



Pós-Graduação em Ciência da Computação

DANIEL ROSENDO

**A HIGH-LEVEL AUTHORIZATION FRAMEWORK FOR
SOFTWARE-DEFINED NETWORKS**



Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2017

DANIEL ROSENDO

**A HIGH-LEVEL AUTHORIZATION FRAMEWORK FOR
SOFTWARE-DEFINED NETWORKS**

*A M.Sc. Dissertation presented to the Center for
Informatics of Federal University of Pernambuco in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.*

Advisor: *Prof^a Dr^a Judith Kelner*

Co-Advisor: *Prof^a Dr^a Patricia Takako Endo*

RECIFE

2017

Catálogo na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

R813h Rosendo, Daniel
A high-level authorization framework for software-defined networks / Daniel Rosendo – 2017.
77 f.: il., fig., tab.

Orientadora: Judith Kelner.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências.

1. Redes de computadores. 2. Internet das coisas. I. Kelner, Judith (orientadora). II. Título.

004.6 CDD (23. ed.) UFPE- MEI 2017-111

DANIEL ROSENDO

**A HIGH-LEVEL AUTHORIZATION FRAMEWORK FOR
SOFTWARE-DEFINED NETWORKS**

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Master of Science em Ciência da Computação.

Aprovado em: 14/03/2017.

BANCA EXAMINADORA

Prof. Dr. Nelson Souto Rosa
Centro de Informática / UFPE

Profa. Dra. Rossana Maria de Castro Andrade
Departamento de Computação/UFC

Profa. Dra. Judith Kelner
Centro de Informática / UFPE
(Orientadora)

*I dedicate this thesis to all my family, friends and professors
who gave me the necessary support to get here.*

Acknowledgements

I would first like to thank my grandmother (Maria), my parents (Djalma and Luzineide), and my brothers (Diomedes and Diogo) for all support they gave me during these two years of hard work. Many thanks to them for cheering me up in some moments. Likewise, I would like to thank all my family members and friends (Danilo and Eduardo).

Next, I would like to thank my advisor Dra. Judith Kelner and co-advisor Dra. Patricia Endo for their patience and believing and guiding me during my research. I feel extremely pleased in thanking them.

I would also like to thank all researchers at the Networking and Telecommunications Research Group (GPRT) for sharing their knowledge, offering relevant feedbacks, giving me new ideas and suggestions to improve this work, and encouraging me (in special: Djamel, Rafael, Marcos, Alexandre, Ernani, Rodrigo, Greg, Thiago, Wesley, Glauco, Moisés, Demis, Késsia, and Cani). Without their support, I would have never finished my research.

Next, I would like to thank external researchers (Fabien Autrel, Nate Foster, Steffen Smolka, Adrian Lara, and Robert Soulé) for their attention and patience for answering my questions via e-mails. Many thanks to them for contributing and helping me to improve this work.

Finally, I would like to thank the Fundação de Amparo a Ciência e Tecnologia de Pernambuco (FACEPE) for funding this work through grant IBPG-0745-1.03/14.

*Try not to become a man of success,
but rather try to become a man of value.*

—ALBERT EINSTEIN

Abstract

Network Access Control (NAC) management is a critical task. Misconfigurations may result in vulnerabilities that may compromise the overall network security. Traditional access control setups rely on firewalls, IEEE 802.1x, VLAN, ACL, and LDAP. These approaches work well for stable and small networks and are hard to integrate and configure. Besides, they are inflexible and require per-device and vendor-specific configurations, being error-prone. The Software-Defined Networking (SDN) paradigm overcomes architectural problems of traditional networks, simplifies the network design and operation, and offers new opportunities (programmability, flexibility, dynamicity, and standardization) to manage these issues. Furthermore, SDN reduces the human intervention, which in turn also reduce operational costs and misconfigurations. Despite this, access control management remains a challenge, once managing security policies involves dealing with a large set of access control rules; detection of conflicting policies; defining priorities; delegating rights; reacting to dynamic network states and events. This dissertation explores the use of SDN to mitigate these problems. We present HACFlow, a novel SDN framework for network access control management based on the OrBAC model. HACFlow aims to simplify and automate the NAC management. It allows network operators to govern rights of network entities by defining dynamic, fine-grained, and high-level access control policies. To illustrate the operation of HACFlow we present through a step by step how the main management tasks are executed. Our study case is a Smart City network environment. We conducted many experiments to analyze the scalability and performance of HACFlow, and the results show that it requires a time in the order of milliseconds to execute all the management tasks, even managing many policies. Besides, we compare HACFlow against related approaches.

Keywords: Software-defined Networks. Internet of Things. Security management. Policy-based management. Autonomic and cognitive management.

Resumo

Gerenciar o controle de acesso entre recursos (usuários, máquinas, serviços, etc.) em uma rede é uma tarefa crítica. Erros de configuração podem resultar em vulnerabilidades que podem comprometer a segurança da rede como um todo. Em redes tradicionais, esse controle de acesso é implementado através de *firewalls*, IEEE 802.1x, VLAN, ACL, and LDAP. Estas abordagens funcionam bem em redes menores e estáveis, e são difíceis de configurar e integrar. Além disso, são inflexíveis e requerem configurações individuais e específicas de cada fabricante, sendo propensa à erros. O paradigma de Redes Definidas por Software (SDN) supera os problemas arquiteturais das redes tradicionais, simplifica o projeto e operação da rede, e proporciona novas oportunidades (programabilidade, flexibilidade, dinamicidade, e padronização) para lidar com os problemas enfrentados em redes tradicionais. Apesar das vantagens do SDN, o gerenciamento de políticas de controle de acesso na rede continua sendo uma tarefa difícil. Uma vez que, gerenciar tais políticas envolve lidar com uma grande quantidade de regras; detectar e resolver conflitos; definir prioridades; delegar papéis; e adaptar tais regras de acordo com eventos e mudanças de estado da rede. Esta dissertação explora o paradigma SDN a fim de mitigar tais problemas. Neste trabalho, apresentamos o HACFlow, um *framework* SDN para gerenciamento de políticas de controle de acesso na rede baseado no modelo OrBAC. HACFlow tem como principal objetivo simplificar e automatizar tal gerenciamento. HACFlow permite que operadores da rede governe os privilégios das entidades da rede através da definição de políticas de controle de acesso dinâmicas, em alto nível, e com alta granularidade. Para ilustrar o funcionamento do HACFlow apresentamos um passo a passo de como as principais tarefas de gerenciamento de controle de acesso são realizadas. Nosso estudo de caso é um ambiente de rede de uma cidade inteligente. Vários experimentos foram realizados a fim de analisar a escalabilidade e performance do HACFlow. Os resultados mostram que o HACFlow requer um tempo na ordem de milissegundos para executar cada uma das tarefas de gerenciamento, mesmo lidando com uma grande quantidade de regras. Além disso, nós comparamos HACFlow com propostas relacionadas existentes na literatura.

Palavras-chave: Redes Definidas por Software. Internet das Coisas. Gerenciamento de segurança. Gerenciamento baseado em políticas. Gerenciamento autônomo e cognitivo.

List of Figures

| | | |
|------|---|----|
| 2.1 | Components of an SDN Architecture. Adapted from: ONF (2014) | 21 |
| 2.2 | OpenFlow structure. Adapted from: yuba.stanford.edu | 23 |
| 2.3 | OpenFlow match field constraints. Source: flowgrammable.org | 23 |
| 2.4 | Access control models. | 25 |
| 2.5 | The abstract and concrete levels of the OrBAC model. Adapted: orbac.org | 25 |
| 3.1 | Network access control using a firewall. | 29 |
| 3.2 | Network access control using PNAC. | 29 |
| 3.3 | Network access control using VLAN. | 30 |
| 3.4 | Network access control using LDAP. | 31 |
| 3.5 | Combining different solutions of traditional network to control the access of network entities. | 31 |
| 3.6 | SDN-based solutions. | 32 |
| 4.1 | HACFlow Framework Architecture. | 38 |
| 4.2 | HACFlow skeleton classes. | 39 |
| 4.3 | How HACFlow translates an OrBAC policy into OpenFlow. | 41 |
| 4.4 | High-level policy definition in HACFlow. | 43 |
| 4.5 | HACFlow policy granularity and expressiveness. | 44 |
| 4.6 | <i>BeanShell</i> and <i>Temporal</i> contexts. | 45 |
| 4.7 | <i>Prova</i> context definition for vulnerability alert. | 46 |
| 4.8 | HACFlow reaction against a vulnerability alert. | 47 |
| 4.9 | Reaction against a user authentication. | 48 |
| 4.10 | Smart City case study scenario. | 50 |
| 4.11 | Creating the <i>Organization</i> predicate in MotOrBAC tool. | 51 |
| 4.12 | Creating a <i>context</i> and setting its <i>definition</i> | 52 |
| 4.13 | Creating the <i>garbageA</i> entity and assigning its <i>class definition</i> | 52 |
| 4.14 | Attribute values of the <i>garbageA</i> entity. | 52 |
| 4.15 | Attribute values of two different services within the <i>cameraA</i> entity. | 53 |
| 4.16 | Creating the <i>PolicyA</i> security rule. | 54 |
| 5.1 | Experimental setup. | 57 |
| 5.2 | Steps to react to an authentication. | 58 |
| 5.3 | Steps to react to a vulnerability alert. | 59 |
| 5.4 | Steps to react to a dynamic policy. | 60 |
| 5.5 | Steps to delegate a role. | 61 |

| | | |
|-----|--|----|
| 5.6 | Scalability: high-level to low-level policy inference. | 63 |
| 5.7 | Policy inference steps: security rule filter and policy translation. | 63 |
| 5.8 | Frenetic, FRESCO, OpenSec, and HACFlow syntax comparison to create a policy. | 67 |
| 5.9 | CPU performance comparison provided by CPUBoss. | 68 |

List of Tables

| | | |
|------|--|----|
| 1.1 | Research Challenges AHMAD et al. (2015) ; WICKBOLDT et al. (2015) and ONF Requirements and Problems to be Solved TR-516 (2015) | 18 |
| 4.1 | HACFlow Framework Scope. | 37 |
| 4.2 | HACFlow REST API resources. | 42 |
| 5.1 | Network Entity Authentication. | 58 |
| 5.2 | HACFlow process: authentication event. | 58 |
| 5.3 | Network Vulnerability Alert. | 59 |
| 5.4 | HACFlow process: vulnerability alert. | 59 |
| 5.5 | Dynamic Security Policy. | 61 |
| 5.6 | HACFlow process: dynamic policy. | 61 |
| 5.7 | Role delegation. | 62 |
| 5.8 | HACFlow process: role delegation. | 62 |
| 5.9 | High-level to low-level policy inference. | 63 |
| 5.10 | Summary of the comparison analysis. | 65 |
| 5.11 | Features Implemented by Different SDN-based Network Access Control (NAC) Solutions. | 66 |
| 5.12 | High-level to low-level policy translation for a single rule. | 69 |
| 5.13 | Required time to HACFlow and OpenSec react to a networking event. | 70 |

List of Acronyms

| | | |
|--------------|---|----|
| AAA | Authentication, Authorization, and Accounting | 20 |
| ABAC | Attribute-Based Access Control | 24 |
| ACL | Access Control List | 16 |
| AD | Active Directory | 30 |
| CBAC | Coalition Based Access Control | 24 |
| DAC | Discretionary access control | 24 |
| DDoS | Distributed Denial of Service | 46 |
| DPI | Deep Packet Inspection | 46 |
| FRP | Functional Reactive Programming | 32 |
| ISP | Internet Service Provide | 24 |
| LDAP | Lightweight Directory Access Protocol | 16 |
| MAC | Mandatory Access Control | 24 |
| NAC | Network Access Control | 16 |
| NAS | Network Access Server | 28 |
| ONF | Open Networking Foundation | 18 |
| OrBAC | Organization Based Access Control | 25 |
| PNAC | Port-based Network Access Control | 16 |
| RBAC | Role-Based Access Control | 24 |
| REST | Representative State Transfer | 20 |
| SDN | Software-Defined Networking | 16 |
| SEK | Security Enforcement Kernel | 34 |
| TCAM | Ternary Content-Addressable Memory | 40 |
| TCP | Transmission Control Protocol | 22 |
| TLS | Transport Layer Security | 22 |
| TMAC | Team-based Access Control | 24 |
| VLAN | Virtual Local Area Network | 16 |
| XACML | eXtensible Access Control Markup Language | 30 |

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 16 |
| 1.1 | Motivation | 16 |
| 1.2 | Problem Statement | 17 |
| 1.3 | General and Specific Goals | 19 |
| 1.4 | Organization of the Dissertation | 19 |
| 2 | Background | 20 |
| 2.1 | Software Defined Networks | 20 |
| 2.2 | OpenFlow Standard | 22 |
| 2.3 | Authentication, Authorization and Accounting | 24 |
| 2.4 | Access Control Models | 24 |
| 2.4.1 | Organization Based Access Control | 25 |
| 2.4.1.1 | Organization | 25 |
| 2.4.1.2 | Role and Subject | 26 |
| 2.4.1.3 | Activity and Action | 26 |
| 2.4.1.4 | View and Object | 26 |
| 2.4.1.5 | Context Definition | 26 |
| 2.4.1.6 | Class Definition | 27 |
| 2.4.1.7 | High-level Security Policy Definition | 27 |
| 2.5 | Concluding Remarks | 27 |
| 3 | Related Work | 28 |
| 3.1 | Network Access Control in Traditional Networks | 28 |
| 3.2 | Network Access Control in SDN | 32 |
| 3.3 | Candidates to Comparison Against HACFlow | 33 |
| 3.4 | Concluding Remarks | 35 |
| 4 | HACFlow | 36 |
| 4.1 | HACFlow Framework | 36 |
| 4.1.1 | Overview | 36 |
| 4.1.2 | Architecture | 37 |
| 4.1.2.1 | OrBAC API | 38 |
| 4.1.2.2 | Policy Skeleton | 39 |
| 4.1.2.3 | Entity Manager | 39 |
| 4.1.2.4 | Event Listener | 40 |
| 4.1.2.5 | Policy Translator | 40 |

| | | |
|----------|--|-----------|
| 4.1.2.6 | REST API | 41 |
| 4.1.3 | Step-by-step: High-level Policy Definition | 41 |
| 4.1.3.1 | Security Policy Expressiveness and Granularity | 44 |
| 4.1.4 | Step-by-step: Dynamic Security Policies | 44 |
| 4.1.5 | Step-by-step: Reacting to Network Events | 45 |
| 4.1.5.1 | Vulnerability Alert | 46 |
| 4.1.5.2 | Authentication Event | 47 |
| 4.1.6 | Step-by-step: Role Delegation | 47 |
| 4.2 | Case Study: Applying HACFlow in a Smart City | 48 |
| 4.2.1 | Overview | 49 |
| 4.2.2 | Defining High-level Goals | 49 |
| 4.2.3 | Defining the Network Entities | 50 |
| 4.2.3.1 | The Abstract Level | 51 |
| 4.2.3.2 | The Concrete Level | 51 |
| 4.2.4 | Defining High-level Security Policies | 53 |
| 4.2.5 | Enforcing Security Policies in the Network | 54 |
| 4.3 | Concluding Remarks | 55 |
| 5 | Evaluation and Comparison | 56 |
| 5.1 | HACFlow Performance Evaluation | 56 |
| 5.1.1 | Scenario Description and Methodology | 56 |
| 5.1.2 | Network State Changes and Events | 57 |
| 5.1.2.1 | Authentication Event | 58 |
| 5.1.2.2 | Vulnerability Alert | 59 |
| 5.1.3 | Dynamic Security Policy | 60 |
| 5.1.4 | Role Delegation | 61 |
| 5.1.5 | High-level to Low-level Policy Inference | 62 |
| 5.1.6 | Discussion | 64 |
| 5.2 | Comparison Against Existing Solutions | 64 |
| 5.2.1 | Overview | 64 |
| 5.2.2 | Qualitative Analysis | 65 |
| 5.2.2.1 | Framework Features | 65 |
| 5.2.2.2 | Syntax Simplicity | 66 |
| 5.2.3 | Quantitative Analysis | 68 |
| 5.2.3.1 | Policy Translation | 69 |
| 5.2.3.2 | Event Reaction Delay | 69 |
| 5.2.4 | Discussion | 70 |
| 6 | Conclusion | 71 |
| 6.1 | Difficulties Found | 72 |

| | | |
|-----|---|-----------|
| 6.2 | Future Work and Open Challenges | 73 |
| 6.3 | Statement of the Contributions | 74 |
| | References | 75 |

1

Introduction

During the last decade, advances in the Internet architecture and communication system technologies together with the introduction of the Software-Defined Networking (SDN) and Internet of Things (IoT) paradigms contributed to the growth of the network and the number of interconnected heterogeneous devices. These devices exchange information and interact with each other and with humans and machines. Ensuring the security and privacy of these entities and defining and managing access rights to protect them from unauthorized access become a challenge [SICARI et al. \(2015\)](#).

In this scenario, Network Access Control (NAC) management is a critical task. There are several devices in the network, each one with different features; and at the same time, there are also many users with different levels of access rights to these devices. For instance, in a Smart City, we may have many garbage can with embedded sensors spread in a neighborhood and users can have different interest on monitoring these garbage cans; a simple citizen would like to know if a given garbage can is free for his/her use, while a city manager would like to know if it is necessary to increase the number of garbage cans considering their usage per day.

From the network operator perspective, security tasks, like authorizing the access between and for each network entity (users, sensors, printers, services, among others), are complex and challenging to manage due to many reasons. For instance, misconfigurations may result in vulnerabilities that may compromise the overall network security. Besides, large and dynamic network environments and sensitive information also increase the management complexity.

1.1 Motivation

In traditional networks, the NAC management relies on a series of network devices like firewalls, routers, and switches, together with protocols, standards and technologies like RADIUS, IEEE 802.1x Port-based Network Access Control (PNAC), Access Control List (ACL), Virtual Local Area Network (VLAN), Lightweight Directory Access Protocol (LDAP) (OpenLDAP and Active Directory), among others.

Those solutions are inflexible and require per-device and vendor-specific manual config-

urations, which are prone to errors. Besides, misconfigurations may result in vulnerabilities, that may compromise the overall network security.

Furthermore, changes in the network require manual reconfigurations in the network devices to comply the established network security policy. Also, managing and maintaining them are expensive, which even in a small network, requires a management team [LIU et al. \(2016\)](#). Therefore, these approaches work well for stable though they are small networks and are hard to integrate and configure.

Due to those concerns, there is a need for a more sophisticated access control solutions based on high-level and autonomic policy implementations to minimize costs, and the network administrator effort and errors [KREUTZ et al. \(2015\)](#).

The SDN paradigm stands to replace those low-level configurations by high-level network access control mechanisms. SDN offers new opportunities (programmability, flexibility, dynamicity, and standardization) to overcome the above limitations [ONF \(2014\)](#).

Differently from the traditional approaches, SDN eliminates the need for configuring and integrating the many vendor-specific network devices. The basic concept of SDN is its introduction and use of controllers that have a complete view of the entire network offering an easier configuration of programmable switches and consequently simplifying the management of the traffic between the network resources [LARA; RAMAMURTHY \(2016\)](#). Besides, the SDN-based approach reduces the human intervention, which in turn reduce operational costs and misconfigurations.

Despite the new opportunities and benefits provided by the SDN paradigm, those problems in traditional networks perseveres in SDN. Furthermore, access control management remains a challenge, once managing security policies involves dealing with a large set of access control rules; detection of conflicting policies; defining priorities; delegating rights; reacting to dynamic network states and events.

1.2 Problem Statement

A survey about security in SDN [AHMAD et al. \(2015\)](#) highlights some research challenges in areas that need to be properly addressed before deploying SDN commercially. Two of those challenges regard to the **Synchronization of Network Security and Network Traffic**, and the **Network Security Automation**.

About **Synchronization of Network Security and Network Traffic**, the authors point that a stable and robust security policy deployment requires global analysis of policy configuration of all network entities to avoid conflicts and inconsistencies that may result in security breaches and network vulnerabilities. Furthermore, the network security and traffic need to be synchronized due to network changes and events.

In **Network Security Automation**, the authors point the need to automate the network security configuration, avoiding the human intervention, and manual configurations, which are

prone to errors. They highlight that configuration complexity is one of the main reasons for security breaches in enterprise networks.

The authors in [WICKBOLDT et al. \(2015\)](#) discuss challenges and management requirements in SDN. From all their listed challenges, this work is related to the following: **From High-level Rules to Network Configuration**, and **Autonomic and In-Network Management**. The first regards the loss of low-level information when using high-level commands or rules. That way, the lost information needs to be reconstructed in the process of translating high-level rules into low-level configurations. The second regards the autonomic reaction to network events.

The Open Networking Foundation (ONF)¹ promotes the adoption of SDN through open standards development, such as the OpenFlow protocol [TS-020 \(2014\)](#). Recently, the ONF Technical Recommendation document [TR-516 \(2015\)](#) specified new requirements to be met by the SDN architecture, such as **Security** and **Network Interaction Policies**.

Regarding **Security** requirements, the document determines the use of access control by enforcing policies that govern rights for each network entity. On the other hand, for **Network Interaction Policies**, the document highlights the need to create mechanisms to express, distribute, and manage interaction policies that define which operations can be performed by network entities. It also refers to the policy delegation process, that consists in delegating the rights (security policies) from one entity to others.

Table 1.1 summarizes those gaps.

