SDNKeeper: Lightweight Resource Protection and Management System for SDN-based Cloud

Xue Leng*, Kaiyu Hou†, Yan Chen†, Kai Bu* and Libin Song†
*College of Computer Science and Technology, Zhejiang University, China
†Department of Electrical Engineering and Computer Science, Northwestern University, USA
Email: lengxue_2015@outlook.com, kyhou@u.northwestern.edu, ychen@northwestern.edu
kaibu@zju.edu.cn, libinsong2020@u.northwestern.edu

***Abstract*—Even though, SDN-based cloud has the merit of allowing more flexibility in network management, the security of network accessing and the correctness of network configuration in SDN-based cloud however have not been effectively addressed yet. In this paper, SDNKeeper, a generic and fine-grained policy enforcement system in SDN-based cloud is proposed, which can defend against unauthorized attacks and avoid network resource misconfiguration. With the usage of SDNKeeper, numerous flexible network management policies can be created by administrators, which give administrators the discretionary room on controlling the network resources. To be specific, SDNKeeper can reject any unauthorized network access request at Northbound Interface (NBI), which located between application plane and control plane. Moreover, compared with other traditional policy-based access control systems, SDNKeeper is totally application-transparent and lightweight, which is easy to implement, deploy and runtime configure. Based on the prototype implementation and evaluation, we conclude that SDNKeeper can perform access control accurately with negligible computation overhead whilst the throughput degradation is still within the acceptable range.**

***Index Terms*—Software Defined Networking, SDN-based Cloud, Network Management, Access Control, Unauthorized Attack**

I. Introduction

Combining the programmability of SDN and elasticity of cloud, SDN-based cloud as a new paradigm, which provides a more flexible, efficient and convenient way to control and manage the fundamental network service, comes into our sight. Its abstract architecture is depicted in Fig.1 (a). With broad application prospects, SDN-based cloud has been used in Cloud Data Center and Carrier Networks, such as CloudFabric [1] developed by Huawei and NovoDC [2] developed by China Mobile. With reference to the prediction data provided by International Data Corporation [3], the worldwide market of SDN-based Cloud Data Center is expected to reach $12.5 billion by 2020.

In the architecture shown in Fig.1, SDN controller is the core component of SDN-based cloud, managing the fundamental network. Thus, the security of SDN controller is quite important for the whole system. To better illustrate our motivation, we take Northbound Interface (NBI) as the center point and investigate SDN-based cloud architecture from two perspectives. *On one hand*, the application plane will be focused on when we look up from NBI. In order to

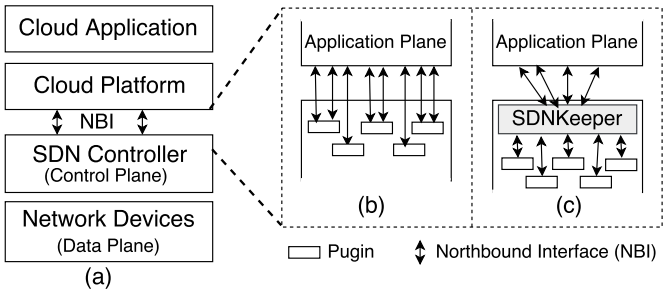

Fig. 1: Architecture of SDN-based cloud, as well as the application scenario of *SDNKeeper*.

make better use of SDN, various of third-party applications are developed, which get even modify the resource information in the controller and manipulate network resources in the data plane to achieve business demands. For example, the security applications will insert their own filter rules into the data plane, as well as collect and analyze the information stored in the controller. But the security of northbound requests sent from these applications, did not attract enough attention. Due to the absence of checking mechanisms on REST API accessing, a malicious user or an attacker can paralyze the controller. For instance, if a monitoring application has other permissions beyond getting information such as update permission, it will become a threat and can tamper with the data in the controller based on its monitoring statistics. For another example, a registered user can modify any data in the controller through sending a corresponding REST request, this dangerous operation will disturb the controller and destroy the network. Although there are some related studies ([4], [5]), they only verify users' legitimacy while leave the verification of user operation out, which is precisely the key to solve the security problem mentioned above as well as an issue addressed in this paper.

On the other hand, when we look down from NBI, the plugins[1] located in the controller will be spotlighted. To provide a powerful SDN controller, various of plugins with specific functions are developed to cooperate with core projects. Take OpenDaylight [6], a mainstream SDN controller, for example, currently, there are 62 projects opening thousands of REST

[1]Plugins are projects located in controller and provide specific functions for upper level applications.

