Université Hassan 1er

Centre d'Études Doctorales

Faculté des Sciences et Techniques
Settat Sciences

THÈSE DE DOCTORAT

Pour l'obtention de grade de Docteur en Sciences et Techniques

Formation Doctorale : Mathématiques, Informatique et Applications

Spécialité : Informatique.

Sous le thème

Towards a Secure and Reliable Software Defined Network

Présentée par :

Youssef Qasmaoui

Soutenue le: *02 Juin 2021*

A la Faculté des Sciences et Techniques de Settat devant le jury composé de :

Pr. Ahmed Mouhsen	PES	FST de Settat	Président
Pr. Driss El Ouadghiri	PES	Faculté des Sciences de Meknès	Rapporteur
Pr. Abderrahim Marzouk	PES	FST de Settat	Rapporteur
Pr. Mohamed Hanini	PH	FST de Settat	Examinateur
Pr. Abdelkrim Haqiq	PES	FST de Settat	Directeur de thèse

Année Universitaire: 2020/2021

The only system which is truly secure is one which is switched off and unplugged locked in a titanium lined safe, buried in a concrete bunker, and is surrounded by nerve gas and very highly paid armed guards. Even then, I wouldn't stake my life on it.

-- Gene Spafford, Director, Computer Operations, Audit, and Security Technology (COAST) Project, Purdue University

REMERCIEMENTS

J’exprime mes sincères remerciements et toute ma gratitude, en premier lieu, à Mr. Abdelkrim HAQIQ, professeur à la FST de Settat et directeur du laboratoire Informatique, Réseaux, Mobilité et Modélisation (IR2M), qui a dirigé et encadré cette thèse. C’est grâce à son expertise, ses conseils et son écoute que ce travail a pu aboutir. Je le remercie aussi pour le temps considérable qu’il a passé à relire et corriger le manuscrit de thèse, les articles et les présentations. J’aimerais aussi souligner que son travail ne se limite pas à l’aspect scientifique mais aussi humain et relationnel. Il a su guider mon travail avec intelligence et a fait preuve de disponibilité en toute circonstance.

Je tiens à remercier vivement les rapporteurs de mon manuscrit de thèse, Mr. Driss El Ouadghiri, professeur à la Faculté des Sciences de Meknès et Mr. Abderrahim Marzouk, professeur à la FST de Settat, qui ont accepté de lire et de juger ce travail. Je leur exprime ma profonde gratitude.

Je suis très sensible à l’honneur que me fait Mr. Ahmed Mouhsen, professeur à la FST de Settat, d’avoir accepté de présider le Jury de ma soutenance. Que ce travail soit pour moi l’occasion de lui exprimer ma grande estime.

Je tiens à remercier chaleureusement Mr. Mohamed Hanini, professeur à la FST de Settat, d’avoir accepté de siéger à mon jury de thèse en tant qu’examinateur. Je considère sa présence parmi le jury comme un témoignage de haute valeur.

Merci également à mes collègues du laboratoire IR2M, pour leurs conseils, encouragements, et pour la sympathie qu’ils m’ont témoignée.

Ma gratitude va également vers le Prof. Yassine MALEH de l’École Nationale des Sciences Appliquées de Khouribga pour son soutien continu, sa disponibilité inconditionnelle, un frère plus qu’un ami d’étude, une seule famille nous réunit. Cette thèse est dédiée à sa mère qui vient de passer son chemin vers DIEU « qu’elle repose en paix ».

Je tiens à remercier toute ma famille pour leur amour inconditionnel, de soins et d'encouragement. Je serai éternellement reconnaissant à ma mère Mme. Khadija Chouaibi pour instiller en moi l'amour de l'apprentissage et le désir continu pour plus de connaissances. Je dédie ce modeste travail à l’âme de mon père Mr. Moahmed Qasmaoui. Je lui dois beaucoup plus que je ne serais jamais capable d'exprimer.

Je remercie sincèrement mon épouse Mme Samia Fellah, qui m'a toujours soutenu inconditionnellement, qui m'a toujours encouragé à continuer dans les moments de stress et qui m'a donné toutes les chances de réussir, sans oublier mon adorable petit prince Jad Qasmaoui.

Je remercie chaleureusement mes frères Mr. Abdellah, Abderrahim, Abdelhaim, Ibrahim, Noredine, Mohamed, Ahmed et mes sœurs Mme Fouzia, Rachida, Aicha, Meryem, Fatima, mes amis et tous les membres de ma famille pour leurs encouragements et leur soutien moral qui m'ont permis de faire ma thèse dans de bonnes conditions.

Enfin, je vous remercie de vos bénédictions. Pour le courage et la patience que vous m'avez donnés pour accomplir ce travail

ABSTRACT

Today's communications networks are increasingly heterogeneous, with more and more new communications technologies, especially wireless communications. They are also more and more dynamic, first because wireless communications allow very fast changes of topology and the speed of appearance of new equipment and communicating applications. They are facing several challenges, particularly their evolution, which is becoming more and more complex to manage, and the requirement of real-time operations.

The concept of Software Defined Networks (SDN) has emerged to manage such heterogeneous and dynamic networks. Separating the control and data planes provides the flexibility required for these new types of networks while preserving communications performance levels.

An SDN architecture includes an SDN controller, which enables the management of the control plane, and is an essential element for the proper operation and management of the network. However, due to its centralized nature, the SDN controller can be a bottleneck and affect the scalability of the network, or it can be a prime target for potential hackers to perpetrate a denial-of-service attack against the network, for example.

The controller itself plays a central role in the SDN architecture and, as such, is quite easily exposed to multiple attacks. The controller makes decisions via so-called flow rules that the network administrator defines. The forwarding plane contains applications that can generate flow rules. These applications are allowed to access the routing rule management module present in the controller to store their rules, introducing vulnerabilities in the network, making the flow rules unreliable.

The objective of this thesis was to fill this gap and analyze the security of the SDN infrastructure, identify vulnerabilities and weaknesses, and propose corresponding solutions and improvements. First, we proposed a new version of SOLID-FLOW to improve the integrity of the flow rule database inside the SDN controller. Second, we proposed a new approach that could be applied to multiple SDN controllers. It is based on the implementation of Intel SGX technology in the development of control solutions. The integration of this technology into the SDN controller "Floodlight" has shown promising results.

Keywords: Software Defined Network, Open Flow, Security, Integrity, SDN Controller, Intel SGX.

RÉSUMÉ

Les réseaux de communications actuels sont de plus en plus hétérogènes, avec de plus en plus de nouvelles technologies de communications, notamment les communications sans fil. Ils sont également de plus en plus dynamiques, d'abord par le fait que les communications sans fil permettent des changements très rapides de topologie, mais aussi par la rapidité d'apparition de nouveaux équipements et applications communicantes. Ils font face à plusieurs défis qui concernent particulièrement leur évolution, qui devient de plus en plus complexe à gérer, et l'exigence des opérations à temps réel.

Pour pouvoir gérer de tels réseaux hétérogènes et dynamiques, le concept des réseaux définis logiciel (ou SDN : Software Defined Networks) a émergé. En séparant les plans de contrôle et de données, il permet la flexibilité requise pour ces nouveaux types de réseaux, tout en préservant les niveaux de performance des communications.

Une architecture d'un réseau SDN comprend un contrôleur SDN, qui permet la gestion du plan de contrôle, et qui est un élément essentiel pour le bon fonctionnement du réseau et sa bonne gestion. Toutefois, en raison de sa nature centralisée, le contrôleur SDN peut constituer un goulot d'étranglement et affecter l'évolutivité du réseau, ou il peut être une cible de choix pour les pirates potentiels qui voudraient perpétrer une attaque par déni de service contre le réseau, par exemple.

Le contrôleur lui-même joue un rôle central dans l'architecture SDN et, de ce fait, il s'expose assez facilement à plusieurs attaques. Le contrôleur prend les décisions via des règles appelées (Flow-Rules) qui sont définies par l'administrateur réseau. Le plan de transfert contient des applications qui peuvent générer des règles de flux, ces applications sont autorisées à accéder au module de gestion des règles de routage présent dans le contrôleur pour y stocker leurs règles, ce qui introduit des vulnérabilités dans le réseau rendant ainsi les règles de flux peu fiables.

L'objectif de cette thèse était de combler cette lacune et d'analyser la sécurité de l'infrastructure SDN, d'identifier les vulnérabilités et les faiblesses, et de proposer des solutions et des améliorations correspondantes. Premièrement, nous avons proposé une nouvelle version de SOLID-FLOW pour améliorer l'intégrité de la base de données des règles de flux à l'intérieur du contrôleur du SDN. Deuxièmement, nous avons proposé une nouvelle approche qui pourrait être appliquer à plusieurs contrôleurs SDN. Elle se repose sur l'implémentation de la technologie Intel SGX dans le développement des solutions du contrôle. L'intégration de cette technologie dans le contrôleur SDN "Floodlight" a démontré des résultats prometteurs.

Mots-clés : Réseaux Définis Logiciel (SDN) ; Sécurité ; Intégrité ; Contrôleur SDN ; OpenFlow; Intel SGX.

TABLE OF CONTENTS

Abstract .... **I**

Résumé .... **II**

List of Figures .... **V**

List of Tables .... **VII**

List of Acronyms .... **VIII**

General Introduction **1**

1. Context 1
2. Problem statement 2
3. Objectives and contributions 2
4. Thesis organisation 3

Chapter 1: Software Defined Networks: Architectures and Applications **6**

1. Introduction 6
2. Software Defined Network 7
   - 2.1 Definition 7
   - 2.2 SDN Architecture 8
     - 2.2.1 Data Plane 9
     - 2.2.2 Control Plane 9
     - 2.2.3 Application Plane 11
   - 2.3 Sdn Networks Characteristics 12
   - 2.4 Openflow Communication Protocol 13
   - 2.5 Flow Tables 14
   - 2.6 SDN Controllers 20
   - 2.7 SDN Applications 26
     - 2.7.1 Multimedia and QoS 26
     - 2.7.2 Cloud Computing and Data Centers 26
     - 2.7.3 Security and Privacy 27
3. Conclusion 28

Chapter 2: SDN Security Attacks and Mitigations 29

1. Introduction 29
2. SDN Security Solutions in Control Plane 29
   2.1 Lack of access control and accountability 31
   2.2 Fake flow rules 32
   2.3 Authentication and Authorization 36
   2.4 Scalability & availability 40
   2.5 Unauthorized controller access 46
   2.6 DoS attacks 49
   2.7 Malicious flow rules 52
3. Conclusion 52

Chapter 3: Enhanced Solid-Flow: An Enhanced Flow Rules Security Mechanism for SDN 55

1. Introduction 55
2. Related works 56
3. The proposed model 59
   3.1 System Model 59
   3.2 Solid-Flow design principles 62
4. Performances Analysis and Discussion 66
   4.1 System reactivity 67
   4.2 The CPU rate used at the SDN controller 68
   4.3 SDN Controller response time to attacks 69
5. Conclusion 70

Chapter 4: Secure Software Defined Networks Controller Storage using Intel Software Guard Extension 71

1. Introduction 71
2. Background 73
3. Intel SGX architecture 74
4. Related Works 78
5. The proposed SGX SDN controller 80
   5.1 SDN storage module overview 80
   5.2 SDN enabled SGX architecture 81
6. Implementation and evaluation 82
   6.1 Implementation setup 82
   6.2 Performance analysis 84

7. Conclusion 85

Conclusion and Future Works 86

Thesis Publications 88

References 89

Appendices 97

Appendix 1: Mininet Emulator 97

Appendix 2: Mininet simple code over python script to run pings 99

Appendix 3: SGX_ec256_public_t.cpp file loaded by IStorageSourceService.java 103

Appendix 4: Sample code of modified SDN (floodlight) controller storage module 105

LIST OF FIGURES

Fig. 1. The overall structure of the thesis 4
Fig. 2. Open Flow –Table 9
Fig. 3. SDN Architecture showing the tree layers (Application plan, Control plan, Data Plane) 12
Fig. 4. OpenFlow Commutator 14
Fig. 5. Traffic Matching, Pipeline Processing, and Flow Table Navigation 15
Fig. 6. Table-miss flow entry flowchart 16
Fig. 7. OpenFlow Messages 18
Fig. 8. General Overview of SDN Controller 21
Fig. 9.Security Threat Vectors Map in SDN 30
Fig. 10. FLOVER architecture 32
Fig. 11. PERM-GUARD 36
Fig. 12. Two ONIX controller coordinating and their views of the underlying network state 41
Fig. 13. High-level overview of HyperFlow 42
Fig. 14. The architecture of IRIS-HiSA controller cluster 46
Fig. 15. Floodlight controller with security extensions 49
Fig. 16. SOMs detection process 51
Fig. 17. SD-Anti-DDoS four state diagram 52
Fig. 18. Floodlight Architecture 60
Fig. 19. Attack timeline 61
Fig. 20. Enhanced Solid-Flow architecture 61
Fig. 21. The system reactivity under attacks 68
Fig. 22. CPU usage 69
Fig. 23. SDN controller response time to attacks 70
Fig. 24. Trusted execution environment 73
Fig. 25. Intel SGX implementation 74
Fig. 26. Intel SGX software stack 75
Fig. 27. Intel SGX trustworthy storage process 76
Fig. 28. Intel SGX architecture 77
Fig. 29. TLSonSGX system design 79
Fig. 30. The proposed SDN enabled SGX model 82

LIST OF TABLES

Table I. SDN Controller's Feature Comparison Table 23

Table II. PERMOF PERMISSION SET 31

Table III. Example OpenFlow rule set used to illustrate coverage and modify violation 34

Table IV. A summary of control layer mediation policies for data flows initiated from the application layer to data plane (A to D) and from the data plane to application layer (D to A) 47

Table V. Parameters of algorithms 63

Table VI. Implementation parameters 67

Table VII. Summary of the code changes 83

Table VIII. Collected measures and overhead 84

LIST OF ACRONYMS

AESM	: Application enclave service manager
AHA	: Actual Hash
API	: Application programing interface
ARRs	: Alias reduced rules
BGP	: Border Gateway Protocol
CLI	: Command-line interface
CST	: Controller State
CT(ER)	: Number of wrong hash counter
CVF	: Controller Verification
DDoS	: Distributed Denial Of Service
DHA	: Data Store Hash
DOS	: Denial of service
DPT	: Default Periodic Timer
EBI	: East Bound API
eHIPF	: Enhanced history-based IP filtering
EPC	: Enclave Page Cache
ETH	: Ethernet
GPL	: General Public License
GT	: Game theory
HKDF	: KDF based on HMAC
HMAC	: keyed-hash message authentication code
IDC	: International Data Corporation
iDVV	: integrated device verification value
Intel SGX	: Intel Software Guard Extensions

IOT	: Internet of things
IS-IS	: Intermediate system to intermediate system
JNI	: Java Native Interface
KDC	: Key distribution center
KDF	: Key derivation function
LAGs	: Link Aggregation Groups
MAC	: Media Access Control
NBI	: Northbound Interface
NOS	: Network Operating System
OF protocol	: OpenFlow protocol
OF	: OpenFlow
ONF	: Open Networking Foundation
OSGi	: Open Services Gateway Initiative
OSPF	: Open Shortest Path First
OvS	: Open virtual SWicthe
PE	: Provisioning Enclave
PT	: Periodic Timer
QE	: Quoting enclave
QNOX	: NOX supporting Quality of Service
QOS	: Quality of service
RC	: Resource controller
RCA	: Rule-based Conflict Analysis
RT	: Reactive Timer
SBI	: Southbound Interface
SDK	: Software Development Kit
SDN	: Software Defined Networks
SDNs	: Software Defined Networks

SEK	: Security enforcement kernel
SNAC	: Secure Networking and Computing
SOMs	: Self-organizing maps
SSH	: Secure Shell
T_HRMAX	: Maximum Threshold
T_HRMIN	: Minimum Threshold
TCP	: Transmission Control Protocol
VLAN	: Virtual local area network
VSDN	: Video over Software Defined Networking
WBI	: West Bound Interface
Web UI	: Web User Interface

General introduction

1. CONTEXT

With the growth in the use of information technology, there is a huge increase in the amount of traffic circulating in networks due to the large number of connected devices and modern internet applications, such as social networks and document sharing. Network administrators need to manage a wide range of data formats, service types and devices, which is difficile with traditional network management tools that were not designed to handle scalable topologies on a very large scale.

The concept of software defined networking (Software Defined Networking - SDN) is the solution to meet the needs of users of these network services and applications. The SDN architecture allows network operators to dynamically react and manage their infrastructures in response to changes in the behavior of traffic in the network. SDN allows the separation between the control plan, which is responsible for making decisions about the routing of flux in the network and the management of the services and applications offered by the network, and the data (or transmission) plan, which is responsible for transmitting the flux data according to the guidelines provided by the control plan.

As SDN technology gains momentum and several Internet providers and data center administrators are gradually adopting it, there is growing interest in the security issues that may arise concerning its deployment in production. A recent survey of security threats in SDNs classify types of attacks according to categories such as unauthorized access, data access, malicious applications, denial of service (DoS), and configuration issues (Scott-Hayward, S., O'Callaghan, G. and Sezer, S., 2013). The results of this study also show that the control plan and the data plan are always infected simultaneously in most attacks, which explains the complementary relationship between these two planes in the SDN concept.

The centralized architecture of the SDN is proving to be a double-edged sword. On the one hand, it gives the network administrator a global view through the controller to facilitate network management. On the other hand, this centralization is a vulnerability for SDN networks, taking over a controller is taking over the overall network.

In this work, we are interested in the attacks and limitations of targeting the control plane layer, attacks mostly coming from application plans as they are able to generate flow rules, those applications are allowed to access the data store present within the controller to store their rules, which introduces vulnerabilities in the network making the flow rules untrustworthy. Access to data store can only be trusted if the security problems inherent in software-defined networks are resolved. We proposed some mechanisms to improve flow rules database integrity inside the SDN controller, the second part in which we are interested is securing the Execution environment of the controller itself, the controller is a program running on host OS. Therefore, every impact to the OS is consequently putting the control plan into suffering.

2. PROBLEM STATEMENT

According to SDN's specifications (Open Networking Foundation, 2013), the controller is the only entity responsible for the decision to route flux in the network. At the first phase the flow rules are dispatches to the underlaying infrastructure which contain network equipment such as routers and switches. Those flow rules are predefined by the network administrator, in case of upcoming packets, the switches use its flow tables to forward packets to destination. On the other hand, when the switch receives a packet belonging to a new flux, it asks the controller to dispatch flow rule matching the new destination, the controller respond with the corresponding new rout and the switch forward the packet to the next node. The role of the controllers is crucial to a good networking infrastructure, if the controller is compromised or disturbed the totality of the network become paralyzed.

The main components of the controller itself remain the flow rules management module which is responsible for adding, modifying and deleting flow rules from the controller storage module. An unauthorized entity can alter the existing flow rules leading to a dispatch of packets to a man in the middle or even deleting flow rules, which paralyze the network resulting in a denial of service. Protecting the controller flow rules management component is essential to a more secure and stable SDN network.

3. OBJECTIVES AND CONTRIBUTIONS

As applied research, this study attempts to help address the strategic gap in SDN networks security that has been identified as a significant, real, global problem. In highlighting the "know and do" gap underlying this study, we have tried to tackle this problem boldly. Research overcomes a number of challenges in conducting research in new areas of securing SDN

networks. The foundations have been laid for industry and scholarly literature in this field to contribute to knowledge. The establishment of this research has highlighted multiple areas in which this thesis has been added to the body of knowledge and practice.

This research makes the following significant contributions to networks security:

- The first contribution of this research is trying to solve one of the most critical security issues of SDN controller, by reinforcing the integrity of SDN Flow rule management inside the SDN controller. we propose an enhanced SOLID-FLOW to improve flow rules database integrity inside the SDN controller.
- The second contribution is an SGX enabled SDN controller, a new controller ensures the integrity and the confidentiality in a trusted execution environment by leveraging a recent hardware technology called intel SGX.

4. THESIS ORGANISATION

This thesis consists of four chapters, with a general introduction, which presents the context, the problem statement, the objectives and contributions and the thesis. The first chapter provides an overview of Software Defined Networks (SDNs). It also introduces other concepts that are useful to well understand the approaches that we have defined.

The second chapter review the various works that propose solutions to secure the SDN controller. We will mainly present the aspects considered by our work as well as other related aspects.

The third chapter focuses on our first contribution consisting of a new approach to mitigate flow rules integrity issues. The fourth chapter will present our second contribution regarding the SDN controller security, based on the implementation of Intel SGX to more harden the execution environment.

Lastly, we will conclude the thesis by providing a summary of our contributions and their limitations. We will also provide some perspectives. The overall structure of the thesis is described in Fig. 1.

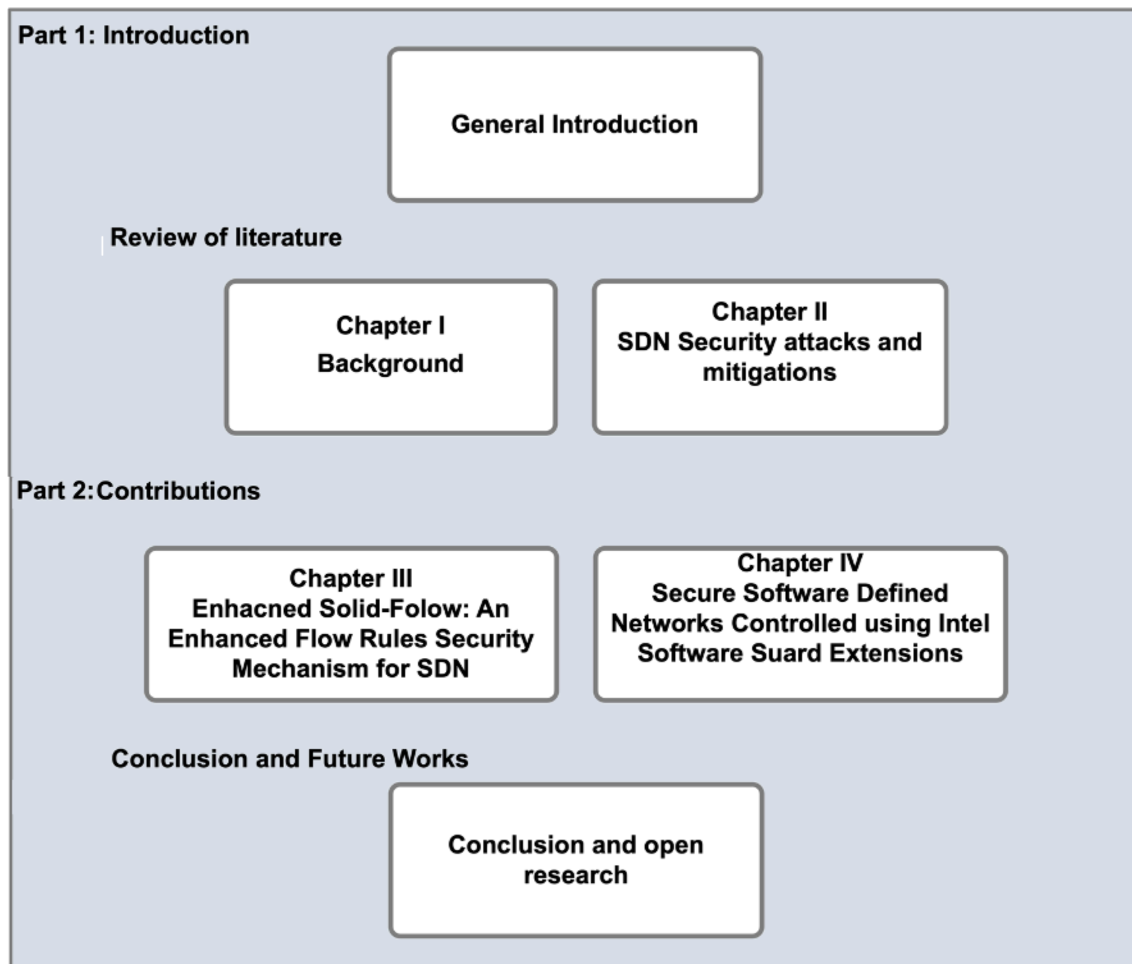


Fig. 1. The overall structure of the thesis

1st Part: Literature Review

Chapter 1: Software Defined Networks: Architectures and applications

1. INTRODUCTION

In Conventional networks, the deployment and administration of network infrastructures and services is a difficult task and involves the configuration of a huge number of individual network components such as routers and switches, typically via proprietary interfaces such as Serial interfaces or console cables. Packet forwarding mechanism is controlled via complex routing protocols that need advanced configuration skills such as OSPF (Moy, 1998), BGP and EGP (Filsfils, C., Previdi, S. B., & Vasseur, 2011). The nonexistence of a centralized general view of the network state, as well as the absence of relevant networking abstractions, make it challenging to implement high-level and real-time forwarding policies in traditional IP networks. As a result, the network devices are configured rather than programmed, making innovation and the implementation of new networking services a heavy slow task (Nunes et al., 2014).

Software Defined Networking (SDN) is a relatively new paradigm that aims to manage and configure computer networks, which purposes to address this problem through detaching the control intelligence from forwarding Devices, such as switches and routers, and handing it to a logically centralized component, i.e. the SDN controller (Cui et al., 2016; Z. Hu et al., 2015; Khattak et al., 2014). The separation of the control plane from the data plane in SDN makes the network more agile. The complex and labor-intensive work of configuring individual networking devices, sometimes configuration is done in place, is now replaced by the easier and more efficient way of 'programming' the network elements in real-time.

In SDN, the SDN controller hides the complexity and details of the underlying network infrastructure by providing an abstraction layer and northbound/southbound interfaces. Network services and policies are deployed as applications that sit on top of the SDN controller, running on a logically centralized controller (J.C. et al., 2020). The controller communicates with the forwarding elements, i.e. the data plane, through a northbound interface, e.g. the OpenFlow protocol (Ying Qian et al., 2016).

As a result of the separation of the control plan form data plan as well as the increased level of abstraction, SDN makes the network more programmable, flexible, and agile, which dramatically simplifies network management and configuration by reducing time and complexity, and enables faster innovation.

The implementation and deployment of new network services and routing policies, which take a large amount of time, effort and skills in traditional networks, become a reasonably practicable task in an SDN-enabled network.

SDN has acquired incredible momentum, both in the industry sector and among the research community, and has been successfully adopted in data centers and Wide Area Networks (WANs) (Cziva et al., 2016; J.C. et al., 2020; J. Son et al., 2017). For example, Google has constructed an SDN-based WAN in its internal backbone network to globally connect its many data centers. As a result, network performance has been improved dramatically, and the link utilization has increased from 30 ∼ 40% to close to 100% (Voellmy & Wang, 2012). This tremendous achievement is a result of more fine-grained control over the forwarding of network traffic, and the increased network programmability provided by SDN. The growth of SDN will likely continue in the future, which is confirmed by the fact that the leaders of networking vendors such as Cisco (CISCO, 2020), Huawei (Huawei, 2020), Juniper (Juniper, 2019) and Hewlett Packard (Fruhlinger, 2019) are increasingly supporting SDN-based products and lunching new SDN based solution.

2. Software Defined Network

2.1 Definition

The Open Network Foundation (ONF) has defined the SDN network as: *«Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services»* (ONF, n.d.)

The software defined network (SDN) is a new paradigm aiming to transform the old-fashioned network and their infrastructures, the way they are managed and configured. The concept of SDN relay on separating control plane which holds all the intelligence of the network from the data plane handling the packets transfer. The data plane is basically a logically centralized

controller, which englobe a global vision of the network, the controller most significant role is the decision making about what goes where, traffic management using flow-rules (Lara et al., 2013). This global view of the network offered by the SDN networks makes it easy for a system admin to change topology, prioritize traffic verify QoS, and enhance security.

The Data plane is responsible for transmitting the traffic over the SDN network, from a source to a destination, these operations are executed based on what flow rules are present with the data plane devices (routers and switches). The communication between the controller and the underlaying plane equipment is ensured via a standard communication protocol called OpenFlow, the open flow protocol handle all type of communication and operation between the controller and the data plane layer. For example, to modify the forwarding table the controller sends data over OpenFlow protocol to the openvswitches (switches with OpenFlow enabled). In the other hand one of the greatest advantages offered by the SDN network is its programmability allowing IT admin to configure, manage, control all the network from SDN API (Application programing interface) (Farahmandian & Hoang, 2016).

2.2 SDN ARCHITECTURE

Common network infrastructures generally use equipment (switches, routers, etc…) whose configuration depends on the manufacturers. Once deployed, it is difficult to evolve these infrastructures by adopting new protocols or by adding and removing equipment. Thus, the idea of Software Defined Networks (SDNs) emerges.

SDN is a new paradigm for defining the behavior of network equipment using control software (Ahmad et al., 2015). Consequently, SDN separates the data plan and the control plan. The data plan defines the equipment of the network and the connections between them while the control plan defines the behavior of the network and the of this network.

The purpose of SDN technology is to offer some open interfaces that enable the development of software that can manage the connectivity supplied by a collection of network resources and the flow of network traffic through them, along with possible examination and adjustment of traffic that may transit in the network (Sasaki et al., 2016). The ONF (Open Networking Foundation) defined three main principles of SDN (Sasaki et al., 2016):

- Separating the controller and data planes:
- Decouple control plan from the data plan. However, control must be executed within the data plan system.

- A central control: This aspect will provide a high overview of the network and can enhance the deployment of decision making instead of local control, therefore changes latency are less significant.
- Exposure of abstract network resources and state to outer applications.

These principals are clarified by the SDN architecture, which is built around three principal layers: Application plane layer, Controller plane layer and Data plane layer.

2.2.1 DATA PLANE

The data plane includes the forwarding hardware i.e. switches/routers, and incorporates all software interfaces and hardware elements (Yan et al., 2016). The forwarding operations are handled using a forwarding table called flow table, which contains rules that manage forwarding operations. Fig. 2. Shows the Flow Table Entry architecture.

Fig. 2. Open Flow –Table

2.2.2 CONTROL PLANE

Network intelligence is installed in the software-based logically centralized SDN controller; this controller communicates through a standard southbound API (OpenFlow). The control layer regulates and manages the forwarding decisions of the hardware (Yan et al., 2016). ONF designed the SDN architecture with only a single Controller but new scenarios use multiple

distributed controllers (Oktian et al., 2017)(Bianco et al., 2017) to enhance the availability and scalability of network ressources. The performance of SDN controllers is characterized by throughput, which is the amount of flows processed per second, and latency, which is the time it takes to install a new rule.

Several controllers have been developed, the majority of which are open source and support the OpenFlow protocol. These controllers differ in their programming languages, the version of OpenFlow supported, the techniques used such as multi-threading and the performance such as throughput. Below is a non-exhaustive list of SDN controllers:

- Nox (Gude et al., 2008): the first publicly available controller written in C language, the use of a The only thread in its core limits its deployment. Several NOX bypasses have been later proposed, notably the Nox in its multithreaded version called NOX-MT (Tootoonchian et al., 2012).
- QNOX (Jeong et al., 2012), NOX supporting Quality of Service (QoS), FortNOX (Porras et al., 2012), an extension of NOX implementing a conflict detection analyzer in the flow rules caused by applications, and finally, the POX Controller (Prete et al., 2014), a NOX-based controller in python language, whose purpose is to improve the performance of the original NOX controller.
- Maestro (Maestro, n.d.): uses multithreading technology to perform bottom parallelism level, keeping a simple programming model for application developers. It achieves its performance through the distribution of tasks from the core to the threads.
- available and the minimization of memory consumption. In addition, Maestro can process requests from multiple streams through a single runtime task, increasing its efficiency. efficiency.
- Beacon (Beacon, n.d.): developed in Java by Stanford University, it is also based on multi-threaded technologies and is multiplatform. Its modular architecture allows the manager to run only the services desired.
- SNAC (SNAC, n.d.): uses a web application to manage network rules. A language of Flexible rules definition and easy-to-use interfaces have been integrated to configure network devices and control their events.
- Floodlight (Floodlight, n.d.-a): is a variant of Beacon characterized by its simplicity and performance. It has been tested with physical and virtual OpenFlow switches. It is now supported and improved by a large community of developers, including manufacturers such as Intel, Cisco, HP, Big switch and IBM.

- McNettle (Voellmy & Wang, 2012): it is an SDN controller programmed by Nettle (Voellmy & Hudak, 2011), which is a DSL (Domain Specific Language) integrated in Haskell, and allows programming of networks. using OpenFlow. McNettle operates in multi-core servers that share their memory to achieve global visibility, high performance and low latency.
- RISE (Ishii et al., 2012): designed for large-scale network experiments, RISE is a Trema-based controller (Controller, n.d.). The latter is a framework programmed in Ruby and C. Trema provides an integrated test and debugging environment, including a set of tools for development.
- Ryu (Ryu, n.d.): is a framework developed in Python. It allows the separation between domains without using a VLAN. It supports up to the latest version of OpenFlow 1.5.
- Opendaylight (OpenDaylight, n.d.): is an open source project implemented in java and cross-platform. It supports the OSGi Framework (Hall & Cervantes, 2004) for local programming of the controller and REST (Fielding, R. T., & Taylor, 2000) bidirectional. Companies such as ConteXteam, IBM, NEC, Cisco, Plexxi, and Ericsson are very active in this project.