Table 1.1: Research Challenges [AHMAD et al. \(2015\)](#); [WICKBOLDT et al. \(2015\)](#) and ONF Requirements and Problems to be Solved [TR-516 \(2015\)](#).

| |
|---|
| 1. Synchronization of Network Security and Network Traffic |
| 1.1. Global analysis of policy configuration of all the network entities. |
| 1.2. Synchronization to network state changes and events. |
| 2. Network Security Automation |
| 2.1. Change from manual to automatic network security configurations. |
| 3. From High-level Rules to Network Configuration |
| 3.1. Reconstruct the lost information in the process of translating high-level rules into low-level configurations. |
| 4. Autonomic and In-Network Management |
| 4.1. Autonomic reaction to network events. |
| 5. Security |
| 5.1. Govern rights by enforcing, access control policies. |
| 6. Network Interaction Policies |
| 6.1. Mechanisms to express, distribute, and manage policies. |
| 6.2. Right delegation between network entities. |

Therefore, based on those research challenges, requirements, and problems to be solved, we point out the following research questions:

- How to simplify the NAC management in SDN networks?

¹<https://www.opennetworking.org/>

- How to allow the definition of high-level access control policies to configure the network?
- How to automate the reaction of security policies against network state changes and events?
- How to maintain the synchronization between high-level policies and the network configurations?

1.3 General and Specific Goals

In face of the challenges, problems, and requirements presented in the Subsection 1.1 and summarized in Table 1.1, this work has as main goal to simplify and automate the network access control management in SDN.

As specific goals, we can highlight:

- Analyze the policy-based management in SDN.
- Provide an autonomic translation of high-level security policies into low-level OpenFlow rules to configure the network.
- Provide an autonomic synchronization of security policy configurations against network state changes and events

1.4 Organization of the Dissertation

The rest of this work is organized as follows. Chapter 2 presents background information on SDN, OpenFlow, NAC management, and access control models. In Chapter 3 we survey related work in traditional and SDN-based networks regarding NAC management. Chapter 4 presents the architecture of the HACFlow framework and describes its main components. Besides, we present a practical example of using HACFlow in a Smart City scenario. These examples demonstrate the policy expressiveness of HACFlow. After that, we analyze the performance and scalability of HACFlow in Chapter 5. Also, we compare HACFlow against similar SDN-based work. Finally, Chapter 6 concludes this work, presenting the difficulties found and future work. Lastly, we discuss open challenges.

2

Background

In this chapter, we present the main concepts related to our research problem. First, we make an overview of the SDN architecture, its contributions, and deployment challenges. Next, we describe the OpenFlow protocol presenting its message types and basic structure. Then, we present the whole Authentication, Authorization, and Accounting (AAA) process. Lastly, we present some access control frameworks and detail the OrBAC model.

2.1 Software Defined Networks

The SDN concept aims to simplify network management tasks and leverage innovation in communication networks. The SDN paradigm offers new opportunities (programmability, flexibility, dynamicity, and standardization) to solve many problems in traditional networks [AHMAD et al. \(2015\)](#).

The main change in the SDN architecture consists in decoupling the control plane from the data plane [ONF \(2014\)](#). The SDN architecture relies on the following components depicted in Figure 2.1 and detailed below.

The **Application Layer** is in the top of the SDN architecture and communicates with the SDN controller through a northbound Representative State Transfer (REST) API. SDN applications run on top of the controller, and consist of business applications that determine the network logic and behavior [ONF \(2014\)](#).

Those applications implement network functions and technologies, such as traffic engineering (load balancing, quality of service policies, energy aware routing, and so on); data center networking (detects operational problems, live network migration, optimize network utilization, and so on); security and dependability (security policy enforcement, flow-based network access control, DoS attack mitigation, among others), among others [KREUTZ et al. \(2015\)](#).

SDN applications enable developers, and network and data-center operators to programmatically manage the network, eliminating the need of manual, per-device, and vendor-specific configurations, which are prone to errors and present in traditional networks.

The **Control Layer** or control plane, consists of SDN controllers (Floodlight, HP VAN

SDN, NOX, OpenDaylight, among others). It is in the middle of the SDN architecture and is the logic of the network, where all traffic decisions are made through the setup of technologies such as OpenFlow flow rules in switches [ONF \(2014\)](#).

The control layer offers to SDN applications network statistics and a global view of the entire network topology. This global view allows an easier way to guarantee the consistency and completeness of security policy enforcement [LIU et al. \(2016\)](#). Besides, the control plane is responsible for enforcing application's configurations in OpenFlow switches.

The **Infrastructure Layer** or data plane has the role of forwarding packets according to flow rules enforced by SDN controllers in switch's flow tables. It is in the bottom of the SDN architecture and is composed of many OpenFlow-enabled switches.

A southbound API allows the communication between the control plane and the data plane. This communication occurs through the OpenFlow protocol, the most accepted and widely used implementation [BLIAL; BEN MAMOUN; BENAINI \(2016\)](#). There are other available southbound protocols, such as I2RS, PCE-P, BGP-LS, FORCES, OMI, OvSDB, NetConf/Yang, among others [NADEAU; GRAY \(2013\)](#); [PUJOLLE \(2015\)](#). We give more details about the OpenFlow protocol in Section 2.2.

The switch-to-controller communication has a fundamental role in the SDN operation. Both of them exchange relevant information, such as network events, statistics, capabilities, query configuration parameters, and others [TS-020 \(2014\)](#).

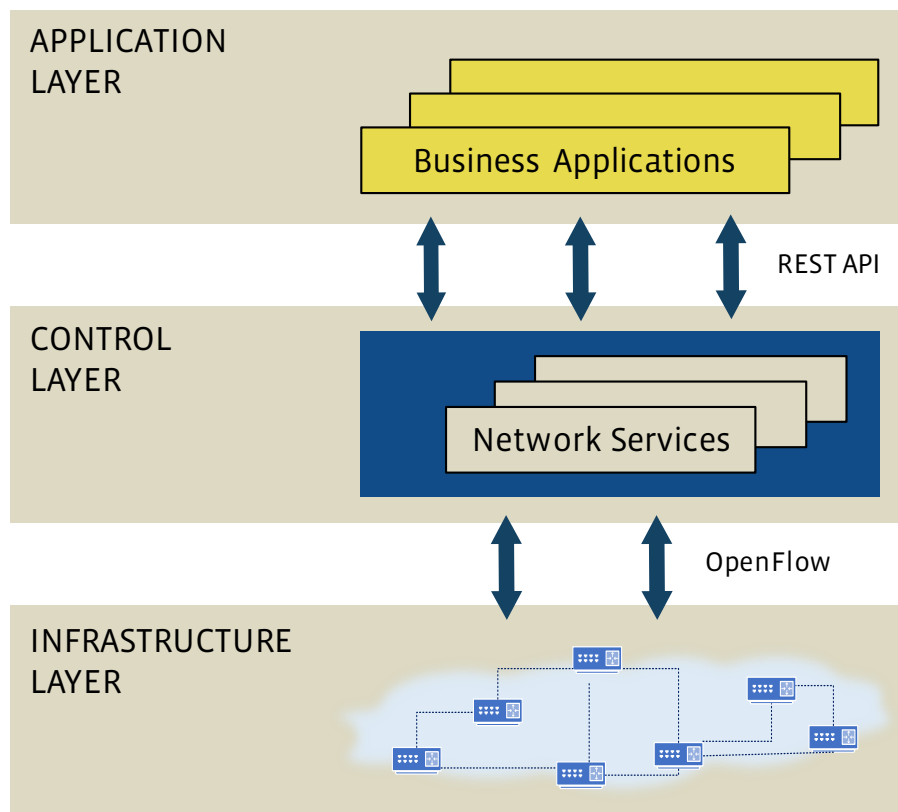


Figure 2.1: Components of an SDN Architecture. Adapted from: [ONF \(2014\)](#).

Despite their contributions, the SDN deployment also provides new challenges in many areas. Some of them are security challenges (lack of access control and accountability, DoS attacks, flooding attacks, and so on); network management challenges (high-availability and resilience, performance and scalability, monitoring, and visualization), among others. A complete and detailed list can be found in [TR-516 \(2015\)](#), [AHMAD et al. \(2015\)](#), and [WICKBOLDT et al. \(2015\)](#).

2.2 OpenFlow Standard

As explained, the SDN architecture separates the data plane from the control plane, and each OpenFlow-enabled switch is managed by a centralized SDN controller through OpenFlow messages. It is through this protocol and an OpenFlow channel, that the SDN controller exchanges information and executes management operations (insert, remove, and update flow rules) in a network switch.

This channel may be encrypted using Transport Layer Security (TLS) or run directly over Transmission Control Protocol (TCP). The three types of messages supported by the OpenFlow protocol are controller-to-switch, asynchronous, and symmetric [TS-020 \(2014\)](#).

- **Controller-to-switch** messages allow the controller to manage the switch's state by adding, deleting or modifying flow entries in a flow table. Those messages are initiated by the controller, and through them, it obtains the capabilities of a switch, send packets, and receive notifications for completed operations (like flow setup successful).
- **Asynchronous** messages are sent by the switches to the controller. Those messages may indicate an error (notify a problem in the switch), a switch state change (removal of a flow rule, sent only if OFPFF_SEND_FLOW_REM flag is enabled), or a packet arrival (packets forwarded to the controller).
- **Symmetric** messages are sent without requesting. Those messages are hello (used in connection startup), echo (to verify connection liveness, or measure latency or bandwidth), and experimental (staging area for features that aims to offer additional functionality).

The structure of an OpenFlow message consists in a header, common structures, and stats. The **header** defines terms such as the protocol version, message type, and length, among others. The **common structures** consist of a port, flow instructions, actions, experimenter and finally 42 types of matching fields which 13 are required, meaning that SDN OpenFlow capable switches must support at least those fields. **Stats** includes individual flow statistics. Figure 2.2 shows the OpenFlow structure.

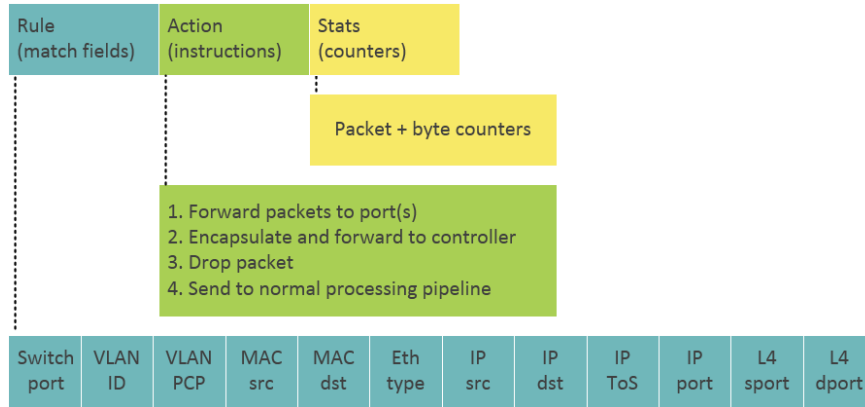


Figure 2.2: OpenFlow structure. Adapted from: yuba.stanford.edu.

It is important to highlight that most of the OpenFlow matching fields present constraints. SDN developers have to take care of each of them. If not in accordance, it is not possible to create OpenFlow flow rules. Figure 2.3¹ show those constraints.

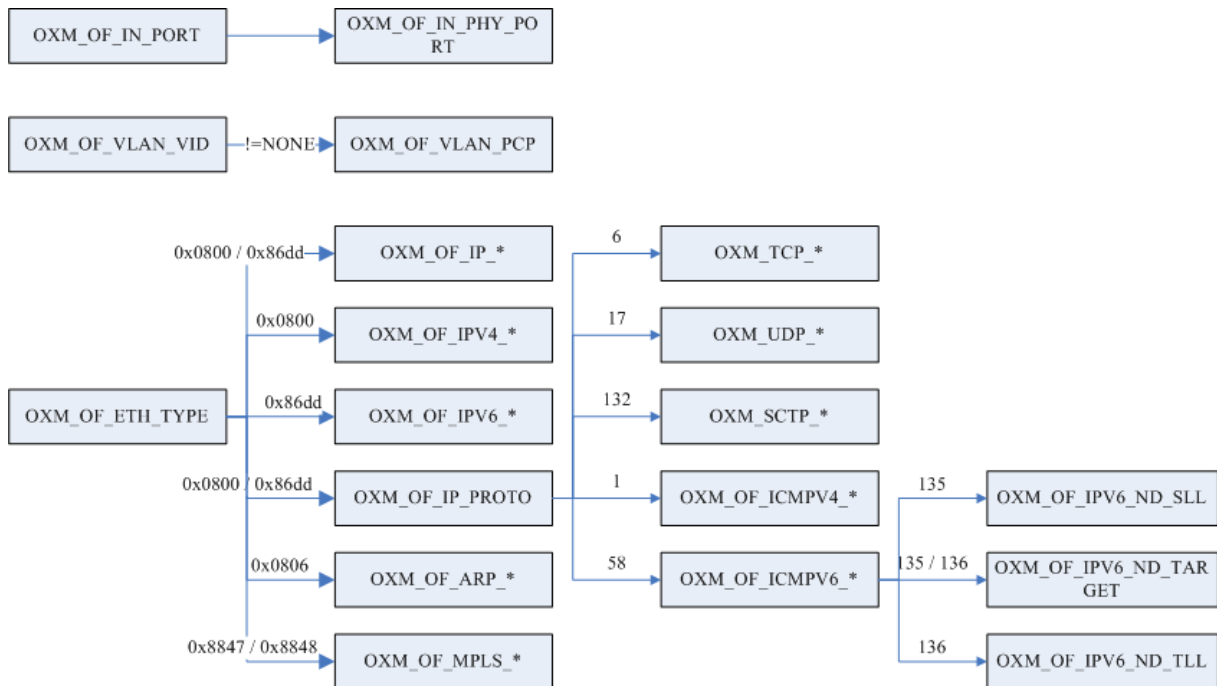


Figure 2.3: OpenFlow match field constraints. Source: flowgrammable.org.

As one may note, the ingress physical port field (*OXM_OF_IN_PHY_PORT*) depends on the ingress port field (*OXM_OF_IN_PORT*), that means, to define the physical port of the switch in the OpenFlow message, it must have the ingress port field defined. The same idea applies to the remaining fields (*OXM_OF_ETH_TYPE*, *OXM_OF_IP_PROTO*, and *OXM_OF_ICMPV6_TYPE*).

¹http://flowgrammable.org/sdn/openflow/message-layer/match/#tab_ofp_1_3

2.3 Authentication, Authorization and Accounting

Effective network security and management combines the AAA processes [CONVERY \(2007\)](#). Those mechanisms provide proper protection while access is made to a variety of network resources (hosts, servers, printers, users devices, and so on).

Authentication is the first step of the whole AAA process. It uniquely identifies entities through credentials provided by them. Once offered, the credential is compared to registered one stored in a database, and if a match occurs, the entity being authenticated gets access to the network. This access is limited, meaning that this entity is still not authorized to access the network resources.

Therefore, the Authorization process must be started next. It will be responsible for granting access to network resources to the authenticated entity. The granularity of the access granted depends on the implemented authorization mechanism.

In some configurations, when a user successfully authenticates, it can access the overall network. In others, the user is assigned to a VLAN in a VLAN-segmented network, and obtains access to a particular set of network resources. Those configurations are coarse-grained.

On the other hand, in a more sophisticated and fine-grained configuration, the access control goes a step further. The access is granted by enforcing a set of security policies on the network. Those policies (applied to a group or a single user) protect the privacy of various network entities (users, hosts, services, and so on). Each one, with different capabilities of accessing the network resources.

Finally, Accounting is the last step of AAA process. Here, the network access is audited and is the basis for billing Internet Service Provide (ISP) customers. All entities that accessed the network are recorded, registering which resources they accessed, and when they disconnected from the network [CONVERY \(2007\)](#).

2.4 Access Control Models

In the literature, there are many access control framework solutions; known examples include Role-Based Access Control (RBAC) [SANDHU et al. \(1996\)](#), Team-based Access Control (TMAC) [THOMAS \(1997\)](#), Discretionary access control (DAC) [SANDHU; MUNAWER \(1998\)](#), Mandatory Access Control (MAC) [OSBORN; SANDHU; MUNAWER \(2000\)](#), Coalition Based Access Control (CBAC) [COHEN et al. \(2002\)](#), and Attribute-Based Access Control (ABAC) [WINSBOROUGH; LI \(2002\)](#). Figure 2.4 presents their timeline.

Those approaches present some limitations such as no support delegation, not providing restrictions to control rights' propagation, no possible way to specify contextual requirements (e.g. temporal or location-based requirements), impossibility to define security policies for various organizations within a unique framework, among others [KALAM et al. \(2003\)](#).

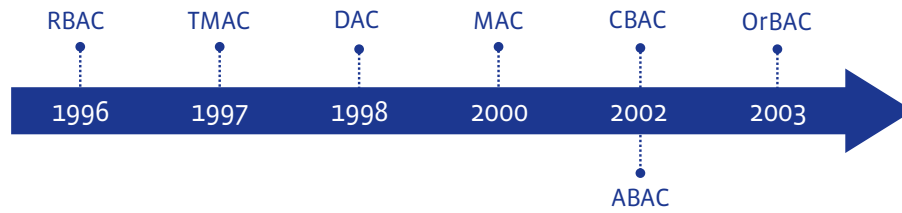


Figure 2.4: Access control models.

2.4.1 Organization Based Access Control

The Organization Based Access Control (OrBAC) model comes to supply the limitations of the access control models cited and integrate concepts like hierarchy, context, and role delegation. Differently of these approaches, OrBAC allows the creation of security policies at the abstract level and is independent of implementation, making it applicable to many scenarios. Due to that, this model was consolidated and is widely used [KALAM et al. \(2003\)](#).

Figure 2.5 depicts the abstract and concrete levels of OrBAC. The abstract level is composed of the *Organization*, *Role*, *Activity*, *View*, and *Context* predicates. When applied together, they generate an abstract policy that can be a *Permission*, *Prohibition*, *Recommendation* or *Obligation*. Consequently, it may infer a set of concrete rules composed by *Subject*, *Action*, and *Object* entities. All of these predicates, entities, and relationships will be detailed next.

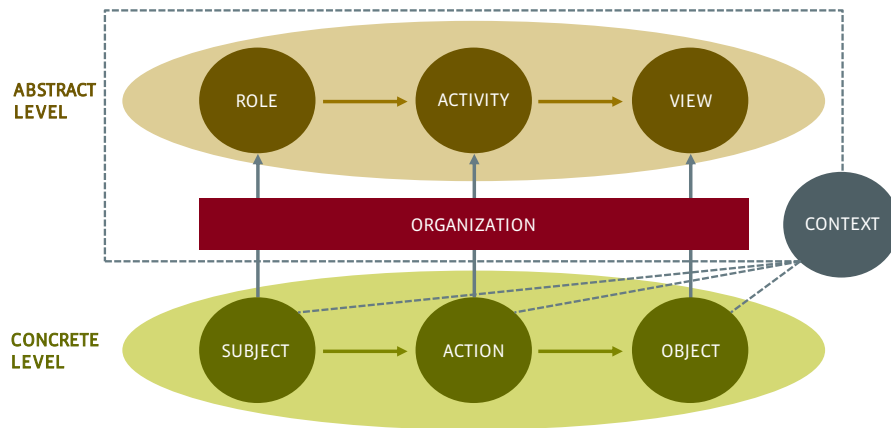


Figure 2.5: The abstract and concrete levels of the OrBAC model. Adapted: orbac.org.

2.4.1.1 Organization

The abstract security policy management is centered on the *Organization* predicate. Many things, such as a network, a city, a company, and a firewall can be modeled as an OrBAC Organization. Each one, may be structured into several sub-organizations, meaning that we can divide, for example, a city or an university in sub-sets (sub-cities, sub-universities), each one having its own security policies.

As we are proposing a framework for NAC management in SDN-based networks, the Organization may be modeled as many domains (a company, industry, a university, a smart city, a smart building, and so on). All following predicates (*Role*, *Activity*, *View*, and *Context*) must be assigned to this *Organization* to create the abstract rules. For the following examples, we will consider that we are modeling a company named CompanyX.

2.4.1.2 Role and Subject

The *Empower* relationship links a *Subject* entity to a *Role* predicate. A set of *Subjects* can be grouped in one or more *Roles*, and those roles can be structured hierarchically. Assuming that the subject Alice works in CompanyX at the Statistics department, we link them as *Empower(CompanyX, Alice, statistics)*. A *Subject* is not necessarily a person, it can also be a host. For example, assuming that we have a machine named HostA on CompanyX located at the statistics department, we can link them with the same idea: *Empower(CompanyX, HostA, statistics)*.

2.4.1.3 Activity and Action

The *Consider* relationship links an *Action* entity to an *Activity* predicate. *Actions* can be grouped in *Activities*, and the latter can be organized hierarchically. An action represents operations that a subject can perform over objects, once the subject is authenticated and the access is granted. Assuming that a subject has network access to objects from CompanyX, we may represent this network access as *Consider(CompanyX, networkAccess, Access)*.

2.4.1.4 View and Object

The *Use* relationship links a concrete *Object* to an abstract *View*. *Objects* can also be grouped in one or more *Views*, and *Views* may be structured hierarchically. An *Object* represents inanimate entities, like a machine or a network service. Assuming that we have a machine named HostA with a WebMail service on CompanyX, we can associate this service to HostA as *Use(CompanyX, WebMail, HostA)*.

2.4.1.5 Context Definition

The *Define* relationship links a context to an *Organization*, *Subject*, *Action*, and *Object*. The *Context* entity is used to create dynamic security policies. The context associated with a rule determines its state that can be active or inactive depending on the context condition. A context can represent a temporal condition (e.g. hours, days, months), a network state (e.g. safety, vulnerable, congested), a sensor state, among others. OrBAC permits the combination of contexts (e.g. temporal and network state) to express more complex contextual conditions. Assuming that we have a temporal context in CompanyX named StrictContext, with a definition like between 14 PM and 22 PM only on Fridays. We can link this context as *Define(CompanyX,*

Alice, networkAccess, WebMail, StrictContext). In this way, the subject Alice will be able to access the WebMail service only within circumstances.

2.4.1.6 Class Definition

Classes can be assigned to concrete entities (*Subject, Action, Object*) to include additional information for them. Such additional information is strictly related to the modeling domain (a firewall, a city, an university, and so on) and comes in attributes defined in a class. As an example, the HostA object may require additional particular attributes such as IP address, MAC address, host identifier, and so on.

