# ROLE BASED ACCESS CONTROL FOR SOFTWARE DEFINED NETWORKING: FORMAL MODELS AND IMPLEMENTATION

by

ABDULLAH AL-ALAJ, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
August 2020

## DEDICATION

*This dissertation is affectionately dedicated to my mother, Nofa Alsheiab, for being a woman like no other. It is also dedicated to my wife, Aminah Altaani, and our sons, Ammar and Ghaith. I am blessed to have a family like this.*

## ACKNOWLEDGEMENTS

My passion in the research domain of security in Software Defined Networking (SDN), access control in particular, led my way to the doctoral program at the University of Texas at San Antonio (UTSA), where my advisors furthered my understanding of the field, helped me establish my goals as a researcher, equipped me with various research tools, and opened a door for me to the scientific research world.

Foremost, I would like to express my sincere gratitude to my advisors, Prof. Ravi Sandhu and Dr. Ram Krishnan, for the continuous support of my Ph.D. study and research, for their patience, motivation, enthusiasm, and immense knowledge. Their guidance helped me in all the time of research and writing of this dissertation.

I would like to thank my dissertation committee: Dr. Palden Lama, Prof. Gregory B. White, Dr. Xiaoyin Wang (in Proposal), and Dr. Weining Zhang for their valuable observations and insightful comments.

My sincere thanks also goes to members of the Institute for Cyber Security (ICS), UTSA, Farhan Patwa and James Benson for their help and time at the beginning of my research journey. Also, I would like to thank Suzanne Tanaka and Susan Allen from the ICS and the CS department for their assistance during my Ph.D. journey.

I also thank all my fellow labmates in the ICS for the stimulating discussions we have had and continuous support.

Also, I would like to thank my mother, wife, sisters, brothers, friends, and every wonderful person in my life.

August 2020

# ROLE BASED ACCESS CONTROL FOR SOFTWARE DEFINED NETWORKING: FORMAL MODELS AND IMPLEMENTATION

Abdullah Al-Alaj, Ph.D.
The University of Texas at San Antonio, 2020

Supervising Professors: Ravi Sandhu, Ph.D. and Ram Krishnan, Ph.D.

The architecture of Software Defined Networking (SDN) provides the flexibility in developing innovative networking applications for managing and analyzing the network from a centralized controller. Since these applications directly and dynamically access critical network resources, any privilege abuse from controller applications could lead to various attacks impacting the entire network domain. It is believed that SDN can, in time, prove to be one of the most impactful technologies to drive a variety of innovations in network technology. However, the security community is relatively slow in embracing SDN. As a result, the security concern is ranked one of the top issues that slow full adoption of this technology.

When network applications submit an operation to manipulate network state or request state information, the controller should employ methods to identify unauthorized access requests submitted by applications. Access control is a natural solution for preventing unauthorized operations and avoid insecure access to network resources. However, at present there is no widely accepted authorization system for applications for SDN. One reason for lack of such system is the absence of clear definition of an access control model for SDN Applications.

We believe the reason why the security community is slow in embracing access control in SDN is mainly because proper access control solutions and use cases are currently not sufficiently exposed to them yet. A deeper understanding of access control in SDN technology will help security researchers produce new, better, and effective solutions.

In this dissertation, we show our steps towards developing effective operational and administrative role-based access control models for SDN. In an attempt to understand and develop effective authorization solutions for SDN, we first formalize an access control system pertaining to an authorization system from literature called security-enhanced Floodlight.

Second, we propose a role based access control model for SDN controller apps, we called SDN-RBAC, complaint with generally accepted academic concepts of RBAC. We implement SDN-RBAC model with multi-session support in Floodlight controller and use hooking techniques to enforce the security policy without any change to the code of the controller. The implementation verifies the model's usability and effectiveness against unauthorized access requests by controller applications and shows how the framework can identify application sessions and reject unauthorized operations in real time.

Third, to cater for the need to a more granular access control and the need for applying minimum privileges on applications, we propose ParaSDN, an enhanced model that provides a fine grained access control using the concept of parameterized roles and permissions. To demonstrate the applicability and feasibility of our proposed model, we configured proof of concept use cases and implemented a prototype in the controller.

Finally, we introduce a concept of proxy and custom operations to extend the capabilities of SDN controller, and provide fine grained custom permissions specialized for the administration of SDN access control. With these extended features, we present SDN-RBACa, an administrative model to manage access control actions that define network app authorizations. Through proof of concept use cases and implementation, we demonstrate the usability of proxy operations and custom permissions and show how they enable and facilitate the administration of access control in SDN.

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1: INTRODUCTION

Software Defined Networking (SDN) has become one of the most important network architectures for simplifying network management and enabling innovation through network programmability. Thereby, SDN promises to provide the scale and versatility necessary for different fields including data centers, Internet of Things (IoT) [12], cloud computing and virtualization [29]. SDN gets more popularity due to the flexibility in developing controller applications (apps) for extending the capabilities of the SDN controller.

The emergence of the SDN paradigm has changed the way networks are built and managed. Although SDN was born in academia [13,14,34], its movement is driven by the successful adoption by researchers and industry. For example, Google has been at the forefront of companies who have taken advantages of SDN [30]. In 2011 leading companies including Google, Facebook, Microsoft, Yahoo and Verizon founded Open Networking Foundation (ONF) for accelerating research in SDN [51]. With increasing adoption and research, gradual deployment of SDN is anticipated to grow in the near future.

At the core of SDN is decoupling the control logic of the network (control plane) from the forwarding hardware (data plane). This promises to facilitate network programmability via providing an abstract view to the network infrastructure in order to develop specialized networking applications (application plane). The control plane, and the so-called controller, works as a network operating system (NOS) and thus is responsible for controlling and managing the whole network through a holistic visibility across the network resources.

SDN controllers act as the interaction point between network applications and the network infrastructure. They provide an application programming interface (Northbound API) such that applications implemented on top of the controller perform the actual network management [15, 22, 24, 35, 40]. On the other hand, the underlying communication between the controller and the switch is performed using the OpenFlow protocol [34] (Southbound API) standardized by the Open Networking Foundation (ONF) [23].

When network applications submit an operation to manipulate network state or request state information, the controller should employ methods to identify unauthorized access submitted by applications. Preventing unauthorized operations will avoid insecure access to network resources and protect against network misconfiguration so as to ensure correct functioning of the network entities. Therefore, it is vital that all controller solutions inherently embrace access control systems to circumvent unauthorized manipulation of network logic.

## 1.1 Main Features Provided by SDN

The benefits provided by SDN architecture are mainly driven by three main features: (i) dynamic flow control, (ii) logically centralized control with a network-wide visibility, and (iii) network programmability.

**Dynamic Flow Control:** One of the basic features of SDN is that network apps can dynamically control network behavior by installing traffic forwarding rules into OpenFlow enabled switches [5]. Several network applications can benefit from this capability, such as dynamic load balancing [53].

**Logically Centralized Control with a Network-Wide Visibility:** SDN provides the opportunity of holistic visibility across all network devices in the network infrastructure, i.e., the data plane. All devices in the data plane are connected to a central controller, or a so-called logically centralized controller, which receive control messages from applications (e.g., flow statistics collection, flow rule insertion, etc.). The controller collects network status information by frequently querying the data plane devices via openFlow messages and making the results available to network apps. A network app thus has a central view to all data plane devices, given that an app is authorized to access such information. Such network-wide capability is required by many applications, for example network monitoring applications [8] and network management applications [31].

**Network Programmability:** Since SDN controller provides capability for data plane devices to be controlled by applications and provides the later with real time network status information, this opens the scope for extending the controller's capabilities with new intelligent network functionalities, just like programming a smartphone app (e.g., Android) [27].

## 1.2 Motivation

Although the programmability aspect of SDN simplifies network management and enables innovation in communication networks via network APIs, it is one of the features that makes SDN more vulnerable to malicious code exploits and attacks. This is because the abstractions of the underlaying data plane resources and flows provided by the controller can be easily used for exploitations, cyber-attacks, and, most dangerously, reprogramming the entire network.

Network applications dynamically access network resources via generating network operations like flow rule insertion, network state inquiry, port configuration, etc. These operations are received by the controller which submits them to OpenFlow switches. OpenFlow switches, as a dummy forwarding device, has no means to verify application operations and so absolutely trust them and operate accordingly.

Verifying application operations sent to the data plane is an absolute prerequisite for maintaining a consistent network security policy which is the responsibility of the controller. However, current SDN controllers do not implement methods for controlling application's access rights on the infrastructure. This allows buggy or malicious applications to run arbitrary commands which makes SDN network vulnerable to various attacks from application plane. Therefore SDN OS requires an effective authorization mechanisms to ensure secure network operations [19].

The application plane within a single network might consist of a wide range of network apps for network management. These apps send OpenFlow messages and request access the data plane resources. As a request mediator, the controller is one of the most critical yet vulnerable component in the network. Typically, people assume absolute trust in the security of control plane which make it even more vulnerable to attacks.

SDN apps that are residing in the SDN controller and written in the same language of the controller are of a major security concern. This is because they are compiled as part of the controller and have direct access to various controller native classes, their methods and data. Intuitively, the more permissions available to an app the more resources accessible through these permissions, the more exposure to network attack surface. As a result, applying the principle of least privilege is vital in access control for SDN apps, especially with the fact that such app may be malicious, buggy, or having logic flaws. Without these measures, there is an increased danger that an app may be granted more access to resources than minimum because of missing or poor control by the controller.

Access control is a native solution for restricting unauthorized access to system resources and for a secure data exchange among entities in computer systems. Many access control models have been proposed and few of them have received practical deployment [28, 45–47]. Notably, among these models, role-based access control (RBAC) [20, 46] has received considerable attention from businesses, academia, and standard bodies due to many features including its flexibility in supporting various access control policies and in simplifying access control administration. Because the motivation behind SDN architecture is to simplify network management, and because the motivation behind RBAC is to simplify the task of access control policy setup and administration, it is very practical to adopt RBAC for SDN.

## 1.3 Problem Statement

Current Software Defined Networking technology is lacking access control models and enforcement for protecting network resources residing in the SDN controller from unauthorized access by OpenFlow applications.

## 1.4 Thesis Statement

*Role-based access control model and its extensions comprise an effective approach for the specification and administration of dynamic access control for Software Defined Networking*

## 1.5 Scope and Assumption

The scope of this dissertation is to develop foundational aspects of access control for SDN applications and its administration. Some of the assumptions taken during this work, and its limitations, are as follows.

- The foundation of access control models for SDN applications will be based on the current SDN architecture supported by Open Networking Foundation (ONF).
- In our implementation of access control models in Floodlight (Java based), we use the app's fully-qualified name (e.g., net.floodlightcontroller.datacapmngr.DataCapMngr) as an app ID. The fully-qualified name of an app is composed of the full package name followed by the app's class name. We assume this ID is trusted and authenticated.
- We confine our attention to network resources accessed by different OpenFlow applications and managed by a single controller. Other issues in the context of multiple controllers are planned to be pursued in a future work.
- We consider that all network devices are connected only via OpenFlow enabled switches.
- SDN-RBACa model for access control administration assumes that administrative users are enabled for SDN controllers. How to implement and authenticate administrative users in SDN environment is out of scope of this dissertation.
- Several SDN controllers are available but this dissertation primary focuses on Floodlight platform which is one of the widely used platforms for SDN academic research.

## 1.6 Summary of Contributions

The central contributions of this dissertation are as follows:

- We have developed and implemented a foundational role-based access control model (*SDN-RBAC*) for SDN applications. We have identified different approaches in which the system can handle application sessions in order to reduce exposure to the network attack surface in case of application being compromised, buggy, or malicious. Through proof-of-concept prototype, we have implemented our model with multi-session support in Floodlight controller and used hooking techniques to enforce the security policy without any change to the code of the Floodlight framework. We verified the model's usability and effectiveness against unauthorized access requests by controller applications and showed how the framework can identify application sessions and reject unauthorized operations in real time.

- We have developed ParaSDN, an access access control model to allow system administrators to specify granular access controls for SDN applications using the concept of parameterized roles and permissions. A proof of concept prototype has been implemented in an SDN controller to demonstrate the applicability and feasibility of our proposed model by enhancing access control granularity for SDN with support of role and permission parameters.

- We have introduced an approach for creating custom SDN operations to extend the capabilities of SDN services and provide fine grained custom permissions specialized for the administration of access control in SDN. These custom SDN operations are introduced to enable the creation of administrative units necessary for access control administration. Through proof of concept prototype and use cases, we demonstrate the usability of custom permissions and show how they facilitate the administration of access control in SDNs.

- We have presented SDN-RBACa, an administrative model to manage the access control actions that define network app authorizations.

## 1.7 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 covers background and literature review. It provides an overview of the SDN architecture and its need for access control. We then discuss the literature of access control for SDN and present a formal access control model we have developed and published [4] as an initial effort towards understanding access control for SDN. Chapter 3 presents SDN-RBAC, a foundational model to enable role-based access control model for SDN applications. It discusses different approaches for handling sessions and discusses a use case, framework implementation and performance evaluation. We also show model's usability and effectiveness in a controller framework. The model has been published in [3]. In Chapter 4, we propose ParaSDN, an access control model to allow system administrators to specify granular access controls for SDN applications using the concept of parameterized roles and permissions. Through a proof of concept prototype in an SDN controller, we demonstrate the applicability and feasibility of our proposed model in enhancing access control granularity for SDN with support of role and permission parameters. The model has been submitted for publication. Chapter 5 presents SDN-RBACa, an administrative model to manage the access control actions that define network app authorizations. The chapter also discusses an approach to extend the capabilities of SDN controller via creating custom SDN operations to enable the creation of custom permission necessary for the administration of access control in SDN. We also demonstrate the usability of custom permissions and show how they enable and facilitate the administration of access control in SDN. The work in this chapter has been submitted for publication. Chapter 6 concludes the dissertation and give a discussion about the future work.

# CHAPTER 2: BACKGROUND AND LITERATURE REVIEW

In this chapter, we provide an overview of the three-layered SDN architecture and discuss the literature of access control in SDN.

## 2.1 Overview of SDN Architecture

Software defined networking (SDN) decouples the network control and data planes. The network intelligence and state are logically centralized and the underlying network infrastructure is abstracted from applications. SDN enhances network security by means of global visibility of the network state where a conflict can be easily resolved from the logically centralized control plane. Hence, the SDN architecture empowers networks to actively monitor traffic and diagnose threats to facilitate network forensics, security policy alteration, and security service insertion. The separation of the control and data planes, however, opens many security challenges. Figure 2.1 shows the general architecture of SDN.

## 2.2 The Need for Access Control for SDN Apps

SDN has two principal properties which make the foundation of networking innovation on one hand, and the basis of security challenges on the other. First, the ability to control a network by software, and second, centralization of network intelligence in network controllers. Since most of the network functions can be implemented as SDN applications, malicious applications if not stopped early enough, can spread havoc across a network.

Applications can pose serious security threats to network resources, services and functions. In SDN, applications running on the controller implement a majority of the functionalities of the control plane and are typically developed by other parties than the controller vendors. These applications inherit the privileges of access to network resources, and network behavior manipulation mostly without proper security mechanisms for protecting network resources from malicious activities. Hence, authentication and authorization of the increasing number of applications in

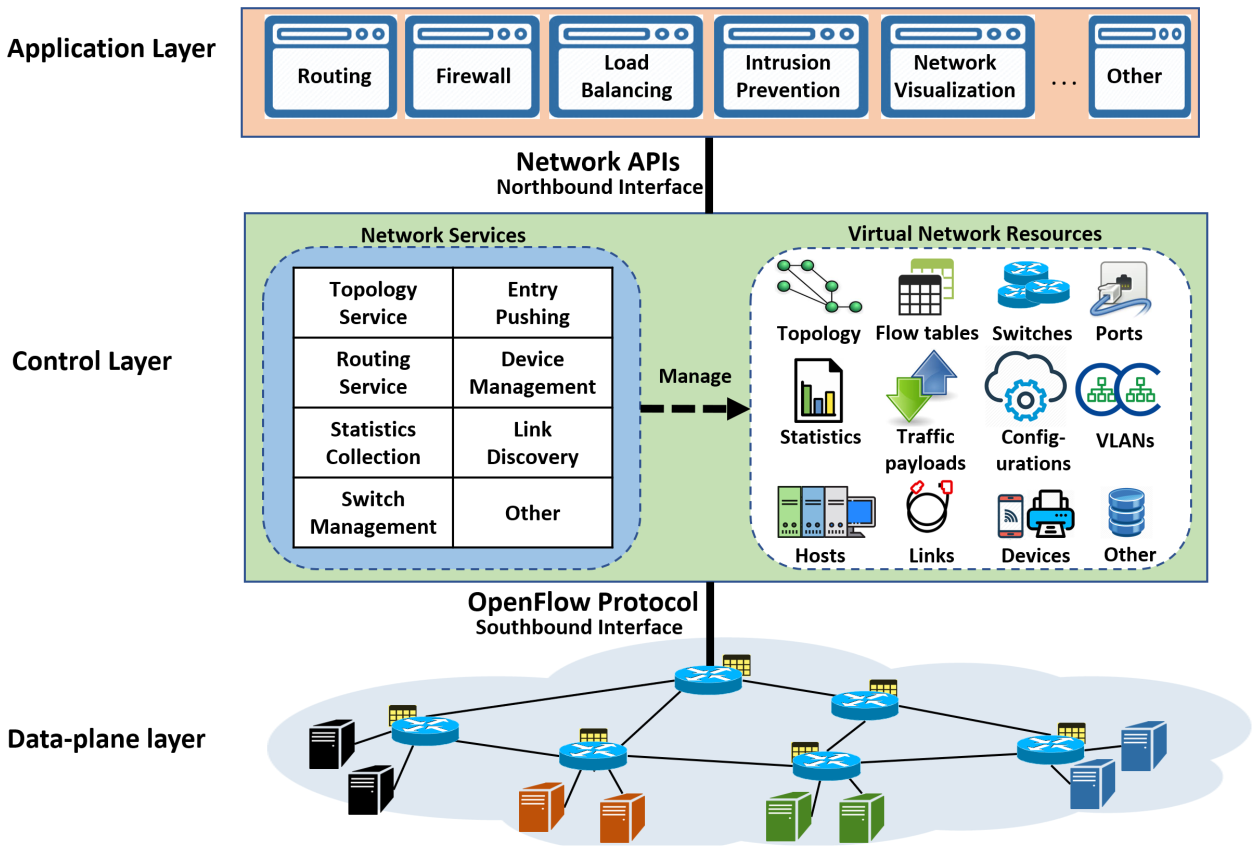


**Figure 2.1**: General Overview of SDN Architecture.

programmable networks with centralized control architecture is a major security challenge.

Today there are no existing authorization mechanisms to establish a trust relationship between the controller and applications in SDNs. Hence, a malicious application can potentially create havoc in the network since the SDN controllers provide abstractions that are translated to configuration commands for the underlying infrastructure by applications. Since applications implement most of the network services in SDN, proper access control is needed to ensure the security of a network.

## 2.3 SDN Planes and Security Perspectives

### 2.3.1 Control Plane

Because the control plane (e.g., Floodlight controller) is a centralized decision-making entity, it can be highly targeted for compromising the network or carrying out malicious activities in the network due to its pivotal role. One of the main security challenges and threats existing in the control plane is threats from applications which requires building access control systems to prevent unauthorized access to network resources.

Applications implemented on top of the control plane can pose serious security threats to the control plane. Generally, the controller security is a challenge from the perspectives of controller capability to authenticate applications and authorize resources used by applications with proper isolation, auditing, and tracking.

### 2.3.2 Application Plane

The network applications dictate the behavior of the forwarding devices. The applications submit high-level policies to the control plane, which is responsible for enforcing these policies by implementing them as flow rules on network forwarding devices. An SDN application plane consists of one or more network applications (e.g. security, visualization etc.) that interact with controller(s) to utilize abstract view of the network for their internal decision making processes.

Many different SDN applications have already been developed, and the the current focus is to have an App Store support for SDNs [62], where customers can dynamically download and install network apps. Most SDN applications fall into one of the following five categories: traffic engineering, mobility and wireless, measurement and monitoring, security and dependability, and data center networking.

### 2.3.3 Data Plane

The data plane is composed of networking equipments such as switches specialized in packet forwarding. However, unlike traditional networks, these are just simple forwarding elements with no embedded intelligence to take decisions. These devices communicate through standard OpenFlow interface with the controller. An OpenFlow enabled forwarding device has a forwarding table, which has three parts: 1) Rule matching; 2) Actions to be executed for matching packets; and 3) Counters for matching packet statistics. The rule matching fields include Switch port, Source MAC, Destination MAC, Ethernet Type, VLAN ID, Source IP, Destination IP, TCP Source Port, TCP Destination Port. A flow rule may be defined a combination of these fields.

Because network applications dictate the behavior of the forwarding devices in the data plane, network devices in the data plane need to be secured from unauthorized access by apps because flow rules inserted by apps reflect the network security policy.

## 2.4 Access Control for SDN apps: Literature Review

A number of security issues have been identified concerning SDN applications [2, 18, 19, 32] with specific issues related to application authorization. Several approaches have been proposed to protect the SDN resources from unauthorized access by network applications and limit access rights of applications to SDN resources. We classify the application authorization into two main categories. Firstly, permission-based application authorization which includes techniques wherein application authorization is driven by assigning a permission set to applications. Secondly, role-based application authorization in which application authorization is driven by assigning a role to applications.

### 2.4.1 Capability based Approaches

The problem of granting network applications the full access to network resources without protection is identified in pemrOF [54]. The authors proposed a permission system to network applications. PermOF proposed a set of different permissions for apps to interact with the underlying network, which are then checked and enforced at the API entry. The system's goal is to apply minimum privilege to the applications for protecting the network from attacks caused by unauthorized access.

OperationCheckPoint [48] is motivated by the consideration that applications should not be granted complete privilege of network access. The authors adopted the concept of PermOF and they, in a manner similar to that of the Android permissions system, defined a permission set to which the application must subscribe on initialization. Also, they implemented the system in Floodlight controller with a component for permission checking before authorizing application commands. OperationCheckpoint maps a unique application ID to the set of permissions granted to that application. Network administrator can use this ID to add or remove permissions to applications with an ability for applications to query the controller and discover their assigned permissions. For securing and hardening the methods of Floodlight, OperationCheckpoint needs to scan all related functions (java methods) in Floodlight source code and modify the function if it performs an operation related to each of the permissions in the permission set.

Also, inspired by Android permission system, [36] proposed a permission system based on OF messages' states. The permission system has five states for each of the permissions. The state is used as the unit to which the permission details can be applied. They proposed two main permission sets in the permission policy of SDN applications: OpenFlow message permission set and OS resource permission set. The permission sets of the network application can be structured in a format similar to XML. When an application tries to perform an action, a wrapper function verifies whether the action is allowed, based on the permission policy of the application.

Banse et al [9] proposed a permission system which allows for authorization and access control for network resources. In their design, network resources managed by a controller must not be accessible by applications that do not possess the appropriate permission. This registration process needs to incorporate an identification of the application as well as the negotiation of access control mechanisms, such as permissions. The system provides a registration service which applications can use to register themselves. The registration service includes application identification as well as a permission set negotiation process. Before authorizing access requests, a check against the set of allowed permissions is performed.

The authors in [39] introduced a security framework called AEGIS to prevent malicious network applications from misusing controller APIs. In their system, security access rules are defined based on the relationships between applications and data in the SDN controller. For performing a permission check, the usage of each API call is verified in real time using these security access rules. The system uses API hooking to intercept the execution flow of network applications and protect the controller from malicious applications.

In all the aforementioned approaches, there is a direct relation between network operations and applications which complicates the management of the permission system which is of a major consideration in access control.

### 2.4.2 Role-based Approaches

Prior to SE-Floodlight controller, as a solution to the problem of malicious applications, Porras et al proposed FortNOX [41], a more basic version of SE-Floodlight implementation on the NOX controller. Similarly, FortNOX implements role-based authorization for determining the security authorization of each OpenFlow application with three roles. SE-FloodLight [42] is an extension and improvement of the FortNOX system.

SM-ONOS [56] proposed a permission system at four-level granularity. First, code packages are classified as either app or non-app OSGi bundles. Next, app bundles are assigned either admin or user role with the appropriate permissions. Non-administrative API-permissions then granted to apps followed by network-level permissions. Based on API-level permissions from SM-ONOS, [50] proposed information flow control among apps for the ONOS controller.

Tseng et al [49] proposed Controller DAC to prevent SDN controllers from API abuse. Controller DAC has a policy engine which predefines an APIs request thresholds for each OpenFlow app and their permission scope. They also predefined a priority for each app either individually or via the API-role. By referring to [41, 42], Controller DAC proposes three API-roles Admin, Security, and DEFAULT with different priorities representing different authority levels.

None of the previous works comply with the standard RBAC model and none of them describe the concept of sessions for SDN apps. However, in our work we describe different approaches for handling and deploying sessions in the context of standard role-based access control. Besides, we propose a fine-grained role based access control for SDN using parameterized roles and permissions.

Because SDN's motivation is to simplify network management, and because RBAC's motivation is to simplify administration of authorizations, it is very important to think about the administration of access control in SDN. In this regard, several administrative models have been proposed in the literature for RBAC administration [16, 17, 33, 37, 43, 44, 52]. To the best of our knowledge, this dissertation discusses administration of access control in SDN for the first time in literature.

## 2.5 Access Control Model for SE-Floodlight Controller

### 2.5.1 Overview

One of the inherent features of SDN is to enable the communication between OpenFlow applications and network devices in the data plane. This interaction should be coordinated by SDN controllers to control access to different network resources and enforce security policies in the data plane. SDN controllers provides this mediation to manage how OpenFlow applications deploy network functionalities and manipulate the network behavior.

The application plane within a single network may consist of multiple applications that send OF messages to access the data plane for flow rules installation, network device configuration, and inquiring about network state information. Application co-existence calls attention to two main SDN network security issues. First, it emphasizes the need for an authorization mechanism for managing application permissions to issue various network operations. Second, it points out the need for resolving the conflicts that may arise among various application commands.

To address these issues, in this section we present a formal access control model we have developed as an initial effort towards understanding and designing an effective access control for SDN.

**Table 2.1**: Types of data exchange operations along with the minimum authorization role [42].

| Type ID | Type of Data Exchange Operation | Minimum Auhorization Role | Open Flow Message Type |
|---|---|---|---|
| *t1* | Flow removal messages | APP | OFPT_FLOW_REMOVED |
| *t2* | Flow error reply | APP | OFPT_ERROR |
| *t3* | Echo requests | APP | OFPT_ECHO_REQUEST |
| *t4* | Echo replies | APP | OFPT_ECHO_REPLY |
| *t5* | Barrier requests | APP | OFPT_BARRIER_REQUEST |
| *t6* | Barrier replies | APP | OFPT_BARRIER_REPLY |
| *t7* | Switch get config | APP | OFPT_GET_CONFIG_REQUEST |
| *t8* | Switch config reply | APP | OFPT_GET_CONFIG_REPLY |
| *t9* | Switch stats request | APP | OFPT_STATS_REQUEST |
| *t10* | Switch stats report | APP | OFPT_STATS_REPLY |
| *t11* | Packet-In return | APP | OFPT_PACKET_IN |
| *t12* | Flow rule mod | APP | OFPT_FLOW_MOD |
| *t13* | Packet-Out | SEC | OFPT_PACKET_OUT |
| *t14* | Vendor actions | ADMIN | OFPT_VENDOR |
| *t15* | Vendor features | ADMIN | OFPT_FEATURES |
| *t16* | Switch port status | ADMIN | OFPT_PORT_STATUS |
| *t17* | Switch port mod | ADMIN | OFPT_PORT_MOD |
| *t18* | Switch set config | ADMIN | OFPT_SET_CONFIG |

The formal model is designed based on SE-Floodlight authorization framework as a reference. This formal model and discussions on associated security aspects along with proposed solutions for security flaws has been published in the following venue [4].

- Abdullah Al-Alaj, Ravi Sandhu, and Ram Krishnan. A Formal Access Control Model for SE-Floodlight Controller. In Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, pages 1-6. ACM, 2019.

Although SE-Floodlight authorization system does not adopt RBAC sessions, it adopts the concept of role hierarchy, inheritance relation between roles which, compared to other authorization systems [49, 50, 56], make it the reference for our initial work for formalizing and analyzing apps authorization in SDN.

### 2.5.2 Authorization Framework of SE-Floodlight Controller

Prior studies have addressed access rights of SDN apps to protect against insecure access to data plane resources [41]. SE-Floodlight [21, 42] is a security extension for Floodlight in which Porras et al *informally* proposed Role-based access control (RBAC) to enforce the security policy on the data exchange operations between network apps and the forwarding switches in addition to flow rule conflict resolution strategy driven by roles.

The SE-Floodlight authorization system aims to control access to all OpenFlow messages exchanged between apps and the data plane. Each data exchange operation is abstracted using an OpenFlow operation type. For example, operations like insertion, deletion, and update of a flow rule are all of type 'Flow rule mod'. In SE-Floodlight, types of data exchange operations are assigned to roles and then roles are assigned to apps.

The authorization system identifies three authorization roles, namely, ADMIN, SEC, and APP with a total order role hierarchy. For instance, apps in SEC role indirectly have permission to receive packet-in notifications which is originally assigned to the lower role APP. This is analogous to role inheritance relation in RBAC model [20, 46].

Table 2.1 identifies the types of data exchange operation types that alternate between apps and OpenFlow switches along with the minimum role that must be assigned to an app to perform an operation of that type. The table also shows, for each operation type, the corresponding OpenFlow message type required to carry the operation.