APIs to upper applications. This situation can be described as Fig.1 (b). Related work [7] focuses on the security of plugins themselves and the communication between plugins and core projects of the controller. To the best of our knowledge, there is not a unified interface or tool can manage these plugins' REST APIs and perform access control on REST access requests.

To address these two issues mentioned above, we propose **SDNKeeper**, the first fine-grained and generic policy enforcement system for protecting and managing resources in SDN-based cloud. To accomplish this system, we have to overcome the following challenges: 1) *Applicability*, the proposed system should work smoothly with existing controllers and applications; 2) *Administrator friendliness*, the system should provide a convenient and formalized way for network administrators to express their intentions; 3) *Centralized management*, the system should be able to manage all the plugins inside the controller, including their REST APIs and requests accessing them; 4) *Hot update*, the update of the system should be transparent to other components.

In this paper, SDNKeeper is presented, which can intercept unauthorized access requests and assist administrators to manage network resources. As shown in the Fig.1 (c), NBI is the indispensable interface connecting upper applications and the controller. Performing access control on NBI can reserve the high-level abstraction information from users/upper layer applications, as well as block the illegal access requests out of the controller, thereby protecting and saving resources in the controller.

SDNKeeper performs access control on NBI based on the policies defined by administrators. In order to provide administrators a convenient way to express intentions to protect and manage resources, we design a policy language with readable and operable format, which can narrow down to any specific attribute of requests and resources. A *policy interpreter* is designed to parse these policies into a controller-processable format then issue them to the data store. After a request coming to access the controller, *permission engine* will check its legality with policies issued before. The benign requests will continue to be processed by the controller, while the illegal ones rejected by policies will be blocked outside the controller. In sum, all 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 through monitoring the access requests.
- We design a fine-grained policy language for administrators to define management policies, realizing centralized management of resources.
- We implement SDNKeeper with the feature of hot-update, to be specific, it refers to policy hot-update. Administrators can update policies on the fly and 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 computation overhead.

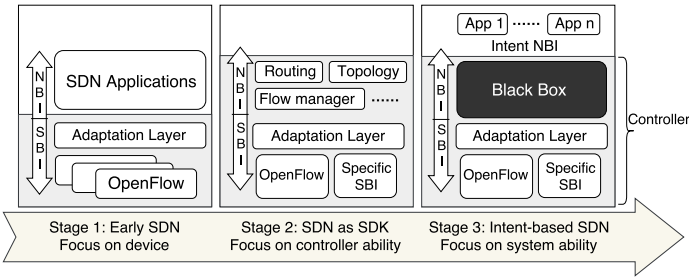

Fig. 2: The evolution of Northbound Interface.

The rest of paper is structured as follows. Firstly, research background (Section II-A) and related work (Section II-B) are presented. And following that, an overview of SDNKeeper is described in Section III, including application scenarios and the architecture of SDNKeeper. In Section IV and Section V, we illustrate the design details of policy and permission engine. We implement and evaluate the SDNKeeper prototype in Section VI. Finally, in Section VII we conclude the paper.

II. Background and Related Work

A. 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 ([8], [9]), enterprise networking ([10]), telecommunication networking ([11], [12]) and data center/cloud networking ([13]–[15]). The global cloud service providers like Microsoft Azure [16], [17], IBM [18] and Google [19] are all leading to use SDN in their cloud network architectures. According to the data from Synergy Research Group [20], $12 billion revenues have been reached by the global cloud infrastructure service in the third quarter of 2017. The emergence of SDN brings 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.

In large-scale networking, the resource management as well as the network creation and configuration are completely relying on network administrators, so that human errors are inevitable. When creating network, administrators need to keep in mind which network segment is available and whether the *VlanId* has been assigned. With a slight negligence, networks that need to be isolated could communicate with each other. These bad operations are usually hard to be aware until receiving alerts from physical network devices or tenants. What's more, it is more challenging to trace these faults due to lack of administrator operation records. Under this circumstance, a tool assisting in managing network resources becomes important to network administrators.

Besides these inherent drawbacks, when adopting SDN to provide fundamental network service in large-scale scene such as cloud, a series of new issues will arise. First, attackers can

send malicious requests utilizing APIs obtained from applications accidentally. Secondly, the requests are sent correctly, but the content were tampered with during transmission. Finally, the registered user may directly issue malicious requests to SDN controller. 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 SDN controller through NBI, the best way to protect resources is to perform access control on NBI. As depicted in Fig. 2, the evolution of NBI has gone through three stages [21]. With the rise of focus, more and more specific plugins integrated inside the controller even become a black box for upper applications. Due to lack of internal details, performing access control inside these plugins is not advisable. Thus, a wise design is proposed in this paper, performing access control on NBI, which adapts to the evolution of NBI and makes the system compatible.