2.4.1.7 High-level Security Policy Definition

Security policies are defined at the abstract level. The *Permission, Prohibition, Obligation*, and *Recommendation* relationships link the *Organization, Role, Activity, View*, and *Context* predicates, resulting in the final definition of a policy. The example below summarizes the axioms required to implement a high-level security policy.

$$\begin{aligned} &\forall Alice \forall networkAccess \forall WebMail \forall statistics \forall Access \forall HostA \forall StrictContext \\ &\mathbf{Permission}(CompanyX, statistics, Access, HostA, StrictContext) \wedge \\ &\mathbf{Empower}(CompanyX, Alice, statistics) \wedge \\ &\mathbf{Consider}(CompanyX, networkAccess, Access) \wedge \\ &\mathbf{Use}(CompanyX, WebMail, HostA) \wedge \\ &\mathbf{Define}(CompanyX, Alice, networkAccess, WebMail, StrictContext) \\ &\rightarrow \mathbf{Is_permitted}(Alice, networkAccess, WebMail) \end{aligned}$$

As one may note, in this example we defined the permission rule *Permission(CompanyX, statistics, Access, HostA, StrictContext)*. This security rule will be only active in this context condition (between 14 PM and 22 PM only on Fridays) and Alice (empowered in *statistics*) will be able to access the WebMail service. Out of this condition, the policy will be inactive and Alice loses her access.

2.5 Concluding Remarks

This chapter presented the main concepts related to our proposal. We presented the application, control, and infrastructure layer of the SDN architecture. We described the OpenFlow protocol presenting its message types, basic structure, and constraints. Then, we presented the whole AAA process. Lastly, we presented different access control frameworks in literature. We described in detail the OrBAC model presenting its abstract and concrete levels and relationships.

3

Related Work

In this Chapter, we survey related work in traditional and SDN-based networks regarding NAC management. We describe the benefits and limitations of each approach presenting some insights against our solution. First, we present access control technologies and standards used in traditional networks. Then, we present SDN applications and programming languages that aim to improve the security policy management. Lastly, we present some SDN-based candidates to compare against HACFlow.

3.1 Network Access Control in Traditional Networks

Firewalls are widely deployed to secure a network by providing access control. It protects a private network against unauthorized access and attacks from the public Internet. A firewall analyzes all the network communication flow (incoming and outgoing traffic) and determines, based on predefined security rules, which operations can be executed. Despite their wide adoption, firewalls do not analyze the internal traffic between network entities inside a private network (see Figure 3.1). In that case, a malicious user could attack another user, host, or service in the private network. Therefore, the integration with other mechanisms should be done to protect the network.

With HACFlow, as it provides a more granular and deeper enforcement (in switches) approach, the malicious traffic, once detected, will be immediately blocked at the source of communication, that means, at the switch that the attacker is connected to. Furthermore, adapting to the frequent network state changes and events, results in an even harder firewall deployment [AHMAD et al. \(2015\)](#). HACFlow was designed to automatically react to those state changes and events.

The IEEE 802.1x¹ standard is a PNAC solution widely used. It consists of a supplicant, software running on a client device; an authenticator: Network Access Server (NAS) like a switch or router; and an authentication server: like RADIUS², DIAMETER³, and so on. Typical

¹<https://www.ietf.org/rfc/rfc3580>

²<https://tools.ietf.org/html/rfc2865>

³<https://tools.ietf.org/html/rfc6733>

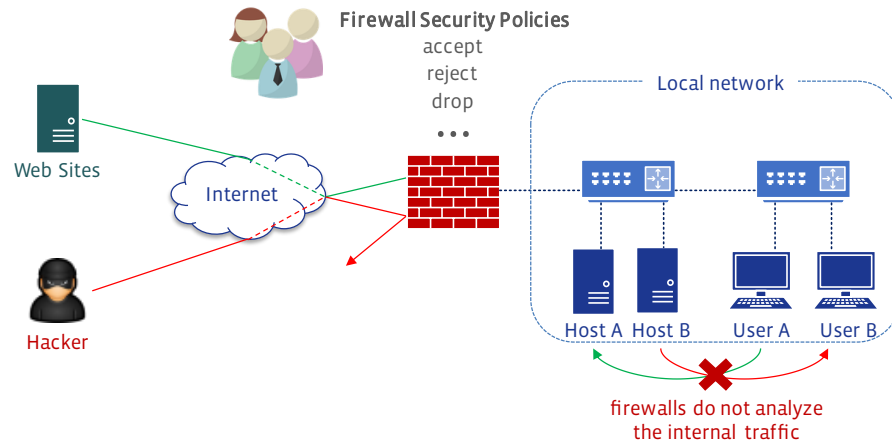


Figure 3.1: Network access control using a firewall.

configurations combine the 802.1x and RADIUS (as the AAA server) standards to provide access control by using another common mechanism named ACL.

In that setup, when a user successfully authenticates, the RADIUS server returns access privileges to the NAS port. But, ACLs present some limitations when applied to flexible environments that require continuously access control policy updates. They are also limited to only supporting allow or deny decisions for access control, that means, once a user authenticates it may access or not all network resources. Therefore, this solution is not so fine-grained (see Figure 3.2).

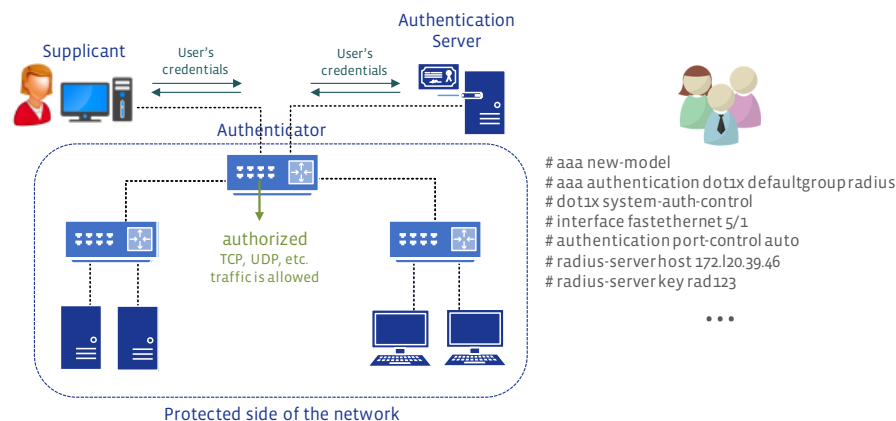


Figure 3.2: Network access control using PNAC.

Another typical configuration consists of including VLAN assignment in the previous setup to isolate the access to network entities in a VLAN-segmented network. In this setup, when a network entity successfully authenticates, its switch port is dynamically assigned to a VLAN (dynamic VLAN). VLANs make a virtual separation by grouping hosts that are not on the same network switch. It marks packets through VLAN tagging (IEEE 802.1Q tag). A switch port in a trunk mode allows traffic tagged with any VLAN to be sent to connected switches.

As a VLAN isolates a group of hosts setting them in different VLANs, to allow inter-

VLAN communication, a router or a layer three switch must be manually configured in the trunk mode. In this setup, any host in a VLAN will be able to communicate to any host on the other one (coarse-grained access control), which may result in vulnerabilities (see Figure 3.3).

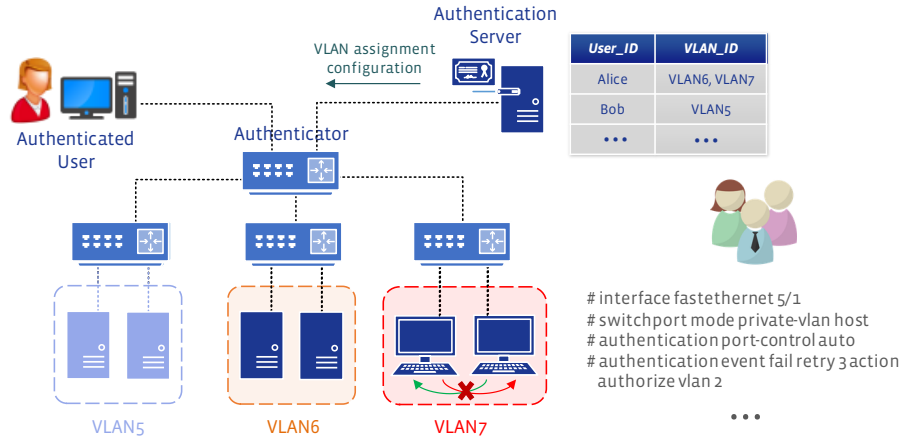


Figure 3.3: Network access control using VLAN.

If the network operator requires a more granular (fine-grained) access control (e.g. a single host in a VLAN being able to communicate to a particular set of hosts (not all) in the other VLAN), using VLANs it is not achievable. Another limitation of using VLANs is that they are restricted to a number of 4096 ports, a number that can be easily achieved in large scenarios. HACFlow overcomes those limitations by providing a fine-grained access control solution.

More sophisticated approaches like OpenLDAP ⁴ and Active Directory (AD) ⁵, that implement the LDAP, may be combined with 802.1x and RADIUS to overcome some of the previous limitations. For example, OpenLDAP and AD allow more complex configurations and offer support to dynamic network updates.

LDAP is a protocol for accessing and maintaining distributed directory information services. Those directories may be composed of sensible and critical data. LDAP may be used to control access to various network entities like users, hosts, services, printers, and others. That way, network operators may define directory services to control the access to many network entities. Therefore, granting access for users to those directories become critical (see Figure 3.4).

Differently from ACL, OpenLDAP and AD are not limited to only allow or deny policies; they provide a more granular approach (grouping by role, or defining some circumstances). OpenLDAP and AD may also be integrated with the Kerberos ⁶ technology. Kerberos aims to replace the ACL approach by a more advanced authorization model like ABAC and the eXtensible Access Control Markup Language (XACML) standard. Despite their improvements, setting those services is often hard to integrate and configure HU et al. (2014).

⁴<http://www.openldap.org/>

⁵<https://technet.microsoft.com/en-us/library/9a5cba91-7153-4265-adda-c70df2321982>

⁶<https://www.ietf.org/rfc/rfc4120>

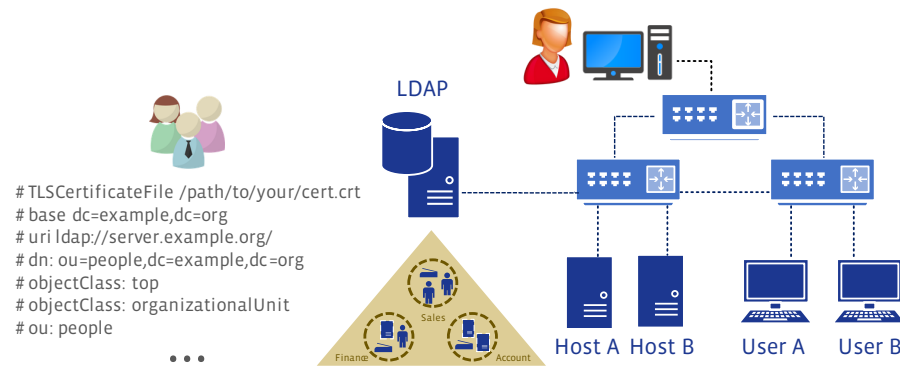


Figure 3.4: Network access control using LDAP.

The aforementioned approaches (firewalls, PNAC, VLAN, LDAP, and Kerberos), normally rely on manual configurations (in firewalls, RADIUS server, routers, switches), being highly exposed to misconfigurations, and resulting in a time-consuming task [MATIAS et al. \(2014\)](#). Therefore, the employment of those technologies becomes harder in dynamic and large networks scenarios, requiring a management team. Besides, there is a lack of granularity and expressiveness to implement the network access control, requiring the combination of different solutions. Figure 3.5 depicts the combination of these solutions (firewall, PNAC, VLAN, LDAP) to control the access in a traditional network. HACFlow aims to mitigate those limitations by allowing network configuration in a high-level, expressive, and fine-grained way and providing mechanisms to automate the network re-configurations.

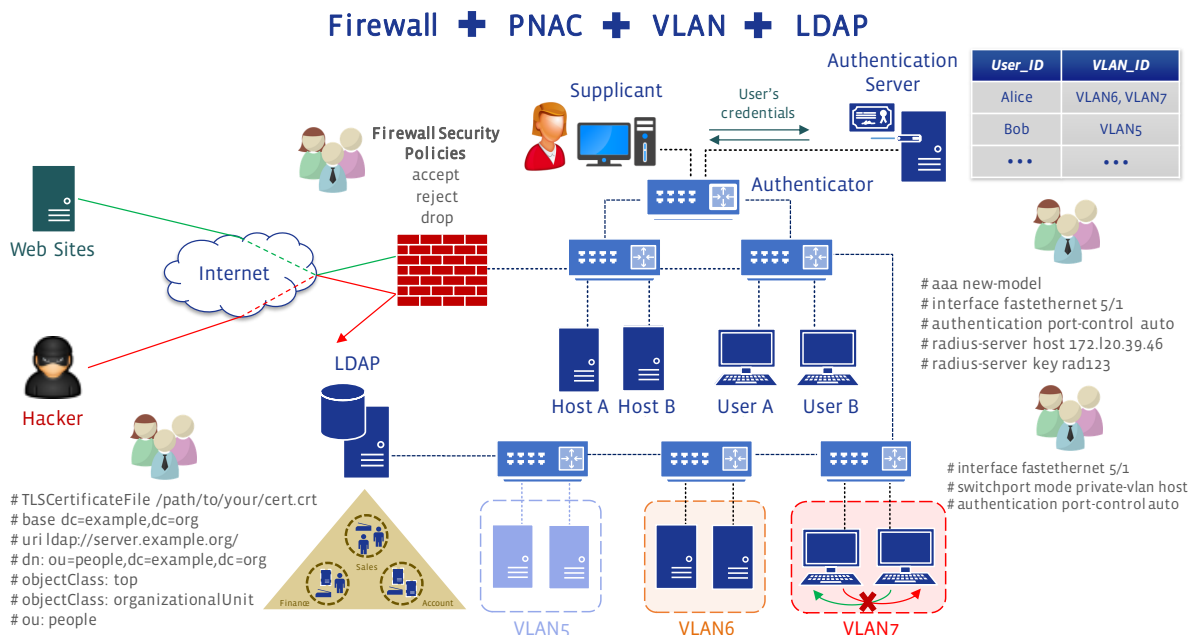


Figure 3.5: Combining different solutions of traditional network to control the access of network entities.

3.2 Network Access Control in SDN

Recently, many efforts have been made to define high-level **SDN programming languages**, such as Frenetic [FOSTER et al. \(2011\)](#), Procera [VOELLMY; KIM; FEAMSTER \(2012\)](#), PonderFlow [BATISTA; FERNANDEZ \(2014\)](#), and **SDN network applications**, such as Cloud-Watcher [SHIN; GU \(2012\)](#), FRESCO [SHIN et al. \(2013\)](#), FlowNAC [MATIAS et al. \(2014\)](#), OpenSec [LARA; RAMAMURTHY \(2016\)](#). Figure 2.4 presents their timeline.

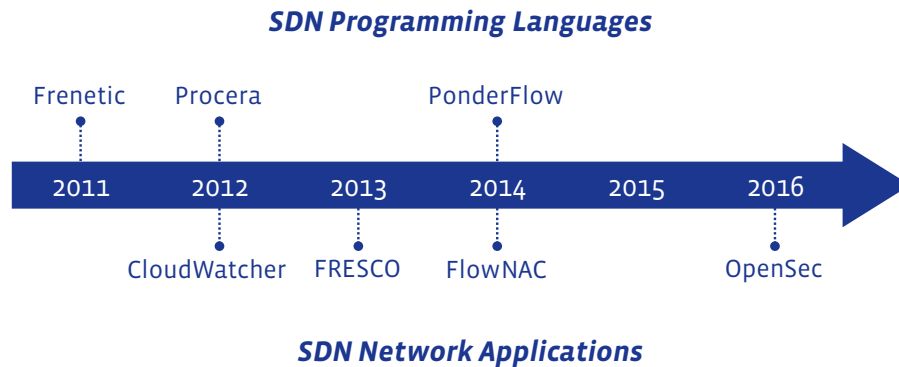


Figure 3.6: SDN-based solutions.

These proposals focused on specific aspects (expressing packet-forwarding policies, expressing access control lists, security policy enforcement, flow-based access control, fine-grained access control, and so on), while seeking to facilitate and automate the network management.

[SHIN; GU \(2012\)](#) proposed CloudWatcher, a framework that provides security monitoring services that allow network operators to create security policies that define which flow must be investigated by those services. CloudWatcher consists of a policy manager, a routing flow rule translator (considers the best path to send the monitoring traffic to security services), and an enforcer to setup flow on switches.

The main weakness of this framework is that it only uses four matching fields of the OpenFlow protocol (source/destination IP address and source/destination port) to create the security policy. This limitation restricts the variety of packets to inspect. Another limitation is that the security policies defined in CloudWatcher are script-based (in a low-level, using IP addresses), and not in a high-level way (using high-level names, like users, hosts, and services), as in HACFlow.

[VOELLMY; KIM; FEAMSTER \(2012\)](#) proposed Procera, a framework to express event-driven network policies based on Functional Reactive Programming (FRP), like in Frenetic. Similar to HACFlow, Procera focus on two network management problems: (1) configure the network using a high-level language/model, and the (2) dynamic network reconfiguration according to the frequent network state changes and conditions. Both solutions define the high-level policies, translates them into OpenFlow flow rules, and enforce the switch-level rules on the underlying network infrastructure.

In Procera, the implementation of reactive network policies relies on the following four control domains. (1) Time: depends on the date or time of day; (2) data usage: depends on the amount of data usage or data transfer rate; (3) status: device's or user's privilege; and (4) flow: based on various field values of a flow rule [KIM; FEAMSTER \(2013\)](#). Those domains can be combined to implement richer network policies.

In HACFlow, the reactive policies rely on context definitions provided by the OrBAC model. Those contexts can also be combined. Despite the Procera contributions, we cannot find any scalability (increasing the number of security policies) and performance (time needed to translate the high-level policy into OpenFlow flow rule; and the required time to Procera reacts to network state changes and events). The Section 5.1 presents those and more analysis regarding the HACFlow framework.

[MATIAS et al. \(2014\)](#) proposed FlowNAC, a Flow-Based Network Access Control solution based on an extended version of the IEEE 802.1x standard. FlowNAC allows the definition of policies that specifies which network services a user can access. Differently from HACFlow, FlowNAC does not focus on dynamic policies. Furthermore, FlowNAC enforces its policies proactively, meaning that the flow setup of all policies occurs in advance of the connection of a user to the network. This approach results in the deployment of flow rules not required at the current moment, and consequently, the waste of switches' resources (like TCAM memory).

On the other hand, HACFlow enforcement is reactive, that means, on-demand and as a result of a networking event (i.e. user authentication), network state change (i.e. vulnerability state), or policy context change (i.e. a user can access a host only on weekends). The advantage of the proactive approach against the reactive one is that the user requesting the access to a network host has not to wait for the flow setup (time to a controller pushes flow rules to OpenFlow switches) to access this host.

[BATISTA; FERNANDEZ \(2014\)](#) proposed a framework based on the Ponder language, named PonderFlow. Ponder allows the management and specification of security policies for distributed systems. The PonderFlow framework extends the Ponder language to enable the creation of OpenFlow flow rules.

The main limitation of PonderFlow is the absence of a policy conflict resolution mechanism. Furthermore, PonderFlow does not translates high-level network policies into OpenFlow. On the other hand, once HACFlow is based on the OrBAC model, it is able to detect and solve conflicting policies. HACFlow also translates high-level security policies into OpenFlow flow rules, leaving to controller's application the enforcement of those rules on the network.

3.3 Candidates to Comparison Against HACFlow

From the SDN-based solutions that allow network administrators to define high-level security policies to configure the network, we selected three to compare against HACFlow,

they are: Frenetic [FOSTER et al. \(2011\)](#), FRESCO [SHIN et al. \(2013\)](#), and OpenSec [LARA; RAMAMURTHY \(2016\)](#). We chose them due to their similarities with HACFlow and available data for conducting the comparisons.

We describe them next and Section 5.2 makes a comparison regarding (1) Framework Features, (2) Policy Definition Simplicity, (3) Time to Translate Policies, and (4) Time to React Against Network Events. We chose these points once they are relevant aspects offered by each solution to simplify and automate the network access control management in SDN.

[FOSTER et al. \(2011\)](#) proposed Frenetic, a high-level language for OpenFlow networks based on FRP and SQL-like queries. The Frenetic architecture consists of an implementation of the FRP operations (to define high-level policies), a run-time system (to translate high-level policies into low-level packet-processing rules, and to manage the enforcement of flow rules), and the NOX SDN controller. Recently, Frenetic has been extended in two main directions.

HACFlow and Frenetic manage the network traffic by defining high-level policies. In both, network managers do not take care of how those policies will be implemented and enforced on the network.

[SHIN et al. \(2013\)](#) proposed FRESCO, a security framework focused on enforcing security constraints of SDN applications. With FRESCO, those applications can replicate security functions like firewalls and attack deflectors. In FRESCO, network operators define high-level security policies based on a scripting language.

This language, relies on the block, deny, allow, redirect, and quarantine security primitives. Those high-level security policies are translated into OpenFlow flow rules and enforced on OpenFlow-enabled switches. While, in HACFlow, the high-level policies are based on the OrBAC model and automatically translated into flow rules to be enforced on the network by SDN applications. Thus, network operators focus on the high-level goals, not requiring taking care of how the low-level OpenFlow flow rules will be implemented.

FRESCO and HACFlow react automatically (by reprogramming OpenFlow switches) to network alerts according to predefined configurations implemented by network managers. Furthermore, both can detect and solve conflicting high-level policies. Those features avoid the setup of overlapping of OpenFlow flow rules.