With app co-existence, an app may insert a flow rule that causes a conflict with another pre-existing flow rule in the switch. For resolving conflicts, SE-Floodlight associates each role with a priority limit that represents the maximum priority value that can be assigned to flow rules produced by apps of this role. Flow rules produced by an app of a specific role have higher precedence than rules produced by apps in a lower role. An app also uses this value to prioritize its own set of flow rules within the sub-range of priorities corresponding to its role.

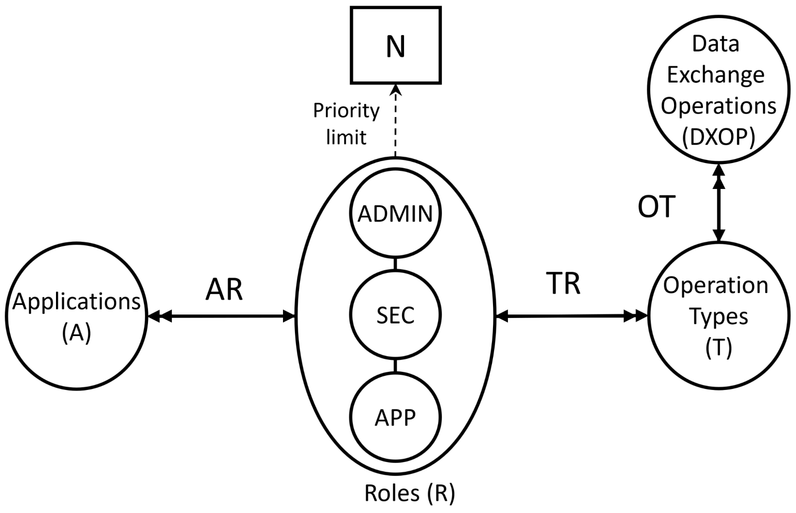


**Figure 2.2**: SE-Floodlight conceptual authorization model [1].

### 2.5.3 Formalized SE-Floodlight Access Control Model

In this section we formalize the authorization model of SE-Floodlight [42] and describe its components.

#### Overview

The basic components of SE-Floodlight authorization model include: Apps (A), Roles (R), Data Exchange Operations (DXOP), and types of DXOPs. We also discuss the credential entity which is implicitly included in the model. The model and the relations between the components are shown in Figure 2.2, and discussed below.

**Apps (A):** This component represents the two types of OpenFlow applications. Firstly, remote applications installed in a remote host and utilizes REST API calls to communicate with controller and written in any appropriate programming language. Secondly, local applications installed in the same process context as the controller and written using the programming language API that the controller uses, which is java in SE-Floodlight case.

**Roles (R):** Roles in SE-Floodlight are used for two main purposes: application permission autho-

[1] Arrows denote binary relations with the single arrowhead indicating the one side and double arrowheads the many side.

rization and role based conflict resolution. As an authorization component, a role is a ion of data exchange operations which can be assigned to different OpenFlow applications. Three default roles are implemented in a hierarchical authorization scheme, namely ADMIN, SEC and APP. Only one role can be assigned to each application.

**Data Exchange Operations (DXOP) and Operation types (T):** These operations represent OpenFlow messages exchanged between the dataplane and the apps. These operations belong to different operation types. For example, flow table modification messages add, modify, modify_strict, delete, and delete_strict are grouped under the type 'Flow rule mod', represented by the message type OFPT_FLOW_MOD. Also, operations that query the system for statistical information about flows, ports, tables, queues, etc, are all of type 'Switch stats request', represented by OFPT_STATS_REQUEST message type. Table 2.1 summarizes all these data exchange operations and their corresponding messages within an OpenFlow v1.0 stack.

**Credentials:** Credentials are used for both authorization and authentication. Each app is uniquely identified by an administrator-assigned credential which also contains the authorization role assigned to the app. The credential is added to each message sent by the app. When a message is submitted, the role embedded within the credentials is extracted to identify the app and its role.

### Formal SE-Floodlight Access Control Model

For this model, we assume the SDN infrastructure has multiple switches controlled by a single controller within the same slice. For simplicity and easier reference, we created a basic formal model without flow rule conflict resolution in Table 2.2 and an extended model including the conflict resolution in Table 2.3.

As shown from the definition in Table 2.2, an app can be assigned to only one role denoted by *AR* relation. *TR* relation denotes that an operation type can be assigned to one role. The type of each data exchange operation can be specified using *type* function. The *Authorization Rule* is stated based on all the previously defined relations and functions considering the effect of role inheritance.

**Table 2.2**: SE-Floodlight Authorization Model Definitions without Flow Rule Conflict Resolution.

**- Basic Sets and Functions:**

*A*: a finite set of OpenFlow apps.

*T:* a finite set of types of data exchange operations.

*R* = {*ADMIN*, *SEC*, *APP*}: a fixed set of three roles.

>: a total order on *R* where *ADMIN* > SEC and *SEC* > *APP*.

*AR* $\subseteq$ *A* $\times$ *R*, a many-to-one relation, i.e., (*a*,$r_1$)$\in$*AR* $\wedge$ (*a*,$r_2$)$\in$*AR* $\Rightarrow$ $r_1 = r_2$, mapping each app to one role.

*TR* $\subseteq$*T*$\times$*R,* a many-to-one relation, i.e., (*t*,$r_1$)$\in$*TR* $\wedge$ (*t*,$r_2$)$\in$*TR* $\Rightarrow$ $r_1 = r_2$, mapping each operation type to one role.

*DXOP*: a set of possible data exchange operations where each operation *op* $\in$*DXOP* contains a flow rule and a priority if $o = '\text{add flow rule}'$.

*type: DXOP* $\rightarrow$ *T*, a function specifying the type of each operation. Equivalently viewed as a many-to-one relation *OT* $\subseteq$ *DXOP* $\times$ *T* , where (*o*,$t_1$)$\in$*OT* $\wedge$ (*o*,$t_2$)$\in$*OT* $\Rightarrow$ $t_1 = t_2$.

**- Authorization Rule:**

*Authorization_rule: A* $\times$ *DXOP* $\rightarrow$ {*T, F*}, checks whether *a* $\in$ *A* has the right to perform an operation *o* $\in$ *DXOP.*

*Authorization_rule* (*a* : *A*, *o* : *DXOP*) $\equiv$($\exists r_1, r_2 \in$ *R*$\cdot$ $(a, r_1) \in AR \wedge (type(o), r_2) \in TR \wedge r_1 \geq r_2)$.

The authorization model is extended in Table 2.2 in which the rule conflict algorithm contributes to authorizing 'add flow rule' operations. The function *priorityLimit* assigns a natural number to each role. This number represents the maximum priority an app in this role can assign to new flow rules submitted with 'add flow rule' operation. It is used to resolve conflicts between different flow rules. *Authorization_rule*$_{op='add\, alow\, rule'}$ checks, using the *RCA* function, whether an app has the right to insert a new flow rule . Finally, *Authorization_rule*$_{op\in DXOP-'add\, flow\, rule'}$ checks whether an app has the right to perform all operations other than 'add flow rule' operations that are not mediated by the *RCA* function.

It should be noted that if *Authorization_rule*$_{op='add\, alow\, aule'}$ returns true, which means a successful addition of new flow rule, then the access control model updates the set *FT* of the target switch by adding the new flow rule and removing the conflicting rule, if any. This happens only

**Table 2.3**: SE-Floodlight Authorization Model Definitions with Flow Rule Conflict Resolution.

**- Basic Sets and Functions:**

All basic sets and functions from Table \ref{P01-tbl:SEFloodlightModelDefinitions}.

*FR*: a set of all possible flow rules where for each $fr_i \in$ *FR* there should be a priority.

*priority_limit*: *R* $\rightarrow \mathbb{N}$, the mapping of role to the highest priority an app in *r* $\in$ *R* may assign to its flow rules, where *priority_limit*(*ADMIN*) > *priority_limit*(*SEC*) > *priority_limit*(*APP*).

*S*: Set of switches in the network slice.

*FT*: *S* $\rightarrow 2^{FR}$, the set of flow rules currently in a switch's flow table.

*rule*: *DXOP* $\rightarrow$ *FR*, a function that returns the flow rule $fr_c \in FR$ of an operation *op* $\in$ *DXOP* given that $type(op) =$'Flow Rule Mod'.

*priority: FR* $\rightarrow \mathbb{N}$, the mapping of a flow rule $fr_c \in FR$ to its priority.

*RCA(fr$_c$: FR, pr$_c$*:$\mathbb{N}$, s$_t$:S) $\rightarrow$ {*Reject*, *Add*, *Exchange*}, a function uses rule-based conflict analysis described in [42] that returns the result of a request to add of new flow rule *fr$_c$* into *FT*($s_t$) submitted with priority *pr$_c$*. *'Reject*', *'Add*', *or 'Exchange'* indicates whether $fr_c$ is, rejected, added without removing pre-existing rules, or exchanged with a conflicting flow rule $fr_i \in FT(s_t)$, respectively.

**- Authorization Rules:**

*Authorization_rule*$_{op='add\,flow\,rule'}$ *: A* $\times$*S* $\rightarrow$ {*T, F*}, checks whether *a* $\in$ *A* has the right to insert a flow rule $rule(op)$ into *FT*($s_t \in S$).

*Authorization_rule*$_{op='add\,flow\,rule'}$ *(a* : *A*, *s$_t$*:*S*) $\equiv$
$(\exists r_1, r_2 \in$ *R*$\cdot$ $(a, r_1) \in AR \wedge (type(op), r_2) \in TR \wedge r_1 \geq r_2) \wedge$
$(RCA(rule(op), priority(rule(op)), s_t) \in \{Add, Exchange\})$.

*Authorization_rule*$_{op \in DXOP-'add\,flow\,rule'}$*: A* $\times$ *S* $\rightarrow$ {*T, F*}*,* checks whether *a* $\in$ *A* has the right to perform a non-flow-rule-insertion operation.

*Authorization_rule*$_{op \in DXOP-'add\,flow\,rule'}$ *(a* : *A*, *s$_t$*:*S*) $\equiv$
$(\exists r_1, r_2 \in$ *R*$\cdot$ $(a, r_1) \in AR \wedge (type(op), r_2) \in TR \wedge r_1 \geq r_2)$

**Table 2.4**: Administrative Model for SE-Floodlight.

| **Function** | **Condition** | **Update** |
|---|---|---|
| $addApp(a)$ | $a \notin A$ | $A' = A \cup \{a\}$ |
| $deleteApp(a)$ | $a \in A \wedge (a,r) \in AR$ | $AR' = AR \setminus \{(a,r)\}$, $A' = A \setminus \{a\}$ |
| $addType(t)$ | $t \notin T$ | $T' = T \cup \{t\}$ |
| $deleteType(t)$ | $t \in T \wedge (o,t) \in OT \wedge (t,r) \in TR$ | $OT' = OT \setminus \{\forall (o,t) \in OT\}$, $TR' = TR \setminus \{(t,r)\}, T' = T \setminus \{t\}$ |
| $addRole(r)$ | $r \notin R$ | $R' = R \cup \{r\}$ |
| $deleteRole(r)$ | $r \in R \wedge (a,r) \in AR \wedge (t,r) \in TR$ | $AR' = AR \setminus \{\forall (a,r) \in AR\}$, $TR' = TR \setminus \{\forall (t,r) \in TR\}$, $R' = R \setminus \{r\}$ |
| $assignApp(a,r)$ | $a \in A \wedge r \in R \wedge (a,r) \notin AR$ | $AR' = AR \cup \{(r,a)\}$ |
| $revokeApp(a,r)$ | $a \in A \wedge r \in R \wedge (a,r) \in AR$ | $AR' = AR \setminus \{(a,r)\}$ |
| $assignType(t,r)$ | $t \in T \wedge r \in R \wedge (t,r) \notin TR$ | $TR' = TR \cup \{(t,r)\}$ |
| $revokeType(t,r)$ | $t \in T \wedge r \in R \wedge (t,r) \in TR$ | $TR' = TR \setminus \{(t,r)\}$ |
| $assignOp(o,t)$ | $o \in DXOP \wedge t \in T \wedge (o,t) \notin OT$ | $OT' = OT \cup \{(o,t)\}$ |
| $revokeOp(o,t)$ | $o \in DXOP \wedge t \in T \wedge (o,t) \in OT$ | $OT' = OT \setminus \{(o,t)\}$ |

if the result of rule conflict algorithm *RCA* returns 'Add' or 'Exchange'. Also, the model registers the flow rule priorities so that they can be used in future authorization decisions.

### Administrative Model

Next we discuss the administrative model that is used for the creation and maintenance of the system's basic element sets, functions, and relations. It is assumed that all administrative functions are performed by a network operator user with enough privileges. Table 2.4 formally specifies the administration functions to manage the apps, roles, operations, and operation types. The second column shows the condition required to perform the function and the third column shows the corresponding updates to the authorization system.

As shown in Table 2.4, administration functions for managing registration and de-registration of apps are *addApp* and *deleteApp*, respectively. Roles are created and removed from the system using *addRole* and *deleteRole* functions. Types are created and removed from the system using *addType* and *deleteType* functions. When a role is deleted, all assignment relations between the deleted role and any app or operation type must be found and deleted from the system as shown in

**Table 2.5**: Configuration of the Formal Access Control Model defined in Table 2.2 for the Use Case Scenario in Section 2.5.4.

$A = \{LS, LB, NIP, FW, OC\}$,

$R = \{APP, SEC, ADMIN\}$ with a total order $>$ on R ,as defined in Table 2.2,

$T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}, t_{17}, t_{18}\}$, as labled in Table 2.1,

$AR = \{(LS, APP), (LB, APP), (NIP, SEC), (FW, SEC), (OC, ADMIN)\}$,

$TR = \{(t_i, APP), (t_{13}, SEC), (t_j, ADMIN)|(t_i \in T|1 \leq i \leq 12, t_j \in T|14 \leq j \leq 18)\}$,

$DXOP = \{'add\,flow\,rule', 'packet\,in', 'flow\,stats', 'packet\,out'\}$,

$Type('add\,flow\,rule') =' Flow\,rule\,mod'$, $Type('packet\,in') =' Packet - In\ return'$,
$Type('flow\,stats') =' Switch\,stats\,request' =' Switch\,stats\,report'$,
$Type('packet\,out') =' Packet - Out'$,

$AuthorizationRule(LS,' add\,flow\,rule') = true$,
$AuthorizationRule(LB,' add\,flow\,rule') = true$,
$AuthorizationRule(FW,' add\,flow\,rule') = true$,

$AuthorizationRule(LS,' packet\,in') = true$, $AuthorizationRule(LB,' packet\,in') = true$,
$AuthorizationRule(NIP,' packet\,in') = true$,
$AuthorizationRule(FW,' packet\,in') = true$ $AuthorizationRule(OC,' packet\,in') = true$,

$AuthorizationRule(LB,' flow\,stats') = true$,
$AuthorizationRule(FW,' packet\,out') = true.$

the third column.

Functions *assignApp* and *revokeApp* are used to create and delete a relation between apps and roles. Operation types are assigned to roles using *assignType* and revoked using *revokeType* function. Operations are assigned to their types using *assignOp* function and revoked using *revokeOp* function.

### 2.5.4 Use Case Scenario

Applying this formal model by network operators depends on the use case under implementation. In this section we configure the formal model for a use case scenario and show the relevant authorization aspects.

In this use case we have five OpenFlow apps, namely Learning Switch (*LS*), Load Balancer (*LB*), Network Intrusion Prevention (*NIP*), Firewall (*FW*), and Operator Console (*OC*) app. *LS* app requires insertion of flows in switches that routes packets of all devices in the topology after they have been learned using Packet-In messages, so it needs permission to operation types ‘Flow rule mod’ and ‘Packet-In return’.

The *LB* app requires permission to collect flow statistics from the switches and to distribute traffic among deferent servers/links accordingly. It requires permission to operation of type ‘Flow rule mod’, ‘Switch stats request’ , and ’Switch stats report’ so it is enough to be assigned the APP role.

The *NIP* app detects and blocks intrusion attempts on the network. In order to do so, it requires permission to receive packet-in notifications for performing packet inspection to drop malicious traffic and it needs permission to insert flow rules for forwarding sanitized traffic to the correct destination [55]. The Operation types required by this app are ‘Flow rule mod’ and ‘Packet-In return’. Despite that it is enough to assign it to the APP role, this app is intended to enforce security policy and so should be assigned to the SEC Role in order to replace conflicting flow rules inserted by apps in APP role.

The *FW* app [55] that performs basic firewall tasks such as enforcing Access Control List (ACL) on OpenFlow switches. For each incoming Packet-In message, the firewall compares the header fields against each rule in the sorted list sequentially from the highest priority. The app matches every single packet against the firewall rules. Hence, the Forwarding app uses Packet-Out messages to forward each packet. It forwards a packet through sending a Packet-Out message with an appropriate action for an ALLOW decision, while it drops a packet through sending a Packet-Out message without specifying an Output action for a DROP decision. So this app needs permission to ‘Packet-In return’ and ‘Packet-Out’ operation types.

Finally, the (*OC*) app is capable of performing all operation types and to configure and read network state, so its assigned the ADMIN role. The configuration of the formal access control model for this use case scenario is given in Table 2.5.

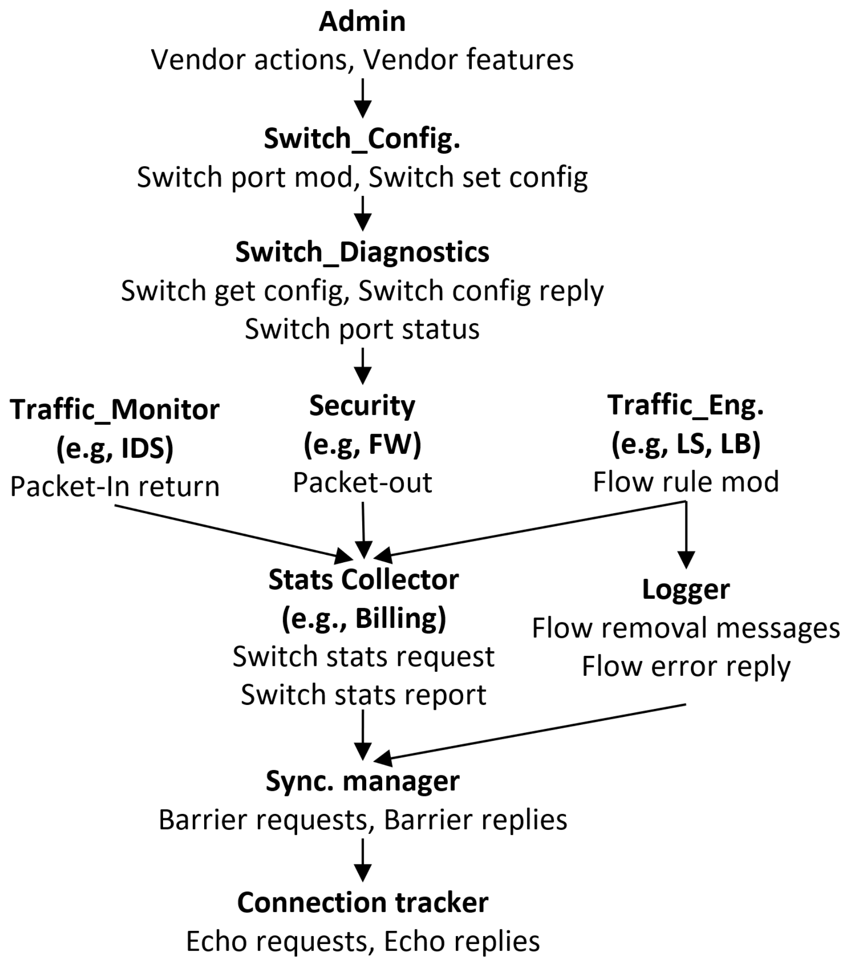


**Figure 2.3**: Proposed Role Hierarchy.

### 2.5.5 Discussion and Proposed Extensions

In this section we discuss some of the issues that may violate access control principles and call for modifications in the model.

#### Over-privileged apps

SDN apps should be confined to the principle of least privilege. However, following the fixed set *R* and the *TR* relation as shown in Table 2.1 may grant an app the permission to one or more operations. All apps in the lowest role *APP* will have the permission to add flow rules whereas some apps don't require this permission. For example, a billing app requires only to read statistics of the port connected to a host's device (NIC). It computes a monthly bill for a customer based on the sent and received bytes. APP role grants this app the permission to insert flow rules which violates the least privilege principle and this could be maliciously exploited to attack the controller or other apps. The network state can be modified and then the controller and other apps might take decisions based on this inconsistent state.

**App upgrading problem**

To satisfy network security needs and respond to security threats, flow rules inserted by security apps should have higher priority than those of traffic engineering apps. Therefore, based on the SE-Floodlight access control model, such apps will be upgraded from APP to SEC role only to satisfy the priority requirement. As a result, they will be granted permission to 'packet-Out' operation type. This is a violation of the least privilege principle of access control. The NIP app, discussed in Section 2.5.4, is an example of this case.

**Limitations of role hierarchy**

Some apps might need to perform different set of operations and the network operator wants to assign them the same priority limit. This is impossible in SE-Floodlight authorization system due to the total order relation between roles. Furthermore, assigning roles to apps based on the tasks they achieve is limited with the existence of only three role levels and the way operation types are assigned to roles in Table 2.1.

**App downgrading problem**

When a role *r* is assigned to an app *a*, it provides the permission to possibly multiple operation types. If the network operator later wants to downgrade *a*, by revoking only one operation type *t* from *a* for example. Revocation of *t* cannot be directly applied to *a*, it should be done through *revokeType*(*t*,*r*) function. However, this kind of revocation doesn't work because *r* is most likely shared by multiple apps and this will downgrade all apps in *r*. This can be done only by creating a new role $r'$ that has all operation types in *r* except *t* then applying $assignApp(a, r')$. This scenario calls for a more flexible role hierarchy.

For addressing the above problems, we refine the total order hierarchy and propose a partial order role hierarchy as shown in Figure 2.3. We modify the *AR* relation to be many-to-many relation to allow for assigning multiple roles to one app. Types of operations are assigned to roles based on the task they achieve taking into consideration inherited permission types. In this hierarchy we consider only app interactions with the forwarding infrastructure and omit app requests to read and write controller data stores that maintain information about end hosts, network topology, etc.

Also, roles are organized based on the sensitivity of operation type set managed by this hierarchy. An operation type that may alter the network state should be part of higher roles whereas lower roles should encapsulate non-harmful operation types. The role name indicates the general function achieved by operation types in the role and its inherited permissions. For resolving rule conflicts, higher roles should have higher priority limit as they have more power in the authorization system. Incomparable roles should be assigned same priority limit or based on the network operator's configuration.

App's access rights can be managed by manipulating *AR* and *TR* relations or by creating/deleting a role inheritance relation. Each role in this partial order hierarchy encapsulates less permission types compared to the roles of the total hierarchy of SE-Floodlight and the application-level roles of [56]. We don't propose direct assignment of permission types to apps since we believe it would be a management burden for network operators.

This role hierarchy avoids the limitations of the total role hierarchy with a more flexible and finer grained operation type-to-role assignment. Based on this hierarchy, apps can be assigned appropriate roles and thus network operators can avoid *over-privileged apps*. Also, as a result of this flexible app-role assignment, *app upgrading problem* is fixed because priorities for incomparable roles can be configured by network operator. In this case app upgrading will not be conducted only for priority limit reasons.

*App downgrading problem* is solved by this hierarchy since each role has a small number of strongly related operation types that can be granted as all-or-none basis. As a result, for an app $a$ that is assigned role *r*, revoking type *t* from *a* can be done by invoking *revokeApp*(*a*,*r*) followed by $assignApp(a, r')$ where $r'$ is the immediate ascendant or *r*.

# CHAPTER 3: ENABLING ROLE-BASED ACCESS CONTROL FOR SDN APPLICATIONS

In this chapter, we discuss a role-based access control model for SDN controller apps, refered to as SDN-RBAC. We discuss different approaches in which the system can handle app sessions which help in applying the least of privilege principle. We also provid the functional system specifications required for app session management and for making access control decisions at the session level. We verify the model's usability and effectiveness by implementing a prototype in a popular SDN controller. Significant portion of this chapter has been published in the following venue [3].

- Abdullah Al-Alaj, Ram Krishnan, and Ravi Sandhu. SDN-RBAC: An Access Control Model for SDN Controller Applications. In 2019 4th International Conference on Computing, Communi- cations and Security (ICCCS), pages 1-8. IEEE, 2019.

## 3.1 Motivation and Background

SDN apps that are residing in the SDN controller and written in the same language of the controller are of a major security concern. This is because they are compiled as part of the controller and have direct access to various controller native classes, their methods and data. Intuitively, the more permissions available to an app the more resources accessible through these permissions, the more exposed the network attack surface. As a result, applying the principle of least privilege is vital in access control for SDN apps. The key idea is to minimize the amount of operations available to an app at a given time.

In SDN, it is most likely that one controller app performs several networking tasks, either sequentially or concurrently. If the app executes all these tasks in one session this means higher exposure to the network attack surface in case of app being compromised, buggy, or malicious. This ensures that cooperation of multiple sessions is required to perform all the tasks of a complete SDN app process, either sequentially or concurrently, so that accountability can be enforced and damage caused by either a session mistake, or an accident or deception can be avoided or minimized.

This initiates the need for serious efforts in creating access control models for SDN controllers where, usually, human being has no direct control of the running apps. To address this issue we propose a Role-based Access Control Model for the apps residing in the SDN controller.

In this context the concept of a session has several motivations. It supports the least privilege principle in the sense that an app can delay the activation of roles currently unused in a session until they really required [10,46]. Also, it permits delaying the creation of particular sessions until they really required. All these serve to reduce the amount of operations executable by an app at a given time which reduces the amount of resources accessible by these operations and thus the attack surface.

To address the above problems, in this chapter we depict our work in the design and implementation a formal role-based access control model (*SDN-RBAC*) for SDN applications that helps in applying least of privilege principle at the level of applications and their sessions. We also identify different approaches in which the system can handle application sessions in order to reduce exposure to the network attack surface and, as a result, reduce possible damage that the system incurs in case of application being compromised, malicious, has vulnerability, or even crashed.

## 3.2 The *SDN-RBAC* Model

### 3.2.1 Formal Model

In this section we introduce *SDN-RBAC* with its basic element sets and functions. Being able to create roles for SDN apps, which contain optimum number of permissions, is one of the challenges in SDN environment. We believe that deciding which permissions to assign to which roles is completely up to the app's function. Currently, there is no any reference standard that states which kinds of controller apps should use which kind of permissions. There is also no satisfactory system that can identify the permissions appropriate for the different categories of apps. We believe that this topic by itself is a research area that needs further study.

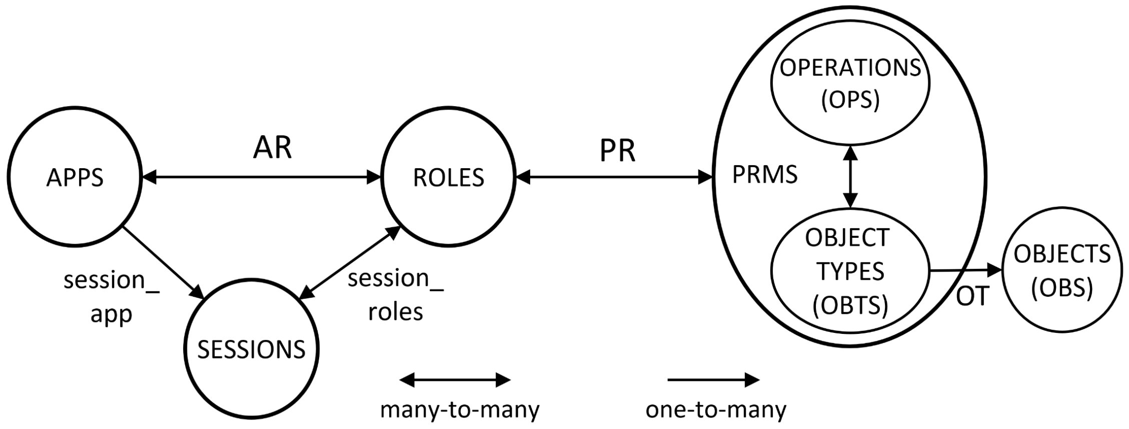


**Figure 3.1**: Conceptual *SDN-RBAC* Model.

SDN-RBAC has the following basic components: Controller Apps ($APPS$), Roles ($ROLES$), Operations ($OPS$), Objects ($OBS$), and Object Types ($OBTS$). The conceptual model and the relations between the components of *SDN-RBAC* are shown in Figure3.1, and discussed below.

