Securing network applications in software defined networking

Yuchia Tseng

Paris Descartes University

Doctor School EDITE

Securing Network Applications in Software Defined Networks

Yuchia Tseng

A thesis presented for the degree of
Doctor of Philosophy

President:	**Ahmed Serhrouchni,** *Professor at Télécom ParisTech*
Reviewers:	**Abdelhamid Mellouk,** *Professor at University Paris-Est Créteil*
	Abderrezak Rachedi, *Associate Professor (HdR) at University Paris-Est Marne-la-Vallée*
Examiners:	**Hossam Afifi,** *Professor at Télécom SudParis*
	Makhlouf Hadji, *Senior Researcher (HdR) at IRT SystemX*
	Samia Bouzefrane, *Associate Professor (HdR) at CNAM*
Supervisor:	**Farid Naït-Abdesselam,** *Professor at Paris Descartes University*
Co-Supervisor:	**Zonghua Zhang,** *Associate Professor (HdR) at IMT Lille Douai*

Acknowledgements

I am extremely thankful to my supervisor, Farid Naït-Abdessalem, for his guidance, encouragement and patience during the past years of my studies. He has always showed an open-minded spirit by giving me the liberty to pursue and explore new ideas I found most interesting in my research. He was also very supportive when I requested him to work on security problems of the new emerging Software-Defined Networking architecture, although it was not his priority in research when I started. Since then, I can guess that he has changed his mind. I am very grateful to him also for all the advices and suggestions towards the development of a good research and for showing me the way to investigate well-designed solutions and obtain results of better quality. I have been extremely lucky to have a supervisor who cared much about my work and responded to my queries so promptly. Working with him throughout my PhD journey was one of the most memorable experiences of my life.

Zonghua Zhang, the co-supervisor of my thesis, opened up the world of computer networking to me and taught me how to conduct a top-notch research. In addition to always being available and providing detailed help whenever I needed, he also offered me a lot of opportunities to attend conferences and introduced me to many world-class researchers in this field. His passion for research and rigorous research approach have influenced me immensely.

I would also like to thank the members of my thesis committee: Prof. Ahmed Serhrouchni, Prof. Hossam Afifi, Dr. Makhlouf Hadji, and Prof. Samia Bouzefrane for their acceptance to be part of the committee and evaluate my work. My particular gratitude goes also to Prof. Abdelhamid Mellouk and Prof. Abderrezak Rachedi for having accepted to review my manuscript and provide insightful comments.

Special thanks to Ruan He and Montida Pattaranantakul for their guidance on the *Controller DAC* project. They supported me to continue exploring this idea and brainstormed with me to develop this small idea into a working system with the current SDN controller OpenDaylight. I would also like to express my thanks to Ruby Krishnaswamy for her help on the *SENAD* project. Her sharp intellect and rich experience of the system gave me a truly enjoyable and rewarding experience.

I enjoyed my time at Paris Descartes University during my master and PhD. I thank the current and previous members of the laboratory LIPADE: Ahmed Mehaoua, Ajmal Sawand, Chafia Iken, Chin Nao Wang, Hassine Moungla, Jocelyne Elias, Lilia Idir, Medina Hadjem, Nicole Vincent, Osman Salem, Pavlos Moraitis, Rongrong Zhang, Themis Palpanas, and Xin Huang.

My sincere thanks also goes to my manager Gilles Desoblin at IRT-SystemX, who provided me the opportunity to join their team and who also encouraged me to finish my PhD. Without this valuable support, this thesis would be incomplete. Besides that, I would like to thank my colleagues at IRT-SystemX for broadening my research field with optimization: Makhlouf Hadji and Niezi Mharsi, whose encouragements have never stopped to finish my thesis. I thank our partners from Orange Labs and Nokia Bell Labs for the stimulating discussions: Bela Berde, Drissa Houatra, Gopalasingham Aravinthan, Ilhem Fajjari, Lionel Natarianni, and Sofiane Imadali.

To Jane, Joshua Leung, Joshua Wu, Neil, Tiffany Hua, and Winnie, a heartfelt thank you, for giving me enormous help on improving my writing skills. A huge thank you to all the friends in the church, for their unceasing prayers and boundless tolerance. To my brothers, for their support throughout this endeavor. To my lovely wife, my "reine de neige" Guoguo and my "petite grenouille" Maimai, for their sweetest company in my life. Above all, to my parents, Huiyen Tseng and Hsiutuan Lin, for their enduring love and for teaching me to value education.

Abstract

The rapid development and convergence of computing technologies and communications create the need to connect diverse devices with different operating systems and protocols. This resulted in numerous challenges to provide seamless integration of a large amount of heterogeneous physical devices or entities. Hence, Software-defined Networks (SDN), as an emerging paradigm, has the potential to revolutionize the legacy network management and accelerate the network innovation by centralizing the control and visibility over the network. However, security issues remain a significant concern and impede SDN from being widely adopted.

To identity the threats that inherent to SDN, we conducted a deep analysis in 3 dimensions to evaluate the security of the proposed architecture. In this analysis, we summarized 9 security principles for the SDN controller and checked the security of the current well-known SDN controllers with those principles. We found that the SDN controllers, namely ONOS and OpenContrail, are relatively two more secure controllers according to our conducted methodology. We also found the urgent need to integrate the mechanisms such as connection verification, application-based access control, and data-to-control traffic control for securely implementing a SDN controller. In this thesis, we focus on the app-to-control threats, which could be partially mitigated by the application-based access control. As the malicious network application can be injected to the SDN controller through external APIs, i.e., RESTful APIs, or internal APIs, including OSGi bundles, Java APIs, Python APIs etc. In this thesis, we discuss how to protect the SDN controller against the malicious operations caused by the network application injection both through the external APIs and the internal APIs.

We proposed a security-enhancing layer (SE-layer) to protect the interaction between the control plane and the application plane in an efficient way with the fine-grained access control, especially hardening the SDN controller against the attacks from the external APIs. This SE-layer is implemented in the RESTful-based northbound interfaces in the SDN controller and hence it is controller-independent for working with most popular controllers, such as OpenDaylight, ONOS, Floodlight, Ryu and POX, with low deployment complexity. No modifications of the source codes are required in their implementations while the overall security of the SDN controller is enhanced. Our developed prototype I, *Controller SEPA*, protects well the SDN controller with network application authentication, authorization, application isolation, and information shielding with negligible latency from less than 0.1% to 0.3% for protecting SDN controller against the attacks via external APIs, i.e, RESTful APIs. We developed also the SE-layer prototype II, called *Controller DAC*, which makes dynamic the access control. *Controller DAC* can detect the API abuse from the external APIs by accounting the network application operation with latency less than 0.5%.

Thanks to this SE-layer, the overall security of the SDN controller is improved but with a latency of less than 0.5%. However, the SE-layer can isolate the network application to communicate the controller only through the RESTful APIs. However, the RESTful APIs is insufficient in the use cases which needs the real-time service to deliver the OpenFlow messages. Therefore, we proposed a security-enhancing architecture for securing the network application deployment through the internal APIs in SDN, with a new SDN architecture dubbed *SENAD*. In *SENAD*, we split the SDN controller in: (1) a data plane controller (DPC), and (2) an application plane controller (APC) and adopt the message bus system as the northbound interface instead of the RESTful APIs for providing the service to deliver the OpenFlow messages in real-time. The role of the DPC is dedicated for interpreting the network rules into OpenFlow entries and maintaining the communication between the control plane and the data plane. We then secure by design the APC, which plays the role as the runtime of the networks applications. In APC, the network applications are deployed in the secure environment including authentication, access control, resource isolation, and applications monitoring. We show that this approach can easily shield against any deny of service, caused by the resource exhaustion attack or the malicious

command injection. Furthermore, we also implemented a network application to detect an OpenFlow specific attack, called priority-bypassing attack, for evaluating the feasibility of the *SENAD* architecture. The evaluation of this architecture shows that a *packet_in* message takes around 5 ms to be delivered from the data plane to the network applications on the long range. Thanks to the SE-layer and *SENAD* architecture, the SDN controller can protect against the malicious operations from the network application through both the external APIs and the internal APIs.

Table of contents

List of figures

List of tables

Chapter 1

Introduction to SDN

As a result of various Internet services, computer networks has been recognized as a critical role in modern life over the past half century. The rapid development and convergence of computing technologies and communication create the need to connect diverse devices with different operating systems and protocols. The seamless integration of a large amount of heterogeneous physical devices or entities poses many challenges, and how to establish a secure, accurate, and reliable communication channel between these different service infrastructure involve many non-trivial issues. Especially, it is challenging to correctly and effectively implement management tasks for these networks. As the control plane is coupled with the data plane in today's network, the control plane on each device exchanges information with each other, decides how the packets should be processed on the device, and configures the data plane as depicting in Figure 1.1 [45]. Since the control plane is distributed on the devices, it does not have a global view of the network and cannot make good network-wide decisions. Operators spend tremendous effort and time on configuring network devices. Managing these networks to provide reliable and secure network services is a central problem for computer networking research. Therefore, Software-defined networks(SDN) as an emerging paradigm has the potential to revolutionize legacy network management by centralizing the control plane [143]. Figure 1.2 is the high-level view of the SDN architecture [45]. This thesis is intended to develop a suite of effective approaches to achieve a secure and reliable SDN architecture. We introduce first the SDN architecture in this chapter. In Chapter 2, we conduct a 3-dimensional analysis on the security of the SDN controller. By following this analysis, we found the urgent need to secure the data exchange between the control plane and the application plane. Hence, we propose a security-enhancing layer in Chapter 3 and secure the network application deployment in Chapter 4. The conclusion will be given in Chapter 5.

1.1 The architecture of Software-defined Networks

Software-defined networks(SDN) refers to a network architecture where the forwarding state in the data plane is managed by a remotely controlled plane decoupled from the former. 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. The basic components in the architecture SDN consists of three layers and three interfaces are as following.

1.1.1 The data plane

The data plane consists of the networking appliances which forwards the packets. The data plane enables data transfer to and from end hosts, handling multiple conversations through multiple protocols, and manages conversations with remote peers. SDN decouples the data and control planes and implements the control plane in software instead [21]. Examples of SDN/OpenFlow-enabled switch implementations include Switch Light, Open vSwitch, Pica8, Pantou, and XorPlus, etc [45].

1.1.2 The control plane

The control plane, namely the SDN controller, makes decisions about where traffic is sent. The control plane in the SDN architecture can therefore be seen as the "network brain". The main function of the controller should provide a centralized way to manage the networks and exchange the data between the data plane and the network application. Moving the control plane to software allows dynamic access and administration. A network administrator can shape traffic from a centralized control console without having to touch individual switches and change any network switch's rules or configuration [20]. It exists more than twenty SDN

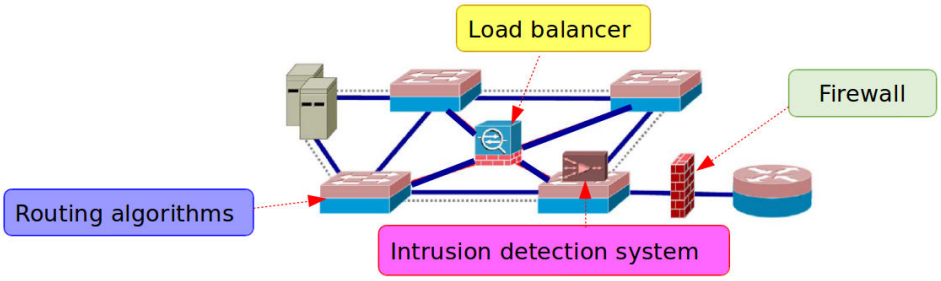


Fig. 1.1 Legacy network architecture

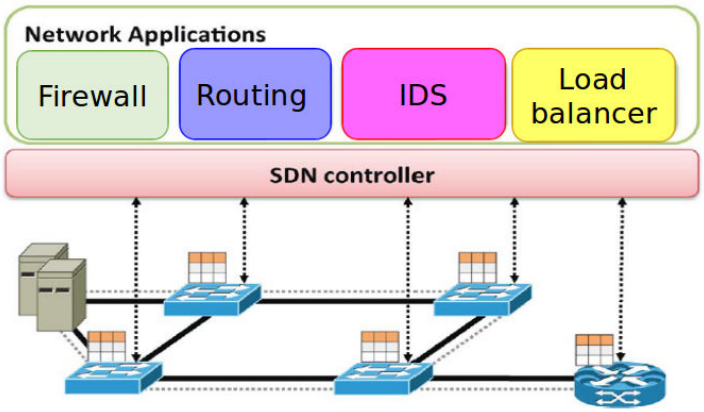


Fig. 1.2 SDN architecture

controllers in the market [125]. Here, we enumerate the open source and active SDN controllers which are related to our work in this thesis.

Floodlight. Project Floodlight is a popular SDN controller for the OpenFlow protocol written in Java. Floodlight supports two kinds of applications: Module applications and RESTful applications. Module applications are applications that are implemented in Java, and compiled together with the controller. These applications run as a part of the floodlight code, in the same process. REST applications are applications that use Floodlight's RESTful API to communicate with the controller. Using this API, information can be obtained from the controller, and route information can be sent to the controller. The RESTful API is more limited than the module application API in the sense of the interaction with the controller, but it can be used to avoid the malicious network application injection by decoupling from the controller via TCP/IP communication [35].

Ryu. Ryu is a highly modular, small SDN controller written in Python. The core of Ryu is smaller than other controllers. Ryu provides software components with well defined API that make it easy for developers to create new network management and control applications. Ryu supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config [94]. This controller is highly recommended after the analysis based on Analytic Hierarchy Process (AHP) proposed by [111].

OpenDaylight. The core of the OpenDaylight controller is the Model-Driven Service Abstraction Layer (MD-SAL). In OpenDaylight, underlying network devices and network applications are all represented as objects, or models, whose interactions are processed within the SAL. This is achieved by the YANG models which provides generalized descriptions of a device or application's capabilities without requiring either to know the specific implementation details of the other. Even more, OpenDaylight includes support for a broad set of protocols in SDN platform – OpenFlow, OVSDB, NETCONF, BGP and many more – that improve

programmability of modern networks and solve a range of user needs. OpenDaylight is based on Kafka container, which enables the module can be turn on and off dynamically without interfering other modules [101]

ONOS. ONOS is a native distributed SDN controllers for forming the controller cluster. The controller cluster provide high controller deployment scalability and avoid the one-single-point-of-failure problem in SDN. An ONOS cluster can be scaled as needed to provide the amount of control plane capacity needed. If more switches are added to the network, new instances of ONOS can be easily added into the cluster without service interruption. Like OpenDaylight, ONOS also uses the Apache Karaf framework [95].

OpenContrail. OpenContrail is built using standards-based protocols and provides the necessary components for network virtualization. It simplifies especially the creation and management of virtual networks. In addition, it focuses on addressing the challenges of large-scale managed environments, including multi-tenancy, network segmentation, network and access control etc [73].

Other controllers. The most popular are NOX [86] [93] and POX[15]. NOX is the original controller for OpenFlow, developed alongside the OpenFlow protocol. As such, it attracted many research attention. POX is a sibling of NOX, written in Python, meant to be more modern. Open Mul is an OpenFlow controller written in C, designed for performance and reliability. They claim this controller is ideal for "mission-critical" environments [78].

1.1.3 The application plane

The application plane is the set of applications that leverage the functions offered by the northbound interface to implement network control and operation logic. This includes applications such as routing, firewall, ACL, load balancers, monitoring, intrusion detection system (IDS), scan detector, DDoS attack mitigator and so forth (refer to Figure 1.2) [38] [113] [128] [131]. Essentially, a network application defines the policies, which are ultimately translated to southbound-specific instructions that program the behavior of the forwarding devices. For example, the IDS network application can trace network-wide traffic information, user migration, and packet payload, etc. If the IDS application recognizes malware traffic, it could automatically isolate those packets before they infect the network. However, the deployment of the network application still remains many security issues.

1.1.4 The southbound interfaces

The southbound interfaces(or APIs) in the SDN architecture are used to communicate between the SDN controller and the network devices on the data plane. The southbound interfaces can

be open or proprietary. It facilitates efficient control over the network and enables the SDN controller to dynamically make changes according to real-time demands and needs. OpenFlow, which was developed by the Open Networking Foundation (ONF), is the probably most well-known southbound interface. It is an industry standard that defines the way the SDN controller interacts with the forwarding plane to make adjustments to the network. With OpenFlow, entries can be added and removed to the internal flow-table of switches and potentially routers to make the network more responsive to real-time traffic demands [124]. While OpenFlow is not the only one available or in development, there are other southbound interface proposals such as ForCES, Open vSwitch Database (OVSDB), POF, OpFlex, OpenState, revised open-flow library (ROFL), hardware abstraction layer (HAL), and programmable abstraction of data path (PAD) [45]. As OpenFlow is the most well-known and popular southbound interface, we discuss the OpenFlow-based SDN in this thesis.

1.1.5 The northbound interfaces

The northbound interfaces (APIs) are used to communicate between the SDN controller and the network applications running over the network. The northbound interfaces can be used to facilitate innovation and enable automation of the network to align with the needs of different applications by benefiting the northbound programmability. However, the northbound interfaces are arguably the most critical APIs in the SDN environment, since the value of SDN is tied to the innovative applications. The northbound interfaces must support a wide variety of applications because one size will likely not fit all. For example, network applications could be optimized via the northbound interface to develop load balancers, firewalls or other network services. The northbound interfaces are also used to integrate the SDN controller with the NFV orchestrator or cloud system like Puppet, Chef, Ansible and OpenStack. This is why the northbound interfaces are currently the most nebulous component in a SDN environment. A variety of possible interfaces exists for SDN northbound interfaces like RESTful APIs, Java APIs, Python APIs or message queue etc [123].

1.1.6 The westbound/eastbound interfaces

The boundary of westbound/eastbound interfaces is not yet definite. In general, these two interfaces are used for enabling the communication between different SDN or non-SDN domains. The westbound interface serves as an information conduit between SDN control planes and different network domains driven by different SDN controllers. It allows the exchange of network state information to influence routing decisions of each controller, but at the same time enables the seamless setup of network flows across multiple domains. The

eastbound interface communicates the control planes with the non-SDN domains. In this way, both domains should ideally appear to be fully compatible to each other. For example, the SDN domain should be able to use the routing protocol deployed between non-SDN domains [82].

1.2 Conclusion

The seamless integration of a large amount of heterogeneous physical devices or entities poses many challenges, and how to establish a secure, accurate, and reliable communication channel between these different service infrastructures involve many non-trivial issues. Fortunately, SDN as an emerging paradigm revolutionizes legacy network management by providing a centralized way to control the network. The SDN architecture includes three main layers, which are the data plane, the control plane, and the application plane, and three interfaces(or APIs) for the communication between different layers, which are the southbound interfaces, the northbound interfaces, and the westbound/eastbound interfaces. The data plane consists of the network devices for forwarding the network packets according to the rule flows enforced from the control plane. The control plane, namely the SDN controller, functions as a network operating system, can load different operation logics as the network application for enabling network intelligent. The communication between the data plane and the control plane is the southbound interface. The famous one is OpenFlow protocol. The interface between the control plane and the application are called northbound interface for enabling the SDN controller become programmable. The westbound/eastbound interfaces are not yet definite, but in general, these two interfaces are used for communicating different SDN or non-SDN domains. However, the security issues remain a major concern for SDN to being widely deployed. Hence, we intend to develop effective approaches for achieving a secure and reliable environment for SDN in this thesis. In the following chapters, we will conduct a security analysis for exploiting the vulnerabilities of the SDN controller. Based on this analysis, we propose a security-enhancing layer(SE-layer) to secure the interaction between the network application and the controller with low deployment complexity and low latency. A novel SDN architecture, *SENAD*, for securing network application deployment is also introduced. This approach can protect the SDN controller effectively against the DoS attack caused by resource exhaustion attack, code injection and command injection.

Chapter 2

3D Analysis on the Security of SDN Controller

In this chapter, we conduct a comprehensive 3-dimensional security analysis of a SDN controller and resume the nine security principles identified in this chapter. Finally, we evaluate the security implementations with the security principles on the current SDN controllers.

2.1 Motivation and background

SDN is as susceptible to attacks as traditionally managed networks. A long and continuous series of attacks have revealed many areas of network vulnerability. The SDN controller is key to any software-defined network and offers attackers a target not present in earlier network technologies [117]. Clearly, any successful attack on the controller will halt or disrupt network operations [44]. To address this problem, some researchers have proposed to harden the SDN control plane. For example, FortNOX [109] hardens the kernel of the SDN controller (NOX) to avoid malicious rule injection from the application plane. Extending FortNOX, SE-Floodlight [110] enhances the security of the overall control plane through authentication, role authorization and network application operation auditing. Rosemary [130] isolates the network application from the control plane via inter-process communication (IPC), which enables the SDN controller to be immune from resource exhaustion attacks. Moreover, the distributed controllers have also been proposed for providing a more scalable control plane, such as ONOS [95], Onix [139], Kandoo [133] and the three levels controller architecture for IoT (Internet of Things)[48]. Moreover, a distributed architecture brings new challenges for data consistency between controllers [42]. However, the SDN controller remains vulnerable to many threats, such as data inconsistency, OpenFlow messages flooding, topology spoofing,

and malicious scanning, etc. Although a number of surveys on SDN/OpenFlow security also precede this analysis such as [45], [121], [75], [59], [122], [142], and [118], but none of them provides a comprehensive study on the security of the SDN controller. For example, [59] adopts STRIDE to evaluate the security of SDN. Conversely, [114] uses STRIDE to evaluate the security of the southbound protocol OpenFlow. [118] presents a survey of the hybrid SDN models by analyzing their techniques, inter-paradigm coexistence and interaction mechanisms. [99], proposed by Open Networking Foundation (ONF), includes a coarse-grained security analysis for the SDN controller with the STRIDE model. Here, we conduct a 3-dimensional, fine-grained and controller-specific security analysis. Based on [99], [100] ONF introduces 31 security requirements for the SDN controller without detailing the use cases. In our work, we summarize to nine security principles the identified requirements from the listed attacks. [98] proposes eight principles to secure SDN, specifically securing OpenFlow and the data plane of SDN. [111] uses a feature-based decision making template to compare and identify SDN controllers. [30] compares and evaluates the two controllers, OpenDaylight and ONOS, in particular their northbound interfaces, based on the use cases such as discovering the network, adding a new network function and changing an existing function. [148] employs a quantitative approach to analyze the security of the SDN controllers using threat/effort model. [127] is a framework to exploit the security of SDN, specifically the control plane. As none of the above surveys provides a specific survey for an SDN controller, we conduct here this first analysis following 3 dimensions and evaluate the security of multiple SDN controllers. This analysis is concluded in the Figure 2.3, showing that none of one single dimension can be used to cover alone all the known attacks on an SDN controller. For this reason, the 3D approach can provide a better and comprehensive analysis on the security of the SDN controller.