To detect and solve conflicts, FRESCO includes a Security Enforcement Kernel (SEK) module integrated to the NOX SDN controller. This feature avoids flow rules from a security SDN application to compete with non-security-critical ones [PORRAS et al. \(2012\)](#). Therefore, supporting FRESCO in other SDN controllers require the implementation of this module, not being a straightforward integration.

On the other hand, the OrBAC component in HACFlow framework detects and solve conflicting policies. The HACFlow deployment does not require any extension or modification in SDN controllers, as FRESCO requires.

Similarly to CloudWatcher, [LARA; RAMAMURTHY \(2016\)](#) proposed OpenSec, a security framework to automate the implementation of security policies. In OpenSec, the

network operator defines high-level goals (high-level security rules) to determine by which processing units (DDoS, DPI, spam detection, among others) a traffic must be monitored. While, HACFlow implements high-level security policies to determine which actions network entities can perform.

OpenSec and HACFlow react dynamically to network alerts by enforcing switch-level rules. Besides, they allow network managers to previously define how this reaction must be implemented/enforced according to the alert received.

3.4 Concluding Remarks

This Chapter presented the advantages and drawbacks of existing NAC solutions in traditional network, such as firewall, PNAC, VLAN, and LDAP. We explained how each solution may be used to implement network access control, presenting its expressiveness and granularity. Next, we presented the SDN-based network access control solutions, that may be divided as SDN applications and programming languages. Lastly, we presented SDN-based candidates to compare against HACFlow.

4

HACFlow

This chapter presents our proposed framework named HACFlow. Then, we detail how to operate HACFlow to govern rights of network entities in a smart city case study scenario.

4.1 HACFlow Framework

Applying high-level security policies to govern users' rights involves issues like managing a broad set of policies, solving conflicting rules, and dealing with the dynamic nature of networks. This complexity becomes even harder when managing large networks and critical data. In this section, we present the HACFlow framework and explain how it overcomes the challenges, problems, and requirements presented in Chapter 1. Next, we describe the HACFlow architectural components and detail the role of each one to improve the network access control management.

4.1.1 Overview

HACFlow is a High-level Access Control management framework for SDN based on the OrBAC model. HACFlow aims to simplify and automate the NAC management providing mechanisms to define dynamic, fine-grained, and high-level access control policies, detect and solve conflicting policies, delegate roles, and react to network state changes and events. Next, we pass through the points highlighted in Table 1.1 and explain how HACFlow addresses each of them.

Synchronization of Network Security and Network Traffic: HACFlow takes advantage of the OrBAC model to address these challenges. As all the network entities are mapped into OrBAC, we can achieve that global analysis as well as detect and solve conflicting policies. HACFlow also provides a mechanism to guarantee the security synchronization by reacting automatically to network changes. **Network Security Automation:** HACFlow mitigates those concerns providing mechanisms to automate management tasks, and allows network operators to configure the network by defining high-level security policies.

From High-level Rules to Network Configuration, and Autonomic and In-Network Management: HACFlow addresses these challenges by allowing the definition of high-level

security policies and translating them into low-level OpenFlow flow rule configurations. Furthermore, HACFlow allows creating dynamic and contextual security policies that automatically react to network events.

Security and Network Interaction Policies: HACFlow allows network operators to govern rights to network entities by enforcing access control policies. Besides, HACFlow provides mechanisms to express, distribute, and manage network interaction policies. **Express** directive refers to the creation of high-level security rules based on the OrBAC model. **Distribute** implies to the automatic conversion from high-level policies to low-level OpenFlow flow rules and enforce them on the network devices. **Manage** includes the resolution of conflicting policies, definition of rule priorities, creation of dynamic policies based on contexts, automatic reaction to network state changes and events, and delegating rights.

It is important to highlight that HACFlow is not responsible for the identification and authentication of network entities, neither for the integrity of policy file data and monitoring the network. We consider that a third party SDN application authenticates the network entities and such SDN application uses HACFlow for authorizing them. The integrity of the policy file data could be required when deploying HACFlow in production. Lastly, we consider that third party monitoring systems must notify HACFlow about network state changes and events, in order to HACFlow react properly to them. Table 4.1 lists the features covered and those considered outside the scope of the HACFlow framework.

Table 4.1: HACFlow Framework Scope.

Features Covered

1. Govern rights for each network entity in a fine-grained way
2. Define high-level and dynamic security policies
3. Translate high-level policies into low-level OpenFlow flow rules
4. Delegate rights between network entities
5. React automatically to network state changes and events

Features Not Covered

1. Identification and authentication of network entities
2. Maintain the privacy and integrity of policy data
3. Monitor the network

4.1.2 Architecture

The HACFlow framework architecture is composed of many sub-components that work together to comply with all the previous issues. Next, we describe the role of each one, and how they interact with each other. The architectural components are OrBAC API, Policy Skeleton, Entity Manager, Event Listener, Policy Translator, and REST API Interface. Figure 4.1 depicts the HACFlow framework architecture and their interaction.

As one may note, we integrated the OrBAC model inside the HACFlow architecture. The REST API and the Event Listener components interacts with OrBAC through its API.

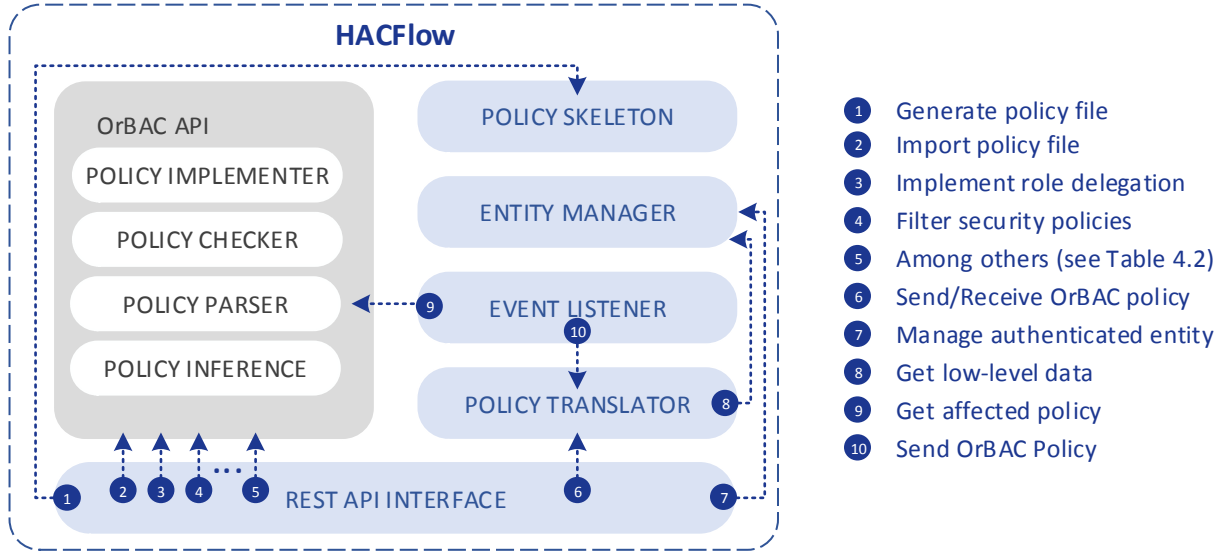


Figure 4.1: HACFlow Framework Architecture.

The REST API component is composed of many methods that interacts with OrBAC (see items 2 to 5) to import a policy file, implement role delegation, filter security policies, among others (a complete list can be found in Table 4.2). Methods of the REST API component also interacts with the Policy Skeleton component to get a pre-configured security policy file (see item 1), interacts with the Entity Manager to manage the authenticated network entities (see item 7), and with the Policy Translator component to translate the security policies get from OrBAC (see item 6).

The Event Listener implements an OrBAC interface in order to be notified about contextual changes in security policies (see item 9). Once the Event Listener receives a changed policy, it communicates with the Policy Translator (see item 10) to translate the OrBAC security rule into an OpenFlow flow rule. In the translation process, the Policy Translator must reconstruct the low-level data (lost when achieving higher levels of abstraction), to do so, it interacts with the Entity Manager component to obtain those low-level data (see item 8). Lastly, HACFlow sends the OpenFlow flow rules to a third party SDN application.

4.1.2.1 OrBAC API

As we said, HACFlow takes advantage of the OrBAC model to define high-level security policies. We choose the OrBAC model due to its high-level of abstraction to define security policies, completeness, and advantages (hierarchical structure, context-aware policies, role and rule delegation, among others) when compared to similar approaches, as presented in the Subsection 2.4.1.

We included the OrBAC API as a sub-component of HACFlow. The OrBAC component is one of the most important features in HACFlow. Its main role is to allow the definition of high-level and context-aware security policies. Besides, it provides mechanisms to detect and solve conflicting policies. The OrBAC API is composed of policy implementer, policy checker,

policy parser, and policy inference.

The **policy implementer** allows the creation of predicates (*Organization, Role, Activity, View, and Context*), entities (*Subject, Action, and Object*), and abstract permission and prohibition policies. Then, the **policy checker** checks for constraints and conflicts in those abstract policies. Next, the **policy parser** generates the concrete rules from the abstract policies. Lastly, the **policy inference** infer the concrete rules considering its states, that can be active or inactive (in/out of context), and preempted or not preempted (lower/higher priority).

4.1.2.2 Policy Skeleton

As HACFlow uses OrBAC to create the high-level security policies, we needed to make some configurations to enable HACFlow to implement some tasks. Therefore, the Policy Skeleton generates a policy template file containing those configurations, avoiding human intervention and misconfigurations.

The first configuration regards to the role delegation and revocation. HACFlow provides to network operators all configurations required to setup this feature in OrBAC, enabling them to delegate roles.

The second configuration refers to the creation of *class definitions* in OrBAC. As explained in Subsection 2.4.1.6, those classes provide additional information to network entities. Those classes allow HACFlow to derive the OpenFlow flow rules from the high-level security policies. Therefore, we defined the following classes in Figure 4.2.

The *ID* attribute links an authenticated network entity to a *subject* and *object* in OrBAC. The *IP_PROTO* and *PORT* attributes determine the IP protocol, and port number used to communicate the source and destination network entities. The remainder low-level data is obtained by the Policy Translator component in HACFlow.



Figure 4.2: HACFlow skeleton classes.

4.1.2.3 Entity Manager

The Entity Manager keeps a record of authenticated network entities. Its role is to maintain the synchronization of this record with the network. This information is provided by SDN controllers, once they have a global view of network entities.

This record contains additional information obtained from the authentication. It includes the entity *identifier*, *IP address*, *MAC address*, *connected switch*, and *switch port*. Such information is used by the policy translator to construct the OpenFlow flow rule.

4.1.2.4 Event Listener

The Event Listener is one of the main features of HACFlow. It automatically processes network events and policy's context changes. HACFlow receives those events from the controller and they can be the result of a user authentication, a vulnerability alert detected by a security service, among others. The main role of the event listener is to maintain the synchronization between the high-level security policies to network configurations.

In order to HACFlow be able to react to those events, network operators must previously define context conditions and link it to a security policy. Therefore, HACFlow allows the network operator to describe how to react in case malicious traffic is detected. Subsection 4.1.5 exemplifies.

4.1.2.5 Policy Translator

One of the main features provided by HACFlow is the autonomic policy translation. It allows network operators to define high-level goals without taking care of how they will be implemented in the network.

As HACFlow allows network operators to define policies in a high-level way. Such high level of abstraction results in the loss of low-level network information (e.g. IP address, MAC address, port number, connected switch, among others).

Therefore, these low-level data must be reconstructed in the translation process. This way, while translating a high-level security policy into a low-level OpenFlow flow rule, the HACFlow architectural components (OrBAC API, Policy Translator, and Entity Manager) must work together to perform the translation.

As an example, once a user authenticates on the network, a third party SDN application responsible for authenticating the user will notify HACFlow (through its REST API) to start the authorization process. At first, once notified HACFlow will (step one) get the user's high-level policies through the OrBAC API and (step two) pass them to the Policy Translator. Next, the Policy Translator (step three) gets from the Entity Manager the low-level data and then translates the high-level rule (OrBAC policy) into the low-level rule (OpenFlow flow rule). Lastly, HACFlow (step four) returns the OpenFlow rules to the third party application to enforce the user's rules in the network. Finally, the user will be able or not to access the network entities (hosts, printers, services, among others). Figure 4.3 depicts the whole translation process.

HACFlow provides a smart translation process. While translating, HACFlow checks for OpenFlow constraints (see Figure 2.3) and authenticated entities. The security policies are translated and enforced if both (source and destination) network entities were authenticated. This makes the translation process faster, and mainly saves switch's resources like Ternary Content-Addressable Memory (TCAM).

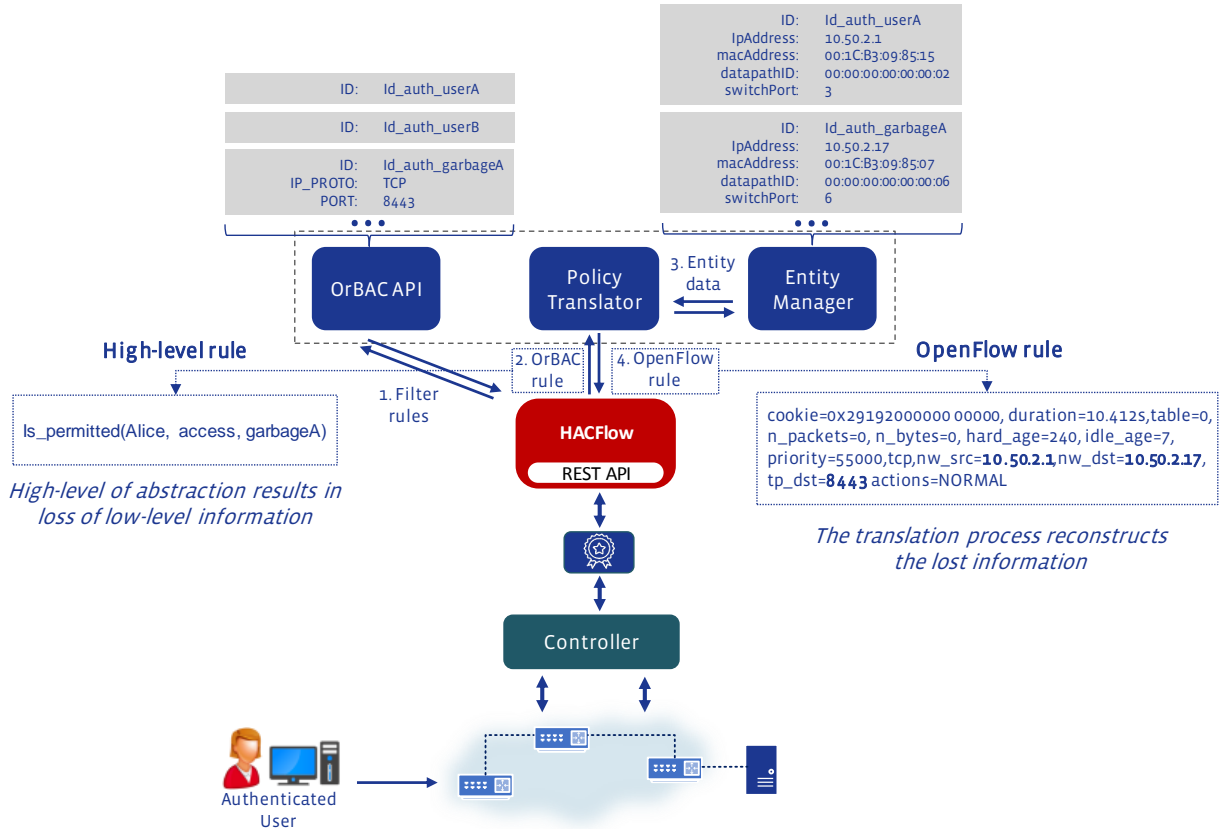


Figure 4.3: How HACFlow translates an OrBAC policy into OpenFlow.

4.1.2.6 REST API

The REST API is the point of interaction with SDN applications. Through this API, the HACFlow framework receives network events and responds to them, maintaining the security policies synchronized to the network configurations. Besides, it provides methods to load policy file data, register authenticated entities, get entity's security policies, delegate roles, among others. Table 4.2 summarizes all REST API methods of HACFlow.

4.1.3 Step-by-step: High-level Policy Definition

In this subsection, we describe how a network operator can express their high-level goals into high-level security policies. In a nutshell, operators should first define the environmental network model (i.e., a hospital or a smart city); in other words, the domain in which the abstract policies will be referred. Then, they must specify the entities (users, hosts, services, actions, and so on), classify those entities as *Subject*, *Action*, or *Object*, and group them in *Roles*, *Activities*, or *Views* predicates.

Lastly, those entities and predicates may be put together to create permission and prohibition security policies. Those policies say who can access what in which circumstances the network resources. The Section 4.2 demonstrates how to create those high-level security rules.

Required steps:

Table 4.2: HACFlow REST API resources.

| Method | Resource | Parameters | Description |
|--------|--------------------------------------|---|---|
| POST | /policyupload | file | Uploads a policy file. |
| GET | /abstractpermissions | none | Returns all security permissions. |
| GET | /abstractprohibitions | none | Returns all security prohibitions. |
| GET | /allabstractrules | none | Returns all security policies (permissions and prohibitions). |
| GET | /allconcreterules | none | Returns all concrete rules (permissions and prohibitions). |
| GET | /rulepriorities | none | Returns the priority of all concrete rules (permissions and prohibitions). |
| GET | /authenticity/highlevelrules/{id} | entityId | Returns all security policies of a single network entity (permissions and prohibitions). |
| GET | /authenticityrules/{id}/permissions | entityId | Returns all security permissions of a single network entity. |
| GET | /authenticityrules/{id}/prohibitions | entityId | Returns all security prohibitions of a single network entity. |
| POST | /authenticity/openflowrules | identification ipAddress macAddress dataPathId switchPort | Registers an authenticated network entity and returns its security policies (permissions and prohibitions). |
| GET | /authenticities | none | Returns all authenticated network entities. |
| POST | /authenticated | identification ipAddress macAddress dataPathId switchPort | Registers an authenticated network entity. |
| POST | /unauthenticated | identification ipAddress macAddress dataPathId switchPort | Unregisters an authenticated network entity. |
| PUT | /nofify/networkstatechange | type state | Notifies a network state change or event. |
| GET | /context/{org} | organization | Returns all contexts definitions defined in an organization. |
| GET | /classmembers/{class} | className | Returns the attributes and values of instances of a single class. |
| POST | /delegate/role | organization role grantee | Delegates a role. |

| Method | Resource | Parameters | Description |
|--------|--------------|---------------------------------|-----------------|
| POST | /revoke/role | organization role grantee | Revokes a role. |

1. Create the *Organization*, *Role*, *Activity*, *View*, and *Context* predicates.
2. Create the *Subject*, *Action*, and *Object* entities.
3. Assign entities of the previous step to a *Class definition*.
4. Link these predicates and entities using the *Permission* and *Prohibition* relationships to compose and obtain the high-level policy.
5. Manually solve conflicting policies detected by OrBAC through priority assignment.
6. Load the security policy file data into HACFlow.

Completed these six steps, HACFlow is able to implement the security policies in the network. Steps 1 to 5 can be done using the MotOrBAC graphical tool [AUTREL et al. \(2008\)](#). Figure 4.4 summarizes the previous six steps to define a high-level policy in HACFlow (steps A and B). As one may note, at first (step A) the network operator define its high-level security policies and then (step B) import the policy in HACFlow.

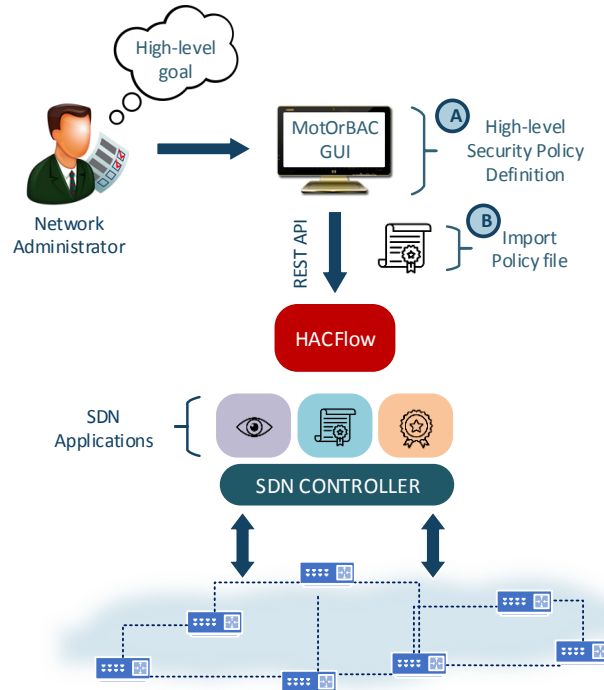


Figure 4.4: High-level policy definition in HACFlow.

In step 5, we aim to extend the OrBAC model to provide a semi-automated policy conflict resolution mechanism, based on some configuration parameters specified by the policy manager. Section 6.2 present more details.

4.1.3.1 Security Policy Expressiveness and Granularity

HACFlow allows network operators to define high-level access control policies in a much more fine-granular and expressive way when compared to the solutions of traditional networks (see Section 3.1). The Figure 4.5 depicts different levels of granularity and expressiveness to define a variety of security policies in HACFlow.