- **Apps** ($APPS$)**:** The set of OpenFlow apps residing in the SDN controller.
- **Roles** ($ROLES$)**:** The authorization roles assigned to apps.
- **Objects** ($OBS$)**:** Data and objects (resources) managed by the controller and should be protected from unauthorized access. The controller manages these resources to maintain a consistent state of the network infrastructure. A specific port instance in a particular switch instance, and a device instance are examples of objects.
- **Object Types** ($OBTS$)**:** The type under which a specific object instance or group of object instances are categorized. For example, all port instances within the authority of ($VLANid = 10$) can be associated to the type 'PORT-VLAN-5' and all ports within ($VLANid = 10$) can be associated to the type 'PORT-VLAN-10'. Also, 'LINK-ACC' could be the type of all link instances attached to the hosts in the Accounting Department.
- **Operations** ($OPS$)**:**. Operations performed on objects and exposed by the controller as a service. For example, the Device Service exposes operations to query the list of devices/hosts

**Table 3.1**: Formal Definitions of *SDN-RBAC*.

| | |
|---|---|
| **-** | **Model Element Sets:** |
| - | $APPS$, $ROLES$, $OPS$, $OBS$ and $OBTS$, a finite set of OpenFlow apps, roles, operations, objects and object types, respectively. |
| - | $PRMS = 2^{OPS \times OBTS}$, the set of permissions. |
| - | $SESSIONS$, a finite set of sessions. |
| **-** | **Assignment Relations:** |
| - | $PR \subseteq PRMS \times ROLES$, a many-to-many mapping permission-to-role assignment relation. |
| - | $AR \subseteq APPS \times ROLES$, a many-to-many mapping app-to-role assignment relation. |
| - | $OT \subseteq OBS \times OBTS$, a many-to-one relation mapping an object to its type. |
| **-** | **Mapping Functions** |
| - | $assigned\_perms(r : ROLES) \rightarrow 2^{PRMS}$, the mapping of role *r* into a set of permissions. Formally, $assigned\_perms(r) \subseteq \{p \in PRMS \vert (p, r) \in PR\}$. |
| - | $app\_sessions(a : APPS) \rightarrow 2^{SESSIONS}$, the mapping of an app into a set of sessions. |
| - | $session\_app(s : SESSIONS) \rightarrow APPS$, the mapping of session into the corresponding app. |
| - | $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$, the mapping of session into a set of roles. Formally, $session\_roles(s) \subseteq \{r \in ROLES \vert (session\_app(s), r) \in AR\}$. |
| - | $type : OBS \rightarrow OBTS$, a function specifying the type of an object, where $(o, t_1) \in OT \wedge (o, t_2) \in OT \Rightarrow t_1 = t_2$ |
| - | $avail\_session\_perms(s : SESSIONS) \rightarrow 2^{PRMS}$, the permissions available to an app in a session = $\bigcup_{r \in session\_roles(s)} assigned\_perms(r)$. |

based on one of Mac Address, VLAN id, IPv4 Address, IPv6 Address, or a combination of them. Inserting flow rules and reading switch statistics are another examples of operations.

- **Sessions ($SESSIONS$):** A mapping between an app and an activated subset of app roles. An app can have multiple sessions and a session belongs to only one app.

As shown in Table 3.1, permissions $PRMS$ is the set of all possible combinations between the set of operations $OPS$ and object types $OBTS$. The function $type$ returns the type of an object that was associated to it via the $OT$ relation.

**Table 3.2**: Specifications of system functions.

| **Function** | **Authorization Condition** | **Update** |
|---|---|---|
| $createSession(a : APPS, s : SESSIONS, ars : 2^{ROLES})$ | $ars \subseteq \{r \in ROLES \mid (a, r) \in AR\} \wedge s \notin SESSIONS$ | $SESSIONS' = SESSIONS \cup \{s\}, app\_sessions'(a) = app\_sessions(a) \cup \{s\}, session\_roles'(s) = ars$ |
| $deleteSession(a : APPS, s : SESSIONS)$ | $s \in app\_sessions(a)$ | $app\_sessions'(a) = app\_sessions(a) \backslash \{s\}, SESSIONS' = SESSIONS \backslash \{s\}$ |
| $addActiveRole(a : APPS, s : SESSIONS, r : ROLES)$ | $s \in app\_tsessions(a) \wedge (a, r) \in AR \wedge r \notin session\_roles(s)$ | $session\_roles'(s) = session\_roles(s) \cup \{r\}$ |
| $dropActiveRole(a : APPS, s : SESSIONS, r : ROELS)$ | $s \in app\_sessions(a) \wedge r \in session\_roles(s)$ | $session\_roles'(s) = session\_roles(s) \backslash \{r\}$ |
| $checkAccess(s : SESSIONS, op : OPS, ob : OBS)$ | $\exists r \in ROLES : r \in session\_roles(s) \wedge ((op, type(ob)), r) \in PR$ | |

A role can be assigned multiple permissions as expressed by the $PR$ relation. An app might need to have multiple permissions to access different network resources which may result in requiring multiple roles. This is expressed by the $AR$ relation. The function $assigned\_perms$ is used by the system to get the set of permissions attached to a role.

An app may have multiple session instances running at the same time. Each session may have different combinations of active roles adequate for accomplishing its task. It is important for authorization purposes to identify all instances of an app sessions, the system uses the function $app\_sessions$ for this purpose.

Each session executes on behalf of only one app which is constant during the session's lifetime. The system uses the function $session\_app$ to identify this app. Generally, at the time of session establishment and during the lifetime of a session, an app can activate any subset of the roles attached to it that is suitable for the session's task to be accomplished. The function $session\_roles$ is used by the system to identify all roles currently active for a particular session. The effective permissions available to a session will then be the permissions assigned to all the effective roles activated by the app for that session.

The function $avail\_session\_perms$ returns the union of all permissions assigned to sesssion's active roles. It should be noted that object access requests from an app's session is authorized based on the type of the requested object. The *type* function is used for this purpose.

### 3.2.2 System Functions Specifications

System functions in SDN-RBAC define features for the creation and deletion of app sessions, role activation and deactivation in a session, and for calculation of an access decision. The specifications of system functions are shown in Table 3.2, and discussed below.

- **createSession**: The system function $CreateSession$ creates a new session $s \in SESSIONS$ for a given app $a \in APPS$ as owner and an active role set $ars \in ROLES$ to be used during the session lifetime. The function is valid if and only if the active role set is a subset of the roles assigned to that app. The system updates the set $SESSIONS$ by adding $s$ to it. This also updates $app\_sessions$ and the $session\_roles$ mappings.

- **deleteSession**: The function $deleteSession$ deletes a given session $s \in SESSIONS$ for a given app $a \in APPS$. The function is valid if and only if the session is owned by the given app. The system updates the set $SESSIONS$ by removing $s$. The mapping $app\_sessions$ is also updated by this removal.

- **addActiveRole** and **dropActiveRole**: For repairing the network security policy at runtime, adding or doping of session's active roles might be also required. The activation and deactivation of a roles during a session is done by the system functions $addActiveRole$ and $dropActiveRole$, respectively. Adding an active role is valid if and only if the role is assigned to the app, and the session is owned by that app. Drop an active role is valid if and only if the session is owned by the app and the role is an active role of that session.

- **checkAccess**: The function $checkAccess$ returns whether an app's session is or is not allowed to perform a given operation on a given object. The session has the privilege to perform the operation on that object if and only if a permission that combines that operation to the object type is assigned to (at least) one of the session's active roles set.

## 3.3 Session Handling Approaches

In a multi-session SDN app, app sessions can have an independent existence and run sequentially or simultaneously without reference to each other. An atomic session instance is the one which has a self-contained task definition and is not dependent on other session instances (i.e., a session that is not described in terms of other sessions and has no interaction with other session instances). See Fig. 3.2 (a).

In other cases, it is possible to have inter-session dependency and the execution of one session affects another one. Inter-session dependency initiates the need for inter-session interaction at runtime as will as functions and conditions for session creation/deletion, nomination of session's active role set, and adding/dropping an active role to a session. Fig. 3.2 shows several cases for multi-session apps and various methods for inter-session interaction. The relations between different sessions from different apps is beyond the scope of this discussion.

An executing session instance may, conditionally, initiate the creation of one or several session instances. In other cases, a session is created when another session completes. Given a system with a complete view of an app's entire functionality, all possible sessions, the task that should be achieved by each session, and inter-session dependencies among them, then a complete view of session-to-session relations can be represented using a directed graph. For example, if session creation happens conditionally based on another session, a directed edge between these two sessions, starting from the initiating instance, may indicate the condition and the active role set required for session creation as indicated in Fig. 3.2(c) and (e), respectively.

The management of inter-session dependency and inter-session interaction can be done either by the developer (developer-driven approach), the app system (system-driven approach), or the sessions themselves (session-driven approach). We believe that the interaction among app sessions should be well defined and managed. In this section we discuss different approaches for handling session instances of an SDN controller app.

### 3.3.1 Developer-driven Session Handling

This approach requires that the developer has full and prior knowledge of all possible sessions and roles required for each one to achieve its task. This information is provided to the controller before app execution and the system is configured accordingly. i.e., the controller knows before app execution what session instances will be created, the tasks that will execute in each session, and the set of roles required for it to execute correctly.

For each session instance, the *developer* should specify at *design time* different session handling aspects: First, the task that will be achieved by each session. Second, the condition (or precondition) under which a particular session may be created/deleted (e.g., after exceeding a bandwidth consumption cap, after new device detected, at system start-up, etc.). It should be noted here that the developer knows in advance the condition/criteria of a particular session creation as it is fixed and hard coded in the application and cannot be configured at runtime by the administrator or the controller. Example, create *Data Cap Enforcing* session if a host exceeded a bandwidth consumption cap. So, this session will start only after this condition is met. Creating this session cannot happen under any other circumstances. Third, the active role set that should be activated during session creation (e.g., $Packet\text{-}in\ Handler$ and $Flow\ Mod$ roles for a one-hour *deep packet inspection* session that will temporarily inspect traffic payload incoming from black-listed hosts). Finally, adding or dropping an active role for a session (e.g., add $DeviceTracking$ role to the *transmission rate monitoring* session).

### 3.3.2 System-driven Session Handling

In this approach, the controller has full control on session handling. The developer only provides the set of roles required by the app and then she has no discretion on determining any of other sessions' properties at runtime. Shipped with adequate capabilities, the *controller* should have the ability to specify at *runtime* what session instances will be created and how to handle them. Given an app and the set of roles required by the app, the controller should figure out each task that might execute in a separate session and the set of roles required for it to execute correctly. This approach is challenging and the hardest to implement.

For each session instance, the *controller* should specify *dynamically* at *runtime* the various properties: First, the set of sessions required to achieve the entire app's functionality. Second, the condition under which a particular session instance may be created/deleted (e.g., after attack detected, completion of another session, change of risk value, etc.). It should be noted here that, in contrast to the developer-driven approach, the developer doesn't know why a particular session could be created/terminated. The criteria for creating a session could be computed dynamically by the controller or configured by the administrator at runtime. For example, creating *intrusion prevention* session based on the outcome of statistical analysis or risk assessment. Third, the active role set that should be activated during session creation (e.g., $Routing$ and $Link\ Handler$ roles for a session that recomputes shortest path after a new link discovery), and Finally, adding or dropping an active role for a session.

### 3.3.3 Smart Sessions

For deploying this category of session handling, inter-session interaction should be conducted via a well defined set of session interaction APIs designed specifically for this purpose and managed by the system. The app developer should comply to these APIs during app design. These APIs allow sessions to get information about other sessions like names of currently active sessions, their active roles, their status (e.g., idle, up time, etc.) as will as they provide a way for passing information and notifications between sessions (e.g., results of calculations) as indicated in Fig. 3.2 (d).

A session is smart in the sense that it can take decisions based on the result of communications via inter-session interaction APIs. Thus, it can adjust its behavior to take knowledgeable decisions on future session interaction API calls and on different session handling aspects: First, the condition under which a particular session instance may be created/deleted (e.g., start *traffic redirection* session after an alarm is fired by *packet inspection* session). Second, the active role

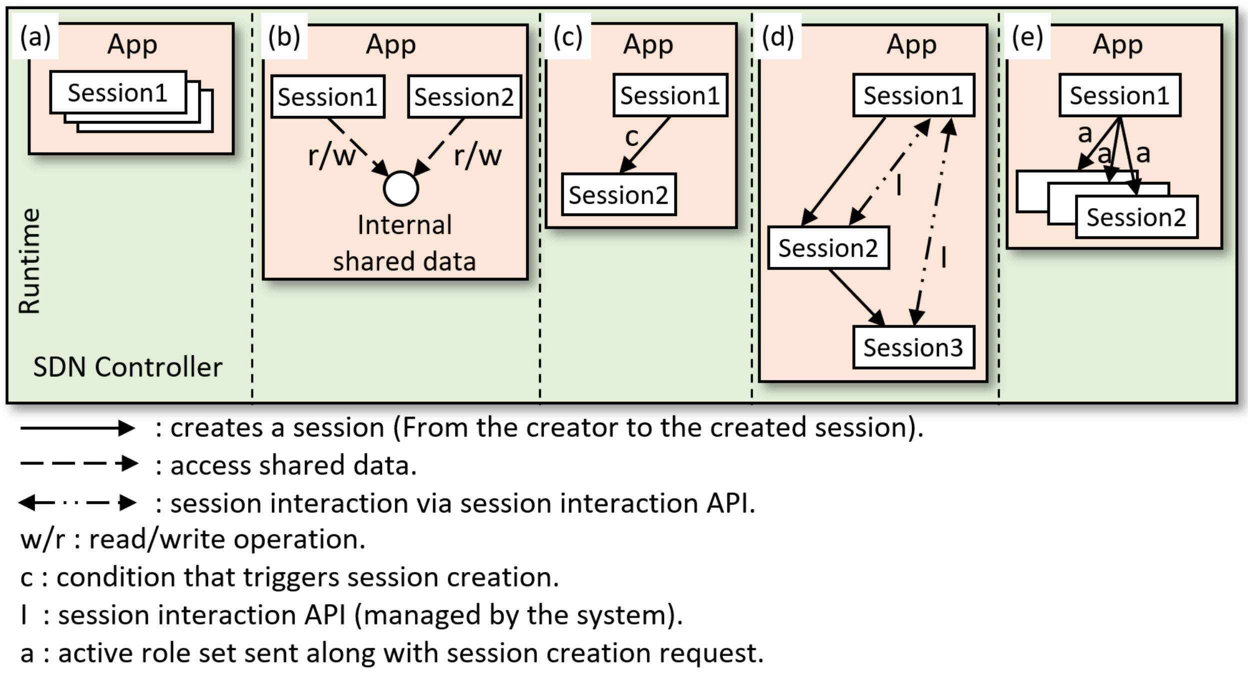

(a)
App
Session1
(b)
App
Session1
Session2
r/w
r/w
Internal shared data
(c)
App
Session1
c
Session2
(d)
App
Session1
Session2
Session3
I
I
(e)
App
Session1
a
a
a
Session2
Runtime
SDN Controller
: creates a session (From the creator to the created session).
: access shared data.
: session interaction via session interaction API.
w/r : read/write operation.
c : condition that triggers session creation.
I : session interaction API (managed by the system).
a : active role set sent along with session creation request.

**Figure 3.2**: Multi-session apps and methods for inter-session interaction. (a) App with atomic sessions. (b) Two sessions access shared data. (c) Conditional session creation. (d) Interaction via inter-session interaction APIs. (e) Active role set sent from master session to slave sessions.

set that should be activated during session creation (e.g., $Packet\text{-}in\,Handler$ role and $Flow\,Mod$ role for a *deep packet inspection* session if *web-traffic filtering* session detected malicious payloads.), Third, adding or dropping an active role for a session (e.g., add $Device\,Tracking$ role to the *transmission rate monitoring* session).

### 3.3.4 Master-Slave Sessions

In this approach, a master session initiates the creation of one or several slave sessions and provides the system with the required roles to be activated for each one, as indicated in fig. 3.2(e). Slave sessions help the master session in achieving a subtask. So this approach has two restrictions: First, the active role set of any slave session should be a subset of the master session's active role set. Second, the master session cannot terminate during the life of a slave session. During its execution, master session passes control to slave sessions and waits until completion. When completed, each slave session passes results and control back to the master session. This approach

**Table 3.3**: Roles assigned to $DataUsageCapMngr$ app and other selected roles from SDN-RBAC.

| **Role** | **General Functionality** |
|---|---|
| Device Handler | permissions for querying the controller about devices |
| Bandwidth Monitoring | permissions to read the bandwidth consumption for switch ports. |
| Flow Mod | permissions to insert/update/delete flow rules into a switch's flow tables. |
| Link Handler | permissions to get information about network links |
| Device Tracking | permissions to get notifications about changes on network devices (added, removed, Moved, Address Changed, etc.) |
| Port Handler | permissions to read information about ports and their status |
| Routing | permissions to get and compute routes between various source and destination nodes |

can be considered a special case of smart-sessions approach as it can use inter-session interaction APIs. App developer should be aware of what sessions should be master and which ones should be slave and design the app to apply this dominance via these APIs.

## 3.4 Use Case Scenario: A Multi-session App

In this section we describe a use case scenario in which an SDN app has two tasks that run in two separate sessions. Table 3.4 shows the configuration of the use case in SDN-RBAC. The app's main functionality is to limit the amount of traffic that any particular host transfers within a period of time. In order to achieve this, the app requires access to bandwidth consumption statistics of all hosts' attachment points. When a device exceeds the data usage cap, the app inserts a flow rule to rate-limit or temporarily quarantine a host who has exceeded the cap. We called the app $DataUsageCapMngr$.

To be able to get the required permissions, we associate the app with three roles described in the first three rows in Table 3.3. These three roles are composed totally of eleven permissions. So, for space limitation and convenience, we avoid showing all permissions in the use case configuration in table 3.4. We show only selected permissions enough to understand the app's use case and its model configuration aspects.

**Table 3.4**: The configuration of the $DataUsageCapMngr$ and its two sessions as a use case in SDN-RBAC[1].

- **Use case sets:**
- $APPS = \{DataUsageCapMngr\}$.
- $ROLES = \{Device\,Handler, Bandwidth\,Monitoring, Flow\,Mod\}$ .

  $D =$ set of all network devices. $FT =$ set of all flow tables in all switches, $PS =$ set of all port statistics in all switches.
- $OBS = \{D, FT, PS\}$.
- $OBTS = \{DEVICE, PORT\text{-}STATS, FLOW\text{-}TABLE\}$.
- $OT = \{(D, DEVICE), (PS, PORT\text{-}STATS), (FT, FLOW\text{-}TABLE)\}$.
- **Permissions:**
- $PRMS = \{p_1, p_2, p_3\}^1$ with $p_1 = (getAllDevices, DEVICE), p_2 = (getBandwidthConsumption, PORT\text{-}STATS), p_3 = (addFlow, FLOW\text{-}TABLE)\}$.
- **Permissions assignment:**
- $PR = \{(p_1, Device\,Handler), (p_2, Bandwidth\,Monitoring), (p_3, Flow\,Mod)\}$.
- $assigned\_perms(Device\,Handler) = \{p_1\}^1, assigned\_perms(Bandwidth\,Monitoring) = \{p_2\}^1, assigned\_perms(Flow\,Mod) = \{p_3\}^1$
- **Role assignment:**
- $AR = \{(DataUsageCapMngr, Device\,Handler)$ $(DataUsageCapMngr, Bandwidth\,Monitoring), (DataUsageCapMngr, Flow\,Mod)\}$
- **Sessions:**
- $SESSIONS = \{DataUsageAnalysisSession, DataCapEnforcingSession\}$.
- $app\_sessions(DataUsageCapMngr) = \{DataUsageAnalysisSession, DataCapEnforcingSession\}$.
- $session\_app(DataUsageAnalysisSession) = \{DataUsageCapMngr\}$, $session\_app(DataCapEnforcingSession) = \{DataUsageCapMngr\}$.
- **Active role sets:**
- $session\_roles(DataUsageAnalysisSession) = \{Device\,Handler, Bandwidth\,Monitoring\}$.
- $session\_roles(DataCapEnforcingSession) = \{Flow\,Mod\}$.

[1] Sets with this mark in the table include minimum elements enough to understand the use case.

Instead of executing the app in one monolithic process with all three roles active at once, we separate its functionality into two main tasks each to be running in a different session instance. We moved the sensitive task of inserting flow rules into a separate session.

We called one session $DataUsageAnalysisSession$ which probes for statistics on a regular basis (every 5 seconds). This session reads the bandwidth consumption for switch ports, analyzes the data and stores the results into an object 'usageCapBlackList' managed by the system. This session is created with an active role set composed of two roles as shown in $session\_roles$ function in Table 3.4. The other session is called $DataCapEnforcingSession$ which requires inserting flow rules and so its active role set is composed of the $FlowMod$ role as shown in $session\_roles$ function in Table 3.4.

## 3.5 Framework Implementation

In order to demonstrate our proof-of-concept prototype, we developed and ran the framework in Floodlight platform v1.2 release [22]. The Floodlight platform is deployed on a virtual machine that has 8GB of memory and runs on Ubuntu 14.04 OS installation. We created a topology with three virtual switches (Open vSwitch v2.3.90) connected to each other and each switch is connected to two hosts. Switches are connected to the controller and hosts are virtual machines that has 2GB and run Ubuntu 14.04 OS server.

We implemented our RBAC system in Floodlight platform and used hooking techniques without any change to the code of Floodlight modules. We implemented hooking for all operations exposed by Floodlight services to controller apps. We used AspectJ [6] which is a seamless aspect-oriented extension to Java. Our system intercepts methods before execution. When a session issues a request the hooked API invokes the RBAC policy engine for performing access verification and reply back. This system can be deployed to all other Java-based SDN controllers.

An overview of SDN-RBAC framework architecture is shown in Fig. 3.3. It contains three main components: interception component which represents the system's policy enforcement point (PEP), request evaluation component which represents the policy decision point (PDP), and SDN-

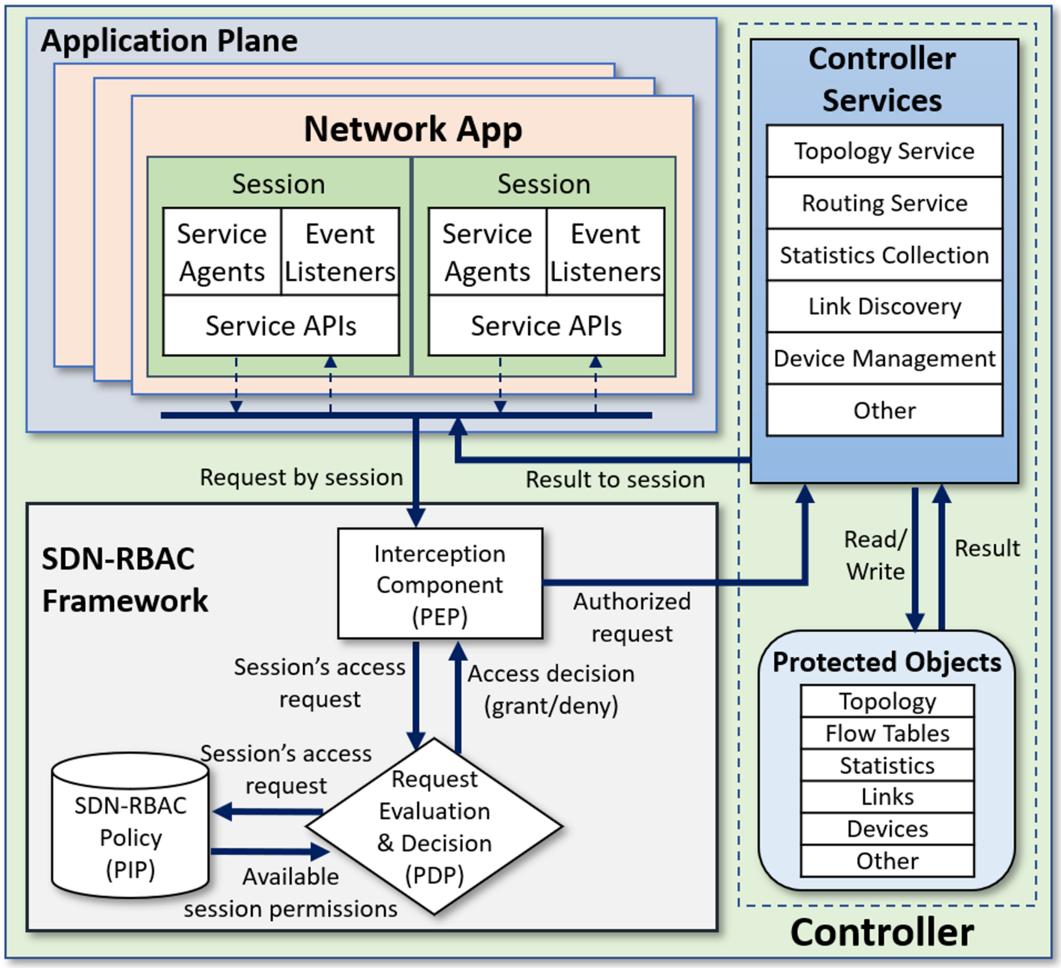
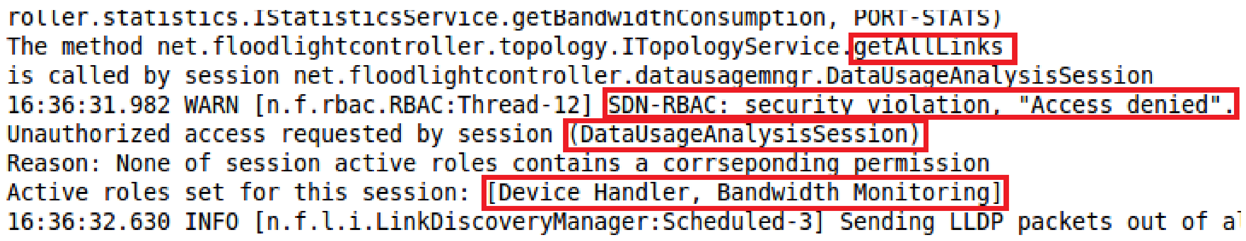


**Figure 3.3**: Overview of SDN-RBAC architecture.

RBAC policy which represents the policy information point (PIP) in the system.

We developed the $DataUsageCapMngr$ app described in Section 3.4 and configured in the SDN-RBAC model as shown in Table 3.4. The first session $DataUsageAnalysisSession$ is designed to probe for port bandwidth statistics on a regular basis (every 5 seconds). After reading the bandwidth consumption for switch ports and analyzing the data to find cap limit violations, it stores the list of hosts who has exceeded the cap limit into a list 'usageCapBlackList' managed by the system. The second session $DataCapEnforcingSession$ is designed to check periodically (every 60 seconds) for black listed hosts in order to insert flow rules to isolate them from the network. It reads the object 'usageCapBlackList' for this manner.

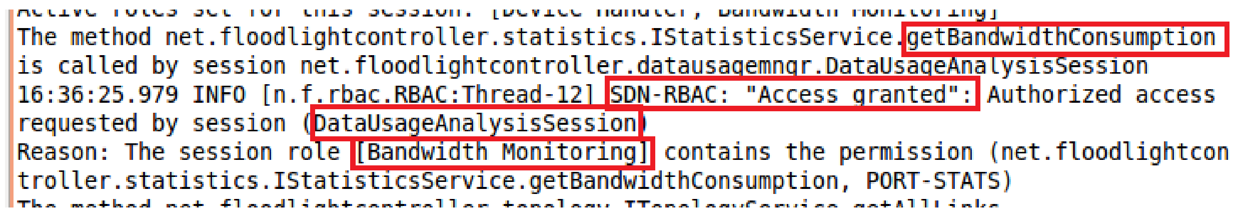
It should be noted that there is no direct interaction between these two sessions. They are



```
roller.statistics.IStatisticsService.getBandwidthConsumption, PORT-STATS)
The method net.floodlightcontroller.topology.ITopologyService.getAllLinks
is called by session net.floodlightcontroller.datausagemngr.DataUsageAnalysisSession
16:36:31.982 WARN [n.f.rbac.RBAC:Thread-12] SDN-RBAC: security violation, "Access denied".
Unauthorized access requested by session (DataUsageAnalysisSession)
Reason: None of session active roles contains a corrseponding permission
Active roles set for this session: [Device Handler, Bandwidth Monitoring]
16:36:32.630 INFO [n.f.l.i.LinkDiscoveryManager:Scheduled-3] Sending LLDP packets out of a
```

**Figure 3.4**: Snapshot of authorization check result for *getAllLinks()* operation requested by $DataUsageAnalysisSession$ - Access Denied.

running simultaneously and not directly dependent on each other. i.e., one won't crash/stop/start based on the state of the other. Also, the active role set is not provided by either one to the other and neither of them adds/drops an active role for the other. The tasks and roles associated with each session is determined at design time. This deployment is compliant with the 'Developer-driven' session handling approach described in Section 3.3.1 and uses the inter-session interaction method indicated in Fig. 3.2 (b).



```
The method net.floodlightcontroller.statistics.IStatisticsService.getBandwidthConsumption
is called by session net.floodlightcontroller.datausagemngr.DataUsageAnalysisSession
16:36:25.979 INFO [n.f.rbac.RBAC:Thread-12] SDN-RBAC: "Access granted": Authorized access
requested by session (DataUsageAnalysisSession)
Reason: The session role [Bandwidth Monitoring] contains the permission (net.floodlightcon
troller.statistics.IStatisticsService.getBandwidthConsumption, PORT-STATS)
```

**Figure 3.5**: Snapshot of authorization check result for *getBandwidthConsumption()* operation requested by $DataUsageAnalysisSession$ - Access Granted.