The two main issues mentioned above are critical to the security of fundamental network. However, at present, they have not been addressed efficiently. Also, among the existing work, there is no access control mechanism performed at northbound level of SDN. In a word, an access control mechanism is important and necessary for protecting and managing network, which is the key point this paper worked on.

B. Related Work

Based on our work, we divide the related work into the following categories, access control in SDN, access control in cloud and policy language, then compare our work with these papers from above three aspects.

Access Control in SDN. Work [22]–[24] perform access control on southbound and data plane. And SDNShield [7] performs access control on the plugins inside the controller through modifying plugins' codes. However, they did not consider the malicious requests coming from northbound. Work [25], [26] design new controller architectures and APIs to prevent malicious operations, which weaken the applicability and flexibility of SDN controller. Then, PGA [27], FlowGuard [28] are designed for Access Control List (ACL). They can only control the packets routing in the existing network, while cannot control the network configuration in SDN-based cloud, as well as access requests from northbound.

For the work focusing on northbound, AAA [4], a project of OpenDaylight controller, can realize fundamental user authentication and authorization. In the latest version of *Nitrogen*, AAA can filter access requests at the coarse resource granularity, while unable to make decisions according to the attributes of requester, resource and environment.

Access Control in Cloud. Prior access control mechanisms in cloud [29]–[34] are in allusion to protect data security and user privacy. They have not provided effective protection for SDN. Although, work [35], [36] are based on SDN, they only control the network and detect attacks within and between cloud. Controller has not been taken into their concerned.

Policy Language. Due to the expansion of scale and increased management complexity, a variety of policy-driven languages expressing intentions of administrators are proposed to manage and secure the network [37]. These languages focus on either network management or security management, e.g., PCIM [38] can be mapped into network device configurations, while XACML [39] is designed based on XML and consists of four components to perform access control. Compared with it, our work is lightweight and easy to implement and deploy to achieve the purpose of access control on REST requests. Overall, SDN-based cloud is a new scenario, realizing a lightweight policy enforcement mechanism for both network management and protection is still challenging and unresolved.

Therefore, designing a lightweight access control system to protect and manage resources for SDN-based cloud is challenging and of great significance.

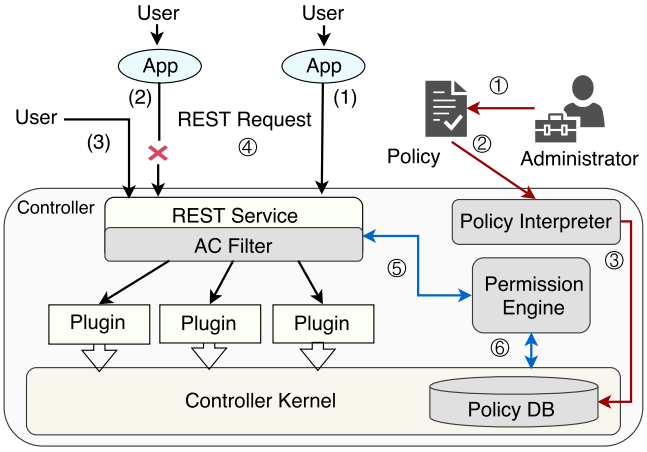

Fig. 3: Architecture and workflow of *SDNKeeper*.

III. SDNKeeper Overview

In this section, after presenting two application scenarios of SDNKeeper, we give a bird's-eye view of SDNKeeper as well as a macroscopic description of its main components. To better understand the mechanism of SDNKeeper, we demonstrate the workflow within its architecture.

A. Application Scenario

To focus on demonstrating the core design of SDNKeeper and make our work understandable without any ambiguity, we make two assumptions here: **1) SDN controller**, which performs permission checking, **is safe**. A large body of research and techniques ([40]–[42]) make SDN controller safer and more robust, and our work mainly focuses on the legitimacy of access requests and the security of resources. Therefore, we assume that the controller is safe. **2) Administrator**, who specifies policies, **is credible**. Administrator has the highest authority so that can configure the whole network via making policies. Hence, attacking network through hijacking administrator will not be discussed in this paper.