This chapter focuses on the points as follow:

- Conduct a specific security analysis on an SDN controller by covering 3 dimensions: (1) the components dimension, (2) the characteristics dimension and (3) the STRIDE model.
- Conclude with and summarize countermeasures in nine principles as a necessary instructions to secure SDN controllers from the specified known attacks.
- Evaluate the security of five open-source and active SDN controllers following the nine security principles.

2.2 First dimension

In this section, we discuss the vulnerabilities of the SDN controller in the first dimension, which are the essential components that a SDN controller is composed of. As a matter of

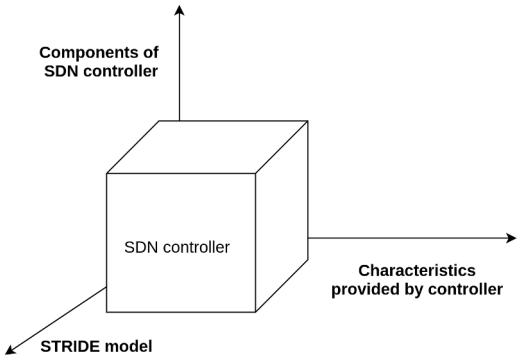


Fig. 2.1 3D approach of security analysis on SDN controller

fact, the actual implementations of SDN controllers are different from each other. Fortunately, most of the well-known SDN controllers, including OpenDaylight, ONOS, Floodlight, Ryu, OpenContrail, NOX and POX, are implemented based on a typical architecture composed of the key components, which are northbound interfaces (NBIs), built-in functions(i.e., core services or basic services), computing resources, eastbound/westbound interfaces and southbound interfaces (SBIs) as depicted in Figure 2.2 [126]. Every component will be discussed in detail in the following sections.

With this structure, we discuss the vulnerabilities of these components layer by layer.

2.2.1 Northbound interfaces vulnerabilities

The NBIs are application-programming interfaces (e.g., RESTful APIs or native Java APIs) in charge of the interaction between the application plane and the control plane. As the northbound interfaces in SDN are non-standardized, they are very diverse and are more challenging to secure. One approach is to use APIs such as general APIs (Java, C, or Python etc) [130] [43] [149] [136] [93] [15] [87], RESTful API [95] [101] [35] [1] [73] [51], RESTCONF [101], SDMN API [10], PANE API [31], and NVP NBAPI [139] etc. Another approach is to use programming languages such as FML [140], Frenetic [12], Nettle [26], Procera [27], Pyretic [40], NetKAT [36], NetCore [41], and other query-based languages [135] which specify the security constraints to the app-to-control access. The diversity of NBIs increases the difficulty to protect against malicious network application injection.

Code injection

A poor access control of NBIs can allow an injected malicious network application, i.e., code injection, to call unauthorized functions. For example, ONOS allows users to dynamically and programmatically configure various ONOS components via one of the Northbound ser-

vices, called `ComponentConfigService` to enhance the configurability of ONOS. This feature is especially useful when users need to adjust the properties of various ONOS components (e.g., threshold values, switches to enable certain features) to fulfill different network requirements. Meanwhile, this feature introduces a new security threat to the control plane. A minor change in the configuration of an ONOS component may completely change the network behavior. Such tunable parameters should only be manipulated by the trusted entities via `ComponentConfigService`. Specifically, if the *PACKET_OUT_ONLY* parameter value is set to be true, the overall performance of the target network will become degraded [126]. Even worse, the poor control of NBIs can allow malicious applications to call system level command, e.g., via JNI, to read/write memory or to shut down the system [110].

Flow rules injection

Different from the data plane manipulation through the controller, which removes the flow rules kept on the data plane, flow rules injection abuses NBIs to flush out the precedent flow rules, even ones with higher priority. As most of the OF switches available on the market have limited content-addressable memory (TCAMs) [37] [88], if the NBIs do not constrain the operations of the network application, a malicious network application can over-insert through the controller. The new coming flow entries will flush out and replace the existing ones, without considering the priority of flow entries (i.e., the flow entries with lower priority can replace the flow entries with higher priority) on data plane [145].

Flow rule manipulation

If the NBI is not secured, API can be abused to compromise the SDN controller. The malicious network application can request to insert a large number of rules to slow down the controller responding time. Conversely, a malicious application can also frequently delete the rules in a switch if there are no constraints to the network application, so that every time when a new packet comes, it would be sent to the controller to request for a rule to apply. This attack can affect the performance of both the data plane and the control plane by consuming the controller's resources and degrading the overall performance [70]. In addition to the attacks mentioned beforehand, as the control plane is decoupled from the data plane, data leakage can happen on the NBIs. This problem will be discussed in section 2.3.5.

2.2.2 Computing resources vulnerabilities

Basic computing resources include computing capacity, memory, storage, and networking resources. In a monolithic SDN controller implementation (e.g. RYU, POX, NOX), the

network applications are compiled and then run as part of the controller module. While some of the controllers like OpenDaylight and ONOS allow instantiation of network applications at run-time without restarting the controller module by implementing the OSGi bundles, network applications still share the same computing resources. These resources, namely memory, CPU and networking, make the network applications inseparable from controller though the internal APIs. In this case, a malicious network application can damage the SDN controller easily.

Resource Exhaustion

A malicious network application can create multiple memory objects, large number of threads, an infinite loop or a non-stop growing linked list to run out the resources of the controller's host. These malicious operations can use up to all available system memory and all available CPU resources. Controllers, such as NOX, Beacon, Floodlight and OpenDaylight, do not limit memory allocations by its applications, often resulting in the controller crashing with an out of memory error [130] [70]. In this way, the malicious application may use up all available system resources and affect the performance of the other applications or the controller itself, even causing a system-level DoS.

Internal storage data tampering

As a controller owns built-in storage to keep statistics record, manages flow rules or handles packets, the storage record may be tampered once the NBI does not constrain the operation of the network application. If a malicious application can access the controller's data storage or internal memory, the abuse of such trust could lead to various types of attacks impacting the entire network. For example, the *packet_in* counter value is kept in the controller's internal storage for the usage of DoS detector or traffic monitoring. However, a malicious application can clear *packet_in* counter in the internal storage to confuse the DoS detector application. The controller also contains the network link information and flow rules in data storage, once the app can modify these values, the topology and flow rules will be tampered. Another example is to tamper the topology related information stored in memory. As Floodlight contains network link information, when an application accesses the Floodlight controller, it can modify the values in the data structure (i.e., network link information), which can easily confuse other network applications [130].

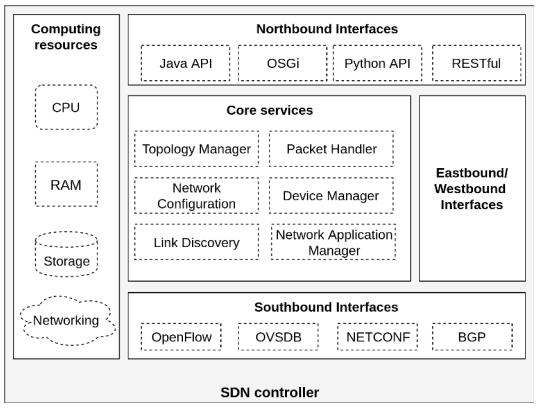


Fig. 2.2 The essential components of the SDN controller

2.2.3 Core services vulnerabilities

Built-in functions work as core services in a SDN controller, providing basic services to controller, like network topology information updating, flow rules interpretation, packet handler, network configuration and coordinating events among network applications, etc.

Core services manipulation

As core services of the SDN controller share information with all the network applications to provide data consistency, they allow malicious applications to manipulate this services chain. For example, applications running on OpenDaylight can easily register services or directly call them to use their network functions. Since OpenDaylight allows applications to change the services of other applications dynamically without constraint, an attacker can also leverage this ability to get information about the services used by a target application and then removes all these services so that the target application cannot use any of the services provided by the controller. For example, `ArpHandler` module is a default application that manages ARP requests and replies packets in OpenDaylight. A third party malicious application can get access to the registered services of the target application in the `Activator` class through `getServiceProperties()`. Then, the malicious application can call `unregister()` to delete the whole services of the target application. In this case, the malicious app can successfully unregister all the services of the `ArpHandler`. Consequently the `ArpHandler` cannot receive any messages from the controller. Since ARP packets play the role of an initiator for network communications, the network managed by the controller loses its functionality [126].

Topology spoofing

The topology monitoring modules provide insight into the global network visibility; unfortunately, a malicious end user can poison the network visibility provided by the SDN controller. Faked ARP packets can be used to spoof the network view of a SDN controller, which is similar to the ARP spoofing attack in traditional network. An attacker can generate the spoofing ARP packet to poison the network topology of SDN controller, which causes the black hole or DoS in data plane [83]. Similar to ARP spoofing, LLDP packets can also be used to poison the network topology in SDN [137]. For example, a fake topology attack can be launched on an SDN controller to poison its view of the network using detrimental *packet_in* messages sent by the switches. These malicious *packet_in* messages could be generated by distrusted switches themselves or by end hosts, which can send arbitrary LLDP messages spoofing connectivity across arbitrary network links between the switches. When the controller tries to route traffic over these phantom links, it results in packet loss, and if this link is on a critical path, it could even lead to a blackhole. With the poisoned network visibility, the upper-layer SDN controller may be totally misled, leading to serious hijacking, denial of service or man-in-the-middle attacks [83] [137]. The SDN controllers, including OpenDaylight, Floodlight, POX, and Maestro, can be affected due to ARP spoofing [83]. The difference between them is that the attacker could possess a botnet or compromise more than one OF switch in the data plane for generating the LLDP packets. This attack is OpenFlow specific and cannot succeed in the traditional network [137].

Malicious device connection

The SDN controller also has a device management module to provide the devices information on the data plane, such as the OF switch hardware address to identify switches. In the case where two switches have the same identification information, the turning on of a second switch and its connection to the controller can terminate the controller's connection with the first switch, favoring a connection with the new one [8]. Hence, if an attacker forges the information of a switch, the creation and connection of this forged switch will deny the connection of the legitimate one.

Malicious flow attack (Network-wide DoS)

The SDN controller, as a receiver, needs to verify that packets are generated from the data plane in order to ensure the non-reputation of the flows. Unfortunately, since the SDN controller is separated from the data plane, the data plane will typically ask the controller to obtain flow rules when the data plane gets new network packets that it has no idea on how to handle. By

Component	Vulnerability
Northbound interface	Code injection[110] [126] Flow rule manipulation[70] Data leakage[119] Flow rule injection[145]
Computing resources	Resource exhaustion[130][70] Storage data tampering[130]
Core services	Core services manipulation[126] ARP spoofing[83] LLDP spoofing[83][137] Malicious device connection[8] Malicious flow attack[8][131][47]
Eastbound/Westbound interfaces	Faked controller connection[99]
Southbound interface	Malicious scanning[131] *packet_in* messages[83][131][55][129] $flow\_mod$ message manipulation [109][110][147] Man-in-the-middle[76]

Table 2.1 Vulnerabilities from the essential components of SDN controller

exploiting this key property, the crafted flow requests from the data plane can mislead controller to insert many useless flow rules in the data plane, thus making the data plane hard to store flow rules for normal network flows (i.e., data plane resource consumption). As the data plane is dump in SDN, the SDN controller should be secured from inserting these malicious rules [131] [47] [8]. If the compromised switch is on a critical path in the network, it may result in significant latency or packet drops. For example, the attacker can just repeatedly send exactly 1K flows from a host with arbitrary source addresses to ensure that flow rules never time out at the switch and the whole data plane will be congested. This attack works for Floodlight, POX and Maestro, which completely populate the TCAM (as they use source/destination IP pairs as keys). This causes them to exhibit high latencies (40 - 80 ms) for any new flow rule installation [83].

2.2.4 Eastbound and Westbound interfaces

East/westbound APIs are the interfaces required for the communication of the SDN controller with distributed controllers or heterogeneous networks controllers (e.g., Closed-Flow [116]). Currently, each controller implements its own east/westbound API [45]. For example, Onix defines the interfaces as data import/export functions [139]. The advanced message queuing

protocol (AMQP) [25] is used by DISCO [77]. ONOS adopts the Cassandra database(prototype 1) and CloudRAM (prototype 2) for synchronizing data consistency among distributed nodes [103]. Attacker can forge a controller and intercept the communication between the data plane and legitimate controller. Theoretically, the malicious connection can install special rules on all forwarding devices for its malicious purposes. Taking counter falsification as another example, an attacker can try to guess installed flow rules and, subsequently, forge packets to artificially increase the counter. Such attacks would be especially critical for billing and load balancing systems [99].

2.2.5 Southbound interfaces vulnerabilities

The southbound interfaces are in the charge of the communication between the data plane and the control plane. On the data plane, a mix of OpenFlow-enabled physical devices(e.g., HP 8200 ZL series [6], Pica8 3920 [18]) and virtual devices (e.g., Open vSwitch [34], vRouter [74]) can coexist. The southbound interfaces should allow the control platform to manage the underlying networking devices using different protocols, such as OpenFlow, NETCONF [61], PCEP [90], Border Gateway Protocol (BGP) [13], Open vSwitch Database Management Protocol (OVSDB)[66], SNMP [91], MPLS Transport Profile (MPLS-TP) [60] and Locator/ID Separation Protocol (LISP) [65], etc. For example, Onix supports both the OpenFlow and OVSDB protocols. OpenDaylight provides a service layer abstraction (SLA) that allows several southbound protocols to communicate with the controller. OpenDaylight was designed to support the different protocols including OpenFlow, OVSDB, NETCONF, PCEP, SNMP, BGP, and LISP [101]. OpenFlow is one of the most popular southbound protocols. In this paper we will discuss how the vulnerabilities of OpenFlow can be used to affect the SDN controller (another reason is that few works discuss the security issues due to the non-OpenFlow southbound interfaces).

packet_in message flooding

The *packet_in* message is a way for the switch to send a captured packet to the controller in OpenFlow. This might happen due to an explicit action as a result of a match asking for this behavior, from a miss in the match tables, or from a `ttl`(time to live) error. In this case, one of the major threats to the controller is Denial- of-Service (DoS) and Distributed Denial-of-Service (DDoS) attacks from the data plane due to the communication bottleneck between the data plane and the control plane, which an adversary could exploit by mounting a control plane saturation attack that disrupts network operations. Indeed, even scanning or DoS activity can produce more impacts in SDN than traditional networks. For instance, malicious hosts can

mutate hosts with different IPs and MACs or forges arbitrary packet data. When the flow comes, the OF switch will treat them as new flow and initiate a *packet_in* message to controller to ask as to which flow rule to apply. Subsequently, when the controller receives too many *packet_in* messages to process, it causes the normal flows to be dropped [55][83][129].

The IGMP packets can be used to generate *packet_in* messages because the SDN controller maintains multicast groups as multicast trees, each group has a unique multicast IP that is used by its members to send/receive messages. Any receivers who are interested in joining/leaving a particular group must send IGMP messages to the controller, which are then forwarded as *packet_in* messages for the maintenance of multicast groups. Malicious hosts can forge IGMP join/leave requests to multicast groups, ultimately leading to DoS for legitimate members existing on the data plane due to the bandwidth saturation of controller [83].

flow_mod message manipulation

flow_mod message is one of the main messages in OpenFlow, allowing the controller to modify the state of an OF switch. This message structure contains the match fields, timeout (idle/hard) priority and actions, etc. However, the priority attribute can bring in the priority-bypassing attack, which will be discussed in section 2.3.6. The action attribute generates the flow rule conflict, particularly the Rule Circuit problem [110] (Section 2.3.6). If the timeout attribute is not well configured, i.e., the expiration time is too long for the connection, the opportunity for the *packet_in* flooding attack is increased. In OpenFlow 1.0, the header fields is 12-tuple. In OpenFlow 1.4, the header fields became 44-tuple. This aggregation calls for more *packet_in* messages to query the flow rule from the control plane and augments the possibility for *packet_in* message flooding as well as increases the number of flow rules, saturating the storage [114]. Similar to the vulnerabilities of northbound interfaces, the man-in-the-middle attack can be used to target the SDN controller on the southbound interface, making TLS from OpenFlow 1.3 optional [76]. This will be discussed in section 2.3.5. The time to process the *packet_in* and *packet_out* messages can be used to fingerprint the SDN-based network for the furthermore attack.

2.3 Second dimension

The SDN controller enables networking innovation to become more rapid and agile, but also exposes new vulnerabilities to the attacker. In this section, we will discuss the characteristics provided by the SDN controller, which bring in new security issues to the network.

2.3.1 Open

The security of open source software is a key concern for organizations planning to implement it as part of their software stack, particularly the SDN controller, which plays a major role on the SDN architecture. In contrast, traditional networking appliances such as switches, routers and intrusion detection system (IDS) are vendor specific, which are a ”black-box” and provide limited ability for users to experiment their own networking services. On one hand, the current mainstream SDN controllers, e.g., OpenDaylight, ONOS, Floodlight, Ryu and NOX, are open sources and developers can develop network applications as middle-boxes that interact with the controller and OF switches for accelerating network innovation. On the other hand, this gives the attacker opportunity to exploit the bugs existing in SDN controller or network applications. [127] proposes an assessment framework, called DELTA, to automate and standardize the vulnerability identification process in SDNs. In its evaluation, DELTA reproduced 20 known attack scenarios, across diverse SDN controller environments, including OpenDaylight, ONOS and Floodlight, and also discovered seven novel SDN application mislead attacks. On the one hand, the opening of the SDN controller brings in more opportunities for networking innovations; on the other hand, it makes the SDN controller vulnerable to the attacker.

2.3.2 Programmable

Programmability is considered a major characteristic of SDN by opening northbound interfaces as APIs to support the third-party network application to ease the deployment of new networking functionalities. However, this makes SDN controller susceptible to the injection of the malicious third-party applications [126]. Differing from the NBI vulnerabilities discussed in section 2.2.1, we will emphasize on the vulnerabilities due to the co-existing of network application in this section.

Application conflict

Consider the basic challenge of implementing two or more network applications on the same controller. Suppose application 1 initiates a series of flow rule insertions designed to quarantine the flows to and from a local web server. Application 2, a load-balancing network application, redirects incoming flow requests to an available host within the local web server pool. Suppose the web server quarantined by application 1 subsequently becomes the preferred target for new connection flows, as this quarantined server is now the least loaded server in the pool. In this case, should application 1 or application 2 prevail [110]? Hence, Google’s OpenFlow-based B4 competes traffic engineering logic into a monolithic application, where the arbitration of

Characteristics	Vulnerability
Open	Vulnerability exploitation[126][127]
Programmable	Application conflict [110][132] Execution chain interrupting[126][127]
Off-the-shelf	Command injection[130][70] Buggy application affection[130][99][81] Zero-day attack[130][99]
Flow-based rule	Flow rule conflict[109][110][132][57][23] [134][50] Priority bypassing[147]
Decoupled	Man-in-the-middle[76] Data leakage[119]
Centralized	*packet_in* flooding DoS[83][131][55][129] Malicious flow DoS[8][131][47] Flow rule manipulation[70] Code injection(Controller-level DoS)[110][126] Command injection(System-level DoS)[130][70]

Table 2.2 Vulnerabilities of characteristics provided by SDN controller

conflicting flow rules occurs fully within the application [138]. Unfortunately, the monolithic design for a SDN controller are not extensible, reusable, secure, or reliably maintainable [130].

Execution chain interrupting

The control plane level DoS includes service chain interference between network applications and configuration errors. As the SDN controller is programmable, it needs to mediate the execution chain amongst the network applications. Service chain interference denotes interference by a malicious application. A chained execution of network applications may be interfered because a malicious application may participate in a service chain and drop control messages before other awaited applications. A malicious application may also fall in an infinite loop to stop the chained execution of applications [127]. For example, Floodlight does not guarantee the integrity of the message reception order of applications nor the control message. A malicious application may leverage such vulnerability to launch an attack against the controller itself or the network by removing the entire payload value of the message and hand the forged message over to the next application [126].

2.3.3 Off-the-shelf

Comparing to traditional network, SDN controller is hardware non-specific and can run on a general purpose operating system, Linux. Unfortunately, this makes the SDN controller prone to inherent bugs and vulnerable to its runtime, including the host operating system, compiler, shell, or buggy application [130]. In this case, the SDN controller might be compromised due to the zero-day attack.

Command injection

Generally speaking, a network application is only allowed to run on the SDN controller to exchange data. However, command injection denotes the execution of a system level command by a malicious application, which is different from code injection described in section 2.2.1 that executes functions provided by the controller. Command injection can execute a system command. For example, the malicious network application may execute `System.exit(0)` , a compiler-level command on JVM, to terminate the controller instance or sends a system-level command to shutdown OS [130] [70].

Buggy application affection

As the SDN controller runs on a general purpose OS, which may contain other buggy applications, vulnerabilities through the privileges of other applications (e.g., script injection, shellshock, rootkit, etc.) can be used to attack the controller. These coexisting applications on the OS provide a back door to attack directly the controller runtime [130] [81].

Zero-day attack

As the SDN controller runs on a general-purpose OS (technically Linux), the attacker can exploit to adversely affect computer programs and then further compromise the controller since the exploitable bug's existence was disclosed. The challenge to generating immunity for the SDN controller to the zero-day attack is that the patch to fix the bug is cannot be applied immediately. Once the patch to fix the problem cannot be applied immediately to the runtime of controller, the whole network risks to be compromised [130].