2.2.3 Application plane

Applications and services explore possibilities offered by the control and infrastructure layer. The abstract application layer is located over the control plane layer and enables the unsophisticated development of network applications (Sasaki et al., 2016), such as network virtualization, traffic monitoring, etc. Those applications interact with the control layer through the northbound API.

Application can include SDN specific applications belonging to SDN providers itself. However, it can also have a third-party application developed to meet specific needs such as applications related to network automation to better align with the needs of the applications running on it, network configuration and management to enable real-time configuration and ease of administration, network monitoring, network troubleshooting, network policies and security. Those applications often present high-risk rate, specifically third-party applications, the threats generally target the control plan because of its high value, hens securing the access to control data provided by the SDN controller is mandatory.

Fig. 3. shows the architecture of the typical SDN network as described by the Open networking foundation.

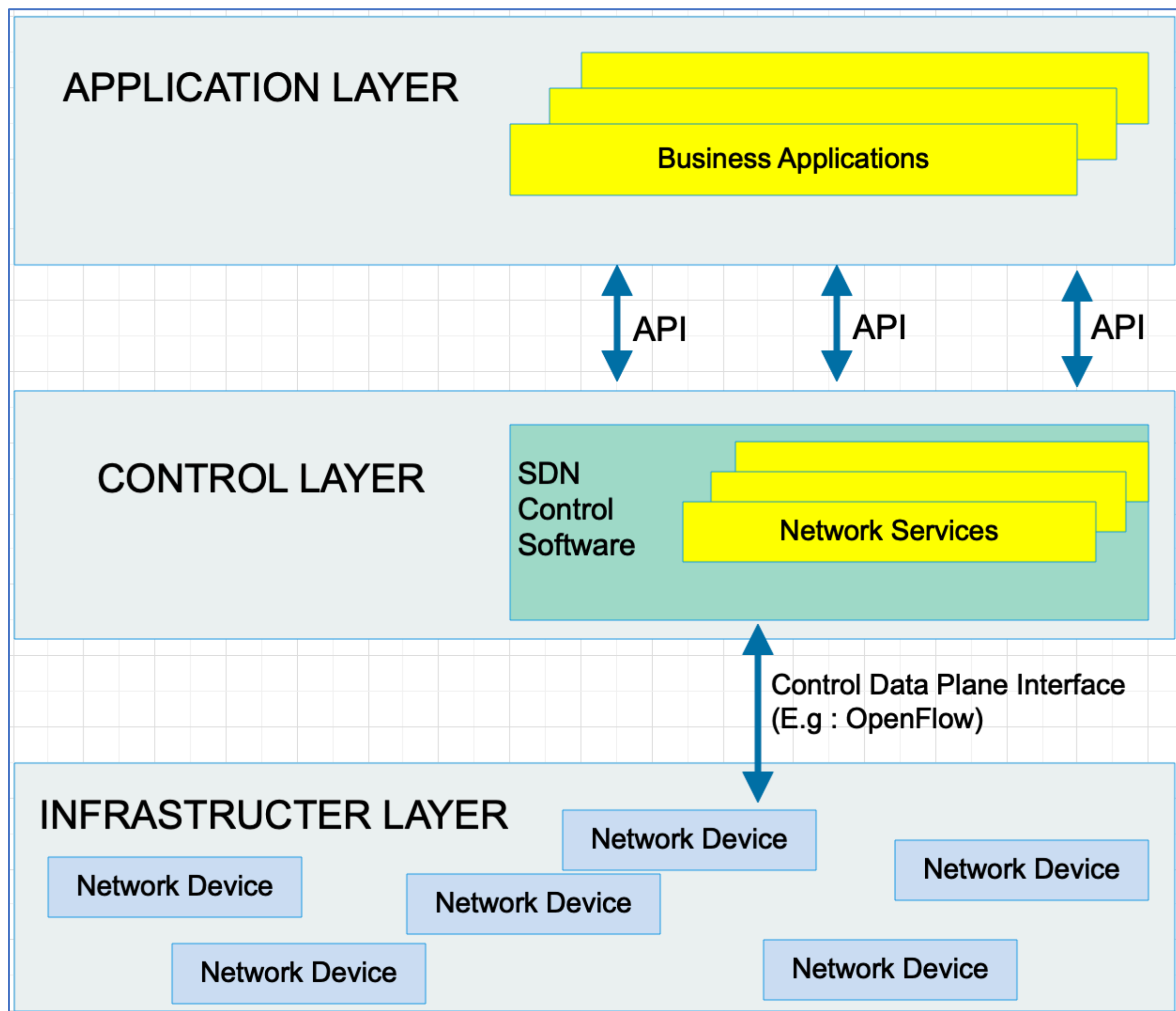


Fig. 3. SDN Architecture showing the tree layers (Application plan, Control plan, Data Plane)

2.3 SDN NETWORKS CHARACTERISTICS

The ONF (Open Networking Foundation), define the SDN as a new paradigm to new generation of network management and flow control. The most relevant SDN architecture are specification:

- **Directly programable:** the control of the network is directly programable because its dissociated from traffic management functionalities.
- **Agile :** The separation between control functionalities and traffic transmission offer the IT administrators the possibility to dynamically adjust traffic on the global network, this agility offers a better solution for network evolution.
- **Central management:** the intelligence of the network is centralized at the controller level maintaining a global view of the network state, the SDN controller is a program defined as a policy engine and seen as one logical switch.

- **Live management:** SDN offers to IT manager the ability to push flow rules and redirect traffic in real-time without any need to physical access to data plane infrastructure. Real time management offers many advantages in term of fast deployment, security and optimization.
- **Standards based an open:** SDN network relies on OpenFlow which is an open protocol, hence there is no need to depend on any proprietary solutions.
- **Visibility** : The global view of the network offers better understanding of how things are actually going on the network, knowing the actual state of the network make decision making more efficient and simple.
- **Orchestration**: The management of individual network component mark a huge complexity in term of mass configuration, SDN offers the possibility to orchestrate multisource and distant equipment's such as could networks from a single interface.
- **Flexibility** : Network IT manager can easily change the scall of a network, often a huge charge on a part of the network can result in instable networking services. IT managers can easily add or remove equipment from the network such as compromised component's, block or redirect traffic.
- **Performance**: SDN network offers better performance in case of load balancing, fault tolerance Fast and equipment failure handling;
- **Standardization** : The interoperability between network equipment's from different manufacturer is solved by the use of the standard communication protocol OpenFlow.

With all these advantages SDN is quickly been adopted as the v 2.0 of networking, offering better performance, more agility and central decision making.

2.4 OPENFLOW COMMUNICATION PROTOCOL

The OpenFlow protocol is a network protocol tightly associated with Software-Defined Networking (SDN). SDN is a network architecture that allows network administrators to control traffic from a centralized Controller.

A Controller is an application that manages flow control in the SDN environment. The OpenFlow protocol allows a server to instruct network switches where to send data packets. In a non-OpenFlow or legacy switch, packet forwarding (the data path) and route determination (the control path) occur on the same device. A switch using the OpenFlow protocol separates the data path from the control path.

The OpenFlow protocol is used on the control plane (which is centralized on the SDN Controller) to communicate with the data plane (which is distributed among the network nodes) in an SDN network.

Using the OpenFlow specifications, a switch can be configured to operate with similar results to a legacy switch, without having to manually re-configure the switch if the network changes. A selection of Allied Telesis switches works with version 1.3 of the OpenFlow specification. These switches enable the OpenFlow protocol on a per-port basis, so you can choose which ports of the switch will be controlled by the OpenFlow feature, as shown in Fig. 4.

Non-OpenFlow-enabled ports, continue to support existing features of the device. An OpenFlow enabled port will handle all untagged and VLAN tagged traffic. A hybrid OpenFlow port allows some VLAN tagged traffic to be processed as non-OpenFlow protocol traffic. This is achieved by setting the port to trunk mode and adding VLANs to the port. Untagged traffic and tagged traffic for all other VLANs are handled by the OpenFlow protocol.

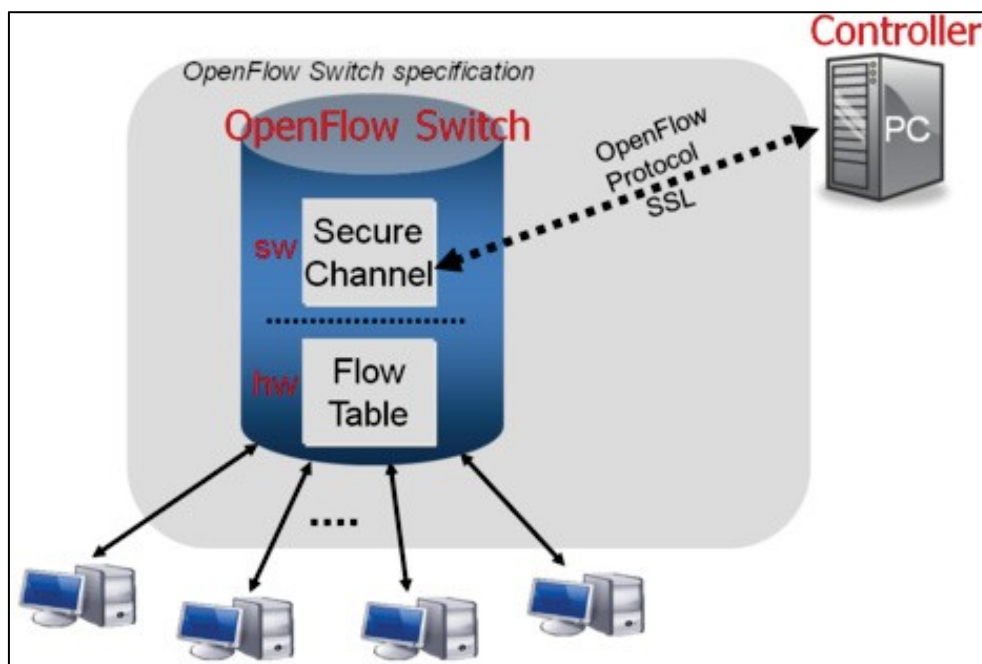


Fig. 4. OpenFlow Commutator

2.5 FLOW TABLES

Using the OpenFlow switch protocol, the controller can add, update, and delete flow entries in flow tables, both reactively (in response to packets) and proactively.

Reactive Flow Entries are created when the controller dynamically learns where devices are in the topology and must update the flow tables on those devices to build end-to-end connectivity.

For example, since the switches in a pure OpenFlow environment are simply forwarders of traffic, all rational logic must first be dictated and programmed by the controller. So, if a host on switch A needs to talk to a host switch B, messages will be sent to the controller to find out how to get to this host. The controller will learn the host MAC address tables of the switches and how they connect, programming the logic into the flow tables of each switch. This is a reactive flow entry.

Proactive Flow Entries are programmed before traffic arrives. If it's already known that two devices should or should not communicate, the controller can program these flow entries on the OpenFlow endpoints ahead of time.

- **Traffic Matching, Pipeline Processing, and Flow Table Navigation**

In an OpenFlow network, each OpenFlow switch contains at least 1 *flow table* and a set of *flow entries* within that table. These *flow entries* contain *match fields*, *counters* and *instructions* to apply to matched packets.

Typically, you'll have more than a single *flow table*, so it's important to note that matching starts at the first flow table and may continue to additional flow tables of the pipeline. The packet will first start in table 0 and check those entries based on priority. Highest priority will match first (e.g. 200, then 100, then 1). If the flow needs to continue to another table, *goto* statement tells the packet to go to the table specified in the instructions.

This pipeline processing will happen in two stages, ingress processing and egress processing as show in Fig. 5.

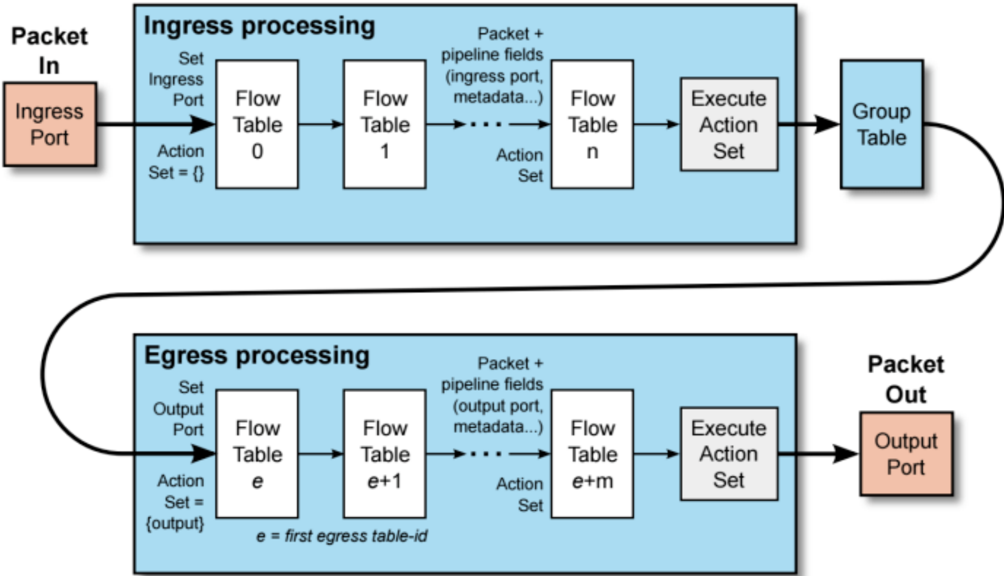


Fig. 5. Traffic Matching, Pipeline Processing, and Flow Table Navigation

If a matching entry is found, the instructions associated with the specific flow entry are executed. If no match is found in a flow table, the outcome depends on the configuration of the Table-miss flow entry.

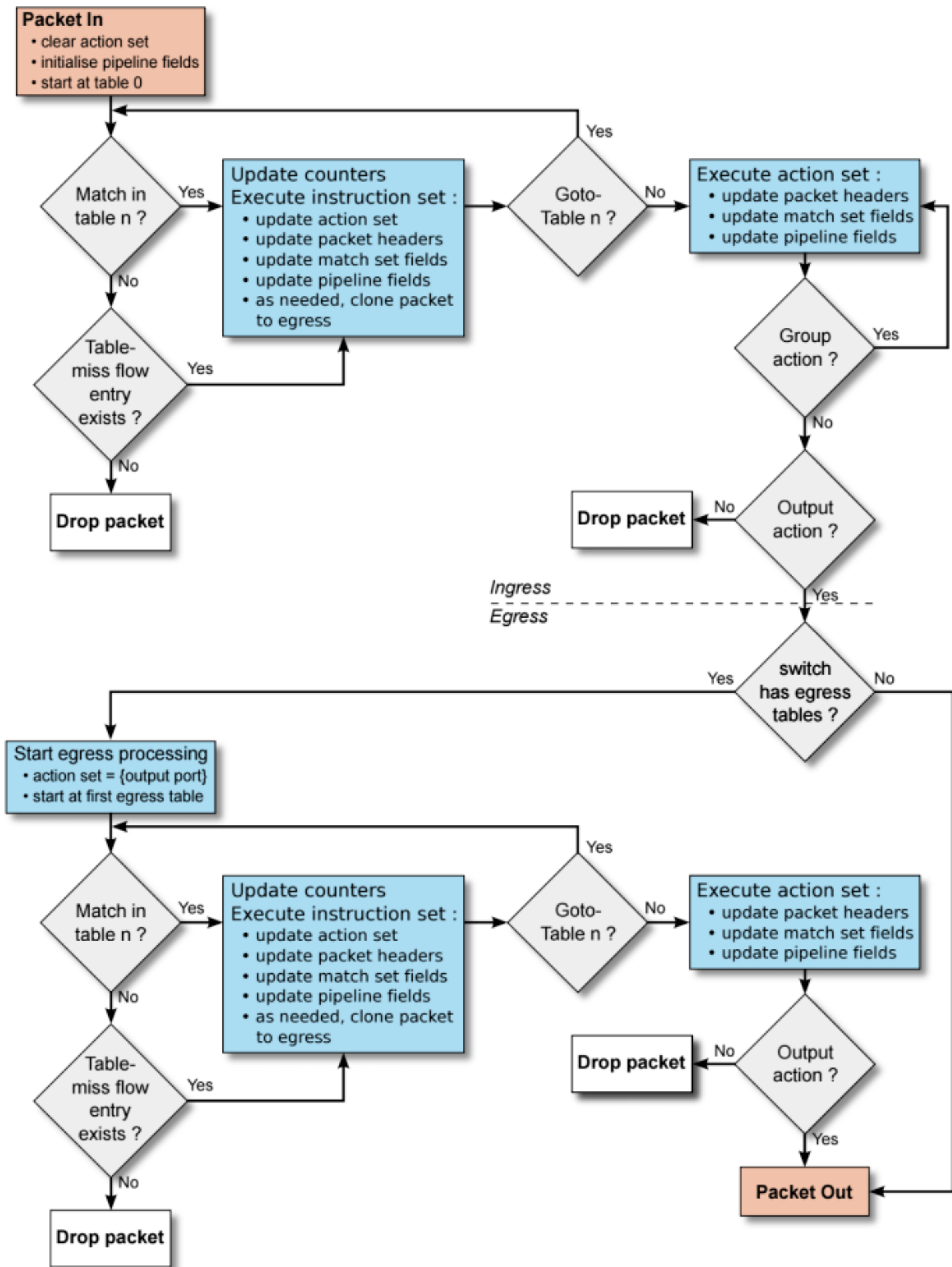


Fig. 6. Table-miss flow entry flowchart

- **Table-miss flow entry**

The Table-miss flow entry is the last in the table, has a priority of 0 and a match of anything. It's like a catch-all, and the actions to be taken depend on how you configure it. You can forward the packet to the controller over the OpenFlow Channel, or you could drop the packet, or continue with the next flow table, as show in Fig. 6.

OpenFlow Ports

"OpenFlow ports are the network interfaces for passing packets between OpenFlow processing and the rest of the network. OpenFlow switches connect logically to each other via their OpenFlow ports…"

There are three types of ports that an OpenFlow switch must
support: *physical* ports, *logical* ports, and *reserved* ports.

Physical Ports

Physical ports are switch-defined ports that correspond to a hardware interface on the switch. This could mean one-to-one mapping of OpenFlow physical ports to hardware-defined Ethernet interfaces on the switch, butt doesn't necessarily have to be one-to-one. OpenFlow switches can have physical ports that are actually virtual, and map to some virtual representation of a physical port. This is similar to the way you virtualize hardware network interfaces in compute environments.

Logical Ports

Logical ports are switch-defined ports that do not correspond directly to hardware interfaces on the switch. Examples of these include LAGs, tunnels and loopback interfaces. The only differences between *physical ports* and *logical ports* is that a packet associated with a logical port may have an extra pipeline field called *Tunnel-ID*, and when packets are received on *logical ports* that require communication to the controller, both the logical port and underlying physical port are reported to the controller.

Reserved Ports

The OpenFlow *reserved ports* specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as "normal" switch processing.

There are several flavors of required reserved
ports: ALL, CONTROLLER, TABLE, IN_PORT, ANY, UNSET, LOCAL.
The CONTROLLER port represents the *OpenFlow Channel* used for communication between the switch and controller.

In hybrid environments, you'll also see NORMAL and FLOOD ports to allow interaction between the OpenFlow pipeline and the hardware pipeline of the switch.

OpenFlow-only Switches vs. OpenFlow-hybrid switches

There are two types of OpenFlow switches: OpenFlow-only, and OpenFlow-hybrid.

OpenFlow-only switches are "dumb switches" having only a data/forwarding plane and no way of making local decisions. All packets are processed by the OpenFlow pipeline, and can not be processed otherwise.

OpenFlow-hybrid switches support both OpenFlow operation and normal Ethernet switching operation. This means you can use traditional L2 Ethernet switching, VLAN isolation, L3 routing, ACLs and QoS processing via the switch's local control plane while interacting with the OpenFlow pipeline using various classification mechanisms.

You could have a switch with half of its ports using traditional routing and switching, while the other half is configured for OpenFlow. The OpenFlow half would be managed by an OpenFlow controller, and the other half by the local switch control plane. Passing traffic between these pipelines would require the use of a NORMAL or FLOOD reserved port. Fig. 7. Show open flow messages.

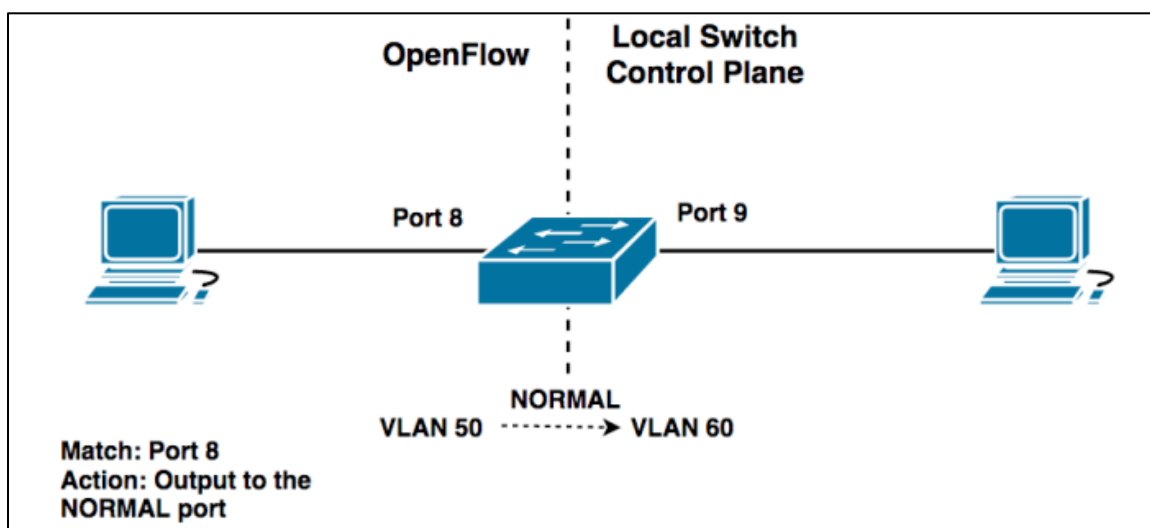


Fig. 7. OpenFlow Messages

OpenFlow Protocol supports 3 message types, each with their own setup of sub-types:

- Controller-to-switch
- Asynchronous
- Symmetric

- **Controller-to-switch Messages**

  Controller-to-switch Messages and initiated by the controller and used to directly manage or inspect the switch. These messages include:
  - **Features** – switch needs to request identity
  - **Configuration** – set and query configuration parameters
  - **Modify-State** – also called ‘flow mod’, used to add, delete and modify flow/group entries
  - **Read-States** – get statistics
  - **Packet Outs** – controller send message to the switch, either full packet or buffer ID.
  - **Barrier** – Request or reply messages are used by controller to ensure message dependencies have been met, and receive notification.
  - **Role-Request** – set the role of its OpenFlow channel
  - **Asynchronous-Configuration** – set an additional filter on asynchronous message that it wants to receive on OpenFlow Channel
- **Asynchronous Messages**

  Asynchronous messages are initiated by the switch and used to update the controller of network events and changes to the switch state. These messages include:
  - **Packet-in** – transfer the control of a packet to the controller
  - **Flow-Removed** – inform controller that flow has been removed
  - **Port Status** – inform controller that switch has gone down
  - **Error** – notify controller of problems
- **Symmetric Messages**

  Symmetric messages are initiated either by the switch or controller and sent without solicitation. These messages include:
  - **Hello** – introduction or keep-alive messages exchanged between switch and controller
  - **Echo** – sent from either switch or controller, these verify liveness of connection and used to measure latency or bandwidth

- **Experimenter** – a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space.
- **OpenFlow Connection Sequence**
- Switch can initiate the connection to the controller's IP and default transport port (TCP 6633 pre-OpenFlow 1.3.2, TCP 6653 post), or a user-specified port. Controller can also initiate the connection request, but this isn't common. TCP or TLS Connection Established. Both send an OFPT_HELLO with a populated version field Both calculate the negotiated version to be used. If this cannot be agreed upon an OFPT_ERROR message is sent. If each support the version, the controller sends an OFPT_FEATURES-REQUEST to gather the Datapath ID of the switch, along with the switch's capabilities.

2.6 SDN CONTROLLERS

A controller is the central component of any SDN infrastructure, as it has the global view of entire network including data plane SDN devices. It links these tools with management applications and performs flow actions determined by application policy among the devices. In this section we present the generic architecture of the controllers, and the evolution towards modern controllers. We also present the classification, comparison, and use case enhancements for 34 different controllers.

A. Architecture of SDN Controllers

The controller in a software defined network, also referred as Network Operating Systems (NOS), is the core and critical component responsible for making decisions on managing traffic in underlying network.

The ideas put forth in the literature for different controllers do not change the basic design of the controller, but rather vary in terms of modules and capabilities. Hence, we find that it is less beneficial for the reader to present individual architectures. Here, as shown in Fig. 8, we present the overall architecture and discuss its various modules.

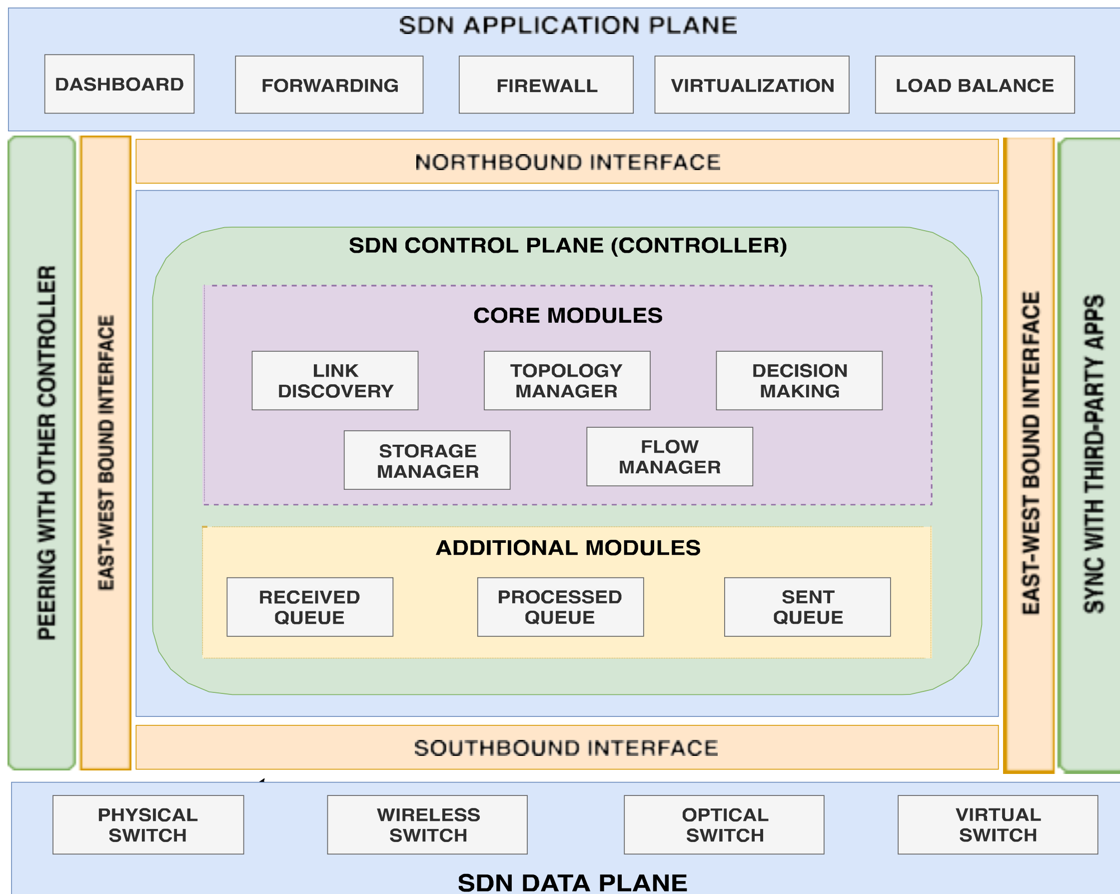


Fig. 8. General Overview of SDN Controller

Controller Core: The controller's core functions are specifically related to topology and traffic flow. The link discovery module periodically transmits external inquiries

Ports that use packet out message. These request messages are returned from the packet in message, which allows the controller to create the network topology. The topology itself is retained by the manager of topology. This provides the module for decision making to find optimal paths between the network nodes. The paths are designed such that during path installation, the various QoS policies or security policies can be implemented. In addition, for the collection of performance information and management of various incoming and outgoing packet queues, the controller can also have a dedicated statistics collector/manager and queue manager. Flow Manager is one of the key modules that interacts directly with the flow entries and flow tables of the data plane. For this reason, it uses the Southbound interface.

Interfaces: The central controller is surrounded by various interfaces for other layers and devices to communicate with. Southbound Interface (SBI) describes a set of rules for processing

This enables the forwarding of packets between forwarding devices and controllers. SBI allows the controller to intelligently include physical and virtual network equipment. OpenFlow

(OF) is the most widely used SBI and is a de facto industry standard. OF is primarily responsible for identifying flows and classifying network traffic based on a predefined set of laws. The controller uses Northbound Interface (NBI) on the opposite end to allow developers to integrate their applications with controller and data plane devices. A number of Northbound APIs are supported by controllers, but most of them are based on the REST API. West Bound Interface (WBI) is used for inter-controller communication. For this reason, there is no common communication interface, so different controllers use various mechanisms. In addition, heterogeneous controllers do not typically interact with one another. The East Bound API (EBI) expands the controller's ability to connect with legacy routers. The most widely used protocol for this purpose is BGP.

B. Qualitative Benchmarking

A detailed view of the different properties of the controllers is given in Table I. We do not address each individual controller ally in the interest of space and the fact that not all initiatives have detailed information on their internal workings. Instead, we present the controllers' characteristics and design choices.

Programming Language: Using various programming languages such as C, C++, Java, Java Script, Python, Ruby, Haskell, Go and Erlang, controllers have been written. In certain instances, using a single language, the entire controller is constructed.

While multiple languages are used in their core and modules in many other controllers in order to deliver successful memory allocation, they can be executed on multiple platforms or most significantly, under some conditions, achieve higher performance.

Architecture: A controller's main design decision is its architecture that can be centralized or distributed. In small scale networks, Centralized controllers are often used, while distributed controllers can spread over several networks.

Of realms. They may also be defined as flat, where all controller instances have equal duties, or hierarchical, where there is a root controller.

Programmable Interface (API): Northbound API (NBI) typically enables the controller to promote applications such as monitoring topology, flow forwarding, virtualization of the network,

Based on the network events created by data plane devices, load-balancing and intrusion detection. Low-level APIs such as the Southbound API (SBI) are on the other hand, responsible

for allowing communication between switches or routers controlled by the controller and the SDN. In addition, multiple controllers from different domains use the east-west API (EWBI) to create peering with each other in a distributed or hierarchical environment. All APIs are not provided by all controllers, and only a few have personalized them for their own unique use.

Platform and Interface: These features define how the controller is implemented to be compliant with a particular operating system. The bulk of controllers are installed on top of a Linux distribution. In addition, in order to configure and display statistical statistics, some controllers provide administrators with graphical or web-based interfaces.

Threading and Modularity: For lightweight SDN deployments, a single-threaded controller is more suitable. In addition, multi-threaded controllers are suitable for industrial applications, such as 5G, SDN-WAN and optical networks. On the other hand, the modularity of a controller enables multiple applications and functionalities to be combined. High modularity enables a controller to execute a task in a distributed environment more quickly.

License, Availability, and Documentation: Most of the Open-Source controllers mentioned in this article are licensed. A few, however have a proprietary license, which means they are only available for research purposes or by special requests. For developers, ongoing maintenance of these controllers is also a difficult job, which is why a lot of them do not receive regular updates. The source code, however is accessible online, allowing anyone to make more improvements according to the requirements. We also found that the majority of them lack adequate documentation when accessing them online. On the contrary, those that are regularly updated have comprehensive and updated documentation for all versions available and even community-based support.