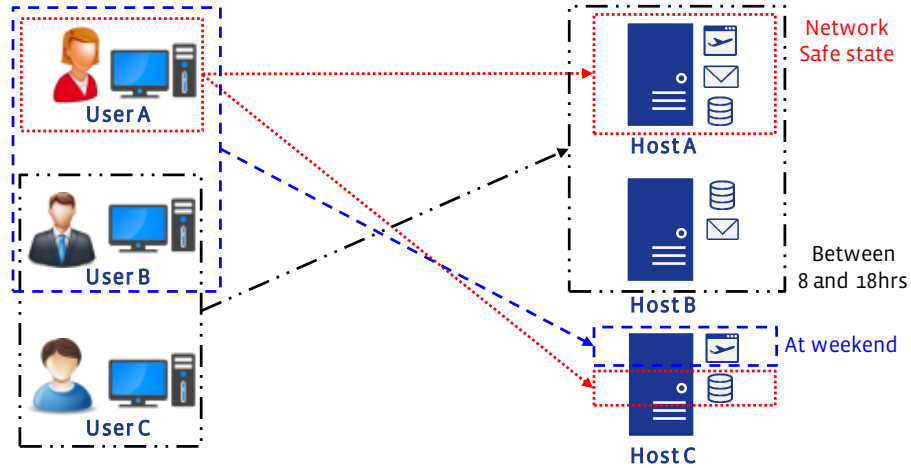


Figure 4.5: HACFlow policy granularity and expressiveness.

As one may note, the two first security policies regarding the *User A* allow it to access all the resources in *Host A* and a single resource in *Host C*. Besides, another security policy allow the users *User A* and *User B* to access all the resources in *Host A* and *Host B*. Lastly, we have a policy that allows both *User A* and *User B* to access only the database service in *Host C*.

Furthermore, all the access control policies previously described contains a circumstance in which the users will be able to access each resource. In that case, we have circumstances such as network state, hours of the day, and days of the week.

4.1.4 Step-by-step: Dynamic Security Policies

In the previous subsection, we explained how a network operator may create security policies. In this subsection, we demonstrate how a contextual condition is linked to a security policy to make it dynamic. Next, we describe how HACFlow automatically reacts to a dynamic policy to reconfigure the network.

Suppose that an enterprise network operator wants to control the access to a network resource (for instance, financial report system) imposing some circumstances (day of a week, an hour of a day, among others). The operator determines that the access must only occur during the working hours from Monday to Friday.

To implement this high-level goal the operator must:

1. Create the *Context* predicate.

2. Create the context definition (determines the circumstance).

These steps can be done using the MotOrBAC tool. The context definition, in this example, can be expressed in two different ways. The first in *BeanShell* and the second in *Temporal* context type. Figure 4.6 depicts this definition.

As we have a dynamic security policy that may have its state changed at any moment the network must be reconfigured to comply with the enterprise high-level goals. Therefore, HACFlow provides mechanisms to do it automatically, without human intervention.

All the steps to HACFlow react to a dynamic policy are:

1. The OrBAC API notifies the affected security rules.
2. The Policy Translator translates the security rule into OpenFlow flow rules.
3. HACFlow sends the OpenFlow flow rules to be enforced and change the network configuration.

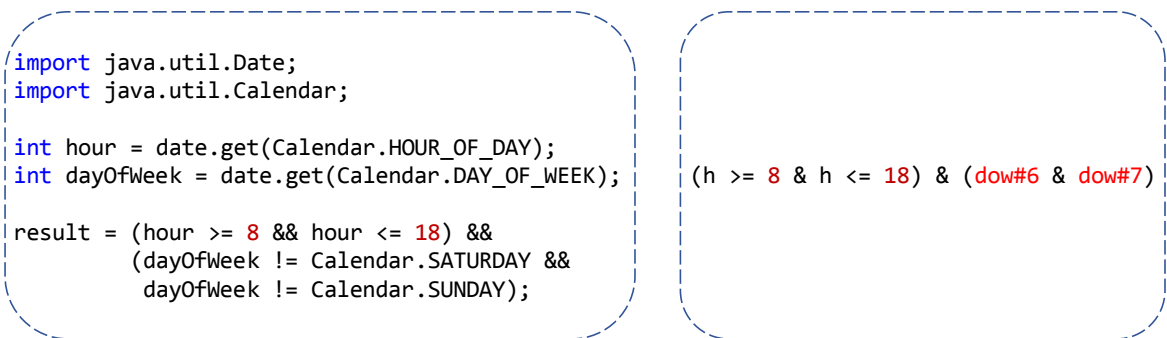


Figure 4.6: *BeanShell* and *Temporal* contexts.

Once HACFlow reacts to a dynamic policy, the network entities linked to this policy will be able to access (if the policy becomes active) or not (if the policy becomes inactive) the network resources in the policy. Subsection 5.1.3 presents a performance analysis of the reaction of HACFlow against dynamic policies.

4.1.5 Step-by-step: Reacting to Network Events

Besides reacting autonomously to dynamic security policies, HACFlow also provides mechanisms to automatically react to network state changes and events. In this subsection, we describe how a network operator can configure HACFlow to interpret network events and alerts. HACFlow receives these events and alerts through its REST API (see Table 4.2). We present two examples, the first represents a vulnerability alert and the second an authentication event. Furthermore, we show the steps required to HACFlow react to those events and reconfigure the network.

4.1.5.1 Vulnerability Alert

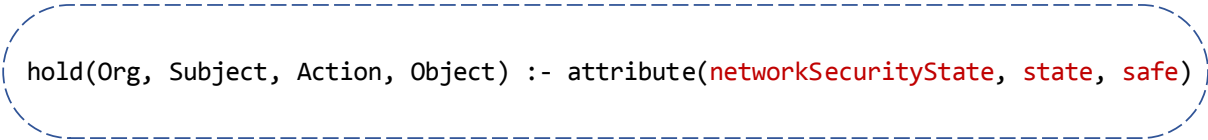
A network state change could be a vulnerability alert sent by a security monitoring system on the network like Distributed Denial of Service (DDoS), Deep Packet Inspection (DPI), and so on. It is important to highlight that HACFlow does not provide any network monitoring mechanism to trigger events. But, it allows network operators to say how HACFlow must react against those network events.

As an example, suppose that a security monitoring system detects an attack and sends an alert notifying that the network is vulnerable. As a result, the operator wants to immediately block any access to the storage service to protect the overall enterprise data.

To implement this high-level goal, the operator must execute the following steps:

1. Create the *Object* entity (representing the network event) and assign to a *Class definition*.
2. Create the *Context* predicate.
3. Create the context definition (determines the circumstance).

Supposing that in step one the entity "*networkSecurityState*" was created and assigned to a class with the attribute "*state*", the context definition must look like the one depicted in Figure 4.7.



```
hold(Org, Subject, Action, Object) :- attribute(networkSecurityState, state, safe)
```

Figure 4.7: *Prova* context definition for vulnerability alert.

Next, when HACFlow receives this event it will automatically reconfigure the network to block any traffic to the storage service. Therefore, it will execute the following tasks:

1. Through the OrBAC API, change the entity attribute (created in step one above) according to the alert received.
2. The OrBAC API notifies the affected security rules.
3. The Policy Translator translates the security rule into OpenFlow flow rules.
4. HACFlow sends the OpenFlow flow rules to change the network configuration.

Figure 4.8 illustrates the response of HACFlow against a network state change event. Note that the context definition in Figure 4.7 starts with the value "*safe*" as the default network state. Once HACFlow is notified of the alert, it changes the *object* attribute to "*vulnerable*". As a result, the policy becomes out of context and HACFlow reconfigures the network. Subsection 5.1.2.2 presents a performance analysis of the reaction of HACFlow against a vulnerability alert.

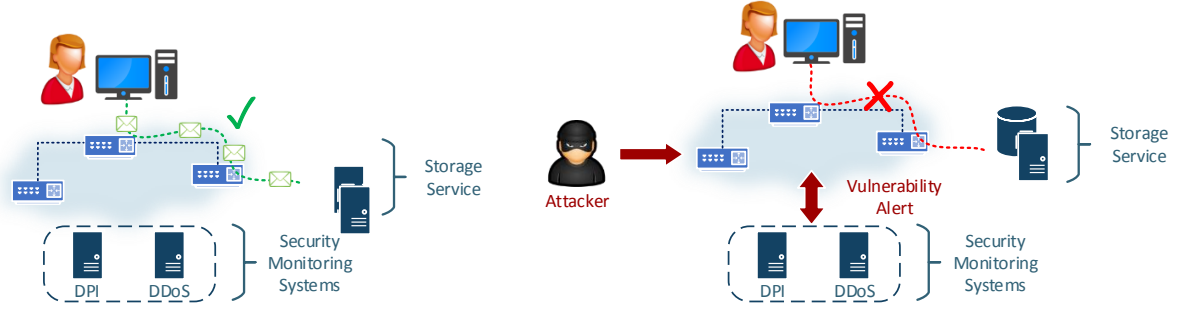


Figure 4.8: HACFlow reaction against a vulnerability alert.

4.1.5.2 Authentication Event

Another example of a networking event is an authentication. Once a network entity (user, host, service, among others) authenticates in the network by an SDN application, this application will use HACFlow as the authorization entity. Therefore, it sends to HACFlow an authentication alert. As a result, HACFlow returns a set of OpenFlow flow rules to be enforced in the network and enable the entity to access the network resources. Once HACFlow receives the authentication alert it will:

1. The OrBAC API filter entity's security rules.
2. The Policy Translator translates the security rule into OpenFlow flow rules.
3. HACFlow sends the OpenFlow flow rules to be enforced and change the network configuration.

Figure 4.9 depicts the reaction of HACFlow against a user authentication. As one may note, at first (step one) assuming that a user authenticates, a third party SDN application sends a request to HACFlow (step two). After that (step three), HACFlow gets, translates, and returns the user's security rules to such SDN application. Lastly (step four), these policies (translated to OpenFlow flow rules) are enforced on the network and the user get access or not to the network resources. Subsection 5.1.2.1 presents a performance analysis of the reaction of HACFlow against a network entity authentication.

4.1.6 Step-by-step: Role Delegation

Delegate and revoke roles is one of the main features of HACFlow. This feature allow network operators to delegate/revoke rights from a network entity to others without the need of creating new policies or assigning network entities to different roles. It is important to highlight that the role delegation feature is not supported by any SDN-based solution presented in related work section. In this subsection, we describe how a network operator can delegate and revoke roles to network entities.

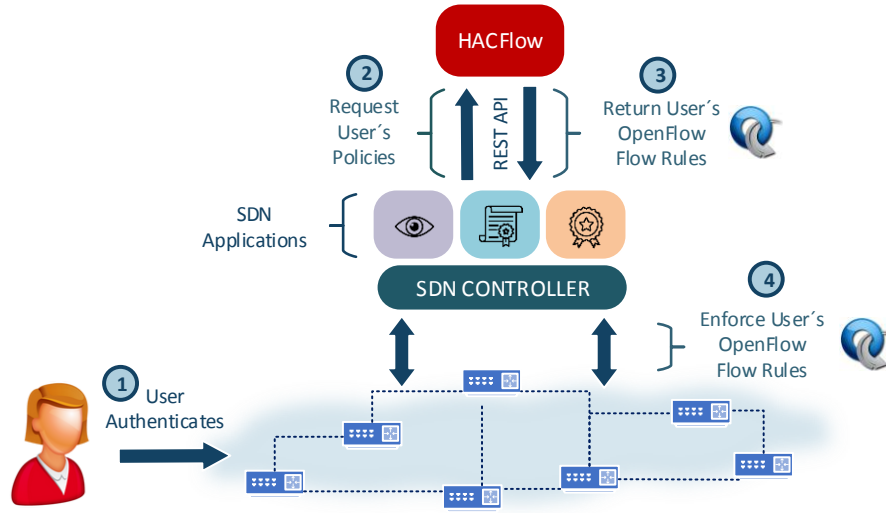


Figure 4.9: Reaction against a user authentication.

The HACFlow REST API provides methods to network operators delegate and revoke roles. Therefore, the network operator must simply execute the following single step:

1. Call the REST API delegate/revoke method passing the network entity and the role being granted/revoked.

Next, HACFlow automatically implements the role delegation or revocation executing the following steps:

1. The OrBAC API assigns/unassigns the network entity to a role.
2. The OrBAC API notifies the delegated/revoked security rules.
3. The Policy Translator translates the security rules into OpenFlow flow rules.
4. HACFlow sends the OpenFlow flow rules to be enforced and change the network configuration.

Once the delegation has been implemented, the network entity is able to access the network resources granted to it. The Subsection 5.1.4 presents a performance analysis of the role delegation/revocation in HACFlow.

4.2 Case Study: Applying HACFlow in a Smart City

In this subsection, we present how network operators can use HACFlow to govern rights to network entities through the implementation of network access control policies in a high-level and human-readable way. Besides, we demonstrate the HACFlow expressiveness to define many security policies as network configurations. Our case study is a smart city network environment

and we present a step-by-step from the operator's high-level goals to low-level OpenFlow flow rules. We show how these goals may be expressed as high-level security policies and their representations as switch-level rules.

4.2.1 Overview

The policy expressiveness is an important aspect that allows network operators to express many common networks' configurations. It aims to enable operators to define behaviors that describe their high-level goals, instead of define instructions that say how these goals must be implemented in the network [AOUADJ et al. \(2014\)](#).

Next, we demonstrate the HACFlow policy expressiveness on defining network access control rules. We will present a variety of network configuration examples implemented in HACFlow. These examples allow us to identify strengths and weaknesses/limitations of our high-level policy definition approach.

Our case study scenario consists of a smart city network controlled by an SDN controller. The city contains two smart services: the first is a **waste management system** used to optimize the trash collection routes and offer rubbish levels to citizen, and the second one is a **video surveillance system** for public safety composed of cameras around the city. Figure 4.10 depicts our case study example.

In this case study, we assume that all network entities (*citizen*, *cameras*, and *garbage*) are authenticated by an SDN application and this application uses HACFlow as the authorizing entity. In the next Subsection, we present the city administrator's high-level security goals to control the access of the network entities. Thinking on these security goals is the first step to define high-level security policies in HACFlow.

After that, in Subsection 4.2.3 we demonstrate how operators define network entities and group them to compose the *abstract* and *concrete* levels of the OrBAC model. Besides, we explain how *contexts* are defined to create static and dynamic security policies. Next, in Subsection 4.2.4 we demonstrate how these high-level goals are expressed as high-level security rules. Lastly, the Subsection 4.2.5 shows how these security policies are enforced in the network and their representation as OpenFlow flow rules.

4.2.2 Defining High-level Goals

In this subsection, we present the high-level goals of a city administrator that aims to govern the rights of each network entity. So, imagine that it wants to take control of **who** (which network entity) can access **what** (connected garbage, cameras, among others) **in which circumstances** (a day of a week, an hour of a day, and so on) each network resource. In that case, the city administrator defines the following **high-level goals**:

1. PolicyA: The citizen may verify any garbage bin fullness when it is not in maintenance.

2. PolicyB: The police must be able to access the live view service of all cameras any hour in any day of the week.
3. PolicyC: A garbage can exchange statistics information to other ones to take better decisions (machine-to-machine communication).
4. PolicyD: The citizen cannot access the live view camera service between 1 AM to 6 AM.

Once defined, these goals must be expressed as high-level security policies in HACFlow. The next two subsections detail how this can be achieved.

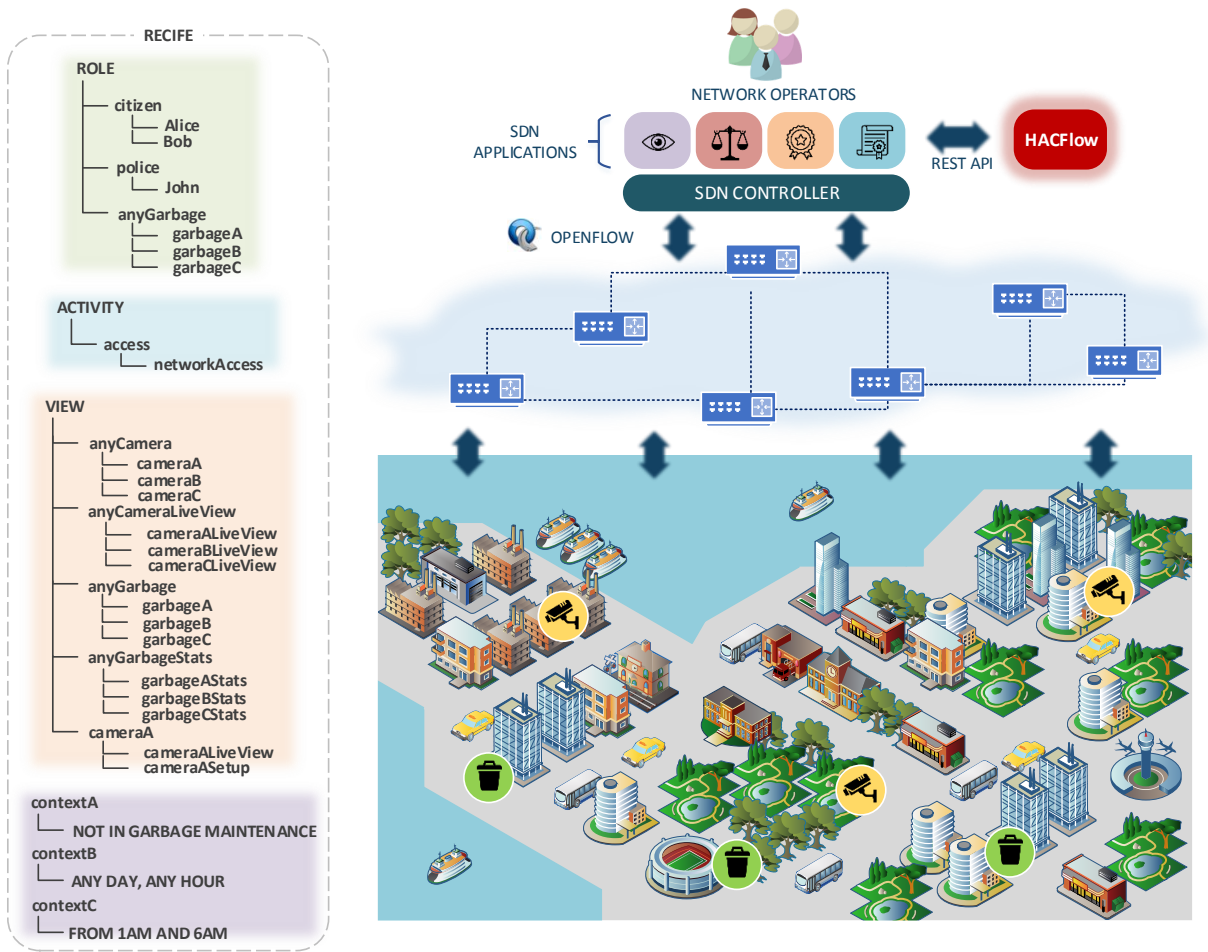


Figure 4.10: Smart City case study scenario.

4.2.3 Defining the Network Entities

As we explained in Subsection 4.1.3, the two first steps to create high-level security policies in HACFlow consist in defining the *abstract* and *concrete* levels of the OrBAC model, that means, create the *Organization*, *Role*, *Activity*, *View*, and *Context* predicates and create

the *Subject*, *Action*, and *Object* entities. The next two subsections detail how to create these *predicates* and *entities*.

4.2.3.1 The Abstract Level

As shown in Figure 4.10, the *abstract level* relies on: *Recife* as the **Organization**; *citizen*, *police*, and *anyGarbage* as the **Roles**; *access* as the **Activity**; *anyCamera*, *anyCameraLiveView*, *anyGarbage*, and *anyGarbageStats* as the **Views**. Figure 4.11 shows the creation of the *Organization* predicates using the MotOrBAC tool. The other ones are created in the same way.

Furthermore, we *Defined* the following three **Contexts**: *contextA* with the definition "*not in garbage maintenance*", *contextB* with the definition "*any day, any hour*", and *contextC* with the definition "*from 1AM to 6AM*". We point that we have a static context (*contextB*) and two dynamic contexts (*contextA* and *contextC*). Therefore, depending on which context a policy is linked to, we may have static and dynamic security rules. Figure 4.12 depicts the creation of a context and its definition in MotOrBAC.

Notice that *anyGarbage* is both, a *Role* and a *View*. This allows that the entities inside *anyGarbage* (*garbageA*, *garbageB*, and *garbageC*) have access to another network resources and make them accessible by others. For instance, operators may create an access control rule that allows the *garbageA* *Role* to communicates with other network entity (*garbageB* or *garbageC*) to optimize trash collection (*garbageA* is the source of the communication). On the other hand, using the *anyGarbage* *View*, operators may define a security rule that allows the *citizen* to consult rubbish level of *garbageA* (here, *garbageA* is the destination of the communication).

Figure 4.11: Creating the *Organization* predicate in MotOrBAC tool.

4.2.3.2 The Concrete Level

The *concrete level* is composed of the following **Subjects**: *Alice* and *Bob* empowered in *citizen* *Role*; *John* empowered in *police* *Role*; and *garbageA*, *garbageB*, and *garbageC* empowered in *anyGarbage* *Role*. The following **Action**: *networkAccess* considered in *access* *Activity*.

Lastly, we have the following **Objects**: *cameraA*, *cameraB*, and *cameraC* used in *anyCamera* *View*; *cameraALiveView*, *cameraBLiveView*, and *cameraCLiveView* used in *anyCam-*

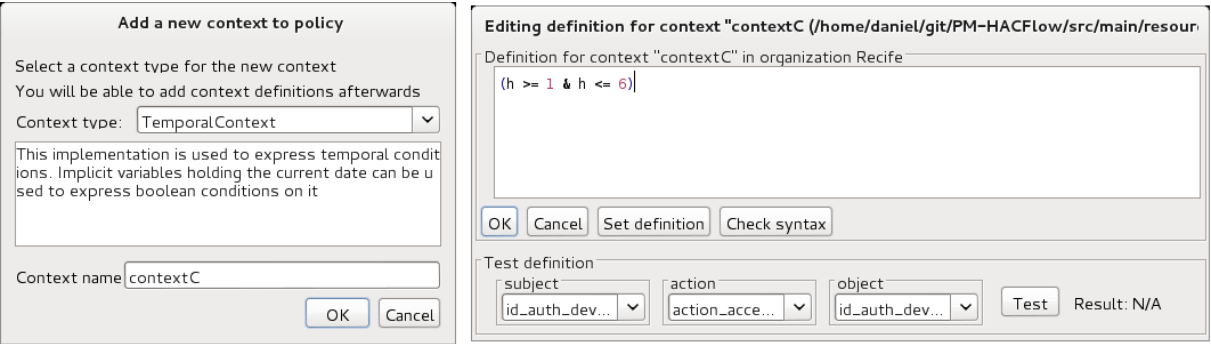


Figure 4.12: Creating a *context* and setting its *definition*.

eraLiveView View; *garbageA*, *garbageB*, and *garbageC* used in *anyGarbage View*; and the *garbageAStats*, *garbageBStats*, and *garbageCStats* used in *anyGarbageStats View*. Figure 4.13 shows the creation of a concrete entity in MotOrBAC.