2.3.4 Centralized

The SDN controller centralizes the control over network to ease the handling of network situations dynamically. However, this amplifies the Denial-of-service (DoS) or Distributed Denial-of- service (DDoS) attack [55] [129] [9].Researchers have shown that the extensive

communication between the data and control plane can potentially result in a bottleneck for the whole system, a situation that is exacerbated when a single controller manages a set of OpenFlow switches. Moreover, since the installation of rules on the switches is driven by the traffic generated from network users, an attacker can exploit this behavior to attack the control plane by flooding an OpenFlow switch with a large number of unique flows [84]. Even worse, the control plane saturation attack can be performed through a variety of network protocols, including TCP, UDP ICMP, or IGMP [55] [129] [85] [54].

As we discussed beforehand, the packet flooding can be launched due to OF messages from a forged host packet or a IGMP (Section 2.2.5), data plane malicious flow (Section 2.2.3), or an app-to-control flow rule manipulation (Section 2.2.1). Moreover, resource exhaustion attack(Section 2.2.2) and malicious command execution (Section 2.3.3 and Section 2.2.1) can stop the services provided by the controller.

2.3.5 Decoupled

The SDN controller splits the data plane and the application plane from the control plane, which makes the two interfaces, i.e., southbound interfaces and northbound interfaces, risk the man-in-the-middle attack. TLS, as one of the most common solutions, encrypts the communications of these two channels but increases the performance overhead and elevate the difficulty of implementation on the data plane [76]. However, the lack of implementing TLS in SDN allows a compromised application to leak out some sensitive information, such as network configuration and traffic statistics, and enables more advanced attacks by allowing attackers to conduct analysis on the flow tables, the control software and the models of OF switches [131].

Man-in-the-middle attack

In SDN, the man-in-the-middle attack can exist in the data plane (between OF switches), southbound API (between control plane and data plane), northbound API (between control plane and application plane) and eastbound/westbound API (between controllers). In regular networks, an attacker has to wait until an operator logs into each switch management interface using an insecure protocol (e.g., Telnet, SNMPv2) to capture credentials. However, the OpenFlow specification (up to v1.3.0) makes TLS optional. Thus there is the risk of a successful man-in-the-middle attack in an in-band SDN architecture. An attacker can immediately seize full control of any downstream switches and execute quasi-imperceptible eavesdropping attacks that would be difficult to detect [76].

Data leakage

The application plane is decoupled from the control plane to provide business logic to the SDN controller, which enables network management intelligence. However, the ability to manage the network state via the network application means that even the access to the network application should be protected. OpenDaylight, ONOS, and Ryu each provide a GUI for interacting with the controller. In the case of ONOS, there is no authentication/authorization required to access the GUI or to apply the REST calls. From inside the network, the IP address of the device hosting the controller is required. OpenDaylight provides some security, requiring a username/password to log in to the controller GUI, namely the DLux module. Ryu offers a basic topology viewer rather than a full GUI. The topology viewer provides a graphic illustration of the network topology, link status and flow entries. It is not secured. Despite the extensive security enhancements introduced in SE-Floodlight, the web user interface using REST is not access-controlled. For secure deployment, this controller interface should be protected to prevent data leakage [119].

2.3.6 Flow-based rule

In SDN, forwarding decisions are flow-based instead of destination-based. A flow rule is broadly defined by a set of packet field values acting as a match (filter) criterion, a set of actions (instructions), timeout, counter, and priority etc. But these attributes can be used to compromise SDN controllers, such as priority-bypassing attack [147] and flow rule conflict [109] [110] [132] [57].

Flow rule conflict

The SDN controller, as a network OS, mediates the flow rules, namely flow entries, from the application plane. One of its major challenges is resolving the conflict of flow rules generated by the different network applications or by malicious applications [109] [110] [132]. One example is that attackers can use the `set` action in the flow rule to compromise the network. The purpose of the `set` instruction in OpenFlow is similar to NAT or proxy in the traditional network, which modifies the packet's header information for the enablement of virtual tunnels between hosts. A virtual tunnel can be used to circumvent an inserted flow rule to prevent two hosts from establishing a connection. [109] and [110] have proven that this action can be used to mislead the SDN controller to send packets to unexpected destinations. The rules conflict is also studied in a number of works [132] [57] [23] [134] [50]. Interestingly, all reported threats and attacks affect all versions (1.0 to 1.3) of the OpenFlow specification.

Priority bypassing attack

Packet match fields in the flow rule are used for table lookups depending on the packet type, and typically include various packet header fields, such as Ethernet source address, IPv4 destination address or ingress port etc. In a priority-based mechanism, the packet is matched against the table and only the highest `priority` flow entry that matches the packet must be selected. The priority bypassing attack implies that a malicious application with the lowest priority predominates in rule making when higher priorities fail and is executed on the data plane [147]. [147] presents several attack models that cause higher priority rule-based policies to fail, including IP-passing, MAC-passing, VLANs-crossing, as well as DoS. With the increase of the header fields in OpenFlow, attackers have more opportunities to launch this attack.

2.4 Third dimension

In this section, the threat STRIDE model will be used to classify the threats to the SDN controller, which we have discussed in Section 2.2 and Section 2.3.

Spoofing

An attacker may introduce a malicious entity into the network and connect the SDN controller with a spoofing identity. The malicious entity can be (1) data-to-control faked packet, including spoofing ARP packet [83], LLDP packet [83][137] and IGMP packet [83], (2) an controller(Section 2.2.4), (3) a networking device (Section 2.2.3) or (4) a malicious network application injection (Section 2.3.2).

Tampering

The tampering problem is mainly due to the malicious network application injection. Any resource data, including log information, backup flow table contents, policy (Section 2.2.1), configuration data (Section 2.2.3), and network topology information (Section 2.2.2), can be modified in the controller. An attacker modifies the data so as to clear the attack log or to prepare further attacks on OpenFlow switches [99].

Repudiation

A repudiation attack is defined as one party participating in a transaction or communication and later claiming that the transaction or communication never took place. The controller may experience this kind of attack from applications or from upstream/downstream controllers when

STRIDE	Attack
Spoofing	ARP spoofing[83] LLDP spoofing[83][137] IGMP packet [83] Malicious device connection[8] Malicious network application injection [70][126] Faked controller connection[99]
Tampering	Flow rule manipulation[70] Core services manipulation[126] Internal storage tampering[130]
Repudiation	Execution chain interrupting[126][127] Application conflict [110][132] Flow rule conflict[109][110][132][57][23][134][50]
Information Disclosure	Vulnerability exploitation[126][127] Malicious scanning[131] Man-in-the-middle[76] Data leakage[119]
Denial of Service	*packet_in* flooding DoS[83][131][55][129] Code injection(Controller-level DoS)[110] [126] Command injection(System-level DoS)[130][70] Resource exhaustion[130][70] Malicious flow attack[8][131][47]
Elevation of Privileges	Priority-bypassing attack [147] Flow rule injection[145] Flow rule circuit[109][110] Buggy application affection[130][99][81] Zero-day attack[130][99]

Table 2.3 STRIDE model analysis of threats to SDN controller

there are controller hierarchies [99]. There are at least three needs for the SDN controller to provide the non-reputation assurance as the SDN controller suffers from execution chaining manipulation (Section 2.3.2), flow rule conflict (Section 2.3.6), and application conflict (Section 2.3.2), as these problems are due to the input conflict of co-existing network applications.

Information Disclosure

As the SDN controller is decoupled from the data plane and the application plane, the two interfaces, i.e., the southbound interfaces and the northbound interfaces, can leak out sensitive information, which we have discussed respectively in Section 2.3.5 and in Section 2.3.5. This enables more advanced attacks by allowing attackers to conduct analysis on network properties.

Denial of Service

The centralized control of the SDN controller makes it a profitable target to launch DoS/DDoS attacks, including packet-level, controller-level, system-level and network-wide level attacks. The packet-level DoS means the *packet_in* flooding attack (Section 2.2.5); the controller-level DoS is possible because a network application can maliciously manipulate built-in functions to block the normal services provided by the SDN controller (Section 2.2.1); the system-level includes the malicious system command injection to stop controller runtime (Section 2.3.3) and resource exhaustion attack (Section 2.2.2); the network-wide level DoS is the malicious flow which misleads the SDN controller to insert many useless flow rules and then statures the data plane, which was discussed in Section 2.2.3.

Elevation of Privileges

There are at least two ways that allow for attackers to increase privileges on the SDN controller. As the SDN controller can provide an API for third-party applications to be installed, a malicious application may make a policy that conflicts with the policy from administrators or security applications to bypass the administration policy or security policy [99], i.e., priority-bypassing attack (Section 2.3.6). Moreover, a malicious application can also abuse the API to insert numerous lower priority flow rules and then flush out the rule with higher priority [145], i.e., flow rule injection (Section 2.2.1). Flow rule circuit [109][110] can generate a flow rule circuit and bypass the flow rule with high priority. Last but not least, an attacker may utilize the vulnerabilities of the SDN controller software (e.g., software design error and software code error) to elevate the privileges of the controller. SDN controller software can be run on a server or a virtual machine so an attacker may utilize the vulnerabilities of the OS to elevate its

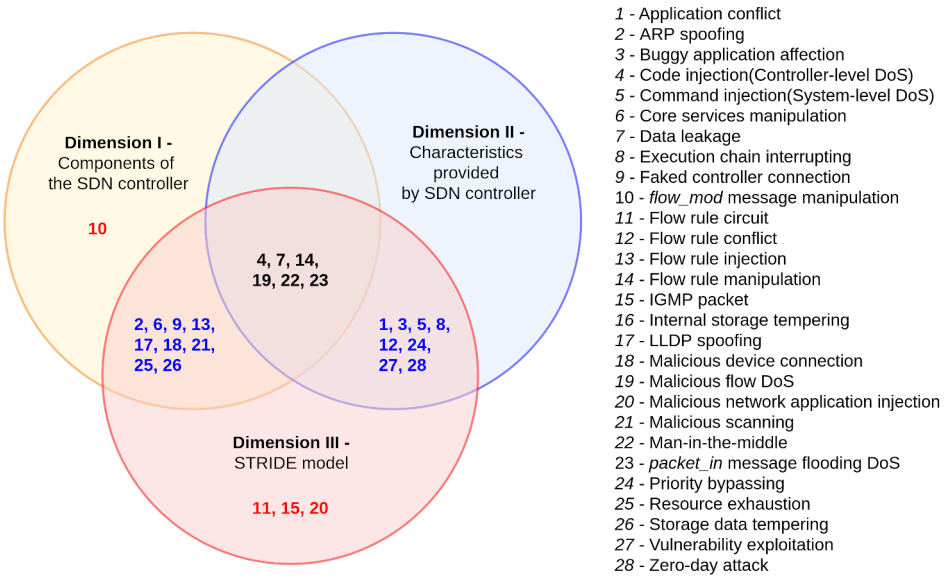


Fig. 2.3 Key observation of the security analysis of the 3D approach

privileges [99], i.e., buggy application affection (Section 2.3.3) and zero-day attack (Section 2.3.3).

We conclude the key observation of this 3-dimensional analysis in Figure 2.3. The attacks highlighted in black mean that they can be covered by any dimension analysis (4, 7, 14, 19, 22, and 23). The attacks highlighted in blue mean that they can be covered by two dimensions. For example, the attacks 2, 6, 9, 13, 17, 18, 21, 25, and 26 can be covered by both the Dimension I and the Dimension III. The attacks 1, 3, 5, 8, 12, 24, 27, 28 can be covered by both the Dimension II and the Dimension III. The attacks highlighted in red mean that the attacks can be covered by single dimension analysis. For example, the attacks 10 can be covered only by the Dimension I and the attacks 11, 15, and 20 which can be covered only by the Dimension III.

2.5 Security principles for the SDN controller

After conducting the security analysis, we resume nine the principles for enhancing the security of the SDN controller here.

2.5.1 Principle 1: Data-to-control flow identification and verification

The data-to-control flow should contain the information including host migration, switch status (switch hardware address/port updating), and packet-level information (packet processed by which network application). These information can used to detect topology poisoning, faked

device connection, malicious flow attack, and network application coexistence conflict. [137] proposes two approaches that can secure the link discovery procedure without the burden of manual effort, which are the authentication for LLDP packets and the verification for switch port property. The authentication for LLDP packets, TopoGuard adds a controller-signed TLV into the LLDP packet and check the signature when receiving the LLDP packets. The signature TLV is calculated over the semantics of the LLDP packet (i.e., DPID and Port number). In this case, the adversary can hardly manipulate the LLDP packets. However, this approach suffers from the fact that it fails to defend against the Link Fabrication attack in an LLDP relay/tunneling manner. An idea to mitigate the relay-based Link Fabrication is to verify of the switch port property by adding some extra tags to track the traffic coming from each switch port to decide which device is connected to the port for checking if any host resides inside the LLDP propagation. The verification of every OF switch connection can protect against malicious device connection [8]. The malicious flow which causes the network-wide DoS can be identified by abnormal host migration (forged host) [83] [47]. [132] suggests putting a flow tag to identify which network application is controlling the flow on the data plane and thus avoiding the conflict. Hence, identifying the data-to-control flow helps SDN controller to process the packet securely.

2.5.2 Principle 2: Entity authentication

Identity of all entities (e.g., switch, application, and the SDN controller) that access the SDN controller via interfaces must be authenticated by the SDN controller. Certificate and shared keys are common methods for identity verification [100]. Network application authentication as a first protection secures the SDN controller from fake controller connection and malicious network application injection (e.g., malicious operation like core services manipulation [126], flow rule manipulation[70], command injection [130][70], code injection [110][126], and execution chain interrupting [127][126]). We can authenticate the application with digital signatures as the fingerprint of application like the scheme used in [110] and [130]. [110] integrates the security extensions in the control layer, including the authentication service. Even more, a role-based source authentication module in [110] provides digital signature validation for each flow-rule insertion request to provisionally limit a candidate flow rules priority based on the application's operating role. [130] examines the authentication of applications based on its signed key (application authorization module). A public key is provided and all application developers are required to sign their applications with this key. The Certificate Management Service in OpenDaylight [102] is used to manage the keystores and certificates to provide the TLS communication. Apart from authentication, access control, which will be discussed in Section 2.5.3, is also necessary to reinforce the controller from malicious operations.

2.5.3 Principle 3: Security boundaries definition and access control

The SDN controller functions as a platform where several different network applications run with different purposes (e.g., load-balancer, firewall, intrusion detection system and Network address translation, etc). As the SDN controller opens a programmable API to thirty party applications, app-to-control threats such as data tampering, illegal function calling or flow rule manipulations can be used to attack the SDN controller. In this case, the definition of security dependencies and control access according to the permission boundaries is one of the efficient ways to protect the controller from dangerous network application operations. We classify the access control of SDN in three categories: role-based permission set approach, application-based permission set control and high-level SDN language approach.

Role-based permission control

The AAA module in OpenDaylight provides the role-based authorization service to differentiate the permission of administrators and users, as well as connect with the LDAP server to extract and to translate group information from LDAP into OpenDaylight roles [102]. Nonetheless, the application-based permission control is still an ongoing work. So far, only the read permission to protect the reading of the resources has been done in OpenDaylight [102].

Similarly, [109] and [110] propose a role-based authorization in a more fine-grained way. The role-based authentication recognizes by default three authorization roles among those agents that produce flow rule insertion requests. The first role is that of human administrators, whose rule insertion requests are assigned the highest priority. Second, security applications are assigned a separate authorization role. Flow insertion requests produced by security applications are assigned a flow rule priority below that of administrator-defined flow rules. Finally, non-security-related network applications are assigned the lowest priority. Roles are implemented through a digital signature scheme in which [109] is preconfigured with the public keys of various rule insertion sources. If a legacy network application does not sign its flow rules, then they are assigned the default role with the priority of a standard network application. The security mode of ONOS [96] adopts the role-based authorization proposed in [109] to secure the ONOS controller.

Application-based permission control

[144], [120], [70], and [56] propose different permission set to limit operations of network application. *PermOF* is another system that prevents a network application from invoking system calls without permission. Specifically, it proposes the use of 18 permission sets under four distinct categories (Read, Notification, Write, and System). The former three categories are

OpenFlow-related permissions for accessing the controller's resources, whereas the remaining category controls the access to the local resources provided by the SDN controller's host. This permission set is incorporated into the controller kernel, as the network application needs to access to controller, the controller requires the corresponding permissions. The category system is used as the network application calls the services from the controller's host. To achieve this, it adds an extra layer between host and network applications by modifying the source code of host OS [144]. This approach isolates control flow and data flow, and enables the controller to mediate all the applications' activities with the outside world. However, experimental evaluation on this proposed access control system is not yet available.

OperationCheckpoint [120] adopts the part of the permission set of *PermOF* in constraining the northbound API access and implements the permission set in the SDN controller Floodlight. In contrast, *PermOF* does not show any preliminary implementation results. *OperationCheckpoint* saves the unique application IDs mapped to the set of permissions granted to that application and the network administrator can add or remove application permissions by this unique ID. Its REST API can call for applications to query the controller and discover their assigned permissions as well as logs all unauthorized operation attempts to a log file for auditing. *OperationCheckpoint* is implemented in Floodlight for securing its methods to carry out the functions described by each of the permissions in the permission set. Unfortunately, the implementation of this permission set seems unfeasible for large-scale or complicated controller schemes, as it needs to scan all the related functions in the source codes and modify them.

Inspired by Android permission control, [70] is a permission system based on OF messages states and the actions. It classifies the OF messages in five state categories with five actions: (1) Sending OF *packet_out* message, (2) Sending $flowmod\_add$ message, (3) Sending $flowmod\_delete$ message, (4) Sending $flowmod\_modify$ message and (5) Sending *stats_request* message. In addition, there are two permissions that limit the network applications' access to the controllers' resources at the running time. The first one is DATABASE for controlling the access to internal storage and the second one is SYSTEMCALL for limiting to execute system calls. The permission set of the network application can be structured in a format similar to XML. The network administrators can use this system to select permissive operations, thereby allowing the necessary features for the network application to be specified. In contrast to other approaches, this approach provides the permission system with five states for each of the permissions, which helps application users to determine when the application uses each OpenFlow protocol. However, we believe that *OperationCheckpoint* can achieve the same goal by mapping its access control to the OpenFlow message.

In addition, [145] [39] [146] propose a solution for a controller-independent secure integration of external SDN applications. Unlike *PermOF* and *OperationCheckpoint*, these solutions are effective as long as the applications reside in the controller and do not address the security challenges that arise while applications are deployed externally to the controller.

[56] introduces a security framework called AEGIS to prevent controller APIs from being misused by malicious network applications. The usage of API calls in AEGIS is verified in real time by security access rules that are defined based on the relationships between applications and data in the SDN controller. AEGIS monitors network applications running on top of the controller's core modules, and intercepts the execution flow of applications through API hooking.

High-level SDN languages and controllers

These emerging proposals [83], [27], [28], [12], [40], [22], [58] embrace a variety of programming paradigms and introduce abstractions on all levels of network programming entities, such as packet, network topology and composition operation. [83] provides a policy language that enables administrators to specify validation checks on incremental flow graphs. Frenetic [12] designs different pieces of high-level language for state querying, packet forwarding and logic composition. Their following work, NetCore programming language [41], expands the query language and the packet-processing language, and provides formal semantics for this novel language. A parallel work, Procera [27], embraces a fully reactive programming paradigm and introduces a different abstraction of the network resources. Pyretic [40] proposes dramatically different programming models for packet, network topology and policy operation. All these SDN policy languages serve as vehicles to specify concrete network forwarding policies. SDNShield [22] permission language specifies the behavior privileges of SDN apps, and thus has different semantics and scope with the above SDN policy languages. Kinetic [58] looks for an intuitive domain specific language for implementing dynamic network policies. It exposes a language that allows operators to express network policy that maps directly to a model checker based on computation tree logic(CTL).

2.5.4 Principle 4: Network application resource control

Different from access control, the resource usage of the net- working app should be controlled to avoid control crashes due to the exhaustion attack from the network application. The control here includes resource monitoring, limitation and separation.

Resource isolation

One of the root causes of the fragility of the controllers is their tight coupling with applications. Thus, one effective approach to protecting the controller from the malicious behaviors is to isolate the network applications from the controller. `Yanc` [81], an SDN controller platform, adopts UNIX-like permission to prevent network applications from exposing the network configuration and stating it as a file system. In `yanc`, network applications are independent processes, provided by multiple sources. Applications benefit from the virtual file system (VFS) layer used to distribute the file system, as well as namespaces that are used to isolate applications with different views (e.g., slices).

[130] separates network application from the controller and each new application in Rosemary is invoked as a new process. Rosemary's network application connects to the SDN controller process through the IPC (inter-process communication). Moreover, Rosemary compartmentalizes the controller's kernel modules by separating not only the applications, but also the modules in the controller. Hence, this design increases the robustness of a controller because it only runs necessary services. The services in the Rosemary kernel communicate with each other through an IPC channel, and the implication is that if a service crashes, other services are immune to this crash. In contrast, controllers such as Floodlight, NOX, and POX implement all necessary functions in a single zone.

LegoSDN [32] uses AppVisor, a proxy service, to separate network applications from the controller. There are two parts in AppVisor: AppVisor Proxy in the control layer and AppVisor Stub in application layer. They communicate with each other via a RPC-like mechanism based-on UDP.

The network application in SE-Floodlight [110] should be instantiated as a separate process and ideally operated from an independent, non-privileged account. The connections to the server can be secured using standard SSL communication with either server or mutual authentication.

[19] proposes a strong software isolation mechanisms by partitioning the control plane both on the controller and on the switches. They separate the controller into Isolated Virtual Controllers (IVC) and Isolated Virtual Switches (IVS) along with per-flow, per-tenant, per-host, and per-application. This configuration ensures that a malicious flow or a network application cannot compromise the whole controller and limits the damage. However, it causes performance overhead due to the frequent creation and deletion of new virtual machines. An extreme configuration case would be making per-flow, per- app, and per-host into independent processes. Eventually, they adopt a per-tenant granularity and attempt to find a balance between security and performance.

Network application monitoring

An application that keeps allocating memory can consume all of the available memory in a host, thereby affecting other applications. Separation of the network application from the controller may not be sufficient in mitigating this effect. We need resource monitoring and limitation for each network application. For example, Rosemary limits the resources that each application can use and incorporates resource monitoring services that track and recognize the resource utilization of each application.