Table I. SDN Controller's Feature Comparison Table

Name	Programming Language	Architecture	Northbound API	Southbound API	EastWestbound API	Supported Platform	Interface	License	Multithreading	Modularity	Consistency	Documentation
Beacon (Erickson, 2013)	Java	Centralized	ad-hoc	OpenFlow 1.0	-	Linux, MacOS, Windows	CLI, Web UI	GPL 2.0	Yes	Fair	No	Fair
Beehive (Yeganeh & Ganjali, 2014)	Go	Distributed Hierarchical	REST	OpenFlow 1.0, 1.2	-	Linux	CLI	Apache 2.0	Yes	Good	Yes	Limited
DCFabric (GitHub, n.d.)	C, Javascript	Centralized	REST	OpenFlow 1.3	-	Linux	CLI, Web UI	LGPL 3.0	Yes	Good	Yes	Fair
Disco (Phemius	Java	Distributed Flat	REST	OpenFlow 1.0	AMQP	-	-	Proprietary	-	Good	No	Limited

et al., 2014)												
Faucet (Stuart, 2016)	Python	Centralized	-	OpenFlow 1.3	-	Linux	CLI, Web UI	Apache 2.0	Yes	-	Yes	Good
Floodlight (Floodlight, n.d.-b)	Java	Centralized	REST, Java RPC, Quantum	OpenFlow 1.0, 1.3	-	Linux, MacOS, Windows	CLI, Web UI	Apache 2.0	Yes	Fair	Yes	Good
FlowVisor (Sherwood et al., 2009)	C	Centralized	JSON RPC	OpenFlow 1.0, 1.3	-	Linux	CLI	Proprietary	-	-	No	Fair
HyperFlow (Tootoonchian & Ganjali, 2010)	C++	Distributed Flat	-	OpenFlow 1.0	Publish and subscribe messages	-	-	Proprietary	Yes	Fair	No	Limited
Kandoo (Hassas Yeganeh & Ganjali, 2012)l	C, C++, Python	Distributed Hierarchical	Java RPC	OpenFlow 1.0-1.2	Messaging Channel	Linux	CLI	Proprietary	Yes	High	No	Limited
Loom (Mentel et al., 2016)	Erlang	Distributed Flat	JSON	OpenFlow 1.3-1.4	-	Linux	CLI	Apache 2.0	Yes	Good	No	Good
Maestro (Maestro, n.d.)	Java	Centralized	ad-hoc	OpenFlow 1.0	-	Linux, MacOS, Windows	Web UI	LGPL 2.1	Yes	Fair	No	Limited
McNettle (Voellmy & Wang, 2012)	Haskell	Centralized	-	OpenFlow 1.0	-	Linux	CLI	Proprietary	Yes	Good	No	Limited
Meridian (Banikazemi et al., 2013)	Java	Centralized	REST	OpenFlow 1.0, 1.3	-	Cloud-based	Web UI	-	Yes	Good	No	Limited
Microflow (Narayanan et al., 2012)	C	Centralized	Socket	OpenFlow 1.0-1.5	-	Linux	CLI, Web UI	Apache 2.0	Yes	-	No	Limited
NodeFlow (NODEFLOW, n.d.)	JavaScript	Centralized	JSON	OpenFlow 1.0	-	Node.js	CLI	Cisco	-	-	No	Limited
NOX (Gude et al., 2008)	C++	Centralized	ad-hoc	OpenFlow 1.0	-	Linux	CLI, Web UI	GPL 3.0	Yes (Nox-MT)	Low	No	Limited
Onix (Koponen et al., 2010)	C++	Distributed Flat	Onix API	OpenFlow 1.0, OVSDB	Zookeeper	-	-	Proprietary	Yes	Good	No	Limited
ONOS (Bianco et al., 2017)	Java	Distributed Flat	REST, Neutron	OpenFlow 1.0, 1.3	Raft	Linux, MacOS, Windows	CLI, Web UI	Apache 2.0	Yes	High	Yes	Good
OpenContrail	C, C++, Python	Centralized	REST	BGP, XMPP	-	Linux	CLI, Web UI	Apache 2.0	Yes	High	Yes	Good

(OpenContrail, n.d.)												
OpenDaylight (OpenDaylight, n.d.)	Java	Distributed Flat	REST, RESTCONF, XMPP, NETCONF	OpenFlow 1.0, 1.3	Akka, Raft	Linux, MacOS, Windows	CLI, Web UI	EPL 1.0	Yes	High	Yes	Good
OpenIRIS (Lee et al., 2014)	Java	Distributed Flat	REST	OpenFlow 1.0-1.3	Custom Protocol	Linux	CLI, Web UI	Apache 2.0	Yes	Fair	No	Limited
OpenMul (OpenMUL, n.d.)	C	Centralized	REST	OpenFlow 1.0, 1.3, OVSDB, Netconf	-	Linux	CLI	GPL 2.0	Yes	High	No	Good
PANE (Ferguson et al., 2013)	Haskell	Distributed Flat	PANE API	OpenFlow 1.0	Zookeeper	Linux, MacOS	CLI	BSD 3.0	-	Fair	No	Fair
POF Controller (D. Hu et al., 2015)	Java	Centralized	-	OpenFlow 1.0, POF-FIS	-	Linux	CLI, GUI	Apache 2.0	-	-	No	Limited
POX (Prete et al., 2014)	Python	Centralized	ad-hoc	OpenFlow 1.0	-	Linux, MacOS, Windows	CLI, GUI	Apache 2.0	No	Low	No	Limited
Ravel (A. Wang et al., 2016)	Python	Centralized	ad-hoc	OpenFlow 1.0	-	Linux	CLI	Apache2.0	-	-	Yes	Fair
Rosemary (S. Shin et al., 2014)	C	Centralized	ad-hoc	OpenFlow 1.0, 1.3, XMPP	-	Linux	CLI	Proprietary	Yes	Good	No	Limited
RunOS (Shalimov et al., 2015)	C++	Distributed Flat	REST	OpenFlow 1.3	Maple	Linux	CLI, Web UI	Apache2.0	Yes	High	Yes	Fair
Ryu (Ryu, n.d.)	Python	Centralized	REST	OpenFlow 1.0-1.5	-	Linux, MacOS	CLI	Apache 2.0	Yes	Fair	Yes	Good
SMaRtLight (Botelho et al., 2014)	Java	Distributed Flat	REST	OpenFlow 1.3	BFT-SMaRt	Linux	CLI	Proprietary	-	-	No	Limited
TinySDN (B. T. de Oliveira et al., 2015)	C	Centralized	-	OpenFlow 1.0	-	Linux	CLI	BSD 3.0	No	-	No	Limited
Trema (Controller, n.d.)	C, Ruby	Centralized	ad-hoc	OpenFlow 1.0	-	Linux	CLI	GPL 2.0	-	Good	No	Fair
Yanc (Monaco et al., 2013)	C, C++	Distributed Flat	REST	OpenFlow 1.0-1.3	yanc File System	Linux	CLI	Proprietary	-	-	No	Limited
ZeroSDN (Kohler et al., 2018)	C++	Distributed Flat	REST	OpenFlow 1.0, 1.3	ZeroMQ	Linux	CLI, Web UI	Apache 2.0	-	High	Yes	Fair

2.7 SDN Applications

SDNs have a wide variety of applications in network environments, and allow for customized control, an opportunity to eliminate equipment

They also simplify the development and deployment of new network services and protocols. According to IDC (International Data Corporation), the SDN market, which is composed of infrastructure equipment, control and virtualization software, applications, etc. network and security, was valued at $12.5 billion in 2020. This section presents some examples of SDN applications.

2.7.1 Multimedia and QoS

Today's Internet architecture is based on sending packets without taking into consideration the type of packets and the quality of the transmission. Multimedia applications such as video streaming, video on demand, video conferencing, WebTV, etc., all require stable network resources and tolerate transmission errors and delays (Sanaei & Mostafavi, 2019). Based on the centralized view of the network offered by SDN allows you to select, depending on the throughput, paths different for the various traffic flows. In (Owens II & Durresi, 2015) the authors provide the Video over Software Defined Networking (VSDN), a network architecture that determines the optimal trajectory by using a network overview. The VSDN was implemented in NS3, and its behavior was analyzed using the complexity of messages between the VSDN and the host/switches. (adding a new flow table, requesting a new service, deleting the service), the bandwidth and transmission delay on the network. VSDN is a research project in course with further improvements to be made.

2.7.2 Cloud computing and data centers

The last few years have been strongly marked by a new technology which is the cloud computing. This technology consists of concentrating resources (storage, computing, etc.) in data processing centers and to offer them as a platform or service to the user (Beshley et al., 2019; Liu et al., 2020) . This trend has increased the amount of data being handled and consequently the explosion in the number of data centers. This situation poses a challenge for the networks that carry the traffic, especially for large operators with several connected data centers. Internet giant Google has been experimenting for 3 years with the benefits of SDN in the management and control of its data centers around the globe. The project consists in building

an experimental core network called B4 (S. Jain et al., 2013b) in parallel with its network operational core that carries Internet traffic. B4 connects the distributed data centers of Google by separating the control and data plans. This allows for rapid deployment of new services and the centralization of the traffic engineering department in an SDN controller, which increased network throughput by 30% and network resource utilization by up to 95%. IT Resource Management is the most important challenge of Cloud networks. SDN is highly regarded as one of the newest solutions to easily configure and manage the Cloud and data centers. Oracle SDN (Macías, S. G., Gaspary, L. P., & Botero, 2020) is an example. system that provides a virtualized data center network. This system dynamically connects virtual machines with network servers. Using the manager interface of Oracle Fabric, virtual network configuration and monitoring is possible anywhere, and deployment of new services such as firewall, load balancing and routing becomes on demand. According to Oracle, their proposal increases by 30 times the performance of the applications.

2.7.3 Security and Privacy

Thanks to SDN's centralized architecture, the controller can detect attacks quickly. and limit their effects. Several researchers have presented mechanisms to detect denial of service attacks (Dos and DDos). For example, in (Juba et al., 2013)the authors propose to isolate the network device for intra-LAN attacks using OpenFlow. In the suggested architecture, the IDS component is in charge of detecting attacks by recognizing the devices not secured and inform the controller. The results showed that this architecture is quite efficient on a real LAN network. A security plan has been proposed in (Hussein et al., 2016). The role of this plan is to inspect all flow exchanges between the switches and the controller in order to prevent IP spoofing attacks. Simulation results show that the introduction of this plan allows to reinforce, in real time, the different levels of security with a minimum of configuration while keeping a very low overload on the controller, since all processing is placed outside the controller. This has allowed to quickly block attacks on the controller and switches and trace their sources. A new IoT security architecture for oT networks by leveraging the underlying features supported by Software Defined Networks (SDN) was proposed by (Karmakar et al., 2019). The security architecture restricts network access to authenticated IoT devices, using granular policies to secure the flows in the IoT network infrastructure and provide a lightweight protocol to authenticate IoT devices.

3. Conclusion

This chapter introduced the SDN paradigm and its most fundamental concepts. One objective was to show that this paradigm consists of a set of layers of abstraction (routing, distribution, and specification) intended to hide the nature of the complex and dynamic of a physical network. It also allowed us to present the context and issues surrounding the last layer of abstraction of this paradigm, to namely the Northbound API. Finally, we gave some SDN applications. These applications show the interest that SDN has aroused among researchers and industrialists in the field of networks.

We believe that this interest is due to the simplification of network equipment architecture and the abstraction of the network. Through the communication interfaces provided by the controller, developers can communicate with the network and propose new services. While the SDN concept offers several advantages such as ease of network management and the introduction of new applications, it raises new security challenges especially at the control plane level since it concentrates all the complexity and intelligence of the network. In the next chapter, we present a literature review on security challenges.

Chapter 2: SDN Security Attacks and Mitigations

1. Introduction

The fundamental characteristic of the SDN architecture, i.e. the separation of the control plane from the data plane, represents a two-sided concept from a security perspective. On the one side, SDN allows the implementation of network security functions and policies in a new and more straightforward approach, which has not been possible in traditional networks (R. Jain, 2012; S. Jain et al., 2013a; Jeong et al., 2012). We refer to this as 'Security via SDN'. On the other hand, the new architecture of SDN brings a range of new potential attack vectors, including attacks against the control plane and the data plane (Akhunzada et al., 2016; Phemius et al., 2014; Shaghaghi et al., 2018). We refer to this aspect of SDN security, i.e. the security of the SDN platform itself, as 'Security of SDN'. While there have been quite a lot of works exploring how SDN can be used to implement different network security functionality, i.e. 'Security via SDN', there has been relatively less attention to the security of the SDN architecture and platform, i.e. 'Security of SDN', which is the focus of this thesis. This chapter first provides a broad overview of key works related to 'Security via SDN', and then discusses key works related to the security of the SDN architecture itself, i.e. 'Security of SDN'. More detailed discussions of works that are specifically relevant to the contributions presented in this thesis are provided in the corresponding chapters.

2. SDN Security Solutions in control Plane

As mentioned before, the different architecture of SDN with its logically centralized control plan creates potentially new vulnerabilities and attack vectors that do not exist in traditional networks, or not to the same extent. In (Ahmad et al., 2015), seven main threat vectors are identified in the SDN design that potentially stands in the way of achieving a secure network environment and keeping all network devices operating properly. Fig. 9 shows an overview of these threats, which are indicated respectively as (1) forged or faked traffic flows, (2) attacks on vulnerabilities in switches, (3) attacks on control plane communications, (4) attacks on vulnerabilities in controllers, (5) lack of trusted mechanisms between controller and

management applications, (6) attacks on vulnerabilities in administrative stations, and (7) lack of trusted resources for forensics and remediation.

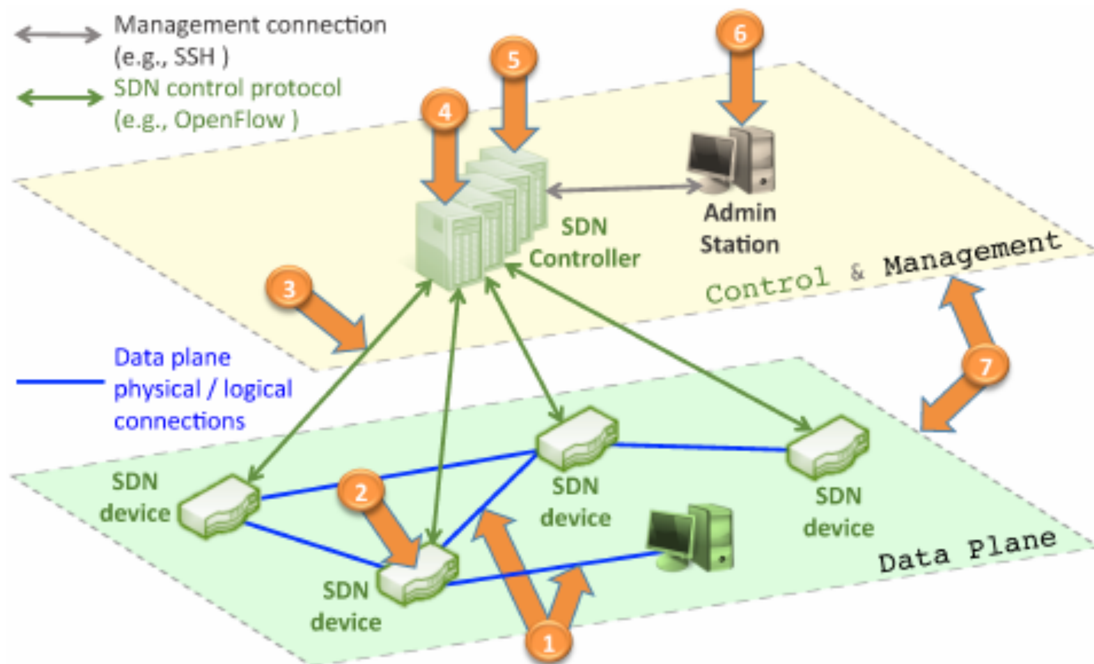


Fig. 9.Security Threat Vectors Map in SDN

Inspired by this, we group our discussion of key works on the security of the SDN architecture into four categories: security of the control plane, security of the data plane, security of the application layer, and security of the southbound interface.

Securing the control plan from various types of threats, which can be resumed as scalability and availability, unauthorized controller access, fake flow rules and DoS attacks, these threats have been a challenge since the introduction of SDN. Many solutions were proposed to solve these issues (Lorenz et al., 2017).

The SDN controller performs an intermediate operation between the network hardware and applications by abstracting the network complexity from applications. Thus, the centralized control architecture enabled by SDN makes it easy to deploy new applications that would collect network information through the controller. Therefore, various network programming languages such as Frentic (Foster et al., 2011) Procera (Voellmy et al., 2012) and NetCore (Monsanto et al., 2012) were proposed to simplify the development of applications in SDN. Furthermore, FRESCO (S. Shin et al., n.d.) was introduced to enable the development of OpenFlow security applications, besides language programming. Different frameworks were suggested to solve the compliance between SDN application and network security policies.

2.1 Lack of access control and accountability

SDN applications need to follow these applications to claim the network specifications regarding the information. To dress this issue, one of the critical solutions is permOF (Wen et al., 2013), which is a well-grained permission system that evolves a set of OpenFlow permission and runtime isolation mechanism applying for permissions. The permission set is sorted into reading, notification, write, and system permissions. Table II shows these permissions and further sub-categories.

Table II. ***PERMOF PERMISSION SET***

Category	Permission
	read topology
Read	read all flow
	read statistics
	read pkt in the payload
	flow mod route
	flow mod drop
Write	flow mod modify hdr
	modify all flows
	set device config
	set flow priority
	pkt in event
Notification	flow removed event
	error event
	topology event
	network access
System	file system access
	process runtime access

The read permission defines what information application can retrieve from the network through the controller. The write explains whether an application can modify or not the state of the controller or switches, notification handle permission of notifying the application if a specific event acquires and the system category manages access to local resources of the operating system.

2.2 Fake flow rules

Fake flow rules remain the first and most critical issue to deal with, any flow rules residing within the controllers is legitimate and expose the controller to propagate false and misleading routing information, hence the importance given to solve this issue among the researcher community.

Many types of research conducted to propose FLOVER, which is a model-checking system that verifies the flow policies against the security policies of a network and make decision which is transferred to the controller.

In the SDN architecture, FLOVER (S. Son et al., 2013) is executed as an OpenFlow application, but logically it is located in parallel to the controller. The controller is modified to request FLOVER's permission on every new flow rule generation or modification. Primarily, the controller communicates flow tables and security policies to FLOVER which contains critical security information to the network preconfigured by the network administrator, so FLOVER can evaluate requested changes accordingly and respond to them. FLOVER also uses the controller to access the current state and the network information's and statistics, like flow rule tables on the switches. Fig. 10 shows the architecture of FLOVER alongside the communication between Open Flow controller, OV switches and the FLOVER application.

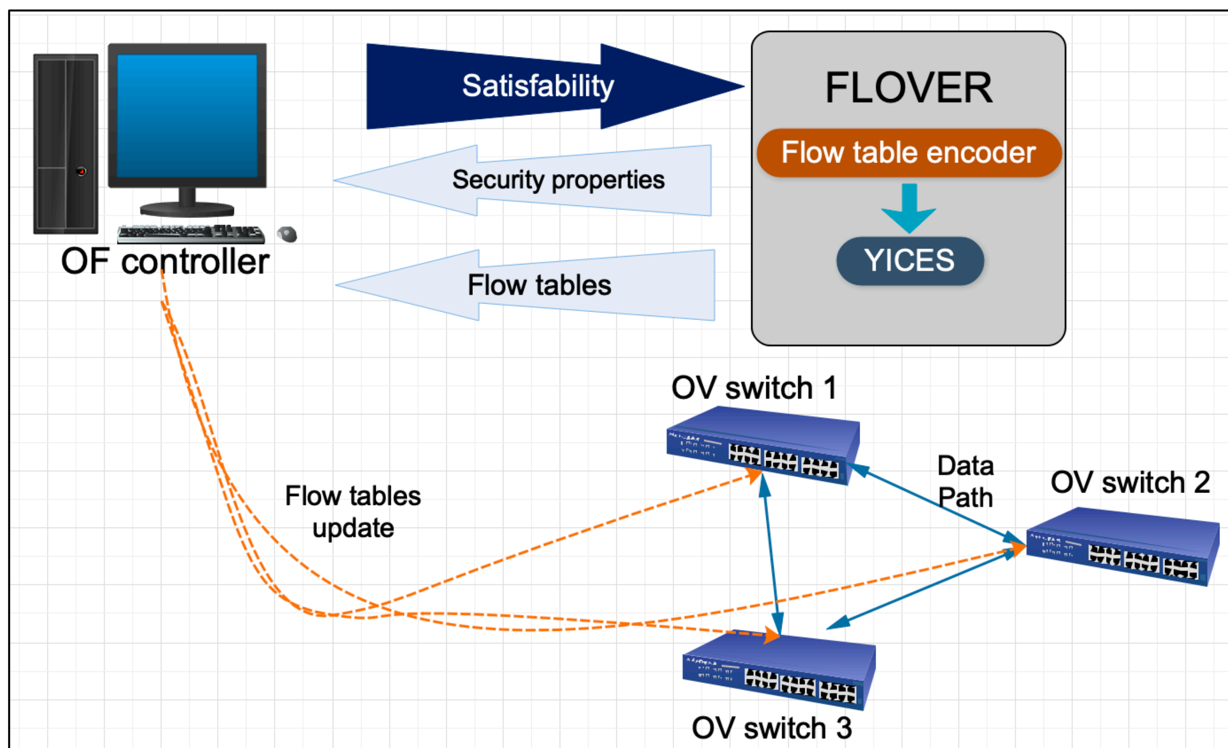


Fig. 10. FLOVER architecture

Flover formalize two key components of Flover framework:

a) Non-bypass Property Representation:

A non-bypass security property states a feature within a specific flow rule set. Formally, a non-bypass security property is a way of first order logic involving a universal quantifier, conditions and an action. An action is either forward or drop. The conditions part of a non-bypass security property is a combination of boolean formula over flow rule set fields. The maximum number of fields is a max of 15. The condition for each field is represented with a boolean expression stipulating a range of non-negative integers because every field consists of a number of bits with a length that varies from 3 to 64

b) Flow Rule Set Representation

Table III is an extract of the flow rule set. Every flow entry (flow rule) has a flow table number specifying where it goes. Every flow rule is attached to conditions over fields and a set of actions. The condition for each field is a set of positive integers, which supports wildcard Formula. The action set for each flow rule require to have one of forward, drop or goto as a finishing action. The action set for a flow rule can have several set commands that alter the packet corresponding the condition of the flow entry. nomatch represents that a flow table has no corresponding flow rule. When there is no matching rule for an incoming packet, an OF-switch sends the packet to the OF controller and requests a new flow rule in the default setting. Thus, for packet p, the possible final action set of FlowRule in FlowTable can be modeled using the following logic forms.

$$FlowRule_i \in FlowTable_k, Cond_i(p) = \bigwedge_{j=0}^{n} F_j(p) \in [iLow_{i,j}, iHigh_{i,j}]$$

$$FlowRule_i(\text{p}) = \{a | a = \ action_{final} \text{ of } FlowRule_i \ s.t \ Cond_i(p) = true \ \}$$

Table III. ***Example OpenFlow rule set used to illustrate coverage and modify violation***

	Condition				
Flow table	Field 1 SRC ip	Field 2 SRC port	Field 3 Dest ip	Field 4 Dest port	Action set
1	5	0.19	6	0.19	{ (drop) }
1	5	0.19	7.8	0.19	{ (set f ield1 10), (goto 2) }
1	6	0.19	6.8	0.19	{ (forward) }
2	10.12	0.19	0.12	0.19	{ (set f ield3 6), (forward) }

Another flow rule conflict management system is FortNox. FortNox (Porras et al., 2012) employs a security enforcement kernel (SEK) to enforce flow controls for active defense against different threats. FortNox is an enhancement engine that is responsible for enforcing and avoiding rule conflicts from separate security authorizations. FortNox uses two protection mechanisms:

- Rule prioritization, which make sure that any new flow rule that contradicts the rules produced by applications is simply overridden because of the highest priority: FortNOX defines by default three authorization roles among applications that produce flow rule insertion requests. These roles may be enlarged with sub-roles, as needed when implemented. The first role is that of IT administrators, whose rule insertion requests are inserted the highest priority within FortNOX's conflict resolution scheme, beside the highest flow rule priority traits sent to the switch. Second, security applications are allocated with deferent authorization role. These security applications generate flow rules that may further restrict the administrator's static network security policy depending on newly perceived runtime threats, such as a malicious flow, an infected internal equipment, a blacklist-worthy external host, or a surfacing mischievous combined traffic pattern. Flow insertion requests generated by security applications are defined with a flow rule priority under that of administrator-defined flow rules. Finally, non-security-related OF applications are assigned the smallest priority. Roles are implemented through a digital signature structure, in which FortNOX configuration is preset with the public keys of several rule

insertion sources. FortNOX increase NOX's flow rule insertion interface to include a digital signature per flow request. If an ancient OF application does not decide to sign its flow rules, those rules are assigned the default role and priority of a common OpenFlow application.

- The conflict detection algorithm affects each new flow rule: To detect a divergence between a newly inserted OpenFlow rule and the existing OpenFlow rule set, the source IP and destination IP addresses, their ports, and wild card members, all rules get converted, including the candidate rule, into a representation called alias reduced rules (ARRs), and then perform the conflict analysis on these ARRs. An alias reduced rule is simply a derivation of the flow rule in which we expand the rule's match criteria to explicitly incorporate set operation transformations and wildcards. An initial alias set is created, containing the first rule's IP addresses, network masks, and ports (where 0 (zero) represents any port). If the rule's action causes a field substitution via a set action, the resultant value is added to the alias set, which is then used to alter the criteria side of the ARR. A pairwise analysis of the candidate ARR to the actual set of ARRs that represent the active rule set. If there is an intersection between both the source and addresses, the combination of the respective sets is used as the subsequent rule's alias set.

In (M. Wang et al., 2016) the authors propose PERM-GUARD a system for managing and authorizing flow rules. PERM-GUARD uses an authentication/authorization model to verify the validity of the controller's flow rules through identity-based signature, this solution effectively filters out unauthorized flow rules created by valid applications and traces their creator in a timely and accurate manner.

PERM-GUARD provides following security functionalities:

- managing the flow-rule-production authorizations set of an authorized SDN application,
- verifying the authenticity of flow rules,
- Authenticating the information's of an SDN application that generate flow rules insertion requests.
- Monitoring irregular behaviors of malicious SDN applications.

To accomplish all these functionalities, PERM-GUARD uses four security elements in controller, which are shown in Fig. 11.

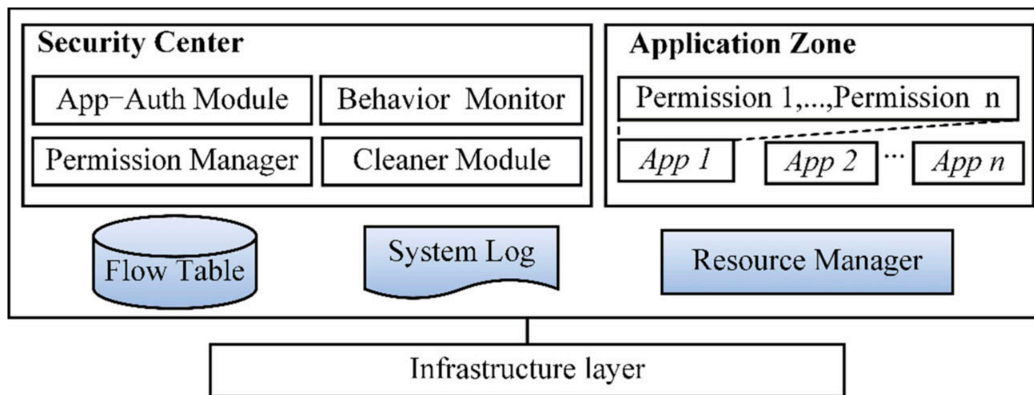


Fig. 11. PERM-GUARD

- App-Auth Module. The app-auth module validate the identity of the application that generate flow rules insertion. Fake flow rules that are generated by unregistered applications will be handled as invalid and droped. If the flow-rule's source is a valid application, then the App-Auth module verifies the generator's flow-rule-production permissions. The new flow rule will be recognized if it is has been inserted in an authorized manner; else, it will be taken care of as invalid rule and rejected.
- Behavior Monitor Module. The behavior monitor component saves unusual behaviors of local and remote SDN applications that interact with the SDN controller, e.g., unauthorized attempts to add or alter flow rules on the SDN controller, prohibited registrations on the controller, etc. The behavior monitor normaly send the monitoring results to the permission manager component periodically.
- Permission Manager Module. The permission manager module configures the flow-rule-production permissions set of an application depending to its behavior records sent by the behavior monitor module. The permission manager will send the latest flow-rule-production permission set of every application to the app-auth module permissions after it is done with modification.
- Cleaner Module. The cleaner module purges the information in temporary buffer created by PERMGUARD to block system cache information from being illegally exploited.

2.3 Authentication and Authorization

The lack of the implementation of any authentication and authorization mechanism by third-party applications compromises the normal network behavior. FRESCO (Monsanto et al., 2012) a security-specific application development framework for Open Flow networks was proposed, FRESCO is a security application development platform that facilitates the exportation of API scripts, which help security experts develop threat detection logic and security monitoring as

programming libraries. Moreover, FRESCO programming framework was presented to help attain rapid design and modular composition of different security mitigation and detection modules using Open Flow.

The FRESCO framework consists of an application layer (which provides an interpreter and APIs to support composable application development) and a security enforcement kernel (SEK, which enforces the policy actions from developed security applications). Both components are integrated into NOX, an open-source openflow controller. FRESCO's application layer is implemented using NOX python modules, which are extended through FRESCO's APIs to provide two key developer functions: (i) a FRESCO Development Environment [DE], and (ii) a Resource Controller [RC], which provides FRESCO application developers with OF switch- and controller-agnostic access to network flow events and statistics.

Developers use the FRESCO script language to instantiate and define the interactions between the NOX python security modules (we present FRESCO's scripting language in Section 4.3). These scripts invoke FRESCO-internal modules, which are instantiated to form a security application that is driven by the input specified via the FRESCO scripts (e.g., TCP session and network state information) and accessed via FRESCO's DE database API. These instantiated modules are triggered (executed) by FRESCO DE as the triggering input events are received. FRESCO modules may also produce new flow rules, such as in response to a perceived security threat, which are then processed by the controller's security enforcement kernel [SEK].

The FRESCO development environment (DE) offers security researchers with valuable information and tools to produce security controls. To achieve this goal, FRESCO DE is designed with two thoughts. Initially, the environment exposes an API that allows the programmers to identify threats and proclaim flow restrictions while abstracting the NOX application and OF protocol complexities. Secondly, the component must relieve applications from the necessity to complete redundant data collection and management operations that are common across network security applications. The FRESCO development environment offers four main functions:

- script-to-module translation,
- database management,
- event management,
- instance execution.

Script-to-module translation: This functionality automatically interprets FRESCO scripts to modules, and creates occurrences from modules, thus abstracting the implementation complexities of generating OF controller extensions. It is also accountable for validating the registration of modules. Registration is performed via a registration API, which allow an authorized administrator to generate a FRESCO application identifier (ID) and an encryption key pair. The developer includes the newly registered application ID into the FRESCO script, and then encrypts the script with the provided private key. The naming convention of FRESCO applications includes the application ID, which is then used by FRESCO to make correspondence between the appropriate public key and the application. Furthermore, to registering modules, the module manager synchronizes how modules are associated to each other and provides input and event values to each module.

To streamline the development of security applications, FRESCO stipulates its own script language to support developers in creating security functions from fundamental modules. The textual language, sculpted after the Click language, necessitates the use of six variables per instance of modular element:

- type.
- Input.
- Output.
- Parameter.
- Action.
- Event.

To design modules through a FRESCO script, programmers must create firstly an instance of a module, and this new instance information is declared in type variable. For example, to use a function that achieves a specific action, a developer makes a new instance of the ActionHandler component (represented as type: ActionHandler inside a FRESCO script).

Developers can define a script's input and output, and schedule events for it to process by defining the script's input, output, variables, and event parameters. Multiple value sets for these parameters can be defined by using a comma as separator.

Defining an instance is very comparable to initializing a function in C or C++. A module began with the module name, two parameters for representing the amount of inputs and outputs, and left braces ({). The numbers of inputs and outputs are used to sanity check the script

through module translation. Like C or C++ functions, a module implementation ends with a right brace (}).

The action parameter symbolizes actions performed by a module depending on some conditions, where the conditions are specified by one of the inputs. There may be several conditions separated by semicolons in the action.

In (M. Wang et al., 2016), the authors proposed a modular SDN security-control communication architecture, KISS, with innovative solutions in the context of key distribution and secure channel support. Besides, they suggest iDVV, the integrated verification value of the device, a code protocol for generating a secret code that is deterministic but indistinguishable from chance. This allows local but synchronized generation/verification of keys at both ends of the channel even by message. IDVV should make a significant contribution to both the robustness and simplicity of authentication and secure communication problems in SDN.

In order to protect and authenticate requests between two networking devices, integrated device verification values (iDVVs) are sequentially generated. The generator is constructed such that the indistinguishability-from-random and determinism properties are in its output series. As a result, on both ends of the channel, the same sequence of random-looking secret values is generated, enabling the stable, decentralized generation/verification of per-message keys at both ends. However, there is no way an opponent can know, predict or produce an iDVV if the seed and key initial values and the state of the generator are kept secret.

In other words, an iDVV is a specific secret value created by an A device (for example, a forwarding device) that can be checked locally by another B device (e.g. a controller). To satisfy the needs of SDN, the iDVV generation is made versatile. Therefore, iDVVs can be generated: (a) on a message basis; (b) for a sequence of messages; (c) for a specific time interval; and (d) for a single session of communication. The main benefits of iDVVs are their low cost and the fact that they can be produced locally, I.e., without any prior arrangement needing to be formed.

The iDVV generation is simpler and faster than standard KDF algorithms and related solutions such as HKDF (RFC 5869).