During the lifetime of the app, our access control system keeps mediating all sessions access requests for performing security authorizations. It can identify each session, mediate each access request and send it for authorization check based on SDN-RBAC configuration. To show that our system can identify and reject any unauthorized operations submitted at the session level, we forced $DataUsageAnalysisSession$ to practice the permission $(getAllLinks, LINK)$ which is assigned to the role $Link\ Handler$. This role is not a member of the active role set of the session $DataUsageAnalysisSession$. Thus, $(getAllLinks, LINK)$ it is not a member of the available session permissions. A snapshot of the execution result is shown in Fig. 3.4. It shows how our system can identify and reject this unauthorized access from this session. On the other hand,

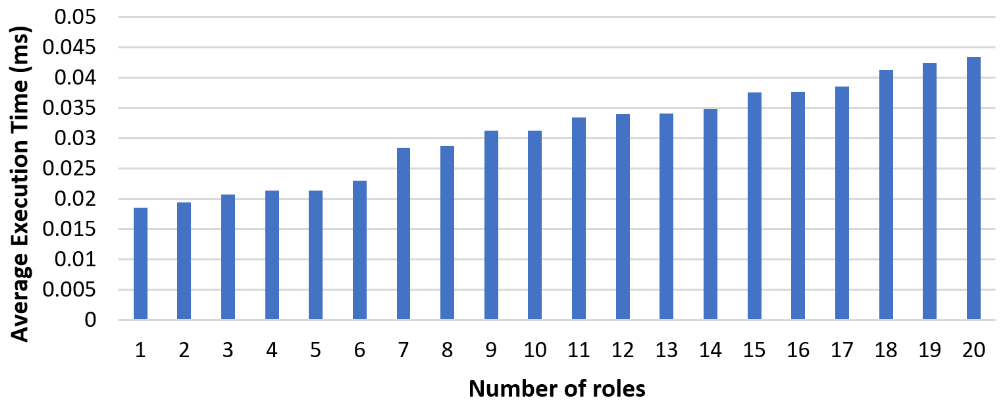


**Figure 3.6**: Average execution time required to finish the tested operations, including and excluding SDN-RBAC.

Fig. 3.5 shows how $DataUsageAnalysisSession$ was able to pass the authorization check when $getBandwidthConsumption$ operation was called.

## 3.6 Performance Evaluation

For evaluating the performance of our proposed system, We selected fifty operations covered by twenty different roles and we incrementally assigned these roles to one test app running in one session. Despite the fact that this app doesn't require all these roles, the purpose of this test is to check the overhead caused by SDN-RBAC components on the system's performance by reporting the execution time with different security policies. We change the security policy by changing the active role set of the app's session. In the first security policy one role is assigned to the session's active role set, in the second policy two roles where assigned, and so on until twenty roles.

For each security policy, the session executes all fifty operations. The system is set to find the time required by all SDN-RBAC components to finish execution and make an access control decision for each operation submitted by the session. The timer starts at the beginning of the Policy Enforcement Point (PEP) and stops when Policy Decision Point (PDP) finishes execution. The total time is calculated for all fifty operations. We repeated this test hundred times for each security policy. The average elapsed time for authorizing fifty operations is reported as shown in Fig. 3.6. It should be noted here that execution times does not include floodlight's boot-up time, the time for loading the SDN-RBAC policy and creating the corresponding relations.

This evaluation of SDN-RBAC shows that SDN-RBAC adds on average 0.031 millisecond overhead on the performance of Floodlight controller to execute fifty network operations. Authorization overhead is inevitable in SDN-RBAC as well as other authorization models. Therefore, we believe that SDN-RBAC introduces acceptable evaluation overheads.

# CHAPTER 4: FINE GRAINED ROLE BASED ACCESS CONTROL FOR SDN ENHANCED WITH PARAMETERIZED ROLES AND PERMISSIONS

In this chapter, we propose ParaSDN, an access control model to enhance access control granularity for SDN with support of role and permission parameters.

## 4.1 Motivation

Software Defined Networking (SDN) has become one of the most important network architectures for simplifying network management and enabling innovation through network programmability. Network applications submit network operations that directly and dynamically access critical network resources and manipulate the network behavior. Therefore, validating these operations submitted by SDN applications is critical for the security of SDNs.

The granularity of access provided by current access control systems for SDN applications is not sufficient to satisfy access control requirements for SDN applications. A feasible access control mechanism should allow system administrators to specify constraints that allow for applying minimum privileges on applications. In this chapter, we introduce ParaSDN, an access control model to address the above problem using the concept of parameterized roles and permissions. Our model provides the benefits of enhancing access control granularity for SDN with support of role and permission parameters. We implemented a proof of concept prototype in an SDN controller to demonstrate the applicability and feasibility of our proposed model in identifying and rejecting unauthorized access requests submitted by controller applications.

Information about network resources stored in the SDN controller are highly valuable which makes it an attractive target for attackers and increases the potential to be gained via unauthorized access. Also, the potential damage that can be done increases dramatically if this unauthorized access is done by compromised, buggy, or malicious SDN apps.

In prior works on access control for SDN apps [3], permissions are created based on object types rather than specific object instances. For example, a permission to read an object of type flow rule is generally used to read every flow rule in a switch. In another example, a permission to access a network device allows access to all network devices in the SDN controller. However, based on their network functions, different network apps require access to different network resources.

However, in real SDN environments, there is often the need to assign permissions to subset of all object instances that have the same object type. For example, in a campus network environment with multiple departments it might be required for an app to access switches in CS department only. Moreover, there is a need to assign permissions with a more accurate granularity depending on the content of the object. For example, in some network environments it is required that an app can only access and modify flow rules that handle web traffic only.

This requires an access control system that restricts apps' access scope to unique object instances. An easy solution for this problem is to have an access control system in which a separate permission is created for each single object. However, adopting such approach in role-based systems has known problems and hard to manage as it requires creating and managing huge number of permissions, roles, permission-role associations, and app-role associations.

A more flexible access control mechanism should allow the system administrator to directly specify the constraint that every app can only access and modify specific object instances commensurate to its authorization requirements. Because of the known advantages of role-based access control especially in facilitating access control management, we are proposing ParaSDN, an access control model that addresses the above problems using the concept of parameterized roles and permissions.

Role-based authorizations proposed in the literature for SDN don't provide fine grained access control that can be customized for complex use cases in SDNs [41,42,49]. However, in this chapter we present a more convenient approach for creating roles and permissions that suit complex SDN use cases.

Works in [1, 25, 26] used the concept of parameterization with roles and privileges. However, their formalization is not well structured in a complete model, which make it hard to adopt in different contexts. In this work, we introduce a formal definition for parameterized roles and permissions that conform to the standard RBAC model and more flexible to adopt in a variety of use cases.

## 4.2 ParaSDN Components Overview

In this sesction, we present an overview of the model components and give examples of the syntax and semantic in the context of SDN environment.

### 4.2.1 Parameters

A *parameter* is a name:value pair that, when assigned to a permission, indicates a subset of network resources that an app can exercise using this permission. A parameter value: (1) may be a list of network resources identified directly via resource IDs. For example, the parameter attachment_point can have the value {0x1:1, 0x1:2, 0x2:1} to indicate the listed switch:port combinations; (2) might be a label that indicates a group of network resources. For example, the parameter dept with a value of CS can indicate all switch ids in the CS department; or (3) point to a property existing in the requested object. For example, the parameter traffic with the value of web indicates the set of TCP ports used for Web protocol.

An aggregate parameter can be defined as a label that indicates set of potential individual parameter values. For example, the parameter dept with a value of CS can indicate all switch ids in the CS department. Also, the parameter web can indicate all ports numbers used for web traffic. Parameter value aggregation (1) simplifies the job of parameter management (as they only have to directly deal with a limited number of parameter aggregates, rather than a potentially high number of individual parameter values) and (2) enhances managements scalability (as the growth of number of parameter values (e.g., recourse IDs) does not necessarily imply additional load in parameter management).

Each parameterized role is associated with a finite set of parameters. The range of each parameter is represented by a finite set of atomic values. For example, the range of dept parameter

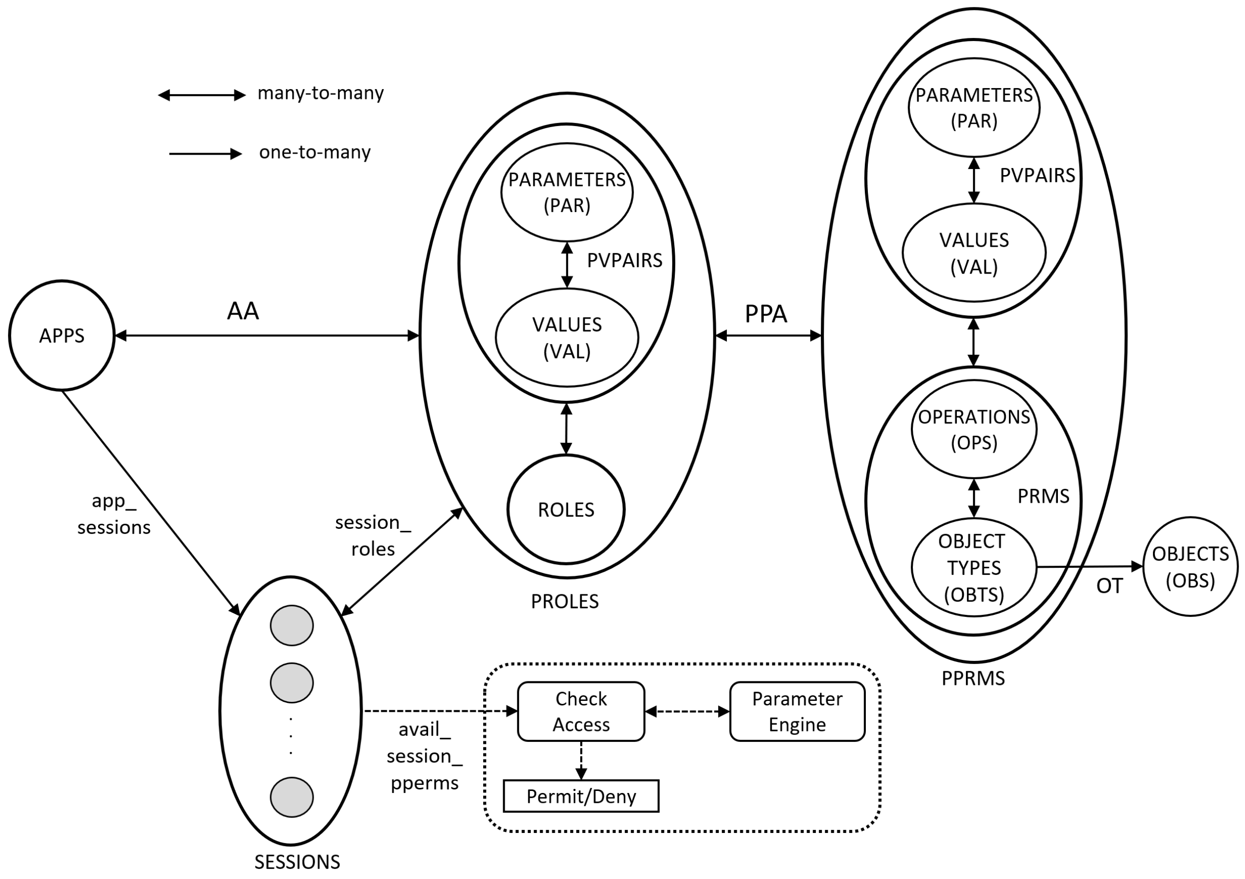


**Figure 4.1**: ParaSDN Conceptual Model.

is a set of department names that share the network infrastructure. Each parameter can either be atomic or set-valued from its declared range. For a particular parameter *p*, it is range is composed only from those values authorized by the system administrator. Different categories of parameters for SDN with examples of each category are discussed in section 4.6.

### 4.2.2 Parameterized Permissions

A *parameterized permission* is represented by the ordered pair:

((op$_i$, ot$_i$), {(par$_1$, val$_1$), (par$_2$, val$_2$), ...})

where (op$_i$, ot$_i$) combines a network operation with an object type in the ordinary permission format, and {(par$_1$, val$_1$), (par$_2$, val$_2$), ...} is a subset of parameter:value pairs. In the parameterized permission, the object type ot$_i$ indicates all object instances of that type on which operation op$_i$ can be exercised. If it is used alone, it provides a very course-grained access privilege and impractical for many SDN security policies. In many situations, what is required is to provide access to subset of object instances of that type. This is achieved with the help of the parameters associated with this permission. The semantics of this parameterized permission is that an app can execute the operation $\text{op}_i$ on only object instances of type $\text{ot}_i$ that satisfy the restrictions imposed by the parameter values.

Permission parameters are not assigned values within the permission itself; instead, their values are defined when it is associated with a parameterized role whose parameter values already defined, i.e., permission parameters are steered by role parameters.

So, when security architects create a parameterized permission, they initialize parameter values with a special value $\perp$ which means unknown. For example, the parameterized permission:

((addFlow, FLOW-RULE), {(dept, $\perp$), (traffic, $\perp$)})

indicates that an app can insert flow rules in switches of as-yet-unknown department(s) and these rules can handle traffic of as-yet-unknown type. If the values of parameters dept and traffic are CS and web then an app can add flow rules that handle Web traffic in switches of CS department.

### 4.2.3 Parameterized Roles

A *parameterized role* is represented using an ordered pair:

($\text{r}_i$, {($\text{par}_1$, $\text{val}_1$), ($\text{par}_2$, $\text{val}_2$), ...})

where $\text{r}_i$ represents a role name, and {($\text{par}_1$, $\text{val}_1$), ($\text{par}_2$, $\text{val}_2$), ...} is a set of parameter:value pairs. Initially, all role parameters are assigned a special value $\perp$ which means unknown. For example,

(Flow Mod, {(dept, $\perp$), (traffic, $\perp$)})

is a parameterized role that includes permissions to read, update, insert, and delete flow rules in switches of as-yet-unknown department(s) and these rules can handle traffic of as-yet-unknown type. If the values of parameters dept and traffic are CS and web then an app can exercise these operations only to flow rule instances that reside in switches of CS department and handle traffic destined to Web servers.

### 4.2.4 Parameter Value Assignment

At the time of role engineering, there is no need to worry about actual parameter values at the level of permissions and roles. As mentioned above, parameterized permissions and parameterized roles are instantiated with parameter values assigned a special value $\perp$ which means unknown.

A parameterized permission is assigned to a parameterized role via the administrative action $assignPPerm(pp, pr)$, where $pp$ is a parameterized permission and pr is $a$ parameterized role. At this time, no actual parameter values are assigned. This is demonstrated in step 1 of Fig. 4.2 (a). Because parameter values are assigned based on the requirements for an app to access system resources, their values will remain unknown until app-to-role assignment via $assignApp(a, pr, valset)$ administrative action, where $a$ is an app, $pr$ is a parameterized permission, and $valset$ is the set of values to be supplied to $pr$. The values in $valset$ propagates automatically to corresponding permission parameters. This app-to-role assignment and value propagation is demonstrated in steps 2 and 3 of Fig. 4.2 (a). The final state of the parameterized role and parameterized permission as associated with app $a$ is shown in Fig. 4.2 (b).

### 4.2.5 Parameter Verification

We consider an app's access request to an object as a right of access claim by that app to that object. This claim requires verification by the access control system. We use specific functions, called Verifiers, to check the validity of this claim by comparing the parameter values in the actual access rights of the app (i.e., the available parameterized permissions of the session) with the properties of requested object. Based on the examples discussed above, for example, a verifier VRuleSwitch will be called after exercising the permission ((addFlow, FLOW-RULE), (dept, CS), (traffic, web)). It is used to verify that a flow rule that is being submitted by an app for insertion is to be inserted in an authorized switch. That is in switches if the CS department. The verifier exploits information from the object, i.e., the flow rule, and parameter values from the parameterized permission, i.e., CS department. If the accessed switch is within the switches of CS department, a positive response is returned, otherwise the verifier returns negative response.

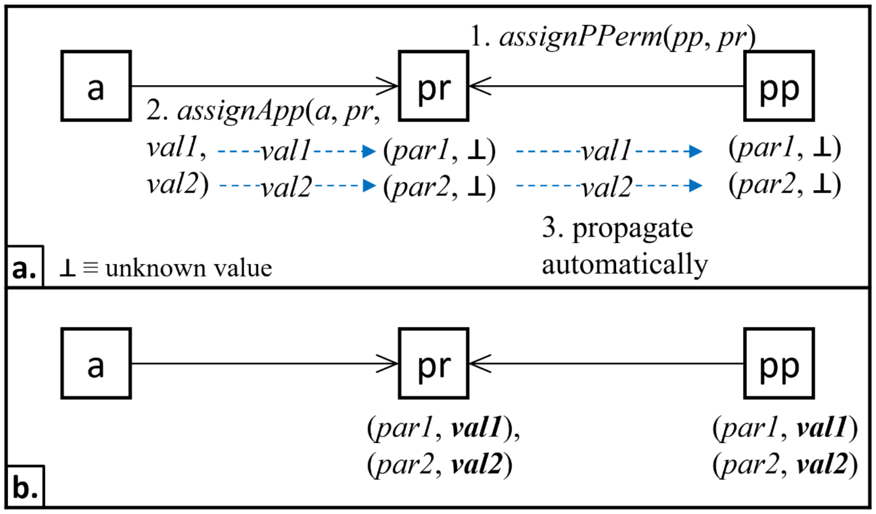

1. assignPPerm(pp, pr)
a
pr
pp
2. assignApp(a, pr, val1, val2)
val1
val2
(par1, ⊥)
(par2, ⊥)
3. propagate automatically
a.
⊥ ≡ unknown value
(par1, val1), (par2, val2)
(par1, val1) (par2, val2)
b.

**Figure 4.2**: Parameter values assigned via assignApp administrative action propagate automatically from role parameters to permission parameters.

It worth mentioning that one verifier can serve multiple parameterized permissions. For example, the same verifier VRuleSwitch will be called with the permission ((deleteFlow, FLOW-RULE), (dept, CS), (traffic, web)). Associating one verifier with multiple parameterized permissions reduces the management effort when dealing with large number of permissions. Also, one parameterized permission might require multiple verifiers. For example, another verifier that will be invoked for any of the above parameterized permissions is VRuleTraffic which verifies that the accessed flow rule handles Web traffic. Number of verifiers required to be called for one parameterized permission depends on the number of parameters associated with the permission.

## 4.3 ParaSDN Conceptual Model and Definition

The conceptual model and the relations between the components of ParaSDN are shown in Fig. 4.1. ParaSDN has the following basic components: OpenFlow apps APPS, roles ROLES, operations OPS, objects OBS, object types OBTS, the parameter set PAR, and the set of parameter values VAL.

The basic sets and functions in ParaSDN are shown in Table 4.1. APPS refer to the set of OpenFlow apps. ROLES is the set of role names. OPS is the set of all operations exposed by the controller services to apps and performed on objects. For example, the Device Service exposes

**Table 4.1**: ParaSDN Formal Model Definition.

**1.Basic Sets:**

- APPS, ROLES, OPS, OBS, OBTS, PAR, and VAL: set of apps, roles, operations, objects, object types, parameters, and parameter values.
- For each *par* $\in$ PAR, Range(*par*) represents the parameter's range, a finite set of atomic values. We assume VAL includes a special value "$\perp$" to indicate that the value of a parameter is unknown.
- parType: PAR $\rightarrow$ {set, atomic} specifies parameter type as set of atomic valued.
- PRMS $\subseteq$ OPS $\times$ OBTS, set of ordinary permissions.
- SESSIONS, set of sessions.

**2.Assignment Relations:**

- OT $\subseteq$ OBS $\times$ OBTS, a many-to-one relation mapping an object to its type, where $(o, ot_1) \in$ OT $\wedge$ $(o, ot_2) \in$ OT $\Rightarrow ot_1 = ot_2$.
- PVPAIRS $\subseteq$ PAR $\times$ VAL, a many-to-many mapping parameter to value assignment relation. For convenience, for every pvpair = (par$_i$, val$_i$) $\in$ PVPAIRS, let pvpair.par = par$_i$ and pvpair.val = val$_i$.
- PPRMS $\subseteq$ PRMS $\times$ $2^{PVPAIRS}$, a relation mapping a permission role to subset of (parameters , value) combinations. For convenience, for every pp = ((op$_i$, ot$_i$), PVPAIRS$_i$) $\in$ PPRMS, let pp.op = op$_i$, pp.ot = ot$_i$, and pp.PVPAIRS = PVPAIRS$_i$.
- PROLES $\subseteq$ ROLES $\times$ $2^{PVPAIRS}$, a relation mapping a role to subset of combinations of parameters and their values. For convenience, for every pr = ( r$_i$, PVPAIRS$_i$) $\in$ PROLES, let pr.r = r$_i$ and pr.PVPAIRS = PVPAIRS$_i$.
- PPA $\subseteq$ PPRMS $\times$ PROLES , a many-to-many mapping parameterized permission to parameterized role assignment relation.
- AA $\subseteq$ APPS $\times$ PROLES, a many-to-many mapping app to parameterized role assignment relation.

**3.Derived Functions:**

- assigned_pperms: PROLES $\rightarrow$ $2^{PPRMS}$, the mapping of parameterized role into a set of parameterized permissions. Formally, assigned_pperms(pr) = {pp $\in$ PPRMS | (pp, pr) $\in$ PPA}.
- app_sessions: APPS $\rightarrow$ $2^{SESSIONS}$, the mapping of an app into a set of sessions.
- session_app : SESSIONS $\rightarrow$ $2^{APPS}$, the mapping of session into the corresponding app.
- session_roles: SESSIONS $\rightarrow$ $2^{PROLES}$, the mapping of session into a set of parameterized roles. Formally, session_roles(s) = {pr $\in$ PROLES | (session_app(s), pr) $\in$ AA}.
- type: OBS $\rightarrow$ OBTS, a function specifying the type of an object defined as type(o) = {$t \in$ OBTS | $(o, t) \in$ OT}.
- avail_session_pperms: SESSIONS $\rightarrow$ $2^{PPRMS}$, the parameterized permissions available to an app in a session. Formally, avail_session_pperms(s) = $\bigcup_{pr \in session_roles(s)}$assigned_pperms(pr).

**4.Parameter Verification Functions:**

- VERIFIERS = {V$_1$, V$_2$, ..., V$_n$} a finite set of Boolean functions.
  For each V$_i$ $\in$ VERIFIERS.V$_i$ : SESSIONS $\times$ OPS $\times$OBS $\times$ PVPAIRS $\rightarrow$ {True, False}.
- param_verifier: OBTS $\times$ PAR $\rightarrow$ VERIFIERS, a function that maps a combination of object type and parameter to the corresponding verification function needs to be evaluated.

**Table 4.2**: Parameter Checking Functions.

**A. Verifiers:**
Language LVerify is used to define each verifier $V_i$(*s: SESSIONS, op*: OPS, *ob*: OBS, pvpair : PVPAIRS) in VERIFIERS.

**B. CandidateVerifiers**: a function that maps each object type to its applicable set of verifiers.

```
CandidateVerifiers(ot: OBTS, pvpairs : $2^{PVPAIRS}$){
    verifiers = {};
    For each pvpair$_i$ $\in$ pvpairs do
        V$_i$ = param_verifier(ot, pvpair$_i$.par);
        verifiers := verifiers $\cup$ {(V$_i$ $\times$ pvpair$_i$)};
    return verifiers;
}
```

**C. ParamCheck**: a function that checks an object against all candidate verifiers until the first failure is discovered or a true is returned as the final outcome.

```
ParamCheck(s: SESSIONS, op: OPS, ob: OBS, pvpairs: $2^{PVPAIRS}$){
    For each (V$_i$ $\times$ pvpair$_i$) $\in$ CandidateVerifiers(type(ob), pvpairs) do
        if $\neg$V$_i$(s, op, ob, pvpair$_i$)
            return false;
    return true;
}
```

operations to query the list of devices/hosts based on one of Mac Address, VLAN id, IPv4 Address, IPv6 Address, or a combination of them. Inserting flow rules and reading port statistics are another examples of operations. OBS is the set of object instances that are managed by the controller and should be protected from unauthorized access. They are managed by the controller to maintain a consistent state of the network infrastructure. An element in OBTS represents the type of a specific object instance. For example, FLOW-RULE, DEVICE, and LINK refer to the type of actual instances of flow rules, devices, and links respectively.

PAR represents the set of all parameters in the system. This could be atomic or set valued as determined by the type of the parameter. Type of a parameter, set or atomic, is specified by the function parType. VAL is a set of parameters values used in the system. PRMS is the set of permissions, where a permission combines a network operations with an object type. The set SESSIONS represents a mapping between an app and an activated subset of parameterized roles. An app can have multiple sessions and a session belongs to only one app. OT is relation for the combinations between objects and their types. PVPAIRS is a subset of parameter:value pairs.

**Table 4.3**: Language LVerify to form verifiers.

| |
|---|
| $\varphi$ ::= $\varphi \land \varphi \mid \varphi \lor \varphi \mid (\varphi) \mid \neg\varphi \mid \exists x \in$ set.$\varphi \mid \forall x \in$ set.$\varphi$ \| set setcompare set \| atomic $\in$ set \| atomic atomiccompare atomic setcompare ::= $\subset \mid \subseteq \mid \not\subseteq$atomiccompare ::= $< \mid = \mid \leq$ |
| set ::= setpar.val \| ConstSet |
| atomic ::= atomicpar.val \| ConstAtomic |
| setpar $\in$ {pvpair \| pvpair $\in$ PVPAIRS $\land$ parType(pvpair.par) = set} |
| atomicpar $\in$ {pvpair \| pvpair $\in$ PVPAIRS $\land$ parType(pvpair.par) = atomic} |

PPRMS defines the set of parameterized permissions as discussed in Section 4.2.2. PROLES defines the set of parameterized roles as discussed in Section 4.2.3.

**Table 4.4**: App authorization function.

| **Function** | **Authorization Condition** |
|---|---|
| checkAccess(s: SESSIONS, op: OPS, ob: OBS) | $\exists$pr $\in$ PROLES : pr $\in$ session_roles(s), $\exists$pp $\in$ PPRMS : (pp, pr) $\in$ PPA $\land$ (op, type(ob)) = (pp.op, pp.ot) $\land$ **ParamCheck**(s, op, ob, pp.PVPAIRS) = True. |

Functions required for parameter verification are defined in part 4 of Table 4.2. VERIFIERS is a set of boolean functions defined by security administrators for parameter verification. Each $V_i \in$ VERIFIERS is applied on an object and a parameter to check whether an object satisfies the requirements of the parameter. Param_verifier is a function that returns a verifier tat need to be executed at the time of access request. It maps an (object type, parameter) pairs to their related verifier.

In our model, parameter checking and verification process is an essential part of evaluating each session's access request. It requires different components to communicate as illustrated in Table 4.2. Security administrators firstly need to define a parameter verification function $V_i$ (or so-called a verifier) that must be executed to find whether an object fulfills the requirements of a parameter. Verifiers are defined by means of the language LVerify defined in Table 4.3. The language LVerify allows to create conditions that involves parameter values and information about the object. In this language, ConsSet and ConsAtomic are constant sets and atomic values.

Because not all the verifiers need to be executed for a requested object, security administrators need to specify the subset of verifiers applicable to the requested object and the permission parameters. An access request might need to execute multiple verifiers depending on the number parameters associated with the parameterized permission under the check. At the time of access request, the function CandidateVerifiers receives all parameters associated with a parameterized permission need to be checked. This function is responsible of retrieving the set of applicable verifiers and submitting this set to the function ParamCheck for evaluation. In order to do this, it passes the object type along with each parameter to param_verifier function that retrieves the applicable verifier.

For an object and set of permission parameters, it should be mentioned that the function CandidateVerifiers doesn't deal with the parameter values or the object instances themselves, however, it relies on the parameter name and the object types to fetch a relevant verifier. On the other hand, verifiers use information about actual object and actual parameter values for evaluation.

The function ParamCheck receives the applicable verifiers for the object and verifies if the object can be accessed based on the provided parameter values. It achieves this by invoking the verifiers one by one. For finding the final outcome of session's access request, the system function CheckAccess is used. This function is formally defined in Table 4.4. As part of final decision, it invokes the function ParamCheck to evaluate the compliance of the object with the permission parameters. It is responsible of returning the final decision whether an app's session is or is not allowed to perform a given operation on a given object.

## 4.4 App and Permission Assignment

The specification of a complete list of administrative functions is out of the scope of this work. We only show two administrative functions $assignApp(a, pr, valset)$ and $assignPPerm(pp, pr)$ due to their relation to the parameter values as described in Section 4.2.4. The formal specification of these two administrative functions is shown in Table 4.5. The function $assignApp(a, pr, valset)$ assigns an app $a$ to a parameterized role $pr$ and assigns the values in $valset$ to parameters in $pr$. The values in $valset$ propagates automatically to the corresponding permission parameters in every

**Table 4.5**: Formal specification of assignApp(a, pr, valset) and assignPPerm(pp, pr) administrative functions.

| **Function** | **Authorization Condition** | **Update** |
|---|---|---|
| assignPPerm(pp, pr) | pp $\in$ PPRMS $\wedge$ pr $\in$ PROLES $\wedge$ (pp, pr) $\notin$ PPA | PPA' = PPA $\bigcup$ {(pp, pr)} |
| assignApp(a, pr, valset) | a $\in$ APPS $\wedge$ pr $\in$ PROLES $\wedge$ valset $\in$ VAL $\wedge$ (a, pr) $\notin$ AA | //Assign values to role parameters.<br>**For each** pr_pvpair$_i$ $\in$ pr.PVPAIRS, v$_i$ $\in$ valset, $1 \leq i \leq$ $\|$pr.PVPAIRS$\|$ **do**<br>    pr_pvpair$_i$.val = v$_i$<br>//Pass parameter values from pr to its member parameterized permissions.<br>**For each** pp $\in$ PPRMS : (pp, pr) $\in$ PPA **do**<br>    **For each** pr_pvpair$_i$ $\in$ pr.PVPAIRS, pp_pvpair$_i$ $\in$ pp.PVPAIRS, $1 \leq i \leq$ $\|$pr.PVPAIRS$\|$ **do**<br>        pp_pvpair$_i$.val = pr_pvpair$_i$.val<br>AA' = AA $\bigcup$ {(a, pr)} |

parameterized permission $pp$ associated with $pr$. A parameterized permission $pp$ can be assigned to a role parameterized $pr$ via the $assignPPerm(pp, pr)$ function.