The hard limit specifies the value for which a network application cannot exceed in terms of resources. For example, if a hard limit value for the memory item is 2 GB, then the application is terminated if it attempts to allocate more than 2 GB of memory. The soft limit defines a recommended value that each application may nonetheless surpass. Violations of the soft limit result in an alert that is passed back to the application and reported to the network operator.

2.5.5 Principle 5: Provide high availability of service

The SDN controller centralizes the control over the network, exposing it to the risk of failure from one single point. This, however, can be mitigated by the use of distributed controllers, which logically centralizes and physically distributes the controllers. A robust recovery system can also help the controller to recover services from crash.

Distributed controllers

ONOS [103] as a native distributed controller adopts a master-slave mode to deploy the controller nodes. Once the master node crashes, it will randomly select another slave node to be upgraded as the master node to handle the control over the whole network. The distributed SDN control plane (DISCO) [77] is implemented on top of the Floodlight [35], which provides control plane functionalities to distributed networks by using Advanced Messaging Queuing Protocol (AMPQ) [25]. It is composed of two parts i.e., intra-domain and inter-domain. The intra-domain modules enable network monitoring and manage flow prioritization for the controller to compute paths of priority flows. The inter-domain part manages communication among controllers and uses the channels provided by the messenger agent to exchange network-wide information with other controllers. Another example is HyperFlow, which supports a distributed event-based control plane for OpenFlow. It logically centralizes but physically distributes the controllers.

In [141], the authors propose a coordination framework on top of the control plane to allow controllers working as controller clusters by supporting the communication among controllers via the communication library JGroups [33]. In this framework, the controllers elect a master

Principle	Threat to mitigate
Principle 1	Malicious flow DoS[8][131][47] ARP spoofing[83] LLDP spoofing[83][137] Malicious device connection[8] Application conflict [110][132]
Principle 2	Core services manipulation[126] Flow rule manipulation[70] Command injection[130][70] Code injection[110][126] Execution chain interrupting[126][127] Faked controller connection[99]
Principle 3	Core services manipulation[126] Flow rule manipulation[70] Command injection[130][70] Code injection[110][126] Execution chain interrupting[126][127] Storage data tampering[130]
Principle 4	Resource exhaustion[130][70]
Principle 5	*packet_in* flooding DoS[83][131][55][129] Code injection(Controller-level DoS)[110] [126] Command injection(System-level DoS)[130][70] Resource exhaustion[130][70] Malicious flow attack(DoS)[8][131][47]
Principle 6	Man-in-the-middle[76] Data leakage[119]
Principle 7	Application conflict [110][132] Flow rule conflict[109][110][132][57][23] [134][50] Execution chain interrupting[126][127]
Principle 8	*packet_in* flooding DoS[83][131][55][129]
Principle 9	Buggy application affection[130][99][81] Zero-day attack[130][99]

Table 2.4 Security principles for threats to SDN controllers

node in the controller cluster, which conducts and maintains the global controller-switch mapping in the network. Other nodes periodically monitor the master node, and if the master node is found to be inaccessible, it is immediately replaced by one of the other nodes. Thus, the proposed framework does not expose a single-point-of- failure. In doing so, it helps the SDN controller to avoid the challenge of one-single-point-of-failure [24]. It allows network operators to deploy multiple controllers, being capable of local decision making, in order to maximize controller scalability and minimize the flow-setup time. There are more examples of distributed controllers such as HP VAN SDN, PANE, and Fleet [136][31][7], while it is demonstrated in [52] that simply utilizing multiple controllers in SDNs cannot protect the network from single-point-of-failures. The reason is that the load of controllers carrying the load of the failed controller can exceed their capacity and hence worsen the situation, e.g., a cascade of controller failures.

In [11], a hybrid control model is proposed to combine the centralized control with some distributed control behavior. The security-enhancing architecture in this hyper control model can be found in [79], which aims to secure flow installations and prevents the malicious use of distributed control by applying device authentication (`Trust Manager`) and checking each end-to-end request. In addition, in order to handle Byzantine attacks and guarantee that each switch can correctly update its flow tables (even in the presence of compromised controllers that issue false instructions), [4] proposed a resilient mechanism to secure distributed controllers. One of the challenges of distributed controllers is to keep the data consistency among the nodes. For this, Onix, ONOS, and SMaRtLight provide strong data consistency among the distributed controllers during node updates in the network [95][139][51].

Another example design is proposed in Kandoo [133], which is essentially a hierarchical controller framework that consists of two or three layers. Kandoo adopts the bottom layer as a group of controllers with no interconnection, and no knowledge of the network-wide state. Controllers at this layer run local control applications (i.e., applications that can function using the state of a single switch). The top layer is a logic-centralized controller that maintains the network-wide state. Controllers at this layer handle most of the frequent events and effectively act as shields.

The hierarchical controllers in [49] and [48] are even proposed with more than two layers. For instance, in order to reduce the traffic load and to avoid a single point of failure at the controller node, [48] distributes the controllers role by introducing three levels of control: Principal Controller, Secondary Controller, the Local Controller. The Principal Controller plays the role to obtain a global view of the network infrastructure and the Local Controller acts locally by managing and relaying signaling messages from ordinary nodes to the Secondary Controller.

System recovery

The quick recovery mechanism of controller can be used to restore the services provided by controller once it crashes. [19] and [14] propose to recover the failed controller. In particular, [19] automatically uses the mechanism of rolling back to a pristine state. As in the existing SDN controller, once the controller is compromised, the damage remains over time. Rollback can help the controller to revert periodically the process to its pristine state. [14] introduces a fail-over scheme with per-link Bidirectional Forwarding Detection sessions and preconfigured primary and secondary paths computed by an OpenFlow controller. [17] uses the *CPRecovery* approach for the resilience of the failures in SDN. This mechanism is implemented as a network application. Unlike the distributed controllers' approach, where the controller would need to collect the information from the data plane, *CPRecovery* allows inter-communication between controllers using a `messenger` component to provide an interface for receiving and sending messages through SDN controllers.

2.5.6 Principle 6: Assuring data confidentiality and non-repudiation

To protect the controller from the information disclosure and data tampering, the SDN controller should keep data confidentiality and non-repudiation during data exchange and storage. Using on open standard is likely to bring benefits in both portability and interoperability, especially due to the diversity of the NBIs — a more generic approach to secure them is recommended. For example, transport layer used to exchange data between app-to-control, data-to-control and different controllers, is highly recommended to secure the communication channel with the TCP enhancement techniques which have been previously proposed for this purpose and are widely deployed [62] [67] [63] [68]. Therefore in the case of security functionalities such as encryption, authentication, and integrity, adoption of these existing techniques is recommended over development of a new transport layer solution in SDN. To avoid data leakage, the SDN controller can adopt these techniques to secure channels including the southbound, northbound, and eastbound/westbound interfaces. Authenticating the LLDP packets and host entity are one of proposed solutions in TopoGuard to prevent the man-in-the-middle attack in the data plane [137] [76]. These techniques can be adopted to protect the data in controller storage non-repudiation. Note that the use of legacy protocols or algorithms (e.g., MD5, Transport Layer Security (TLS) 1.0) are no longer recommended by standard organizations as they have been proven to be insecure, and should be avoided [98].

To identify and provide common compatibility and interoperability between different controllers, it is necessary to have a secure and standard east/westbound interfaces as SDNi [64] proposes. [82] suggests a finer distinction between eastbound and westbound horizontal

interfaces, referring to westbound interfaces as SDN-to-SDN protocols and controller APIs and eastbound interfaces as standard protocols for the communication with legacy network control planes (e.g., PCEP [90] and GMPLS [89][221]) [45].

2.5.7 Principle 7: Keeping operation records traceable and accountable

Data such as the network application operation record, flow entry, and resource usage should be kept traceable and accountable in log files, monitoring records, and flow rules identified with network application.

Network application operation accounting

The log record is useful for network troubleshooting non-reputation services as a malicious network application can modify flow tables or inject malicious flow rule, etc. [110] introduces an audit subsystem that traces all security-related events occurring in the control layer. With this auditing record, SE-Floodlight can report to network administrator the event time, message type, full message content, the application credential, the disposition (outcome) of the message, and optional message specific field attributes. Crash-Pad in LegoSDN [32] is an event-driven mechanism which classifies failures into (i) Fail-stop failures and (ii) Byzantine failures. Fail-stop failures are network application crashes and Byzantine failures are the output of network application when network policies are violated. Crash-Pad takes a snapshot of the state of a network application prior to its processing of an event and should a failure occur, it reverts back to this snapshot. This is unlike the OFRewind, which is a network-event-based audit system for recording and playing back SDN control-plane traffic [29]. OFRewind is able to replay these network events through topologies for troubleshooting or debugging network device and control plane anomalies.

App Zone of Rosemary [130] monitors flow rule enforcement operations initiated by an application to investigate the presence of any abnormal behavior. For example, an operation is considered abnormal if an application tries to enforce too many flow rules that cannot be inserted into the data plane. In this case, the flow rule enforcement module monitors this operation, and reports if it finds these abnormal behaviors. In the current state, it has a threshold-based detector, which generates an alert when it finds an application that tries to deliver more flow rules than a predefined threshold value. [146] and [145] record every request from network application to ease the accounting as attacks happen and detect flow rule conflicts.

Rule conflict resolution

There are several different approaches to mitigate the security issues caused by malicious flow rules injection and misconfigured or tampered flow rules. FlowTags [132], an extended SDN architecture in which middleboxes add `tags` to outgoing packets, provide the necessary causal context (e.g., source hosts or internal cache/miss state). These `tags` are used on switches and (other) middleboxes for systematic policy enforcement. The challenge here is that the FlowTags need to customize the switch by adding `tags` in the packet in order to identify which network applications work on it.

FortNOX and SE-Floodlight incorporate a live rule conflict detection engine in the controller kernel, which mediates all flow rules insertion requests. FortNOX presents a conflict analysis algorithm integrating in the controller's kernel; SE-Floodlight improves the computational performance of this conflict analysis algorithm (logarithmic vs. linear) and support for multi-switch deployments. The rule conflict analysis is performed using an algorithm called the Alias Set Rule Reduction, which detects rule contradictions, even in the presence of dynamic flow tunneling using `set` and `goto` actions. When such conflicts are detected, the controller can choose either to accept or reject the new rule, depending on whether the rule insertion requester is operating with a higher security authorization than that of the authors of the existing conflicting rules [109] [110].

FlowGuard extends HSA [108] for inline rule conflict detection and resolution in the context of firewalls in order to build more robust firewalls in SDN environments. Still, it does not deal with the problem of conflict resolution among competing network applications. FlowGuard operates on single switch environment and its algorithm does not handle inter-table dependencies [57]. FLOVER [134], a model checking system based on the Yices SMT solver existing on the application plane, verifies the instantiated flow rules which does not violate the network's predefined security policy.

NICE way [80], a network application, addresses the data inconsistency problem in SDN, especially those caused by events happening at different switches and end hosts or communication delays with the controller. It does so by using model checking to explore the state space of the entire system, which includes the controller, the switches, and the hosts.

2.5.8 Principle 8: Flow statistics and Rate limiting

Packets rate limiting control avoids flooding DoS attacks in the data-to-control flow. The SDN controller should define the life cycle for the data-to-control flow, e.g., shorten the flow idle connection timeout, and control the flow rate. SPHINX [83] detects OF messages DoS attacks, i.e., *packet_in*, on the SDN controller by observing flow-level metadata to compute

the rate of *packet_in* messages. It raises an alarm if *packet_in* messages throughput is above the administrator-specified threshold.

[129] proposes the system Avant-Guard, which introduces a data plane solution to reduce the amount of data-to-control plane interactions that arise during such attacks in the data plane. It implements a SYN proxy module and only exposes those flows that finish the TCP handshake. This switch-based solution introduces actuating triggers over the data plane's existing statistics collection services. These triggers are inserted by control layer applications to both register for asynchronous call backs and insert conditional flow rules that are only activated when a trigger condition is detected within the data plane's statistics module. Avant-Guard needs a customized switch, which allows for new functionality to be applied to every packet at high speeds, but also presents a significant deployment challenge requiring network operators to deploy new switches. On the one hand, Avant-Guard needs customized switches, on the other hand, this switch-based approach makes Avant-Guard controller-independent. This means that all SDN controllers can benefit from Avant-Guard to mitigate DoS attacks with little configuration effort. The connection migration component of Avant-Guard improves resilience against TCP SYN flood attacks better than other than protocols such as UDP or ICMP.

FloodGuard [55] introduces a protocol-independent defense framework for SDN networks, which contains two new modules: proactive flow rule analyzer and packet migration. To preserve network policy enforcement, proactive flow rule analyzer dynamically derives proactive flow rules by reasoning the runtime logic of the SDN controller and its applications. To protect the controller from being overloaded, packet migration temporarily caches the flooding packets and submits them to the controller using rate limit and round-robin scheduling. FloodGuard aims to defeat more generic saturation attacks in SDN, including UDP and ICMP, not only limited to only TCP protocol as in Avant-Guard.

Similar to Avant-Guard, the OFX framework [9] [71] is an extension module that contains security functions to protect the controller from data-to-control flood attack. However, unlike Avant-Guard, the OFX security module does not require implementation changes to any part of the OpenFlow stack and can be deployed on existing OF switches. In particular, the OFX framework consists of the components distributed on the data plane and control plane. The OFX library exists on the control plane by providing control applications an interface to handle all the modules related to controller-to-switch communication, including loading the module onto the switches. A developer can write in OFX modules to prevent flood attacks, such as push alerts, which allow a switch to signal to its controller whenever the packet- or byte-rate of a flow exceeds a threshold.

2.5.9 Principle 9: Best practice for securing the runtime of the controller

If a malware infects the host running a controller, it can abuse the controller quite easily. Such threats would be addressed through following best practices in host and network management, e.g., regular patch management, isolating the controller from other unnecessary applications, and fine-grained permission control. Although Rosemary spawns a new process for every network application, and uses IPC method to communicate between the controllers, it still cannot fully protect the controller from the malicious applications on the controller's host. The Security Enhanced Linux (SELinux), which provides the MAC mechanisms used to support and to ensure secure communication between processes, should also be activated [106]. A shim layer between the SDN controller and the OS can be used to protect the controller from malicious or buggy applications installed on the OS to illegally access the controller by modifying the OS's source code [144]. Moreover, in order to avoid zero-day attacks, the fewer programs on runtime with the SDN controller, the securer it will be. Last but not least, use defaults to make the controller secure. This includes default behavior, default algorithms, default key length, types of certificate, pre-defined access control policies, default password, closing unnecessary ports (services) and the data-to-control flow life cycle configuration (timeout setting), etc [98] [119].

2.6 Study on the security of controller implementation

We use the nine principles to analyze the security of five active and open source SDN controllers, which are OpenDaylight, ONOS, Ryu, Floodlight, and OpenContrail.

The controllers have been selected based on their design. ONOS and OpenDaylight are designed for scale-out i.e., multiple distributed controller instances. Floodlight has been designed as multiple-thread controller for enhancing performance, and Ryu is included and based on both security features and extensibility. OpenContrail is aimed at network virtualization initially [119].

All of them have the host tracing and link discovery modules to identify the host and switch connection. For instance, Ryu uses `switches.py` to identify OpenFlow Datapath ID (DPID) and port ID, `tracker.py` to trace host with MAC, IP and location(attached port). The `LinkDiscoveryManager.java` module and `LocationManagementApp.java` in Floodlight identify the switch DPID and Port ID but it adopts only MAC for tracing host. None of them verifies the legitimation of connection [137].

OpenDaylight and ONOS provide the network application authentication by the digital signature; in contrast, Floodlight and Ryu do not. The call of RESTful API in OpenContrail can be authenticated by auth token provided by OpenStack keystone [72]. OpenDaylight

AAA services are based on the Apache Shiro Java Security Framework, which performs authentication, authorization, cryptography, and session management for OpenDaylight [102]; ONOS security mode [96] extends FortNOX [109] RBAC (role-based access control) mode, which can be configured as *role* in the ONOS policy. The ONOS security mode can apply the access control features for the network application by specifying the permission-related information. Unfortunately, the policy cannot configure the request quota in ONOS security mode. OpenContrail also provides API-level RBAC relying on user credentials obtained from OpenStack keystone. Note that all of these controllers provide only RBAC mode, but the application-based access control remains a security issue to SDN control [144] [70] [120] [22].

OpenDaylight, ONOS and OpenContrail adopt the controller cluster to support high availability instead of disaster recovery. OpenDaylight and ONOS both can be configured as distributed mode [119]. The OpenContrail system provides an east-west interface (BGP) used to peer with other controllers for controller cluster configuration. From the point of view of disaster recovery, although none of SDN controllers amongst these five selected controllers have the ability for system recovery; but as OpenDaylight, ONOS and OpenContrail integrate with databased as pertinent data storage, which can provide a disaster recovery for these data. For example, OpenDaylight moves from SQLite to H2 as persistent data storage in Lithium distribution. The recover tool (`org.h2.tools.Recover`) can be used to extract the contents of a database file, even if the database is corrupted. RAMCloud database used by ONOS [103] provides a fast crash recovery promise [46]. Similarly, Cassandra, default database used in OpenContrail, can restore data from a snapshot when the table schema exists. In contrast, Ryu and Floodlight keep the data only in memory, which make it become difficult to restore the data from crash.

All these controllers run over Transport Layer Security (TLS) to provide authentication and confidentiality for securing the channel to connect with the data plane. The controllers including OpenDaylight, ONOS, Floodlight and OpenContrail can establish the identity of clients using TLS authentication to its northbound interfaces in [112] and [97]. In contrast, the northbound interfaces in Ryu do not yet adopt TLS to secure. The request of the northbound interfaces should be authenticated by password in OpenDaylight and ONOS. The API server of OpenContrail works in either authentication mode or non-authentication mode. In authentication mode, API server connects to authentication server to authenticate each request. Only requests from authenticated user are accepted. Unfortunately, Floodlight and Ryu do not provide any security mechanism to limit the requests of the northbound APIs, which should not be considered as a good practice in term of principle 9.

For the five controllers keep the log files for network application operation accounting [119]. For example, the Java-based controllers OpenDaylight, ONOS, and Floodlight uses `log4j` to

Principle		ODL (Lithium)	ONOS (Junco)	Ryu (3.5)	FL (1.2)	OC (R4.0)
1	Host/Switch identifying	✓	✓	✓	✓	✓
	Connection verification					
2	Switch authentication	✓	✓	✓	✓	✓
	App authentication	✓	✓			✓
3	RBAC	✓	✓			✓
	App-based access control		partial			
4	Resource control	partial (JVM)	partial (JVM)		partial (JVM)	
5	Distributed System recovery	✓ partial (H2)	✓ partial (RAMCloud)			✓ partial (Cassandra)
6	Southbound	✓	✓	✓	✓	✓
	Northbound	✓	✓		✓	✓
7	Operation accounting	✓	✓	✓	✓	✓
	Rule conflict resolution		✓			✓
8	Traffic control					
9	Secure default configuration	✓	✓			✓

Table 2.5 Security analysis of SDN controller implementation

record the log messages in different levels like INFO or ERROR report. OpenContrail uses a command-line utility to retrieve system log messages, object log messages, and trace messages.

Of the five controllers analyzed in this work, only ONOS and OpenContrail implement policy conflict resolution. In ONOS, the application describes its network requirements in the form of "intents" and ONOS translates these intents with respect to the network configuration, which is supported by a shared data store [119]. Data models also play a central role in the OpenContrail System. Its data model consists of a set of objects, their capabilities, and the relationships between them. The data model permits applications to express their intent in a declarative rather than an imperative manner. In this way, the flow rule can be expressed without describing its control flow and avoid the logic error [72].

As the controllers OpenDaylight, ONOS and Floodlight are based on JVM (Java virtual machine), which can be configured to provide a JVM-level computing resource control, which means the malicious application cannot crash the host of these controller; however, the malicious application can still exhaust the JVM computing resource or attack other legitimate applications running on controller.

After conducting an analysis, we find that these five controllers lack of the security mechanisms such as connection verification, traffic control, and app-based access control. For resource usage control and system recovery are partially support. In contrast, OpenContrail and ONOS provide a relatively securer services compared with other controllers.

The analysis of this section is resumed in Table 2.5. The symbol (✓) means the security principle is considered and realized in the controller implementation. The abbreviations of ODL, FL, and OC represent OpenDaylight, Floodlight, and OpenContrail respectively.

2.7 Conclusion

This chapter presents a 3-dimensional approach to study the security issues of the SDN controller. We conclude that the southbound interface, the northbound interface and core services are the three weakest components of the SDN controller. The southbound interface suffers from the data-to-control spoofing packets and flow flooding. Attackers tamper core services of the controller via the northbound interface. The characteristics centralization and off-the-shelf make the SDN controller vulnerable from DoS and elevation of privileges(mainly due to the injection of malicious application to the controller's run-time) respectively. Moreover, the SDN controllers lack still the security mechanisms like connection verification, application-based access control, and data-to-control traffic control. In the following chapters, we attempt to mitigate the app-to-control threats due to the malicious network application operations.