As discussed before, the combination of two SDN devices, e.g., forwarding device such as switches and SDN controller, occurs through the use of KDC, under the protection of the secret keys gained from registration (Kkf , resp. Kkc). The result of the association protocol is the supply of two random secrets to both devices: a seed seedij, and an association key keyij . The

iDVV mechanism is bootstrapped by implementing these two secret values in both the controller and the switch, to run the iDVV generation algorithms.

Note that setting up and generating an iDVV values are executed in a deterministic way, so that this is achieved locally at both ends. Nevertheless, as iDVVs will be implemented as keys by cryptographic primitives such as encryption functions or MAC, they have to be fuzzy from random. Hashing primitives are normal choices for iDVV algorithms, since they provide indistinguishable-from-random variables if one or more of the input variables are identified only by the source and the destination. This demonstrate why it is important that seed and association key are conducted encrypted and therefore known only to the connecting devices. Besides, all of the variables seed, key, and idvv in the algorithms below should have the same length in order to avoid information leakage, which the 256 bits is chosen at iDVV design.

After the bootstrap with the first idvv value, the idvv_second function is invoked on-demand (over, synchronously at the two sides of the channel) to autonomously produce authentication or encryption keys that will be used for securing the communications. The key stays the only constant shared secret between the devices. The seed progresses to a new indistinguishable-fromrandom value every time idvv_next is called to create a new iDVV. The new seed is the result of a hashing primitive H over the actual seed and actual idvv (line 2). The new idvv, output of function idvv_next, is the resulut of a hashing primitive H over the combination of the new seed and association key key.

Stable communication protocols are ambiguous in the iDVV mechanism and can be used in a variety of ways, in a number of protocols, as a key-per-message or key-per-session, etc. The only main issue about creating iDVV is holding it 1We omit the device-identifying subscriptions in the variables for readability. 3 The two ends of the channel are aligned. So, in this respect, we are presenting some suggestions.

2.4 Scalability & availability

SDN controller is responsible for handling all network requests and events. As the size of the network keeps growing the controller can reach a state of a bottleneck. Benchmarking of the SDN NOX (Gude et al., 2008) controller shows that it can support up to 30k demand this rate can be a serious issue to networks with high requests.

To assuage this matter levelling parallelism in multicore systems was proposed by Tootoonchian et al. (Bianco et al., 2017) their approach showed that minor alterations to the NOX controller increase its performance by an order of greatness on a single core. It means that

a single controller can support a further larger network, given adequate controller channel bandwidth with acceptable latency. More improvement can be achieved by reducing the number of requests sent to the controller using DIFANE (Gude et al., 2008) unique switches, called authority switches. These switches are used to discharge the tasks of the controller and handle the packets in the data plane. Most micro-flows are handled in the data-plane to reduce the need to request the control-plane and increase the scalability.

Another way to resolve the scalability issue is to distribute the control plane tasks to multiple controllers. ONIX (Koponen et al., 2010) delivers a distributed control plane platform while preserving the reliable network-wide state, as shown in Fig. 12.

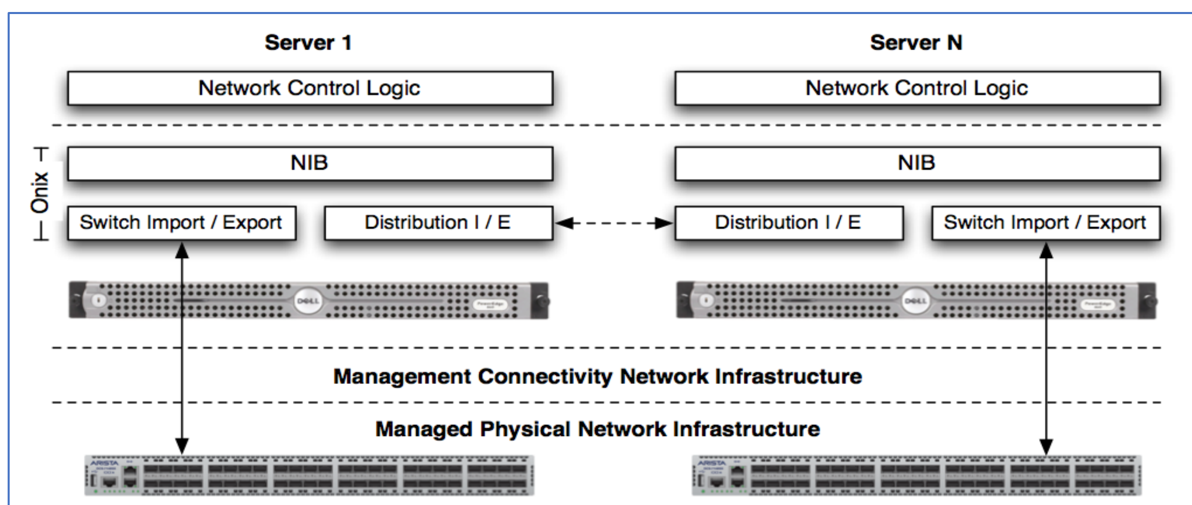


Fig. 12. Two ONIX controller coordinating and their views of the underlying network state

There are four components in a network controlled by Onix, and they have very distinct roles.

- Physical infrastructure: This component contains network switches and routers, also any other network components (such as load balancers) that support an API allowing Onix to read and write. State controlling the element's comportment (such as forwarding table records). These network components don't have to run any programs further than that mandatory to support this interface and accomplish basic communications.
- Connectivity infrastructure: The connectivity between the physical networking equipment's and Onix (The control part) transits the connectivity infrastructure. This control passage may be used both in-band (where the control management shares the same forwarding equipment's as the data traffic on the network), or out-of-band (where a distinct physical network is implicated in handling control traffic). The connectivity infrastructure needs to support two-way communication between the Onix controller

and the open-flow enabled switches, and optionally maintain convergence on link breakdown. Common routing protocols (like as IS-IS or OSPF) are appropriate for establishing and maintaining forwarding operations in the connectivity infrastructure.

- Onix: Onix is a distributed system which turns on a cluster of one or more physical servers, each of which may run several Onix instances. As an SDN controller, Onix is incharge of providing the control logic and programmatic access to the network (reading and writing network state). An Onix instance is also responsible for disseminating network status to other instances within the cluster in order to scale to very large networks (millions of ports) and to provide the necessary resilience for production deployments.
- Control logic: On top of Onix's API, the network control logic is implemented. The desired network behavior is defined by this control logic; Onix simply provides the primitives required to access the appropriate network state.

Many other solutions to the scalability issue were presented in the literature such as HyperFlow (Tootoonchian & Ganjali, 2010), which consists of a physically distributed and logically centralized control platform.

The HyperFlow framework is a C++ NOX application created to ensure a single network-wide view of all controllers. An instance of the HyperFlow application is run by each controller. The implementation of HyperFlow includes minor modifications to the core controller code, primarily to provide sufficient hooks for intercepting commands and serializing events, as show in Fig. 13.

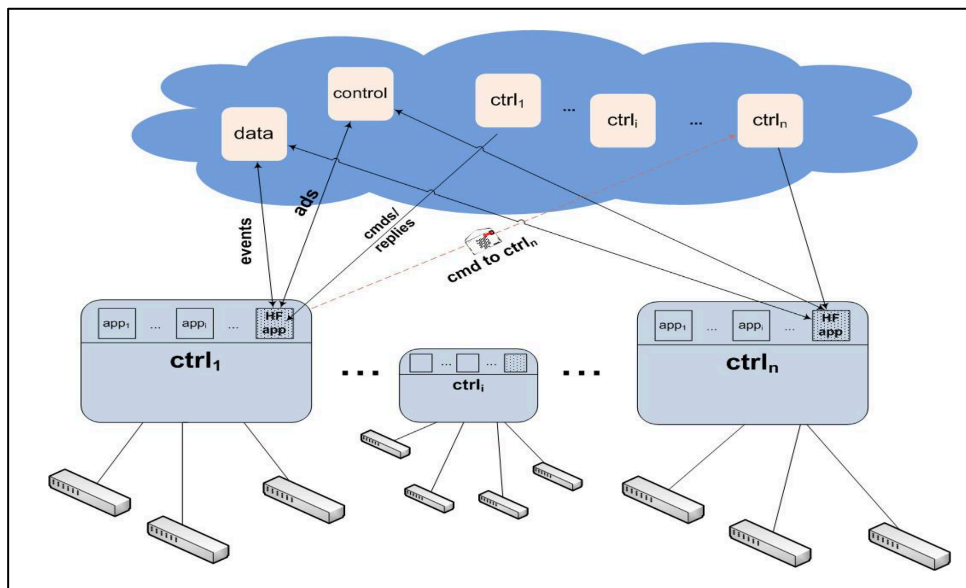


Fig. 13. High-level overview of HyperFlow

Each controller runs NOX atop the HyperFlow program, subscribing to the control, data and its own canal in the publish/subscribe framework (depicted with a cloud). Events are released to the data channel and periodic advertisements from the controller are sent to the control channel. Controllers publish commands targeting a controller directly to its channel. In the source controller, the answers to the commands are written.

Hyper flow operates as described below:

- Initialization: Upon launching NOX, the HyperFlow program launches the WheelFS client and storage services, subscribes to the data and control channels of the network, and begins regularly advertising itself in the control channel. The advertising interval among controllers in a network must be greater than the highest round-trip time. The advertising message contains the controller's information, including the switch identifiers it directly controls.
- Publishing events: The HyperFlow framework collects all of the built-in (OpenFlow message events) NOX events as well as the events that are recorded with HyperFlow by the applications. It then selectively serializes (using the library of Boost serialization) and publishes those that are created locally and affect the state of the controller. To this end, applications must be configured to tag incidents that impact their status. In addition, the parent event of any non-built-in event they cause should be defined by applications. This allows HyperFlow to track and propagate each high-level event back to the underlying lower-level event instead. Using this approach, we make sure that the number of propagated events is restricted by the number of OpenFlow message events that the local controller produces. The name of the published messages includes the source controller identifier and the publisher's local event identifier. This method effectively partitions the message namespace between controllers and prevents write conflicts from occurring. Moreover, Hyperflow have a cached copy of a message (file) never becomes stale. Therefore, Hypeflow instructs WheelFS to alleviate consistency by using semantic signals. We note that this requires that a replication policy suitable for network configuration be established by the network operator.
- Both message forms include the publisher controller ID id of the publisher (ctrlid). Events and commands also include the publisher's locally generated event identifier (eventid). Commands also include the switch identifier that the command is aimed at. Ments and fetches stored file copies as soon as possible from nearby controllers.

- Replaying events: All published events are replayed by the HyperFlow framework, since source controllers selectively filter out and only publish the events required to recreate the application state on other controllers with the support of apps. The HyperFlow program deserializes and fires it upon receipt of a new message on the network data channel or the controller's own channel.
- Redirecting commands targeted to a non-local switch: Only the switches connected to it directly can be controlled by a controller. The HyperFlow application intercepts when an OpenFlow message is about to be transmitted to those switches and publishes the command to the network control channel in order to program a switch that is not under the direct control of the controller. The name of the published message implies that it is a command and also includes the identifier of the source controller, the identifier of the destination switch and the identifier of the local command (similar to the event message identifier).
- Proxying OpenFlow messages and replies: The HyperFlow application collects command messages that target a switch under its control (identified in the name of the message and sends them to the destination switch. The HyperFlow program manages the mapping between the message transaction identifiers (xid) and the source controller identifiers to route the replies back to the source controller. The HyperFlow application tests the locally generated xid OpenFlow message events of the controller. If the event xid is located in the map of the xidcontroller, the event is prevented from being further processed and released to the data channel of the network. The message name includes all identifiers for the controller. Upon receipt, the original source controller picks up and replays the incident.
- Health checking: The HyperFlow program listens to advertisements for the controller in the network control channel. If a controller does not re-advertise itself, it is believed to have failed for three promotional periods. For every switch that was attached to the failed controller, the HyperFlow application fires a switch leave case. Upon controller failure, HyperFlow configures the failed controller-associated switches to connect to another controller. Alternatively, 4 nearby controllers may serve to take over the IP address as a hot standby for each other.

Shin et al (J. Shin et al., 2017) proposed IRIS-HiSA, a cluster architecture for distributed controller, it’s the main objective is to support uninterrupted load balancing and failover with horizontal scalability, as is done in existing work, but one of IRIS-HiSA's distinctive features

is to provide transparency between the data plane and the switches. Thus, the switches do not need to know the internal details of the controller cluster, and they simply access the same way a single controller is accessible.

IRIS-HiSA has two main goals:

1. Provide high scalability and availability: IRIS-HiSA utilizes a pool of instances of a distributed controller to provide the SDN controller with ample scalability and availability. All controller instances run in active mode for high scalability, and the traffic loads among the controller instances are well balanced to optimize the usefulness of computational and networking resources. For high availability, seamless failover is provided by IRIS-HiSA. The switches connected to them are seamlessly migrated to the other controller instances, even though some of the controller instances go down.
2. Allow transparent access to the controller cluster: The IRISHiSA controller cluster appears as a single logical controller for straightforward access to a controller cluster by covering the internal information of the controller cluster. Using the representative IP address of the controller cluster, switches may thus connect to the controller cluster without knowing the number of instances of the controller or the address of each instance of the controller. As IRIS-HiSA offers smooth load balancing and failover, internal information such as load balancing and failover need not be understood by the switch.

As shown in Fig. 14, The IRIS-HiSA architecture consists of a pool of examples of controllers, a HiSA controller, and a HiSA switch. The HiSA switch connects both the controller instances and the HiSA controller, and there is no logical restriction to the number of controller instances.

By tracking the state of both the controller instances and switches, the HiSA controller supports load balancing and failover.

An OpenFlow capable switch is a HiSA switch, and it makes the controller cluster seem to be a single logical controller. For example, by using the representative IP addresses of the controller cluster, switches connect to the controller cluster via a HiSA switch. It queries a HiSA controller with a PACKET IN message because a HiSA switch does not have any matching rules for link setup messages like a SYN packet. Upon receiving a PACKET IN message, the HiSA controller uses the load balancing algorithm to assign the control instance to manage the new link. In order to change the IP/MAC packet header, it also sends flow rules to the HiSA switch, which makes a direct link between the switch and a particular instance of the controller.

The HiSA controller also enables switches to be seamlessly transferred to another controller if the instance of the controller to which the switches are attached fails. The controller cluster therefore appears, from the perspective of a switch, as a single logical controller fitted with good robustness and high scalability.

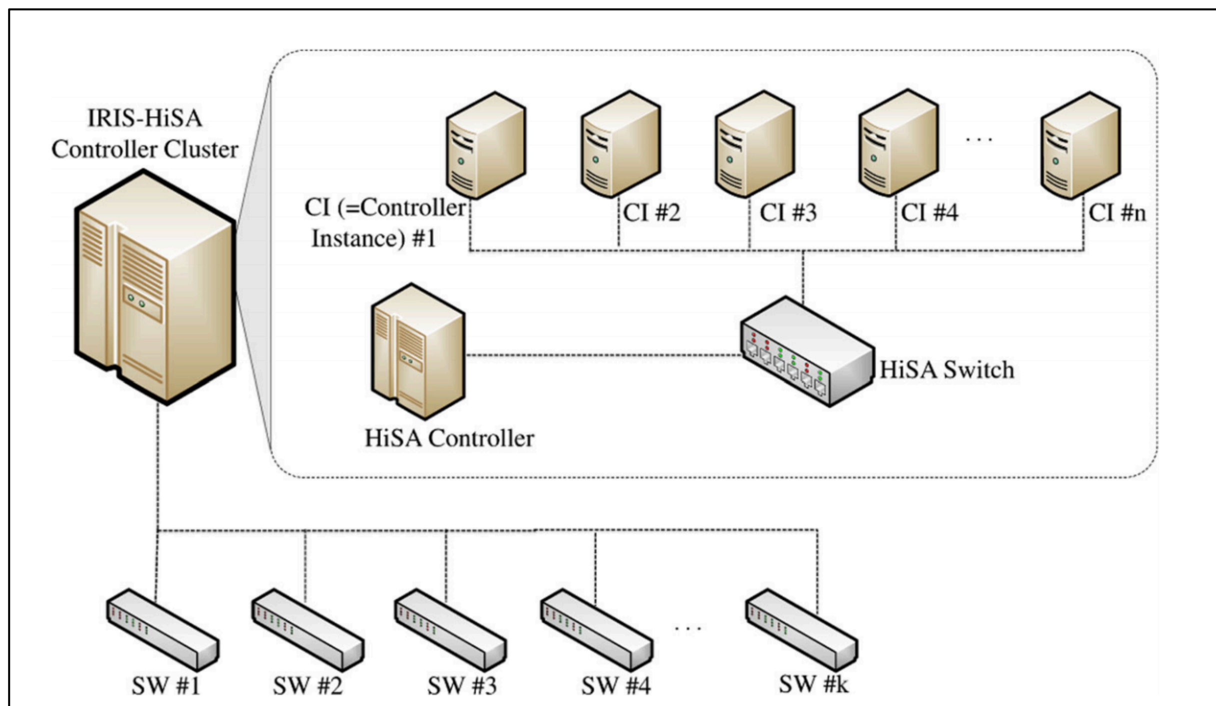


Fig. 14. The architecture of IRIS-HiSA controller cluster

2.5 Unauthorized controller access

SDN Applications access the controller for a wide range of reasons, it's primordial to ensure that these applications work within their respective perimeter with the legitimate functional requirement. For that, securing the controller from the malicious application must be ensured.

SE-Floodlight (Ahmad et al., 2015) controller was proposed as an extended version of the Floodlight controller. SE-Floodlight introduced several security enhancement methods:

- Privilege separation by adding a secure programmable northbound API, this enables SE-Floodlight to work as a truly independent mediator between the application layer and the data plane.
- A runtime integrity validator of the modules that generate flow rules. The runtime copy of the local flow-rule producer is compared to the original image installed by the administrator.
- A Rule Reduction (ARR) algorithm that manages inline rule-conflict detection.

- Role-based conflict resolution by comparing the authoritative roles of the producers of the conflicting rule
- PACKET_OUT Control: packet control generated by OpenFlow apps can be blocked by the administrator.
- Security Audit: is a subsystem that tracks all security events

SE-Floodlight incorporates a Security Compliance Kernel (SEK) into the Floodlight Controller for the purpose of mediating all operations of data exchange between the application layer and the data plane. The SEK applies the application to the data plane mediation scheme shown in Table I for each process. In Table I, the Minimum Authorization column identifies the minimum function that must be delegated to an application to perform the procedure. SE-Floodlight implements a hierarchical authorization function system with three default authorization functions, as discussed in the previous Section. The lowest authorization function, APP, is primarily intended for traffic engineering applications (non-security-related) and provides ample permission for most such applications for flow control. The Security Authorization function, SEC, is designed for applications implementing security services. For applications like the operator console app, the highest authorization function, ADMIN, is intended.

The aim of the mediation service is to provide for a given SE-Floodlight deployment a configurable permission model in which both the collection of roles can be expanded and their permissions can be customized for each newly specified role.

The default Mediation Policy that is allocated to each available contact between the application layer and the data plane is provided in column 3 of Table IV. First, the inherent objective of any OpenFlow application is the ability to define or override existing flow policies within a switch (row 1). However as defined in Section IV-C, SE-Floodlight incorporates Rule-based Conflict Analysis (RCA) to ensure that each candidate flow rule submitted does not conflict with an existing flow rule whose author is aligned with a higher authority than the candidate rule's author.

Table IV. A summary of control layer mediation policies for data flows initiated from the application layer to data plane (A to D) and from the data plane to application layer (D to A)

Flow Direction	Data Exchange Operation	Mediation Policy	Minimum Authorization
01: A to D	Flow rule mod	RCA (Section IV-C)	APP

02: D to A	Flow removal messages	Global read	APP
03: D to A	Flow error reply	Global read	APP
04: A to D	Barrier requests	Permission	APP
05: D to A	Barrier replies	Selected read	APP
06: D to A	Packet-In return	Selected read	APP
07: A to D	Packet-Out	Permission	SEC
08: A to D	Switch port mod	Permission	ADMIN
09: D to A	Switch port status	Permission	ADMIN
10: A to D	Switch set config	Permission	ADMIN
11: A to D	Switch get config	Permission	APP
12: D to A	Switch config reply	Selected read	APP
13: A to D	Switch stats request	Permission	APP
14: D to A	Switch stats report	Selected read	APP
15: A to D	Echo requests	Permission	APP
16: D to A	Echo replies	Selected read	APP
17: D to A	Vendor features	Permission	ADMIN
18: A to D	Vendor actions	Permission	ADMIN

A rule conflict occurs in OpenFlow when the candidate rule permits or disables a network flow that is otherwise forbidden (or permitted) by the current rules. Conflicts are either direct or indirect within OpenFlow. A direct dispute exists when an existing rule is contravened by the candidate rule (e.g., an existing rule forwards packets between A and B while the candidate rule drops them). If the candidate rules logically pairs with pre-existing rules that contravene an existing rule, an indirect dispute arises.

Event alerts used to monitor the status of the flow table provide a second class of operations. These operations do not alter network policies, but provide knowledge required to make informed flow management decisions for traffic engineering applications. These operations are described by the default permission model as two types of public reading. The Global Read reflects events in the data plane that are streamed to all applications involved in receiving them (rows 2 and 3). Chosen read operations apply to individual events for which an application can register to receive switch state-change alerts via the controller (rows 5, 6, 12, 14, and 16). Selected read alerts are responses to permission-protected behaviors that will be addressed next. The Packet-In notification is an exception in that it is received in response to the insertion of the flow rule (vetted via RCA) that activates the switch to alert the application when the packets meeting the flow rule requirements are received or when no matching flow rule is found.

The third class of behaviors includes those needing specific permission (Rows 4, 7-11, 13, 15, 17, and 18). Such activities either make direct adjustments to the network flow policies enforced by the switch or allow the operator to the SEK empowers an OpenFlow stack from the bottom of Fig.15 to serve multiple applications in parallel, implements control layer arbitration when apps generate contradictory flow logic, and imposes the limitations of application permission mentioned in Table IV. The SEK extends Floodlight with five key components.

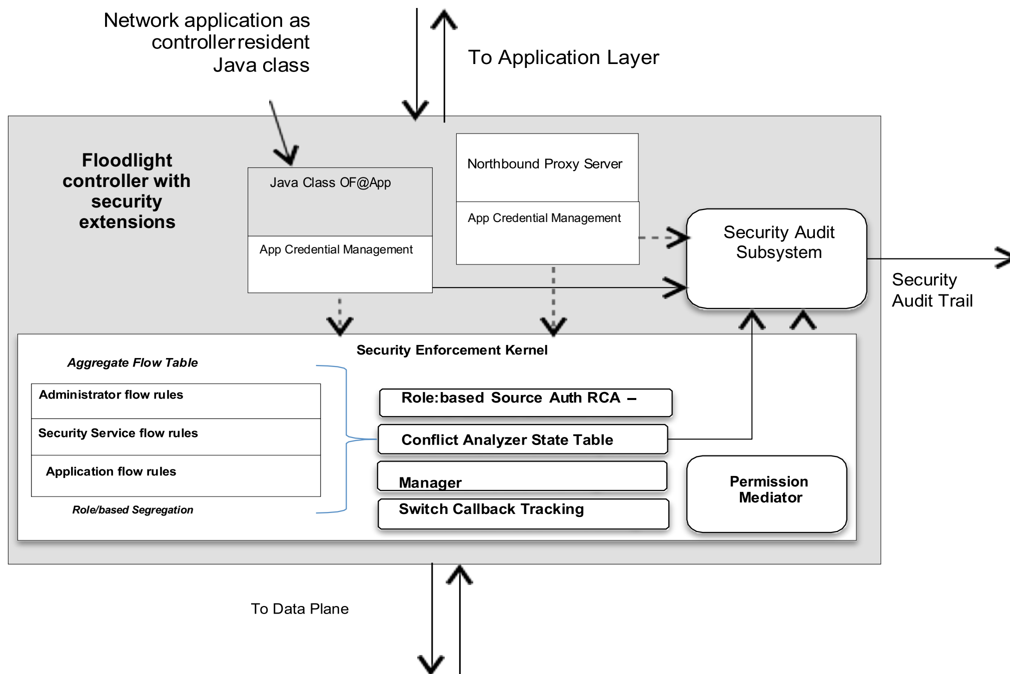


Fig. 15. Floodlight controller with security extensions

A *Role-based Source Authentication* module provides digital signature validation for each flow-rule insertion request to temporarily restrict the priority of a candidate flow rule based on the operating position of the application.

In recent works, (Wu et al., 2019) proposed a switch migration scheme by adopting noncooperative game to improve the control plane scalability in SDN, by designing a novel load balancing monitoring scheme to detect the load imbalance between controllers and trigger migrating switches. Then, the authors used a noncooperative game among controllers to decide switch migration to get the maximizing overall profits.

2.6 DoS attacks

The centralized nature of SDN is revealed to be a single point of breakdown that can be exploited by one of the Internet's most old, high risk and major security threat known as

Distributed Denial of Service (DDoS) Attack. A DDoS attack is a dispersed and harmonised attack that begins from multiple network devices. Essentially, the strategy of this attack is to send a huge volume of spoofed IP packets from disparate points in order to make the network resources unattainable to legitimate users. Over recent years, the attackers have got smarter and have been constantly enhancing and using advanced DDoS attack methods to inflict more economical and financial costs.

While it is a standard security issue, DDoS attack mitigation is still a severe threat to almost every technology , Braga et al presented in (Braga et al., 2010) a detection method that consists of self-organizing maps (SOMs) to identify abnormal/injected flows, as shown in Fig. 16.

The proposed solution consists of three modules:

- A flow Collector module that is responsible for gathering flow from switches.
- A feature Extractor module that extracts relevant data distinguishing DDOS attacks.
- A classifier module analyzes extracted data then classifies them as normal or abnormal.

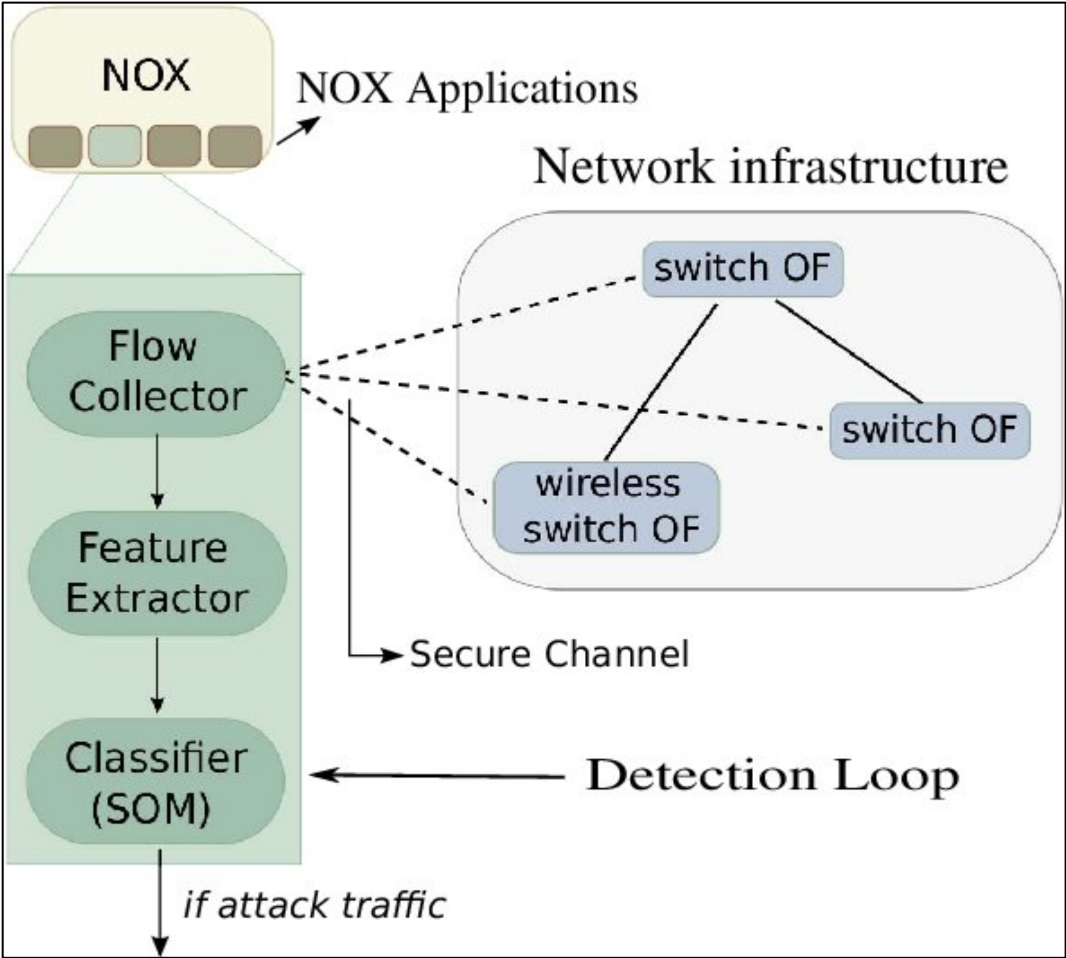


Fig. 16. SOMs detection process

Dridi et al. (Dridi & Zhani, 2016) suggested SDN-Guard, a new system that can effectively protect SDN networks against DoS attacks dynamically by potential redirection of malicious traffic, flow timing adjustment and aggregation of flow rules. The solution relays on four modules (Flow management module, Rule aggregation module, Monitoring module).

De Assis et al. (Assis et al., 2017) proposed a stand-alone DoS / DDoS defensive approach for SDN called Game Theory (GT) Game Theory which is an analytical tool that can model negotiation situations and deal with many problems in different areas -Holt-Winters for Digital Signature (HWDS), which combines detection and anomaly identification provided by an HWDS system with a decision-making model based on GT.

Yunhe et al. introduced in (P. Porras, S. Cheung, M. Fong, K. Skinner, 2015) a system named Software-Defined Anti-DDoS divided into four modules detection, a Trigger Module, Detection Module, Traceback Module, and Mitigation Module, respectively. These modules correspond to four states:

- **Init State**: It is the primary state of the SD-Anti-DDoS system. A packet_in trigger is used to program attack detection in this state. If no abnormal event was found, it will remain in the Init State. Else, the system will run a Detection State.
- **Detection State**: This state is handling the detection of a DDoS attack. The attack detection will start to verify the existence of a DDoS attack in the network.
- **Traceback State**: If the detection module detects a DDoS attack, the Traceback State will start and the system will try to trace the attack path and the attack origin switch.
- **Mitigation State**:The Mitigation module will try to stop the threat from its source and will clean the infected switch from malicious flow entries.

Fig. 17. Illustrate the four state of the above Anti-DDoS attacks.

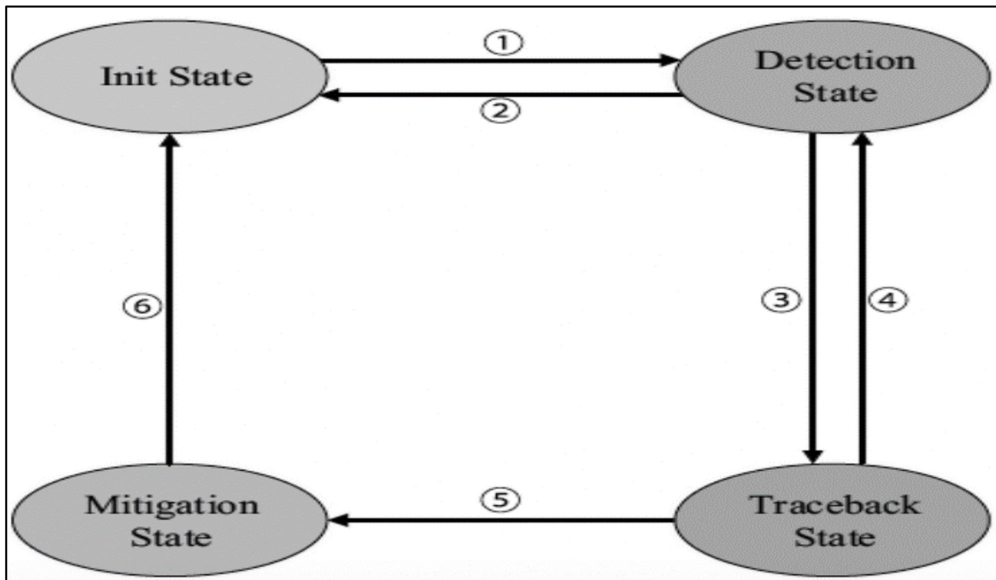


Fig. 17. SD-Anti-DDoS four state diagram

In recent works, (Phan & Park, 2019) propose an efficient solution to tackle DDoS attacks in the SDN-based cloud environment. The authors proposed a new hybrid machine learning model based on support vector machine and self-organizing map algorithms to improve the traffic classification. The model introduce an enhanced history-based IP filtering scheme (eHIPF) to improve the attack detection rate and speed. Finally, the authors present a novel mechanism that combines both the hybrid machine learning model and the eHIPF scheme to make a DDoS attack defender for the SDN-based cloud environment.