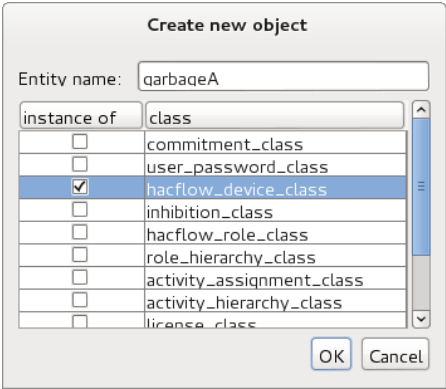


Figure 4.13: Creating the *garbageA* entity and assigning its *class definition*.

Note that while creating the network entity, we assigned its *class definition* (step three of Subsection 4.1.3). These classes were depicted in Figure 4.2 and were generated automatically by the *Policy Skeleton* module in HACFlow. Figure 4.14 shows how we filled the *garbageA* entity attributes in MotOrBAC.

| garbageA | Attribute | Value |
|----------|-----------|-------------------------|
| | IP_PROTO | TCP |
| | PORT | 8443 |
| | ID | id_auth_entity_garbageA |

Figure 4.14: Attribute values of the *garbageA* entity.

It is important to highlight that all network entities must have the *ID* attribute of the *class definition* filled with the certificate identifier used to authenticate these entities. Through this unique *ID*, the HACFlow filters the entity’s security rules and returns them to the SDN application to enforce the rules in the network. Besides, once a network entity authenticates, HACFlow requires additional information of this entity such as IP address, MAC address, connected switch, among others.

Furthermore, the *class definitions* allow operators to define more granular security rules. For example, in Figure 4.10 we have the *cameraA View* and inside it we have the *cameraALiveView* and *cameraASetup Objects*. These *Objects* represent two different services of *cameraA* and the access to them may require different privileges. Therefore, the *class definitions* for these services may look like the ones depicted in Figure 4.15.

| cameraALiveView | Attribute | Value | cameraASetup | Attribute | Value |
|-----------------|-----------|-----------------|--------------|-----------|-----------------|
| | IP_PROTO | TCP | | IP_PROTO | TCP |
| | PORT | 8776 | | PORT | 8777 |
| | ID | id_auth_cameraA | | ID | id_auth_cameraA |

Figure 4.15: Attribute values of two different services within the *cameraA* entity.

This way, network operators may create a security policy that allows the *police* to access the *cameraA*. In that case, the *police* will be able to access both *cameraALiveView* and *cameraASetup* services (coarse-grained policy). On the other hand, operators may define a fine-grained security rule that allows the *citizen* to have access to only the *cameraALiveView* service.

Other important aspect of our case study example regards the hierarchical structure of the network entities. This reduces the management effort once operators may control the access of a group of entities (i.e. *citizen*) to a group of other ones (i.e. *anyCameraLiveView*) creating only a single security policy. Therefore, operators do not need to manage each entity individually by creating many security policies to control their access.

4.2.4 Defining High-level Security Policies

Once the city administrator has defined his/her high-level goals and the network operator defined the abstract (*Roles*, *Activities*, *Views*, and *Contexts*) and concrete (*Subjects*, *Actions*, and *Objects*) levels as detailed in Subsection 4.2.3 and depicted in the left hand of Figure 4.10, the next step consists in linking them to compose permissions and prohibitions security rules. The following **high-level security policies** are the result of the high-level goals defined in Subsection 4.2.2.

1. PolicyA: *Permission(Recife, citizen, access, anyGarbage, contextA)*
2. PolicyB: *Permission(Recife, police, access, anyCameraLiveView, contextB)*
3. PolicyC: *Permission(Recife, anyGarbage, access, anyGarbageStats, contextB)*
4. PolicyD: *Prohibition(Recife, citizen, access, anyCameraLiveView, contextC)*

These high-level policies will infer a set of entity-specific security rules that will be translated into OpenFlow flow rules. Next, the switch-level rules will be enforced on the underlying smart city network. Figure 4.16 shows how a high-level security policies is created in the MotOrBAC tool.

Figure 4.16: Creating the *PolicyA* security rule.

Once created, those policies may be in conflict. In that case, the network operator must solve them to avoid malfunctions and policy inconsistencies. Therefore, priorities can be defined between the conflicting security policies.

4.2.5 Enforcing Security Policies in the Network

The last step to control the access of the network entities consists in enforcing the security rules in the network. Will be those rules that really will allow or deny the communication between the network entities. Next, we demonstrate the inferred OrBAC security rules from the high-level policy. Besides, we show the low-level OpenFlow flow rule representation of the first inferred security rule.

Abstract rule 1: *PolicyA: Permission(Recife, citizen, access, anyGarbage, contextA)*

Concrete rule 1.1: *Is_permitted(Alice, access, garbageA)*

Concrete rule 1.2: *Is_permitted(Alice, access, garbageB)*

Concrete rule 1.3: *Is_permitted(Alice, access, garbageC)*

Concrete rule 1.4: *Is_permitted(Bob, access, garbageA)*

Concrete rule 1.5: *Is_permitted(Bob, access, garbageB)*

Concrete rule 1.6: *Is_permitted(Bob, access, garbageC)*

As one may note, from a single high-level security policy we obtained six security rules. The hierarchical policy structure defined by the network operator explains this amount of security rules generated. When the user *Alice* authenticates in the network, HACFlow filters her security rules (1.1, 1.2, and 1.3) and then translates each one into low-level OpenFlow flow rules. Below, we describe how the **Concrete rule 1.1** looks like after translated. Note that *PolicyA* is a permission, therefore we have the incoming and outgoing flows, from the *Alice's* switch perspective.

OpenFlow flow rule 1.1 (output / ovs-ofctl): *cookie=0x29192000000 00000, duration=10.412s, table=0, n_packets=0, n_bytes=0, hard_age=240, idle_age=7, priority=55000,tcp,nw_src*

=10.50.2.1,nw_dst=10.50.2.17,tp_dst=8443 actions=NORMAL

OpenFlow flow rule 1.1 (input / ovs-ofctl) : *cookie=0x291920000000 00000, duration=10.742s, table=0, n_packets=0, n_bytes=0, hard_age=240, idle_age=7, priority=55000,tcp,nw_src=10.50.2.17,nw_dst=10.50.2.1,tp_src=8443 actions=NORMAL*

Here, we assume that once authenticated, the network entities get the following IP addresses: *Alice* (10.50.2.1), *garbageA* (10.50.2.17), *garbageB* (10.50.2.18), and *garbageC* (10.50.2.19). Besides, each *garbage* has a service (listening on port 8443) that provides the garbage fullness. Therefore, the **OpenFlow flow rule 1.1 (output)** means that all packets with source IP 10.50.2.1 (*Alice*), and destination IP 10.50.2.17 and port 8443 (*garbageA*) will be forwarded to *garbageA*.

Furthermore, as the **OpenFlow flow rule 1.1** is a permission, it will be enforced in the path from *Alice*'s switch to *garbageA*'s switch. If *PolicyA* were a prohibition policy, the flow rule would be needed to enforce only in *Alice*'s switch, dropping the packet at the source of the communication.

We also point that, as the *PolicyA* was defined with *contextA*, when a *garbage* needs a maintenance, the *PolicyA* will be out of context (becomes inactive) and HACFlow will automatically remove this flow rule from the network. As a result, *Alice* cannot consult the *garbage* fullness until the maintenance finishes.

4.3 Concluding Remarks

This Chapter presented the HACFlow framework. We showed how HACFlow overcomes the challenges, problems, and requirements presented previously. We justified the integration of the OrBAC model in our solution and described the role and interaction between the components of the architecture. A step-by-step description explained how to operate HACFlow to implement its main management tasks. We demonstrated the benefits of using HACFlow in a smart city scenario.

5

Evaluation and Comparison

In this chapter, we analyze the performance and scalability of the main management tasks provided by HACFlow. Then, we compare HACFlow against Frenetic [FOSTER et al. \(2011\)](#), FRESCO [SHIN et al. \(2013\)](#), and OpenSec [LARA; RAMAMURTHY \(2016\)](#).

5.1 HACFlow Performance Evaluation

In this section, we analyze the performance and scalability of the main management tasks provided by HACFlow. They are (1) reacting to network state changes and events (i.e. authentication event and vulnerability alert), (2) reacting to dynamic security policies, (3) delegating a role, and (4) converting high-level policies into low-level OpenFlow flow rules.

5.1.1 Scenario Description and Methodology

In this subsection, we describe our methodology and experimental setup used to conduct all experiments. At first, we configured our testbed with one physical host and three virtual machines. Next, once we have configured our environment, we developed a SDN application that uses HACFlow to control the access to the network resources. Lastly, we configured our experimental network topology.

Our testbed consists of a machine with Ubuntu 15.04 operating system containing an Intel(R) Core(TM) i7-3770 CPU 3.40GHz with eight cores and 16GB of RAM memory. On top of this physical machine, we used the hypervisor Oracle VirtualBox ¹ version 5.0.0 to create and configure three virtual machines.

The first virtual machine refers to the HACFlow framework. HACFlow was developed using the Spring Framework version 4.3.6 and was deployed in Tomcat ² version 8 and the machine was configured with 1 processor, 2GB of RAM, and Ubuntu 15.04. The second one is the Mininet network emulator version 2.2 with Open vSwitch 2.4.0 ³. It was configured with 1

¹<https://www.virtualbox.org/>

²<http://tomcat.apache.org/>

³<http://openvswitch.org/>

processor, 1GB of RAM, and Ubuntu 14.04. Lastly, the third one refers to the HP VAN SDN controller version 2.5.15. It was configured with 1 processor with 4 cores, 10GB of RAM, and Ubuntu 12.04.

Next, we developed a prototype SDN application for the HP VAN SDN controller that interacts with HACFlow through its REST API. This application uses HACFlow as the authorization entity to control the access to the network by enforcing the OpenFlow security rules provided by HACFlow.

The network topology used in all experiments consists of one SDN controller, three OpenFlow-enabled switches, and two hosts (a user and a server). This topology was emulated using the Mininet ⁴ tool.

Lastly, all experiments were executed 64 times, except the last one (Subsection 5.1.5) that we run 256 times. All the experimental results presented in the following tables include the mean and standard deviation to perform each task. Figure 5.1 depicts our overall experimental setup.

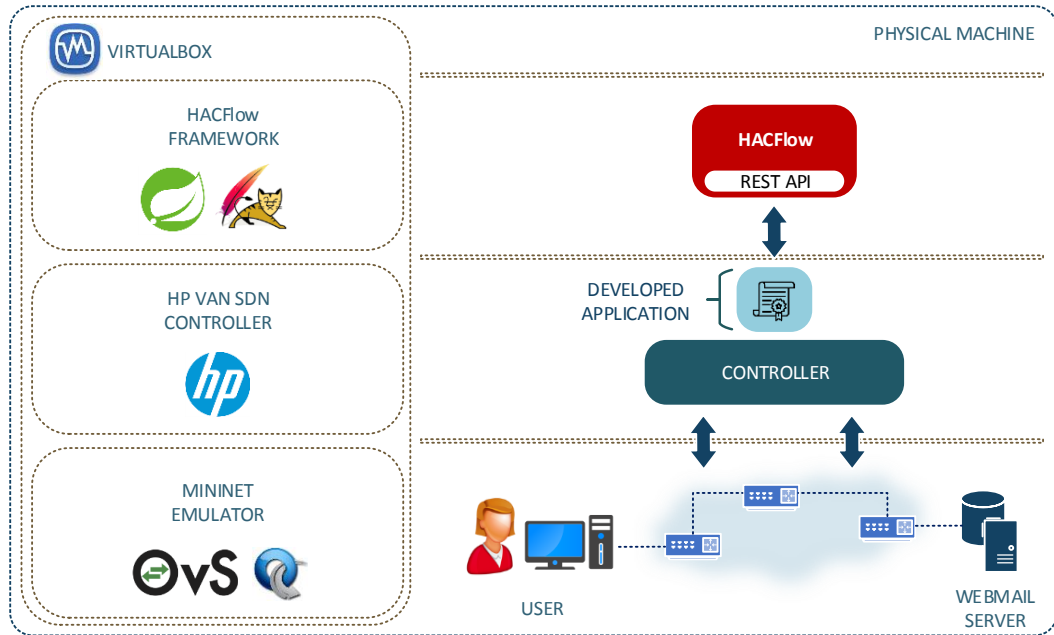


Figure 5.1: Experimental setup.

5.1.2 Network State Changes and Events

In this subsection, we analyze the reaction of HACFlow against network state changes and events. We present two experiments, the first refers to the required time for HACFlow process the security policies of an authenticated user. The second refers to the time required for HACFlow to block the access of a user to a server after a vulnerability alert is detected.

⁴<http://mininet.org/>

5.1.2.1 Authentication Event

Once a network entity (user, server, service, among others) authenticates on the network, the next step is to get access to a variety of network resources according to the predefined security policies. Figure 5.2 depicts these steps.

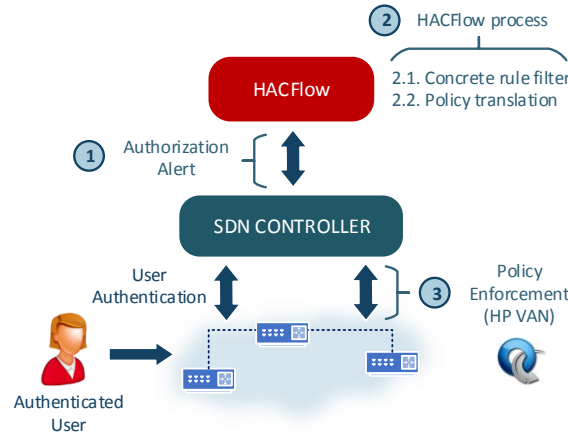


Figure 5.2: Steps to react to an authentication.

When a third party SDN application authenticates a user, (step 1) this application notifies HACFlow to obtain the user's rules. Next, (step 2) HACFlow processes the user's rules. At first, (step 2.1) it filters the security rules and extracts the low-level data (set in *class definitions*), and then (step 2.2) translates the OrBAC security rules into OpenFlow flow rules. Lastly, (step 3) the third party application enforces the user's policies and the user is able to access the network resources. The required times to execute these steps are in Table 5.1 and Table 5.2.

Table 5.1: Network Entity Authentication.

| Total time | 1. Authorization alert (synchronous call) | 2. HACFlow process | 3. Policy Enforcement (HP VAN) |
|-------------|--|--------------------|--------------------------------------|
| 283.0011 ms | 19.0581 ms (6.7%) (s.d 5.361) | 4.9807 ms (1.8%) | 258.9622 ms (91.5%) (s.d 17.3656) |

Table 5.2: HACFlow process: authentication event.

| 2. HACFlow process | |
|---------------------------|--------------------------|
| 2.1. Security rule filter | 2.2. Policy translation |
| 4.9373 ms (s.d 1.719) | 0.0434 ms (s.d 0.008) |

We highlight that this delay only occurs at the first time the network entity authenticates. Besides, HACFlow presents the lowest time (4.98 ms in average, or 1.8% of the total time) when compared to the other tasks in the whole process. In step three, the HP VAN controller enforces six OpenFlow flow rules (three switches, each one with the in/out flows) from a single high-level permission security policy. As it is a permission rule, we need two flows (in/out) in each switch.

5.1.2.2 Vulnerability Alert

HACFlow is also able to react to network state changes according to the needs defined by the network operator. Once HACFlow receives a new alert it automatically reconfigures the network to comply with the configured security policies. Next, we describe all the steps of the whole process and Figure 5.3 depicts them.

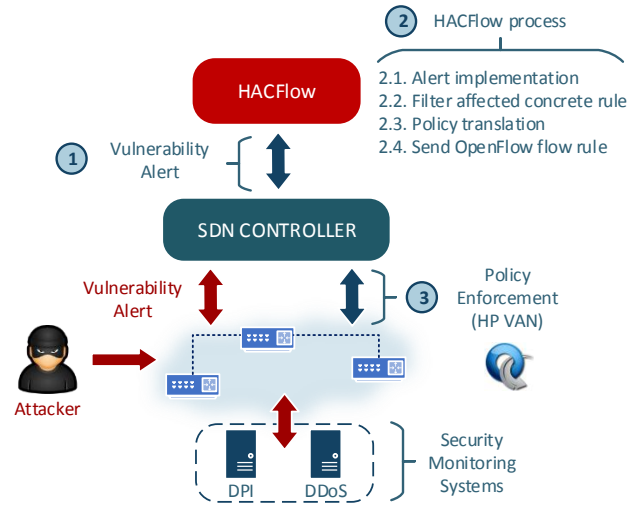


Figure 5.3: Steps to react to a vulnerability alert.

When a network alert is triggered by a security monitoring system (IDS, DPI, DDoS, among others) the (step 1) SDN application notifies HACFlow to reconfigure the network. Then, (step 2) HACFlow processes the new alert, which includes the following four steps: (step 2.1) implement the alert by changing its state, (step 2.2) filter the security rules affected by this alert, (step 2.3) translate them to OpenFlow flow rules, and (step 2.4) send the flow rules to the SDN application. Lastly, (step 3) the application reconfigures the network enforcing the flow rules. Table 5.3 and Table 5.4 present the results.

Table 5.3: Network Vulnerability Alert.

| Total time | 1. Vulnerability alert (asynchronous call) | 2. HACFlow process | 3. Policy Enforcement (HP VAN) |
|------------|---|--------------------|------------------------------------|
| 25.6325 ms | 1.8369 ms (7.2%) (s.d 0.2931) | 8.2408 ms (32.1%) | 15.5548 ms (60.7%) (s.d 1.6153) |

Table 5.4: HACFlow process: vulnerability alert.

| 2. HACFlow process | | | |
|------------------------------|---------------------------------------|--------------------------|---|
| 2.1. Alert implementation | 2.2. Filter affected security rule | 2.3. Policy translation | 2.4. Send OpenFlow flow rule (asynchronous call) |
| 0.168 ms (s.d 0.073) | 6.4918 ms (s.d 1.4716) | 0.0518 ms (s.d 0.009) | 1.5292 ms (s.d 0.237) |

From the results, we highlight that in this experiment the (3) policy enforcement is a flow removal (i.e. once the network is unsafe, the user loses his/her access). Due to that, the HP VAN requires less time to enforce (remove) the flow rules from the switches, when compared to the previous experiment (that add flow rules). Besides, we point that HACFlow needed 8.24 ms in average (32.1% of the total time) to react to a vulnerability alert, resulting in a fast reaction if compared to a manual reconfiguration (human-intervention), commonly in traditional networks.

5.1.3 Dynamic Security Policy

Dynamic security policies are context-aware policies that may have their state changed (active or inactive) depending on some circumstances (day of a week, an hour of a day, and so on). It can change at any time and HACFlow should be able to automatically react to these changes, not requiring any manual and per-device configuration on network devices. Next, we describe the steps required to HACFlow react to a dynamic policy. Figure 5.4 depicts those steps.

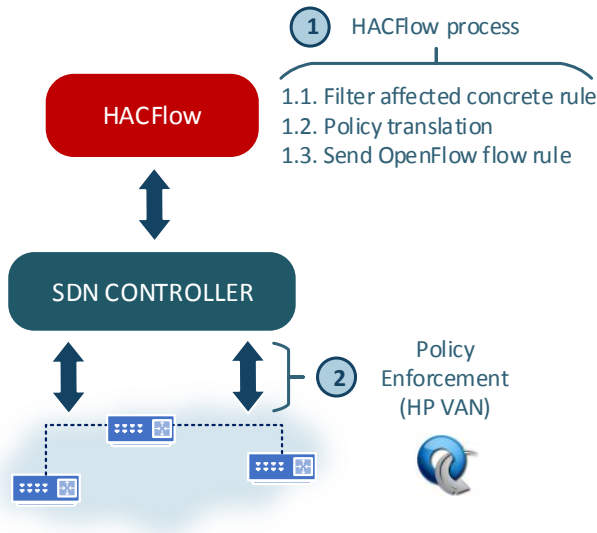


Figure 5.4: Steps to react to a dynamic policy.

Changes in a dynamic police are detected by the OrBAC component inside HACFlow. Therefore, once detected (step 1) HACFlow will process these changes by (step 1.1) filtering the affected security rules through the OrBAC API, (step 1.2) converting them into OpenFlow flow rules, and (step 1.3) sending the flow rule to a SDN application. Lastly, (step 2) this application will enforce all flow rules to reconfigure the network according to the new circumstances. The results of this experiment are presented in Table 5.5 and Table 5.6.

In this experiment, we simulate a security policy being out of context, that means, being out of a circumstance or restriction imposed by the network operator, like out of working hours, for example. In that case, the enforcement in step two results in the removal of flows. The results point out that HACFlow required 7.57 ms in average (30.9% of the total time) to react to a single dynamic security policy.

Table 5.5: Dynamic Security Policy.