## 4.5 Framework Architecture and Parameter Engine Components

In this section, we show how ParaSDN is designed to integrate role parameters in the decision process, and also we introduce how ParaSDN works by presenting its operational scenario. As shown in Fig. 4.3, ParaSDN consists of four main components: (1) Policy Enforcement Point (PEP), (2) Policy Decision Point (PDP), (3) Policy Information Point (PIP), and (4) Parameter Engine. The General functionality of the Parameter Engine itself is distributed among multiple components, namely, Parameter Check Point (PCP), Verifiers Retrieval Point (VRP), and multiple Parameter Verification Points (PVPs). These components function together to provide parameter evaluation essential for generating an access control decision fundamental for security policy enforcement.

When an app's session submits an access request, the authorization flow involves intercepting the session's access request by PEP, passing the request to the PDP, and inquiring the PIP for

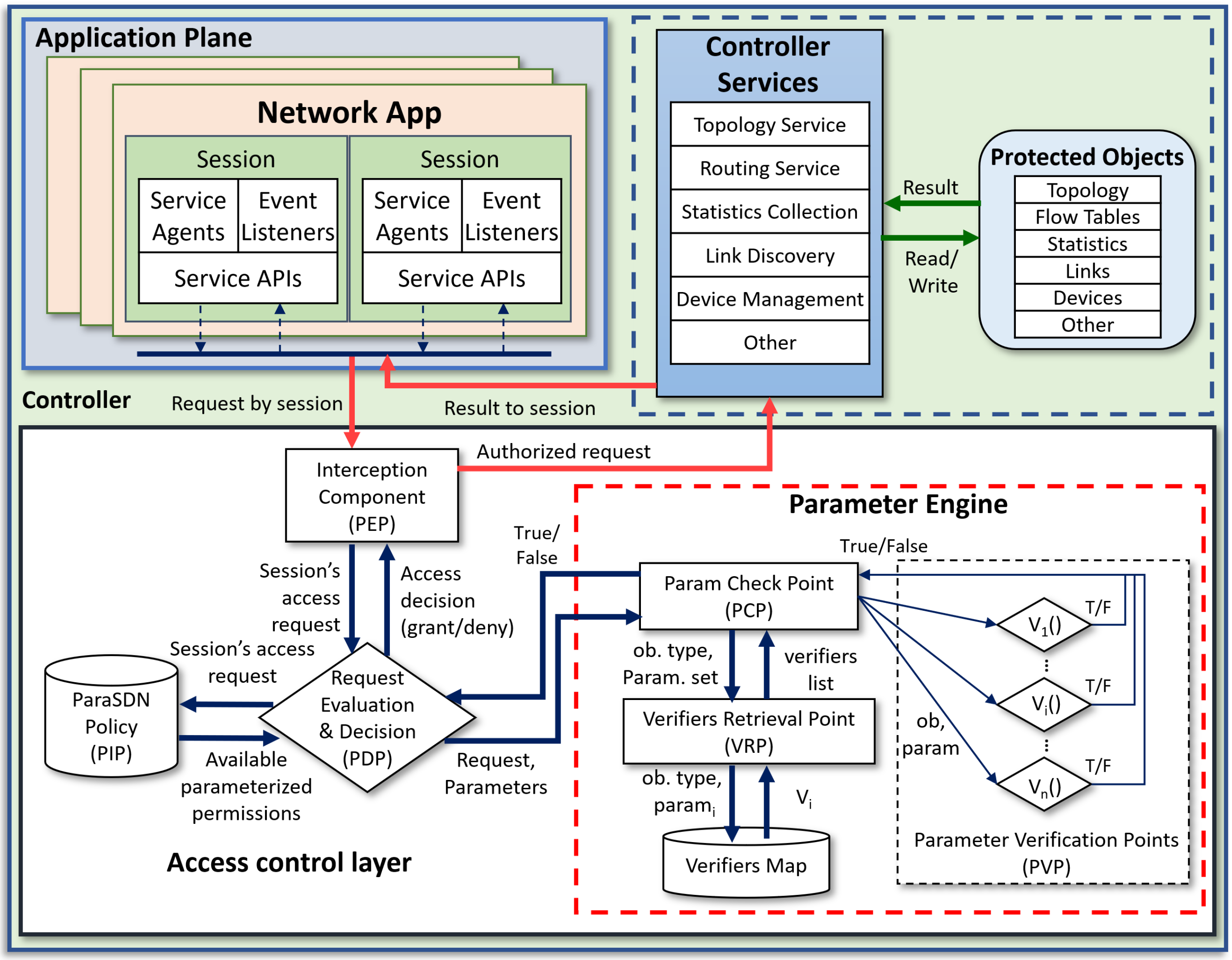


**Figure 4.3**: General Overview of the ParSDN system components and Architecture

available session parameterized permissions. Before involving parameters in the authorization process, the regular permissions in the parametrized permissions available for a session provides the right to operate on all objects of specific type. However they not enough for identifying certain set of objects accessible by a session. The authorization flow is proceeded by sending the object and the parameters to the Parameter Engine for verification.

The first component of the Parameter Engine is the PCP. It represents a central point in the Policy engine. It is responsible of receiving the object and the permission parameters and verifying if the object can be accessed based on the provided parameters' values. In order to do this, the PCP must check if the requested object complies with the requirements of each and every parameter associated with the permission. This is done by invoking candidate verifiers each represents a

**Table 4.6**: Examples for Flow-driven Parameters for SDN.

| **Parameter** | **Description** |
|---|---|
| tcp_src, tcp_dst | TCP source/distination port |
| udp_src, udp_dst | UDP source/distination port |
| vlan_id | VLAN id |
| ip_proto | IP protocol |
| ipv4_src, ipv4_dst | IPv4 source/distination address |
| ipv4_src_mask, ipv4_dst_mask | IPv4 source/distination subnet mask |

parameter verification point (PVPs).

Each PVP is a boolean expression designed by the security administrator to verify if an object satisfies the requirement of the parameter. In other words, each PVP receives an object and a parameter and evaluates the session's right to access the object based on the parameter value. If any PVP returns FALSE, which means that the requirements of that parameter is not satisfied, the PCP stops the whole parameter verification process and returns false to the PDP. On the other hand, the PCP will return true if and only if the object satisfies all the perimeter requirements, i.e., all PVPs return TRUE. The PDP logic relies on this result to allow or deny access to the requested object.

Before the PCP calls any PVP, it need to specify the subset of PVPs need to be invoked. We design the VRP as responsible of identifying these PVPs and submitting them to the PCP. The VRP does this by referring to the VerifiersMap which maps pairs of object type and parameter to their applicable PVP.

## 4.6 Parameter Categories for SDN

We identify four categories of parameters that can be used with parameterized roles and permissions for SDN environment.

**1. Topology-specific parameters:** parameters to identify subsets of network switches, links, or ports. For example, the set-valued parameter switch_id with a value of 00:00:00:00:00:00:00:01 assigned to a role Topology-Visualizer restricts role holders from accessing other switches.

**2. Flow-driven parameters:** represent parameters to identify flow rules. They can be supplied to roles (e.g., ‘Flow Mod’) that authorize access to objects of type FLOW-RULE. For example, parameter tcp_dst assigned a value of 80 will identify all flow rules that manipulate traffic destined to an HTTP server. A parameter ipv4_dst_mask assigned a value of 192.168.5.0/24 identifies flow rules targeting this subnet. i.e., targeting IP addresses in the range 192.168.5.0 - 192.168.5.255 that has subnet mask of 255.255.255.0. Table 4.6 shows examples of Flow-driven parameters.

**3. Application-specific parameter:** This parameter represents an app_id. It is supplied to roles to identify particular app that will operate using this role. For example, assume the parameter app_id is supplied to role ‘Pool Manager’ and app_id is assigned the value "Load Balancer" (assuming "Load Balancer" is an app ID for a load balancer app), this means that this role can operate only by "Load Balancer" app. Every time a request is submitted by a session using this role, a verifier function VApp_id should verify that session_app(s) = app_id(‘Pool Manager’), i.e., session_app(s) = "Load Balancer". Assuming this session is compromised by an app MalApp, this makes session_app(s) = MalApp. As a result, any request using this session will not be granted because the verifier VApp_id will fail since the check session_app(s) = app_id(‘Pool-Manager’) will return false because the parameter value ‘Pool Manager’ is attached as the parameter value in the parameterized role. This requires sending the session id as parameter to the verifier function in order to use session_app(s) in the evaluation process which is already depicted in the formal model in Table 4.1.

**4. Organization-specific parameters:** They represent parameters pertaining to internal organizational structure such as divisions and departments operating internally at some level in the organization hierarchy. For example, a parameter dept with the value of CS or CE associated with a ‘Flow Mod’ role identifies network resources that can be accessed by apps operating under Computer Science or Computer Engineering departments, respectively. These resources might include set of switches, ports and links under the authority of specific department. The interpretation of the organization-specific parameters and the resources associated with them is an internal organization issue. In another example, a parameter tenant with the value tenant1, authorizes an app to access

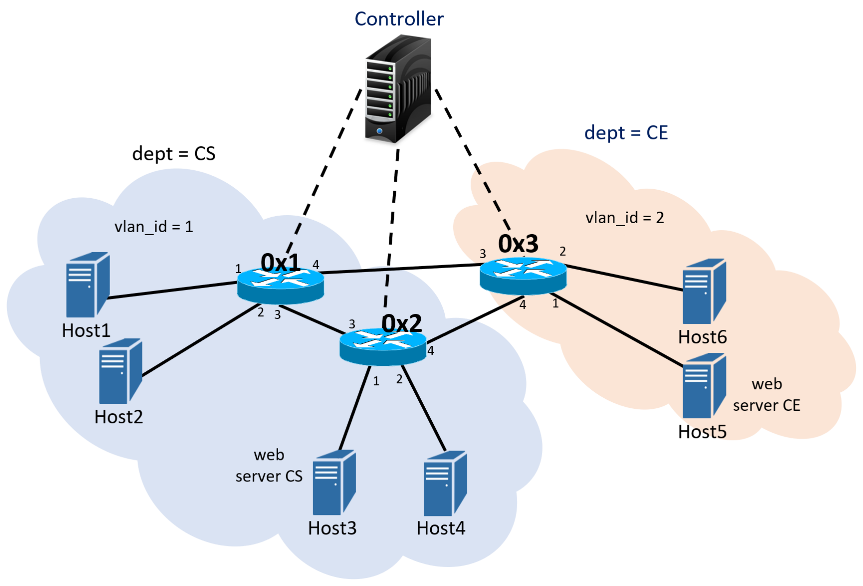


**Figure 4.4**: Topology for proof of concept use case in section 4.7.

tenant1 resources.

## 4.7 Proof of Concept Use Case

In this section we demonstrate and configure a use case in ParaSDN. Assume in a small campus network we have the network infrastructure as depicted in Fig. 4.4. The infrastructure is divided between two departments CS and CE. Assume CS dept independently manages two switches, 0x1 and 0x2 and the four hosts connected to them. Host-3 runs a web server. The CE department separately manages one switch 0x3 and two hosts host-5 and host-6. Host-5 runs a web server. hosts1-4 are assigned vlan_id=1, and hosts-5 and Host-6 are assigned to vlan_id=2. Switches are connected to one controller. The controller has two apps, one for each Department. 'Data Usage Cap Mngr' is authorized on resources of CS dept and 'Intrusion Prevention App' is authorized on resources of CE dept. The basic sets and assignment relations of the use case configuration is shown in Table 4.7.

The app 'Data Usage Cap Mngr' is designed to protect web server on host-3 from any denial-of-service. It needs to monitor bandwidth consumption on attachment points in switches of CS dept. Thus, is assigned to the parameterized role (Bandwidth Monitoring, (attachment_point, 0x1:1,

**Table 4.7**: Configuration of the proof of concept use case of section 4.7 in ParaSDN (Part 1).

**1. Model Basic Sets:**

- APPS = {Data Usage Cap Mngr, Intrusion Prevention App}.
- ROLES = {Device Handler, Bandwidth Monitoring, Flow Mod, Packet-In Handler}.
- OPS = {queryDevice, getBandwidthConsumption, addFlow, readPacketInPayload}.
- OBS = D ∪ PS ∪ FR ∪ PIP, where D = set of all network devices, PS = set of all port statistics in all switches, FR = set of all flow rules, and PIP = set of all packet-in messages.
- OBTS = {DEVICE, PORT-STATS, FLOW-RULE, PI-PAYLOAD}.
- PAR = {vlan_id, attachment_point, dept, traffic}.
- Range(vlan_id) = {1, 2}. Range(attachment_point) = {0x1:1, 0x1:2, 0x2:1, 0x2:2, 0x3:1}. Range(dept) = {CS, CE}. Range(traffic) = {web}.
- parType(vlan_id) = atomic. parType(attachment_point) = set. parType(dept) = set. parType(traffic) = atomic.
- PRMS = {(queryDevice, DEVICE), (getBandwidthConsumption, PORT-STATS), (addFlow, FLOW-RULE), (readPacketInPayload, PI-PAYLOAD)}.
- SESSIONS = {DataUsageAnalysisSession, DataCapEnforcingSession, IntrusionPreventionSession}.

2. **Assignment Relations:**

- OT = {(d, DEVICE) : d ∈ D} $\bigcup$ {(ps, PORT-STATS) : ps ∈ PS} $\bigcup$ {(fr, FLOW-RULE) : fr ∈ FR} $\bigcup$ {(pip, PI-PAYLOAD) : pip ∈ PIP}}.
- PPRMS = {((queryDevice, DEVICE), {(vlan_id, ⊥)}), ((getBandwidthConsumption, PORT-STATS), {(attachment_point, ⊥)}),
  ((addFlow, FLOW-RULE), {(dept, ⊥), (traffic, ⊥)}), ((readPacketInPayload, PI-PAYLOAD), {(attachment_point, ⊥)})}
- PROLES = {(Device Handler, {(vlan_id, ⊥)}), (Bandwidth Monitoring, {(attachment_point, ⊥)}),
  (Flow Mod, {(dept, ⊥), (traffic, ⊥)}), (Packet-In Handler, {(attachment_point, ⊥)})}
- PPA = {(((queryDevice, DEVICE), {(vlan_id, ⊥)}), (Device Handler, {(vlan_id, ⊥)})), (((getBandwidthConsumption, PORT-STATS), {(attachment_point, ⊥)}), (Bandwidth Monitoring , {(attachment_point, ⊥)})),
  (((addFlow, FLOW-RULE), {(dept, ⊥), (traffic, ⊥)}), (Flow Mod, {(dept, ⊥), (traffic, ⊥)})),
  (((readPacketInPayload, PI-PAYLOAD), {(attachment_point, ⊥)}), (Packet-In Handler, {(attachment_point, ⊥)}))}.
- AA = {(Data Usage Cap Mngr, (Device Handler, {(vlan_id, 1)})), (Data Usage Cap Mngr, (Bandwidth Monitoring, {(attachment_point, {0x1:1, 0x1:2, 0x2:1, 0x2:2})})), (Data Usage Cap Mngr, (Flow Mod, {(dept, {CS}), (traffic, web)})), (Intrusion Prevention App, (Device Handler, {(vlan_id, 2)}), (Intrusion Prevention App, (Packet-In Handler, {(attachment_point, {0x3:1})}), (Intrusion Prevention App, (Flow Mod, {(dept, {CE}), (traffic, web)}))}.

**Table 4.8**: Configuration of the proof of concept use case of section 4.7 in ParaSDN (Part 2).

| | |
|---|---|
| **3. Derived Functions:** | |
| – | assigned_pperms((Device Handler, {(vlan_id, $\perp$)})) = {((queryDevice, DEVICE), {(vlan_id, $\perp$)})}.<br>assigned_pperms((Bandwidth Monitoring, {(attachment_point, $\perp$)})) = {((getBandwidthConsumption, PORT-STATS), {(attachment_point, $\perp$)})}.<br>assigned_pperms((Flow Mod, {(dept, $\perp$), (traffic, $\perp$)})) = {((addFlow, FLOW-RULE), {(dept, $\perp$), (traffic, $\perp$)})}.<br>assigned_pperms((Packet-In Handler, {(attachment_point, $\perp$)})) = {((readPacketInPayload, PI-PAYLOAD), {(attachment_point, $\perp$)})}. |
| – | app_sessions(Data Usage Cap Mngr) = {DataUsageAnalysisSession, DataCapEnforcingSession}.<br>app_sessions(Intrusion Prevention App) = {IntrusionPreventionSession}. |
| – | session_roles(DataUsageAnalysisSession) = {(Device Handler, {(vlan_id, 1)}), (Bandwidth Monitoring, {(attachment_point, {0x1:1, 0x1:2, 0x2:1})})}.<br>session_roles(DataCapEnforcingSession) = {(Flow Mod, {(dept, {CS}), (traffic, web)})}.<br>session_roles(IntrusionPreventionSession) = {(Device Handler, {(vlan_id, 2)}), (Packet-In Handler, {(attachment_point, {0x3:1})}), (Flow Mod, {(dept, {CE}), (traffic, web)})}. |
| – | avail_session_pperms(DataUsageAnalysisSession) = {((queryDevice, DEVICE), {(vlan_id, 1)}),<br>((getBandwidthConsumption, PORT-STATS), {(attachment_point, {0x1:1, 0x1:2, 0x2:1})})}.<br>avail_session_pperms(DataCapEnforcingSession) = {((addFlow, FLOW-RULE), {(dept, {CS}), (traffic, web)})}.<br>avail_session_pperms(IntrusionPreventionSession) = {((queryDevice, DEVICE), {(vlan_id, 2)}), ((readPacketInPayload, PI-PAYLOAD), {(attachment_point, {0x3:1})}). ((addFlow, FLOW-RULE), {(dept, {CE}), (traffic, web)})}. |
| **4.Parameter Verification Functions:** | |
| – | VERIFIERS = {VDeviceVlan, VStatsAttachpoint, VRuleSwitch, VRuleTraffic, VPInAttchpoint}. |
| – | param_verifier((DEVICE, vlan_id)) = VDeviceVlan.<br>param_verifier((PORT-STATS, attachment_point)) = VStatsAttachpoint.<br>param_verifier((FLOW-RULE, dept)) = VRuleSwitch.<br>param_verifier((FLOW-RULE, traffic)) = VRuleTraffic.<br>param_verifier((PI-PAYLOAD, attachment_point)) = VPInAttchpoint. |

**Table 4.9**: Configuration of parameter engine functions for use case of section 4.7 (Part 3).

```
A. Verifiers:
A.1. VDeviceVlan(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
    //assume a request from app Data Usage Cap Mngr via DataUsageAnalysisSession with the following:
    //ob = host tagged with vlan_id=1
    //pvpair = (vlan_id, 1)
    (ob.vlan_id = pvpair.val); //will return true
}
A.2. VStatsAttachpoint(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
    //assume a request from app Data Usage Cap Mngr via DataUsageAnalysisSession with the following:
    //ob = 0x1:1
    //pvpair = (attachment_point, {0x1:1, 0x1:2, 0x2:1: 0x2:2})
    (ob ∈ pvpair.val); //will return true
}
A.3. VRuleSwitch(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
    //assume a request from app Data Usage Cap Mngr via DataCapEnforcingSession with the following:
    //ob = flow_rule$_{[switch\_id=0x2, tcp\_dst=80, ...]}$
    //pvpair = (dept, {CS})
    //switches(CS) = {0x1, 0x2}
    (∃d ∈ pvpair.val : ob.switch_id ∈ switches(d)); //will return true
}
A.4. VRuleTraffic(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
    //assume a request from app Data Usage Cap Mngr via DataCapEnforcingSession with the following:
    //ob = flow_rule$_{[switch\_id=0x2, tcp\_dst=80, ...]}$
    //pvpair = (traffic, web)
    (ob.tcp_dst ∈ protocol_ports(pvpair.val)); //will return true
}
A.5. VPInAttchpoint(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
    //assume a request from Intrusion Prevention App via IntrusionPreventionSession with the following:
    //ob = packet-in message with source switch_id = 0x3
    //and out_port = 1
    //pvpair = (attachment_point, {0x3:1})
    (attachment_point(ob.switch_id, ob.out_port) ∈ pvpair.val); //will return true
}
```

```
B. CandidateVerifiers(ot: OBTS, pvpairs : $2^{PVPAIRS}$){
    verifiers = {};
    For each $p_i$ ∈ {dept, traffic} do
        $V_1$ = param_verifier(FLOW-RULE, dept); //$V_1$=VRuleSwitch.
        verifiers := verifiers ∪ VRuleSwitch;
        $V_2$ = param_verifier(FLOW-RULE, traffic); //$V_2$=VRuleTraffic.
        verifiers := verifiers ∪ VRuleTraffic;
    return verifiers; //verifiers = {VRuleSwitch, VRuleTraffic}.
}
```

```
C. ParamCheck(s: SESSIONS, op: OPS, ob: OBS, pvpairs: $2^{PVPAIRS}$){
    //Example for flow rule insertion by DataCapEnforcingSession.
    verifiers = CandidateVerifiers(type(flow_rule$_{[switch\_id=0x2, tcp\_dst=80, ...]}$), {(dept, {CS}),
(traffic, web)}).
    VRuleSwitch(DataCapEnforcingSession, addFlow, flow_rule$_{[switch\_id=0x2, tcp\_dst=80, ...]}$,
(dept, {CS}));
    VRuleTraffic(DataCapEnforcingSession, addFlow, flow_rule$_{[switch\_id=0x2, tcp\_dst=80, ...]}$,
(traffic, web));
    return true;
}
```

0x1:2, 0x2:1, 0x2:2)). When this application notices high transmission of packets destined to the web server, it inserts flow rules to block sender's traffic. This app is authorized to handle web traffic only. For that reason it is assigned to to the parameiretized role (Flow Mod, (dept, CS), (traffic, web)). 'Data Usage Cap Mngr' is allowed to read information about hosts with vlan_id = 1 only. For this reason it is assigned to the parameterized role (Device Handler, (vlan_id, 1)). Thie relations between apps and these parameterized roles is specified in AA relation in item 2 of table 4.7.

The function of 'Intrusion Prevention App' is to inspect packets destined to the web server in host-5. It inserts flow rules to block any malicious activity destined to this web server. Because it is authorized for switch 0x3 only, the app is assigned to parameterized roles (Flow Mod, (dept, CE), (traffic, web))) and (Packet-In Handler, (attachment_point, 0x3-1).

When access requests are submitted by these apps, ParaSDN checks each access requests using the CheckAccess function described in Table 4.4. The Parameter Engine calls the verifiers to verify if apps requests are legitimate based on parameter values. Examples of verifiers for are shown in item A of Table 4.9. For example, the verifier VRuleSwitch Will be called to make sure that the flow rule is inserted in a switch under the authority of the requester app. if app1 tries to insert a flow rule in switch 0x2. The verifier VRuleSwitch will be elected as a candidate verifier based on the object type and the parameter. It will receive the object and the parameter (dept, CS) and verify, based on the condition, that the flow rule will be inserted in switches of CS department. Otherwise, a false is returned and access will be denied. The verifiers in the item A of Table 4.9 gives some assumed access requests based on the use case and the corresponding verifier's decision. The two Apps achieve these tasks via sessions. These sessions and their parameterized roles are shown in item 3 of Table 4.8 via the function session_roles.

## 4.8 Implementation and Evaluation

In order to demonstrate our proof-of-concept prototype, we developed and ran the framework in Floodlight platform v1.2 release [22]. The Floodlight platform is deployed on a virtual machine

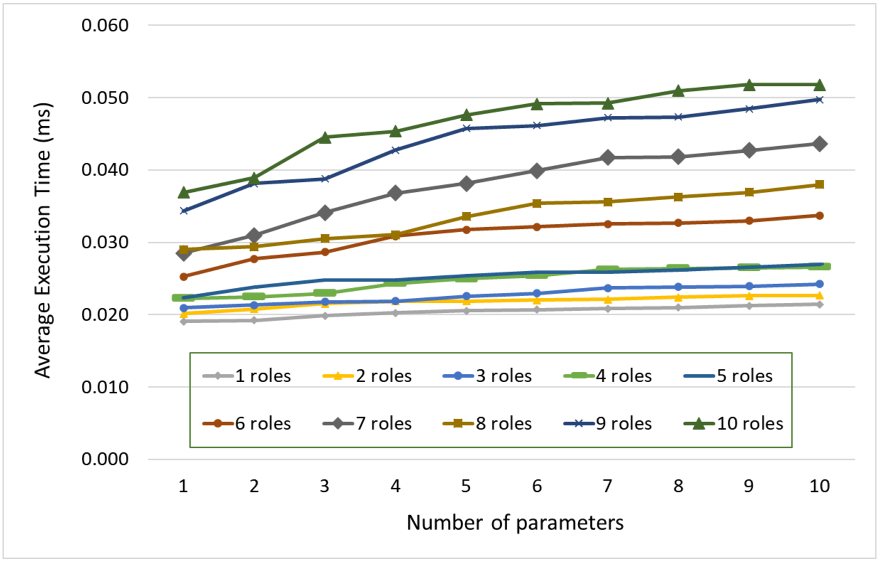


**Figure 4.5**: Average execution time required to finish the tested operations.

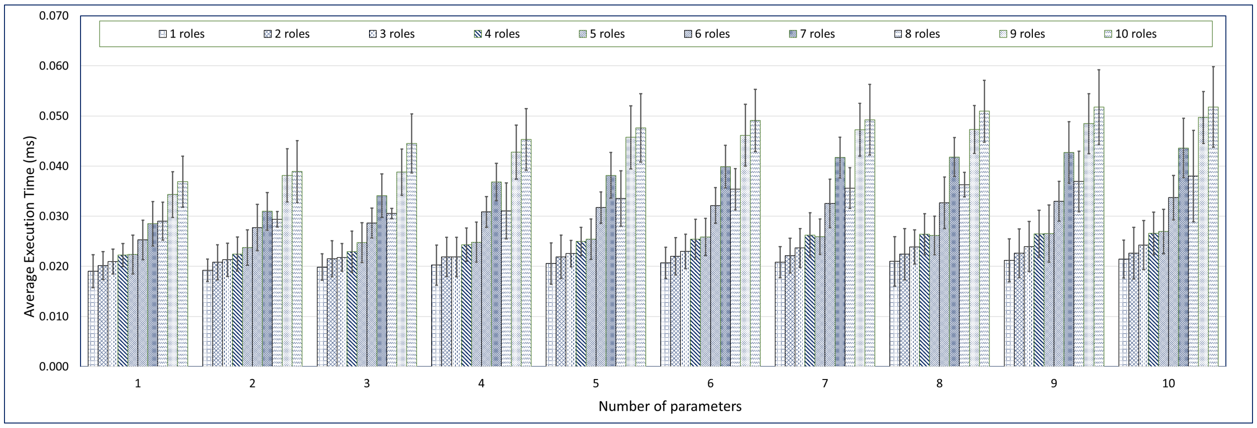


**Figure 4.6**: Average authorization time required to finish the tested operations including error bars.

that has 8GB of memory and runs on Ubuntu 14.04 OS installation. We created a topology with three virtual switches (Open vSwitch v2.3.90) connected to each other and each switch is connected to two hosts. Switches are connected to the controller and hosts are virtual machines that has 2GB and run Ubuntu 14.04 OS server.

We implemented our ParaSDN system in Floodlight platform and used hooking techniques without any change to the code of Floodlight modules. We implemented hooking for all operations exposed by Floodlight services to controller apps. We used AspectJ [6] which is a seamless aspect-oriented extension to Java. Our system intercepts methods before execution. When a session issues a request the hooked API invokes the ParaSDN components for performing access verification and

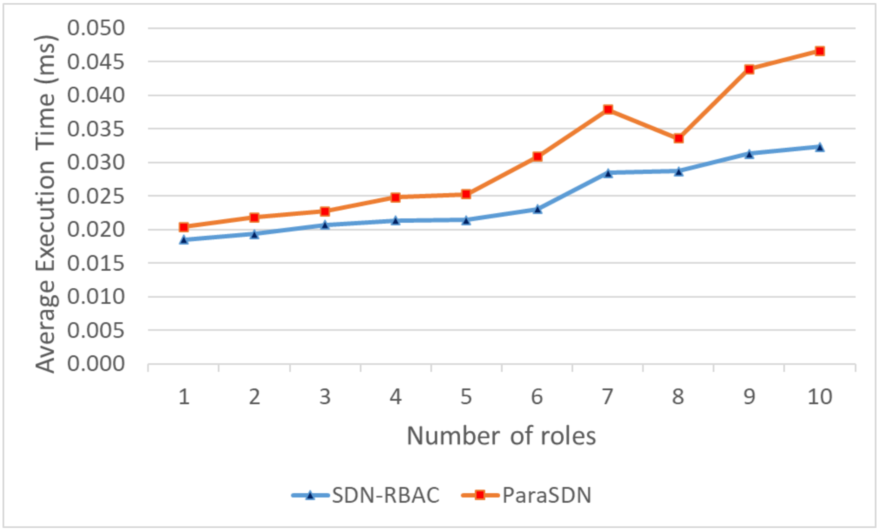


**Figure 4.7**: Overhead imposed by parameters in ParaSDN compared to SDN-RBAC system.

reply back. This system can be deployed to all other Java-based SDN controllers.