2.7 Malicious flow rules

Changing the flow rules within switches must be controlled. FortNox platform allows the NOX controller to check flow rules contradiction instantaneously, then decide the generated flow rules. Blocking flow rules insertion request is achieved using a conflict analysis algorithm when a security application adds a flow rule. FortNOX restricts other applications from inserting conflicting flow rules in the same OpenFlow network.

3. CONCLUSION

This chapter summarizes some of the key proposals and works that discuss potential security threats of SDN technology, at the control plane, data plane, application layer, indicating the key contribution as well as the proposed attack mitigation methods.

None of the works discussed in this chapter have paid close attention to the potential security vulnerabilities of the flow rules in SDN controller. In the upcoming chapters, we will present the thesis contributions to mitigate flow rules attacks and vulnerabilities.

2nd Part: Contributions

Chapter 3: Enhanced Solid-Flow: An Enhanced Flow Rules Security Mechanism for SDN

1. INTRODUCTION

THE SDN - Software-Defined Networking - is the hot topic that shakes the world of the network in recent years. The SDN is recognized today as an architecture for opening the network to the applications and central management. This is far from the original rigid definition in which it was just about to separate the control plane and data. Therefore, OpenFlow (McKeown et al., 2008) is a component of SDN that offers programming and infrastructure simplification. At this level, several models of SDN have been designed in parallel. The work generally focuses on the inherent programmability of equipment (new programmable devices...), the SDN controllers and orchestrators whose mission is to provide a network abstraction layer, network, and virtualization functions that aim to overcome the complexity of the underlying physical network making the configuration more agile. The challenge for network managers is to follow-up on this new stage to take advantage of these new potentials.

As enterprises choose to implement Software Defined Networking (SDN), one of the main concerns raised is SDN security problems. Enterprises are seeking if SDN products will guarantee that their applications, sensitive data, and infrastructure won't be exposed to any security risk related to SDN. With its introduction, new strategies for securing the SDN application layer, control layer, infrastructure layer are needed. In this chapter, we review the attack vectors affecting the control plan in case of SDN implementations and share ways to secure it. After, we propose an enhancement to our approach called "Enhanced Solid-Flow," which will rely on our previous work (Qasmaoui & Haqiq, 2018) to more secure the flow rules database leading to enhance trust in flow rules present in the controller.

In summary, we make the following contributions:

- We propose a new security mechanism based on hashing algorithms and authorization to handle flow rules insertion.

- We design and develop the mechanism that runs in three scenarios for best performance and efficiency and accuracy.
- We evaluate the performance of our solution in deferent scenarios regarding reactivity CPU usage and response time.

The remainder of this chapter is organized as follows: Section II presents SDN architecture and principals. Section III security challenges facing the SDN controller. Section IV summary the proposed solutions to dress SDN security issues. Section IV outlines the enhanced Solid flow architecture and its scenarios, and Section V discusses our implementation and performance evaluation. Finally, Section VI concludes with a discussion of our future works.

2. RELATED WORKS

In the previous chapter, we discussed many solutions presented in the literature to cover the security aspect of SDN networks. Especially the SDN controller security threats as known the SDN controller remains the brain of the network and a central decision making.

To secure the OpenFlow controller-based SDN from various threats the reliable techniques must be conducted to secure these impedimenta. Several approaches and researchers have been conducted; we summarized them as follows.

Tootoochian et al. (Tootoonchian & Ganjali, 2010) proposed a Hyperflow. A distributed event-based control plane for OpenFlow. It allows the network to be scalable and keeps the advantages of centralized network control centralization. Hyperflow application are installed on the OpenFlow controller and put network-wide views in synchronization through producing actions that affect the controller behaviour. Hyperflow constrains decision making to be localized in distinct controllers. Consequently, the response time of forwarding plane request is minimized by the control plane. It also allows the communication between self- managed OpenFlow networks, and this is a crucial character which is not offered in existing OpenFlow network implementations. This guarantees the accessibility of the controller to antagonize DoS attacks.

DoS attack is also recognized as the main threats that overhead straightforwardly the controller processing and facilitates the flooding of MAC tables. So that puts the performance of SDN in many criticizes. Dridi et al. (Dridi & Zhani, 2016) offered SDN-Guard. A pioneering method that redirects the flow of mischievous traffic and enable long timeouts aggregation of flow rules related to malicious flux to secure the SDN controller. SDN-Guard successfully minimized the

effects of DoS markedly minimized up to 32% of controller throughput and control plane bandwidth usage.

Nguyen et al. (Nguyen & Yoo, 2016) presented an extension to SDN controller. It is automatically secure controller from attacks on tracking service.

The extension relays on three factors as follows:

- Port Manager: It is responsible for determining the host which is producing the traffic. It also detains a list of hosts mapped with corresponding MAC addresses.
- Host Probing: It's main role is verifying if the host is reachable or not by generating ICMP echo request.
- Host Checker: To verify if the host can be migrated and prevent ARP poisoning.

Kuerban et al. (Kuerban et al., 2016) offered FlowSec approach to prevent DoS attack on SDN controller. FlowSec calculates the collected controller bandwidth statistics automatically. If the attack is found, the switch will be enforced to reduce traffic by using Floodlight module (Floodlight, n.d.-a), which is also responsible for collecting switch statistics. Then, Suh et al. (Suh et al., 2010) designed a Content-oriented Networking Architecture (CONA) and clarified how it works on NetFPGA-OpenFlow platform. In CONA an access router is able of determine which content gets pulled form attached, hosts. CONA service receives the hosts' content request and responds with corresponding response content. thusly, CONA thusly the accountability and countermeasure can be taken to block threats which consume network resources like DDoS attack. Since controller is a single point failure, TCP SYN FLOOD threats (Type of DDoS attack) can attempt to compromize the controller. Fichera et al. (Fichera et al., 2015) introduced OPERETTA. It is an OpenFlow based resolution to TCP SYN FLOOD threats. OPERETTA is used to allow TCP SYN packets to reach into the controller and bloacck bogus connection request. Later, a study has been conducted by Wang et al. (H. Wang et al., 2015) to data-to- control plane overload attack in the active router and announced FloodGuard. FloodGuard act in two different ways. The first approach is a proactive flow rule analyzer. It's used to conserve network policy hardening by automatically generating proactive flow rules and inserting those flow rules inside the data plane switch. The second approach is a Packet migration. It is used to deliver the flooding packets to OpenFlow controller by exploiting Round-Robin scheduling algorithm after caching them to preserve the controller performance.

Buragohain et al. (Buragohain & Medhi, 2016) offered an SDN framework baptized FlowTrApp. It allows the detection and the mitigation of low and high rate DDoS threats on the

data center. It classifieds arriving attack traffic by comparing it with reliable flow traffic rules. The mitigation happens if the user sends attack traffic repeatedly but not from the first attempt.

The controller remain the one who is responsible for executing the validation of source address, Yao et al. (Feng, T., Bi, J., Hu, H., Yao, G., & Xiao, 2012) designed a solution to source address authentication called Virtual Source Address Validation Edge (VAVE) that can identify the source address of the inbound packet. VAVE favored with many important features. One of these noteworthy characteristics is agility. VAVE agility reduces packet process overload and enhance resource usage performance. An Address Resolution Mapping has been inserted to the controller by Matias et al. (Matias et al., 2012), which can dodge unidentified ARP request by trailing MAC address.

Hong et al. (Porras et al., 2012) introduced TopoGuard a security mechanism that works on SDN controller and automatically affords a real-time detection discovery to poisoning threats on network topology. TopoGuard is able to protect a network topology by using a minimum effect on a normal operation of SDN controller.

Chung et al. (Chung et al., 2013) presented a NICE which automates an OpenFlow application check by applying a test model in network equipment in a systematic way to accelerate the discovery of state space of unchanged controller programs. Porras et al. (Porras et al., 2012) proposed FortNOX, a tool which permits an authorized rule-based and harden security policy to NOX controller (Gude et al., 2008).

Kreutz el at. (Kreutz et al., 2013) designed a trustworthy and dependable SDN controller platform called FortNOX that works on NOX controller. It can authenticate every rule and employs a strong technique to analyze these rules even if they try to put into the flow rule which can cause a conflict in NOX controller. Afterward, Wen et al. (Kreutz et al., 2013) proposed PermOF, a fine-grained permission system to apply minimum privileges in applications and harden permissions at controller API entries. They also presented a technique that provides an access list along with isolation between controller operating system and applications which put the priority on operating system rules. Then, Hu et al. (H. Hu et al., 2014) proposed FlowGuard framework to execute a proper validation and efficient resolution of firewall rule violation in dynamic OpenFlow based networks.

Overall, to mitigate these types of security threats. It is recommended to implement such security policy enforcement, supervising tools and reliable techniques to more secure an SDN

controller. On the other hand, it is valuable to adopt recovery solutions to make sure of the stability of the network.

3. THE PROPOSED MODEL

3.1 System Model

In the previous chapter, we presented the most relevant security threats facing the controller from many perspectives. Then we gave some of the solutions proposed in the literature to mitigate these threats. None of the previous solutions has treated the integrity of the controller config data-store. A malicious application or any mischievous person or system that gained access illegitimately can alter the data present within the config data-store. The main goal of the datastore is to hold the actual configuration of the SDN controller and its environment making it a suitable target, losing control over the storage section expose the SDN network-based implementation to several type of risks, from a simple traffic deviation to denial of services.

In this section, we will introduce a significant enhancement to our approach that aims to improve the controller security, especially the flow rules config datastore from false and illicit flow rules modification, insertion and deletion. The Enhanced Solid-Flow module will be implemented in one of the SDN controller and will be designed following this perspective such as the open-daylight (Ahmad et al., 2015) controller , Floodlight. Fig 18. shows SDN Floodlight controller architecture. Our module will target the storage section of the Floodlight Controller.

The Enhanced Solid flow was designed using java-programing language and was implemented in two scenarios:

- As module running within the controller
- As a module exposed via a secure rest-api.

As mentioned in the previous paragraph, flow rules are generally generated by two manners, either from SDN application operating on the application plan or by users with system administration rights. To ensure that these flow rules are generated by one of the above trusted methods, we present an enhancement of the “Solid-Flow” module, which will be implemented as a plugin, developed using OSGI framework Equinox. Our plugin will rely on several mechanisms such as SHA 256 or SHA 512 (S Gueron, 2012) cryptography hashing function to ensure the integrity of the control flow rules present within the controller data-store.

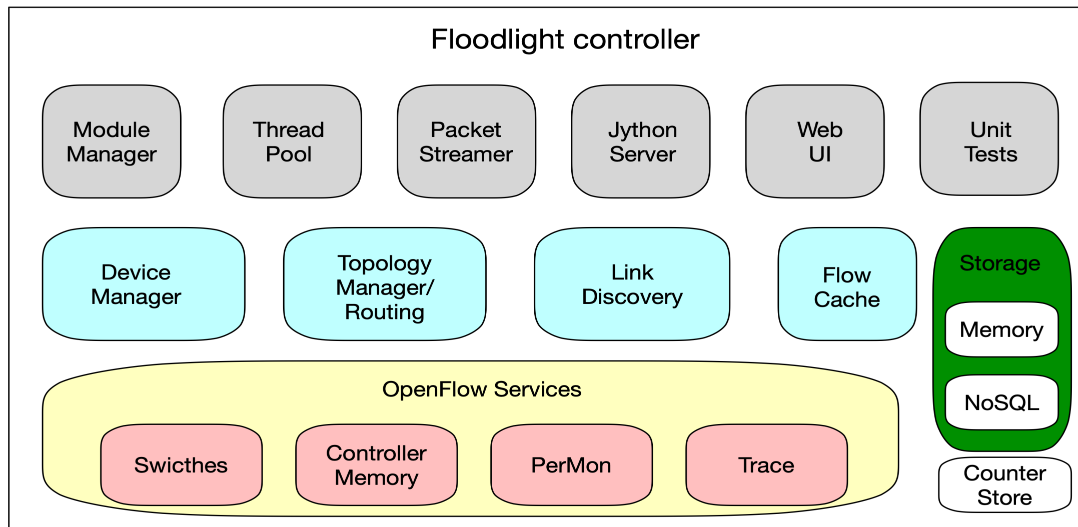


Fig. 18. Floodlight Architecture

Solid-flow is attached to the controller environment as shown in Fig. 20 and will run a periodic check after the initiation state on dynamic periodic check calculated on the fly depending on the number of attacks in the previous cycle.

Our proposed model exposes its services through a secure rest API and can run in standalone mode.

Flow rules check will be also triggered in case of newly arrived legitimate flow from any source whether is an SDN application or users with administrative privilege. This event will be triggered to ensure that during the interval between periodic checks, no flow-rule was inserted illegitimately, resulting in more reliable config-data store.

Each flow rule will trigger several processes from hash calculation passing through checking the type of flow (periodic or check) then depending on the type the Solid-flow will compare hashes and determine the eligibility of the flow rule then decide what to do with it.

Fig. 19 shows two types of flow rules (legitimate flow-rule and illegitimate flow-rule) sources insertion along the time line and the two types of check, a periodic Solid-flow-check and a check before the insertion in the data-store.

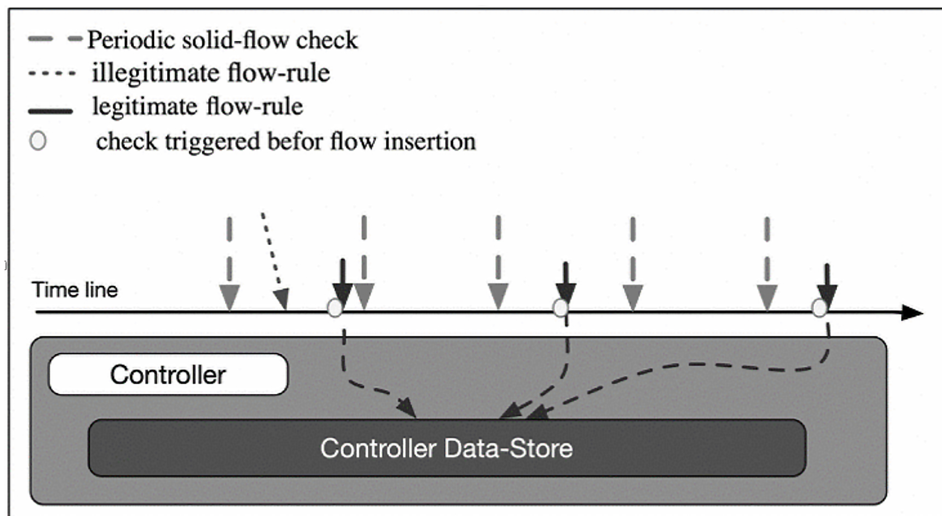


Fig. 19. Attack timeline

Each periodic check will pull a new fresh hash of the flow rules data-store, a hash calculation will be calculated for each flow rules and for the global content of the data store, then it will be compared with the previous hash's stored from earlier calculations.

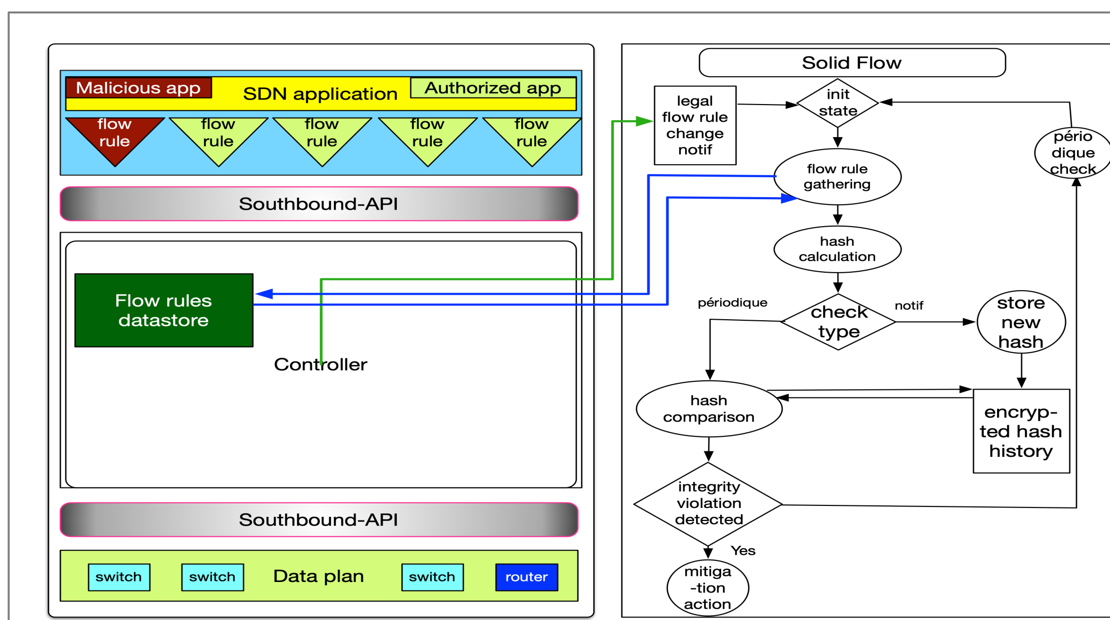


Fig. 20. Enhanced Solid-Flow architecture

When the controller triggers the Solid-flow mechanism by notification, assuming that this change is legal and made by one of the trusted methods using one of the solutions proposed earlier such as permOF (Wen et al., 2013), a new hash will be calculated before inserting the new flow rule in the config data-store. Afterward, solid-flow will confirm to the controller that no change has been made to the config data-store. Subsequently, the new flow rule could be inserted securely, and unique hashes will be calculated and encrypted for the next comparison.

Hash comparison will result either in normal config data-store state or in integrity violation. In the case of integrity violation, mitigation actions will be conducted depending on the check type 'periodic, reactive periodic or notification'.

The system will determine with high precision where the illegitimate change has been made

3.2 Solid-Flow design principles

Our module will run during the controller initialization stage. After the initiation phase, the Solid-Flow mechanism will be prompted to run in three scenarios:

Scenario 1: The periodic hash gathering.

Event 1): Initializing runtime environment and checking controller state.

Event 2): Gathering policies information from the controller flow rules data-store.

Event 3): New hash calculation.

Event 4): Hash comparison (between the latest and previous hash).

Event 5): If integrity violation detected, take mitigation action.

Event 6): Periodic check timer initiation.

Scenario 2: Event initiated by the controller in case of new flow rule arrival.

Event 1): Notification form the controller about new flow inserted by a trusted entity and wait for the Sloid-flow confirmation.

Event 2): Gathering policies information from the controller flow rules data-store.

Event 3) New hash calculation.

Event 4): Hash comparison.

Event 5): If integrity violation detected, take mitigation action. Otherwise, permit to the

controller to insert new flow-rule.

Event 6): Hash storing (new hash stored for feature comparison).

Scenario 3: Reactive periodic mode hash gathering.

Event 1): Initializing runtime environment and checking controller state.

Event 2): Gathering policies information from the controller flow rules data-store.

Event 3): New hash calculation for each rule and the global config data store.

Event 4): Hash comparison (between the latest and previous hash).

Event 5): If integrity violation detected, take mitigation action.

Event 6): Periodic check timer initiation using the proposed formulate.

The above scenarios are translated into algorithms. We propose 3 algorithms related to each

previously described scenario to improve flow rules database integrity inside the SDN controller. All the parameters used in different algorithms are presented in Table V.

Table V. ***PARAMETERS OF ALGORITHMS***

Parameters	Meaning
PT	Periodic Timer
DPT	Default Periodic Timer
RT	Reactive Timer
DHA	Data Store Hash
AHA	Actual Hash
CT(ER)	Number of wrong hash counter
CST	Controller State
CVF	Controller Verification
T_HRMAX	Maximum Threshold (time)
T_HRMIN	Minimum Threshold (time)

Among the parameters, PT, DPT, RT, represent the periodic timer, the default periodic timer, and the reactive timer respectively. DHA and AHA, describe the data store hash and the actual hash. CT(ER), CST and CVF represent the number of the wrong hash counter, the controller state, and the controller verification. T_HRMAX and T_HRMIN represent the maximum threshold and the minimum threshold, respectively. The values of some parameters are set according to the attack model or scenario setting.

For the first scenario, as shown in algorithm 1, we initiate the Periodic Timer PT to a Default Periodic Timer. The Number of wrong hash counter CT(ER) is in state 0. If the Controller State CST is initialized, then the value stored AHA is the same as Data Store Hash DHA. At the beginning of time period DPT, if the stored value Actual Hash AHA is different from that of the Data Store Hash DHA, the Number of the wrong hash counter will be 1, and Take mitigation action.

Algorithm 1: The periodic hash gathering

1: ***Let PT = DPT;***

2: ***Let*** **CT(ER) =0;**

3: ***if*** **CST is initiated then**

4: ***AHA = DHA;***

5: ***End if***

6: At the beginning of time period DPT :

7: ***If AHA == DHA then***

8: ***AHA = DHA;***

9: ***Else if AHA != DHA***

10: **CT(ER) += 1;**

11: **Take mitigation action**

12: ***End if***

13: ***end of time period DPT restart over***

For the second scenario, the event is initiated by the controller in case of a new flow rule arrival. In this case, the controller sends a verification about new flow inserted by a trusted entity and waits for the Sloid-flow confirmation. The newly calculated hash called in the algorithm. Actual Hash AHA is compared with the data stored hash DHA. If integrity violation detected, take mitigation action. Otherwise, permit the controller to insert a new flow-rule. The new hash is stored for feature comparison.

Algorithm 2: Event initiated by the controller in case of new flow rule arrival

1: ***Let PT = DPT;***

2: ***Let*** **CT(ER) =0;**

3: ***Event: event ("controller ask for verification")***

4***: Action: (If AHA == DHA then***

5: ***AHA = DHA;***

6: ***Else if AHA != DHA***

7: **CT(ER) += 1;**

8: **Take mitigation action**

9; ***End if)***

10: ***if*** **CST is initiated then**

11: ***AHA = DHA;***

12: ***End if***

13: At the beginning of time period DPT :

14: Wait-event();

15: ***If AHA == DHA then***

16: ***AHA = DHA;***

17: ***Else if AHA != DHA***

18: CT(ER) += 1;

19: Take mitigation action

20: ***End if***

21. ***end of time period DPT restart over***

For the last scenario, relative to reactive periodic mode hash gathering. We were initializing the runtime environment and checking the controller state. The New hash calculation for each rule and the global config data are calculated and compared (the latest and previous hash). If integrity violation detected, take mitigation action. and the periodic check timer initiation using the proposed formula:

DPT= $(Thrmax - Thrmin)e^{-\lambda} + Thrmin$

Algorithm 3: Reactive periodic mode hash gathering

1: Let PT = DPT;

2: Let CT(ER) =0;

3: Event: event ("controller ask for verification")

4: Action: (If AHA== DHA then

5: AHA=DHA;

6: Else if AHA!=DHA

7: CT(ER)+= 1;

8: Take mitigation action

9: End if)

10: if CST is initiated then

11: AHA=DHA

12: End if

13: At the beginning of time period Dpt :

14: Wait-event();

15: If AHA=DHA then

16: AHA=DHA;

17: Else if HA!=DHA

18: CT(ER) += 1;

19: λ = CT(ER),

20: DPT $(Thrmax - Thrmin)e^{-\lambda} + Thrmin$ **;**

21: Take mitigation action

22: End if

23: end of time period DPT restart over

4. PERFORMANCES ANALYSIS AND DISCUSSION

We have implemented the enhanced version of SOLID FLOW as a separate program that runs within the controller environment, using java, which is the programing language of all recent Open-Flow controllers.

The proposed solution was tested on a floodlight controller, which is a large-scale network simulator. SOLID-FLOW can be used with any other open-flow controller offering a rest API. The program communicates with the floodlight controller using secured calls to the REST-API based on trust-based TLS/SSL channel. In this section, we will present the use case environment variables and calculate different metrics (System reactivity, The CPU rate, response time to attacks).

The tested use case was conducted using a floodlight controller installed on Ubuntu Linux distribution. The SDN data layer was created using Mininet (R. L. S. de Oliveira et al., 2014), which is a software emulator for prototyping an extensive network on a single machine. The Implementation parameters are presented in Table VI.

Table VI. IMPLEMENTATION PARAMETERS

label	value
Emulator	Mininet
Number of nodes	3000
Number of OV switches	300
Number of flow rules	1000
Dynamic Periodic timer	Start from 2 seconds

4.1 System reactivity

In this test, we wanted to demonstrate the reactivity of the system in the reactive mode compared the normal running mode. The conducted test results in a sensitive system that detected the malicious flow rules depending on the checksum fault rate in each flow rules as they acquire and accordingly adjust the periodic timer to match the intensity of the attacks in the previous cycle.

The periodic timer is calculated following the next equation:

TP= $(Thrmax - Thrmin)e^{-\lambda} + Thrmin$

Where :

- TP: The periodic timer.
- Thrmin and Thrmax are the minima and maximum threshold of the time interval with which the Solid Flow performs a periodic check.
- λ: the number of errors in the previous cycle

We have set up a limitation to our mechanism to avoid exhaustion of the CPU usage in case of the significant flow of attacks, assuming the system administrator is present to take appropriate action in time, for that we will have two main scenarios:

1. In case of large number of attacks targeting the controller, the periodic verification time will drop dramatically according to the previously cited equation. The performance of the system may deteriorate if this time becomes very low resulting in large load on the CPU hence a high CPU consumption. The Thrmin will allow setting a minimum threshold not to exceed by our system making it more stable.

2. The case where the system knows a much-reduced number of attacks or outright no attacks, Thrmax makes it possible to set a maximum threshold of the verification time, thus guaranteeing a better rate TPR (True Positive Rate). Indeed, with a very long time of verification, the system may not be aware of the attacks that would occur between the verification sequences as mentioned before in Fig. 20.

Fig. 21. demonstrates the reactivity of the system to the number of attacks. As we can observe the more attacks acquire the shorter time become and vice-versa. This reactiveness allows our module to be more agile in response to the amount of acquiring attacks.

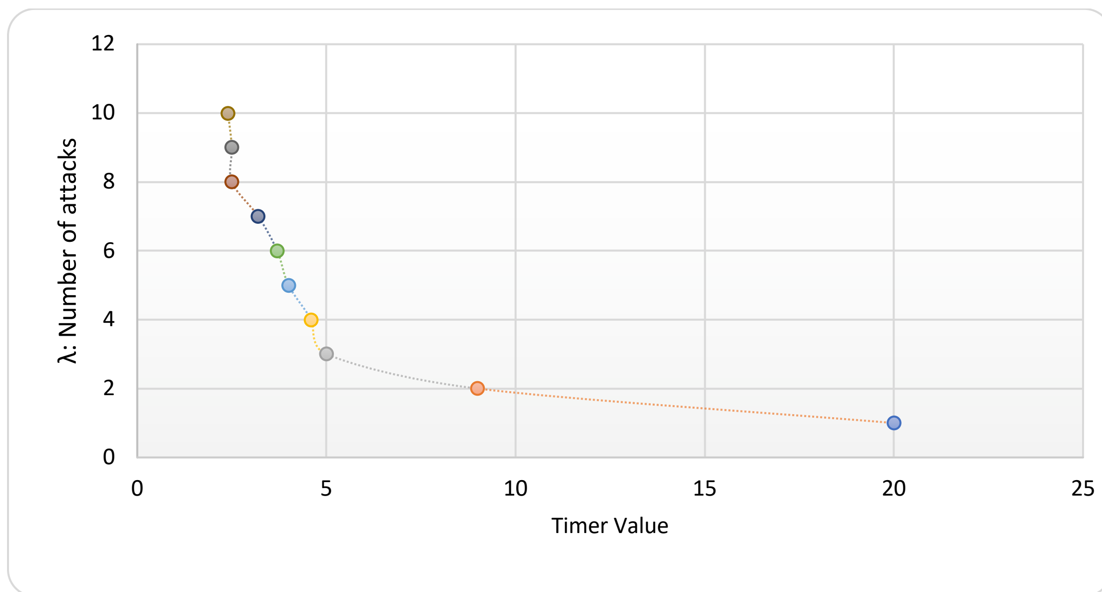


Fig. 21. The system reactivity under attacks

4.2 The CPU rate used at the SDN controller

Running the Solid-Flow program in parallels with SDN controller does not require any high-consumption of CPU while running it in reactive mode enables us to save 40% of CPU usage. The CPU reduction is due the low rate of check in case of low rate of attacks. Using our module in both mode (within the controller environment and as Res-Api module) rendered almost the same results.

The number of flow rules remain significant to our tests and increase the CPU rate as the sloid flow need to deal with large amount of data.

In Fig. 22, we note that the CPU rate used at the SDN controller level changes linearly with the increase in the number of nodes in the SDN network and remains remarkably low. It should also be noted that improving the time to check data integrity in reactive mode significantly

enhances the CPU rate at the controller level, about 40%, allowing better response to potential attacks.

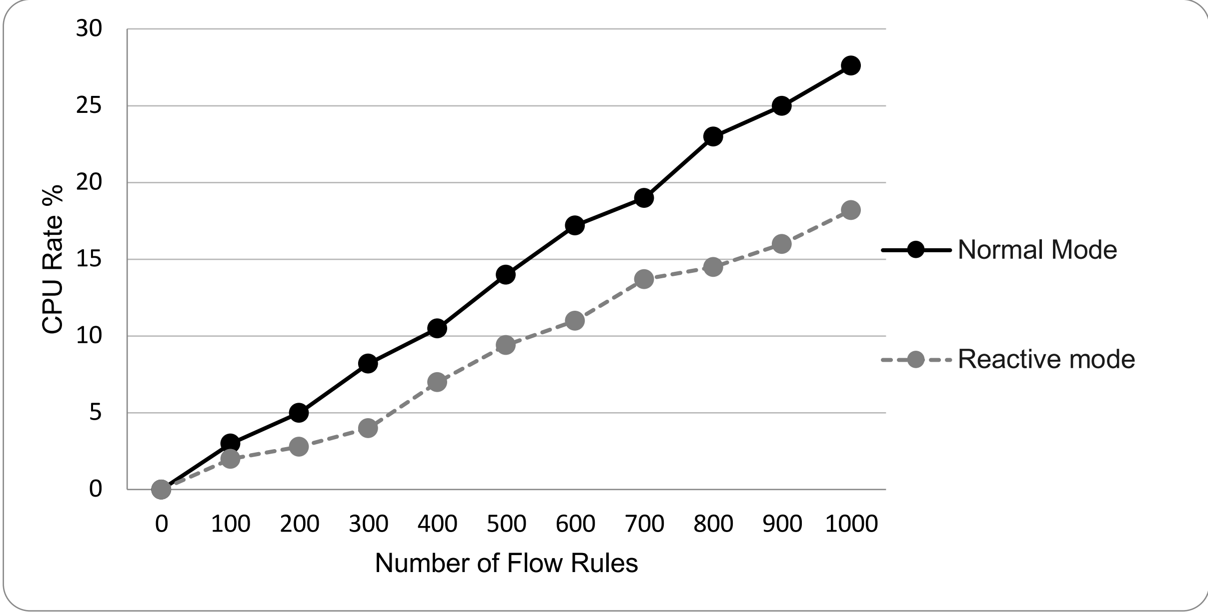


Fig. 22. CPU usage

4.3 SDN Controller response time to attacks

By minimizing requests for data checks using the reactive mode, the SDN controller does not need to be entirely dedicated to processing and responding to periodic checks, which will naturally improve the time of its response to the actual attacks. Fig. 23. clearly shows that the SOLID-FLOW reactive mode effectively makes it possible to have a minimization of the order of 30% of the response time compared to the normal mode.

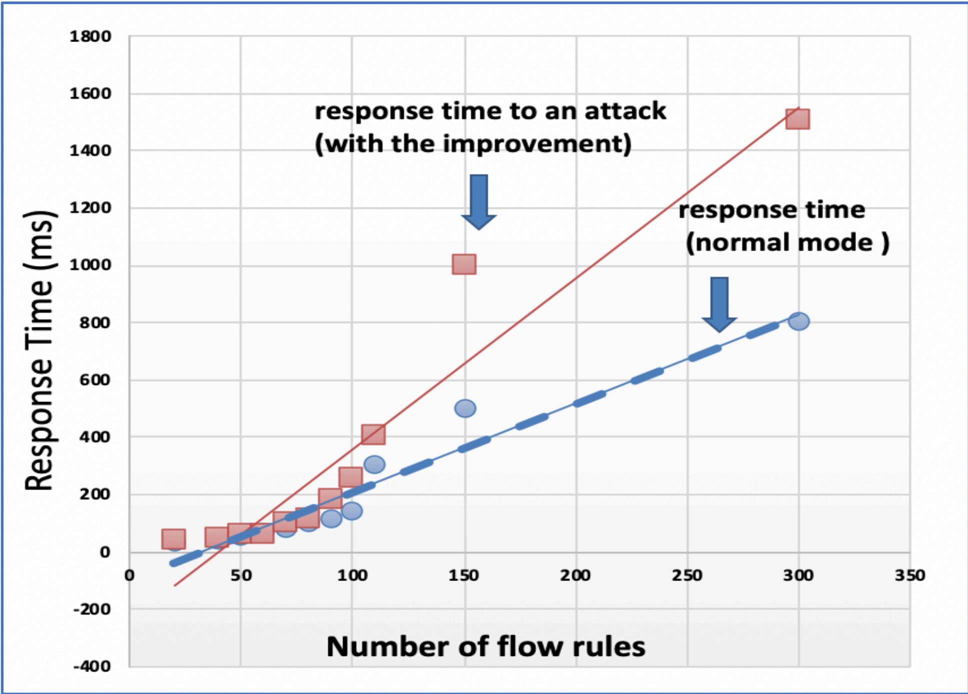


Fig. 23. SDN controller response time to attacks

5. CONCLUSION

Currently, SDN technologies attract a lot of interest in the networking industry. However, their deployment exposes the network to unknown risks and still poorly documented.