Chapter 3

Security Enhancing Layer for SDN

Based on the analysis in Chapter 2, we found that one of the major threats is due to the malicious operations of the network application. Although the access control can be one of approaches to secure the SDN controller against the malicious requests from the network application, but it is insufficient to limit network application behaviors with only authentication and gross-grained authorization or to merely adopt a role-based authorization. We need to control network application in a fine-grained way with AAA, i.e., authenticating network application, authorizing the operations, and accounting or monitoring network applications behaviors [110] [109] [120] [144] [130] [29]. We continue the work [39] to provide a wider and deeper study on how to use the REST-like system to protect SDN controller from malicious network application. Firstly, we explore the functions that can possibly be deployed via the REST-like system, and then add a security-enhancing layer (SE-layer), where it is implemented a protocol to secure the interaction between the network applications with the SDN controller. The prototype I of this SE-layer is called *Controller SEPA*. In this prototype, the SE-layer secures the SDN controller with application authentication(token-based) and authorization. Moreover, it repacks and standardizes the data through the northbound interfaces for hiding the controller's sensitive information from the malicious scanning. It also provides a more fine-grained access control between the application plane and the control plane with. Finally, it enables the controller being decoupled from the network application via TCP/IP socket for avoiding the threat from the resource exhaustion attack. We study the feasibility network applying this framework to five open-source controllers, which include OpenDaylight, ONOS, Floodlight, Ryu and POX. We implemented the SE-layer prototype I *Controller SEPA* upon the controller OpenDaylight and the results show that the deployment operates with very low complexity (most of time the modification of source codes is unnecessary) and with the performance overhead 0.1% -0.3%. We continue to develop the SE-layer prototype II, called *Controller DAC*, by adding the app-to-control request accounting service in this SE-layer,

as we found that malicious network application still can infect the SDN controllers by the API abuse, even the controller is hardened by the static permission control as proposed in the works [120], [144], [70], and [22]. We address the app-to-control threats based on the analysis with the four permission categories: READ, ADD, UPDATE and REMOVE on four open source and active SDN controllers, including OpenDaylight, ONOS, Floodlight, and Ryu. All of these SDN controllers cannot be immune from the attack caused by the API abuse. The prototype II *Controller DAC* works as a controller-independent dynamic access control system for protecting SDN controllers against API abuse. In our implementation upon the OpenDaylight controller, *Controller DAC* requires low deployment complexity for securing SDN controllers, and most of time its operation is independent from underlying SDN controller. The preliminary experimental results show that *Controller DAC* can prevent SDN controllers from API abuse with less than 0.5% performance overhead. This SE-layer can hence enhance the security of the SDN controller in an efficient way with the fine-grained access control.

3.1 Background and motivation

As the northbound interfaces in SDN are still non-standardized and very diverse, this results in remaining many security issues for the app-to-control access control. One approach is to add permission sets for controlling the API request [120] [144] [130] [22]; another approach is to use programming languages to specify the security constraints to the app-to-control access [140] [12] [58]. We introduce the access control based on permission sets here as these works inspire us to design the access control in our SE-layer.

3.1.1 *PermOF* and *OperationCheckpoint* permission sets

In [144], the authors propose 18 permission sets to control network application with the four categories Read, Notification, Write, and System. The Read category is for reading the data such as topology and flows. The Write permission is used to modify the flow entries or send *packet_out* message. Notification is for receiving the network events and System permission is for the system-level calls. The proposal of *PermOF* does not have the real implementation. In [120], the permission control are implemented by assigning permission verification to the related functions. However, it may be infeasible to secure the controller's functions by modifying every function in the source codes. In the worst scenario, it would need to scan all the related functions (methods). For example, in order to control the topology information, *OperationCheckpoint* needs to modify the codes of the two methods: `getAllSwitchMap` and `getLinks` in two different classes `Controller.java` and `LinkDiscoverManager.java`,

respectively, since both of them provide the topology related information in Floodlight. In some SDN controllers, like OpenDaylight and ONOS, it is very hard to scan and find out all these related-methods and harden them. The permission sets for *PermOF* and *OperationCheckpoint* is summarized in Table **??**. The symbols in Table**??** ‡ means the permissions are proposed both in *PermOF* and *OperationCheckpoint*, † only in *PermOF*, and ∗ only in *OperationCheckpoint*.

3.1.2 States-based permission control

Inspired by Android permission control, [70] is a permission system based on OF messages states and the actions. It classifies the OF messages in five state categories as following:

- **INITIAL.** Network application in this state when controller just starts.
- **READY.** Network application in this state after INITIAL state, i.e., it's ready to receive a new event
- **PACKETIN.** Network application in this state if controller receives and forwards a *packet_in* message
- **FLOWREN.** Network application has this permission if controller receives a $flow\_removed$ message
- **PORTSTATUS.** Network application has this permission if controller receives a *port_status* message

These states work with five actions for the network application : (1)Sending OF *packet_out* message, (2)Sending $flowmod\_add$ message, (3)Sending $flowmod\_delete$ message, (4)Sending $flowmod\_modify$ message and (5)Sending *stats_request* message. In addition, there are two permissions for limiting the network application to access the controller's resource at the running time. The first one is **DATABASE** for controlling the access internal storage and the second one is **SYSTEMCALL** for limiting to execute system call. The permission set of the network application can be structured in a format similar to XML. The network administrators can use this system to select permitting operations, thereby allowing the necessary features for network application to be specified. Specifically, this approach provides the permission system with five states for each of the permissions, which helps application users to determine when the application uses each OpenFlow protocol in a more fine-grained way. However, we believe that *OperationCheckpoint* can achieve the same goal by mapping its access control to the OpenFlow message. This permission control also requires to modify the controller codes to be adopted.

3.1.3 Role-based access control

FortNOX and SE-Floodlight propose role-based authorization [109] [110]. The role-based source authentication recognizes by default three authorization roles among those agents that produce flow rule insertion requests. The first role is that of human administrators, whose rule insertion requests are assigned the highest priority. Second, security applications are assigned a separate authorization role. Flow insertion requests produced by security applications are assigned a flow rule priority below that of administrator-defined flow rules. Finally, non-security-related network applications are assigned the lowest priority. Roles are implemented through a digital signature scheme, in which FortNOX is preconfigured with the public keys of various rule insertion sources. If a legacy network application does not sign its flow rules, then they are assigned as the default role and priority of a standard network application. In addition to role-based authentication, SE-Floodlight proposes that northbound messages can be divided into four types for assigning permissions to different roles, which are listed as following [109] [110]:

- **OF message** defines the OpenFlow elements such as the data-path identifier for a specific switch.
- **Controller service message** provides remote access to controller services.
- **Internal message** is used for any configured non-SSL messages such as error events or connection testing etc.
- **Extensions message** controls the access to auxiliary controller functions such as network topology.

While this permission set design is coarse-grained and not used to constrain the operation of network application. As a matter of fact, the ONOS security mode and AAA module of OpenDaylight provide role-based (or called user-based) API authorization [101]. The security-mode of ONOS adopts the "roles" proposed in FortNOX [96]. The Java APIs in ONOS are also secured with static permission set based on network application for the internal APIs [16].

3.1.4 Access control design challenges

The challenge for these permission set proposition is on the implementation: how to implement these permission set on the current SDN controller as it exists more than 20 SDN controllers [125]? We look for an approach which should include the characteristics as following:

Category	Permission
Read	‡ *read_topology*
	‡ *read_all_flow*
	‡ *read_statistics*
	‡ *read_pkt_in_payload*
	∗ *read_controller_info*
Notification	‡*pkt_in_event*
	‡*flow_removed_event*
	‡*error_event*
	†*topology_event*
Write	‡*flow_mod_route*
	‡*flow_mod_drop*
	‡*flow_mod_modify_hdr*
	‡*mofidy_all_flows*
	‡*send_pkt_out*
	‡*set_device_config*
	‡*set_flow_priority*
System	†*network_access*
	†*file_system_access*
	†*process_runtime_access*

Table 3.1 PermOF and OperationCheckpoint permission control

1. **Extensible.** This means this approach should be controller-independent. This should be a generic approach which can be extended to different SDN controller with low deployment complexity.

2. **Feasible.** Some proposition is hard to applied to the actual situation. For example, the approaches like *OperationCheckpoint* [120] needs to scan all the related functions, which increases the cost to implement the access control mechanism and difficult(if not impossible) in some large-scale controllers such as OpenDaylight and ONOS. The modules compose of these projects are independent, this makes hard to cover all the related-modules with the permission control.

3. **Low performance overhead.** The performance overhead for securing the controller with access control should not be low. For example, the FortNOX accounts the rule conflict by implementing the conflict detector in the controller kernel-level [109]. SE-Floodlight extends from FortNOX to detect the flow rule conflicts in controller-level; however, it is not suitable for big networks as it will produce tremendous latency when the controller verifies every flow rule and reduce the controller performance [110].

3.2 App-to-control threats

As SDN enables networking functionalities to be written in software by using open APIs to facilitate development and accelerate network innovations. Unfortunately, problems arise when an network application contains flaws, vulnerabilities, or malicious logic that may interfere with control layer operations. In worse scenarios, the attack from app-to-control can be caused not only by the non-authenticated network app but also by the authenticated one such as in the flow rule conflict [110] [132]. We enumerate the threats here which are targeted to be mitigated with SE-layer.

3.2.1 Data tampering

Once the malicious application can access the controller's data storage or internal memory, the abuse of such trust could lead to various types of attacks impacting the entire network. For example, the *packet_in* count value is kept in the controller's internal storage for the usage of DoS detector or traffic monitoring. However, a malicious application can clear *packet_in* count in the internal storage to confuse the DoS detector application. The controller also contains the network link information and flow rules in data storage, if the network application can modify these values, the topology and flow rules will be tampered [70] [130]. The problem is caused by the poor access control and non-decoupling of the network application. Because if the network application is non-decoupled from the controller, it will be easier to access the controller's runtime resources, such as memory or database. Even the type-safe programming language like Java, the attacker can use JNI(Java native interface) to access the host's memory.

3.2.2 Illegal functions calling

The SDN controllers always contain built-in functions for accelerating network application development. However, once these functions are used maliciously, it will cause the controller to crash or to be manipulated. For example, a malicious network application can terminate the controllers by calling the function `exit()` in Floodlight, OpenDayLight and POX (`sys.exit(0)`) [130]. The `IOFMessage Listener` service in Floodlight can be used to change network application in order to process *packet_in* message, as a result, the malicious network application can interrupt the communication for *packet_in* messages among with other network applications by modifying the order via this service [126]. The illegal function calling problem cannot be mitigated merely by access control as some of the functions are system-level like `exit()`. The malicious application can inject the system-level commands easily if it does not decouple from

the controller's runtime. Hence, the decoupling of the network application and then authorizing the operation can really protect the controller against the illegal function usage.

3.2.3 Malicious scanning

Network application exchanges data with SDN controller via northbound interface. One of the most popular northbound interfaces is RESTful API, which is adapted by controllers such as OpenDaylight, ONOS, Ryu, Floodlight, etc. Unfortunately, [131] proves that an attacker can identify a SDN-based network for the further attacks, eg., DoS/DDoS, by estimating the time of inserting a flow entry in the network. Even worse, [69] and [8] show the risk to expose the type of a SDN controller to the attacker. For example, once the attacker identifies a Floodlight controller which domains a SDN-based network, he/she can launch the malicious requests until saturating the controller's memory, technically JVM.

3.2.4 API abuse

Alough the access control is one of the most general approach to protect the controller against the malicious application. However, the static permission control proposed by [144] [120] cannot really secure SDN controllers from API abuse. We exploit the vulnerabilities of the static access control along with four permissions categories as they propose, i.e., READ, ADD, UPDATE and REMOVE, and point out their vulnerabilities in this section.

API abuse with READ permission

READ permission is the less authorization which can get the related information but without the right to modify them. However, even malicious requests can infect the SDN controllers. We test this attack by requesting to read topology information with Python scripts on four SDN controllers. We found that none of SDN controllers protect this kind of malicious requests. When we use more than 3 scripts to request SDN controllers continuously with a infinite loop, some of them happen to drop the packets from southbound, such as ONOS and Floodlight. In practice, it is difficult to exhaust all the CPU capacity as we can find in Table 3.2 with the malicious requests via RESTful APIs as the network bandwidth is limited. However, according to our observation, we found that when the southbound API and northbound API share the same bandwidth in SDN, i.e., the same NIC(network interface card). Once this network bandwidth is saturated by requests from northbound, which results in dropping the packets from southbound. Even though we do not found OpenDaylight and Ryu to drop the packets from southbound obviously under this attack, but both of them suffer from responding slower.

CPU usage	OpenDaylight	ONOS	Floodlight	Ryu
Basic CPU usage	21.6%	2.7%	2.3%	0%
1 script request	42.3%	27.6%	9.6%	9.6%
2 scripts request	44.7%	36.5%	13.9%	23.6%
3 scripts request	45.7%	36.4%	14.6%	26.9%
4 scripts request	47.3%	36.7%	14.6%	32.9%
5 scripts request	47.2%	36.0%	14.7%	34.2%

Table 3.2 CPU usage overhead caused by malicious requests

API abuse with ADD permission

In OpenFlow-based SDN, forwarding decisions are flow-based. Each flow entry (or flow rule) in a flow table contains match field, actions, counters, priority, and timeout, etc. Basically, when an incoming packet matches a match field in flow entry, the switch performs corresponding actions on the packet and updates the corresponding counters. If multiple rules are matched, the priority field serves as a tie breaker to determine which flow entry should be applied to that packet. However, this design is still easy to compromise, because most of the OF switches available on the market have limited ternary content-addressable memory (TCAMs), with up to 8000 entries [45]. Gigabit Ethernet (GbE) switches for common business purposes have already supported up to 32 000 Layer 2 (L2) + Layer 3 (L3) or 64 000 L2/L3 exact match flows [37]. Enterprise class 10GbE switches are being delivered with more than 80 000 layer 2 flow entries [88]. Other switching devices using high-performance chips (e.g., EZchip NP-4) provide optimized TCAM memory that supports from 125 000 up to 1 000 000 flow table entries [92]. However, in our tests, Open vSwitch has higher capacity to store the flow entries up 148223 with SDN controller Floodlight (version 1.2). But as the OF switches in SDN are dumb, once a malicious network application inserts more than this limitation through controller, the new coming flow entries will flush out and replace the existing ones, without considering the priority of flow entries, i.e., the flow entries with lower priority can replace the flow entries with higher priority and be executed on data plane. Other SDN controllers such as ONOS will crash directly once the number of flow entries in one switch is more than 45000. The OF switches connected with OpenDaylight will produce a considerable latency to find a flow rule to apply, even drop the packet, when the flow entries are up to 140000, and the OpenDaylight's API [1] for fetching flow entries can no longer work.

All of the SDN controllers tested in our research do not provide any protection mechanism for this kind of threat even secured by static permission control. Moreover, a malicious network application can insert a flow rule to reach the unexpected destination, to block another legitimate

[1] /restconf/operational/opendaylight-inventory:nodes

service or to confuse the service provided by other network apps coexisting on the same SDN controller such as the action `set` and `priority` in flow rule [109] [110] [147]. These security issues cannot be resolved by simply allowing or disabling an network application to insert flow entries.

API abuse with UPDATE permission

The poor network application priority management makes the network application has UPDATE permission easy to compromise the network. As aforementioned, if a packet matches multiple rules, the flow entry with highest priority will be applied. However, the coexistence of network applications makes it hard in practice to determine which flow entry should prevail over others [110] [132]. For instance, as we discussed in last chapter, suppose application 1 initiates a series of flow rule insertions designed to quarantine the flows to and from a local Internet server. Application 2, a load-balancing application, redirects incoming flow requests to an available host within the local Internet server pool. Suppose the Internet server quarantined by application 1 subsequently becomes the preferred target for new connection flows, as this quarantined server is now the least loaded server in the pool. In this case, should application 1 or application 2 prevail [110]? Even worse, there is no constraint among network applications to modify the flow entry inserted by other network applications. In static permission control design, network application has the permission to modify flow entry can possess the full operation permission to modify any field in any flow entry inserted by any network applications. A malicious network application can compromise network with UPDATE permission by modifying the priority field in other flow entries and make its own flow entries have highest priority. In OpenDaylight, the API `/sal-flow:update-flow` [2] can be abused to modify any flow entries with any priority inserted by any network application. Similar security problems can happen on ONOS and Ryu due to poor priority management on the APIs such as `/devices/<deviceId>` and `/stats/flowentry/modify` respectively.

Fortunately, this threat is mitigated in Floodlight by adopting separated APIs to modify flow entry in different network application. The `/wm/staticflowpusher/json` with POST method can only modify the flow entry in Statistic Flow Entry module while `/wm/firewall/rules/json` with POST method modifies the flow entries in Firewall module. But the `IOFMessage Listener` service in Floodlight could be used to change network application in order to process *packet_in* message. As a result, the malicious network application can interrupt the communication for *packet_in* messages among other network applications by modifying the order via this service. Similarly, any network application in ONOS can access to global param-

[2] API root: http://<IP>:8080/restconf/operations/

eters in the controller via the northbound service `ComponentConfigService` to set `PACKET OUT ONLY` to be true for degrading the overall performance of the network [126].

API abuse with DELETE permission

Apparently, an network app with DELETE permission can easily compromise network in a violent way, such as directly removing other flow entries. The SDN controllers including OpenDaylight, ONOS and Ryu suffer from this problem as we mentioned in Section 3.2.4. The APIs `/sal-flow:remove-flow` with DELETE method, `/flows/<deviceId>/<flowId>`, and `/stats/flowentry/clear/<dpid>` provided by OpenDaylight, ONOS and Ryu respectively can be used to remove all other flow entries without any consideration of the priority in other flow entries.

In Floodlight, this threat is also mitigated by assigning APIs to different network apps, like `/wm/staticflowpusher/clear/<switch>/json` and `/wm/staticflowpusher/json` with DELETE method can only remove the flow entry in Statistic Flow Entry module, while `/wm/firewall/rules/json` with DELETE method removes the flow entries in Firewall module, and `/wm/acl/clear/json`, and `/wm/acl/rules/json` with DELETE method can only remove the flow entries in ACL module.

Conversely, a malicious application can also frequently delete the rules in a switch if it is only limited based on static permission control, so that every time when a new packet arrives, it should be sent to the controller for requesting a rule to enforce. This attack can affect the performance of both the data plane and the control plane by consuming controller's resource and degrading the overall performance[70] [83]

After a deeper study, we provide a non-exclusive list in Table 3.3 for the APIs needed to be secured from API abuse in these four SDN controllers.

3.3 Services provided by the SE-layer

For mitigating the threats mentioned beforehand, the services provided by the SE-layer are as following.

3.3.1 Authentication and Authorization

To prevent the controller from malicious network application, the basic protection is to authenticate the network application. For example, SE-Floodlight and Rosemary use digital signatures to authenticate the network application [110] [130]. [39] also adopts key pairs to authenticate the network application.

Controllers	READ	ADD
OpenDaylight	/opendaylight-inventory:nodes/node /network-topology:network-topology	/sal-flow:add-flow
ONOS	/devices /link	/flows/ <deviceId> /link
Floodlight	/wm/device	/wm/staticflowpusher/json
Ryu	/stats/flow/<dpid> /stats/group/<groupId>	/stats/flowentry/add /stats/groupentry/add
	UPDATE	DELETE
OpenDaylight	/sal-flow:update-flow	/sal-flow:remove-flow
ONOS	/devices/<deviceId> /links/<linkId>	/flows/<deviceId>/<flowId> /links/<linkId>
Floodlight	IOFMessage Listener	
Ryu	/stats/flowentry/modify /stats/groupentry/modify	/stats/flowentry/delete /stats/groupentry/delete

Table 3.3 The RESTful APIs needed to be secured from API abuse

After the application authentication, the controller should authorize the requests of the network application, including the enforcement of the flow entry. [120], [70], [144], and [39] show the network application-based access control. *PermOF* proposes the use of 18 permission sets under four distinct categories without experimental evaluation of the access control system proposed. *OperationCheckpoint* adopts the part of the permission set of *PermOF* in constraining the northbound interface access and implements the permission set in SDN controller Floodlight, but this permission set does not enable network application users to distinguish malicious applications from benign ones [120]. Moreover, this permission set is not controller-independent, i.e., it should modify the source codes on every controller that attempts to apply this system. [39] proposes a controller-independent solution for securing the integration of external network app via RESTful API. However, it is not able to find the malicious operations of the network application. Inspired by Android permission control, [70] is a permission system based on OpenFlow messages states and the actions.

3.3.2 Accounting

The audit log record is useful for network troubleshooting as well as a data source for security monitoring as an network application modifies flow tables or sends a *packet_out* message etc [120] [39] [110] [32]. For instance, SE-Floodlight [110] introduces an audit subsystem that traces all security-related events occurring in the control layer. With this auditing record, the

controller can report to the network administrator the event time, the message type, and the full message content, etc. Hence, the data tempering, illegal function calling and malicious rule injection can be traced by the accounting records. Hence, we need the accounting service to protect the controller against the API abuse by evaluating the requests from the network application.

3.3.3 Network application isolation

A malicious network application can create multiple memory objects, large number of threads, infinite loops or non-stop growing linked lists to deplete the resources of the controller's host. Controllers such as NOX, Beacon, Floodlight and OpenDaylight do not limit memory allocations to its applications, which can ultimately result in the controller crashing with an out-of-memory error [70] [130]. One of the primary reasons behind the fragility of the controllers is their tight coupling with applications. YANC adopts UNIX-like permission to separate network apps from exposing the network configuration and stating it as a file system [81]. [130] and [19] propose to separate network app from SDN controller even by processes. Rosemary [130] separates network application from the controller by invoking each new network application in Rosemary as a new process. Rosemary's network application connects to the SDN controller process through the IPC (inter-process communication). The basic network services in the Rosemary kernel communicate with each other through an IPC channel, and the implication is that if a service crashes, other services are unaffected. Fortunately, as the SE-layer adopts RESTful API to communicate with network application via TCP/IP. In this case, the network application can be decoupled and run on any host remotely. This can secure a controller from an exhausting resources attack from malicious network application and prevent the controller from crashing. As we repack the service into RESTful APIs, network application can call these services via the network and run the services in a machine isolated from the controller. This is a more secure way than decoupling network application from the controller by process or file system [130] [81].

3.3.4 Information undisclosed

As an attacker should first identify the controller before compromising a SDN-based network [131] [69] [8], hiding the sensitive SDN controller information can be used to avoid the malicious scanning. The threats we aims to mitigates and the corresponding countermeasures are summarized in Table 3.4.

	Authentication/ Authorization	Operation accounting	Information undisclosed	Application isolation
Data tempering	✔	✔		✔
Illegal functions calling	✔	✔		✔
Malicious scanning			✔	
API abuse		✔		

Table 3.4 App-to-control attacks and corresponding countermeasures

3.4 SE-layer Prototype I: *Controller SEPA*

Based on the threat models in Section 3.2, we find that it is insufficient to limit network application behaviors with only authentication and coarse-grained authorization or to merely adopt a role-based authorization. We need to control network application with AAA, i.e., authenticating network application, authorizing network application, and accounting (or monitoring network applications operations) [120] [144] [109] [110] [130] [29]. Even more, for avoiding the system-level function calling by the network application, the SE-layer should be able to separate the network application from the controller for isolating the resource usage.