Based on above two assumptions, we summarize two application scenarios of SDNKeeper as follows. Here we regard resource as all plugins inside the controller, as well as the 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 II-A, also shown in Fig. 3. Request (1) comes from the application and carries illegal information. Request (2) is also sent from benign user and application, but it was tampered with halfway through. And request (3) is sent directly to the controller by the registered user. All these requests are dangerous to the controller and put resources at risk of being tampered with. While SDNKeeper can intercept these malicious requests effectively.

Scenario 2: Managing resources. There are various of plugins inside the controller opening thousands of REST APIs for upper applications. SDNKeeper provides a unified entrance to manage resources, such as controlling which resource can be deleted or which resource can be queried. Administrators just need to add a policy in SDNKeeper to achieve the goal of managing resources in the controller and data plane.

B. Architecture of SDNKeeper

In 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 [43] communicates with SDN controller through REST API provided by plugins inside the controller. Meanwhile, REST Service, as the unique northbound channel in SDN controller, processes all requests sent from cloud. Hence, our key idea of protecting and managing network resources in SDN-based cloud is to perform access control at the NBI level.

In general, SDNKeeper as a fine-grained policy enforcement system provides real-time protection and permission checking for 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 violating the policies.

In our design, SDNKeeper mainly consists of two parts, policy interpreter and permission engine. The complete access control workflow of SDNKeeper is described as follows.

1) 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.

Policy interpreter is a component of SDNKeeper. After the administrator defines policies in policy language based on the global view and security demands, these policies will be issued to the controller. To be specific, policy interpreter parses and transforms the semantic policies into tree-structured data and stores them in the data store of the controller.

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

Generally, in original SDN system, all requests sent from various applications are directly loaded into the controller without checking the legitimacy and correctness. Thus, malicious requests can strike the system without any obstruction. Though, several security inspection techniques ([44]–[46]) are presented for the safety of applications. They can only work offline and unable to ensure the legality of requests sent to a running plugin. In SDNKeeper, 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 IV and Section V, respectively.

IV. Policy Language and Policy Design

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

A. REST Request

REST API[2] is the most common way for tenants to request network resources through NBI in SDN. Almost all SDN controllers today provide REST API, and recommend or require using REST request to access network resources from northbound, like OpenDaylight [6], Floodlight [47], ONOS [48] and Ryu [49]. A REST request has four main parts as shown blow.

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,
        ... }}
```

[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.

1) `Method` defines the HTTP verbs a requester wants to perform. The most commonly used HTTP verbs are `POST`, `GET`, `PUT` and `DELETE`. They correspond to create, read, update/replace and delete operations, respectively.
2) `URI` identifies the network resource provided by the controller. Typically, plugins register their `URI`s in REST Service. For example, the REST Service in OpenDaylight is called *RESTConf*, plugins should tell *RESTConf* what the `URI`s they want to use to identify their resources. A `query` condition with some attributes may follow the `URI` when `GET` verb is requested.
3) `Headers` carry a list of information in HTTP request, such as the content type of this request and the authorization token of a requester.
4) If a requester requests to create (`POST`), update/replace (`PUT`) or delete (`DELETE`) a resource, a Json body with detailed attributes of this resource should be included.

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

B. Policy

A policy in SDNKeeper is designed to determine whether to approve or decline a REST request. 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 or routers (`URI`). All the context in the `Body` part of 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 tenant cannot use resource 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 by format *subject.role* and *subject.user*. For Object, action attributes can be fetched by the format *action.uri* and *action.method*. And 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 access 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 administrator can get any information from the REST request and customize arbitrary 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, finally returning 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 second *if-statement*, Bob can create a *network*, if the type of network is *Vlan*.

A Policy

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

C. 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 intended for individual user group and user only, which have user-related attributes: *role* and *username*. When a request from a certain user comes in, only the related local policies with the matching *role* and *username* will be checked.

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

There are two reasons for designing these two separated policy sets. One is *for performance*. Permission engine only

needs to check global policies and related 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.

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

A Policy File