The SDN networks differ from conventional paradigms, which introduce new security challenges alongside essential practice (although not explicit in standards) such as the use of a dedicated management network, the establishment of an authentication solution, and integrity as TLS, or equipment redundancy in charge of routing. In this chapter, we gave state of the art on the various security threats targeting the control layer and challenges in SDN, and the multiple countermeasures proposed. We also introduced a Solid-Flow module for ensuring the integrity of the data-store.

We have implemented the Solid-Flow module and tested its behavior regarding deferent aspects; the test result was encouraging, as shown in the previous section.

The upcoming chapter will discuss our next contribution to software defined network.

Chapter 4: Secure Software Defined Networks Controller Storage using Intel Software Guard Extension

1. Introduction

In recent years, the network research community has experienced a period of intense activity that has led to the emergence of different architectures or paradigms such as the SDN. The centralization (logical or physical) of the control plan, had to bring the expected flexibility to the network applications and allow to respond to many concrete use cases.

Software-Defined Network is a new paradigm of network architecture that aims to design a data plane that is fully programmable and separated from the control plane (Ismail, 2019). The Control Plane manages decisions about how and where to transmit network traffic through system configuration, management and exchange of routing table information. The Data/Forwarding Plane manages the actual transmission of packets to the destination network according to the logic of the control plan. Behind this separation, there are three main objectives:

- The separation of network intelligence (control plan) from equipment (data plan).
- The provision of a logically centralized view of the global physical network.
- Providing an abstraction of programmable network equipment using Interfaces of application programming (API).

Among the innovations of this new paradigm is the programmability of network equipment and applications. New network applications can be transparently programmed and deployed using standard APIs. However, its implementation in the data plane remains one of the greatest challenges for research.

Protecting sensitive data from been altered or access gained by any authorized manner is present since the beginning of the programming time, a challenge that many have taken the race to solve it (Safaa MAHRACH and Abdelkrim HAQIQ, 2020). SDN technology has been introduced to solve the complexity of configuring network hardware. SDN enabled networks relay on a central decision making called the SDN controller to handle request coming from

network devices such as switches and routers (Dargahi et al., 2017; J. Son & Buyya, 2018). Those requests are transported via API's using open flow protocol (McKeown et al., 2008) and optionally secured using TLS.

The SDN controllers represent the most delicate part of the SDN architecture as it consists of the brain of the network, making it vulnerable to all sort of attacks (Bhushan & Gupta, 2018; D'Cruze et al., 2018). Security issues may vary depending on the level of interest targeted by a malicious person; it goes from Denial of service to traffic redirection and flow rules modification (QASMAOUI, Y., & HAQIQ, 2020). The SDN controller software is run on vast untrusted platforms, including operating systems, hypervisors, firmware, and hardware. This large machine base is growing complex and difficult to verify. For e.g., an OS such as Linux has 17 million line of code, however 662 vulnerabilities related to CVE have been recorded in 2019, such as memory corruption, transverse directory, unauthorized code execution. Execution of normal and security-critical applications running on shared resources controlled by untrusted computing machines raises security threats. Running the SDN programs in such environments represent a considerable threat to its normal operations.

To solve the issue, one of the solutions is Trusted execution environments (TEE). TEE guarantee security by relying on less hardware and software computing base. Hardware is commonly considered to be a stable base since the cost and sophistication of hardware attacks usually are high. This has led to the development of a secure running environment by industrial hardware companies for a safety-critical application that maintains little reliance or less dependency upon the operating system and hypervisor. Up to today, we found two main technology which are ARM Trust Zone Technology, Intel Software Guard Extensions (SGX) (Medina et al., 2019)(Costan, V., & Devadas, 2016).

The objective of this work is to propose a secure architecture by programming new modules and adding security functions at the control plan storage based on Intel SGX. Then, evaluate the impact of SDN architectures at the performance level.

The rest of the chapter is organized as follow. The next section will include a background and related works followed by the proposed model to secure SDN controller storage using Intel SGX, then we present the results of the implementation with discussion. Finally, we conclude this work with a conclusion and perspectives for future work.

2. BACKGROUND

Trusted Execution Environment (TEE) is a tamper-resistant computing ecosystem that works on a separate kernel. It guarantees the validity of the executed programs, the security of the runtime components (e.g. memory, CPU registers, and critical Input / Outputs) and the secrecy of the executed code, data and runtime states are maintained in non-volatile memory (Lefebvre et al., 2018). In addition, the remote certificate shall be given to show its trustworthiness to third parties. The contents of TEE are not static; they can be changed safely. TEE condemns all software-related threats as well as hardware threats against the main memory of the operating system. Attacks leveraging backdoor authentication bugs are futile. Fig. 24 illustrates the difference between a Trusted execution environment and an ordinary OS.

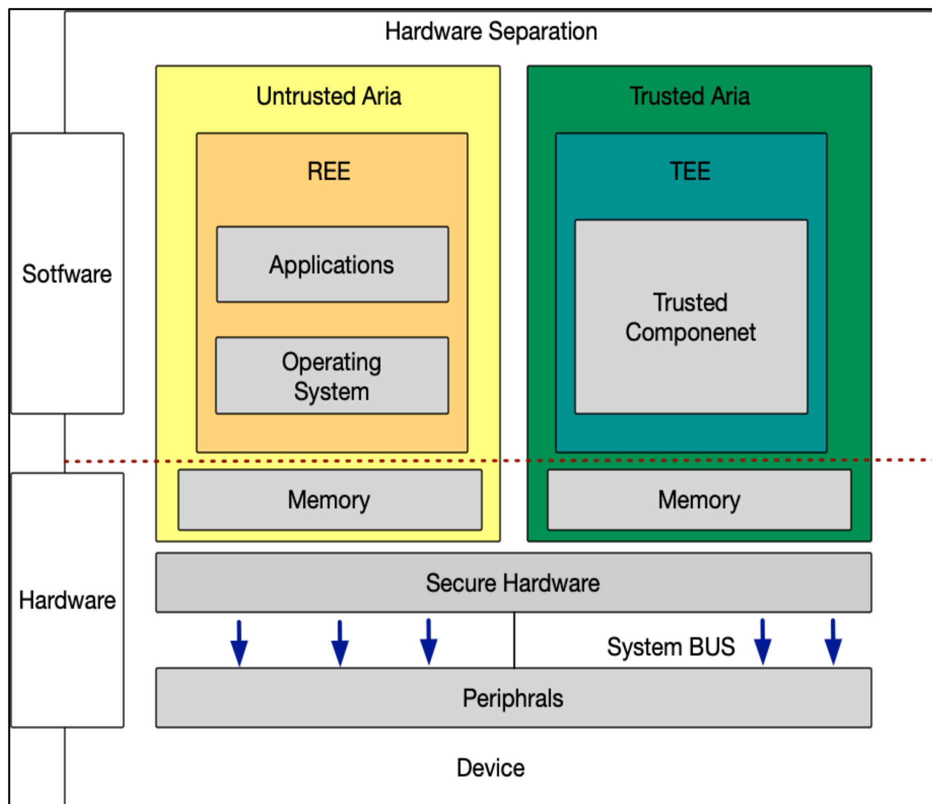


Fig. 24. Trusted execution environment

The most common TEE environments are Intel SGX and ARM TrustZone (Winter, 2008). Both Intel SGX and ARM TrustZone are hardware TEE environments, but the process behind building a trusted environment with trusted code is distinct. Intel SGX provides a trusted environment for trusted programs that run on top of current untrusted device software. Whereas, ARM TrustZone is building a new, trusted ecosystem for trustworthy applications that operate on trustworthy device software and hardware that are only accessible to the trusted Configuration.

In this chapter we focused on intel SGX technology to deploy our secure SDN controller. The Choice of using Intel SGX was taken depending on the much benefit that supersede ARM TrustZone, benefits such as documentation, maturity and the availability of hardware enabled machines. The majority of researchers uses Intel SGX to deploy a trusted execution environment.

3. INTEL SGX ARCHITECTURE

Intel has providing the general-purpose hardware-assisted TEE stated as Intel SGX. Intel SGX is an extension of x86 architecture with innovative new component of security-related instructions. Security-critical applications use these instructions to create a trusted hardware-assisted environment known as an enclave (Costan, V., & Devadas, 2016). The Intel SGX enclave maintains confidentiality by using memory access controls with hardware data structure and integrity protected by encrypting data and code while going beyond the CPU package. Intel SGX is indeed a centralized security model and the CPU package is known to be a trusted computing base TCB.

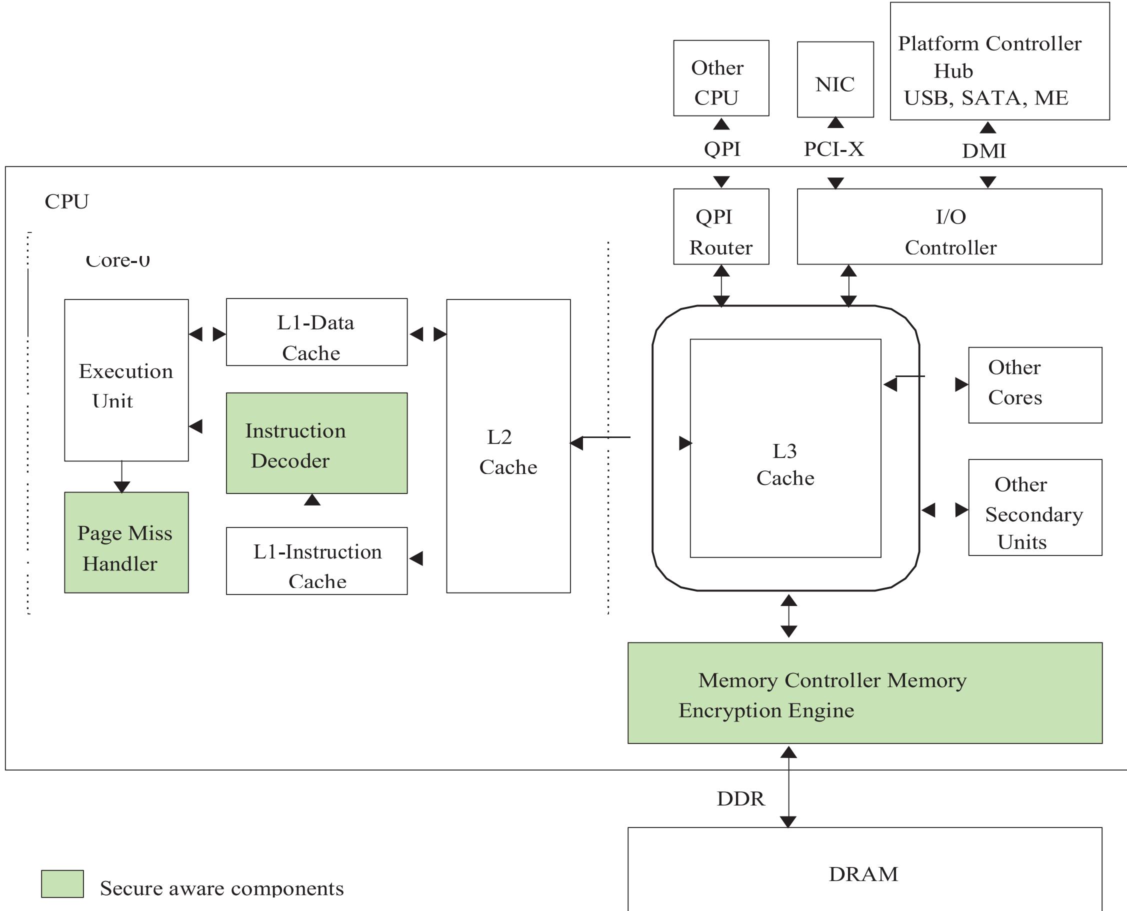
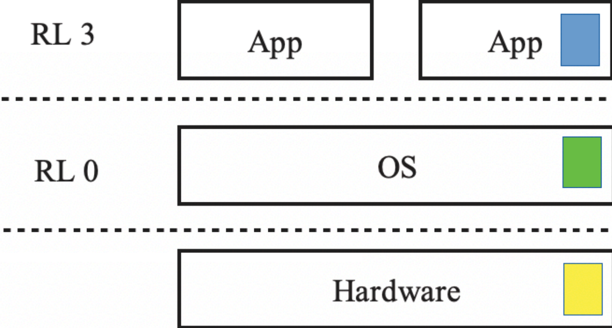


Fig. 25. Intel SGX implementation

As shown in Fig. 25, the SGX processor does not have orthogonal privileged secure application levels as in the TrustZone. The program runs on the same untrusted operating system, but protection is accomplished by matching with meta-data handled by hardware that cannot be read or written by the OS. In order to benefit from SGX, the purpose behind achieving protection over traditional software stacks is to reduce the effort needed to change application code. History shows this is a good option, since its ability to preserve backward compatibility is a major factor in the ongoing supremacy of the Intel architecture.



Enclave – code that require trusted execution environment

Code that manage page table for enclave

Data structures manged by hardware for security checks

RL Ring level
App Application
OS Operating System

Fig. 26. Intel SGX software stack

Intel SGX conceptually provides an application known as processor-reserved memory with a new form of address space.

PRM as shown in Fig. 27, have special security properties. High-privileged software such as operating system and hypervisor does not read or write PRM address space, although it contains low-privileged level code/data. Conversely, there is no high privilege for the code/data in PRM than the system software since the code in it does not access address space for the system software. In addition, the memory mapping decision taken by OS is tested and overridden by the processor if the processor considers an inconsistency with meta-data handled by the processor without OS support, which is called the enclave page cache map

(EPCM). EPCM includes the virtual address required to be used to access the enclave page and access permissions, i.e. to read, write and execute each enclave page.

When the processor exits the CPU chip, it encrypts the PRM pages, which ensures protection against bus tapping attacks. The processor also measures the signature to ensure the integrity of the page before loading it into the cache. This catches OS malicious writing, reading or substituting with other pages, but cannot restrict the OS type from performing such malicious actions.

SGX modifies the CPU components to reach trustworthy storage only in device resources, as shown in Fig. 27. This is because the processor transfers all memory and I/O access requests, so CPU checks are sufficient for protection to be achieved. In SGX, three components are the key changes: instruction decoder, page miss handler, and memory controller. Second, 18 new instructions are implemented in the instruction decoder, which includes 5 user instructions used by the enclave initialization and construction application and 13 supervisor instructions used by the OS to handle the enclave page table. Microcodes related to memory access controls that are caused by page miss handler are also added. Second, when a logical processor is in enclave mode, or when the physical address generated by the page walker machine matches the PRM range, PMH hardware is changed to gain an ability to activate the microcode assist for all address translations. Finally, the memory controller, which specifies the size of the PRM, adds a new register called Processor Reserved Memory Range Registers (PRMRR). The memory controller is also integrated with a memory encryption engine that uses non-standard cryptographic primitives consisting of a slightly changed AES operating mode and a Carter WaymanMAC construction.

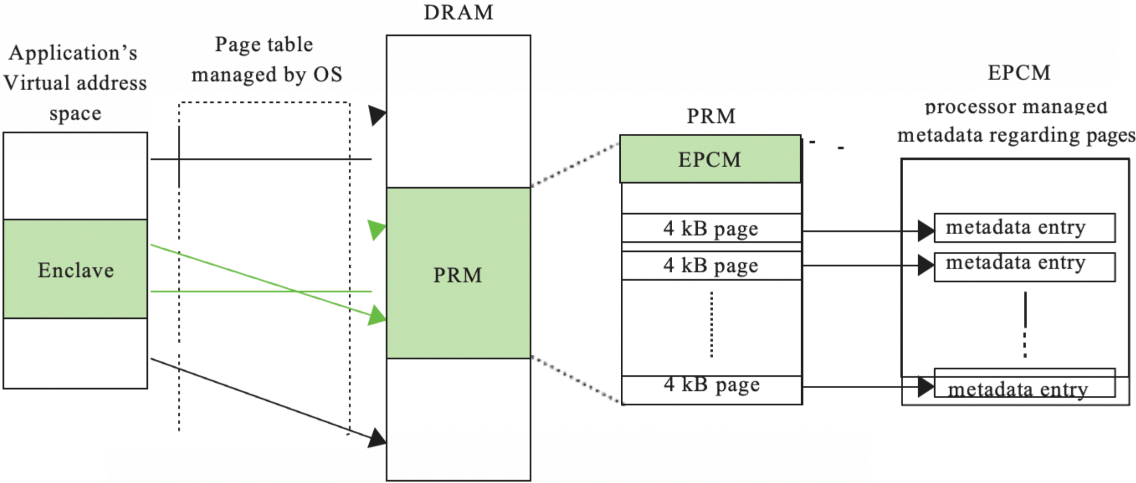


Fig. 27. Intel SGX trustworthy storage process

Hardware-based memory encryption is supported by Intel Software Guard Extensions (Intel SGX) for isolating portions of code and application-specific data in memory. Intel® SGX allows user-level code to delegate private memory regions (called enclaves) designed to be secured against processes operating at higher privilege levels. Such a granular level of control and security is only given by the Intel SGX solution.

Intel SGX, as stated in related works, was used to secure flow tables within Open-Vswitches. The architecture of a typical intel SGX enabled environment is demonstrated by Fig. 28.

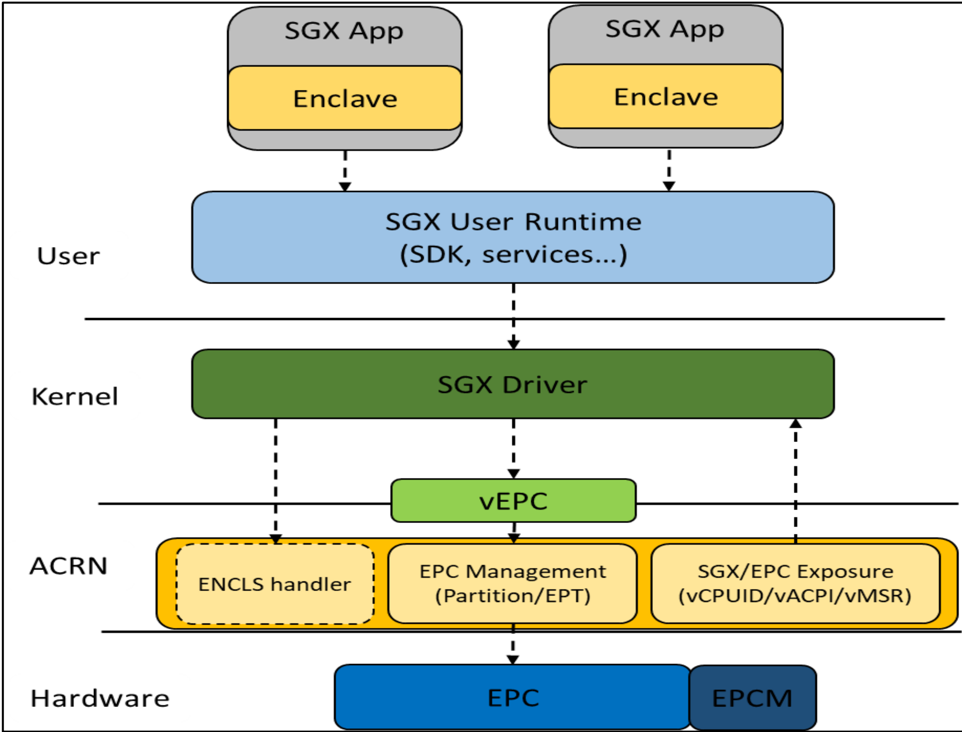


Fig. 28. Intel SGX architecture

SGX is built to be reliable; this is done in a variety of ways, including robust enclave delivery, sealing and attestation. Intel summarizes SGX's protections (Baumann et al., 2015; Kim et al., 2015) as follows;

- Memory is secured against observation and modification from outside the enclave, using an in-die Memory Encryption Engine (MEE) (Shay Gueron, 2016), with a secret that rotates on every boot. This protection notably works against host hypervisors, other enclaves, and anything running in supervisor mode.
- Enclaves will attest or confirm their identities to a competitor with the aid of a permanent hardware identification key for asymmetric encryption.

- Computer calls are designed to schedule and pass power in and out of the enclave. Arguments are safely mapped according to the concept of a static enclave.
- SGX does not protect itself against reverse engineering or side-channel attacks: to counteract this is the responsibility of the client.

4. RELATED WORKS

There are only a few works in the literature that discuss SDN security using SGX. Intel Software Guard Extensions (SGX) has provided the general purpose of the hardware-assisted TEE referred to as Intel SGX. Intel SGX is an expansion of the x86 architecture with a new range of security-related instructions (Intel®, 2014). These instructions are used by security-critical programs to create a hardware-assisted trust environment referred to as an enclave (Graydon et al., 2006). Intel SGX enclave maintains secrecy through hardware-maintained data layout and honesty tests by encrypting data and code when it is outside the CPU package (Simon P Johnson, Uday R Savagaonkar, Vincent R Scarlata & McKeen, 2015). Intel SGX is a centralized security architecture, and the trustworthy TCB computing foundation is known to be a CPU package.

TruSDN is a mechanism for bootstrapping confidence in the technology of SDN (Paladi & Gehrmann, 2017). Supports the safe supply of switches in SGX enclaves, a protected communication channel between switches and SDN controllers, and secure communication between endpoints.

Trusted Click (Coughlin et al., 2017) investigates the viability of network processing in SGX enclaves. Although none of the above methods discusses the credibility and anonymity of OpenFlow flow tables, they can be complemented by OFTinSGX to accomplish this. SCONE allows operators to protect the secrecy and integrity of computing in containers against host root access adversaries (Arnautov et al., 2016). An alternative approach to securing virtual network functions running in containers, which prevents the unnecessary expansion of the trusted computing foundation, is proposed in (Girtler & Paladi, 2017). Event Controller Eviction mitigates DoS attacks and OpenFlow Application overflow (Ying Qian et al., 2016). This framework uses two different frameworks – the learning module and the flow control module – while the case handler system prevents overload and DoS attacks, the OpenFlow flow tables have no security guarantees. OFTinSGX maintains the integrity and confidentiality of OpenFlow tables and the reasoning for forwarding and disposal procedures.

TLSonSGX guarantees that OvS authorities retain communication with SDN controllers and the cryptographic material they use (Paladi et al., 2018). This methodology can be paired with OFTinSGX to provide broader security assurances for OpenFlow switches. Fig. 29 shows the TLSonSGX system design.

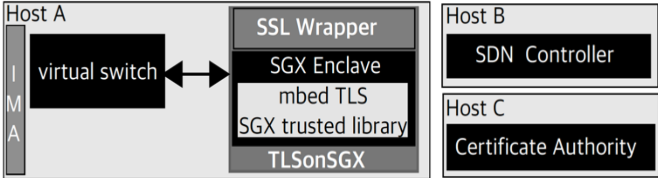


Fig. 29. TLSonSGX system design

In recent works OFTinSGX has been proposed by (Medina et al., 2019), which has four components: SGX OpenFlow table, SGX rule structure, SGX Eviction component, and SGX tables dpif, which helps OvS to delegate its OpenFlow tables and forward logic to enclave memory.

The important limitation of this work is that the abstraction of only the contents of the OpenFlow flow tables does not address all security concerns, as the classifier only includes references to the classification rules. The procedure used to control the OpenFlow flow tables can cover both the contents of the tables and the full description of the rules assigned to the untrusted memory.

Fig. 30 illustrates the OFTinSGX Architecture.

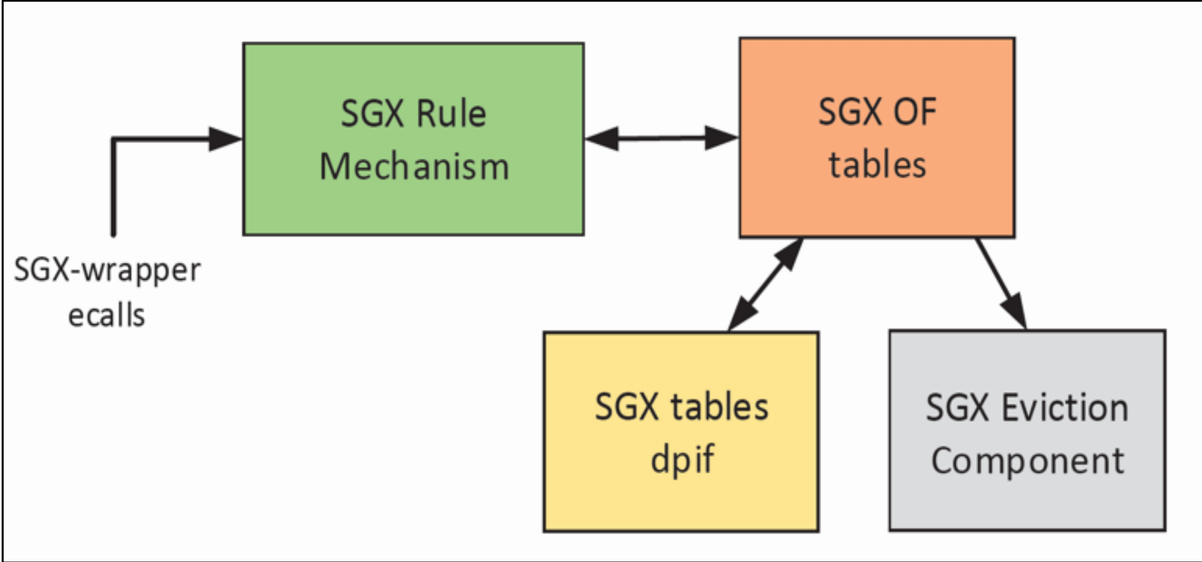


Fig. 30. OFTinSGX architecture

5. THE PROPOSED SGX SDN CONTROLLER

5.1 SDN storage module overview

This section presents the design and the architecture of the proposed model to secure SDN controller storage using Intel SGX

Generally, the SDN controller is relying on a storage module that handles all sensitive data; controller data are mostly configuration files such as flow rules. In our case, we use the floodlight SDN controller to implement our approach. Fig. 31 illustrates the Floodlight architecture. The proposed model consists of rewriting the storage module code taking into consideration the Intel SGX technology.

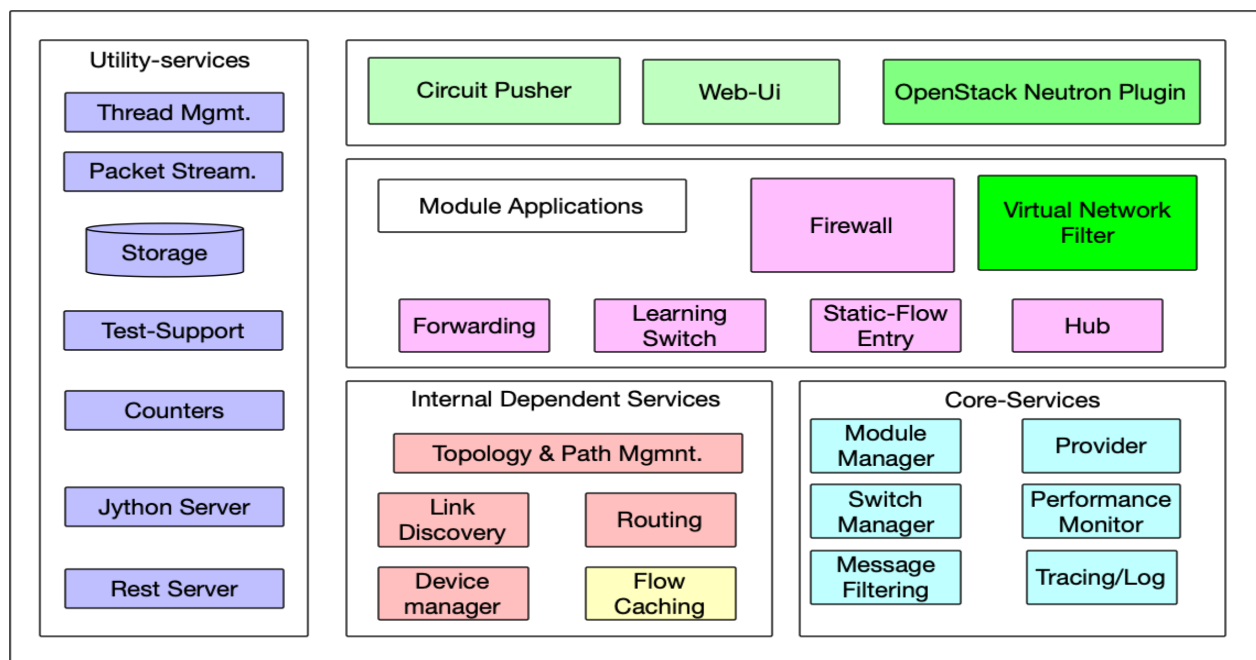


Fig. 31. Floodlight architecture

The storage module consists of four main modules: link discovery module(responsible for discovering and maintaining the status of links in the OpenFlow network), Device manager module (tracks devices as they move around a network and defines the destination device for a new flow), Static flow entry pusher (allows a user to manually insert flows and groups into an OpenFlow network) module and QoS module(give a user a way to simply push QoS state to switches that support these features) interacting with the primary storage module which is mainly a NoSQL database residing in the random-access memory. Fig. 32 shows the interaction between the four-module and the storage module.

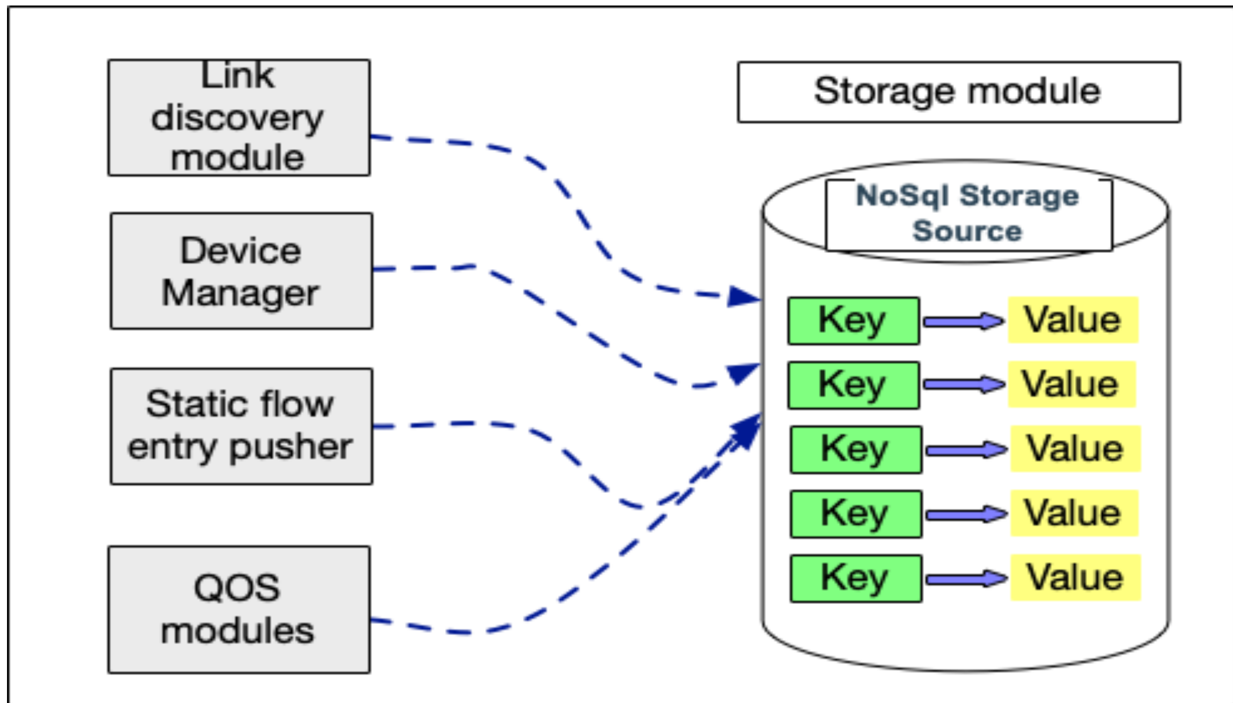


Fig. 32. The proposed storage module

A compromised host can present a huge issue to the NoSQL database making it vulnerable to several attacks such memory dump or memory exhaustion attacks. Our method consists of hardening all the space used by the NoSQL database by implementing the Intel SGX technology to prevent any damage to the RAM area used by the SDN controller.