3.4.1 Design principle of *Controller SEPA*

We continue the work [39] to provide a wider and deeper study on how to use the REST-like system to protect SDN controller from malicious network application. Firstly, we explore more functions that can possibly be deployed via the REST-like system, called *Controller SEPA* (SEPA: **S**ecurity-**E**nhancing **P**lug-in for network **A**pp). Secondly, we study the feasibility of applying this framework to five open-source controllers, which include OpenDaylight, ONOS, Floodlight, Ryu and POX. The high-level architecture of *Controller SEPA* is depicted in Figure 3.1.

Authentication

Controller SEPA can verify the network application based on password or authentication token. Before requiring the data from the controller, the network application should register with the required permissions and then be validated by the network administrator. Once the network application is validated, the request of the network application will be verified by the password.

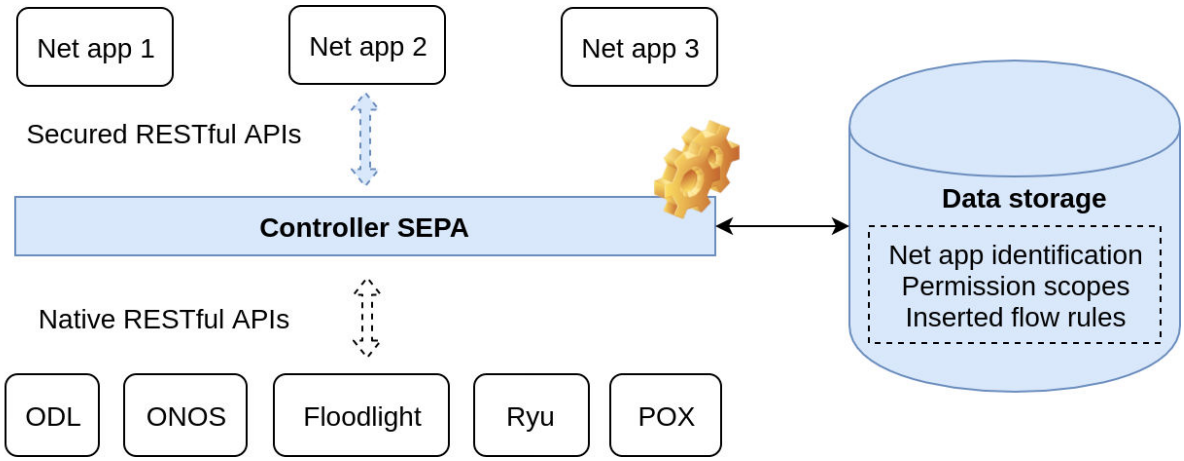


Fig. 3.1 High-level view of *Controller SEPA*

If the password is matched, the controller will deliver a token to the network application and automatically refresh in a time interval, eg., 30 minutes.

Authorization with fine-grained control

When the network application registers on the controller, it should deliver its operation permissions on the SDN controller(actually here is to the SE-layer as it delegates all the interaction between the controller and the network application). The SE-layer will deliver an authentication token, which contains the information about the network application such as id, permission scopes, and priority, etc. *Controller SEPA* will check its permission by this token. Even more, as *Controller SEPA* repacks the services of the controller, it can control them in a more fine-grained way. An application has permission to access the database, it will have full permission to obtain any resources without limitation. But with this module, the network application will be constrained by the permission as to which resources they can get. *Controller SEPA* can even limit their scope in the same resources. For example, if both of the network apps have permission to insert a flow rule, one of them may have full permission to insert a rule for any network with any priority while the other is constrained by the fact that it only has permission to insert the rule for 10.0.0.100/24 to 10.0.0.200/24 with priority between 100 and 200. In doing so, we can control the network application in a more refined way.

Hence, we extend the permission set proposed in [39], as well as referred to the permissions in *PermOF* and *OperationCheckpoint* [120] [144]. We conduct first the possibility of the permission sets proposed by the current existing controllers and summarize the categories of the permission sets in Table 3.5. Furthermore, referring these sets to the proposition in *PermOF* and *OperationCheckpoint*, the permission set used in *Controller SEPA* is as shown in Table 3.6. The symbol ★means this API is officially supported in the SDN controller, *Controller SEPA* can

Category	Permission	ODL	ONOS	FL	Ryu	POX
Read	host.read	★	★	★		✚
Read	switch.read	★	★	★	★	✚
Read	link.read	★	★	★		✚
Read	port.read	★	★		★	✚
Read	flowmod.read	★	★		★	✚
Read	group.read	★	★		★	✚
Read	vlan.read	★	★	★	★	
Read	topology.read	★	★	★		
Read	statistics.read	★		★		
Read	application.read		★	★		
Read	controller.read			★		✚
Write	port.write				★	
Write	flowmod.add	★	★	★	★	
Write	flowmod.write	★	★	★	★	
Write	vlan.add	★	★	★	★	
Write	vlan.write	★	★			

Table 3.5 RESTful-based northbound interface list of the current controllers

use it directly and repack without touching the controller's source codes; the symbol ✚means SDN community has released the contributions for this API. The permission description is as following.

- **host.read** Read all or specific host(s) info
- **switch.read** Read all or specific switch(s) info
- **link.read** Read all or specific link(s) info
- **port.read** Read all or specific ports(es) info
- **flowmod.read** Read all or specific flow entry(ies) info
- **group.read** Read all or specific group info
- **vlan.read** Read vlan info in a flow entry
- **topology.read** Read topology
- **statistics.read** Read statistics

- **application.read** Read other network app info
- **controller.read** Read controller info (listen IP, port)
- **port.write** Update port status
- **flowmod.add** Add a flow entry with any action
- **flowmod.write**Update/remove a flow entry with any action
- **vlan.add** Add a vlan tag in a flow entry
- **vlan.write** Update/remove vlan tag in a flow entry

For example, the APIs for reading *packet_in*, *packet_out*, $feature\_reply$, and $flow\_mod$ events can be used in SPHINX to form the flow graph and detect the malicious flow in the data plane [83]. The APIs used to read port status (`/get/port/*`) can be used in TopoGuard to detect the malicious host migration [137]. We provide a nonexclusive list of the possible APIs because the APIs are demand-driven, i.e., we should know the needs of network application and then offer the necessary APIs; hence, we can only list the basic APIs. More features such as queue, meter, group, MPLS tags and priority-setting etc will be discussed in our future work. We conduct the feasibility study of this permission sets as following with five current mainstream SDN controllers.

Controller-independent

For reducing the deployment complexity, this SE-layer should be controller-independent. This means this should be a generic approach to the most SDN controllers. Hence, the *Controller SEPA* can set up the connection with the SDN controller and the controller delegates operation permission to *Controller SEPA*. The network application only communicates with the services provided by *Controller SEPA*. Hence, it is controller-independent, i.e. it can reduce the deployment complexity for the application to the current SDN controllers [39]. In the following, we discuss case by case why this SE-layer is controller-independent by studying the northbound interfaces of the current mainstream open source controllers, which are OpenDaylight, ONOS, Floodlight, Ryu and POX.

OpenDaylight case

OpenDaylight uses Java API or RESTCONF to communicate with data storage. Hence, we benefit from RESTCONF protocol, which is implemented as an network application in OpenDaylight

Category	Permission	SE-layer API
Read	host.read	/get/device/<all> or <hostId>
Read	switch.read	/get/switch/<all> or <switchId>
Read	link.read	/get/link/<all> or <linkId>
Read	port.read	/get/port/<all> or <portId>
Read	flowmod.read	/get/flowmod/<switchId>/<all>or<entryId>
Read	group.read	/get/flowmod/<switchId>/<all>or<entryId>
Read	vlan.read	/get/vlan/<switchId>/<all>or<entryId>
Read	topology.read	/get/topo
Read	statistics.read	/get/statistics
Read	application.read	/get/app/<all>or<appId>
Read	controller.read	/get/controllerinfo
Write	port.write	/post/port/<switchId>/<entryId>
Write	flowmod.add	/put/flowmod/<switchId>
Write	flowmod.write	/post or remove/flowmod/<switchId>/<entryId>
Write	vlan.add	/put/vlan/<switchId>/<entryId>
Write	vlan.write	/post or delete/vlan/<switchId>/<entryId>

Table 3.6 Permission sets proposition for *Controller SEPA*

(`restconf` module). OpenDaylight provides a rich RESTful API support and we repack these services into *Controller SEPA* as standard RESTful API opening to network application. For example, we repack the `addresses` in `/openflow:n/node-connector/openflow :n:m`[3] to `/get/device/*` for obtaining the host's information such as MAC and IPs and `/openflow:n/table/0/` to `/get/flowmod/<switchId>/*` to get the flow entries in switch n. OpenDaylight supports only role-based control, not application-based, that means, once a network application has the permission to use `restconf` module in OpenDaylight, it will have the full permission to operate the data storage. Therefore, *Controller SEPA* can be a security-enhancing module to provide application-based AAA control for OpenDaylight with low deployment complexity [101].

ONOS case

Similar to OpenDaylight, ONOS also adopts Java API as well as implements RESTCONF protocol as an network application (`org.onosproject.drivers`), we repack the services in ONOS by merging, for example, `/v1/devices` and `/v1/links` into `/get/switches/all` in *Controller SEPA* to get the complete information about switches and their connection status.

[3] API root: opendaylight-inventory:nodes/node

SDN controller	OpenDaylight Beryllium SR2
OS	Ubuntu 16.04
CPU	Intel i7, 8 cores
Memory	16G DDR2
Network simulator	mininet
Topology	Linear with 20 switches * 20 hosts
Security extension	RESTful APIs based on Java Spark

Table 3.7 *Controller SEPA* experimental settings

The ONOS `strict` mode also uses role-based control like OpenDaylight. This means that it can also benefit from the application-based AAA control provided by *Controller SEPA* [95].

Floodlight case

Floodlight supports RESTful APIs natively, such as `/wm/device/` for showing the details of the hosts connections, and `/wm/staticflowpusher/list/<switch>/json` for reading the proactive flow rules in switch. The shortage of authenticating the use of network app can be supplemented in *Controller SEPA*, which can provide the digital signature service to authenticate the network application without touching the source code in Floodlight. Evidently, it can also benefit from the application authorization and accounting services in *Controller SEPA* [35].

Ryu case

Ryu is a component-based SDN controller, which provides complete northbound interface for network application development as shown in Table 3.5. Even more, Ryu (version 3.5) does not yet support the TLS for northbound interface, i.e., no HTTPS for securing the communication between the network application and the SDN controller. *Controller SEPA* can be used as the security-enhancing module to improve the problem of shortage of northbound interface encryption as well as provide app-based AAA control for Ryu [94].

POX case

POX, an early SDN controller, does not use RESTful API, but Python API, as the official northbound interface. That means we should implement manually the RESTful API server in POX, such as Flask, for transferring data in POX via RESTful API. Fortunately, we can find contributions in community for the RESTful API support in POX such as pox-jsonrest on GitHub [15].

	API 1	API 2	API 3	API 4	API 5	API 6
OpenDaylight	15328	15184	67248	1275	1263	1305
Controller SEPA	15355	15194	67269	1281	1265	1309
Percent(%)	0.176	0.066	0.031	0.047	0.158	0.307

Table 3.8 Performance overhead of *Controller SEPA*

Information undisclosed

Controller SEPA repacks all the services provided by the SDN controller, including RESTful APIs (OpenDaylight, ONOS and Floodlight), OSGI bundles(OpenDaylight and ONOS) or Python API (POX) into new standard APIs and exposes them to the network applications. As network application can only communicate with *Controller SEPA* instead of the SDN controller, network application does not know the details concerning the SDN controller or which version is providing the services, i.e., the controller is protected from malicious scanning.

3.4.2 *Controller SEPA* experimental validation

In our implementation, we tested how *Controller SEPA* secures OpenDaylight (version Beryllium-SR2), which runs on a Ubuntu-based(16.04) machine with CPU Intel i7, 8 cores, and 16G DDR2 RAMS. The controller connects with a linear topology with 20 OF switches and 20 hosts on each (total 400 hosts) simulated by mininet. We resume the testbed configuration in Table 3.7. Figure 3.2 shows that *Controller SEPA* repacks the two APIs provided by different SDN controllers, OpenDaylight and Floodlight(1.2) respectively. In this figure, the repacking service provided by Controller SEPA enables: (a) *Controller SEPA* repacks the API `/network-topology:network-topology` provided by OpenDaylight RESTCONF module to `/get/device/all`; (b) Repacking the API `/wm/device` in Floodlight to `/get/device/all`. For example, both of them repack the information about host with IP 10.0.0.1 and MAC 00:00:00:00:00:01 into the same form of response.

After the repackage, both of them become `/get/device/all` and show the same responses to network applications. By doing so, we hide the sensitive information about controller from network applications. Similarly, we also successfully deployed one application can only insert flow entry with action output to a specific port while the other network application can do more actions such as output to controller and flood the packet for testing the fine-grained control on network application. In these implementations, we did not modify any source code of OpenDaylight and provided a more secure and fine-grained control on network application. In Table 3.8, we find the average latency of 20 times tests after *Controller SEPA* repacks

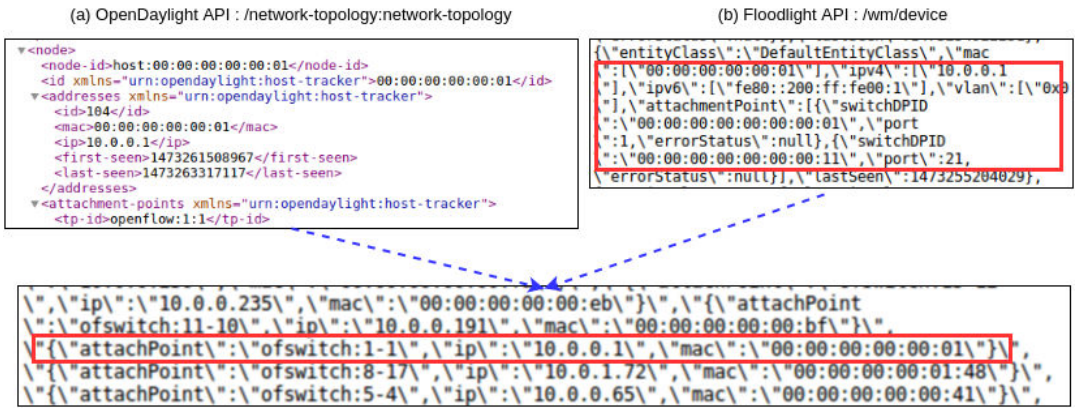


Fig. 3.2 Controller information protection

the services provided by OpenDaylight. The services calling time comparison (millisecond) between OpenDaylight and the repacked services in *Controller SEPA* are as following.

- API 1: get devices info
- API 2: get links info
- API 3: get flow entries
- API 4: insert a flow entry
- API 5: update a flow entry
- API 6: remove a flow entry

We use Java spark framework as RESTful API server and JS as client to call the APIs. The repackage in *Controller SEPA* produces negligible latency, from less than 0.1% to 0.3%. The same design principle can be applied in other SDN controllers as shown in Table 3.6. For API 3 to get the flow entries, which contains the flow entries to allow ICMP packets between all hosts, it closes to 67200 ms after the seventh run from 242000 ms at the first run.

3.5 SE-layer Prototype II: *Controller DAC*

In this section, we develop the SE-layer prototype II *Controller DAC* as the static access control cannot really secure controller from the malicious operations. We enrich the SE-layer prototype I `Controller SEPA` with the dynamic access control for accounting the malicious requests of northbound interfaces from the network applications. We propose SE-layer prototype II, `Controller DAC`, to prevent SDN controllers from API abuse specifically as mentioned in Section 3.2.4.

3.5.1 Design principle of *Controller DAC*

Controller DAC, SDN **Controller D**ynamic **A**ccess **C**ontrol System, provides a controller-independent security-enhancing system for protecting SDN controller against malicious network application with low deployment complexity. It consists of three component, which are a northbound security extension, a controller specific IDS, and a high-level policy engine. Figure 3.3 depicts the high-level architecture of *Controller DAC*. By coordinating these components, *Controller DAC* can provide dynamic access control on network applications by accounting the network application requests. In the following, we introduce the three main components and how they coordinate systematically to secure the SDN controller.

Security extension

The northbound interface in the SDN architecture keeps the communication between the SDN controller and network applications. Generally, it may be any form of APIs such as Java APIs or RESTful APIs. As there are more than 20 SDN controllers that exist in the market today [125], it is infeasible to secure the controller by scanning function by function in each individual SDN controller as [120] attempts. In some cases, like OpenDaylight and ONOS, it produces high deployment complexity to do so. Hence, *Controller DAC* repacks the built-in services provided by northbound APIs as we did in the prototype I *Controller SEPA*. This can be accomplished by repacking RESTCONF services in OpenDaylight and ONOS as well as RESTful APIs in Floodlight and Ryu. [146] shows the repackage produces quite negligible latency (0.1% - 0.3%), which is why we extend this approach to harden APIs. The SDN controller keeps communication exclusively with *Controller DAC* security extension, and security extension exposes these secured RESTful APIs to network applications. Hence, it is controller-independent as *Controller SEPA*, i.e., it can reduce the deployment complexity for the application interacting with the current SDN controllers [39].

This security extension will authenticate and authorize every request from network application as well as checking about the request legality with accounting records provided by controller specific IDS, which we discuss about in more details under Section 3.5.1. This extension adopts both password-based authentication and token-based authentication. Once the password is validated, security extension will deliver a token with expiration validation. Before the token expires, network application can keep calling the services with this authentication token. If the token expires, the network application can automatically refresh the token by delivering the password again. The authorization services provided in security extension can be done by coordinating with the *Controller DAC* policy engine, which we discuss in Section 3.5.1. Thanks to this security extension, every request from network application is uploaded to

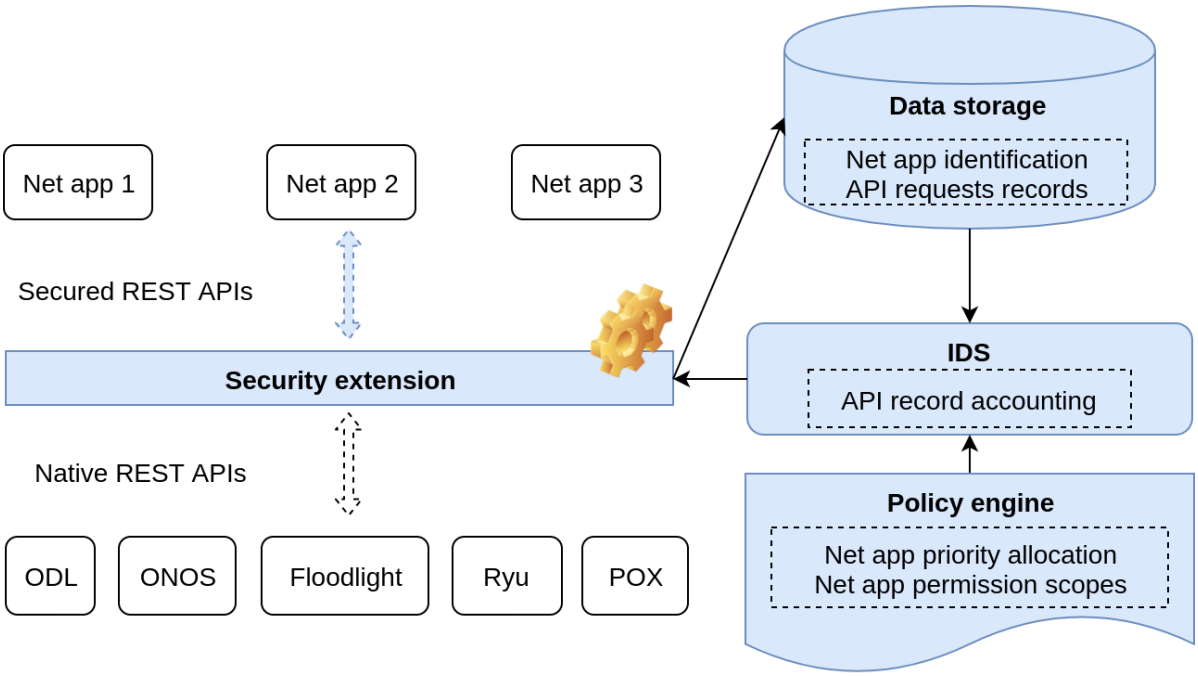


Fig. 3.3 High-level view of *Controller DAC*

the database, including the event time, requested APIs, network application Id, and the request contents.

Policy Engine

Controller DAC has a high-level policy engine which pre-defines the APIs request thresholds for each network application and their permission scopes. Figure 3.4 is the template of how policy engine defines the policies for each network application as they call northbound APIs. For example, ofappX in this policy template has the permission to insert the flow entry(/add/flowentry), but the configuration in policy engine limits it to call this API 50 times per 60 seconds maximum(seconds="60" times="50"), which means ofappX can insert 50 new flow entries per minute maximum. This configuration can prevent this network application to flush out other flow entries inserted by other network applications, as the others could be configured for having higher capacity to insert more flow entries than ofappX per minute.

Moreover, comparing to ofapp1, which has higher priority configuration of, the priority configuration for ofappX is only 80. That means that even ofappX has the permission to modify flow entry(/update/flowentry), but it cannot modify the flow entries inserted by ofapp1.

For fastening the configuration work in policy engine and proving more compatibility with other controllers, policy engine can configure network apps in API-roles instead of configuring them individually. If SDN controllers OpenDaylight installed AAA module or ONOS activated

```
<apps>
      <api-role role="SECURITY">
            <api name="/get/topo,/update/flowentry">
                  <threshold seconds="60" times="100"/>
                  <threshold seconds="3600" times="10000"/>
                  ...
            </api>
            ...
      </api-role>
      <api-role role="DEFAULT">*
            ...
      </api-role>
      ...
      <app appname="ofapp1" api-role="SECURITY" priority="100"></app>
      <app appname="ofappX" api-role="NONE" priority="80">
            <api name="/add/flowentry,/update/flowentry">
                  <threshold seconds="60" times="50"/>
            </api>
      </app>
      ...
</apps>
```

Fig. 3.4 High-level policy template

security mode, their roles configurations can be mapped directly to our API-roles in policy engine. By default, we propose three policy roles as follows by referring to [109] [110], and intend to coordinate this SE-layer with the security extension of the NFV orchestrator[104] [105]:

- ADMIN: This role has the highest operation thresholds and highest network application priority. By default, the network application is configured with this role can use APIs without constraints.
- SECURITY: The network applications with this role can call APIs with higher operation thresholds and higher network application priority than DEFAULT but lower than ADMIN. This role can be used for security-related network applications.
- DEFAULT: For general network applications like topology viewer, routing configuration and performance optimization, they could be configured with this role. This role has lower operation thresholds and lower network application priority than ADMIN and SECUIRTY. Hence, the network applications configured with this role, even if they have permission to modify or remove flow entries, will be prevented from modifying the flow entries inserted by SECURITY role and ADMIN role because these two roles have higher network application priorities.

