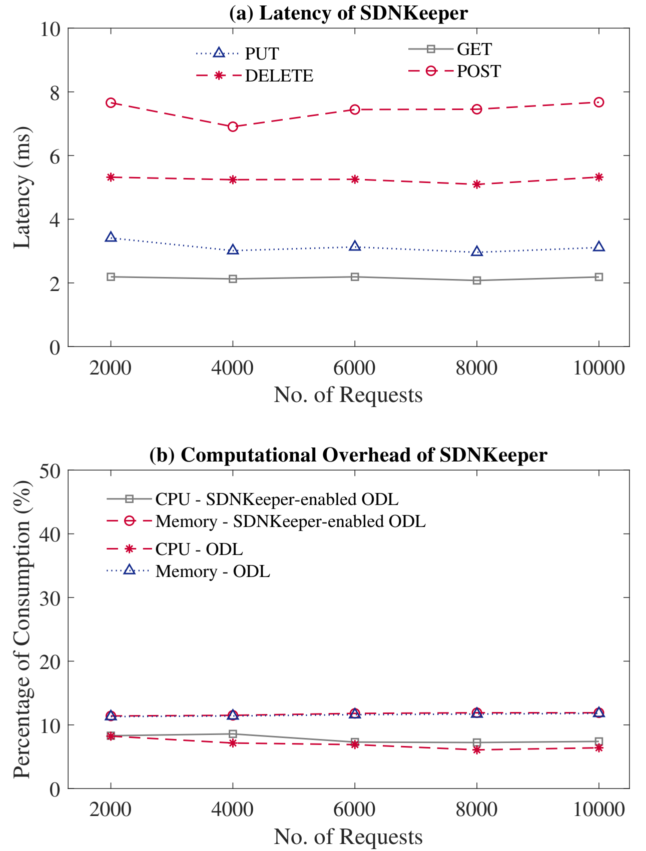

Figure 7: *Evaluation Result*: ***(a)*** latency of SDNKeeper with different number of requests, ***(b)*** computational overhead of SDNKeeper, which is evaluated with 8G of memory and 4 CPU cores.

We then evaluate the latency of SDNKeeper with different numbers of requests. As shown in Fig. 7 (a), when the number of requests increases from 2,000 to 10,000, there is no significant difference in the latency of each request, which means SDNKeeper can work smoothly when processing a large number of requests. In short, SDNKeeper has insignificant computational overhead for policy processing.

7.2.4. ***Throughput Evaluation***

We then evaluate the throughput of SDNKeeper. In the evaluation, we send a large number of REST requests to fulfill the capacity of the controller and measure the number of requests that can be processed per second, i.e., the number of received responses within one second.

As shown in Fig. 6 (c), no matter what the decision is, the performance of the controller is almost unchanged when we increase the number of policies significantly. This result is consistent with the result in latency evaluation.

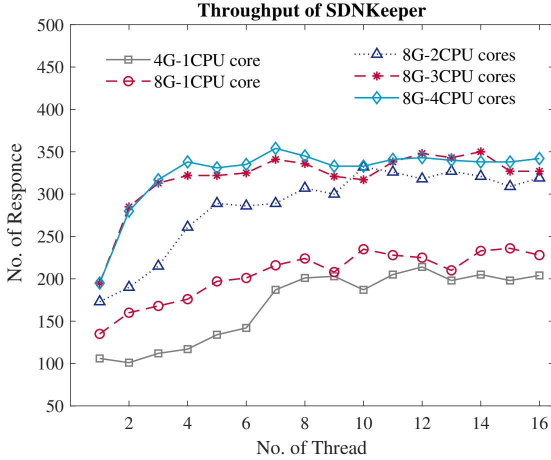

Figure 8: Throughput of SDNKeeper with the decision "ACCEPT" under different configurations.

The number of policies has negligible impact under our semantic tree design.