App requests are intercepted by our framework before reaching to the SDN service. Access will be provided by the service only after successful authorization check. During the lifetime of the app, our access control system keeps mediating all sessions access requests for performing security authorizations. It can identify each session, mediate each access request and send it for authorization check based on ParaSDN configuration.

To evaluate the performance of ParaSDN, we created a test app and assigned the app fifty network operations. The purpose is to perform a pressure test on ParaSDN by executing these operations with different security configurations. Each security configuration is characterized by the number of parameterized roles assigned to the app and number of parameters associated with each of them. We created test parameters and associated them with parameterized roles and created corresponding test verifiers. For each security configuration, the test is repeated for hundred times.

In the first configuration, the fifty operations are executed with one fixed parameterized role and varying number of test parameters. The total authorization time is reported for these fifty operations as shown in Fig. 4.5. Each subsequent test is performed by assigning one more parameterized role to the app and repeating the same previous test with varying number of parameters until ten roles. The execution time for all tests is reported as shown in Fig. 4.5 and the same results including the

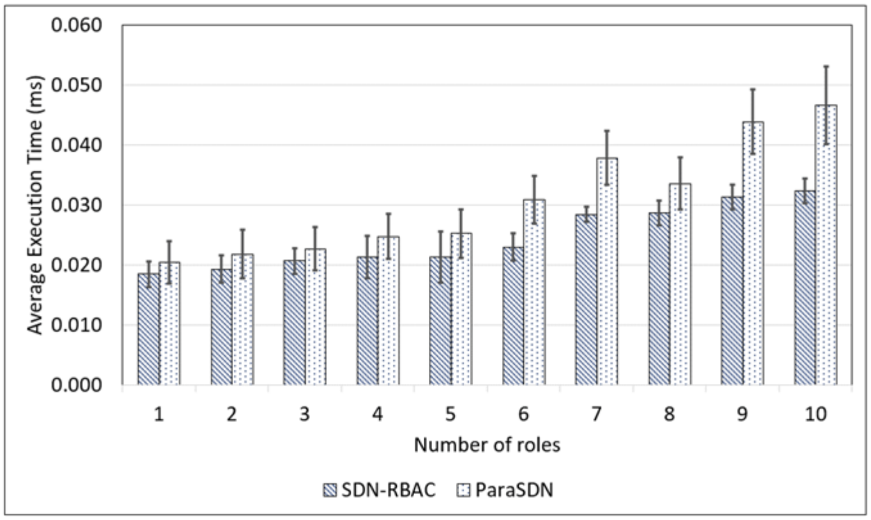


**Figure 4.8**: Overhead imposed by parameters in ParaSDN compared to SDN-RBAC system including error bars.

error bars are shown in Fig. 4.8.

The results in Fig. 4.5 demonstrates that the latency overhead of ParaSDN increases linearly with the number of parameters and the number of roles, thus ParaSDN is highly scalable even if the number of parameters and the complexity of security configuration grow in the future.

To compare the overhead imposed by parameters in ParaSDN with the one without using Parameters, i.e, the SDN-RBAC system, we repeated the same test on SDN-RBAC. We computed the average times required to finish all parameters with fixed number of roles and aligned the results of with that of SDN-RBAC. The results are shown in Fig. 4.7 and repeated again with error bars in Fig. 4.8. The overall results show that ParaSDN adds negligible overhead to the Floodlight controller which doesn't impact the whole controller's performance.

# CHAPTER 5: A MODEL FOR THE ADMINISTRATION OF ACCESS CONTROL IN SDN USING CUSTOM PERMISSIONS

In this chapter, we first present an extension to SDN-RBAC by introducing tasks, and then we present an administrative model for administering app-role and task-role relations, referred to as SDN-RBACa.

## 5.1 Motivation and Scope

The centralized SDN controller in conjunction with network operations provided by controller services result in a programmable network. This programmability allows network administrators to provide network services that enable more flexible, customized, and intelligent networking through applications. SDN offers the possibility for SDN applications to further extend the functionality of the network. These and other features make SDN suitable for technologies like Cloud Computing [7] and IoT [38].

Access rights of network apps must follow the minimum privilege principle. Recently, various methods have been proposed for adopting RBAC for the management of access rights of network apps [3, 4, 41, 49]. Thus, administration of access rights of network apps is necessary. In large SDNs with possibly large number of network apps, and with the possibility of an increased number of network services provided by the controller, the number of roles can be in the hundreds or thousands, and apps can be in the tens or hundreds or thousands. Managing permissions, roles, apps, and their interrelationships could be a tremendous task which needs simplification.

Because the motivation behind adopting RBAC [20, 46] for SDN is to simplify the administration of app authorizations, and because the most commonly carried out administrative activities in SDN-RBAC are maintaining the app-role and permission-role relations, in this chapter we present an extension to SDN-RBAC operational model by introducing tasks, and then we present an administrative model, referred to as SDN-RBACa, for administering app-role and task-role relations. To the best of our knowledge, this is the first time in the literature a model is presented for the administration of access control in SDN.

For designing our administrative model, we adopt concepts from Uni-ARBAC [11] administrative model because it unifies the administrative principles and novel concepts from many administrative models in the literature [16, 17, 33, 37, 43, 44, 52]. Following Uni-ARBAC, in our model instead of administering individual permissions, permissions are combined into tasks which are assigned to roles as a unit. Moreover, roles and tasks are partitioned and assigned into administrative units. Apps are assigned to app-pools from where individual apps are assigned to roles. Administrative users in an admin unit can assign apps to roles only if these apps (via app pools) and these roles are assigned to the admin unit in which this user is a member.

## 5.2 Administrative Units in SDN

Small SDN networks with small number of SDN apps and roles could be managed easily by a single administrator or a single admin unit that handles all network functions and all traffic types. As SDN networks grow larger with more apps, however, they become more complex and difficult to centrally manage all access control components and their associations by a single, fully-trusted administrative authority. Thus, access control administration has to be decentralized into multiple partially-trusted administrative authorities which are assigned appropriate power to change portions of the RBAC state.

In our model, rather than having administrative roles, we adopt the concept of Administrative Units (AU) [11] to decentralize access control administration for SDN-RBAC. In large SDNs, the need for specialized apps to deal with specific network traffic becomes more prominent. For example, Web load balancer, VoIP load balancer, Web Firewall, VoIP Firewall, etc. In order to enable the administration of SDN-RBAC, we have to engineer administrative units based on the functions of network apps and their access rights. For example based on traffic types or organizational entities (e.g., department in a campus network, tenant network slice, etc.).

Engineering of admin units requires that each admin unit manages an exclusive set of roles which is not under the authority of another admin unit. If administration is divided into multiple admin units, each specialized with one traffic type, for example web admin unit, VoIP admin unit, email admin unit, and ftp admin unit, this makes each admin unit responsible for managing exclusive set of roles that handle similar network functions. In this case, web admin unit exclusively manages roles related to network functions that handle web traffic. Similarly VoIP admin unit exclusively manages roles related to network functions that handle VoIP traffic. In another scenario, if admin units are divided based on organizational entities, multiple tenants for example, then each admin unit manage exclusive roles of one tenant. And this admin unit authorizes SDN apps of this tenant (via role assignment) to independently operate on this tenant's resources.

Practically, if an operation allows access to a wide range of resources, and these resources need to be managed by different admin units, this precludes the flexibility in engineering appropriate admin units. The flexibility stems from the presence of operations fine grained enough to provide the convenience in engineering set of roles exclusive for each admin unit.

Because engineering of admin units requires that each admin unit manages an exclusive set of roles and, to some extent, exclusive set of resources which can be accessed by permissions in these roles, it is vital for the operations/APIs exposed by the system under consideration to be fine grained enough to the level necessary to engineer these roles. Otherwise, engineering of such admin units will be infeasible.

Unfortunately, the currant state of the art SDN controllers doesn't provide such fine grained network operations. For example, an app with the permission to add a flow rule can insert a flow rule that manipulates any traffic type. Also, it can insert the flow rule in any switch reachable by the controller. From an administrative point of view, this precludes the coexistence of different administrative units for access control in SDN. For example, to engineer Web AU and VoIP AU, it is necessary to engineer set of roles that only handle web traffic and another set of roles that only handles VoIP traffic. Each set of roles will be exclusively managed by its respective AU. Such capability is not possible by native operations provided by SDN controllers.

A solution for this problem is to create a refined version of the coarse grained operation in a way that satisfies fine grained access control needs of SDN apps and enables engineering of different

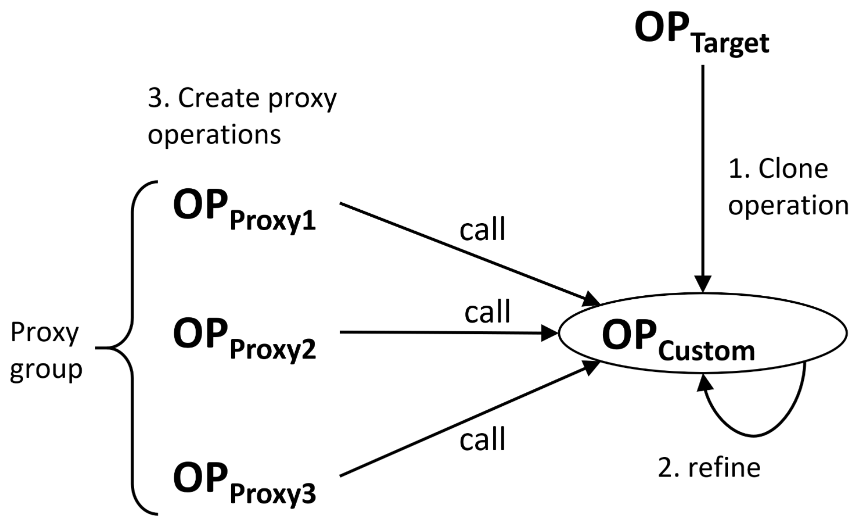


**Figure 5.1**: Target, custom, and proxy operations.

admin units necessary for access control administration. The refined version of an operation is called customized or custom operation as will be described later in the following section.

## 5.3 Custom and Proxy Operations

In this context, an SDN controller operation is a java API call submitted by an SDN controller apps to access network resources. We call these operations as target operations $OP_{Target}$ since they are the current target by SDN apps and need to be refined. We call the refined version as the custom operation $OP_{custom}$. So, a custom operation is the refined version of a target operation.

A custom operation is created by first cloning the target operation and then refining its code by adding a fine grained check on the desired attributes based on which an admin unit is defined. For example, because a web admin manages web-related roles, it requires the existence of network operations that handle only web traffic and disallow treatment of other traffic types. Thus, the target operation is refined by adding a check inside its custom operation to make sure that accessed objects are web-related only. If each custom operation will check for a specific type of traffic (e.g., web, VoIP, ftp, email), then multiple custom operations must be created, one for each traffic type. And because custom operations are exact copies of target operations, plus a refinement code added to it, this approach has some problems: i) it significantly increase the number of lines of the native code in SDN controller, ii) it requires extra effort in refining multiple very close custom operations for one target operation, and ii) it increases the compilation time of the controllers code.

To avoid such problems, we create what we call proxy operations OP$_{Proxy}$. Each proxy operation calls one custom operation and passes a parameter value based on which refinement will be done. The general process for creating custom and proxy operations and their interaction is schematically depicted in Fig. 5.1. The process starts by cloning the target operation OP$_{Target}$ that should be refined. The new resulted operation OP$_{Custom}$ is refined first by adding a new formal parameter to the parameter list of OP$_{Target}$. Then it is further refined by adding the statements to either check the accessed object against the parameter value or adding statements to filter out unauthorized objects based on the parameter value. Proxy operations can be considered as abstractions for custom operations.

Each proxy operation contains a simple call to the custom operation. This call passes a parameter value to the custom operation based on which the refinement will be done and specific objects will be accessed. For ease of reference and review, the name of the proxy operation should reflect the parameter value passed to its custom operation.

By customizing the operations in such a way, a proxy operation becomes not only fine-grained but also expressive and makes the design of access control systems and their administration much easier. Figure 5.2 shows an example for creating a custom operation addFlow(.., traffic) for addFlow operation and then creating three proxy Operations addWebFlow, addVoIPFlow, and addFtpFlow. Using proxy operations makes OP$_{Custom}$ and its parameter abstract from the app. This prevents the app from providing values for parameters of custom operations while submitting any access request.

## 5.4 Custom Permissions

Custom permissions are those permissions that are created using the proxy operations. For example, in SDN-RBAC, the permissions (addFlow, FLOW-RULE) uses the target operation addFlow.

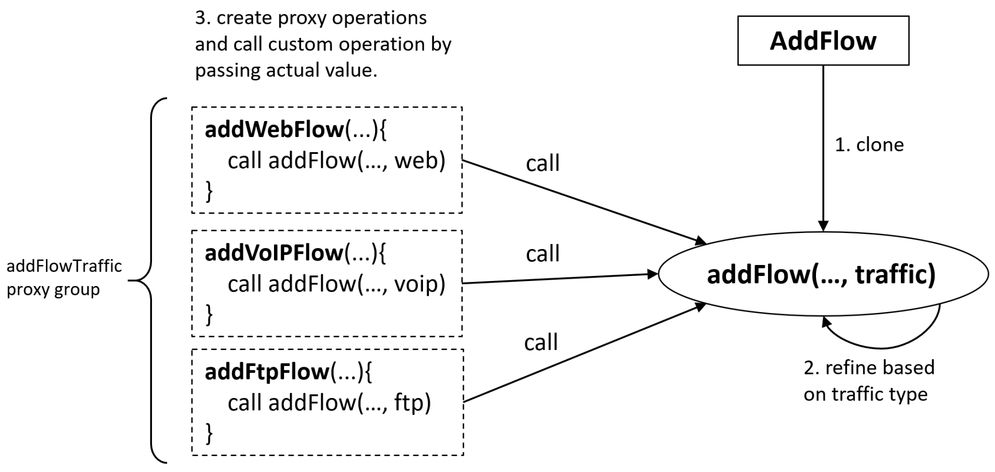

3. create proxy operations and call custom operation by passing actual value.
AddFlow
1. clone
addWebFlow(...){ call addFlow(..., web) }
addVoIPFlow(...){ call addFlow(..., voip) }
addFtpFlow(...){ call addFlow(..., ftp) }
call
call
call
addFlow(..., traffic)
2. refine based on traffic type
addFlowTraffic proxy group

**Figure 5.2**: Example of custom and proxy operations for the target operation addFlow .

After creating the custom operation addFlow(traffic), we create the proxy permissions (addWebFlow, FLOW-RULE) and (addVoIPFlow, FLOW-RULE) for adding flow rules that handle web and VoIP traffic, respectively.

A *proxy group* is the group of operations that invoke the same custom operation and pass different parameter values. Members in a proxy group allow access to different set of objects. Therefore, permissions composed of different proxy operations in one proxy group allows for the creation of specialized roles. This enables exclusive role management by different admin units.

## 5.5 SDN-RBACa Model

In this section, we describe the SDN-RBACa administrative model, along with its formal definitions. The overall structure of SDN-RBACa is illustrated in Fig. 5.3. We consider SDN-RBACa in two parts: the operational model for SDN-RBAC with respect to regular roles and permissions as well as tasks which will be introduced shortly, and the administrative model for administering the app role and task-role relations of the former. These are discussed in the following subsections.

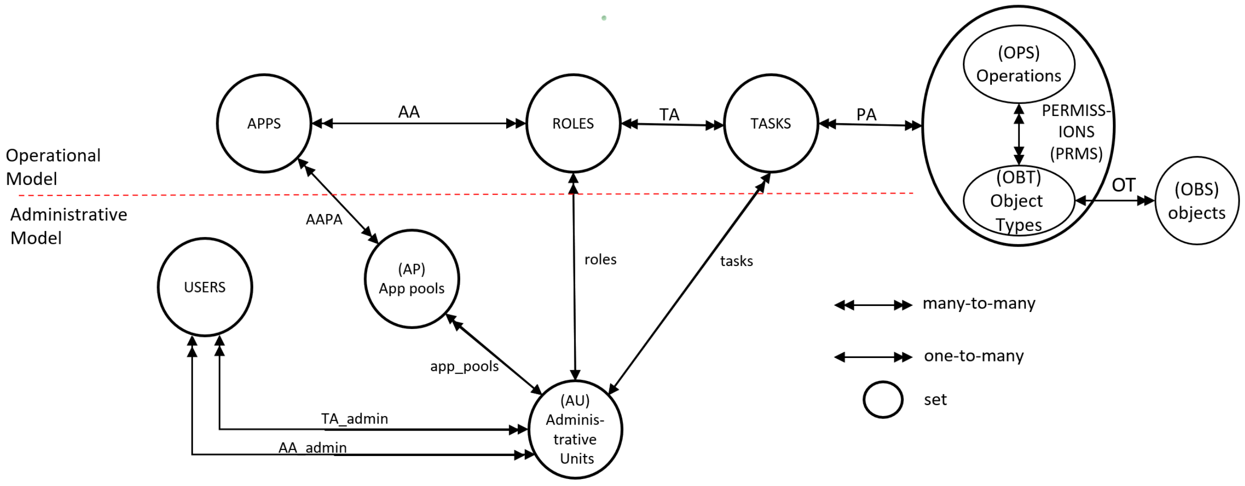


**Figure 5.3**: Conceptual model of SDN-RBACa.

### 5.5.1 Introducing Tasks

We view a *task* as a named set of several related permissions that represent a unit of network function for SDN apps. Adopting tasks for SDN-RBAC [3] has some administrative motivations. i) Because custom permissions (see section 5.4) increase the number of total permissions currently available in the SDN controller, using tasks reduces the extra management overhead entailed from these newly resulted custom permissions. ii) In role engineering process, task-to-role assignment is a more convenient abstraction than assigning individual permissions, especially when these permissions are related. Therefore, adopting tasks as a basic component in SDN-RBAC reduces administration overhead typically associated with managing fine-grained permissions. In the next section we show a the SDN-RBAC model with tasks as a basic component.

### 5.5.2 SDN-RBACa Operational Model

The sets and relations in the top part of Fig. 5.3 represent the SDN-RBACa operational model, which is slightly different from the SDN-RBAC model [3]. The most distinguished difference is that there is a level of indirection in role-permission assignment, so permissions are assigned to tasks and tasks are assigned as units to roles. Adopting tasks has several motivations, as discussed in Section 5.5.1. App-role assignment remains unchanged from SDN-RBAC. For simplicity, we

**Table 5.1**: Formal Definition of SDN-RBACa Administrative Model.

**1.Basic Sets**
- APPS is a finite set of SDN apps.
- OPS is a finite set of operations.
- OBS is a finite set of objects.
- OBTS is a finite set of object types.
- PRMS $\subseteq$ OPS $\times$ OBTS , set of permissions.
- ROLES is a finite set of roles.
- TASKS is a finite set of tasks.
- AP is a finite set of app-pools.
- USERS is a finite set of administrative users.
- AU is a finite set of administrative units.

**2. Assignment Relations (operational):**
- PA $\subseteq$ PRMS $\times$ TASKS, permission-task assignment relation.
- TA $\subseteq$ TASKS $\times$ ROLES, task-role assignment relation.
- AA $\subseteq$ APPS $\times$ ROLES, app-role assignment relation.
- OT $\subseteq$ OBS $\times$ OBTS, a many-to-one mapping an object to its type, where $(o,\ t_1) \in$ OT $\wedge$ $(o,\ t_2) \in$ OT $\Rightarrow t_1 = t_2$.

**3. Derived Functions (operational):**
- type: (o: OBS) $\rightarrow$ OBTS, a function specifying the type of an object. Defined as type(o) = $\{t \in$ OBTS $\mid (o,\ t) \in$ OT$\}$.
- authorized_perms(r: ROLES) $\rightarrow 2^{PRMS}$, defined as authorized_perms$(r)$ = $\{p \in$ PRMS $\mid \exists$t$\in$TASKS, $\exists$r$\in$ROLES : $(t,\ r) \in$ TA $\bigwedge$ $(p,\ t) \in$ PA$\}$.

**4. App Authorization Function:**
- can_exercise_permission(a: APPS, op: OPS, ob: OBS) = $\exists$r $\in$ ROLES : (op, type(ob)) $\in$ authorized_perms(r) $\bigwedge$ (a, r) $\in$ AA.

**5. Administrative App-pools Relation:**
- AAPA $\subseteq$ APPS $\times$ AP, app to app-pool assignment relation.

**6. Administrative Units and Partitioned Assignment:**
- roles(au : AU) $\rightarrow 2^{ROLES}$ , assignment of roles, where r $\in$ roles(au$_1$) $\wedge$ r $\in$ roles(au$_2$) $\Rightarrow$ au$_1$= au$_2$.
- tasks(au : AU) $\rightarrow 2^{TASKS}$ , assignment of tasks, where t $\in$ tasks(au$_1$) $\wedge$ t $\in$ tasks(au$_2$) $\Rightarrow$ au$_1$= au$_2$.
- app_pools(au : AU) $\rightarrow 2^{AP}$ , assignment of app-pool, where ap $\in$ app_pools(au$_1$) $\wedge$ ap $\in$ app_pools(au$_2$) $\Rightarrow$ au$_1$= au$_2$.

**7. Administrative User Assignment:**
- TA_admin $\subseteq$ USERS $\times$ AU.
- AA_admin $\subseteq$ USERS $\times$ AU.

8. **Administrative User Authorization Functions:**
- can_manage_task_role(u : USERS, t : TASKS, r : ROLES) = $\exists$au$\in$AU : (u , au ) $\in$ TA_admin $\bigwedge$ r $\in$ roles(au ) $\bigwedge$ t $\in$ tasks(au)
- can_manage_app_role(u : USERS, a : APPS, r : ROLES) = $\exists$au$\in$AU : ((u, au) $\in$ AA_admin $\bigwedge$ r $\in$ roles(au)) $\bigwedge$ $\exists$ap$\in$AP : ((a , ap) $\in$ AAPA $\bigwedge$ ap $\in$ app_pools(au)).

9. **Administrative Actions:**
- assign_task_to_role(u: USERS, t: TASKS, r: ROLES)
  Authorization condition: can_manage_task_role(u, t, r) = True
  Effect: TA' = TA $\cup$ {(t, r)}.
- revoke_task_from_role(u: USERS, t: TASKS, r: ROLES)
  Authorization condition: can_manage_task_role(u, t, r) = True
  Effect: TA' = TA $\setminus$ {(t, r)}.
- assign_app_to_role(u: USERS, a: APPS, r: ROLES)
  Authorization condition: can_manage_app_role(u, a, r) = True
  Effect: AA' = AA $\cup$ {(a, r)}.
- revoke_app_from_role(u: USERS, a: APPS, r: ROLES)
  Authorization condition: can_manage_app_role(u, a, r) = True
  Effect: AA' = AA $\setminus$ {(a, r)}.

have not considered the SDN-RBAC concepts of sessions and role activation.

The SDN-RBACa operational model is formalized in Table 5.1. The first six components from Item 1 specify the basic sets carried over from SDN-RBAC. TASKS is the set of tasks added to SDN-RBAC. The last three sets belong to the administrative model (see section 5.5.3). Item 2 specifies the assignment relations in the operational model including the additional components which effect the additional indirection between permissions and roles via tasks. Item 3 shows the *type* derived function and shows the *authorized_perms* function which formalizes the interaction between the permission-task and task-role assignments. The authorization function in item IV specifies the authorization required for an app to exercise a permission and access an object, which is that the permission must be authorized to at least one role assigned to the app.

### 5.5.3 SDN-RBACa Administrative Model

In this section we describe the SDN-RBACa administrative model illustrated in the lower part of Fig. 5.3, and formalized in Table 5.1. The administrative model introduces a number of additional components.

First we have the notion of app-pools. Examples of app-pool include 'Web Load Balance Pool' and 'Web Security Pool' as will be described in the use case in Section 5.7. Adopting app-pool facilitates the allocation of several apps that achieve similar network functions to an admin unit. The set of app-pools is denoted as AP. Apps are assigned to app-pools via the AAPA app to app-pool assignment relation which is formally specified in item 5 of Table 5.1.

The set of administrative units is denoted as AU. SDN-RBACa requires that roles are partitioned into different admin units and each role is allocated to exactly one unit for administration. In other words, each admin unit manages an exclusive set of roles which is not under the authority of another admin unit. This roles partitioning is formally specified using the *roles* function in item 6 of Table 5.1. The partitioning concept is further applied to tasks and app-pools via the *tasks* and *app_pools* functions in item 6 of Table 5.1.

The result of roles, tasks, and app-pools partitioning is that an admin unit manages an explicitly assigned partition of roles, to which it can assign apps from an assigned partition of app-pools and tasks from an assigned partition of tasks. The outcome of this partitioning directly impacts the results of authorization functions specified in item 8 of Table 5.1.

Assignment of administrative users to admin units can be done via the TA_admin or the AA_admin relation. An administrative user in TA_admin is authorized to perform the administrative actions which assign a tasks to a roles, while a user in AA_admin is authorized to perform the administrative actions which assign a apps to a roles. It should be mentioned that these capabilities can be separately assigned to two different administrative users, even though they assigned to one administrative unit. Such administrative actions bring apps and permissions together and, in some critical SDN networks, they are best to be done by different network administrators.

Item 8 of Table 5.1 specifies the authorization functions for administrative users. The function can_manage_task_role returns whether a given admin user can assign/revoke a given task to/from a given role. The requirement is that this user must be assigned as TA_Admin to the unique administrative unit which has exclusive authority over this role and this task.

Similarly, can_manage_app_role is an authorization function that returns true or false. This function specifies the conditions for a given user to assign/revoke a given app to/from a given role. The requirement is that this user must be assigned as AA_Admin to the unique administrative unit which has exclusive authority over this role and over an app-pool to which this app is directly assigned via AAPA relation.

The last item in Table 5.1 formalizes the four administrative actions to assign/revoke a task to/from a role or an app to/from a role. This supports the reversibility principle which requires that administrative actions should be reversible. If an administrative user makes a mistake, they can go back.

## 5.6 Task and Role Engineering for SDN using Custom Permissions

In the following two subsections, we discuss the process of engineering tasks and roles using custom permissions. The abstract process is illustrated in Fig. 5.4 and an example is described in

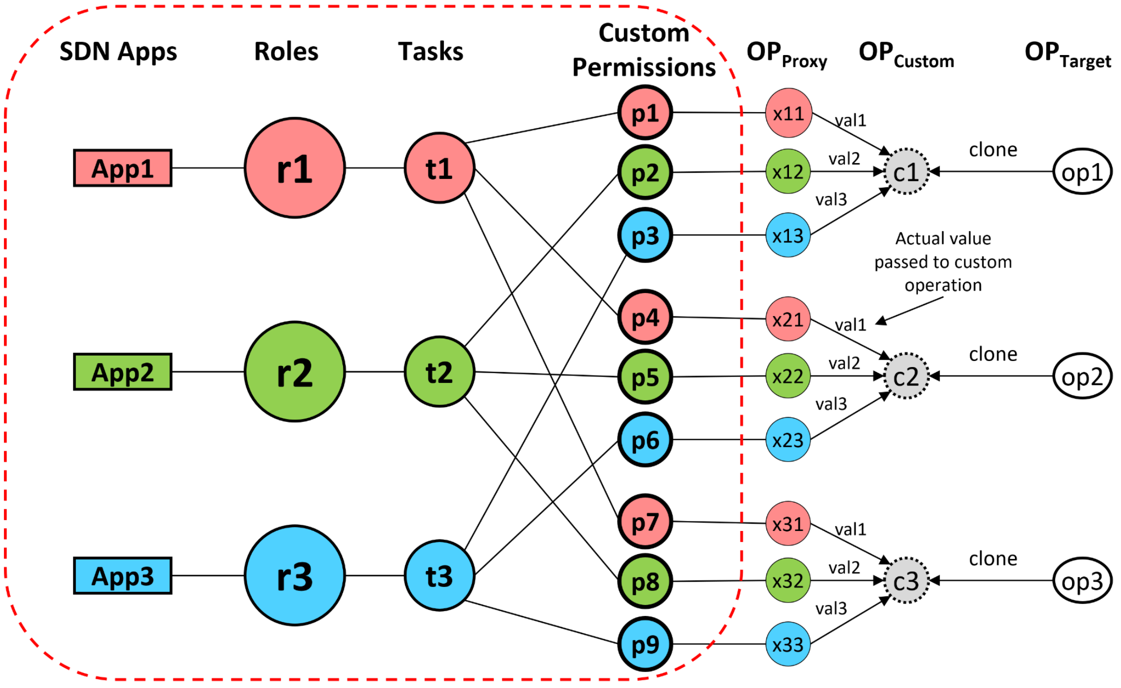


**Figure 5.4**: Conceptual representation of associations between custom permissions, tasks, roles, and apps.

Section 5.6.2.