```
GLOBAL_POLICY {
  system_update {
    if (environment.time > 12pm &&
        environment.time <  1am ) {
      REJECT }}}
LOCAL_POLICY {
  user {
    user_can_get_on_monday {
      if (action.method == 'GET') {
        if (environment.weekday == 'mon') {
          ACCEPT }}}}
  user.Alice {
    alice_cannot_delete_firewall {
      if (action.uri REG '/firewalls/') {
        if (action.method == 'DELETE') {
          REJECT }
        else {
          ACCEPT }}}
      ... }}
```

D. Policy Generation

Enforcing SDNKeeper in SDN-based cloud only requires network administrators to learn a little knowledge. First, the REST APIs and related attributes have already been recorded. Thus, administrators do not need to learn about them. Second, policies in SDNKeeper are Json-based rules. Those access control rules can be created by simply following the description in Section IV, which are totally the same as Json grammar.

Before enforcing SDNKeeper, administrators first need to summarize the characteristics of attacks, business needs and system restrictions. Basic global policies for the whole cloud and various local policies for different types of tenants are created from those characteristics. When a new tenant joins this cloud, administrators only need to assign this tenant into a corresponding role. The authority of this tenant will follow the policies described by the predefined global and local policies. In addition, administrators can also create special policies for particular tenant by adding a new local policy for this tenant on the fly.

V. 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.

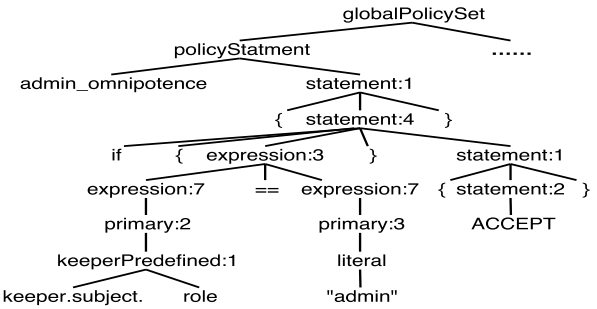

Fig. 4: Semantic tree of global policy.

A. Policy Interpreter

The human-language based policies are required to transfer to computer-processable data structure. In 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 child and right child, we can get the value of root node, *i.e.*, the result value of permission checking.

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

B. 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 against policies, and 3) finally makes a decision on approving or declining this request. We highlight several issues in permission engine design as following.

Policy Conflict. Because of the intersection of different policies, a REST request may be approved by one matched policy and rejected by another matched policy, which brings a policy conflict. As shown below, if user Alice requests to `GET` a *networking* 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 *networking* resource as described in *net_reject_alice* policy. Therefore, it will be unsecured if we make a decision by only one approved policy.

```
GLOBAL_POLICY {
  all_can_get {
    if (action.method == 'GET') {
      ACCEPT }}}
LOCAL_POLICY {
  user.Alice {
    net_reject_alice {
      if (action.uri == '/networks/') {
        REJECT }}}}
```

For the sake of security, we introduce full match strategy in 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 a "REJECT", permission engine will decline this request immediately. If no policy is matched by this request, this request will also be declined. Otherwise, this request should be approved. The complete permission checking process is illustrated in Algorithm 1.

Algorithm 1: Permission Checking