In Fig. 6 (d), we compare the throughput in the OpenDaylight controller with and without SDNKeeper. We vary the number of threads to test the processing capacity of SDNKeeper. From the results we can see that SDNKeeper with the decision "REJECT" always gets the best performance without being affected by the number of threads. While for both original OpenDaylight controller and SDNKeeper-enabled OpenDaylight controller with the decision "ACCEPT", the throughput varies with the number of threads. Based on our experimental environment, when the number of threads is greater than 4, the processing capability is no longer significantly affected by thread's number and close to the ideal. And the performance of SDNKeeper-enabled controller is almost as good as original OpenDaylight controller according to the evaluation results. We then compare the throughput of SDNKeeper under different configurations. As shown in the Fig. 8, with the same number of CPU cores, allocating more memory can make SDNKeeper achieve higher throughput. Similarly, under the same configuration with 8G of memory, increasing the number of CPU cores can improve the throughput when the number of CPU cores is less than 4. In short, SDNKeeper performs access control accurately with negligible effect on the processing capability of the controller.

7.2.5. Computational Overhead Evaluation

We evaluate the computational overhead of SDNKeeper based on the number of requests, particularly the CPU and memory consumed when processing different numbers of requests. Since SDNKeeper running on the OpenDaylight controller as a plugin, we evaluate the computational overhead by measuring the CPU and memory consumed by the original OpenDaylight controller and SDNKeeper enabled OpenDaylight controller. As shown in Fig 7 (b), SDNKeeper consumes approximately 8.19 M of memory when processing thousands of requests. Meanwhile, the average CPU consumption of SDNKeeper is about 28 millicores. For both controllers, the CPU consumption is at its highest when processing 2,000 requests. This is because the execution time of processing 2,000 requests is so short that the system has not reached a stable state. When there are a large number of requests being processed, the system maintains a stable CPU consumption. To summarize, the extra computational overhead introduced by SDNKeeper processing requests is negligible compared to the original SDN controller.

7.3. Discussion

Based on our correctness proof of the system design and model checking result in Section 6.2, we can prove that the system design of SDNKeeper is correct. Meanwhile, from effectiveness evaluation results we can conclude that the system implementation of SDNKeeper is correct. Compared to the southbound interface, the NBI is latency insensitive and infrequent. According to the evaluation results, 0.15 ms extra delay by SDNKeeper in NBI communication is acceptable. In practice, to guarantee the service quality, service providers usually limit the rate of access to the controller, which is smaller than throughput threshold. Furthermore, according to the evaluation results shown in Fig. 6 (d), the throughput of SDNKeeper enabled controller is very close to the throughput threshold of the original controller. Therefore, the throughput degradation introduced by SDNKeeper will not affect the response rate of access requests. To conclude, SDNKeeper can prevent unauthorized requests effectively with negligible impact on the performance of the SDN controller.

8. Conclusion

Having advantages of centralized control and programmability, SDN has been rapidly applied to the SDN-based cloud to provide the network services for upper applications. SDN controller acts as the brain is the most critical component for the whole network. Therefore, efficiently protecting and managing the resources inside the controller becomes an important issue. To address that, we propose SDNKeeper, a lightweight policy enforcement system, to assist administrators in protecting and managing resources. In addition, we design a policy language for administrators to define policies. With these fine-grained policies, SDNKeeper can perform access control for each request to defend against unauthorized attacks and avoid network misconfiguration. What's more, SDNKeeper is application-transparent and enable administrators to update policies on the fly. We adopt formal methods to prove the correctness of the system design. Then we implement the prototype of policy enforcement system and evaluate its performance. The results show that SDNKeeper can accurately block illegal requests outside the controller and work smoothly with negligible computational overhead and insignificant throughput degradation.