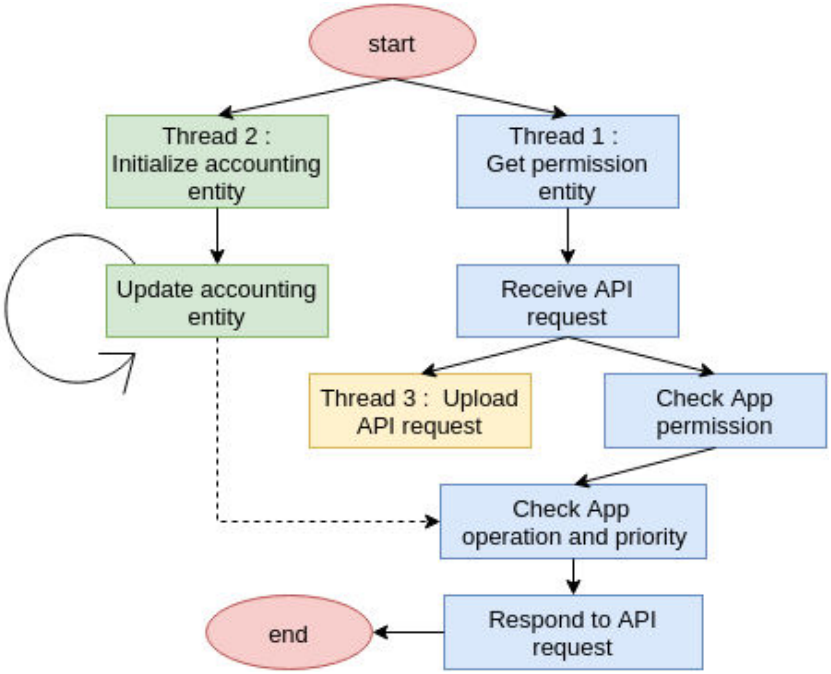


Fig. 3.5 *Controller DAC* work flow

Controller specific IDS

As all the requests from network application is uploaded to database via *Controller DAC* security extension, the SDN controller specific IDS takes the role to accumulate the information about permission scopes and accounting records of network application from database and policy engine. This IDS traces the records with the event time, network application ID, requested APIs, and the message content. Based on this information, this IDS can detect whether network application requests reading data too many times from the controller or is adding additional data into controller, alongside with the thresholds defined in policy engine. Moreover, it detects the legacy for updating and deleting any data in controller based on the network app priority configured in policy engine.

3.5.2 *Controller DAC* experimental validation

In our implementation, we tested how *Controller DAC* secures OpenDaylight (version Beryllium-SR2), which runs on a Ubuntu-based(16.04) machine with CPU Intel i7, 8 cores, and 16G DDR2 RAMS. Controller connects with a linear topology network simulated by mininet, which consists of 20 OF switches and 400 hosts, i.e., there are 20 hosts connecting on each OF switch. In OF tables of each OF switch, the flow entries for full ping among the hosts in networks are inserted. The northbound security extension is written by Java based on Java Spark framework

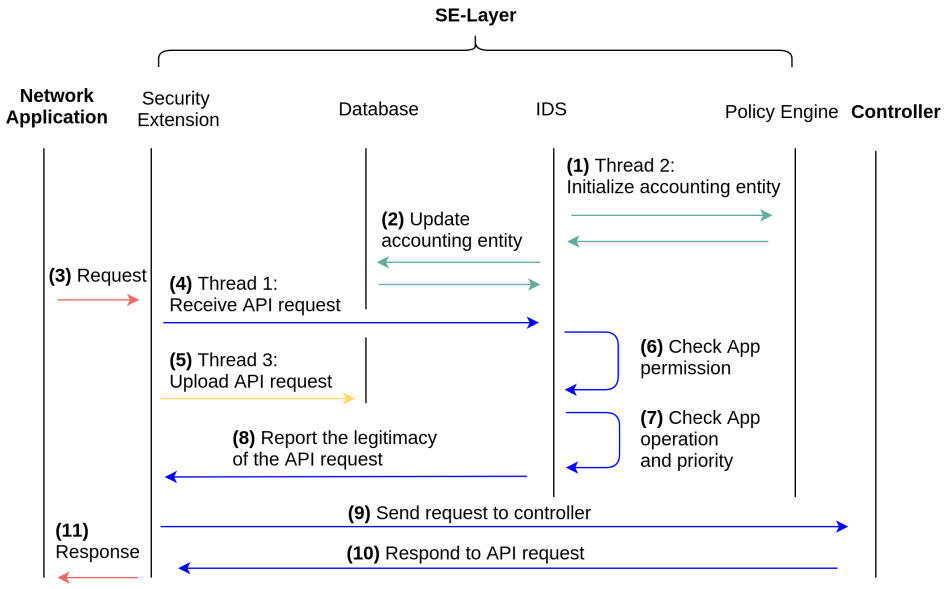


Fig. 3.6 The protocol implemented in the SE-layer

for repacking the OpenDaylight built-in RESTful APIs and securing them with AAA control. The IDS interacts with MySQL as database. To make database resilient, we deploy database on ERS of AWS. The configuration of testbed is resumed in Table 3.9. In order to mitigate the latency caused by network transmission, *Controller DAC* adopts multi-threading processing by opening three threads (1) the first thread is the *Controller DAC main thread*, which validates every request from network application according to permission set and operation accounting record. This thread obtains the permission configuration as well as APIs requests for each network application from policy engine as *Controller DAC* starts. (2) the second thread is used to keep updating the accounting records per second, and store them as a global static entity in memory, which is ready to be called by other thread, and (3) the third thread records every request from network applications on controller and uploads them to database. Figure 3.5 is the work flow of our *Controller DAC* prototype. The policy engine adopts the standard data transmission format XML. All experimental settings are summarized in Table 3.9. Hence, in this SE-layer, we implement a protocol for securing the interaction between the SDN controller and the network applications in the external APIs as depicted in Figure 3.6. Figure 3.6 is mapped from the work flow of the *Controller DAC* in Figure 3.5. The in blue, orange, and green lines of Figure 3.6 represent the thread 1, thread 2, and thread 3 in the Figure 3.5. The thread 1, i.e., the *main thread*, listens to the requests from the network applications, thread 2 keeps updating the accounting results from the data base, and the thread 3 is used to update the database when a new request arrives. The protocol used to validate the request of the network application in this SE-layer can be hence details in 11 steps:

SDN controller	OpenDaylight Beryllium SR2
OS	Ubuntu 16.04
CPU	Intel i7, 8 cores
Memory	16G DDR2
Network simulator	mininet
Topology	Linear with 20 switches * 20 hosts
Security extension	RESTful APIs based on Java Spark
Policy Engine	XML
Database	MySQL 16G on AWS

Table 3.9 *Controller DAC* experimental settings

- Step 1: One thread (thread 2) is used to initialize the accounting entity from the rules in the policy engine
- Step 2: Thread 2 keeps updating the accounting entity per second with the database.
- Step 3: The network application sends a request to the SDN controller. In fact, this request will be captured by the SE-layer.
- Step 4: The main thread, thread 1, receives this request.
- Step 5: Another thread, i.e., thread 3, uploads the request to the database
- Step 6: The main thread checks the permission from the last accounting entity in IDS.
- Step 7: The main thread checks the operation and priority from the last accounting entity in IDS.
- Step 8: The main thread reports the API request legitimacy to the security extension.
- Step 9: If the request is legal, security extension sends request to the controller.
- Step 10: The controller responds the request to the security extension
- Step 11: The security extension repacks the response and sends to the network application.

Figure 3.7 is a part of security extension code, which shows how this extension repacks and hardens OpenDaylight northbound API with dynamic access control. We explain this function as follows.

```
get("/get/topo",(req,res) -> {

    AccountingServiceUtil account = new AccountingServiceUtil();
    account.start(req);

    validatePermission(req);
    validateAccounting(req);

    return new Response(service.getTopo(req));
}, CommonUtil.getJsonTransformer());
```

Fig. 3.7 Snapshot of security extension code in *Controller DAC*

- The return value *Response(service.getTopo(req))* uses to call the RESTful API in OpenDaylight(/ network-topology:network-topology) for getting the network topology. However, before returning this value, the request should be verified by the two functions *validatePermission(req)* and *validateAccounting(req)* for checking the authorization and accounting records respectively. For UPDATE and DELETE permissions, they need one more step to check the network application priority.

- The function *validatePermission(req)* is used to check about the network application permission scope. If this request does not have the assigned permission, the repacked API will throw exception error.

- The function *validateAccounting(req)* gets the accounting records stored in memory from IDS and checks this with the thresholds configured in policy engine. Once it finds this network application runs out its quota to call this API, it returns exception error.

- The lambda expression *get("/get/topo", (req, res) → {...}, ...);* means to repack the service from returning value into "/get/topo" RESTful API, which is based on Java Spark.

- The two lines about starting *AccountingServiceUtil* is to upload this request to database in a new thread.

- Finally, *CommonUtil.getJsonTransformer()* is to convert the data into Json and output it via new RESTful API.

The high level policy will be interpreted in the tree structure as depicted in Figure 3.8. We store this tree with 2 levels HashMap for speeding up the search, where we map the network application to APIs in first level and map each API to thresholds in second level.

Notably, we did not modify any source code of OpenDaylight in our prototype and detected effectively the API abuse. Table 3.10 reports average latency from 20 tests after Controller

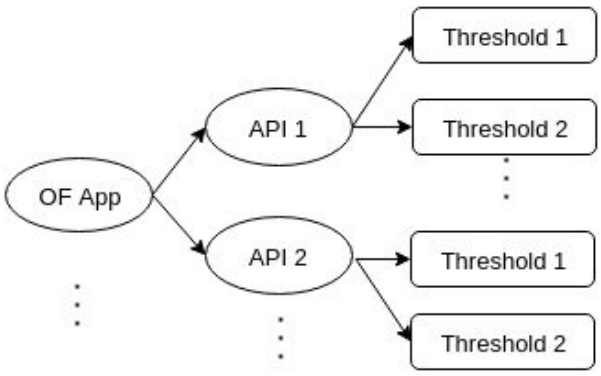


Fig. 3.8 Data structure interpreted from the high-level policy

DAC repacks the services provided by OpenDaylight and secures them with authorization and accounting. The services calling time comparison (millisecond) between OpenDaylight and Controller DAC.

- API 1 : GET method for obtaining network topology(/network-topology:network-topology)
- API 2 : GET method for obtaining flow entries in table 0 of OF switch 1 (/opendaylight-inventory:nodes)
- API 3 : POST method for adding a new flow entry(/sal-flow:add-flow)
- API 4 : PUT method for updating a flow entry(/sal-flow:update-flow)
- API 5 : POST method for removing a flow entry (/sal-flow:remove-flow)

We use Java spark framework as RESTful API server and JS as client to call these APIs. The latency caused by Controller DAC is quite negligible, from 0.061% to 0.446%. The same design principle can be applied in other SDN controllers.

3.6 Complexity analysis

As shown in Figure 3.4, all the policies are passed in the tree with 3 levels as depicted in Figure 3.8. The first level is root(OF app), the second level is APIs, and the third level is the threshold of API request times per second, per minute, or a special time interval (e.g., 100 seconds). The `ofapp1` in Figure 3.4 is configured in the api-role `SECURITY`. In the `SECURITY` role, the APIs "`/get/topo`" and "`/update/flowentry`" are limited to be called 100 times per 60 seconds and 10000 times per 3600 seconds. The 2 APIs will be mapped to the second level in Figure 3.8.

Finally, the request thresholds are mapped to the third level of the searching tree. As we store this tree with 2 levels HashMap for speeding up the search. The first level HashMap is to map the network application to APIs (identified by `AppHashMap` here) , and the second `HashMap` level is to map the APIs to the request thresholds(identified by `APIHashMap` here). Finally, the request thresholds as objects are stored in a ArrayList (identified by `ThresholdList` here).

Time complexity

When the *ControllerDAC* detects the requests from the network application, it will extract the name of the application with the request API. The application name will send to the `AppHashMap` and then find its `APIHashMap`. In this case, the time complexity is *O(1)* as we map the key directly to the value. However, when the key collision in the `HashMap`, the worst case of the data structure of `HashMap` in JVM is *O(n)*, *n* is the number of the variables, because JVM will transfer the data structure in self-balancing tree automatically. In this case, the worst case for `AppHashMap` would be *O(n)*, *n* is the number of the applications. Fortunately, there is a very rare chance that this kind of collision will happen. In the default configuration, we initialize the entity with the default capacity 16, i.e., 2 power 16 (65536). For example, if we have 100 network applications needed to be detect the conflict in *Controller DAC*, the chance to collide is 0.15%. The same, the `APIHashMap` will map the name of request API (as key here) to the `ThresholdList` (as value here), in this case the time complexity is also *O(1)* and *O(m)*, *m* is the number of APIs. Finally, the *ControllerDAC* uses a loop to detect whether the requests are over the threshold configured in the `ThresholdList` and makes the time complexity *O(k)*, *k* is the number of request thresholds configured in the 3.4 for each API. Totally, the time complexity is *O(k)* (we ignore the two time complexity with constants, i.e., the two *O(1)* for the searching time in two HashMap `AppHashMap` and `APIHashMap`) and the worst case is *O(n*m*k)*. Usually, we do not configure too many request thresholds. We define only the request threshold per second, per minute, per hour, or even more per day. Defining too many request thresholds does not make much sense. In this case, the *k* even can be considered as a constant. That is why we do not add the third level `HashMap` for mapping the time to the threshold.

Space complexity

By contrast, the two `AppHashMap` and `APIHashMap` give the space complexity *O(n)* and *O(m)* respectively, *n* is the number of the network applications and *m* is the number of the APIs. The `ThresholdList` also gives the space complexity *O(k)*, *k* is the number of request threshold. In this case, the space complexity is *O(m*n*k)*.

Obviously, our approach is to exchange time complexity with the space complexity, because the responding time is very critical for a SDN controller. We cannot delay the service to the data plane due to check the request times of the network applications. Fortunately, the memory space is always enough for our solution. For example, the most luxury memory requirement of a SDN controller, OpenDaylight, is 8G RAM, but memory of the server for running the SDN controller is always much higher than this, e.g., 16G RAM at least, or even more. Hence, hardware capacity is enough to support our solution.

3.7 Discussion

Obviously, the actual implementations of the SDN controllers are different from each other; however, SDN controller works as a network operating system, which includes the basic components such as internal data storage, built-in functions, core network services, and programmable interfaces [126] [101] [35] [94] [15] [93] [73] [103]. We agree with the proposition of the Rosemary controller [130] that a secure SDN controller should run only the essential network service for keeping its reliability; other network services should be decoupled from the SDN controller if possible. Hence, for securing the SDN controller from malicious network application, the SDN community should clarify as soon as possible the core network services provided by the SDN controller, such as OpenFlow messages processing, network topology providing, flow entry management as well as which network services can be provided by third party applications. The clarification of the basic network services provided by the controller help the design and implement of the SE-layer. We demonstrate the utility of the SE-layer for AAA control with low deployment complexity in this chapter. Basically, the services provided by the SE-layer are more than AAA, which can include as following.

Communication encryption. The shortage of supporting SSL, i.e. HTTPS for network application, makes communication in the northbound interface at risk of being tampered or eavesdropped. Controllers like Floodlight, ONOS and OpenDaylight support HTTPS. Ryu does not and OpenMUL does so only partially [101] [95] [35] [94] [78]. To protect the controller from man-in-the-middle attack, encryption is one of the popular solutions to secure the northbound interface for RESTful API. The SE-layer, working as a proxy, is able to encrypt the communication between network application and SE-layer by using TLS, i.e., HTTPS for RESTful API independently from controller, even if the SDN controller doesn't secure the northbound interface natively.

Rule conflict resolution. In SDN architecture, forwarding decisions are flow-based, which is defined by a set of packet field values acting as a match (filter) criterion, with fields such as actions (instructions), priority, counter and timeout etc. However, a malicious network

	API 1	API 2	API 3	API 4	API 5
OpenDaylight	5219	64199	1297	1278	1510
Controller DAC	5223	64238	1311	1281	1516
Latency	4(0.077%)	39(0.061%)	4(0.321%)	3(0.235%)	6(0.397%)

Table 3.10 Performance overhead of *Controller DAC*

application can insert a flow rules to reach the unexpected destination, to block another legitimate service or to confuse the service provided by other network applications coexisting on the same SDN controller such as the action `set` and `priority` in the flow rule [109] [110] [147]. As each of attack based on the flow rule manipulation are specific, several different approaches are proposed to mitigate the security issues caused by malicious or misconfigured flow rules. The FlowChecker[50] system encodes OpenFlow flow tables into Binary Decision Diagrams (BDD) and uses model checking to verify security properties. Veriflow [23] is a real-time system that slices flow rules into equivalence classes to efficiently check for invariant property violations. However, the evaluation of FlowChecker and Veriflow do not consider the handling of set action commands as in FortNOX and SE-Floodlight [109] [110]. On top of VeriFlow, [115] provides a library to verify correctness properties for network applications on several controller platforms. FlowGuard extends HSA [108] for rule-conflict resolution in the context of firewalls in order to build more robust firewalls in SDN environments. FLOVER [134], a model checking system based on the Yices SMT solver existing on the SDN App plane, verifies the instantiated flow rules which does not violate the network's predefined security policy. SRV checks the priority-bypassing attack by binding topology to check flow rules [147]; FortNOX and SE-Floodlight use the Alias Set Reduce(ASR) method to detect the rule conflict [109] [110]. Our SE-layer not only can keep the records about the network applications' operation history; moreover, it can keep the inserted flow rules from network applications, and uses the flow rule verification system to check the rule conflicts with various detection algorithms, such as SRV or ASR etc, in parallelism [109] [147]. Moreover, the records of request APIs can be use to trace and monitor the network application operation and furthermore recognizes the resource utilization of network application as the complementary protection for the SDN controller [32] [130]. In the next chapter, we will discuss the detection of the flow rule manipulation caused by the *priority* field in the flow rule.

More rich services for network applications The efforts such as described in [120] [144] and [39] show that the ability to notify events proactively (such as $flow\_mod$ or $port\_status$ updated messages) is useful for network app development. Unfortunately, few SDN controllers support this function. *Controller SEPA* can create this service by incorporating with the

	Authentication Authorization	Operation accounting	Information undisclosed	Isolation	Rule resolution	Encryption
Data tempering	✔	✔		✔		
Illegal functions calling	✔	✔		✔		
Malicious scanning			✔			
API abuse		✔				
MitM attack	✔					✔
Rule injection	✔				✔	

Table 3.11 Perspective security services provided by the SE-layer

frameworks such as SSE (Server Sent Events) or Websockets etc. In the next chapter, a message queue-based northbound interface can be used to notify the network events in a proactive way.

Finally, we resume the potential security services as the future work for the SE-layer in Table 3.11. However, one of the shortages of the RESTful API is that it is based on web service, which means it is not event-driven but client-server model. In this model, the network application cannot obtain the network event in a proactive fashion, but it needs to request the controller continuously for receiving the network event, e.g, *packet_in* message, in the real-time. This problem will be resolved in the next chapter by using a decomposable event-driven northbound interface.

3.8 Conclusion

The main concern, which prevents SDN from being widely adopted, is security. Specifically, the SDN controller opens a programmable interface to the third party when accessed by the malicious network application. The current proposals such as [120] [70] [144] [32] [126] uses the permission sets to control the app-to-control access. Unfortunately, the tasks of modifying all controller source codes or even scanning all the related functions for modifying and enhancing its security are unfeasible; even worse, the static access control is insufficient to secure the controller against API abuse. We propose a security-enhancing layer (SE-layer) to protect the data exchange between the controller and the network application. The prototype I,

Controller SEPA , protects the SDN controller in a flexible and efficient way. *Controller SEPA* can work well with OpenDaylight, ONOS, Floodlight, Ryu and POX with low deployment complexity. No modification of their source codes is required in their implementation while the overall security of the SDN controller is enhanced with negligible latency from less than 0.1% to 0.3%. *Controller SEPA* can provide rich services such as network application authentication, authorization (fine-grained), information undisclosed, and network application isolation. Furthermore, although several works have proposed to use permission set to secure SDN controllers, they cannot detect the API abuse with static permission control. Hence, our prototype II *Controller DAC*, enables the access control dynamic, i.e., the SE-layer can detect the API abuse by accounting the network application operation. In this way, the SE-layer can protect the SDN controller with AAA against the malicious operations from the external APIs. We demonstrated that *Controller DAC* works well with OpenDaylight with low deployment complexity. In particular, no modification of their source codes is required in their implementation, while the overall security of the SDN controller can be significantly enhanced. More than authentication, authorization, information undisclosed, and network application isolation, the SE-layer prototype II *Controller DAC* secures the SDN controller with dynamic access control with the performance overhead less than 0.5%. This SE-layer can hence enhance the security of the SDN controller in an efficient way with the AAA control.