```
Input  : request
Output: ACCEPT or REJECT
approved ← false
policy_set ← {Global, Local[role], Local[role][user]}
for policy in policy_set do
    if request matches policy then
        if policy.eval(request) == REJECT then
            return REJECT
        else approved ← true
If approved == true return ACCEPT
return REJECT;
```

Filter Based. Permission engine acts as a filter between application plane and control plane. Therefore, illegal requests will be rejected before reaching the related modules inside the controller, which will never occupy the network resources. Filter based design can also bring benefits in deployment. Typically, controllers have a REST Service module for receiving and distributing REST requests. It will only have a little code changing when adding a new REST filter to REST Service module. In most cases, like OpenDaylight and ONOS, we can enable SDNKeeper in them by adding several dependencies to configuration file.

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

VI. Implementation and Evaluation

There are two major components in the prototype of SDNKeeper: 1) **policy interpreter** parses semantic policies into semantic trees, 2) **permission engine** performs permission checking for each coming request. In this section, we first introduce the implementation of these two components. Then, we evaluate the performance of SDNKeeper from three metrics: **effectiveness**, **latency** and **throughput** and briefly discuss the results in the end.

A. Implementation

We implement SDNKeeper as a plugin of OpenDaylight [6] controller, which works as a filter to control any REST request from upper applications to the controller. As SDNKeeper is application-independent, we can support every network application in OpenStack naturally.

Though, the implementation is inherently related to the controller specification, policy interpreter and permission engine are implemented as the controller-independent Java bundles. Currently, almost all mainstream controllers (*OpenDaylight, ONOS*) use REST API in northbound communication, which makes it easy to deploy SDNKeeper on controllers to perform access control on REST requests. Benefit from the lightweight of SDNKeeper, the deployment of the whole system is lowcost, which only need embed SDNKeeper into the controller as a feature and ask the 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 [4]) 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 are implemented as follows.

*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 data store of the controller. In this progress, ANTLR [50], a language recognition tool, is responsible for reading and parsing semantic policies continuously, then a registered listener will insert the policy tree into data store once a new one is loaded. Finally, all policies will be stored as a tree, so that permission engine just needs to recursively traverse a tree to enforce a policy.

*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 high concurrency. In order to adapt to this character, we adopt Akka [51] to process multiple requests simultaneously through creating a certain number 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. With the *Policy Data Store Listener* in permission engine, the policy cache in memory can be updated at runtime once the administrator refined policies in the data store. And the new policies will take effect on subsequent requests. In order to facilitate 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.

In practice, multiple controllers cooperate with each other as a cluster to provide network services. When working in

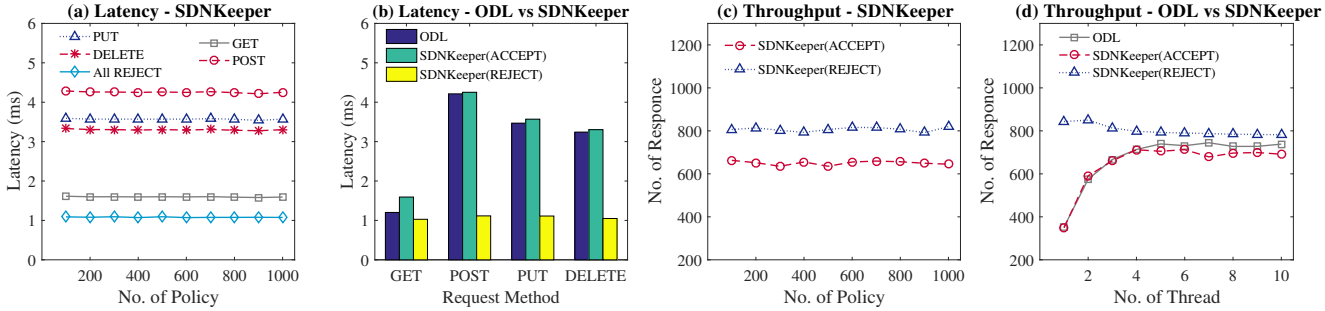

Fig. 5: *Evaluation Result*: ***(a)*** latency of SDNKeeper with different numbers of policies, ***(b)*** latency between original and SDNKeeper-enabled OpenDaylight with 1000 policies, ***(c)*** throughput of SDNKeeper per second with different numbers of policies under 2 threads, ***(d)*** throughput between original and SDNKeeper-enabled OpenDaylight with different numbers of threads.

multi-controller scenario, SDNKeeper can still perform well since the distributed policy data store makes each permission engine on different controllers has the same view of policies.

B. Evaluation

1) ***Methodology****:* We establish the testbed of SDNKeeper on the mainstream SDN controller OpenDaylight (Intel i7-7700 8x3.6GHz, 16GB Memory), and choose Neutron [52], a component providing network service in OpenStack [43], as our test application. Neutron provides 30 kinds of REST API, ranging from networking, firewall, QoS to load balance, with 185 kinds of requests (GET, POST, PUT, DELETE) and 664 related attributes, which we think are enough to evaluate the effectiveness of SDNKeeper.

In our evaluation, tenants send REST requests to OpenDaylight (ODL) through REST API. And SDNKeeper performs access control on REST API between application plane and control plane. We examine the check results of those requests in the controller and the response in tenants to evaluate the performance of SDNKeeper.

We first evaluate the effectiveness of SDNKeeper, i.e., whether SDNKeeper can reject all kinds of unauthorized requests correctly. Then, we measure the extra processing latency introduced by SDNKeeper and REST request throughput comparing between controllers with and without SDNKeeper. Since if an illegal request is rejected by SDNKeeper, the processing time and resources occupancy would be largely reduced. Hence, we evaluate the performance of SDNKeeper in both cases, all decisions are "ACCEPT" and all decisions are "REJECT".

TABLE I: REST API in OpenStack Neutron

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