### 5.6.1 Tasks and Roles with Custom Permission

Because each custom permission is created using a proxy operation, it enables access to specific fine grained resource known in advance before task or role engineering. Now, lets compare the use of a target operation against a proxy operation in creating a permission. As shown in Fig. 5.4, three proxy operations (x11, x12, and x13) are resulted from target operation op1, each one provides access to a resource more fine grained than what is originally provided by the target operation op1. This makes p1 = (x11, ot) more fine grained compared to using op1 to create the same permission, i.e., (op1, ot), where ot is some object type. In turn, because we assign the custom permission p1 to task t1, this makes t1 a fine grained, or more specialized, task. Again, this is compared to using op1 in the first place to engineer the same task. As shown in Fig. 5.4, task t1 is engineered with the three custom permissions p1, p4, and p7 created using the proxy operations x11, x21, and x31, respectively. Each one provides more fine grained access, and thus makes task t1 more fine grained

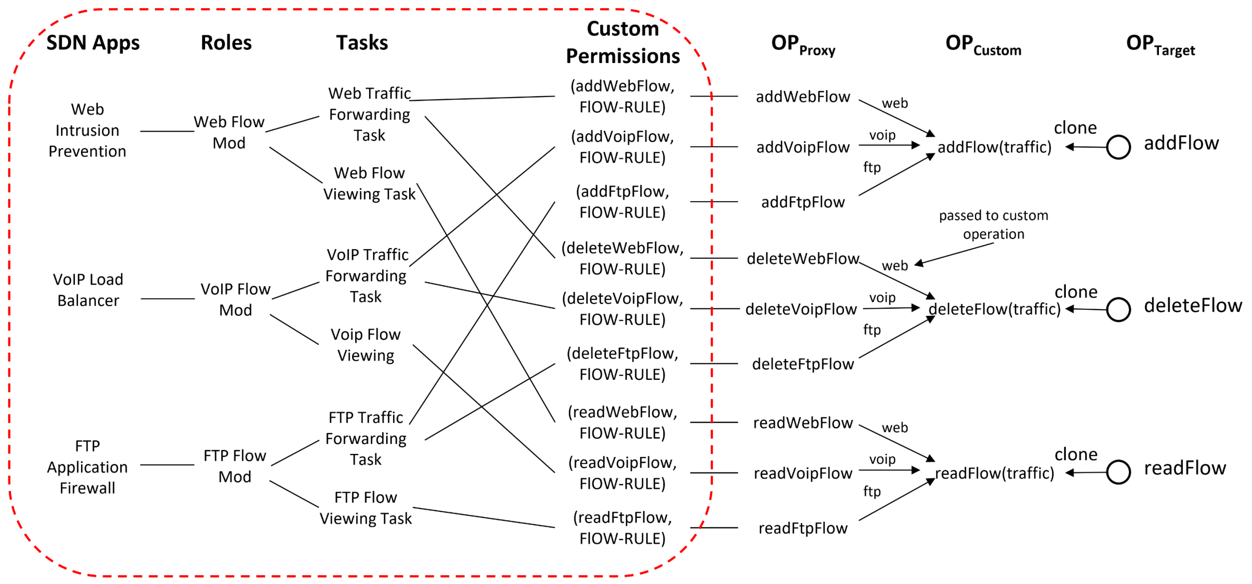


**Figure 5.5**: Example of creating three roles using custom permissions and their associations with tasks and apps.

compared to using the target operations op1, op2, and op3 to create the same permissions. More importantly, this process allows for the creation of more specialized tasks like t2 and t3 in the same way.

The granularity of access resulted from using proxy operations to create custom permissions escalates to roles. For example, roles r1, r2 and r3 in Fig. 5.4 provide more fine grained and specialized access to network resources. Now, imagine that we want to engineer three admin units au1, au2, and au3, each specialized with managing resources accessed by t1, t2, and t3, respectively, then we simply assign each task and to its respective admin unit, and do the same thing with roles r1, r2, and r3. On the contrary, starting the process with op1, op2, and op3 to engineer these roles and tasks preclude the possibility of creating the required admin units.

### 5.6.2 Custom Permissions with ‘Flow Mod’ Role

In this section, we describe an example using the ‘Flow Mod’ SDN role to illustrate the creation of nine proxy operations from three target operations (via custom operations), namely, addFlow, deleteFlow, and readFlow. The example is depicted in Fig. 5.5. These target operations allow network apps to access flow rules that handle any type of traffic. If it is required to have three administrative units, each specialized with one type of traffic, namely, Web, VoIP, and FTP, and if these three target operations are assigned to the three admin units (via permissions, tasks, and roles), this means that an app, specialized with web flows, for example, might access unauthorized flow rules that handle non-web traffic. To solve this problem, three proxy groups are created, one for each target operation. Each proxy operation in a proxy group is specialized with one traffic type. Now, for the 'Web Admin Unit', which is specialized with Web traffic, three custom permissions, namely, (addWebFlow, FLOW-RULE), (deleteWebFlow, FLOW-RULE), and (readWebFlow, FLOW-RULE), will be created by picking the corresponding proxy operation from each proxy group as shown in Fig. 5.5. These three custom permissions contribute to the engineering of the tasks 'Web Traffic Forwarding' and 'Web Flow Viewing', which will be under exclusive authority of 'Web Admin Unit'. These two tasks will be assigned to the role 'Web Flow Mod', which also will be under exclusive authority of the same admin unit. This role can be assigned only by administrative users who are members in 'Web Admin Unit' to apps that handle web traffic and belong to the authority of the same admin unit, such as 'Web Intrusion Prevention'. The same idea applies to 'VoIP Flow Mod' and 'FTP Flow Mod' roles.

## 5.7 Proof of Concept Use Cases

### 5.7.1 Basic Use Case - Web Admin Unit

In this section we discuss a proof of concept use case to demonstrate the use of custom permissions in enabling the administration of SDN-RBAC. The use case configured in the SDN-RBACa administrative model is shown in Tables 5.2 and 5.3.

The use case describes a scenario in which we have one administrative unit, called 'Web Admin Unit'. This admin unit is specialized of managing web resources. This administrative unit exclusively manages five web-related roles as listed in the set ROLES in Table 5.2. It also exclusively manages ten web-raled tasks listed in the set TASKS. All these roles and tasks provide access to web resources, such as flow rules that handle web traffic, packet in headers and payloads that contains web traffic, web pool servers, and statistics about web flows. These resources can be ac-

**Table 5.2**: Configuration of the administrative model for the use case in Section 5.7.1 - Part1.

| **1.Basic Sets** |
|---|
| – APPS = {Web Intrusion Prevention App, Web Application Firewall App, Web Load Balancer App}. |
| – OPS = { readWebPacketInPayload, readWebPacketHeader, readWebFlow, addWebFlow, updateWebFlow, deleteWebFlow, createWebPool, listWebPools, removeWebPool, updateWebPool, createWebMonitor, listWebMonitors, removeWebMonitor, updateWebMonitor, createWebVip, listWebVips, removeWebVip, updateWebVip, createWebMember, listWebMembersByPool, removeWebMember, updateWebMember, readWebFlowByteCount, readAggWebFlowByteCount, readWebFlowPacketCount, readAggWebFlowPacketCount }. |
| – OBS = set of all objects of types PI-PAYLOAD, PI-HEADER, FLOW-RULE, LB-POOL, LB-MONITOR, LB-VIP, LB-POOL-MEMBER, and FLOW-STATS. |
| – OBTS = {PI-PAYLOAD, PI-HEADER, FLOW-RULE, LB-POOL, LB-MONITOR, LB-VIP, LB-POOL-MEMBER, FLOW-STATS}. |
| – PRMS = { (readWebPacketInPayload, PI-PAYLOAD), (readWebPacketHeader, PI-HEADER), (readWebFlow, FLOW-RULE), (addWebFlow, FLOW-RULE), (updateWebFlow, FLOW-RULE), (deleteWebFlow, FLOW-RULE), (createWebPool, LB-POOL), (listWebPools, LB-POOL), (removeWebPool, LB-POOL), (updateWebPool, LB-POOL), (createWebMonitor, LB-MONITOR), (listWebMonitors, LB-MONITOR), (removeWebMonitor, LB-MONITOR), (updateWebMonitor, LB-MONITOR), (createWebVip, LB-VIP), (listWebVips, LB-VIP), (removeWebVip, LB-VIP), (updateWebVip, LB-VIP), (createWebMember, LB-POOL-MEMBER), (listWebMembersByPool, LB-POOL-MEMBER), (removeWebMember, LB-POOL-MEMBER), (updateWebMember, LB-POOL-MEMBER), (readWebFlowByteCount, FLOW-STATS), (readAggWebFlowByteCount, FLOW-STATS), (readWebFlowPacketCount, FLOW-STATS), (readAggWebFlowPacketCount, FLOW-STATS) }. |
| – ROLES = {Web Packet-In Handler, Web Packet Monitor, Web Flow Mod, Web Load Balancing, Web Stats Collector}. |
| – TASKS = { Web Deep Packet Inspection Task, Web Packet Header Inspection Task, Web Flow Viewing Task, Web Traffic Forwarding Task, Web Server Pool Management Task, Web Server Monitor Management Task, Web Pool VIP Management Task, Web Pool Member Management Task, Web Payload Statistics Collection Task, Web Packet Statistics Collection Task }. |
| – AP = {Web Load Balance Pool, Web Security Pool}. |
| – USERS = {web_functions_admin_user, web_apps_admin_user}. |
| – AU = {Web Admin Unit}. |

cessed via twenty six custom permissions as listed in the set PRMS. All these custom permissions are created using the proxy operations listed in the set OPS. The admin unit 'Web Admin Unit' exclusively manages the two web-related app-pools 'Web Load Balance Pool' and 'Web Security Pool' listed in the set AP. Members in these two pools are the three network apps, 'Web Intrusion Prevention', 'Web Application Firewall', and 'Web Load Balancer', specialized with web traffic and require access to web resources. The relation between the two app-pools and the three apps

**Table 5.3**: Configuration of the administrative model for the use case in Section 5.7.1 - Part2.

| |
|---|
| **2. Assignment Relations (operational):** |
| – PA = { {(readWebPacketInPayload, PI-PAYLOAD), (readWebPacketHeader, PI-HEADER), (readWebFlow, FLOW-RULE)} × {Web Deep Packet Inspection Task} ∪ {(readWebPacketHeader, PI-HEADER), (readWebFlow, FLOW-RULE)} × {Web Packet Header Inspection Task} ∪ {(readWebFlow, FLOW-RULE)} × {Web Flow Viewing Task} ∪ {(addWebFlow, FLOW-RULE), (updateWebFlow, FLOW-RULE), (deleteWebFlow, FLOW-RULE)} × {Web Traffic Forwarding Task} ∪ {(createWebPool, LB-POOL), (listWebPools, LB-POOL), (removeWebPool, LB-POOL), (updateWebPool, LB-POOL)} × {Web Server Pool Management Task} ∪ {(createWebMonitor, LB-MONITOR), (listWebMonitors, LB-MONITOR), (removeWebMonitor, LB-MONITOR), (updateWebMonitor, LB-MONITOR)} × {Web Server Monitor Management Task} ∪ {(createWebVip, LB-VIP), (listWebVips, LB-VIP), (removeWebVip, LB-VIP), (updateWebVip, LB-VIP)} × {Web Pool VIP Management Task} ∪ {(createWebMember, LB-POOL-MEMBER), (listWebMembersByPool, LB-POOL-MEMBER), (removeWebMember, LB-POOL-MEMBER), (updateWebMember, LB-POOL-MEMBER)} × {Web Pool Member Management Task} ∪ {(readWebFlowByteCount, FLOW-STATS), (readAggWebFlowByteCount, FLOW-STATS)} × {Web Payload Statistics Collection Task} ∪ {(readWebFlowPacketCount, FLOW-STATS), (readAggWebFlowPacketCount, FLOW-STATS)} × {Web Packet Statistics Collection Task}}. |
| – TA = {{Web Deep Packet Inspection Task, Web Packet Header Inspection Task} × {Web Packet-In Handler} ∪ {Web Packet Header Inspection Task} × {Web Packet Monitor} ∪ {Web Flow Viewing Task, Web Traffic Forwarding Task} × {Web Flow Mod} ∪ {Web Server Pool Management Task, Web Server Monitor Management Task, Web Pool VIP Management Task, Web Pool Member Management Task} × {Web Load Balancing} ∪ {Web Payload Statistics Collection Task, Web Packet Statistics Collection Task} × {Web Stats Collector}}. |
| – AA = {{Web Intrusion Prevention App} × {Web Packet-In Handler, Web Flow Mod} ∪ {Web Application Firewall App} × {Web Packet Monitor, Web Flow Mod} ∪ {Web Load Balancer App} × {Web Flow Mod, Web Load Balancing, Web Stats Collector}}. |
| – OT = {(all payloads in packet-in message, PI-PAYLOAD), (all packet header objects, PI-HEADER), (all flow-rules, FLOW-RULE), (all server pools, LB-POOL), (all server monitors, LB-MONITOR), (all pools virtual IPs, LB-VIP), (all pool members, LB-POOL-MEMBER), (all flow statistics in flow rules, FLOW-STATS)}. |
| **3. Administrative App-pools Relation:** |
| – AAPA = { (Web Intrusion Prevention App, Web Security Pool), (Web Application Firewall App, Web Security Pool), (Web Load Balancer App, Web Load Balance Pool)}. |
| **4. Administrative Units and Partitioned Assignment:** |
| – roles(Web Admin Unit) = {Web Packet-In Handler, Web Packet Monitor, Web Flow Mod, Web Load Balancing, Web Stats Collector}. |
| – tasks(Web Admin Unit) = {Web Deep Packet Inspection Task, Web Packet Header Inspection Task, Web Flow Viewing Task, Web Traffic Forwarding Task, Web Server Pool Management Task, Web Server Monitor Management Task, Web Pool VIP Management Task, Web Pool Member Management Task, Web Payload Statistics Collection Task, Web Packet Statistics Collection Task}. |
| – app_pools(Web Admin Unit) = {Web Load Balance Pool, Web Security Pool}. |
| **5. Administrative User Assignment:** |
| – TA_admin = {(web_functions_admin_user, Web Admin Unit)}. |
| – AA_admin = {(web_apps_admin_user, Web Admin Unit)}. |

are specified in AAPA relation shown in item 3 of Table 5.3.

The functions *roles*, *tasks*, and *app_pools* in item 4 show the partitioned assignment of the five web-related roles, ten web-related tasks, and two web-related app-pools to the admin unit

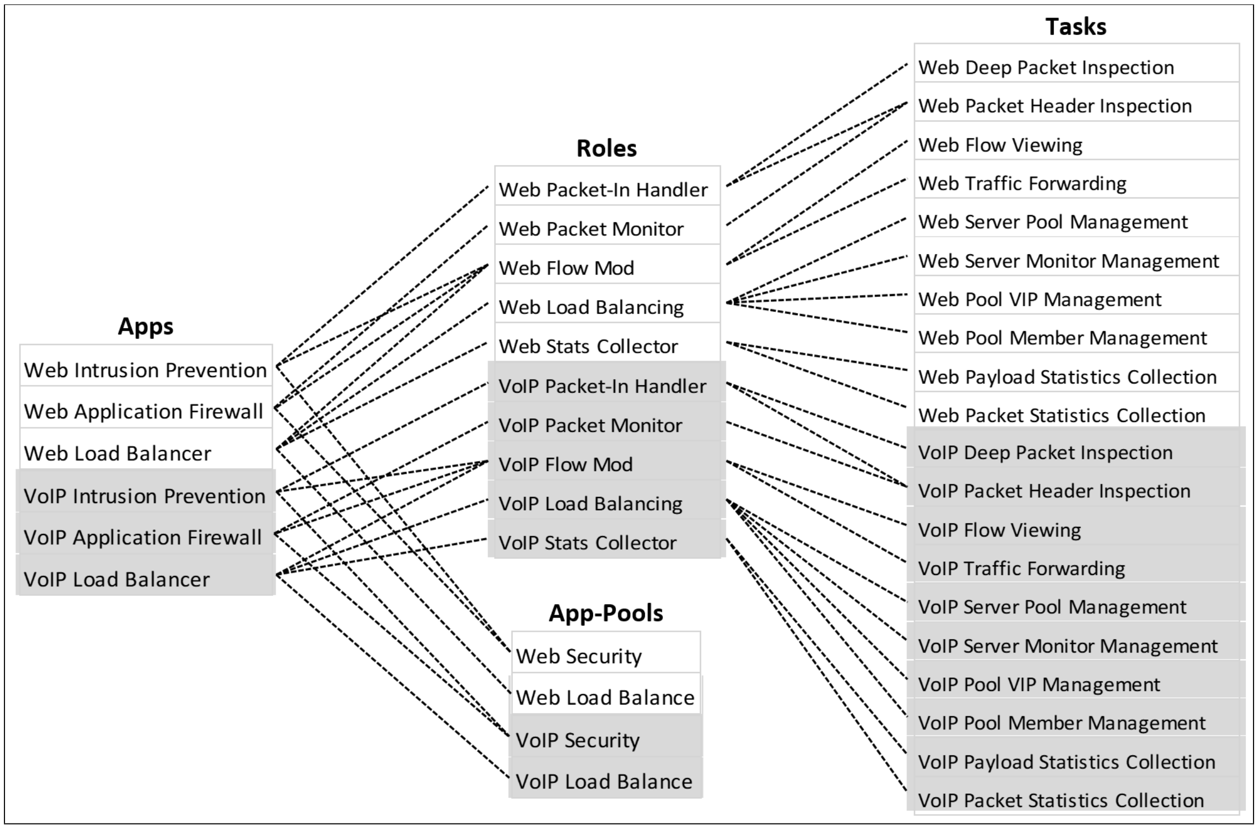


**Figure 5.6**: 'Web Admin Unit' and 'VoIP Admin Unit' (gray) along with tasks, roles, and app pools they exclusively manage. The figure also shows apps that admin units can manage via app-pools.

'Web Admin Unit'. This admin unit has two administrative users, web_functions_admin_user and web_apps_admin_user. The former is authorized, via TA_admin relation, to assign/revoke tasks to/from roles, and the later is authorized, via AA_admin relation, to assign/revoke apps to/from roles. These two relations are specified in item 5 of Table 5.3.

### 5.7.2 Extended Use Case

For the sake of simplicity and readability, the use case in Section 5.7.1 is configured using one administrative unit. In this section, we describe an extension to the use case by including more admin units. The use case in Section 5.7.1 uses only web-related proxy operation from each proxy group that handles multiple traffic types (see section 5.6.2). In this section, we show how we can use another proxy operation from each proxy group to create another administrative unit. Fig. 5.6

**1 . Examples of Authorization Functions:**

- can_manage_task_role(web_functions_admin_user, Web Traffic Forwarding Task, Web Flow Mod) = True
  ***Reason***:
  $\exists$Web Admin Unit $\in$ AU : ((web_functions_admin_user, Web Admin Unit) $\in$ TA_admin) $\bigwedge$
  Web Flow Mod $\in$ roles(Web Admin Unit) $\bigwedge$
  Web Traffic Forwarding Task $\in$ tasks(Web Admin Unit).
- can_manage_task_role(voip_functions_admin_user, Web Server Pool Management Task, Web Load Balancing) = False
  ***Reason***:
  Web Load Balancing $\in$ roles(Web Admin Unit) $\bigwedge$
  Web Server Pool Management Task $\in$ tasks(Web Admin Unit),
  however, (voip_functions_admin_user, Web Admin Unit) $\notin$ TA_admin.
- can_manage_app_role(web_apps_admin_user, Web Intrusion Prevention App, Web Flow Mod) = True
  ***Reason***:
  $\exists$Web Admin Unit $\in$ AU : ((web_apps_admin_user, Web Admin Unit) $\in$ AA_admin) $\bigwedge$
  Web Flow Mod $\in$ roles(Web Admin Unit)] $\bigwedge$
  $\exists$Web Security Pool $\in$ AP: (Web Intrusion Prevention App, Web Security Pool) $\in$AAPA $\bigwedge$
  Web Security Pool $\in$ app_pools(Web Admin Unit).
- can_manage_app_role(web_apps_admin_user, VoIP Application Firewall App, VoIP Flow Mod) = False
  ***Reason***:
  VoIP Flow Mod $\in$ roles(VoIP Admin Unit) $\bigwedge$
  (VoIP Application Firewall App , VoIP Security ) $\in$ AAPA $\bigwedge$ VoIP Security $\in$ app_pools(VoIP Admin Unit)],
  however, (web_apps_admin_user, VoIP Admin Unit) $\notin$ AA_admin.

**2 . Examples of Administrative Actions:**

- assign_task_to_role(web_functions_admin_user, Web Traffic Forwarding Task, Web Flow Mod) is allowed
  ***Reason***:
  can_manage_task_role(web_functions_admin_user, Web Traffic Forwarding Task, Web Flow Mod) = True
- revoke_task_from_role(voip_functions_admin_user, Web Server Pool Management Task, Web Load Balancing) is not allowed
  ***Reason***:
  can_manage_task_role(voip_functions_admin_user, Web Server Pool Management Task, Web Load Balancing) = False
- assign_app_to_role(web_apps_admin_user, Web Intrusion Prevention App, Web Flow Mod) is allowed
  ***Reason***:
  can_manage_app_role(web_apps_admin_user, Web Intrusion Prevention App, Web Flow Mod) = True
- revoke_app_from_role(web_apps_admin_user, VoIP Application Firewall App, VoIP Flow Mod) is not allowed
  ***Reason***:
  can_manage_app_role(web_apps_admin_user, VoIP Application Firewall App, VoIP Flow Mod) = False

**Table 5.4**: Examples of Administrative User Authorization Functions corresponding to some Administrative Actions. Examples belong to extended use case in Section 5.7.2

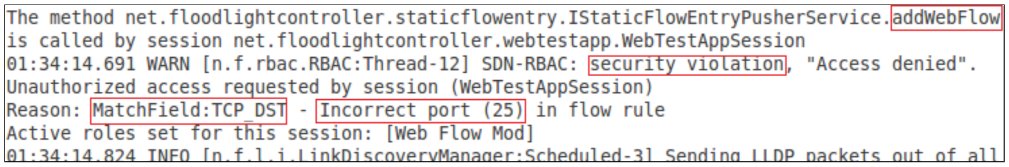
depicts an extended use case with two admin unit 'Web Admin Unit' and 'VoIP Admin Unit'. The configuration of 'Web Admin Unit' is similar to that described in section 5.7.1.

**Administrative User Assignment:**

- TA_admin = {(web_functions_admin_user, Web Admin Unit), (voip_functions_admin_user, VoIP Admin Unit)}.
- AA_admin = {(web_apps_admin_user, Web Admin Unit), (voip_apps_admin_user, VoIP Admin Unit)}.

**Table 5.5**: Administrative user assignment relation for use case in Section 5.7.2

The VoIP-related tasks in Fig. 5.6 are engineered in the same way Web-related tasks are engineered. VoIP-related tasks are engineered using custom permissions which are created using VoIP-related proxy operations. For example, three custom permissions, namely, (addVoIPFlow, FLOW-RULE), (deleteVoIPFlow, FLOW-RULE), and (readVoIPFlow, FLOW-RULE) will be used



```
The method net.floodlightcontroller.staticflowentry.IStaticFlowEntryPusherService.addWebFlow
is called by session net.floodlightcontroller.webtestapp.WebTestAppSession
01:34:14.691 WARN [n.f.rbac.RBAC:Thread-12] SDN-RBAC: security violation, "Access denied".
Unauthorized access requested by session (WebTestAppSession)
Reason: MatchField:TCP_DST - Incorrect port (25) in flow rule
Active roles set for this session: [Web Flow Mod]
01:34:14.824 INFO [n.f.l.i.LinkDiscoveryManager:Scheduled-3] Sending LLDP packets out of all
```

**Figure 5.7**: Screenshot of authorization check result for addWefFlow proxy operation requested by WebTestApp - Access denied because of incorrect tcp_port number.

to engineer the tasks 'VoIP Traffic Viewing' and 'VoIP Traffic Forwarding'. Both of these tasks will contribute to the engineering of 'VoIP Flow Mod' role. A complete configuration for this use case is given in ..

Using the same approach, we can create other admin units, for example, 'Ftp Admin Unit' and 'Email Admin Unit'. It is clear that, by the power of proxy operations and the custom permissions created from them, it becomes more flexible to create more administrative units, each one specialized with different type of traffic.

Table 5.4 shows examples of administrative user authorizations corresponding to some administrative actions based on the extended use case in this section. The table shows the results of the authorization function. The use case assumes the existence of four administrative users assigned to the two admin units as specified in Table 5.5.

## 5.8 Implementation

To demonstrate the effectiveness of custom permissions with our access control, we implemented a prototype on Floodlight, a Java based SDN controller. we developed and ran the prototype in Floodlight SDN controller v1.2 release [22]. The Floodlight platform is deployed on a virtual machine that has 8GB of memory and runs on Ubuntu 14.04 OS installation. We created a topology with three virtual switches (Open vSwitch v2.3.90) connected to each other and each switch is connected to two hosts. Switches are connected to the controller and hosts are virtual machines that has 2GB and run Ubuntu 14.04 OS server. We implemented the access control using AspectJ [6], a seamless aspect-oriented extension to Java. AspectJ ensures that all access requests from apps are intercepted by our access control components. This system can be deployed to all other Java-based

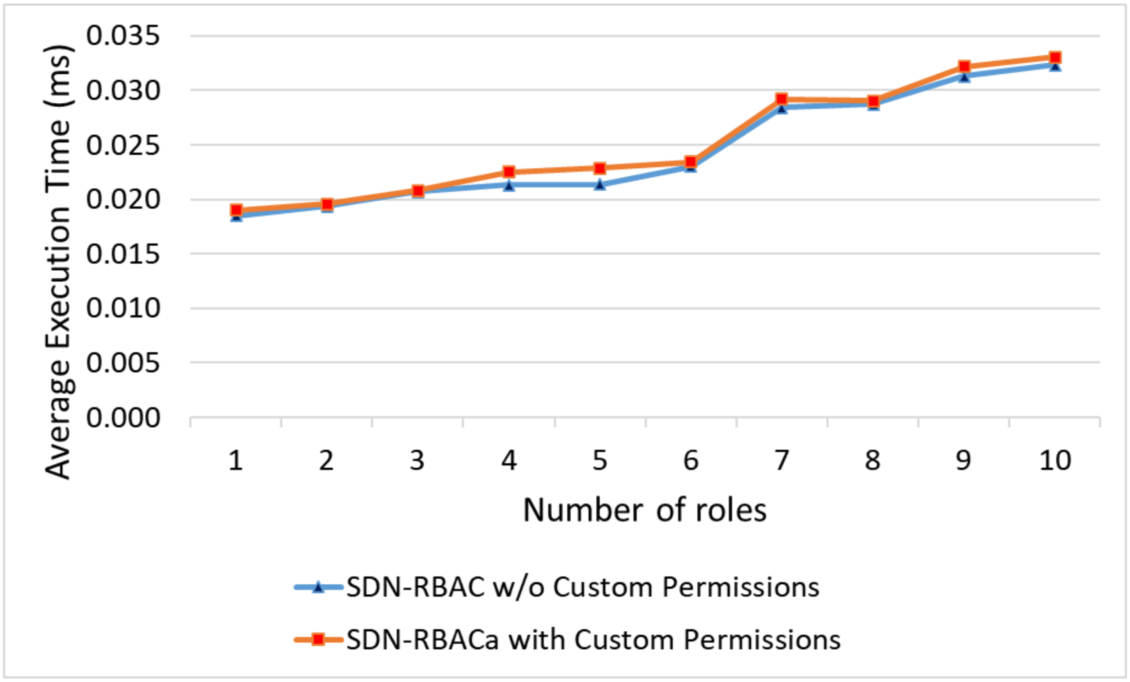


**Figure 5.8**: Average authorization time in SDN-RBAC and SDN-RBACa Operational Model.

SDN controllers.

We created a simple test app, WebTestApp and assigned it to the role ’Web Flow Mod’. Thus, it can access web flow rules only. We designed the app to insert a flow rule with TCP_DST = 25, which is a non-web port. Our refined custom operation addFlow considers ports 80 and 443 as web traffic. The proxy operation addWebFlow only allows ports 80 and 443 to be used for flow rule insertions. The purpose of this test app is to demonstrate how our access control system checks custom permissions and rejects unauthorized access. We created a flow rule and set the TCP_DST match field to 25 using the java instruction: matchbuilder.setExact(MatchField.TCP_DST, TransportPort.of(25));. This causes an access violation since the tcp port number is incorrect. A screenshot of the output console is shown in Fig. 5.7.

## 5.9 Performance Evaluation

To evaluate the effectiveness of our access control system utilizing custom permissions, we created a test app and selected fifty proxy operations, from which we created fifty custom permissions. These custom permissions are assigned to eighteen tasks and ten different roles. We incrementally

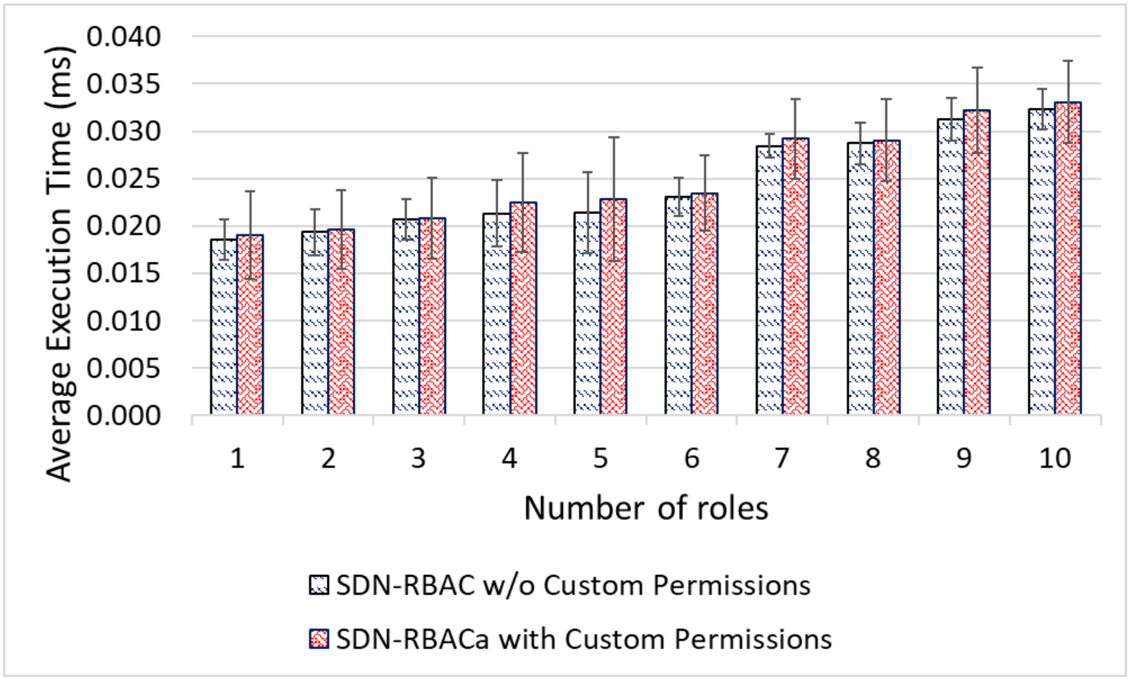


**Figure 5.9**: Average authorization time along with the standard deviation in SDN-RBAC and SDN-RBACa Operational Model.

assigned these roles to the test app which runs in one session. Despite the fact that this app doesn't require all these roles, the purpose of this test is to check the overhead caused by our access control on the system's performance by reporting the execution time with different security policies. We change the security policy by changing the active role set of the app's session. In the first security policy one role is assigned to the session's active role set, in the second policy two roles where assigned, and so on until ten roles.