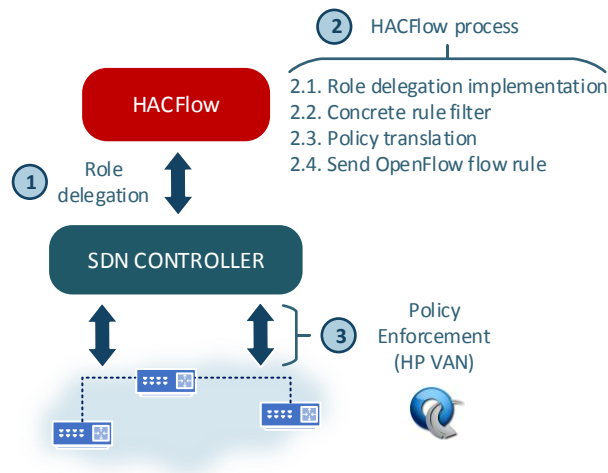
| Total time | 1. HACFlow process | 2. Policy Enforcement (HP VAN) |
|------------|--------------------|-----------------------------------|
| 24.513 ms | 7.5776 ms (30.9%) | 16.9354 ms (69.1%) (s.d 1.839) |

Table 5.6: HACFlow process: dynamic policy.

| 1.1. Filter affected security rule | 1. HACFlow process | | 1.3. Send OpenFlow flow rule (asynchronous call) |
|------------------------------------|-------------------------|--|--|
| | 1.2. Policy translation | | |
| 5.9175 ms (s.d 1.329) | 0.058 ms (s.d 0.019) | | 1.6021 ms (s.d 0.284) |

5.1.4 Role Delegation

Role delegation consists in granting the rights of a role to a network entity. Once a delegation occurs, HACFlow assigns the new security policies to the network entity and (if this entity was authenticated) sends the rules to be enforced in the network. Figure 5.5 depicts the whole process.

**Figure 5.5:** Steps to delegate a role.

Through the HACFlow REST API, the operator (step 1) delegates a role. Then, (step 2) HACFlow processes the delegation according to the following four steps: (step 2.1) implements the delegation by linking the role to a network entity; next (step 2.2) the assigned security rules are filtered; then (step 2.3) the rules are translated into OpenFlow flow rules; lastly (step 2.4) the flow rules are sent to the SDN application. After that, (step 3) this application enforces the rules in the OpenFlow switches. Table 5.7 and Table 5.8 present the results.

According to the results, if we compare all of the management tasks provided by HACFlow, the role delegation is the one that requires a longer time. Despite this, requir-

ing 119.45 ms in average (30.8% of the total time) to delegate a single role linked to a single security policy is still a fast time if you consider the task complexity and if you compare to manual role delegations.

Furthermore, from the results of the deeper analysis of step two (Table 5.8), we point that the most of the time is required by the OrBAC API to implement the role delegation. Besides, the whole process occurs in less than a half of a second, that means, once the network operator delegates a role to a user, the user will have to wait for about 387.3 ms in average to access the assigned network resources. We consider that waiting for about 387.3 ms in average is a fast time once compared to a number of manual reconfigurations that the network operator must implement in traditional network solutions.

Table 5.7: Role delegation.

| Total time | 1. Role delegation (asynchronous call) | 2. HACFlow process | 3. Policy Enforcement (HP VAN) |
|------------|---|---------------------|--------------------------------------|
| 387.365 ms | 1.8535 ms (0.5%) (s.d 0.2199) | 119.4578 ms (30.8%) | 266.0537 ms (68.7%) (s.d 26.0062) |

Table 5.8: HACFlow process: role delegation.

| 2. HACFlow process | | | |
|--|------------------------------|--------------------------|--|
| 2.1. Role delegation implementation | 2.2. Security rule filter | 2.3. Policy translation | 2.4. Send OpenFlow flow rule (asynchronous) |
| 106.0012 ms (s.d 22.724) | 7.06 ms (s.d 2.3778) | 0.0568 ms (s.d 0.045) | 1.7035 ms (s.d 0.291) |

5.1.5 High-level to Low-level Policy Inference

In this subsection, we analyze the scalability of HACFlow to infer the low-level OpenFlow flow rules from the high-level security policies. As you may notice, all previous management tasks required filtering and translating the security rules. Therefore, the policy inference is an important feature provided by HACFlow.

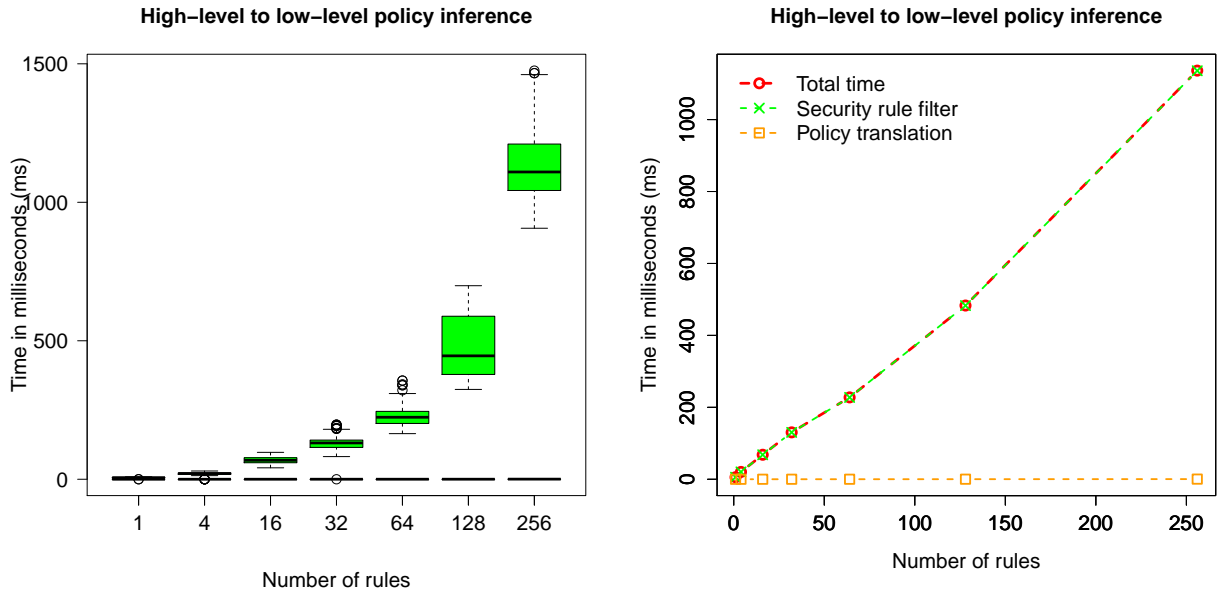
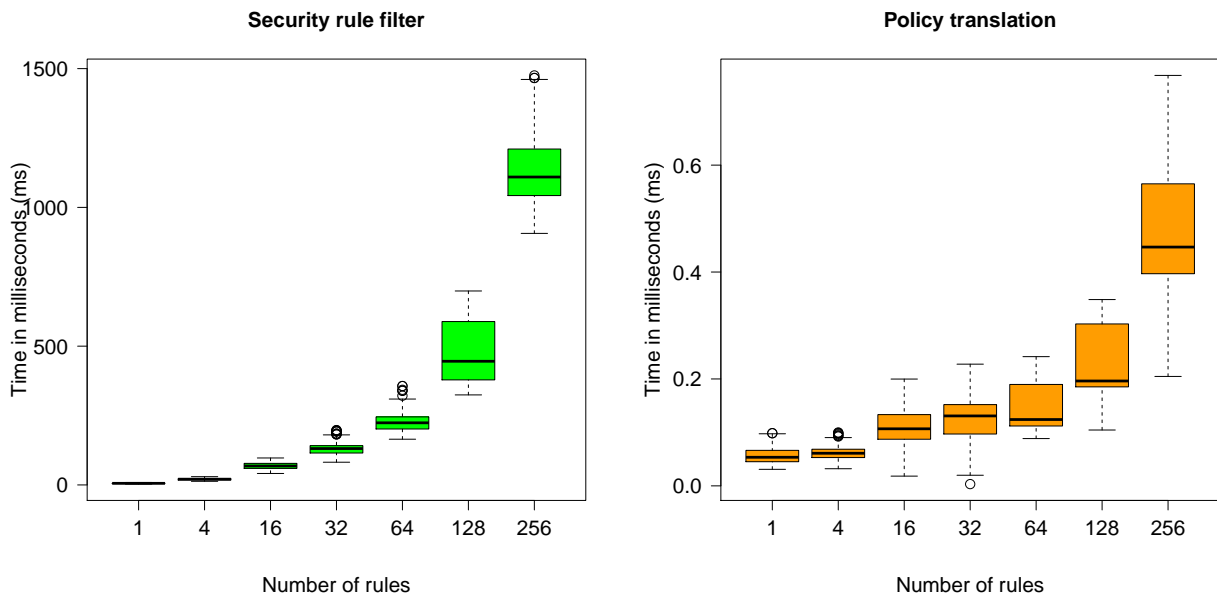
In this analysis, the HACFlow framework infers 1, 4, 16, 32, 64, 128, and 256 high-level security rules into low-level OpenFlow flow rules. We considered using these values once they represent a reasonable number of rules that a single user may have in an environment like a company, a university, a hospital, and so on.

For a deeper analysis, we divided the inference process into two steps. The first (*1. Security rule filter*) refers to the filtering of rules through the OrBAC API and the extraction of low-level data inside the *class definition* of the network entities.

The second one (*2. Policy translation*) refers to the translation process, that means, the process that obtains the OpenFlow flow rules from the rules extracted in the previous step. Table 5.9 presents the scalability results. Figure 5.6 and 5.7 plots them.

Table 5.9: High-level to low-level policy inference.

| Number of rules | Total time | 1. Security rule filter | 2. Policy translation |
|-----------------|--------------|-----------------------------|------------------------|
| 1 | 5.7581 ms | 5.7014 ms (s.d 1.2752) | 0.0567 ms (s.d 0.0155) |
| 4 | 20.4588 ms | 20.3973 ms (s.d 3.7657) | 0.0615 ms (s.d 0.0127) |
| 16 | 68.2220 ms | 68.1110 ms (s.d 12.2247) | 0.1110 ms (s.d 0.0346) |
| 32 | 130.4807 ms | 130.3544 ms (s.d 22.8665) | 0.1263 ms (s.d 0.0405) |
| 64 | 227.6918 ms | 227.5458 ms (s.d 36.4297) | 0.1460 ms (s.d 0.0401) |
| 128 | 483.0941 ms | 482.8634 ms (s.d 114.4093) | 0.2307 ms (s.d 0.0626) |
| 256 | 1136.2124 ms | 1135.7333 ms (s.d 132.8165) | 0.4791 ms (s.d 0.1283) |

**Figure 5.6:** Scalability: high-level to low-level policy inference.**Figure 5.7:** Policy inference steps: security rule filter and policy translation.

According to the results, we highlight that HACFlow needed only 0.4791 ms in average to translate 256 security rules. Furthermore, the whole process required about 1.1 seconds in average to infer the OpenFlow flow rules from the high-level security policies.

Therefore, supposing that, if a user that works in a big company has 256 security rules that control its access to many network resources, when this user authenticates in the network he/she will have to wait about 1.1 second plus the time the SDN controller needs to enforce those rules. It is not so much time if you consider the overall task complexity.

The left side of Figure 5.6 presents a boxplot with both times and at the right side it presents them in addition to a third one as the sum of them. Furthermore, Figure 5.7 separately plots the security rule filter (left side) and the policy translation times (right side). Note that they are in different scales, one goes from 0 to 1500 ms, while the another one goes from 0 to 0.6 ms.

5.1.6 Discussion

The results of the experiments show that HACFlow requires a time in the order of milliseconds to execute its main management tasks (reacting to network state changes and events, reacting to a dynamic security policy, and delegating a role). Furthermore, the results of the high-level to low-level policy inference show that HACFlow required 1.1 seconds in average to translate a considerable number of policies to a single network entity at a time.

Overall, results show that using HACFlow significantly reduce the effort to implement a variety of network management tasks. Besides, HACFlow is faster and less prone to errors if compared to manual and per-device configurations.

It is important to highlight that the required time for the HP VAN SDN controller to enforce the security policies consists in calculating paths, enforcing six OpenFlow flow rules (three switches, each one with the in/out flows), and using a check mechanism to enforce the rules. Besides, this time is controller-specific and HACFlow does not affect it.

5.2 Comparison Against Existing Solutions

In this section, we compare HACFlow against the three related work described in Section 3.3. They are Frenetic [FOSTER et al. \(2011\)](#), FRESCO [SHIN et al. \(2013\)](#), and OpenSec [LARA; RAMAMURTHY \(2016\)](#). We present a qualitative and quantitative analysis.

5.2.1 Overview

In the following subsections, we compare HACFlow against Frenetic, FRESCO, and OpenSec. We performed qualitative and quantitative comparison analysis. Subsection 5.2.2 presents the qualitative analysis, which includes: *i*) the features provided by each SDN-based solution and *ii*) the syntax simplicity to define high-level security policies. In Subsection 5.2.3 we present the quantitative analysis, that compares *i*) the required time to translate high-level

policies into low-level OpenFlow flow rules and *ii*) the required time to react to network state changes and events. Lastly, Subsection 5.2.4 discusses the overall comparison findings.

Table 5.10 summarizes the comparisons presented in the remainder of this section. It individually details in which aspects we compare HACFlow against Frenetic, FRESCO, and OpenSec.

Table 5.10: Summary of the comparison analysis.

| Comparing HACFlow against related work | | | | |
|--|--------------------|-------------------|--------------------|----------------------|
| Candidates | Qualitative | | Quantitative | |
| | Framework features | Syntax Simplicity | Policy Translation | Event Reaction Delay |
| Frenetic | ✗ | ✗ | ✗ | |
| FRESCO | ✗ | ✗ | | |
| OpenSec | ✗ | ✗ | ✗ | ✗ |
| HACFlow | ✗ | ✗ | ✗ | ✗ |

5.2.2 Qualitative Analysis

In this subsection, we present a qualitative comparison of HACFlow against Frenetic, FRESCO, and OpenSec. We analyze the main features provided by each approach and the syntax simplicity to create high-level security policies in HACFlow, Frenetic and FRESCO.

5.2.2.1 Framework Features

Next, we present the main management tasks provided by each SDN-based solution. Besides, we explain the role of each one. Table 5.11 shows the comparison results.

1. **High-level security policy definition:** allows operators to define policies in a high-level way, using high-level names (like users, hosts, services, and so on) instead of IP address, MAC address, among others. Operators do not worry about how these security policies will be implemented in the network;
2. **Hierarchical policies:** permits operators to organize the network entities hierarchically to decrease the management workload (entities are grouped, instead of managing them individually);
3. **Conflict detection and resolution:** lets operators to define conflict-free security policies by providing mechanisms to detect and solve conflicts avoiding inconsistencies and possible vulnerabilities;
4. **Definition of dynamic security policies:** allows operators to define policies according to circumstances like network state, an hour of a day, a day of a week, and so on.;

5. **Reaction to network changes and events:** provides mechanisms to automatically react to network changes and events according to operator's configurations;
6. **Security policy delegation:** permits operators to delegate rights from a network entity to others;
7. **High-level to low-level policy translation:** provide mechanisms to translate high-level policies defined by a network operator into low-level OpenFlow flow rules;

Table 5.11: Features Implemented by Different SDN-based NAC Solutions.

| Features | Frenetic | FRESCO | OpenSec | HACFlow |
|---|----------|--------|---------|---------|
| 1. High-level security policy definition | ✓ | ✓ | ✓ | ✓ |
| 2. Hierarchical policies | | | ✓ | ✓ |
| 3. Conflict detection and resolution | ✓ | ✓ | ✓ | ✓ |
| 4. Definition of dynamic security policies | | ✓ | | ✓ |
| 5. Reaction to network changes and events | ✓ | ✓ | ✓ | ✓ |
| 6. Security policy delegation | | | | ✓ |
| 7. High-level to low-level policy translation | ✓ | ✓ | ✓ | ✓ |

We highlight that all approaches have in common the possibility to create high-level security policies and translate them into OpenFlow flow rules. Also, they are able to react to network events and to detect and solve conflicting policies.

HACFlow is the only one that allows operators to delegate roles to network entities. Besides, HACFlow and OpenSec allow the definition of hierarchical policies.

5.2.2.2 Syntax Simplicity

We compare the syntax simplicity to define high-level policies in Frenetic, FRESCO, OpenSec, and HACFlow. Suppose that the network operator would like to define a policy that *"blocks all TCP traffic from user Bob (192.168.1.22) to the webMail service (IP 192.168.1.33 and port 8090)"*. Figure 5.8 shows the syntax to create this policy in each one.

In Frenetic, the variable *p1* and *p2* are a pattern that describes a packet (such as OpenFlow match fields), and *a1* is the action. The *install* function sends flow rules to the switches that will apply an action to a packet matching the given pattern. This flow rule has the *DEFAULT* priority level. In this example, the *install* function sends to three switches in the network a flow rule that drops packets with this *pattern* (*p1*{*NW_ADDR*:192.168.1.22} and *p2*{*NW_ADDR*:192.168.1.33, *TP_DST*:8090}).

In FRESCO, the operator defines a *function* passing the number of parameters (*#input*) and (*#output*). The variable *type* denotes a FRESCO module and *event* denotes events delivered to a module. The *input* variable is the input for a module (IP address, port, among others), the *output* expresses the output of the module, the *parameter* expresses the input values (such as

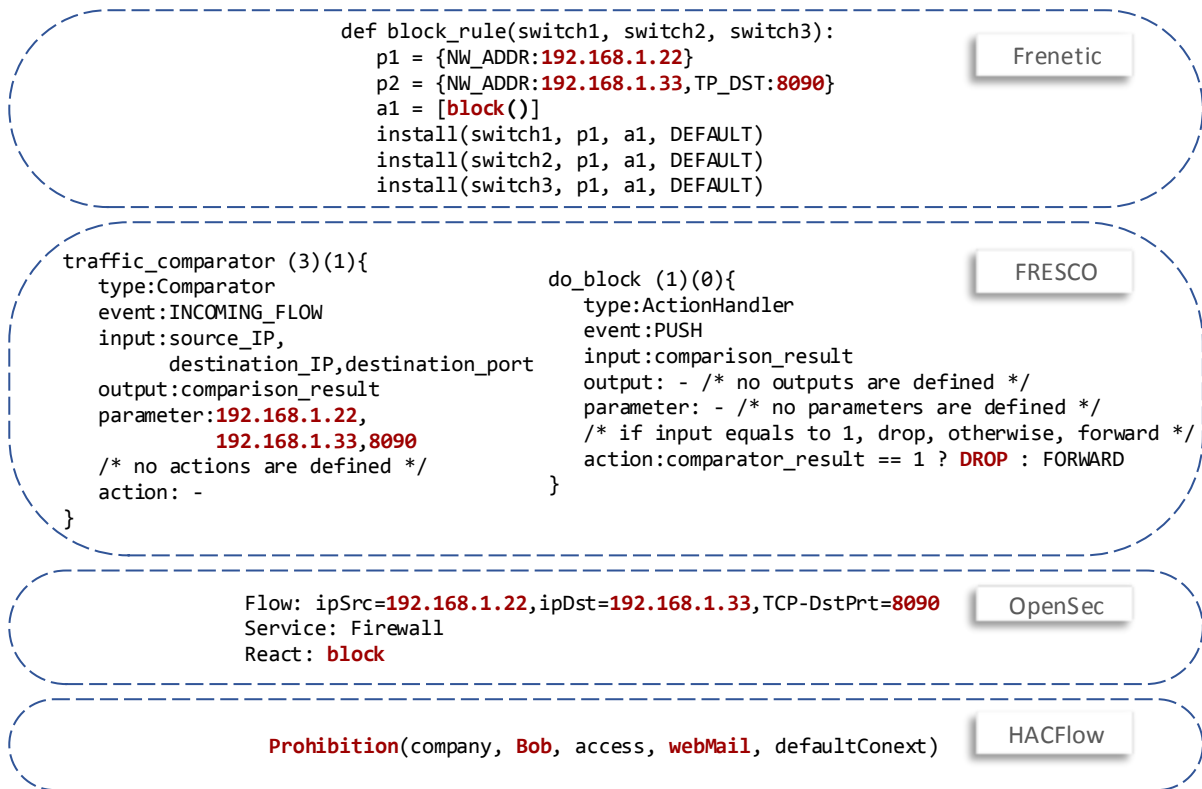


Figure 5.8: Frenetic, FRESKO, OpenSec, and HACFlow syntax comparison to create a policy.

192.168.1.22, 192.168.1.33, and 8090), and the *action* means an action performed in a module (DROP, FORWARD, REDIRECT, MIRROR, and QUARANTINE).

In the FRESKO script example, the *traffic_comparator* function will compare the *source_IP* and *destination_port* of the incoming packet, if a match occurs this function will *output* the value 1 as the *comparison_result*. This value is the *input* for the *do_block* function. The *action* verifies the condition and gets the *DROP* action. Lastly, the *event* named *PUSH*, makes the module send a flow rule to the switch to drop packets.

Lastly, the OpenSec syntax relies on the *flow*, *service*, and *react* fields. The *flow* field regards to the OpenFlow matching fields (such as *ipSrc*, *ipDst*, *TCP-DstPrt*, among others) to describe a flow. The *service* is a security service (like a firewall) that the flow rule must be rerouted to. Finally, the *react* field determines how to react (*alert*, *quarantine*, or *block*) against a flow matching or a malicious content reported by the security service.

Differently from Frenetic, FRESKO and HACFlow, the OpenSec does not block traffic on demand. But, it is able to match a flow pattern and reroute this flow to a firewall. Next, this firewall will be responsible for allowing or denying the access of *Bob* to the *webMail* service.

From this analysis, we point out that HACFlow presents the simplest syntax to create a high-level security policy. HACFlow uses high-level names such as *Bob* and *webMail*, instead of *IP addresses* and *port* (as in Frenetic, FRESKO, and OpenSec) to represent the network entities when defining a policy. In this example, we consider that the network operator has executed the

two first steps (described in Subsection 4.1.3) that consist in defining the *abstract* and *concrete* entities of the OrBAC model, and assigning the class definitions.