Appendix A. Policy Language Syntax

Policy Hierarchy

```
policySet : globalSet? localSet? ;
globalSet : 'GLOBAL_POLICY {' policy* '}'
localSet  : 'LOCAL_POLICY  {' localPolicy* '}'
localPolicy: role.(user)? '{' policy* '}'
policy    : policy_name   '{' statement '}'
```

Policy Statement

```
statement : '{' statement '}'
          | 'ACCEPT' | 'REJECT' | if_state
if_state  : 'if (' expr ')' statement
            ('else' statement)?
expr      : '(' expr ')'
          | expr lop expr | primary aop primary
          | true | false
lop       : '&&' | '||'
aop       : '>=' | '<=' | '>' | '<'
          | '==' | '!=' | 'REG'
```

Primary

```
primary   : predefined | jsonpath | literal
predefined: 'subject.'
              ('role' | 'user')
          | 'action.'
              ('uri'  | 'query' | 'method')
          | 'environment.'
              ('date' | 'time'  | 'week')
jsonpath  : '$.' string ('.' string)*
literal   : int | float | string | bool | null
```

"?" indicates 0 or 1 occurrences of the preceding element. "*" indicates 0 or more occurrences of the preceding element.

References

[1] X. Leng, K. Hou, Y. Chen, K. Bu, L. Song, SDNKeeper: Lightweight Resource Protection and Management System for SDN-based Cloud, in: 2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS), IEEE, 2018, pp. 1–10.

[2] E. Haleplidis, K. Pentikousis, S. Denazis, J. Salim, D. Meyer, O. Koufopavlou, Software-defined networking (SDN): Layers and architecture terminology. RFC 7426, IRTF. (2015).

[3] A. Greenberg, SDN for the cloud, in: Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication, 2015.

[4] Microsoft Azure and Software Defined Networking, accessed on 2018-11-8.
URL https://goo.gl/t2QVUm

[5] IBM Network Services for Software Defined Networks, accessed on 2018-11-8.
URL https://goo.gl/xP6cLh

[6] Google cloud platform, accessed on 2018-11-8.
URL https://goo.gl/B2fMfJ

[7] CloudFabric, a SDN-based data center developed by Huawei, accessed on 2018-11-24.
URL https://goo.gl/mp9E9J

[8] NovoDC, a SDN-based data center developed by China Mobile, accessed on 2018-11-24.
URL https://goo.gl/pdktxv

[9] Synergy Research Group, accessed on 2018-11-8.
URL https://goo.gl/f7yTH9

[10] I. D. Corporation, A report on datacenter by idc, accessed on 2017-07-31.
URL https://goo.gl/ZLv2Pg

[11] AAA, a Project of OpenDaylight Controller, accessed on 2018-01-02.
URL https://goo.gl/LvfRoH

[12] Y. E. Oktian, S.-G. Lee, J. Lam, OAuthkeeper: An Authorization Framework for Software Defined Network, Journal of Network and Systems Management 1–22.

[13] C. R. Taylor, D. C. MacFarland, D. R. Smestad, C. A. Shue, Contextual, flow-based access control with scalable host-based SDN techniques, in: Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on, IEEE, 2016, pp. 1–9.

[14] F. Klaedtke, G. O. Karame, R. Bifulco, H. Cui, Access control for SDN controllers, in: Proc. 3rd ACM HotSDN, 2014, pp. 219–220.

[15] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, G. Gu, A Security Enforcement Kernel for OpenFlow Networks, in: Proc. 1st ACM HotSDN, 2012, pp. 121–126.

[16] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, X. Chen, SDNShield: Reconciliating Configurable Application Permissions for SDN App Markets, in: Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on, IEEE, 2016, pp. 121–132.

[17] C. Banse, S. Rangarajan, A secure northbound interface for SDN applications, in: Trustcom/BigDataSE/ISPA, 2015 IEEE, Vol. 1, IEEE, 2015, pp. 834–839.

[18] Y. E. Oktian, S. Lee, H. Lee, J. Lam, Secure your northbound SDN API, in: Ubiquitous and Future Networks (ICUFN), 2015 Seventh International Conference on, IEEE, 2015, pp. 919–920.

[19] M. Lee, Y. Kim, Y. Lee, A home cloud-based home network auto-configuration using SDN, in: Networking, Sensing and Control (ICNSC), 2015 IEEE 12th International Conference on, IEEE, 2015, pp. 444–449.

[20] H. Kim, N. Feamster, Improving network management with software defined networking, IEEE Communications Magazine 51 (2) (2013) 114–119.