For each security policy, the session executes all fifty proxy operations. The system is set to compute the authorization delay imposed by the access control components to finish execution and make an access control decision for each proxy operation submitted by the session. The timer starts when the call is intercepted by AspectJ hook, and stops when the access decision is calculated based on the available custom permissions for the session. The total time is calculated for all fifty proxy operations. We repeated this test hundred times for each security policy. For overhead comparison, we performed the same test on SDN-RBAC, but without custom operations. The average elapsed authorization times calculated for SDN-RBACa operational model and the SDN-

RBAC model are reported as shown in Fig. 5.8. For ease of comparing the standard deviation, the results are shown again in a bar chart format in Fig. 5.9. It should be noted here that delay times does not include floodlight's boot-up time, the time for loading the policy and creating the corresponding relations.

This evaluation shows that the authorization check of operational model of SDN-RBACa adds an average of 0.0252 ms overhead on the floodlight controller while SDN-RBAC adds 0.0245 ms on average. This observed latency in both cases is negligible. Fig. 5.9 shows that the standard deviation of authorization check time are generally larger for SDN-RBACa operational model compared to SDN-RBAC model. We believe that this is because using tasks in SDN-RBACa operational model introduces additional variance in the authorization check time. As an overall result, the difference in overhead between the two models is also negligible. Therefore, we believe that the operational model of SDN-RBACa introduces acceptable overhead to the controller for the sake of access control administration.

# CHAPTER 6: CONCLUSION AND FUTURE WORK

This chapter summarizes the contributions of this dissertation and provide some future research directions.

## 6.1 Summary

In this dissertation, we identified fundamental elements of access control in SDN. We showed our steps towards evolving effective access control models for a SDN environment. We started by formalizing an authorization system for SDN proposed in the literature for Floodlight controller. Based on the shortcomings of this model, we proposed a role based access control model for SDN controller applications which we named SDN-RBAC. We implemented SDN-RBAC model with multi-session support in Floodlight controller and used hooking techniques to enforce the security policy without any change to the code of the Floodlight platform. We showed how the implementation verifies the model's usability and effectiveness against unauthorized access requests by controller applications, and showed how the framework can identify and reject unauthorized operations in real time.

Due to the need for a more granular access control for SDN apps and the need to apply the least privilege principle, we enhanced our initial work by proposing ParaSDN, an access control model that provides a fine grained access control using the concept of parameterized roles and permissions. To demonstrate the applicability and feasibility of our proposed model, we configured proof of concept use cases and implemented a prototype in an SDN controller.

We followed this work by proposing an administrative model, referred to as SDN-RBACa. The goal of SDN-RBACa is to manage app-role and task-role relations in access control models for SDN. To enable the administration for SDN, we introduced an approach for creating custom SDN operations to extend the capabilities of SDN controller and provide fine grained custom permissions necessary for the engineering of administrative units. Also, to facilitate access control administration, we extended SDN-RBAC with tasks as a unit of network function and a more convenient abstraction than dealing with permissions. Through proof of concept prototype and use cases, we demonstrated the usability of custom permissions and showed how custom permissions simplify the engineering of roles, tasks, and administrative units; hence, enable and facilitate the administration of access control in SDN.

## 6.2 Future Work

SDN is a relatively new technology that incorporates some promising research directions and open research problems that can be explored. For example access control for SDN-enabled Cloud and SDN-enabled IoT. It is appealing to apply what what we have achieved by researching access control in SDN to investigate and study authorization aspects in more complex SDN-enabled technologies, in particular, SDN-enabled Cloud and SDN-enabled IoT infrastructures. The goal is to design and implement access control models to solve open authorization problems in such hot technologies.

**Access control for SDN with multi-tenancy**. One possible research direction in SDN to investigate open problems related to multi-tenancy with multi-controller setup for SDN-enabled technologies. A common north-bound API and the resulting deployment of various third-party applications poses new security threats in SDN. Malicious applications can leverage such an API and attack the SDN network. These threats become even more evident when network resources are shared among user groups, divisions, or even other companies. Network owner may lease part of the network as network slices, each slice has its own network resources monitorred by a specific controller that can be administered by the leasing tenant. Tenants install multiple OF applications into the network controller to access network resources in their domain. Also, Tenants create users and thus applications run on behalf or created users. One research direction is to investigate open authorization problems for SDN with multi-tenancy, and design, and implement access control models to handle existing problems.

**Risk-Aware Access Control for SDN Apps**. Because SDN is a battle space for traffic and information from different sources. Security and privacy of transmitted information should be kept at the highest level. Network traffic is almost vulnerable to all kinds of network attacks. Thus, analyzing traffic and resource misuse and malicious modification of system configuration and security policy files is very important for building an access control mechanism based on risk assessment. It is promising to explore this issue and build an access control model that relies on factors related to risk calculation and assessment. The eventual goal is to build access control models based on the information gathered from assessment factors and based on the experience gained and lessons learned from implementing such risk aware access control and to help in developing more fine grained access control in SDN-enabled technology.

# APPENDIX A: EXTENDED USE CASE CONFIGURATION

This Appendix shows the complete configuration of SDN-RBACa administrative model for the extended use case described in Section 5.7.2. The formal definition of the administrative model is described in Section 5.5.3.

**Table A.1**: Complete use case configuration of SDN-RBACa for two administrative units - part1.

| **1. Basic Sets** |
|---|
| – APPS = { Web Intrusion Prevention App, Web Application Firewall App, Web Load Balancer App,VoIP Intrusion Prevention App, VoIP Application Firewall App, VoIP Load Balancer App }. |
| – ROLES = { Web Packet-In Handler, Web Packet Monitor, Web Flow Mod, Web Load Balancing, Web Stats Collector, VoIP Packet-In Handler, VoIP Packet Monitor, VoIP Flow Mod, VoIP Load Balancing, VoIP Stats Collector }. |
| – OPS = { readWebPacketInPayload, readWebPacketHeader, readWebFlow, addWebFlow, updateWebFlow, deleteWebFlow,createWebPool, listWebPools, removeWebPool, updateWebPool, createWebMonitor, listWebMonitors, removeWebMonitor, updateWebMonitor, createWebVip, listWebVips, removeWebVip, updateWebVip, createWebMember, listWebMembersByPool, removeWebMember, updateWebMember, readWebFlowByteCount, readAggWebFlowByteCount, readWebFlowPacketCount, readAggWebFlowPacketCount, readVoIPPacketInPayload, readVoIPPacketHeader, readVoIPFlow, addVoIPFlow, updateVoIPFlow, deleteVoIPFlow, createVoIPPool, listVoIPPools, removeVoIPPool, updateVoIPPool, createVoIPMonitor, listVoIPMonitors, removeVoIPMonitor, updateVoIPMonitor, createVoIPVip, listVoIPVips, removeVoIPVip, updateVoIPVip, createVoIPMember, listVoIPMembersByPool, removeVoIPMember, updateVoIPMember, readVoIPFlowByteCount, readAggVoIPFlowByteCount, readVoIPFlowPacketCount, readAggVoIPFlowPacketCount }. |
| – OBTS = { PI-PAYLOAD, PI-HEADER, FLOW-RULE, LB-POOL, LB-MONITOR, LB-VIP, LB-POOL-MEMBER, FLOW-STATS,PI-PAYLOAD, PI-HEADER, FLOW-RULE, LB-POOL, LB-MONITOR, LB-VIP, LB-POOL-MEMBER, FLOW-STATS }. |
| – OBS = set of all objects of types PI-PAYLOAD, PI-HEADER, FLOW-RULE, LB-POOL, LB-MONITOR, LB-VIP, LB-POOL-MEMBER, and FLOW-STATS. |

**Table A.2**: Complete use case configuration of SDN-RBACa for two administrative units - part2.

- PRMS = {
(readWebPacketInPayload, PI-PAYLOAD), (readWebPacketHeader, PI-HEADER),
(readWebFlow, FLOW-RULE), (addWebFlow, FLOW-RULE),
(updateWebFlow, FLOW-RULE), (deleteWebFlow, FLOW-RULE),
(createWebPool, LB-POOL), (listWebPools, LB-POOL),
(removeWebPool, LB-POOL), (updateWebPool, LB-POOL),
(createWebMonitor, LB-MONITOR), (listWebMonitors, LB-MONITOR),
(removeWebMonitor, LB-MONITOR), (updateWebMonitor, LB-MONITOR),
(createWebVip, LB-VIP), (listWebVips, LB-VIP), (removeWebVip, LB-VIP),
(updateWebVip, LB-VIP), (createWebMember, LB-POOL-MEMBER),
(listWebMembersByPool, LB-POOL-MEMBER),
(removeWebMember, LB-POOL-MEMBER), (updateWebMember, LB-POOL-MEMBER),
(readWebFlowByteCount, FLOW-STATS), (readAggWebFlowByteCount, FLOW-STATS),
(readWebFlowPacketCount, FLOW-STATS), (readAggWebFlowPacketCount, FLOW-STATS),
(readVoIPPacketInPayload, PI-PAYLOAD), (readVoIPPacketHeader, PI-HEADER),
(readVoIPFlow, FLOW-RULE), (addVoIPFlow, FLOW-RULE), (updateVoIPFlow, FLOW-RULE),
(deleteVoIPFlow, FLOW-RULE), (createVoIPPool, LB-POOL), (listVoIPPools, LB-POOL),
(removeVoIPPool, LB-POOL), (updateVoIPPool, LB-POOL), (createVoIPMonitor, LB-MONITOR),
(listVoIPMonitors, LB-MONITOR), (removeVoIPMonitor, LB-MONITOR),
(updateVoIPMonitor, LB-MONITOR), (createVoIPVip, LB-VIP), (listVoIPVips, LB-VIP),
(removeVoIPVip, LB-VIP), (updateVoIPVip, LB-VIP),
(createVoIPMember, LB-POOL-MEMBER), (listVoIPMembersByPool, LB-POOL-MEMBER),
(removeVoIPMember, LB-POOL-MEMBER), (updateVoIPMember, LB-POOL-MEMBER),
(readVoIPFlowByteCount, FLOW-STATS), (readAggVoIPFlowByteCount, FLOW-STATS),
(readVoIPFlowPacketCount, FLOW-STATS), (readAggVoIPFlowPacketCount, FLOW-STATS)
}.
- AP = {
Web Load Balance Pool, Web Security Pool, VoIP Load Balance Pool, VoIP Security Pool
}.
- TASKS = {
Web Deep Packet Inspection Task, Web Packet Header Inspection Task,
Web Flow Viewing Task, Web Traffic Forwarding Task,
Web Server Pool Management Task, Web Server Monitor Management Task,
Web Pool VIP Management Task, Web Pool Member Management Task,
Web Payload Statistics Collection Task, Web Packet Statistics Collection Task,
VoIP Deep Packet Inspection Task, VoIP Packet Header Inspection Task,
VoIP Flow Viewing Task, VoIP Traffic Forwarding Task, VoIP Server Pool
Management Task, VoIP Server Monitor Management Task,
VoIP Pool VIP Management Task, VoIP Pool Member Management Task,
VoIP Payload Statistics Collection Task, VoIP Packet Statistics Collection Task
}.
- USERS = {
web_functions_admin_user, web_apps_admin_user,
voip_functions_admin_user, voip_apps_admin_user}.
- AU = {Web Admin Unit, VoIP Admin Unit}.

**Table A.3**: Complete use case configuration of SDN-RBACa for two administrative units - part3.

| | **2. Assignment Relations:** |
|---|---|
| – | PA = {<br>{(readWebPacketInPayload, PI-PAYLOAD), (readWebPacketHeader, PI-HEADER),<br>(readWebFlow, FLOW-RULE)} × {Web Deep Packet Inspection Task} ∪<br>{(readWebPacketHeader, PI-HEADER),<br>(readWebFlow, FLOW-RULE)} × {Web Packet Header Inspection Task} ∪<br>{(readWebFlow, FLOW-RULE)} × {Web Flow Viewing Task} ∪<br>{(addWebFlow, FLOW-RULE), (updateWebFlow, FLOW-RULE),<br>(deleteWebFlow, FLOW-RULE)} × {Web Traffic Forwarding Task} ∪<br>{(createWebPool, LB-POOL), (listWebPools, LB-POOL), (removeWebPool, LB-POOL),<br>(updateWebPool, LB-POOL)} × {Web Server Pool Management Task} ∪<br>{(createWebMonitor, LB-MONITOR), (listWebMonitors, LB-MONITOR),<br>(removeWebMonitor, LB-MONITOR), (updateWebMonitor, LB-MONITOR)} ×<br>{Web Server Monitor Management Task} ∪<br>{(createWebVip, LB-VIP), (listWebVips, LB-VIP), (removeWebVip, LB-VIP),<br>(updateWebVip, LB-VIP)} × {Web Pool VIP Management Task} ∪<br>{(createWebMember, LB-POOL-MEMBER), (listWebMembersByPool, LB-POOL-MEMBER),<br>(removeWebMember, LB-POOL-MEMBER), (updateWebMember, LB-POOL-MEMBER)} ×<br>{Web Pool Member Management Task} ∪<br>{(readWebFlowByteCount, FLOW-STATS), (readAggWebFlowByteCount, FLOW-STATS)} ×<br>{Web Payload Statistics Collection Task} ∪<br>{(readWebFlowPacketCount, FLOW-STATS), (readAggWebFlowPacketCount, FLOW-STATS)} ×<br>{Web Packet Statistics Collection Task}, (readVoIPPacketInPayload, PI-PAYLOAD),<br>(readVoIPPacketHeader, PI-HEADER), (readVoIPFlow, FLOW-RULE)} ×<br>{VoIP Deep Packet Inspection Task} ∪<br>{(readVoIPPacketHeader, PI-HEADER), (readVoIPFlow, FLOW-RULE)} ×<br>{VoIP Packet Header Inspection Task} ∪<br>{(readVoIPFlow, FLOW-RULE)} × {VoIP Flow Viewing Task} ∪<br>{(addVoIPFlow, FLOW-RULE), (updateVoIPFlow, FLOW-RULE),<br>(deleteVoIPFlow, FLOW-RULE)} × {VoIP Traffic Forwarding Task} ∪<br>{(createVoIPPool, LB-POOL), (listVoIPPools, LB-POOL), (removeVoIPPool, LB-POOL),<br>(updateVoIPPool, LB-POOL)} × {VoIP Server Pool Management Task} ∪<br>{(createVoIPMonitor, LB-MONITOR), (listVoIPMonitors, LB-MONITOR),<br>(removeVoIPMonitor, LB-MONITOR), (updateVoIPMonitor, LB-MONITOR)} ×<br>{VoIP Server Monitor Management Task} ∪<br>{(createVoIPVip, LB-VIP), (listVoIPVips, LB-VIP), (removeVoIPVip, LB-VIP),<br>(updateVoIPVip, LB-VIP)} × {VoIP Pool VIP Management Task} ∪<br>{(createVoIPMember, LB-POOL-MEMBER), (listVoIPMembersByPool, LB-POOL-MEMBER),<br>(removeVoIPMember, LB-POOL-MEMBER), (updateVoIPMember, LB-POOL-MEMBER)} ×<br>{VoIP Pool Member Management Task} ∪<br>{(readVoIPFlowByteCount, FLOW-STATS), (readAggVoIPFlowByteCount, FLOW-STATS)} ×<br>{VoIP Payload Statistics Collection Task} ∪<br>{(readVoIPFlowPacketCount, FLOW-STATS), (readAggVoIPFlowPacketCount, FLOW-STATS)} ×<br>{VoIP Packet Statistics Collection Task}<br>}. |

**Table A.4**: Complete use case configuration of SDN-RBACa for two administrative units - part4.

- TA = {
  {Web Deep Packet Inspection Task, Web Packet Header Inspection Task} $\times$
  {Web Packet-In Handler} $\cup$
  {Web Packet Header Inspection Task} $\times$ {Web Packet Monitor} $\cup$
  {Web Flow Viewing Task, Web Traffic Forwarding Task} $\times$ {Web Flow Mod} $\cup$
  {Web Server Pool Management Task, Web Server Monitor Management Task,
  Web Pool VIP Management Task, Web Pool Member Management Task} $\times$
  {Web Load Balancing} $\cup$
  {Web Payload Statistics Collection Task, Web Packet Statistics Collection Task} $\times$
  {Web Stats Collector},
  {VoIP Deep Packet Inspection Task, VoIP Packet Header Inspection Task} $\times$
  {VoIP Packet-In Handler} $\cup$
  {VoIP Packet Header Inspection Task} $\times$ {VoIP Packet Monitor} $\cup$
  {VoIP Flow Viewing Task, VoIP Traffic Forwarding Task} $\times$ {VoIP Flow Mod} $\cup$
  {VoIP Server Pool Management Task, VoIP Server Monitor Management Task,
  VoIP Pool VIP Management Task, VoIP Pool Member Management Task} $\times$
  {VoIP Load Balancing} $\cup$
  {VoIP Payload Statistics Collection Task, VoIP Packet Statistics Collection Task} $\times$
  {VoIP Stats Collector}
  }.
- AA = {
  {Web Intrusion Prevention App} $\times$ {Web Packet-In Handler, Web Flow Mod} $\cup$
  {Web Application Firewall App} $\times$ {Web Packet Monitor, Web Flow Mod} $\cup$
  {Web Load Balancer App} $\times$ {Web Flow Mod, Web Load Balancing, Web Stats Collector},
  {VoIP Intrusion Prevention App} $\times$ {VoIP Packet-In Handler, VoIP Flow Mod} $\cup$
  {VoIP Application Firewall App} $\times$ {VoIP Packet Monitor, VoIP Flow Mod} $\cup$
  {VoIP Load Balancer App} $\times$ {VoIP Flow Mod, VoIP Load Balancing, VoIP Stats Collector}
  }.
- OT = {
  (all payloads in packet-in message, PI-PAYLOAD), (all packet header objects, PI-HEADER),
  (all flow-rules, FLOW-RULE), (all server pools, LB-POOL), (all server monitors, LB-MONITOR),
  (all pools virtual IPs, LB-VIP), (all pool members, LB-POOL-MEMBER),
  (all flow statistics in flow rules, FLOW-STATS), (all payloads in packet-in message, PI-PAYLOAD),
  (all packet header objects, PI-HEADER), (all flow-rules, FLOW-RULE), (all server pools, LB-POOL),
  (all server monitors, LB-MONITOR), (all pools virtual IPs, LB-VIP),
  (all pool members, LB-POOL-MEMBER), (all flow statistics in flow rules, FLOW-STATS)
  }.

**Table A.5**: Complete use case configuration of SDN-RBACa for two administrative units - part5.

| **3. App-pools Relation:** |
|---|
| – AAPA = {<br>(Web Intrusion Prevention App, Web Security Pool),<br>(Web Application Firewall App, Web Security Pool),<br>(Web Load Balancer App, Web Load Balance Pool),<br>(VoIP Intrusion Prevention App, VoIP Security Pool),<br>(VoIP Application Firewall App, VoIP Security Pool),<br>(VoIP Load Balancer App, VoIP Load Balance Pool)<br>}. |
| **4. AU and Partitioned Assignment:** |
| – roles(Web Admin Unit) = {<br>Web Packet-In Handler, Web Packet Monitor, Web Flow Mod, Web Load Balancing, Web Stats Collector<br>}.<br>– tasks(Web Admin Unit) = {<br>Web Deep Packet Inspection Task, Web Packet Header Inspection Task,<br>Web Flow Viewing Task, Web Traffic Forwarding Task,<br>Web Server Pool Management Task, Web Server Monitor Management Task,<br>Web Pool VIP Management Task, Web Pool Member Management Task,<br>Web Payload Statistics Collection Task,<br>Web Packet Statistics Collection Task<br>}.<br>– app_pools(Web Admin Unit) = {Web Load Balance Pool, Web Security Pool}.<br>– roles(VoIP Admin Unit) = {<br>VoIP Packet-In Handler, VoIP Packet Monitor,<br>VoIP Flow Mod, VoIP Load Balancing, VoIP Stats Collector<br>}.<br>– tasks(VoIP Admin Unit) = {<br>VoIP Deep Packet Inspection Task, VoIP Packet Header Inspection Task,<br>VoIP Flow Viewing Task, VoIP Traffic Forwarding Task,<br>VoIP Server Pool Management Task, VoIP Server Monitor Management Task,<br>VoIP Pool VIP Management Task, VoIP Pool Member Management Task,<br>VoIP Payload Statistics Collection Task, VoIP Packet Statistics Collection Task<br>}.<br>– app_pools(VoIP Admin Unit) = {VoIP Load Balance Pool, VoIP Security Pool}. |
| **5. Administrative User Assignment:** |
| – TA_admin = {<br>(web_functions_admin_user, Web Admin Unit),<br>(voip_functions_admin_user, VoIP Admin Unit)}.<br>– AA_admin = {<br>(web_apps_admin_user, Web Admin Unit),<br>(voip_apps_admin_user, VoIP Admin Unit)}. |

# BIBLIOGRAPHY

[1] Ali E Abdallah and Etienne J Khayat. A formal model for parameterized role-based access control. In *IFIP World Computer Congress, TC 1*, pages 233–246. Springer, 2004.

[2] Ijaz Ahmad, Suneth Namal, Mika Ylianttila, and Andrei Gurtov. Security in software defined networks: A survey. *IEEE Communications Surveys & Tutorials*, 17(4):2317–2346, 2015.

[3] Abdullah Al-Alaj, Ram Krishnan, and Ravi Sandhu. Sdn-rbac: An access control model for sdn controller applications. In *2019 4th International Conference on Computing, Communications and Security (ICCCS)*, pages 1–8. IEEE, 2019.

[4] Abdullah Al-Alaj, Ravi Sandhu, and Ram Krishnan. A formal access control model for se-floodlight controller. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 1–6. ACM, 2019.

[5] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92, 2010.

[6] AspectJ. Aspectj: A seamless aspect oriented extension to java, 2020. `https://www.eclipse.org/aspectj/`.

[7] Siamak Azodolmolky, Philipp Wieder, and Ramin Yahyapour. Sdn-based cloud computing networking. In *2013 15th International Conference on Transparent Optical Networks (ICTON)*, pages 1–4. IEEE, 2013.

[8] Jeffrey R Ballard, Ian Rae, and Aditya Akella. Extensible and scalable network monitoring using opensafe. *Inm/wren*, 10, 2010.

[9] Christian Banse and Sathyanarayanan Rangarajan. A secure northbound interface for sdn applications. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 834–839. IEEE, 2015.

[10] Khalid Zaman Bijon, Ram Krishnan, and Ravi Sandhu. Risk-aware rbac sessions. In *International Conference on Information Systems Security*, pages 59–74. Springer, 2012.

[11] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. Uni-arbac: A unified administrative model for role-based access control. In *International Conference on Information Security*, pages 218–230. Springer, 2016.

[12] Nikos Bizanis and Fernando A Kuipers. Sdn and virtualization solutions for the internet of things: A survey. *IEEE Access*, 4:5591–5606, 2016.

[13] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4):1270–1283, 2009.

[14] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12, 2007.

[15] Open SDN controller, ,. `http://floodlight.openflowhub.org/`.

[16] Jason Crampton. Understanding and developing role-based administrative models. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 158–167, 2005.

[17] Jason Crampton and George Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and System Security (TISSEC)*, 6(2):201–231, 2003.

[18] Scott-Hayward et al. Sdn security: A survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*, pages 1–7. IEEE, 2013.

[19] Scott-Hayward et al. A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials*, 18(1):623–654, 2016.

[20] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[21] Security Enhanced Floodlight, 2020. `https://www.sdxcentral.com/projects/openflow-sec-security-enhanced-floodlight/`.

[22] Floodlight-Project, 2020. `http://www.projectfloodlight.org/`.

[23] Open Networking Foundation, 2020.

[24] Ryu SDN Framework, 2020. `http://osrg.github.io/ryu/`.

[25] Mei Ge and Sylvia L Osborn. A design for parameterized roles. In *Research Directions in Data and Applications Security XVIII*, pages 251–264. Springer, 2004.

[26] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the second ACM workshop on Role-based access control*, pages 153–159, 1997.

[27] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. {ASM}: A programmable interface for extending android security. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 1005–1019, 2014.

[28] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, 800(162), 2013.

[29] Raj Jain and Subharthi Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[30] Sushant Jain, , et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[31] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, 2013.

[32] Diego Kreutz et al. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 55–60. ACM, 2013.

[33] Ninghui Li and Ziqing Mao. Administration in role-based access control. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 127–138, 2007.

[34] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[35] Python network controller. 2018. `http://www.noxrepo.org/pox/aboutpox/`.

[36] Jiseong Noh, Seunghyeon Lee, Jaehyun Park, Seungwon Shin, and Brent Byunghoon Kang. Vulnerabilities of network os and mitigation with state-based permission system. *Security and Communication Networks*, 9(13):1971–1982, 2016.

[37] Sejong Oh and Ravi Sandhu. A model for role administration using organization structure. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 155–162, 2002.

[38] Mike Ojo, Davide Adami, and Stefano Giordano. A sdn-iot architecture with nfv implementation. In *2016 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6. IEEE, 2016.

[39] Hitesh Padekar, Younghee Park, Hongxin Hu, and Sang-Yoon Chang. Enabling dynamic access control for controller applications in software-defined networks. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, pages 51–61, 2016.

[40] The OpenDaylight platform, 2020. `https://www.opendaylight.org/`.

[41] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 121–126, 2012.

[42] Phillip A Porras et al. Securing the software defined network control layer. In *NDSS*, 2015.

[43] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The arbac97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

[44] Ravi Sandhu and Qamar Munawer. The arbac99 model for administration of roles. In *Proceedings 15th Annual Computer Security Applications Conference (ACSAC'99)*, pages 229–238. IEEE, 1999.

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

[46] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[47] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[48] Sandra Scott-Hayward, Christopher Kane, and Sakir Sezer. Operationcheckpoint: Sdn application control. In *2014 IEEE 22nd International Conference on Network Protocols*, pages 618–623. IEEE, 2014.

[49] Yuchia Tseng, Montida Pattaranantakul, Ruan He, Zonghua Zhang, and Farid Naït-Abdesselam. Controller dac: Securing sdn controller with dynamic access control. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[50] Benjamin E Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-

app poisoning in software-defined networking. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 648–663, 2018.

[51] Steven J Vaughan-Nichols. Openflow: The next generation of the network? *Computer*, (8):13–15, 2011.

[52] He Wang and Sylvia L Osborn. An administrative model for role graphs. In *Data and Applications Security XVII*, pages 302–315. Springer, 2004.

[53] Richard Wang, Dana Butnariu, Jennifer Rexford, et al. Openflow-based server load balancing gone wild. *Hot-ICE*, 11:12–12, 2011.

[54] Xitao Wen, Yan Chen, Chengchen Hu, Chao Shi, and Yi Wang. Towards a secure controller platform for openflow applications. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 171–172, 2013.

[55] Changhoon Yoon, Taejune Park, Seungsoo Lee, Heedo Kang, Seungwon Shin, and Zonghua Zhang. Enabling security functions with sdn: A feasibility study. *Computer Networks*, 85:19–35, 2015.

[56] Changhoon Yoon, Seungwon Shin, Phillip Porras, Vinod Yegneswaran, Heedo Kang, Martin Fong, Brian O'Connor, and Thomas Vachuska. A security-mode for carrier-grade sdn controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 461–473, 2017.

# VITA

Abdullah Al-Alaj (abdullah.al-alaj@utsa.edu) was born in Jordan. After completing his High School at Irbid in June 2000, Abdullah joined Jordan University of Science and Technology (JUST) in Irbid, Jordan. He received his M.Sc. and B.Sc. degrees from the Department of Computer Science at JUST in 2004 and 2006. After completing the M.Sc. degree, he worked as an instructor in the Department of Computer Science at JUST. In Fall 2016, he joined the Department of Computer Science at the University of Texas at San Antonio (UTSA) to pursue his doctoral degree. He joined the Institute for Cyber Security (ICS) at UTSA and started working with Prof. Ravi Sandhu and Dr. Ram Krishnan since 2017. His research interests include designing and implementing access control models to enhance the security of software defined networking.