Despite Frenetic, FRESCO, and OpenSec allow operators to define policies in a high-level way, both still use some low-level data in the policy definition such as *IP address* and *port number*. HACFlow goes a step further and provides a higher level of abstraction when compared to Frenetic and FRESCO. HACFlow obtains those low-level data once a network entity authenticates (IP address, MAC address, connected switch, among others), and when registering network services like the webMail (port number and communication protocol).

Furthermore, in Frenetic operators have to say in which network switches the security rule must be implemented (in this example, we considered a network with three switches). Differently, in HACFlow operators only define the source (*Bob*) and destination (*webMail*) of the communication and the SDN controller decides in which switches this rule must be enforced.

5.2.3 Quantitative Analysis

In this subsection, we provide a quantitative comparison analysis of HACFlow against Frenetic and OpenSec. This analysis consists in measuring the required time to each one translate a security policy into switch-level rules. We also compare HACFlow against OpenSec regarding the time to react to a networking event. We point that, in both experiments, we compare HACFlow against the results provided by OpenSec.

Once we can not get the OpenSec source code to run it in our environment, we needed to create a virtual machine with a configuration as similar as possible to the one used in OpenSec. In OpenSec authors used an Intel Xeon X5650 2.67GHz to conduct their experiments. As we do not have a machine with this processor, we configured a virtual machine with an Intel Core i7-3632QM 2.20GHz (single-core), 2GB of RAM, and Ubuntu 15.04.

It is important to highlight that we used a processor with similar performance (Core i7-3632QM) compared to the one used in OpenSec (Xeon X5650). Figure 5.9 shows a comparison provided by CPUBoss⁵. From this CPU analysis (right side of Figure 5.9), we point out that our processor (Core i7-3632QM) is slightly lower than the Xeon X5650 when using a single-core.

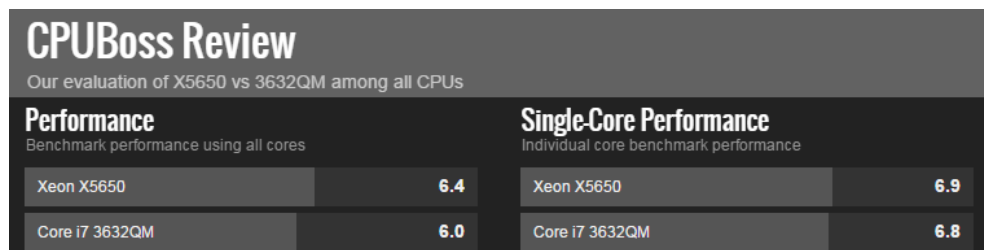


Figure 5.9: CPU performance comparison provided by CPUBoss.

⁵<http://cpuboss.com/cpus/Intel-Xeon-X5650-vs-Intel-Core-i7-3632QM-BGA1224>

5.2.3.1 Policy Translation

We compare HACFlow against Frenetic and OpenSec frameworks regarding the required time to translate a high-level security policy into low-level OpenFlow flow rule. We do not compare against FRESKO once it does not provide any policy translation analysis. Besides, we do not have access to the FRESKO source code project to analyze it in our environment.

The autonomic policy translation is an important feature provided by these frameworks. It allows network operators to define high-level goals without taking care of how they will be implemented in the network.

Differently of OpenSec, we can get access to the Frenetic source code project. We run Frenetic in a virtual machine with the same configuration described at the beginning of the Subsection 5.2.3. To conduct this experiment, we needed to understand the Frenetic code to locate and insert a time stamp inside the translating function, and recompile it. Also, we needed to learn how to create a high-level security policy using the Frenetic syntax.

As we are analyzing the policy translation time, we do not need to configure a network topology. The experiment was executed 256 times and the results are the mean with their respective standard deviation. Table 5.12 presents the results of this analysis. We point that HACFlow and OpenSec required similar times to translate a single security rule and Frenetic required a lower time. Despite this, HACFlow implements more features than Frenetic and OpenSec.

Table 5.12: High-level to low-level policy translation for a single rule.

| Translating a security policy into OpenFlow flow rule | | |
|---|--------------------------|------------------------|
| HACFlow | Frenetic | OpenSec |
| 0.0748 ms (s.d 0.005) | 0.0632 ms (s.d 0.008) | 0.07 ms (s.d 0.002) |

5.2.3.2 Event Reaction Delay

Lastly, we compare HACFlow against OpenSec regarding how long each one requires to react to a network state change and event. We point out that neither Frenetic nor FRESKO provide an analysis of reacting to network events, so we do not compare HACFlow against them.

The autonomic reaction against the dynamic nature of the network is one of the main features provided by these frameworks. This feature mitigates the management efforts and misconfigurations.

To conduct this experiment, we used the same virtual machine configuration described previously. We compare our results with the results provided by OpenSec. The experiment was run 256 times and our results represent the mean with their respective standard deviation. The Table 5.13 shows those results.

Table 5.13: Required time to HACFlow and OpenSec react to a networking event.

| Reaction against network state changes and events | |
|---|---------|
| HACFlow | OpenSec |
| 8.5537 ms (s.d 1.4782) | 8.1 ms |

In this experiment, the reaction time includes the moment that the framework receives the alert until it returns the OpenFlow flow rules to reconfigure the network. That means, the reaction time does not include the time needed to enforce the flow rules in the network, once this time is controller-specific. Therefore, we do not configure a network topology to conduct the experiments.

According to the results of this analysis, we conclude that OpenSec required a lower time (8.1 ms in average) to react against network events. Despite this, HACFlow provides more management features than OpenSec.

5.2.4 Discussion

In this section, we made a qualitative (Section 5.2.2) and quantitative (Section 5.2.3) comparison of HACFlow framework against Frenetic, FRESCO, and OpenSec. Overall, the qualitative analysis shows that HACFlow provides most of the management tasks and offers a simpler syntax to define high-level security policies. Besides, HACFlow is the only one that allows to delegate and revoke roles. On the other hand, the quantitative analysis demonstrates that the results of the required time to HACFlow translate a high-level policy into an OpenFlow flow rule is higher than Frenetic and OpenSec. Furthermore, the time to HACFlow react to network events is higher than the results achieved by OpenSec. Despite this, HACFlow implements more features than Frenetic and OpenSec.

6

Conclusion

Network access control (NAC) management is a critical task. Misconfigurations may result in vulnerabilities that may compromise the overall network security. Current approaches in traditional networks are inflexible and require per-device and vendor-specific configurations, being error-prone. The SDN paradigm overcomes architectural problems of traditional networks and offers new opportunities to manage the network. Despite this, access control management remains a challenge.

This work had as the main goals simplify and automate the NAC management in SDN environments. We achieved this goal by proposing HACFlow, a novel SDN framework based on OrBAC model. We demonstrated that OrBAC and OpenFlow are a powerful combination that allow network operators to define security policies in a high-level, fine-grained, and human-readable way.

HACFlow also provides mechanisms to translate high-level security policies into low-level OpenFlow flow rules, hiding the network configuration complexities from operators. HACFlow allows to create dynamic and conflict-free security rules and is able to automatically react to network state changes and events. Delegate and revoke roles is other main feature provided by HACFlow.

We presented a step-by-step on how to operate the management tasks provided in HACFlow. Our study case demonstrated the benefits of using HACFlow in a smart city scenario. It showed how an operator can express his/her goals as high-level policies and their respective representation in switch-level rules. We also demonstrated the HACFlow expressiveness to define security policies as network configurations.

We analyzed the performance of HACFlow to delegate a role (119.45 ms), reacting to network events (vulnerability alert: 8.24 ms and authentication event: 4.98 ms) and dynamic policies (7.57 ms). We also analyzed its scalability to translate high-level policies into low-level OpenFlow flow rules. The evaluation results showed that using HACFlow significantly reduce the effort to implement a variety of network access control management tasks. Besides, HACFlow was faster and less prone to errors if compared to manual and per-device configurations, common in traditional networks.

We presented a qualitative and quantitative comparison. The qualitative analysis results showed the advantages of HACFlow against Frenetic, FRESCO, and OpenSec. HACFlow offers a simpler syntax to define high-level security policies. While the quantitative results showed that HACFlow required a similar time to translate policies (0.07 ms), but a higher time to react to network events (8.55 ms against 8.1 ms in OpenSec). Despite this, HACFlow provides more management features than these solutions.

6.1 Difficulties Found

Regarding the difficulties found in this work, we highlight the following 1) translation of OrBAC to OpenFlow, 2) configuration of the HP VAN SDN controller and development of an SDN application for it, and 3) comparison of HACFlow against related work.

HACFlow is based on the OrBAC model. The high-level security policies defined in HACFlow are translated into low-level OpenFlow flow rules as network configurations. Therefore, we had to integrate networking concepts in the OrBAC model to allow the OrBAC-to-OpenFlow translation. As an example, consider the following high-level policy *"blocks all TCP traffic from Bob to the webMail service"*. This is a high-level policy that does not make sense for OpenFlow-enabled switches.

So, how to make understood by networking devices the operators' security policies defined in a high-level and human-readable way? How to allow it without compromising the policy expressiveness to define networking configurations? How do we solve this without violating the OrBAC structure? These are some of the challenges that we had to take care while developing the *Policy Skeleton 4.1.2.2*, *Entity Manager 4.1.2.3*, and *Policy Translator 4.1.2.5* sub-components of HACFlow. Allow network operators to define their high-level security policies as network configurations without violating the OrBAC model neither compromising the policy expressiveness was not a trivial task.

The second difficult regards the configuration of the HP VAN SDN controller and the development of an SDN application that uses HACFlow as the authorizing entity. Regarding configuration the HP VAN SDN controller, we had to deal with hardware requirements, controller installation failures, learning how to operate the HP VAN controller, among others. We were guided by [HP:5998-7315C \(2015\)](#) and [HP:5998-4918 \(2013\)](#). On the other hand, to develop the SDN application, we had to configure application files (such as root POM files, module POM files, among others), solved library dependencies, learned the HP VAN REST API, among others. We were guided by [HP:5998-7318 \(2015\)](#).

The last one regards the comparison of HACFlow against related work. We tried to talk with authors by email to get their source code project. One of them provided its code. Others do not provided their code, but answered some questions about their project. While, others unfortunately do not answered us.

From the work that we have access to the source code project, we had to understand

part of its code, make some configurations, and recompile its project to finally conduct the experiments. Resulting in a time-consuming task.

From the work that we can not get access to the code, we had to create a virtual machine with a configuration as next as possible to them (as we do not have a machine with the same configuration). Furthermore, due to the lack information about their testbed configuration we had to contact the authors by email to get this data.

6.2 Future Work and Open Challenges

As future work, we plan to extend HACFlow in two directions. The first one regards the implementation of a semi-automated policy conflict resolution. Currently, the OrBAC component inside of HACFlow allows network operators to solve conflicting policies manually by defining priorities. Operators solve those conflicts one by one, and this may result in mistakes which in turn may result in vulnerabilities.

Besides, performing this task may be cumbersome if we consider an environment with a large number of security policies. Therefore, providing a semi-automated conflict resolution mechanism will simplify even more the network management.

The second one refers to allowing network operators to define not only high-level access control policies. But also Quality of Service (QoS), load balancing, and monitoring policies in a high-level, fine-grained, and human-readable way.

That way, different SDN applications could use HACFlow to define their high-level policies, avoiding the inter-application conflicting problem. This problem, is related to the competing policies between multiple SDN applications [HAN; HU; AHN \(2014\)](#); [PALADI \(2015\)](#).

As open challenges, we point out issues regarding the policy enforcement explosion problem and the need for mechanisms for network monitoring. The policy enforcement explosion problem is related to the enforcement of a huge number of OpenFlow flow rules. The definition of high-level security policies may require multiple low-level OpenFlow flow rules to fulfill the high-level goal.

As presented in Subsection 4.2.5, a single high-level security policy generated six OpenFlow flow rules. How do we may group these related security rules, and enforce a single one instead of six? Therefore, determining mechanisms that aggregate the enforcement of security policies to save switch's resources, such as the use of the limited and power hungry TCAM memory, is required.

The other open challenge regards the need of defining mechanisms for real-time network monitoring in SDN. Real-time monitoring information is critical for a faster and reliable security policy reaction and implementation.

As previously explained, HACFlow reacts to network state changes and events, but it depends on SDN applications that notify HACFlow about those events. Therefore, advances

regarding SDN network monitoring are relevant.

6.3 Statement of the Contributions

In this work, our main contributions can be highlighted as:

1. We demonstrate how SDN, OpenFlow, and the OrBAC model can be used together to improve and automate the network access control management.
2. We propose a framework for the definition of high-level and human-readable policies, trying to simplify the management of access control policies and minimize misconfigurations.
3. We propose a novel solution to define high-level network security policies that dynamically reacts to network events and rule state changes, taking advantage of SDN flexibility and programmability features to reconfigure the network.
4. We show that SDN may be leveraged to offer ACL at a much finer granularity where more flexible rules may be defined, as opposed to existing port and VLAN based rules only.
5. We improve network access control management without modifying the SDN architecture (e.g., the OpenFlow protocol, controllers, and switches).
6. We present a quantitative and qualitative analysis, and a series of examples to motivate and validate our proposed HACFlow framework.

Furthermore, we had a paper (#163955) "*A Network Access Control Solution Combining OrBAC and SDN*" accepted to the Mini-Conference track of the IFIP/IEEE International Symposium on Integrated Network Management (IM 2017). In this paper, we proposed an SDN-based Network Access Control (S-NAC) solution that authenticates and authorizes network entities. This paper was the start point of this research. We evolved the idea and proposed a novel SDN framework with much more management capabilities, as described in Table 4.1.

We also helped to improve the OrBAC model API by notifying a bugfix. Such bug regards the *NotifyContextStateChange()* method in the *AbstractOrbacPolicy* class which was not correctly monitoring concrete rule state changes, making it impossible to create dynamic security policies. The OrBAC API developers corrected this bug, then we tested it to validate the bugfix, and lastly we confirmed the correction to them. In the MotOrBAC web page ¹ (MotOrBAC version 2.5 and OrBAC API 1.5.1 from 12/04/2016), inside the *changelog.txt* file they thank us: "*Big thanks to Daniel Rosendo for pointing this out!*".

¹<http://motorbac.sourceforge.net/index.php?page=news&lang=en>

References

- AHMAD, I. et al. Security in software defined networks: a survey. **IEEE Communications Surveys & Tutorials**, [S.l.], v.17, n.4, p.2317–2346, 2015.
- AOUADJ, M. et al. Towards a modular and flexible SDN control language. In: GLOBAL INFORMATION INFRASTRUCTURE AND NETWORKING SYMPOSIUM (GIIS), 2014. **Anais...** [S.l.: s.n.], 2014. p.1–6.
- AUTREL, F. et al. MotOrBAC 2: a security policy tool. In: CONFERENCE ON SECURITY IN NETWORK ARCHITECTURES AND INFORMATION SYSTEMS (SAR-SSI 2008), LOCTUDY, FRANCE, 3. **Anais...** [S.l.: s.n.], 2008. p.273–288.
- BATISTA, B. L. A.; FERNANDEZ, M. P. PonderFlow: a new policy specification language to sdn openflow-based networks. **International Journal on Advances in Networks and Services Volume 7, Number 3 & 4, 2014**, [S.l.], 2014.
- BLIAL, O.; BEN MAMOUN, M.; BENAINI, R. An overview on SDN architectures with multiple controllers. **Journal of Computer Networks and Communications**, [S.l.], v.2016, 2016.
- COHEN, E. et al. Models for coalition-based access control (CBAC). In: ACM SYMPOSIUM ON ACCESS CONTROL MODELS AND TECHNOLOGIES. **Proceedings...** [S.l.: s.n.], 2002. p.97–106.
- CONVERY, S. Network Authentication, Authorization, and Accounting. **The Internet Protocol Journal**, [S.l.], v.10, n.1, 2007.
- FOSTER, N. et al. Frenetic: a network programming language. In: ACM SIGPLAN NOTICES. **Anais...** [S.l.: s.n.], 2011. v.46, n.9, p.279–291.
- HAN, W.; HU, H.; AHN, G.-J. Lpm: layered policy management for software-defined networks. In: IFIP ANNUAL CONFERENCE ON DATA AND APPLICATIONS SECURITY AND PRIVACY. **Anais...** [S.l.: s.n.], 2014. p.356–363.
- HP:5998-4918. HP VAN SDN Controller Installation Guide, http://h20566.www2.hp.com/hpsc/doc/public/display?sp4ts.oid=5443866&docid=emr_na-c03998700&doclocale=en_us. **White Paper**, [S.l.], 2013.
- HP:5998-7315C. HP VAN SDN Controller 2.5 Administrator Guide, <http://h20566.www2.hp.com/hpsc/doc/public/display?docid=c04647289>. **White Paper**, [S.l.], 2015.
- HP:5998-7318. HP VAN SDN Controller 2.5 Programming Guide, http://h20565.www2.hp.com/hpsc/doc/public/display?sp4ts.oid=5443866&docid=emr_na-c04647292&doclocale=en_us. **White Paper**, [S.l.], 2015.

- HU, V. C. et al. An access control scheme for big data processing. In: COLLABORATIVE COMPUTING: NETWORKING, APPLICATIONS AND WORKSHARING (COLLABORATECOM), 2014 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2014. p.1–7.
- KALAM, A. A. E. et al. Organization based access control. In: POLICIES FOR DISTRIBUTED SYSTEMS AND NETWORKS, 2003. PROCEEDINGS. POLICY 2003. IEEE 4TH INTERNATIONAL WORKSHOP ON. **Anais...** [S.l.: s.n.], 2003. p.120–131.
- KIM, H.; FEAMSTER, N. Improving network management with software defined networking. **IEEE Communications Magazine**, [S.l.], v.51, n.2, p.114–119, 2013.
- KREUTZ, D. et al. Software-defined networking: a comprehensive survey. **Proceedings of the IEEE**, [S.l.], v.103, n.1, p.14–76, 2015.
- LARA, A.; RAMAMURTHY, B. OpenSec: policy-based security using software-defined networking. **IEEE Transactions on Network and Service Management**, [S.l.], v.13, n.1, p.30–42, 2016.
- LIU, J. et al. Leveraging software-defined networking for security policy enforcement. **Information Sciences**, [S.l.], v.327, p.288–299, 2016.
- MATIAS, J. et al. FlowNAC: flow-based network access control. In: THIRD EUROPEAN WORKSHOP ON SOFTWARE DEFINED NETWORKS, 2014. **Anais...** [S.l.: s.n.], 2014. p.79–84.
- NADEAU, T. D.; GRAY, K. **SDN: software defined networks**. [S.l.]: "O'Reilly Media, Inc.", 2013.
- ONF. Software-Defined Networking: the new norm for networks, <https://www.opennetworking.org>. **White Paper**, [S.l.], 2014.
- OSBORN, S.; SANDHU, R.; MUNAWER, Q. Configuring role-based access control to enforce mandatory and discretionary access control policies. **ACM Transactions on Information and System Security (TISSEC)**, [S.l.], v.3, n.2, p.85–106, 2000.
- PALADI, N. Towards secure SDN policy management. In: UTILITY AND CLOUD COMPUTING (UCC), 2015 IEEE/ACM 8TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2015. p.607–611.
- PORRAS, P. et al. A security enforcement kernel for OpenFlow networks. In: HOT TOPICS IN SOFTWARE DEFINED NETWORKS. **Proceedings...** [S.l.: s.n.], 2012. p.121–126.
- PUJOLLE, G. **Software Networks: virtualization, sdn, 5g, security**. [S.l.]: John Wiley & Sons, 2015.
- SANDHU, R.; MUNAWER, Q. How to do discretionary access control using roles. In: ACM WORKSHOP ON ROLE-BASED ACCESS CONTROL. **Proceedings...** [S.l.: s.n.], 1998. p.47–54.
- SANDHU, R. S. et al. Role-based access control models. **Computer**, [S.l.], v.29, n.2, p.38–47, 1996.

- SHIN, S. et al. FRESKO: modular composable security services for software-defined networks. In: NDSS. **Anais...** [S.l.: s.n.], 2013.
- SHIN, S.; GU, G. CloudWatcher: network security monitoring using openflow in dynamic cloud networks (or: how to provide security monitoring as a service in clouds?). In: IEEE INTERNATIONAL CONFERENCE ON NETWORK PROTOCOLS (ICNP), 2012. **Anais...** [S.l.: s.n.], 2012. p.1–6.
- SICARI, S. et al. Security, privacy and trust in Internet of Things: the road ahead. **Computer Networks**, [S.l.], v.76, p.146–164, 2015.
- THOMAS, R. K. Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments. In: ACM WORKSHOP ON ROLE-BASED ACCESS CONTROL. **Proceedings...** [S.l.: s.n.], 1997. p.13–19.
- TR-516, O. **Framework for SDN: scope and requirements**. Version 1.0. Last access: December, 2016, <https://www.opennetworking.org>.
- TS-020, O. OpenFlow Switch Specification: version 1.5.0, <https://www.opennetworking.org>. **White Paper**, [S.l.], 2014.
- VOELLMY, A.; KIM, H.; FEAMSTER, N. Procera: a language for high-level reactive network control. In: HOT TOPICS IN SOFTWARE DEFINED NETWORKS. **Proceedings...** [S.l.: s.n.], 2012. p.43–48.
- WICKBOLDT, J. A. et al. Software-defined networking: management requirements and challenges. **IEEE Communications Magazine**, [S.l.], v.53, n.1, p.278–285, 2015.
- WINSBOROUGH, W. H.; LI, N. Towards practical automated trust negotiation. In: POLICIES FOR DISTRIBUTED SYSTEMS AND NETWORKS, 2002. PROCEEDINGS. THIRD INTERNATIONAL WORKSHOP ON. **Anais...** [S.l.: s.n.], 2002. p.92–103.