[21] D. Levin, M. Canini, S. Schmid, A. Feldmann, Incremental SDN deployment in enterprise networks, in: ACM SIGCOMM Computer Communication Review, Vol. 43, ACM, 2013, pp. 473–474.

[22] H. Ali-Ahmad, C. Cicconetti, A. De la Oliva, V. Mancuso, M. R. Sama, P. Seite, S. Shanmugalingam, An SDN-based network architecture for extremely dense wireless networks, in: Future Networks and Services (SDN4FNS), 2013 IEEE SDN for, IEEE, 2013, pp. 1–7.

[23] A. Basta, A. Blenk, K. Hoffmann, H. J. Morper, M. Hoffmann, W. Kellerer, Towards a cost optimal design for a 5g mobile core network based on SDN and NFV, IEEE Transactions on Network and Service Management.

[24] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, R. Kompella, Towards an elastic distributed SDN controller, in: ACM SIGCOMM Computer Communication Review, Vol. 43, ACM, 2013, pp. 7–12.

[25] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, G. Wang, Meridian: an SDN platform for cloud network services, IEEE Communications Magazine 51 (2) (2013) 120–127.

[26] T. Wang, F. Liu, H. Xu, An Efficient Online Algorithm for Dynamic SDN Controller Assignment in Data Center Networks, IEEE/ACM Transactions on Networking 25 (5) (2017) 2788–2801.

[27] H. developer, Northbound Interface of SDN, accessed on 2018-10-25.
URL https://goo.gl/D2wv2L

[28] C. Rigney, S. Willens, A. Rubens, W. Simpson, Remote au-

thentication dial in user service (RADIUS). RFC 2058, Network Working Group. (1996).

[29] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, J. Arkko, Diameter base protocol. RFC 3588, Network Working Group. (2003).

[30] S. Matsumoto, S. Hitz, A. Perrig, Fleet: Defending SDNs from Malicious Administrators, in: Proc. 3rd ACM HotSDN, 2014, pp. 103–108.

[31] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, S. Krishnamurthi, Participatory Networking: An API for Application Control of SDNs 43 (4) (2013) 327–338.

[32] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, Y. Zhang, Pga: Using graphs to express and automatically reconcile network policies, ACM SIGCOMM Computer Communication Review 45 (4) (2015) 29–42.

[33] H. Hu, W. Han, G.-J. Ahn, Z. Zhao, Flowguard: building robust firewalls for software-defined networks, in: Proceedings of the third workshop on Hot topics in software defined networking, ACM, 2014, pp. 97–102.

[34] Opendaylight, A mainstream SDN controller, accessed on 2018-10-12.
URL https://goo.gl/JwB2G6

[35] Symantec, Cloud Data Protection and Security, accessed on 2018-09-28.
URL https://goo.gl/yud9Mq

[36] DoorCloud, Cloud Access Control, accessed on 2018-10-18.
URL https://goo.gl/bxBAkF

[37] A. R. Khan, Access control in cloud computing environment, ARPN Journal of Engineering and Applied Sciences 7 (5) (2012) 613–615.

[38] R. Charanya, M. Aramudhan, Survey on access control issues in cloud computing, in: Emerging Trends in Engineering, Technology and Science (ICETETS), International Conference on, IEEE, 2016, pp. 1–4.

[39] Y. A. Younis, K. Kifayat, M. Merabti, An access control model for cloud computing, Journal of Information Security and Applications 19 (1) (2014) 45–60.

[40] R. Aluvalu, L. Muddana, A survey on access control models in cloud computing, in: Emerging ICT for Bridging the Future-Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1, Springer, 2015, pp. 653–664.

[41] M. S. Malik, M. Montanari, J. H. Huh, R. B. Bobba, R. H. Campbell, Towards SDN enabled network control delegation in clouds, in: Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on, IEEE, 2013, pp. 1–6.

[42] R. Miao, M. Yu, N. Jain, Nimbus: cloud-scale attack detection and mitigation, in: Acm sigcomm computer communication review, Vol. 44, ACM, 2014, pp. 121–122.