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 III-A. Since these illegal requests have the same format, we simulate these requests through sending REST requests uniformly. Table I lists all types of REST APIs provided by OpenStack Neutron. These APIs are representative to show the correctness of SDNKeeper in rejecting the unauthorized access in SDN-based cloud.

When verifying the effectiveness of intercepting unauthorized requests, we send two kinds of illegal requests: 1) requests accessing resources not belong to current user, 2) requests performing extra operations on his own resources. We create 2789 policies in 3 granularities: 30 policies for all kinds of APIs, 185 policies for all kinds of actions in API, 664 policies for all kinds of attributes and 1910 policies for all possible combinations of two attributes. Then, 2789 related illegal requests are sent to violate them. Note that, we create illegal requests by setting incorrect values to some fields of the request to make it violate one or more policies which need to be checked with. The results show that all of these illegal requests are rejected by SDNKeeper.

3) ***Latency Evaluation****:* In SDNKeeper, matching policies and checking permissions may introduce extra delay in controller when processing a REST request. We evaluate this delay by measuring the latency in tenants from sending a request to receiving the corresponding response. Two experiments are performed in this part: 1) latency in different numbers of policies (Fig. 5 (a)) and 2) latency between controllers with and without SDNKeeper (Fig. 5 (b)). Each test is executed 5 times with 30000 requests.

Fig. 5 (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 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.

Under the same scenario with 1000 policies, we compare the latency between SDNKeeper-enabled OpenDaylight and original OpenDaylight. Since the policy decision affects request processing time, *i.e.*, decision "REJECT" will make the processing time shorter than original, while decision "ACCEPT" will induce a little bit of computation overhead. As

shown in Fig. 5 (b), SDNKeeper with decision "ACCEPT" only introduces about 0.15ms extra delay on average. And the latency is largely reduced about 0.17ms, 3.10ms, 2.36ms and 2.19ms in GET, POST, PUT, DELETE requests respectively, when request is "REJECT" by SDNKeeper. 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. In short, SDNKeeper has insignificant computation overhead for policy processing.

*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 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. 5 (c), no matter what 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. The number of policies has negligible impact under our semantic tree design.

In Fig. 5 (d), we compare the throughput in 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 decision "REJECT" always gets the best performance without being affected by the number of threads. While the throughput of both original OpenDaylight and SDNKeeper-enabled OpenDaylight with decision "ACCEPT" are varied with thread's number. 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 according to evaluation results. In short, SDNKeeper performs access control accurately with negligible effect on the processing capability of the controller.

*5) **Discussion**:* Compared with the Southbound Interface (SBI), the NBI is latency insensitive and infrequent. According to evaluation results, 0.15ms extra delay by SDNKeeper in NBI communication is acceptable. In industry, the throughput threshold will be limited by a reasonable experience value to ensure each request can be processed in the controller. Besides, since permission engine of SDNKeeper is stateless, running it in multi-controller system can get better performance. In a word, SDNKeeper can prevent unauthorized requests effectively with negligible impact on the performance of SDN controller.

VII. Conclusion

SDN-based cloud as a new concept has been widely applied in many industrial scenarios. However, effective resource protection and management in SDN-based cloud have not been addressed yet. In this paper, we propose SDNKeeper, a lightweight policy enforcement system, which can prevent network resources from illegal access requests and assisting network administrator in managing network resources. Through defining fine-grained policies, SDNKeeper can perform access control on each request received by the controller to defend against unauthorized attack and avoid network misconfiguration. Beyond all benefits above, SDNKeeper is also application-transparent and able to support administrators to update policies on the fly. We implement the prototype of policy enforcement system and evaluate its performance. The results show that SDNKeeper can perform access control accurately with negligible computation overhead and acceptable throughput degradation.

Appendix
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.

Acknowledgements

This work is supported by National Key R&D Program of China (2017YFB0801703) and the Key Research and Development Program of Zhejiang Province (2018C01088).

References

[1] "Cloudfabric, a sdn-based data center developed by huawei," accessed on 2018-2-24. [Online]. Available: https://goo.gl/mp9E9J

[2] "Novodc, a sdn-based data center developed by china mobile," accessed on 2018-2-24. [Online]. Available: https://goo.gl/pdktxv

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

[4] "Aaa, a project of opendaylight controller," accessed on 2018-1-2. [Online]. Available: https://goo.gl/LvfRoH

[5] Y. E. Oktian, S.-G. Lee, and J. Lam, "Oauthkeeper: An authorization framework for software defined network," *Journal of Network and Systems Management*, pp. 1–22.