5.2 SDN enabled SGX architecture

SDN enabled SGX model operates as an intermediate system. Specifically, it executes a process daemon that intercepts all the call made to the ordinary storage module by the controller application. These calls are translated into corresponding functions of the SDN enabled SGX model enclave. For instance, when a new open flow rules need to be inserted, the call is made via our interface and call the corresponding function inside the enclave via a JNI (Java Native Interface). The code residing in the enclave return the right value depending on the result of the function via the interface. In this design, all the data structures related to the file system are continuously kept in the Enclave Page Cache (EPC), which is a subset of DRAM that cannot be directly accessed by other software, including system software and SMM code. The CPU's included memory controllers also reject DMA transmissions targeting the EPC, thus protecting it from access by other peripherals.

The NoSQL database files are maintained by the enclave and protected by the encryption mechanisms of the SGX technology, making a dump of the memory useless as its encrypted and cannot be read.

Data is decrypted by the SGX unsealing mechanism: this process occurs while entering the enclave and is thus secured by the CPU borders. Consequently, a dump of the EPC (enclave page cache) would be captured and blocked by the SGX protection mechanisms. Fig. 33 shows the proposed SDN enabled SGX model.

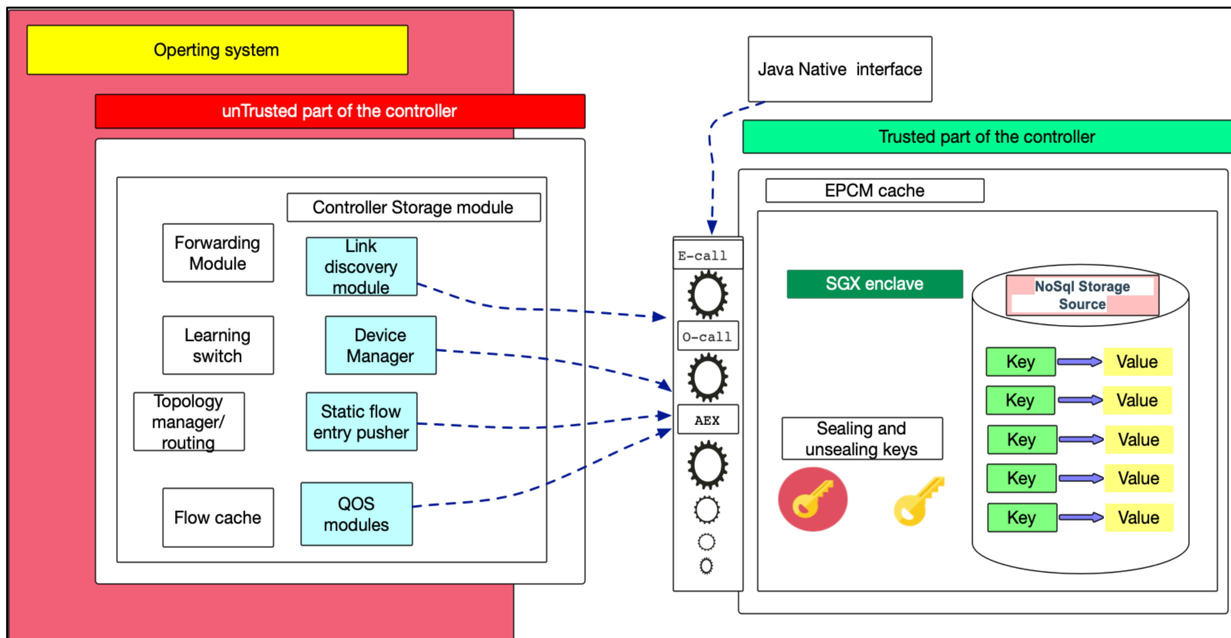


Fig. 30. The proposed SDN enabled SGX model

6. Implementation and Evaluation

6.1 Implementation setup

We implemented the SDN controller enabled SGX on a Linux platform holing an SDN controller named floodlight developed using JAVA programming language. Since intel SGX SDK support only C/C++ language, we used the Java Native Interface (JNI) to make the right call to C functions to handle intel SGX operations. The next scenario demonstrates the different phases of the implementation, this scenario is almost generic to any Intel SGX enabled case:

- *Step 1 : An enclave is created by the untrusted part of the application.*
- *Step 2 : The enclave must be initialized via a launch token, fetched and provided by Intel's launch enclave.*

- *Step 3 : Access to the LE and other architectural enclaves, e.g., the quoting enclave (QE) and the provisioning enclave (PE), is provided by the Intel application enclave service manager (AESM). SGX libraries provide an abstraction layer for communicating with the AESM.*
- *Step 4 : Execution of a trusted function which executes an ECALL.*
- *Step 5 : The ECALL goes through the SGX call interface to bring the executing thread inside the enclave.*
- *Step 6 : An OCALL is executed once the execution in the trusted environment completes.*
- *Step 7 : Finally giving the control back to the caller.*

The program run on HP Proliant gen10 server with the following Configuration:

- Intel XEON E-2174G with intel SGX support.
- 32GB of RAM

To generate traffic we use mininet simulator (R. L. S. de Oliveira et al., 2014), mininet was setup and configured on 5 different computers to be able to generate 2000 request by each one.

The written code mainly focuses on the Storage Source Service, which is the main interface of the storage module of floodlight controller, also some of the dependencies were taken into accounts such as IDebugCounterService and IRestApiService.

Table VII. shows a summary of the code changes made to the storage module inside the floodlight controller. The most significant part of the added code consists of new (JAVA NATIVE INTERFACE) JNI interface and make files, floodlight storage also has been modified to accept the call from OCALL I/O peripherals.

Table VII. ***Summary of the code changes***

Lines of code of SGX enabled SDN controller		
component	***Lines of original code***	***% changed***
Floodlight storage module	3688	9.05% (350 lines)
JNI interface	1203	100% (1203 line)
Other(e.g. make files)	200	100% (200 lines)

6.2 Performance Analysis

Our evaluation is based on the scenario that involves a number of nodes making calls to the SDN controller. In this case, the network devices receive various request to dispatch packets from a script that run on several separated machines. Decision making is sent from the data layer represented by the network devices. Next, the controller responds with the corresponding flow rule. Rules present inside the Open-vswitches are deleted, so the network devices are forced to make calls to the controller. The script sends over 10000 requests to several separated switches. The generated traffic sent to the controller allows us to take statistics and compare them to the normal scenario.

A normal scenario consists of the same use case but with non-modified controller. Table VIII, shows the collected measures and a relative Overhead. Overhead is between 7% and 10% in the first run. The overhead is calculated as shown in the following equation:

$$OH = (E[SGX] - E[normal])/E[normal] * 100$$

The increase of the overhead is caused by the use of SGX enclaves. For each ECALL to the storage module to gather flow rules, the CPU switch to the enclave mode, resulting to an increase of execution time. Overhead increases linearly with the number of e-call invocations. The state of the controller gets stabilized after a while since the flow rules are cached inside the open flow tables within the open vswitches.

Table VIII. ***Collected measures and overhead***

Controller response time			
Number of requests to the SDN controller	***Normal scenario***	***Intel SGX enabled SDN controller***	***Overhead***
10000 (first run)	3300 us	3630	10 %
10000 (second run)	2170 us	2320	7 %
10000 (third run no flow flashing)	300 us	313 us	4.33 %

Table II. shows the Overhead interval between two test cases. The same parameters are kept during the two tests. The deference between the overhead in both runs is due to CPU consumption by the OS itself and other floodlight operations.

The third run on the scenario was conducted in a special case where the open-vswitches are not flushed so flow rules are kept inside the switches flow tables, open flow switches call the SDN controller only if there is no entry matching the upcoming packets. There was a slit increase in time execution in this case and that was because of normal operations executed by the controller itself.

7. CONCLUSION

The concept of SDNs or Software Defined Networks is an architecture that facilitates network management and control, and allows rapid introduction of network services through programming and separation of the control plane from the data plane. With this new architecture, administrators can manage the network in a unified way from the control plane, and can introduce or eliminate any service through the application plane without changing the physical infrastructure. New network applications can be transparently programmed and deployed using standard APIs. Most modern SDN controllers can run on any OS However, its implementation in the data domain remains one of the biggest challenges for storage security at the control plane level.

In this work, we proposed using Intel SGX to provide additional security to the general intent of the Execution Environment of applications.

In order to evaluate the performance impact of our SDN enabled Intel SGX architecture we implemented our SDN controller model enabled SGX on a Linux platform holing an SDN controller named floodlight. The results of our model implementation show the efficiency of our model without any major cost in term of performance.

CONCLUSION AND FUTURE WORKS

SDN can promise many benefits to improve the way network are today managed. Due to the entity of a centralized element in the network, the controller, SDN offers extended control over the network, a global view and much easier management. Hosts linked to network can be directly recognized and managed. Network traffic can be classified by any parameter and routed precisely through any desired route, and new features can be implemented on the go. Ultimately, the network can be shaped according to the demand and provide an efficient way to enhance speed and ease of management. One could call this naively, the ultimate network power.

However, as the proverb goes" with great power, comes great responsibility...", replicates precisely the case of SDN. As demonstrated in Sections 1, new vulnerabilities arise, which need meticulousness attention. For example, attacks targeting the SDN controller, is a serious threat demanding urgent solutions. Compromising a controller can be achieved as easily as by slipping a malicious SDN application into the controller, SDN application can produce flow rules, once installed the malicious app can easily compromise the entire network, it can redirect traffic, block ports or even make the network hosts unreachable. This is a known drawback when the network converts from distributed to a centralized network.

The existing SDN security implementation presented in Section 2, proves that the researcher's community has Offred most interest to the demand for secure SDN solutions and specifically for a secure controller and hence provided with a framework or a system checking solutions. Solutions go from securing application plan, control plan to the ata plan. We worked among the researcher's community to contribute to more secure the SDN controller environment and its components. In chapter 4 we presented the enhanced solid flow which consists of a mechanism that is associated with an SDN controller (locally installed or acting as a standalone module) and act like a guarding to what's happening inside the most delicate part of the controller which is the flow rules datat-store. Flow rules are watched for change (edition, deletion, insertion) and any non-authorized changes are reported to the administrator. The Enhanced solid flow was designed to react depending on the number of attacks acquiring hence a timer is automatically calculated, more attacks mean shorter timer and less means less

checks. The solution offered promising results regarding the detection rate and CPU consumption and quick response to attacks.

The second contribution to the SDN security was focused on the same layer which is the SDN controller, as known the SDN controller remains a program running on a guest OS (e.g Linux distribution) on a conventional hardware. This situation puts the controller in a lot of stress as any risk targeting the OS or the hardware is an eminent threat to the controller so our solution was creating the most secure environment for the controller by providing a trusted execution environment to the controller, our work was achieved by re-writing the SDN controller to support hardware enable security using Intel-SGX as a base of our work. This, the Floodlight controller was partially rewritten by enabling Intel-SGX SDK to provide a trusted execution environment, again we targeted the Storage module by creating enclaves. Hence, everything running on the enclave is secure and encrypted. Our results demonstrated that even if the code source get longer, the execution of the controller did not disturb its functions and response time was mostly increased lightly.

Our future work will remain in the same direction of building a more reliable and secure SDN environment, securing part of the SDN controller is a promising start, so we intend to build a fully Intel-SGX controller allowing to secure the entire program and eventually look for other opportunities such as using ARM trust zone instead of Intel-SGX. We also intend to continue our work on the Enhanced Solid-Flow mechanism by adopting mitigation actions, which certainly complete the work regarding the need for actions upon attacks detections.

In summary, we will continue working on the following ideas:

- Extend the use of Intel SGX on a full source code of an SDN controller.
- Use different TEE such as ARM Trust Zone instead of Intel SGX and compare both scenarios.
- Enlarge our work on the ENHANCED SOLID-FLOW to be able to take mitigation actions.

THESIS PUBLICATIONS

- QASMAOUI, Y., & HAQIQ, A. (2020). Enhanced Solid-Flow: An Enhanced Flow Rules Security Mechanism for SDN. *IAENG International Journal of Computer Science*, *47*(3).2020. **(Scopus)**
- Qasmaoui Y., Maleh Y., and Haqiq A. "Secure Software Defined Networks Controller Storage using Intel Software Guard Extensions", International Journal of Advanced Computer Science and Applications (IJACSA), 11(10), 2020. http://dx.doi.org/10.14569/IJACSA.2020.0111060. **(Scopus)**
- Qasmaoui, Y., & Haqiq, A. "Solid-flow: A flow rules security mechanism for SDN", In *2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)* (pp. 1-7). IEEE. DOI: 10.1109/CloudTech.2017.8284734. October 2017**. (Scopus)**

REFERENCES

Ahmad, I., Namal, S., Ylianttila, M., & Gurtov, A. (2015). Security in Software Defined Networks: A Survey. *IEEE Communications Surveys and Tutorials*, *17*(4), 2317–2346. https://doi.org/10.1109/COMST.2015.2474118

Akhunzada, A., Gani, A., Anuar, N. B., Abdelaziz, A., Khan, M. K., Hayat, A., & Khan, S. U. (2016). Secure and dependable software defined networks. *Journal of Network and Computer Applications*, *61*, 199–221. https://doi.org/10.1016/j.jnca.2015.11.012

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O\textquoterightKeeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., & Fetzer, C. (2016). {SCONE}: Secure Linux Containers with Intel {SGX}. *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

Assis, M. V. O. De, Hamamoto, A. H., Abrão, T., & Proença, M. L. (2017). A Game Theoretical Based System Using Holt-Winters and Genetic Algorithm With Fuzzy Logic for DoS/DDoS Mitigation on SDN Networks. *IEEE Access*, *5*, 9485–9496. https://doi.org/10.1109/ACCESS.2017.2702341

Banikazemi, M., Olshefski, D., Shaikh, A., Tracey, J., & Wang, G. (2013). Meridian: an SDN platform for cloud network services. *IEEE Communications Magazine*, *51*(2), 120–127. https://doi.org/10.1109/MCOM.2013.6461196

Baumann, A., Peinado, M., & Hunt, G. (2015). Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, *33*(3). https://doi.org/10.1145/2799647

Beacon. (n.d.). *"Beacon homepage," https://OpenFlow.stanford.edu/display/ Beacon/Home*. "Beacon Homepage," Https://OpenFlow.Stanford.Edu/Display/ Beacon/Home.

Beshley, M., Pryslupskyi, A., Panchenko, O., & Beshley, H. (2019). SDN/Cloud Solutions for Intent-Based Networking. *2019 3rd International Conference on Advanced Information and Communications Technologies (AICT)*, 22–25. https://doi.org/10.1109/AIACT.2019.8847731

Bhushan, K., & Gupta, B. B. (2018). Distributed denial of service (DDoS) attack mitigation in software defined network (SDN)-based cloud computing environment. *Journal of Ambient Intelligence and Humanized Computing*, 1–13.

Bianco, A., Giaccone, P., Mashayekhi, R., Ullio, M., & Vercellone, V. (2017). Scalability of ONOS reactive forwarding applications in ISP networks. *Computer Communications*, *102*, 130–138. https://doi.org/10.1016/j.comcom.2016.09.007

Botelho, F., Bessani, A., Ramos, F. M. V., & Ferreira, P. (2014). SMaRtLight: A Practical Fault-Tolerant SDN Controller. *Https://Arxiv.Org/Abs/1407.6062*, 1–7. http://arxiv.org/abs/1407.6062

Braga, R., Mota, E., & Passito, A. (2010). Lightweight DDoS flooding attack detection using NOX/OpenFlow. *IEEE Local Computer Network Conference*, 408–415. https://doi.org/10.1109/LCN.2010.5735752

Buragohain, C., & Medhi, N. (2016). FlowTrApp: An SDN based architecture for DDoS attack detection and mitigation in data centers. *2016 3rd International Conference on Signal Processing and Integrated Networks (SPIN)*, 519–524. https://doi.org/10.1109/SPIN.2016.7566750

Chung, C., Member, S., Khatkar, P., & Xing, T. (2013). NICE : Network Intrusion Detection and Countermeasure. *IEEE Transactions on Dependable Secure Computting*, *10*(4), 1–14. http://dblp.uni-trier.de/db/journals/tdsc/tdsc10.html#ChungKXLH13

CISCO. (2020). *Introduction: State of networking for the digital age*. https://www.cisco.com/c/m/en_us/solutions/enterprise-networks/networking-report.html?oid=rpten018612&CCID=cc001406&DTID=odicdc000509#

Controller, T. (n.d.). *Trema controller*. Full-Stack OpenFlow Framework in Ruby and C. Retrieved September 12, 2020, from https://trema.github.io/trema/, 201

Costan, V., & Devadas, S. (2016). Intel SGX Explained. *IACR Cryptol. EPrint Arch.*, *86*, 1–118.

Coughlin, M., Keller, E., & Wustrow, E. (2017). Trusted Click: Overcoming Security Issues of NFV in the Cloud. *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 31–36. https://doi.org/10.1145/3040992.3040994

Cui, Y., Yan, L., Li, S., Xing, H., Pan, W., Zhu, J., & Zheng, X. (2016). SD-Anti-DDoS: Fast and efficient DDoS defense in software-defined networks. *Journal of Network and Computer Applications*, *68*, 65–79. https://doi.org/10.1016/j.jnca.2016.04.005

Cziva, R., Jouët, S., Stapleton, D., Tso, F. P., & Pezaros, D. P. (2016). SDN-based virtual machine management for cloud data centers. *IEEE Transactions on Network and Service Management*, *13*(2), 212–225.

D'Cruze, H., Wang, P., Sbeit, R. O., & Ray, A. (2018). A Software-Defined Networking (SDN) Approach to Mitigating DDoS Attacks. In *Information Technology-New Generations* (pp. 141–145). Springer.

Dargahi, T., Caponi, A., Ambrosin, M., Bianchi, G., & Conti, M. (2017). A Survey on the Security of Stateful SDN Data Planes. *IEEE Communications Surveys & Tutorials*.

Dridi, L., & Zhani, M. F. (2016). SDN-Guard: DoS Attacks Mitigation in SDN Networks. *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, 212–217. https://doi.org/10.1109/CloudNet.2016.9

Erickson, D. (2013). The Beacon Openflow Controller. *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 13–18. https://doi.org/10.1145/2491185.2491189

Farahmandian, S., & Hoang, D. B. (2016). Security For Software-Defined (Cloud, SDN And NFV) Infrastructures--Issues And Challenges. *Eight International Conference on Network and Communications Security*.

Feng, T., Bi, J., Hu, H., Yao, G., & Xiao, P. (2012). InSAVO: Intra-AS IP source address validation solution with OpenRouter. *In Proceedings of INFOCOM*.

Ferguson, A., Guha, A., & Liang, C. (2013). A Northbound API for Sharing SDNs. *Nextstep-Esolutions.Com*, 1–2. http://nextstep-esolutions.com/Clients/ONS2.0/pdf/2013/research_track/poster_papers/ons2013-final64.pdf

Fichera, S., Galluccio, L., Grancagnolo, S. C., Morabito, G., & Palazzo, S. (2015). OPERETTA: An OPEnflow-based REmedy to mitigate TCP SYNFLOOD Attacks against web servers. *Computer Networks*, *92*, 89–100. https://doi.org/https://doi.org/10.1016/j.comnet.2015.08.038

Fielding, R. T., & Taylor, R. N. (2000). Architectural styles and the design of network-based software architectures. *(Vol. 7). Irvine: University of California, Irvine*.

Filsfils, C., Previdi, S. B., & Vasseur, J. P. (. (2011). U.S. Patent No. 7,978,708. *Washington, DC: U.S. Patent and Trademark Office*.

Floodlight. (n.d.-a). *Floodlight OpenFlow Controller*. Floodlight Homepage: Http://Www. Projectfloodlight.Org/Floodlight/.

Floodlight, P. (n.d.-b). Big Switch Networks. *[Online]. Available: Http://Www.Projectfloodlight.Org/Floodlight/*.

Foster, N., Harrison, R., Freedman, M., Monsanto, C., Rexford, J., Story, A., & Walker, D. (2011). Frenetic: A Network Programming Language. In *Sigplan Notices - SIGPLAN* (Vol. 46). https://doi.org/10.1145/2034574.2034812

Fruhlinger, O. (2019). *How networking pros can prepare for the software-defined future*. https://www.hpe.com/us/en/insights/articles/how-networking-pros-can-prepare-for-the-software-defined-future-1904.html

Girtler, D., & Paladi, N. (2017). Component integrity guarantees in software-defined networking infrastructure. *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 292–296. https://doi.org/10.1109/NFV-SDN.2017.8169858

GitHub. (n.d.). *An Open Source SDN Controller for Cloud Computing Data Centers*. [Online]. Available: Https://Github.Com/China863SDN/DCFabric.

Graydon, I., Beatty, E., Paul, S., Us, M. N., & Hauck, J. A. (2006). *Method and apparatus to provide secure application execution*.

Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., & Shenker, S. (2008). NOX: Towards an operating system for networks. In *Computer Communication Review* (Vol. 38).

Gueron, S. (2012). Speeding Up SHA-1, SHA-256 and SHA-512 on the 2nd Generation Intel® Core™ Processors. *2012 Ninth International Conference on Information Technology - New Generations*, 824–826. https://doi.org/10.1109/ITNG.2012.62

Gueron, Shay. (2016). A Memory Encryption Engine Suitable for General Purpose Processors. *Cryptology EPrint Archive, Report 2016/204*.

Hall, R. S., & Cervantes, H. (2004). An OSGi implementation and experience report. *First IEEE Consumer Communications and Networking Conference, 2004. CCNC 2004.*, 394–399. https://doi.org/10.1109/CCNC.2004.1286894

Hassas Yeganeh, S., & Ganjali, Y. (2012). Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, 19–24. https://doi.org/10.1145/2342441.2342446

Hu, D., Li, S., Xue, N., Chen, C., Ma, S., Fang, W., & Zhu, Z. (2015). Design and Demonstration of SDN-Based Flexible Flow Converging with Protocol-Oblivious Forwarding (POF). *2015 IEEE Global Communications Conference (GLOBECOM)*, 1–6. https://doi.org/10.1109/GLOCOM.2015.7417001

Hu, H., Han, W., Ahn, G.-J., & Zhao, Z. (2014). FLOWGUARD: Building Robust Firewalls for Software-Defined Networks. *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, 97–102. https://doi.org/10.1145/2620728.2620749

Hu, Z., Wang, M., Yan, X., Yin, Y., & Luo, Z. (2015). A comprehensive security architecture for SDN. *2015 18th International Conference on Intelligence in Next Generation Networks, ICIN 2015*, 30–37. https://doi.org/10.1109/ICIN.2015.7073803

Huawei. (2020). *Huawei Sees an SDN Future*. https://www.lightreading.com/carrier-sdn/sdn-architectures/huawei-sees-an-sdn-future/d/d-id/702425

Hussein, A., Elhajj, I. H., Chehab, A., & Kayssi, A. (2016). SDN Security Plane: An Architecture for Resilient Security Services. *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, 54–59. https://doi.org/10.1109/IC2EW.2016.15

Intel®. (2014). Intel® Sofware Guard Extensions Programming Reference. *(Cited on Pages 13 and 14.)*.

Ishii, S., Kawai, E., Takata, T., Kanaumi, Y., Saito, S., Kobayashi, K., & Shimojo, S. (2012). Extending the RISE controller for the interconnection of RISE and OS3E/NDDI. *2012 18th IEEE International Conference on Networks (ICON)*, 243–248. https://doi.org/10.1109/ICON.2012.6506564

Ismail, M. F. H. and M. A. (2019). Distributed Shadow Controllers based Moving Target Defense Framework for Control Plane Security. *International Journal of Advanced Computer Science and Applications*, *10*(12), 150–156. https://doi.org/10.14569/IJACSA.2019.0101221

J.C., C. C., J.C., I., & J.F., B. V. (2020). *Security in SDN: A comprehensive survey*.

http://hdl.handle.net/11407/5743

Jain, R. (2012). OpenADN: Mobile Apps on Global Clouds Using Software Defined Networking. *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, 1–2. https://doi.org/10.1145/2307849.2307851

Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., & Vahdat, A. (2013a). B4: Experience with a Globally-Deployed Software Defined Wan. *SIGCOMM Comput. Commun. Rev.*, *43*(4), 3–14. https://doi.org/10.1145/2534169.2486019

Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., & Vahdat, A. (2013b). B4: Experience with a Globally-Deployed Software Defined Wan. *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 3–14. https://doi.org/10.1145/2486001.2486019

Jeong, K., Kim, J., & Kim, Y. (2012). QoS-aware Network Operating System for software defined networking with Generalized OpenFlows. *2012 IEEE Network Operations and Management Symposium*, 1167–1174. https://doi.org/10.1109/NOMS.2012.6212044

Juba, Y., Huang, H.-H., & Kawagoe, K. (2013). Dynamic Isolation of Network Devices Using OpenFlow for Keeping LAN Secure from Intra-LAN Attack. *Procedia Computer Science*, *22*, 810–819. https://doi.org/https://doi.org/10.1016/j.procs.2013.09.163

Juniper. (2019). *What Is the Future of SDN?* https://blogs.juniper.net/en-us/ask-juniper/what-is-the-future-of-sdn

Karmakar, K. K., Varadharajan, V., Nepal, S., & Tupakula, U. (2019). SDN Enabled Secure IoT Architecture. *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 581–585.

Khattak, Z. K., Awais, M., & Iqbal, A. (2014). Performance evaluation of OpenDaylight SDN controller. *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 671–676. https://doi.org/10.1109/PADSW.2014.7097868

Kim, S., Shin, Y., Ha, J., Kim, T., & Han, D. (2015). A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. https://doi.org/10.1145/2834050.2834100

Kohler, T., Dürr, F., & Rothermel, K. (2018). ZeroSDN: A Highly Flexible and Modular Architecture for Full-Range Distribution of Event-Based Network Control. *IEEE Transactions on Network and Service Management*, *15*(4), 1207–1221. https://doi.org/10.1109/TNSM.2018.2873886

Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., & Shenker, S. (2010). Onix: A distributed control platform for large-scale production networks. In OSDI. *In OSDI*, *10*.

Kreutz, D., Ramos, F. M. V, & Verissimo, P. (2013). Towards Secure and Dependable Software-Defined Networks. *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 55–60. https://doi.org/10.1145/2491185.2491199

Kuerban, M., Tian, Y., Yang, Q., Jia, Y., Huebert, B., & Poss, D. (2016). FlowSec: DOS Attack Mitigation Strategy on SDN Controller. *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 1–2. https://doi.org/10.1109/NAS.2016.7549402

Lara, A., Kolasani, A., & Ramamurthy, B. (2013). Network Innovation using OpenFlow: A Survey. *IEEE Communications Surveys & Tutorials*, *PP*(99), 1–20. https://doi.org/10.1109/SURV.2013.081313.00105

Lee, B., Park, S. H., Shin, J., & Yang, S. (2014). IRIS: The Openflow-based Recursive SDN controller. *16th International Conference on Advanced Communication Technology*, 1227–1231. https://doi.org/10.1109/ICACT.2014.6779154

Lefebvre, V., Santinelli, G., Müller, T., & Götzfried, J. (2018). Universal Trusted Execution Environments for Securing SDN/NFV Operations. *Proceedings of the 13th International Conference on Availability, Reliability and Security*. https://doi.org/10.1145/3230833.3233256

Liu, Y., Zeng, Z., Liu, X., Zhu, X., & Bhuiyan, M. Z. A. (2020). A Novel Load Balancing and Low Response Delay Framework for Edge-Cloud Network Based on SDN. *IEEE Internet of Things Journal*, *7*(7), 5922–5933. https://doi.org/10.1109/JIOT.2019.2951857

Lorenz, C., Hock, D., Scherer, J., Durner, R., Kellerer, W., Gebert, S., Gray, N., Zinner, T., & Tran-Gia, P. (2017). An SDN/NFV-Enabled Enterprise Network Architecture Offering Fine-Grained Security Policy Enforcement. *IEEE Communications Magazine*, *55*(3), 217–223. https://doi.org/10.1109/MCOM.2017.1600414CM

Macías, S. G., Gaspary, L. P., & Botero, J. F. (2020). ORACLE: Collaboration of Data and Control Planes to Detect DDoS Attacks. *ArXiv Preprint ArXiv:2009.10798*.

Maestro. (n.d.). *Maestro*. Maestro Homepage: Http://Zhengcai.Github.Io/Maestro-Platform/.

Matias, J., Tornero, B., Mendiola, A., Jacob, E., & Toledo, N. (2012). Implementing Layer 2 Network Virtualization Using OpenFlow: Challenges and Solutions. *2012 European Workshop on Software Defined Networking*, 30–35. https://doi.org/10.1109/EWSDN.2012.18

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., & Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, *38*(2), 69–74.

Medina, J., Paladiy, N., & Arlosz, P. (2019). Protecting OpenFlow using Intel SGX. *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 1–6. https://doi.org/10.1109/NFV-SDN47374.2019.9039980

Mentel, S., Konieczny, M., & Zieliński, S. (2016). Highly customizable framework for performance evaluation of LOOM-based SDN controllers. *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 969–977.

Monaco, M., Michel, O., & Keller, E. (2013). Applying Operating System Principles to SDN Controller Design. *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. https://doi.org/10.1145/2535771.2535789

Monsanto, C., Foster, N., Harrison, R., & Walker, D. (2012). A Complier and Run-time system for Network Programming Languages. In *Sigplan Notices - SIGPLAN* (Vol. 47). https://doi.org/10.1145/2103621.2103685

Moy. (1998). OSPF version 2. *Https://Www.Hjp.at/Doc/Rfc/Rfc2178.Html*.

Narayanan, R., Kotha, S., Lin, G., Khan, A., Rizvi, S., Javed, W., Khan, H., & Khayam, S. A. (2012). Macroflows and Microflows: Enabling Rapid Network Innovation through a Split SDN Data Plane. *2012 European Workshop on Software Defined Networking*, 79–84. https://doi.org/10.1109/EWSDN.2012.16

Nguyen, T., & Yoo, M. (2016). Attacks on host tracker in SDN controller: Investigation and prevention. *2016 International Conference on Information and Communication Technology Convergence (ICTC)*, 610–612. https://doi.org/10.1109/ICTC.2016.7763545

NODEFLOW. (n.d.). NODEFLOW: An openflow controller node style. *[Online]. Available: Http://Garyberger.Net/?P=537*.

Nunes, B. A. A., Mendonca, M., Nguyen, X.-N., Obraczka, K., & Turletti, T. (2014). A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, *16*(3), 1617–1634.

Oktian, Y. E., Lee, S. G., Lee, H. J., & Lam, J. H. (2017). Distributed SDN controller system: A survey on design choice. *Computer Networks*, *121*, 100–111. https://doi.org/10.1016/j.comnet.2017.04.038

Oliveira, B. T. de, Gabriel, L. B., & Margi, C. B. (2015). TinySDN: Enabling Multiple Controllers for Software-Defined Wireless Sensor Networks. *IEEE Latin America Transactions*, *13*(11), 3690–3696. https://doi.org/10.1109/TLA.2015.7387950

Oliveira, R. L. S. de, Schweitzer, C. M., Shinoda, A. A., & Prete, L. R. (2014). Using Mininet for emulation and prototyping Software-Defined Networks. *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, 1–6. https://doi.org/10.1109/ColComCon.2014.6860404

ONF. (n.d.). *Open Networking Foundation (https://opennetworking.org/) 07 / June/ 2021*.

OpenContrail. (n.d.). Release Notes for Contrail Release 2.20. *Available:Https://Www.Juniper.Net/Techpubs/En_US/Contrai L2.2/Information-Products/Topic-Collections/Releasenotes/Index.Html*.

OpenDaylight. (n.d.). *OpenDaylight: A Linux Foundation Collaborative Project*. Http: //Www.Opendaylight.Org/.

OpenMUL. (n.d.). OpenMUL SDN Platform. *[Online]. Available: Http://Www.Openmul.Org/Openmul-Controller.Html*.

Owens II, H., & Durresi, A. (2015). Video over Software-Defined Networking (VSDN). *Computer Networks*, *92*, 341–356. https://doi.org/https://doi.org/10.1016/j.comnet.2015.09.009

P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Y. (2015). *"Securing the Software-Defined Network Control Layer,."*

Paladi, N., & Gehrmann, C. (2017). TruSDN: Bootstrapping Trust in Cloud Network Infrastructure. In R. Deng, J. Weng, K. Ren, & V. Yegneswaran (Eds.), *International Conference on Security and Privacy in Communication Systems (pp. 104-124)* (pp. 104–124). Springer International Publishing.

Paladi, N., Karlsson, L., & Elbashir, K. (2018). Trust Anchors in Software Defined Networks. In J. Lopez, J. Zhou, & M. Soriano (Eds.), *Computer Security* (pp. 485–504). Springer International Publishing.

Phan, T. V, & Park, M. (2019). Efficient Distributed Denial-of-Service Attack Defense in SDN-Based Cloud. *IEEE Access*, *7*, 18701–18714. https://doi.org/10.1109/ACCESS.2019.2896783

Phemius, K., Bouet, M., & Leguay, J. (2014). DISCO: Distributed multi-domain SDN controllers. *2014 IEEE Network Operations and Management Symposium (NOMS)*, 1–4. https://doi.org/10.1109/NOMS.2014.6838330

Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., & Gu, G. (2012). *A security enforcement kernel for OpenFlow networks*. https://doi.org/10.1145/2342441.2342466

Prete, L. R., Shinoda, A. A., Schweitzer, C. M., & Oliveira, R. L. S. de. (2014). Simulation in an SDN network scenario using the POX Controller. *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, 1–6. https://doi.org/10.1109/ColComCon.2014.6860403

QASMAOUI, Y., & HAQIQ, A. (2020). Enhanced Solid-Flow: An Enhanced Flow Rules Security Mechanism for SDN. *IAENG International Journal of Computer Science*, *47*(3).