[43] D. C. Verma, Simplifying network administration using policy-based management, IEEE network 16 (2) (2002) 20–26.

[44] A. Rayes, M. Cheung, Policy-based network security management, uS Patent 7,237,267 (Jun. 26 2007).

[45] L. Lymberopoulos, E. Lupu, M. Sloman, An adaptive policy-based framework for network services management, Journal of Network and systems Management 11 (3) (2003) 277–303.

[46] Q. Li, Y. Chen, P. P. C. Lee, M. Xu, K. Ren, Security Policy Violations in SDN Data Plane, IEEE/ACM Transactions on Networking (TON) 26 (4) (2018) 1715–1727.

[47] F. Hadi, M. Imran, M. H. Durad, M. Waris, A simple security policy enforcement system for an institution using SDN controller, in: 2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST), IEEE, 2018, pp. 489–494.

[48] V. Varadharajan, K. Karmakar, U. Tupakula, M. Hitchens, A Policy-Based Security Architecture for Software-Defined Networks, IEEE Transactions on Information Forensics and Security 14 (4) (2019) 897–912.

[49] W. Han, C. Lei, A survey on policy languages in network and security management, Computer Networks 56 (1) (2012) 477–489.

[50] B. Moore, E. Ellesson, J. Strassner, A. Westerinen, Policy Core Information Model–Version 1 Specification. RFC 3060, Network Working Group. (2001).

[51] T. Moses, et al., Extensible access control markup language (xacml) version 2.0, Oasis Standard 200502.

[52] L. Xu, J. Huang, S. Hong, J. Zhang, G. Gu, Attacking the Brain: Races in the SDN Control Plane, in: 26th USENIX Security Symposium (USENIX Security 17), 2017.

[53] S. Scott-Hayward, Design and deployment of secure, robust, and resilient SDN controllers, in: Network Softwarization (NetSoft), 2015 1st IEEE Conference on, IEEE, 2015, pp. 1–5.

[54] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, et al., Troubleshooting blackbox SDN control software with minimal causal sequences, ACM SIGCOMM Computer Communication Review 44 (4) (2015) 395–406.

[55] A. Gounares, Interactive graph for navigating application code, uS Patent 9,658,943 (May 23 2017).
URL https://www.google.com/patents/US9658943

[56] I. BAKER, K. BASSIN, S. Kagan, S. Smith, System and method to classify automated code inspection services defect output for defect analysis, uS Patent 9,442,821 (Sep. 13 2016).
URL https://www.google.com/patents/US9442821

[57] Code inspections in the intellij platform, accessed on 2018-11-10.
URL https://goo.gl/kDqenJ

[58] Project Floodlight, accessed on 2018-11-7.
URL https://goo.gl/8CxYdF

[59] Onosproject, Onos, accessed on 2018-10-12.
URL https://goo.gl/Sdsc6X

[60] Ryu SDN Framework, accessed on 2018-11-7.
URL https://goo.gl/Mdxewq

[61] A. Bierman, M. Bjorklund, K. Watsen, RESTCONF protocol. RFC 8040, IETF. (2017).

[62] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: International Conference on Computer Aided Verification, Springer, 2002, pp. 359–364.

[63] ANTLR, Another Tool for Language Recognition, accessed on 2018-10-28.
URL https://goo.gl/bxBAkF

[64] Akka, building highly concurrent, distributed and resilient message-driven applications on the JVM, accessed on 2018-12-07.
URL https://goo.gl/3yU63u

[65] S. R. Hussain, O. Chowdhury, S. Mehnaz, E. Bertino, LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE, in: Symposium on Network and Distributed Systems Security (NDSS), 2018, pp. 18–21.

[66] Opendaylight, Neutron Northbound, accessed on 2018-12-23.
URL https://goo.gl/MZ4Fdd

[67] O. Sefraoui, M. Aissaoui, M. Eleuldj, OpenStack: toward an open-source solution for cloud computing, International Journal of Computer Applications 55 (3).