[6] Opendaylight, "A mainstream sdn controller," accessed on 2017-10-12. [Online]. Available: https://goo.gl/JwB2G6

[7] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, and 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.

[8] M. Lee, Y. Kim, and 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.

[9] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[10] D. Levin, M. Canini, S. Schmid, and A. Feldmann, "Incremental sdn deployment in enterprise networks," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 473–474.

[11] H. Ali-Ahmad, C. Cicconetti, A. De la Oliva, V. Mancuso, M. R. Sama, P. Seite, and 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.

[12] A. Basta, A. Blenk, K. Hoffmann, H. J. Morper, M. Hoffmann, and 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*, 2017.

[13] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 7–12.

[14] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang, "Meridian: an sdn platform for cloud network services," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 120–127, 2013.

[15] T. Wang, F. Liu, and H. Xu, "An efficient online algorithm for dynamic sdn controller assignment in data center networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2788–2801, 2017.

[16] A. Greenberg, "Sdn for the cloud," in *Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[17] "Microsoft azure and software defined networking," accessed on 2017-11-8. [Online]. Available: https://goo.gl/t2QVUm

[18] "Ibm network services for software defined networks," accessed on 2017-11-8. [Online]. Available: https://goo.gl/xP6cLh

[19] "Google cloud platform," accessed on 2017-11-8. [Online]. Available: https://goo.gl/B2fMfJ

[20] "Synergy research group," accessed on 2017-11-8. [Online]. Available: https://goo.gl/f7yTH9

[21] H. developer, "Northbound interface of sdn," accessed on 2017-10-25. [Online]. Available: https://goo.gl/D2wv2L

[22] C. R. Taylor, D. C. MacFarland, D. R. Smestad, and 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.

[23] F. Klaedtke, G. O. Karame, R. Bifulco, and H. Cui, "Access control for sdn controllers," in *Proc. 3rd ACM HotSDN*, 2014, pp. 219–220.

[24] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proc. 1st ACM HotSDN*, 2012, pp. 121–126.

[25] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending sdns from malicious administrators," in *Proc. 3rd ACM HotSDN*, 2014, pp. 103–108.

[26] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An api for application control of sdns," vol. 43, no. 4, pp. 327–338, 2013.

[27] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to

[28] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "Flowguard: building robust firewalls for software-defined networks," in *Proceedings of the third* express and automatically reconcile network policies," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 29–42, 2015. *workshop on Hot topics in software defined networking*. ACM, 2014, pp. 97–102.

[29] Symantec, "Cloud data protection and security," accessed on 2017-09-28. [Online]. Available: https://goo.gl/yud9Mq

[30] DoorCloud, "Cloud access control," accessed on 2017-10-18. [Online]. Available: https://goo.gl/bxBAkF

[31] A. R. Khan, "Access control in cloud computing environment," *ARPN Journal of Engineering and Applied Sciences*, vol. 7, no. 5, pp. 613–615, 2012.

[32] R. Charanya and 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.

[33] Y. A. Younis, K. Kifayat, and M. Merabti, "An access control model for cloud computing," *Journal of Information Security and Applications*, vol. 19, no. 1, pp. 45–60, 2014.

[34] R. Aluvalu and 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.

[35] M. S. Malik, M. Montanari, J. H. Huh, R. B. Bobba, and 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.

[36] R. Miao, M. Yu, and N. Jain, "Nimbus: cloud-scale attack detection and mitigation," in *Acm sigcomm computer communication review*, vol. 44, no. 4. ACM, 2014, pp. 121–122.

[37] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, no. 1, pp. 477–489, 2012.

[38] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy core information model–version 1 specification," Tech. Rep., 2001.

[39] T. Moses *et al.*, "Extensible access control markup language (xacml) version 2.0," *Oasis Standard*, vol. 200502, 2005.

[40] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the sdn control plane," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[41] 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.

[42] 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*, vol. 44, no. 4, pp. 395–406, 2015.

[43] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, 2012.

[44] A. Gounares, "Interactive graph for navigating application code," May 23 2017, uS Patent 9,658,943. [Online]. Available: https://www.google.com/patents/US9658943

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

[46] "Code inspections in the intellij platform," accessed on 2017-11-10. [Online]. Available: https://goo.gl/kDqenJ

[47] "Project floodlight," accessed on 2017-11-7. [Online]. Available: https://goo.gl/8CxYdF

[48] Onosproject, "Onos," accessed on 2017-10-12. [Online]. Available: https://goo.gl/Sdsc6X

[49] "Ryu sdn framework," accessed on 2017-11-7. [Online]. Available: https://goo.gl/Mdxewq

[50] ANTLR, "Another tool for language recognition," accessed on 2017-10-28. [Online]. Available: https://goo.gl/bxBAkF

[51] "Akka, building highly concurrent, distributed and resilient message-driven applications on the jvm," accessed on 2018-1-7. [Online]. Available: https://goo.gl/3yU63u

[52] OpenStack, "Neutron," accessed on 2017-10-27. [Online]. Available: https://goo.gl/LDC2jq