Qasmaoui, Y., & Haqiq, A. (2018). Solid-flow: A flow rules security mechanism for SDN. *Proceedings of 2017 International Conference of Cloud Computing Technologies and Applications, CloudTech 2017*, *2018-Janua*. https://doi.org/10.1109/CloudTech.2017.8284734

Ryu. (n.d.). *Ryu SDN Framework*. Ryu Homepage: Http://Osrg.Github.Io/Ryu/.

Safaa MAHRACH and Abdelkrim HAQIQ. (2020). DDoS Flooding Attack Mitigation in Software Defined Networks. *International Journal of Advanced Computer Science and Applications(IJACSA)*, *11*(1), 693–700. https://doi.org/10.14569/IJACSA.2020.0110185

Sanaei, M., & Mostafavi, S. (2019). Multimedia Delivery Techniques over Software-Defined Networks: A Survey. *2019 5th International Conference on Web Research (ICWR)*, 105–110. https://doi.org/10.1109/ICWR.2019.8765278

Sasaki, T., Perrig, A., & Asoni, D. E. (2016). Control-plane isolation and recovery for a secure SDN architecture. *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 459–464. https://doi.org/10.1109/NETSOFT.2016.7502485

Shaghaghi, A., Kaafar, M. A., Buyya, R., & Jha, S. (2018). Software-Defined Network (SDN) Data Plane Security: Issues, Solutions and Future Directions. *ArXiv Preprint ArXiv:1804.00262*.

Shalimov, A., Nizovtsev, S., Morkovnik, D., & Smeliansky, R. (2015). The Runos OpenFlow Controller. *2015 Fourth European Workshop on Software Defined Networks*, 103–104. https://doi.org/10.1109/EWSDN.2015.69

Sherwood, R., Gibb, G., Yap, K., Appenzeller, G., Casado, M., Mckeown, N., & Parulkar, G. (2009). FlowVisor: A Network Virtualization Layer. *Network*, 15. http://www.openflow.org/downloads/.../openflow-tr-2009-1-flowvisor.pdf%5Cnhttp://www.techrepublic.com/whitepapers/flowvisor-a-network-virtualization-layer/2382721

Shin, J., Kim, T., Lee, B., & Yang, S. (2017). IRIS-HiSA: Highly Scalable and Available Carrier-Grade SDN Controller Cluster. In *Mobile Networks and Applications*. https://doi.org/10.1007/s11036-017-0853-6

Shin, S., Porras, P., Yegneswaran, V., & Gu, G. (n.d.). *A Framework For Integrating Security Services into Software-Defined Networks*. *1*(1), 11–12.

Shin, S., Song, Y., Lee, T., Lee, S., Chung, J., Porras, P., Yegneswaran, V., Noh, J., & Kang, B. B. (2014). Rosemary: A Robust, Secure, and High-Performance Network Operating System. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 78–89. https://doi.org/10.1145/2660267.2660353

Simon P Johnson, Uday R Savagaonkar, Vincent R Scarlata, F. X., & McKeen, and C. V. R. (2015). *Technique for supporting multiple secure enclaves* (Patent No. US Patent 8,972,746.).

SNAC. (n.d.). *SNAC: Simple Network Access Control*. Https://Github.Com/.

Son, J., & Buyya, R. (2018). A Taxonomy of Software-Defined Networking (SDN)-Enabled Cloud Computing. *ACM Computing Surveys (CSUR)*, *51*(3), 59.

Son, J., Dastjerdi, A. V., Calheiros, R. N., & Buyya, R. (2017). SLA-aware and energy-efficient dynamic overbooking in SDN-based cloud data centers. *IEEE Transactions on Sustainable Computing*, *2*(2), 76–89.

Son, S., Shin, S., Yegneswaran, V., Porras, P., & Gu, G. (2013). Model checking invariant security properties in OpenFlow. In *IEEE International Conference on Communications*. https://doi.org/10.1109/ICC.2013.6654813

Stuart, J. B. and S. (2016). Faucet:Deploying SDN in the Enterprise. *ACM Queue*, 1–15.

Suh, J., Choi, H., Yoon, W., You, T., Kwon, T. T., & Choi, Y. (2010). Implementation of Content-Oriented Networking Architecture (CONA): A Focus on DDoS Countermeasure. *1st European NetFPGA Developers Workshop*, 1–5. https://mmlab.snu.ac.kr/publications/docs/2010_EU_netfpga_workshop_jhsuh.pdf

Tootoonchian, A., & Ganjali, Y. (2010). *HyperFlow: A Distributed Control Plane for OpenFlow*.

Tootoonchian, A., Gorbunov, S., Ganjali, Y., Casado, M., & Sherwood, R. (2012, April). On Controller Performance in Software-Defined Networks. *2nd {USENIX} Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*. https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/tootoonchian

Voellmy, A., & Hudak, P. (2011). *Nettle: Taking the Sting Out of Programming Network Routers BT -*

Practical Aspects of Declarative Languages (R. Rocha & J. Launchbury (eds.); pp. 235–249). Springer Berlin Heidelberg.

Voellmy, A., Kim, H., & Feamster, N. (2012). Procera: A language for high-level reactive network control. In *HotSDN'12 - Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks*. https://doi.org/10.1145/2342441.2342451

Voellmy, A., & Wang, J. (2012). Scalable Software Defined Network Controllers. *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 289–290. https://doi.org/10.1145/2342356.2342414

Wang, A., Mei, X., Croft, J., Caesar, M., & Godfrey, B. (2016). Ravel: A Database-Defined Network. *Proceedings of the Symposium on SDN Research*. https://doi.org/10.1145/2890955.2890970

Wang, H., Xu, L., & Gu, G. (2015). FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks. *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 239–250. https://doi.org/10.1109/DSN.2015.27

Wang, M., Liu, J., Chen, J., Liu, X., & Mao, J. (2016). PERM-GUARD: Authenticating the Validity of Flow Rules in Software Defined Networking. *Proceedings - 2nd IEEE International Conference on Cyber Security and Cloud Computing, CSCloud 2015 - IEEE International Symposium of Smart Cloud, IEEE SSC 2015*, *37*, 127–132. https://doi.org/10.1109/CSCloud.2015.89

Wen, X., Chen, Y., Hu, C., Shi, C., & Wang, Y. (2013). *Towards a secure controller platform for openflow applications*. https://doi.org/10.1145/2491185.2491212

Winter, J. (2008). Trusted Computing Building Blocks for Embedded Linux-Based ARM Trustzone Platforms. *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, 21–30. https://doi.org/10.1145/1456455.1456460

Wu, G., Wang, J., Obaidat, M. S., Yao, L., & Hsiao, K.-F. (2019). Dynamic switch migration with noncooperative game towards control plane scalability in SDN. *International Journal of Communication Systems*, *32*(7), e3927. https://doi.org/https://doi.org/10.1002/dac.3927

Yan, Z., Zhang, P., & Vasilakos, A. V. (2016). A security and trust framework for virtualized networks and software-defined networking. *Security and Communication Networks*, *9*(16), 3059–3069. https://doi.org/10.1002/sec.1243

Yeganeh, S. H., & Ganjali, Y. (2014). Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking. *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 1–7. https://doi.org/10.1145/2670518.2673864

Ying Qian, Wanqing You, & Kai Qian. (2016). OpenFlow flow table overflow attacks and countermeasures. *2016 European Conference on Networks and Communications (EuCNC)*, 205–209. https://doi.org/10.1109/EuCNC.2016.7561033

R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda and Ligia Rodrigues Prete *(2018)*. Using Mininet for emulation and prototyping Software-Defined Networks. *IEEE Colombian Conference on Communications and Computing (COLCOM)*, pp. 1-6, doi: 10.1109/ColComCon.2014.6860404.

APPENDICES

Appendix 1: Mininet (R. L. S. de Oliveira, C. M. Schweitzer, 2018)

Mininet is a virtual testbed enabling the development and testing of network tools and protocols. With a single command, Mininet can create a realistic virtual network on any type of machine (Virtual Machine (VM), cloud-hosted, or native). Mininet offers the following features:

- Fast prototyping for new networking protocols.
- Simplified testing for complex topologies without the need of buying expensive hardware.
- Realistic execution as it runs real code on the Unix and Linux kernels.
- Open source environment backed by a large community contributing extensive documentation.

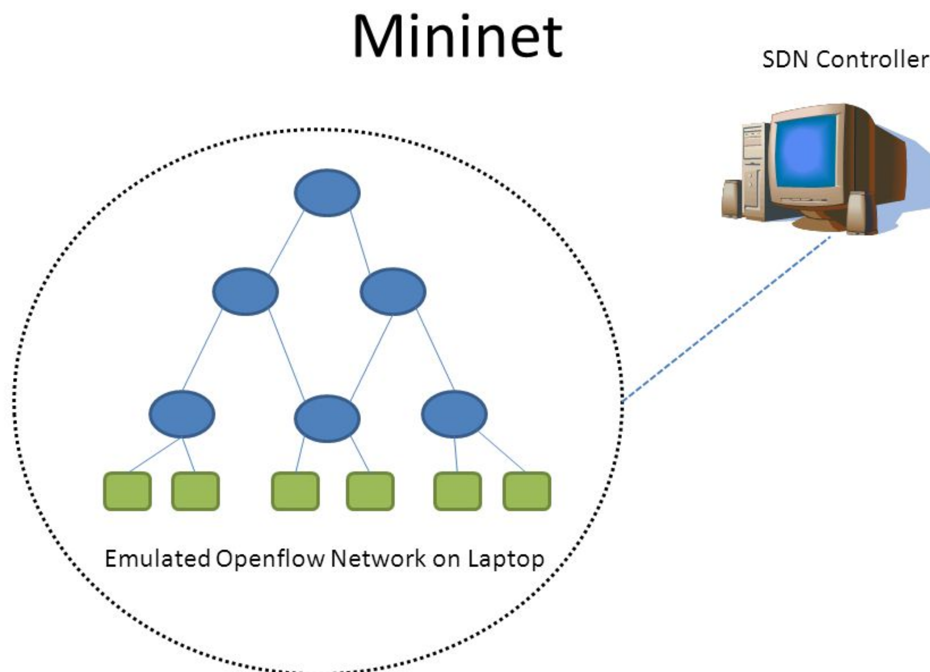


Fig. 31. Mininet emulated network with SDN controller.

Mininet is useful for development, teaching, and research as it is easy to customize and interact with it through the CLI or the GUI. Mininet was originally designed to experiment with OpenFlow2 and Software-Defined Networking (SDN).

Mininet's logical nodes can be connected into networks. These nodes are sometimes called containers, or more accurately, network namespaces. Containers consume sufficiently fewer resources that networks of over a thousand nodes have created, running on a single laptop. A Mininet container is a process (or group of processes) that no longer has access to all the host

system's native network interfaces. Containers are then assigned virtual Ethernet interfaces, which are connected to other containers through a virtual switch 4. Mininet connects a host and a switch using a virtual Ethernet (veth) link.

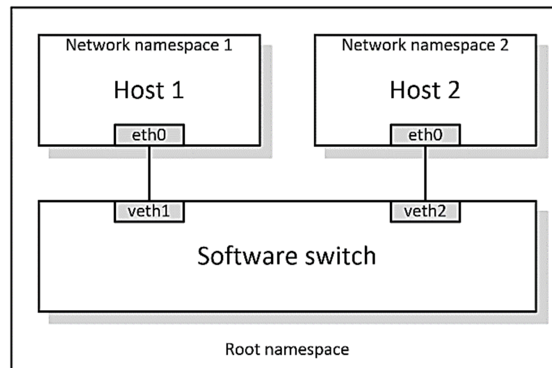


Fig. 32. Network namespaces and virtual Ethernet links

Each container is an independent network namespace, a lightweight virtualization feature that provides individual processes with separate network interfaces, routing tables, and Address Resolution Protocol (ARP) tables. Mininet provides network emulation opposed to simulation, allowing all network software at any layer to be simply run as is; i.e. nodes run the native network software of the physical machine. On the other hand, in a simulated environment applications and protocol implementations need to be ported to run within the simulator before they can be used.

Appendix 2: Mininet simple code over python script to run pings

The following sample code uses python to create the traffic that runs through the network. First, we import Mininet to use its specific commands to create topology in different layers, then we build a flow table and run pings.

```
1.  """ A Testing module for SDN Floodlight
2.  """
3.
4.  from mininet.net import Mininet
5.  from mininet.node import RemoteController
6.  from mininet.topolib import TreeTopo
7.  from mininet.log import setLogLevel
8.  from mininet.cli import CLI
9.
10. import time
11.
12. class pingall(object):
13.     """ A 2 layer  topology
14.                         dpid:1
15.                         +----+
16.                 +--------+ s1 +-------+
17.                 |     p1+----+p2    |
18.                 |p3                p3|
19.               +----+             +----+
20.               | s2 |dpid:2   dpid:3| s3 |
21.               +----+             +----+
22.              p1| |p2             p1| |p2
23.          +-----+ +---+         +---+ +---+
24.          |          |         |        |
25.        +-+--+       |       +-+--+     |
26.        | h1 |       |       | h3 |     |
27.        +----+       |       +----+     |
28.     00:00:00:00:00:01   |  00:00:00:00:00:03 |
29.                +--+-+              +-+--+
30.                | h2 |              | h4 |
31.                +----+              +----+
32.           00:00:00:00:00:02     00:00:00:00:00:04
33.     """
34.
35.     def __init__(self, controller_ip, controller_port):
36.         self.controller_ip = controller_ip
37.         self.controller_port = controller_port
38.         self.tree = TreeTopo(depth=2, fanout=2)
39.
40.         # arguments
41.         # - autoSetMacs=True: set MAC addrs automatically like IP addresses
42.         # - listenPort=9934 (optional): base listening port to open
43.         self.net = Mininet(topo=self.tree,
44.                            controller=None,
45.                            autoSetMacs=True,
46.                            listenPort=6634)
47.         self.net.addController('floodlightctrl',
48.                                controller=RemoteController,
49.                                ip=self.controller_ip,
50.                                port=self.controller_port)
51.         self.sw = self.net.switches
```

```
52.
53.     def kick_start(self):
54.         """ Fire up mininet with customized topology and settings
55.         """
56.         self.net.start()
57.         self.net.waitConnected()
58.
59.     def build_table(self):
60.         """ Construct ovswitches' flow table
61.         """
62.         #
63.         self.sw[0].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:01,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:2')
64.         self.sw[0].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:02,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:2')
65.         self.sw[0].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:03,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:1')
66.         self.sw[0].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:04,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:1')
67.
68.         self.sw[1].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:01,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:2,3')
69.         self.sw[1].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:02,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:1,3')
70.         self.sw[1].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:03,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:1,2')
71.         self.sw[1].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:04,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:1,2')
72.
73.         self.sw[2].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:03,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:2,3')
74.         self.sw[2].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:04,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:1,3')
75.         self.sw[2].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:01,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:1,2')
76.         self.sw[2].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:02,dl_dst=ff:ff:ff:ff:ff:ff,idle_timeout=0,actions=output:1,2')
77.
78.         # For ICMP request/reply and ARP reply
79.         self.sw[0].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,idle_timeout=0,actions=output:2')
80.         self.sw[0].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04,idle_timeout=0,actions=output:2')
81.         self.sw[0].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04,idle_timeout=0,actions=output:2')
82.         self.sw[0].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03,idle_timeout=0,actions=output:2')
83.         self.sw[0].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,idle_timeout=0,actions=output:1')
84.         self.sw[0].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02,idle_timeout=0,actions=output:1')
85.         self.sw[0].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02,idle_timeout=0,actions=output:1')
86.         self.sw[0].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01,idle_timeout=0,actions=output:1')
87.
88.         self.sw[1].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,idle_timeout=0,actions=output:2')
```

```
89.         self.sw[1].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,idle_timeout=0,actions=output:3')
90.         self.sw[1].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04,idle_timeout=0,actions=output:3')
91.         self.sw[1].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01,idle_timeout=0,actions=output:1')
92.         self.sw[1].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03,idle_timeout=0,actions=output:3')
93.         self.sw[1].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04,idle_timeout=0,actions=output:3')
94.         self.sw[1].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,idle_timeout=0,actions=output:1')
95.         self.sw[1].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02,idle_timeout=0,actions=output:2')
96.         self.sw[1].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01,idle_timeout=0,actions=output:1')
97.         self.sw[1].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02,idle_timeout=0,actions=output:2')
98.
99.         self.sw[2].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,idle_timeout=0,actions=output:3')
100.        self.sw[2].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02,idle_timeout=0,actions=output:3')
101.        self.sw[2].dpctl('add-flow', 'in_port=1,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:04,idle_timeout=0,actions=output:2')
102.        self.sw[2].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01,idle_timeout=0,actions=output:3')
103.        self.sw[2].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02,idle_timeout=0,actions=output:3')
104.        self.sw[2].dpctl('add-flow', 'in_port=2,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:03,idle_timeout=0,actions=output:1')
105.        self.sw[2].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,idle_timeout=0,actions=output:1')
106.        self.sw[2].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04,idle_timeout=0,actions=output:2')
107.        self.sw[2].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03,idle_timeout=0,actions=output:1')
108.        self.sw[2].dpctl('add-flow', 'in_port=3,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04,idle_timeout=0,actions=output:2')
109.
110.    def show_table(self):
111.        """ flush the switch flow table
112.        """
113.        print 'FLOW TABLES'
114.        for i in self.sw:
115.            print '*** switch %x ***' % int(i.dpid, 16)
116.            print i.dpctl('dump-flows')
117.
118.    def ping_test(self):
119.        """ sent ping for conctivity test all pings
120.        """
121.        self.net.pingAll()
122.
123.    def __del__(self):
124.        self.net.stop()
125.
126. def main():
```

```
127.     """ Main procedure
128.     """
129.     topo = TwoLayerTreeTopo('127.0.0.1', 9933)
130.     topo.kick_start()
131.     topo.build_table()
132.     topo.show_table()
133.     topo.ping_test()
134.
135. if __name__ == '__main__':
136.     setLogLevel('info')
137.     main()
```

Appendix 3: SGX_ec256_public_t.cpp file loaded by IStorageSourceService.java

The following appendix shows the code used to handle intel SGX operation. We use Java Native Interface to run the cpp code.

```
1.  #include "CryptoEnclave_t.h"
2.  
3.  #include "sgx_trts.h"
4.  
5.  #include "sgx_tcrypto.h"
6.  
7.  #include "stdlib.h"
8.  
9.  #include <string.h>
10. 
11. #define BUFLEN 2048
12. static sgx_aes_gcm_128bit_key_t key = {
13.   0x0,
14.   0x1,
15.   0x2,
16.   0x3,
17.   0x4,
18.   0x5,
19.   0x6,
20.   0x7,
21.   0x8,
22.   0x9,
23.   0xa,
24.   0xb,
25.   0xc,
26.   0xd,
27.   0xe,
28.   0xf
29. };
30. 
31. void decryptMessage(char * encMessageIn, size_t len, char * decMessageOut, size_t lenOut) {
32.   uint8_t * encMessage = (uint8_t * ) encMessageIn;
33.   uint8_t p_dst[BUFLEN] = {
34.     0
35.   };
36. 
37.   sgx_rijndael128GCM_decrypt( &
38.     key,
39.     encMessage + SGX_AESGCM_MAC_SIZE + SGX_AESGCM_IV_SIZE,
40.     lenOut,
41.     p_dst,
42.     encMessage + SGX_AESGCM_MAC_SIZE, SGX_AESGCM_IV_SIZE,
43.     NULL, 0,
44.     (sgx_aes_gcm_128bit_tag_t * ) encMessage);
45.   memcpy(decMessageOut, p_dst, lenOut);
46.   emit_debug((char * ) p_dst);
47. }
48. 
49. void encryptMessage(char * decMessageIn, size_t len, char * encMessageOut, size_t lenOut) {
50.   uint8_t * origMessage = (uint8_t * ) decMessageIn;
51.   uint8_t p_dst[BUFLEN] = {
52.     0
53.   };
54. 
55.   // Generate the IV (nonce)
56.   sgx_read_rand(p_dst + SGX_AESGCM_MAC_SIZE, SGX_AESGCM_IV_SIZE);
57. 
58.   sgx_rijndael128GCM_encrypt( &
```

```
59.     key,
60.     origMessage, len,
61.     p_dst + SGX_AESGCM_MAC_SIZE + SGX_AESGCM_IV_SIZE,
62.     p_dst + SGX_AESGCM_MAC_SIZE, SGX_AESGCM_IV_SIZE,
63.     NULL, 0,
64.     (sgx_aes_gcm_128bit_tag_t * )(p_dst));
65.   memcpy(encMessageOut, p_dst, lenOut);
66. }
```

Appendix 4: sample code of modified SDN (floodlight) controller storage module

The following codes demonstrate a modified code of the floodlight controller to make the necessary calls to Intel SGX.

```
1.  /**
2.  *   Copyright 2011, Big Switch Networks, Inc.
3.  *   Originally created by David Erickson, Stanford University
4.  *
5.  *   Licensed under the Apache License, Version 2.0 (the "License"); you may
6.  *   not use this file except in compliance with the License. You may obtain
7.  *   a copy of the License at
8.  *
9.  *        http://www.apache.org/licenses/LICENSE-2.0
10. *
11. *   Unless required by applicable law or agreed to in writing, software
12. *   distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
13. *   WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
14. *   License for the specific language governing permissions and limitations
15. *   under the License.
16. **/
17.
18. package net.floodlightcontroller.storage;
19.
20. import java.util.List;
21. import java.util.Map;
22. import java.util.Set;
23. import java.util.concurrent.Future;
24.
25. import net.floodlightcontroller.core.module.IFloodlightService;
26.
27. public interface IStorageSourceService extends IFloodlightService {
28.
29.     /** Set the column to be used as the primary key for a table. This should
30.      * be guaranteed to be unique for all of the rows in the table, although the
31.      * storage API does not necessarily enforce this requirement. If no primary
32.      * key name is specified for a table then the storage API assumes there is
33.      * a column named "id" that is used as the primary key. In this case when
34.      * a new row is inserted using the storage API and no id is specified
35.      * explictly in the row data, the storage API automatically generates a
36.      * unique ID (typically a UUID) for the id column. To work across all
37.      * possible implementations of the storage API it is safest, though, to
38.      * specify the primary key column explicitly.
39.      * FIXME: It's sort of a kludge to have to specify the primary key column
40.      * here. Ideally there would be some sort of metadata -- perhaps stored
41.      * directly in the table, at least in the NoSQL case -- that the
42.      * storage API could query to obtain the primary key info.
43.      * @param tableName The name of the table for which we're setting the key
44.      * @param primaryKeyName The name of column to be used as the primary key
45.      */
46.     public void setTablePrimaryKeyName(String tableName, String primaryKeyName);
47.
48.     /** Create a new table if one does not already exist with the given name.
49.      *
50.      * @param tableName The name of the table to create.
51.      * @param indexedColumns Which columns should be indexed
52.      */
53.     void createTable(String tableName, Set<String> indexedColumns);
54.     //*load INtel SXG sdk using JNI interface
55.
56.      static {
57.         System.loadLibrary("sgx_ec256_public_t");
```

```
58.     }
59.
60.     /**
61.      * @return the set of all tables that have been created via createTable
62.      */
63.     Set<String> getAllTableNames();
64.
65.     /** Create a query object representing the given query parameters. The query
66.      * object can be passed to executeQuery to actually perform the query and obtain
67.      * a result set.
68.      *
69.      * @param tableName The name of the table to query.
70.      * @param columnNames The list of columns to return in the result set.
71.      * @param predicate The predicate that specifies which rows to return in the result set.
72.      * @param ordering Specification of order that rows are returned from the result set
73.      * returned from executing the query. If the ordering is null, then rows are returned
74.      * in an implementation-specific order.
75.      * @return Query object to be passed to executeQuery.
76.      */
77.     IQuery createQuery(String tableName, String[] columnNames, IPredicate predicate, RowOrdering ordering);
78.
79.     /** Execute a query created with createQuery.
80.      *
81.      * @param query The query to execute
82.      * @return The result set containing the rows/columns specified in the query.
83.      */
84.     IResultSet executeQuery(IQuery query);
85.
86.     /** Execute a query created with the given query parameters.
87.      *
88.      * @param tableName The name of the table to query.
89.      * @param columnNames The list of columns to return in the result set.
90.      * @param predicate The predicate that specifies which rows to return in the result set.
91.      * @param ordering Specification of order that rows are returned from the result set
92.      * returned from executing the query. If the ordering is null, then rows are returned
93.      * in an implementation-specific order.
94.      * @return The result set containing the rows/columns specified in the query.
95.      */
96.     IResultSet executeQuery(String tableName, String[] columnNames, IPredicate predicate,
97.             RowOrdering ordering);
98.
99.     /** Execute a query and call the row mapper to map the results to Java objects.
100.      *
101.      * @param tableName The name of the table to query.
102.      * @param columnNames The list of columns to return in the result set.
103.      * @param predicate The predicate that specifies which rows to return in the result set.
104.      * @param ordering Specification of order that rows are returned from the result set
105.      * returned from executing the query. If the ordering is null, then rows are returned
106.      * in an implementation-specific order.
107.      * @param rowMapper The client-supplied object that maps the data in a row in the result
108.      * set to a client object.
109.      * @return The result set containing the rows/columns specified in the query.
110.      */
111.     Object[] executeQuery(String tableName, String[] columnNames, IPredicate predicate,
112.             RowOrdering ordering, IRowMapper rowMapper);
113.
114.     /** Insert a new row in the table with the given column data.
115.      * If the primary key is the default value of "id" and is not specified in the
116.      * then a unique id will be automatically assigned to the row.
117.      * @param tableName The name of the table to which to add the row
118.      * @param values The map of column names/values to add to the table.
119.      */
120.     void insertRow(String tableName, Map<String,Object> values);
121.
122.     /** Update or insert a list of rows in the table.
123.      * The primary key must be included in the map of values for each row.
```

```
124.     * @param tableName The table to update or insert into
125.     * @param values The map of column names/values to update the rows
126.     */
127.    void updateRows(String tableName, List<Map<String,Object>> rows);
128.
129.    /** Update the rows in the given table. Any rows matching the predicate
130.     * are updated with the column names/values specified in the values map.
131.     * (The values map should not contain the special column "id".)
132.     * @param tableName The table to update
133.     * @param predicate The predicate to use to select which rows to update
134.     * @param values The map of column names/values to update the rows.
135.     */
136.    void updateMatchingRows(String tableName, IPredicate predicate, Map<String,Object> values);
137.
138.    /** Update or insert a row in the table with the given row key (primary
139.     * key) and column names/values. (If the values map contains the special
140.     * column "id", its value must match rowId.)
141.     * @param tableName The table to update or insert into
142.     * @param rowKey The ID (primary key) of the row to update
143.     * @param values The map of column names/values to update the rows
144.     */
145.    void updateRow(String tableName, Object rowKey, Map<String,Object> values);
146.
147.    /** Update or insert a row in the table with the given column data.
148.     * The primary key must be included in the map of values.
149.     * @param tableName The table to update or insert into
150.     * @param values The map of column names/values to update the rows
151.     */
152.    void updateRow(String tableName, Map<String,Object> values);
153.
154.    /** Delete the row with the given primary key.
155.     *
156.     * @param tableName The table from which to delete the row
157.     * @param rowKey The primary key of the row to delete.
158.     */
159.    void deleteRow(String tableName, Object rowKey);
160.
161.    /** Delete the rows with the given keys.
162.     *
163.     * @param tableName The table from which to delete the rows
164.     * @param rowKeys The set of primary keys of the rows to delete.
165.     */
166.    void deleteRows(String tableName, Set<Object> rowKeys);
167.
168.    /**
169.     * Delete the rows that match the predicate
170.     * @param tableName
171.     * @param predicate
172.     */
173.    void deleteMatchingRows(String tableName, IPredicate predicate);
174.
175.    /** Query for a row with the given ID (primary key).
176.     *
177.     * @param tableName The name of the table to query
178.     * @param rowKey The primary key of the row
179.     * @return The result set containing the row with the given ID
180.     */
181.    IResultSet getRow(String tableName, Object rowKey);
182.
183.    /**
184.     * Set exception handler to use for asynchronous operations.
185.     * @param exceptionHandler
186.     */
187.    void setExceptionHandler(IStorageExceptionHandler exceptionHandler);
```

```
188.
189.    /**
190.     * Asynchronous variant of executeQuery.
191.     *
192.     * @param query
193.     * @return
194.     */
195.    public Future<IResultSet> executeQueryAsync(final IQuery query);
196.
197.    /**
198.     * Asynchronous variant of executeQuery.
199.     *
200.     * @param tableName
201.     * @param columnNames
202.     * @param predicate
203.     * @param ordering
204.     * @return
205.     */
206.    public Future<IResultSet> executeQueryAsync(final String tableName,
207.            final String[] columnNames,  final IPredicate predicate,
208.            final RowOrdering ordering);
209.
210.    /**
211.     * Asynchronous variant of executeQuery
212.     *
213.     * @param tableName
214.     * @param columnNames
215.     * @param predicate
216.     * @param ordering
217.     * @param rowMapper
218.     * @return
219.     */
220.    public Future<Object[]> executeQueryAsync(final String tableName,
221.            final String[] columnNames,  final IPredicate predicate,
222.            final RowOrdering ordering, final IRowMapper rowMapper);
223.
224.    /**
225.     * Asynchronous variant of insertRow.
226.     *
227.     * @param tableName
228.     * @param values
229.     * @return
230.     */
231.    public Future<?> insertRowAsync(final String tableName, final Map<String,Object> values);
232.
233.    /**
234.     * Asynchronous variant of updateRows
235.     * @param tableName
236.     * @param rows
237.     */
238.    public Future<?> updateRowsAsync(final String tableName, final List<Map<String,Object>> rows);
239.
240.    /**
241.     * Asynchronous variant of updateMatchingRows
242.     *
243.     * @param tableName
244.     * @param predicate
245.     * @param values
246.     * @return
247.     */
248.    public Future<?> updateMatchingRowsAsync(final String tableName, final IPredicate predicate,
249.            final Map<String,Object> values);
250.
251.    /**
```

```
252.  * Asynchronous variant of updateRow
253.  *
254.  * @param tableName
255.  * @param rowKey
256.  * @param values
257.  * @return
258.  */
259. public Future<?> updateRowAsync(final String tableName, final Object rowKey,
260.         final Map<String,Object> values);
261.
262. /**
263.  * Asynchronous version of updateRow
264.  *
265.  * @param tableName
266.  * @param values
267.  * @return
268.  */
269. public Future<?> updateRowAsync(final String tableName, final Map<String,Object> values);
270.
271. /**
272.  * Asynchronous version of deleteRow
273.  *
274.  * @param tableName
275.  * @param rowKey
276.  * @return
277.  */
278. public Future<?> deleteRowAsync(final String tableName, final Object rowKey);
279.
280. /**
281.  * Asynchronous version of deleteRows
282.  *
283.  * @param tableName
284.  * @param rowKeys
285.  * @return
286.  */
287. public Future<?> deleteRowsAsync(final String tableName, final Set<Object> rowKeys);
288.
289. /**
290.  * Asynchronous version of deleteRows
291.  *
292.  * @param tableName
293.  * @param predicate
294.  * @return
295.  */
296. public Future<?> deleteMatchingRowsAsync(final String tableName, final IPredicate predicate);
297.
298. /**
299.  * Asynchronous version of getRow
300.  *
301.  * @param tableName
302.  * @param rowKey
303.  * @return
304.  */
305. public Future<?> getRowAsync(final String tableName, final Object rowKey);
306.
307. /**
308.  * Asynchronous version of save
309.  *
310.  * @param resultSet
311.  * @return
312.  */
313. public Future<?> saveAsync(final IResultSet resultSet);
314.
315. /** Add a listener to the specified table. The listener is called
```

```
316.      * when any modifications are made to the table. You can add the same
317.      * listener instance to multiple tables, since the table name is
318.      * included as a parameter in the listener methods.
319.      * @param tableName The name of the table to listen for modifications
320.      * @param listener The listener instance to call
321.      */
322.     public void addListener(String tableName, IStorageSourceListener listener);
323.
324.     /** Remove a listener from the specified table. The listener should
325.      * have been previously added to the table with addListener.
326.      * @param tableName The name of the table with the listener
327.      * @param listener The previously installed listener instance
328.      */
329.     public void removeListener(String tableName, IStorageSourceListener listener);
330.
331.     /** This is logically a private method and should not be called by
332.      * clients of this interface.
333.      * @param notifications the notifications to dispatch
334.      */
335.     public void notifyListeners(List<StorageSourceNotification> notifications);
336. }
```