Chapter 4

Security Enhancing Architecture for SDN

The notion of the SDN paradigm is the bestowal of network decision control to a globally intelligent view, hosted above the data plane. SDN enables this intelligence to be written in software as network applications using open APIs to better facilitate agile development and expedite network innovations. This is achieved by the SDN controller exposing the APIs to the data plane and the application plane, which is necessary to facilitate a wide range of network applications. Hence, the SDN controller offers a dramatic shift from the vendor specific proprietary network infrastructure to an open architecture. The revolution provides network application developers with programmable interfaces to manage the networking appliances on the data plane. Unfortunately, problems arisen from network application flaws, vulnerabilities, and malicious logic that may interfere with control layer operations have remained largely unaddressed for current SDN controllers [130]. Even worse, the diversity of the SDN controllers makes it harder to secure controllers from these malicious network applications [125]. As network events are continuous, the northbound interfaces should be message-driven for ensuring to process the network events in real-time. For this reason, the internal APIs like Java API and Python API as the northbound interfaces are widely adapted for listening the network events, i.e., OpenFlow messages, in the SDN architecture. But this design risks the controller from being injected by the malicious network application because these APIs, so-called internal APIs [39], cannot be decoupled from the controller. Although the northbound interface like RESTful API can be used to decouple the network application from the controller, but it is based on web service, i.e., server-client model, and does not support to listen the continuous network events (not message-driven). If the network application over requests the network events through the RESTful API, this will cause the network to suffer from a serious latency even packets dropping [145].

In this chapter, we discuss how to secure network application deployment in SDN architecture. We propose a security-enhancing architecture for SDN, dubbed *SENAD*, for securing

the network application deployment on the current SDN controller. *SENAD* architecture splits the well-known SDN controller into: (1) a data plane controller (DPC), and (2) an application plane controller (APC). The role of the DPC is dedicated for interpreting the network rules into OpenFlow entries and maintaining the communication with the data plane. The role of the APC is secured by design by providing the network applications authentication, access control, resource isolation, and application monitoring. We show that this approach can easily shield against the deny of service, caused by the resource exhaustion attack or the malicious command injection. The evaluation of our architecture shows that the *packet_in* messages take around 5 ms to be delivered from the data plane to the application plane on the long range. The work scope of this chapter includes:

- We propose a novel SDN architecture *SENAD* to securely deploy the network application. This architecture splits the SDN controller into (1) a data plane controller (DPC) and (2) a application plane controller (APC), and then secures the APC by design. This split will allow the DPC to only interpret the network rules into an OpenFlow entries and to maintain the communication with the data plane. In the other hand, the APC will play the role of a secured-by-design runtime to the network applications.
- We implemented the *SENAD* architecture based on the Floodlight controller and evaluated the performance of the novel architecture, which shows the preliminary results that the latency can be maintained under 5 milliseconds on the long range.
- We tested the feasibility of the *SENAD* architecture by deploying a network application as priority-bypassing attack detector. In this test, the *SENAD* architecture shows high compatibility with the current existing SDN controller (Floodlight).

4.1 Motivation

OpenFlow protocol, as one of the most popular SDN enablers, uses the "*packet_in*" message to inform the SDN controller to process the unknown packets and enforce the routing rule. As shown in Figure 4.1, once the data plane, i.e. OpenFlow switch, receives a new packets, OF switch will pack it into *packet_in* message and send it to the controller. The SDN controller will look for the business logic installed in the network application for enforcing the flow rules in the data plane for processing this packet. The "*packet_in*" message is a way for the switch to send a captured packet to the controller. There are two main reasons why the *packet_in* message is generated: a miss in the match tables or a `ttl` error, which results in asking an action to the controller for this packet. Hence, as the network packets are continuous, the

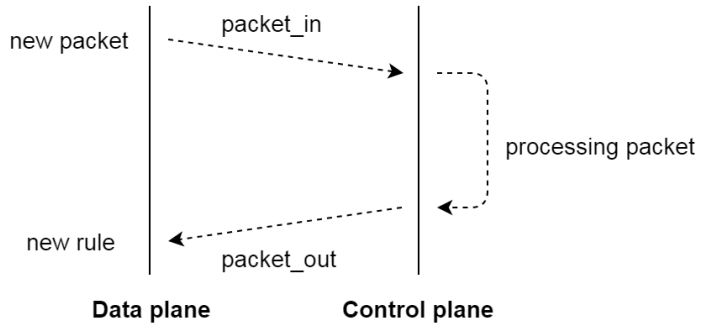


Fig. 4.1 *packet_in* message processing in OpenFlow

detection of the *packet_in* message should be message-driven. But the current decomposable northbound interface RESTful API (TCP/IP) is not message-driven. The controllers including OpenDaylight, ONOS, Floodlight and Ryu, hence adopt the internal APIs, i.e., native Java or Python APIs, to listen to *packet_in* message.

Even worse, one of the root causes of the fragility of the controllers is their tight coupling with applications. Thus, the effective approach is to separate the resource usage of controller from the network applications. This can be done by the file system-based isolation [81], process-based isolation [130] [106], or deploying a shim-layer [144]. We discuss more details of these approaches as follows.

4.1.1 *YANC* file system

YANC [81], an SDN controller platform, adopts UNIX-like permission to separate network applications from exposing the network configuration and stating it as a file system. In *YANC*, network applications are separated by the UNIX-like file system from the SDN controller. Applications benefiting from the virtual file system (VFS) layer used to distribute file system, as well as the namespaces that are used to isolate applications with different views (e.g., slices). The high-level view of *YANC* file system is presented in Figure 4.2.

4.1.2 Rosemary controller proposition

Shin et al. proposed a SDN controller Rosemary [130], which separates network application from controller and each new network application in Rosemary is invoked as a new process. Rosemary's network application connects to the SDN controller process through the IPC (inter-process communication). Moreover, Rosemary compartmentalizes controller's kernel modules by separating not only the network applications, but also the modules in the controller. Hence, this design increases the robustness of a controller, because it only runs necessary services. The

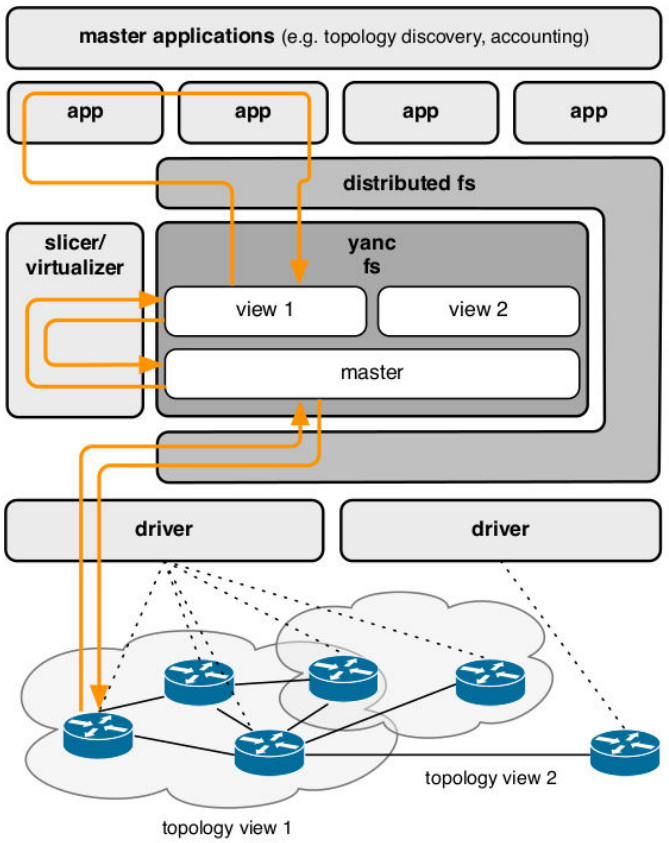


Fig. 4.2 *YANC* system architecture

services in the Rosemary kernel communicate with each other through an IPC channel, and the implication is that if a service crashes, other services are immune to this crash. In contrast, controllers such as Floodlight, NOX, and POX have implemented all necessary functions in a single zone. Rosemary architecture overview is presented in Figure 4.3. Rosemary provides diverse libraries for applications, and each application can choose necessary applications for its operations and spawns a new process for avoiding malicious network application from crashing controller unexpectedly. But this approach should develop a new security-driven SDN controller besides the current existing controllers. Hence, Rosemary controller secure the controller's runtime based on process but without defining the access rights based on user, application, and file on the system. For example, for hardening co-existing applications running on Linux, SELinux even defines more fine-grained the access and transition rights of every user, application, process, and file on the system [106]. Therefore, this motivates us to find a solution which can be widely adopted by the current SDN controllers without high deployment cost and complexity for hardening the SDN controller.

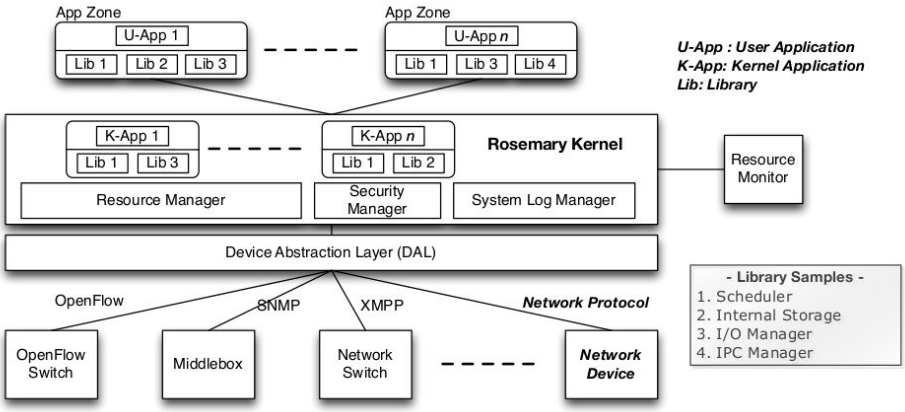


Fig. 4.3 Rosemary controller architecture

4.1.3 System shim-layer isolation

The isolation mechanism used in [144] proposes a shim layer between the OS and the network application to forbid the undesirable interaction between the applications and the OS. In order to provide such isolation, *PermOF* proposes an isolation framework as depicted in Figure 4.4. In the proposed system, controller and applications are isolated in thread containers. On one hand, applications are isolated from controller kernel and applications cannot call any kernel procedures or directly refer to kernel memory. *PermOF* achieves it by carefully craft the kernel code, so that applications are not able to obtain object reference from the kernel memory. This shim layer should be configured and controlled by the controller kernel, so that undesirable interaction between the applications and the controller's runtime will be cut off. This requires also to modify the dynamic library of the programming language or the OS. As to do so, the code of the system which hosts the SDN controller should be modified, but there is no real implementation for this shim layer in [144]. We cannot really estimate the cost and the feasibility for implementing this SDN controller specific shim layer. Instead, we simplify this work by putting the network application in a sandbox. Thank to the container technology, this can be done by importing the container, e.g., Docker, which contains a component called containerd-shim which sits between containerd and runC in Docker runtime for isolating the network application from the OS resource.

4.2 Security issues of network application injection

Alough the work such as [98] and [100] proposes the principles for securing the SDN controller, specifically the app-to-control threats. However, implementing these security principles remains

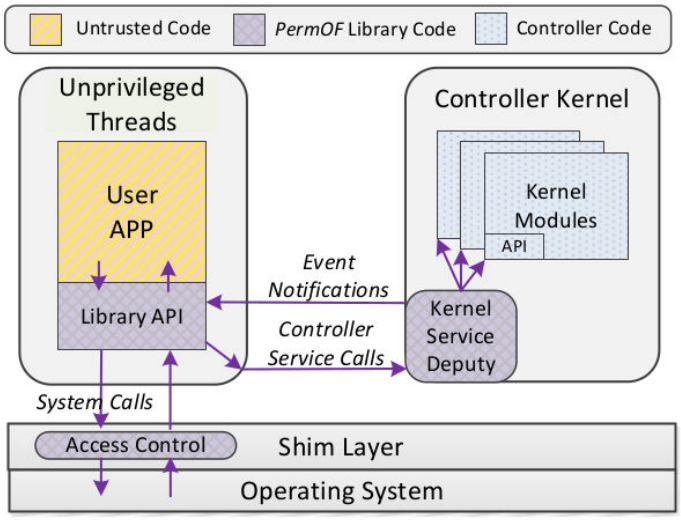


Fig. 4.4 *PermOF* framework architecture

still an unaddressed challenge. One of the main reasons is that the network event is continuously, ie., the OpenFlow protocol uses the *packet_in* message to inform the controller of the network event. However, the current most popular decomposable northbound interface is the RESTful API, which is not message-driven but is based on the server-client model. Hence, mainstream current controllers use internal APIs to listen to network events but this makes the controller vulnerable to the malicious network injection [130]. We will discuss two threats due to the network application injection in this section.

4.2.1 Resource exhaustion

The SDN controller functioning as a networking operating system allows the installation of the networking application to smarten up the network. Unfortunately, SDN controllers including OpenDaylight, ONOS, Floodlight and Ryu, are monolithic, running all the networking applications in a single process. In this case, a malicious network application can easily crash the SDN controller by exhausting the resource of controller's runtime. [130] chooses Floodlight and OpenDaylight as the main target SDN controllers and runs a malicious code in the network application on the SDN controller to evaluate the robustness and the security of the controller.

Floodlight Case Study This test application simply calls an exit or return (with `exit()`) function when Floodlight runs. Figure 4.5 shows the result: when a buggy application accidentally calls an exit function, the Floodlight instance is also killed. Second test is that this malicious application creates multiple memory objects for allocating memory space. The Floodlight instance does not limit memory allocations by its applications, finally resulting in

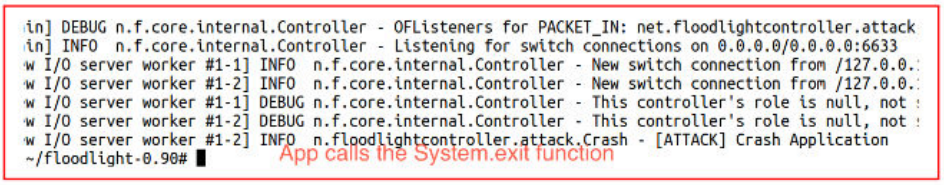


Fig. 4.5 Floodlight crash result

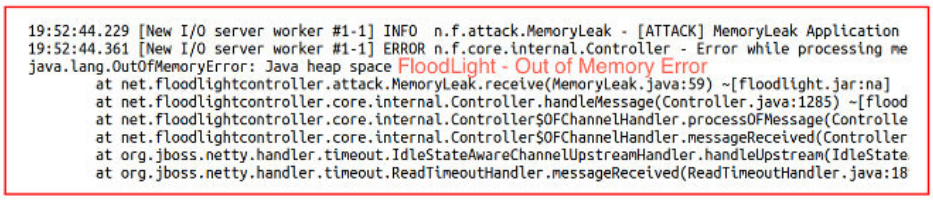


Fig. 4.6 Floodlight memory leakage result

the controller (technically, the JVM) crashing with an out of memory error as shown in Figure 4.6.

OpenDaylight Case Study As OpenDaylight is also Java-based, the case of crashing a Java-based controller can be done by calling a system exit function. The test result is shown in Figure 4.7. OpenDaylight has similar robustness issues to the Floodlight instance.

Hence, from the examples above, we saw that to crash a controller can be done by creating multiple objects or running infinite loops, invoking the system-level function in the networking application such as `System.exit(0)` in JVM to stop the SDN controller. Although OpenDaylight and ONOS run in the OSGi container (karaf) to support the service dynamic import, the controller and the application both run in the same process. Floodlight, as a Java application, runs in a single process with multi-threading. The Ryu framework allows the network application module to inject, but still in a single process in runtime. This kind of monolithic system cannot be immune from these attacks. Rosemary [130] develops a multi-process SDN controller for mitigating these problems. Rosemary spawns network applications independently in a new process and also provides the services to control the resource usage of applications like resource utilization monitoring and limitation and an application permission set. In contrast, we propose an orchestrator to deploy the network application upon the existing SDN controllers instead of developing a new one.

4.2.2 Configuration manipulation

Another security issue due to the malicious network injection is the configuration manipulation of the SDN controller or other network applications. Generally, the request processing from the

network application in the SDN controller (specifically, based on the OpenFlow protocol) can be classified into three types of control: (i) symmetric control flow operations, (ii) asymmetric control flow operations, and (iii) intra-controller control flow operations [127]. Hence, a malicious application can interrupt the service chain as the data transfers between two network applications [22]. Even more, a malicious application can manipulate the network configuration via the built-in function provided by the SDN controller [126]. For example, ONOS allows users to dynamically and programmatically configure various ONOS components via the built-in function `ComponentConfigService` to enhance the configurability of ONOS. This feature is especially useful when users need to adjust the properties of various ONOS components (e.g., threshold values, switches to enable certain features) to fulfill different network requirements. Meanwhile, this feature introduces a new security threat to the control plane. If a malicious logic changes this configuration, this will completely change the network behavior. Similarly, in the SDN controller Floodlight, if a malicious application modifies the *PACKET_OUT_ONLY* parameter value to be set true, the overall performance of the target network will become degraded [126]. Obviously, several research works such as [144] [120] [70] [22] adopts the application-based access control. Yet, the diversity of the SDN controllers makes this infeasible for some projects [125]. For example, the permission control in [120] [70] are implemented by assigning permission verification to the related functions. However, it may be infeasible to secure the controller's functions by modifying every function in the source codes. In the worst scenario, it would need to scan all the related functions (methods). One example is that in order to control the topology information, OperationCheckpoint needs to modify the codes of the two methods: `getAllSwitchMap` and `getLinks` in two different classes `Controller.java` and `LinkDiscoverManager.java`, respectively, since both of them provide the topology related information in Floodlight. In some SDN controllers, like OpenDaylight and ONOS, it is very hard to scan and find out all these related-methods and harden them. Hence, we adopt an approach to centralize the access control in the orchestrator(Service Broker module) instead of scanning the source codes for reducing the deploying complexity.

4.2.3 Threats from the external network application

Differing from the internal network application threats as discussing above, the external network application is separated from the runtime of a SDN controller and communicates to each other through network, such as RESTful APIs. However, as the network events are continuous, if a network application intends to detect network events such as host migration or evaluate QoS/QoE of a streaming data or provide real-time troubleshooting service etc, the RESTful APIs cannot receive the network events in real-time unless it keeps requesting the control plane to update events occurring in the network. However, the RESTful-based or RPC-based

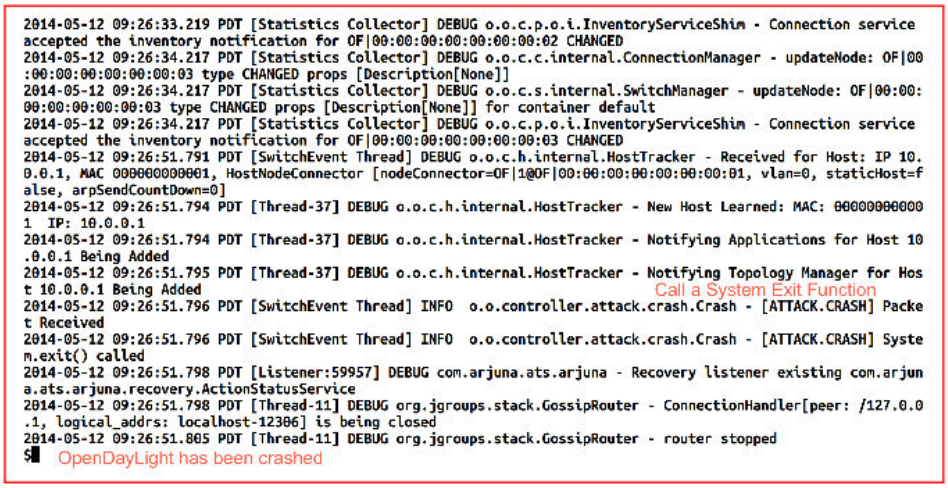


Fig. 4.7 OpenDaylight crash result

application uses the server-client model to listen to the network events. This means the network application needs to continuously call the SDN controller to obtain the network events in real time. Unfortunately, the over request from the network application causes the controller to overload, which increases the latency, and even drops packets in the data plane. Even worse, the RESTful APIs can be manipulated to flush out or modify flow entries with higher priority with lower priority flow entries in the case of poor access control [145]. Hence, implementing the network application as the external instance can not mitigate the API abuse problem evoked by the network application.

4.3 Design principle of the *SENAD* architecture

As we have discussed in Section 4.2, we aim to secure network applications deployment in a generic approach for the current existing SDN controllers. We split the SDN controller into the data plane controller (DPC) and the application plane controller (APC) as illustrated in Figure 4.8. The DPC is dedicated to communicating the data plane, interpreting the network rules to OpenFlow entries, and exchanging data with the APC. The APC works as the runtime of the network application with the security-by-design principle, including application authentication, request authorization, and resource isolation, control and monitoring. In the following, we will present the main components of the APC. The high-level view of the APC is illustrated in Figure 4.9. In the following, we will discuss the main components of the *SENAD* architecture.

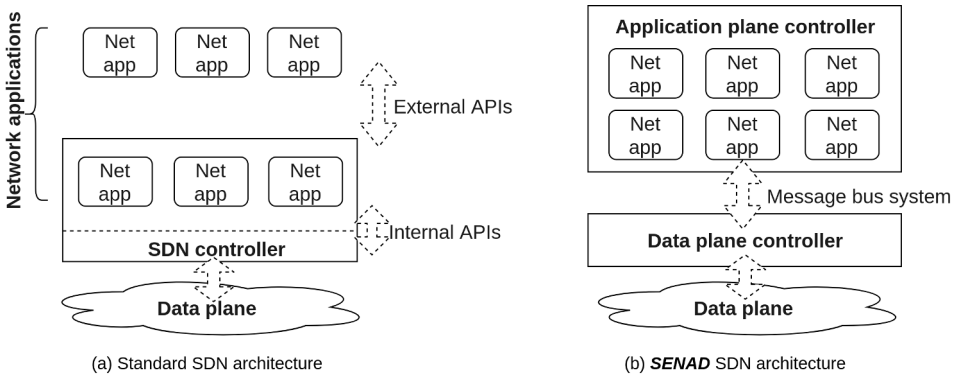


Fig. 4.8 Standard SDN architecture versus *SENAD* SDN architecture

4.3.1 Controller agent

The controller agents are used for the interaction between the APC and the DPC. Instead of the server-client model required by the RESTful APIs, we adopt the publish/subscribe model based on message bus system. This system allows the controller agent in the DPC to push the network events to the subscribers without waiting for the request from the network application. The authorized network application can also publish the data to the DPC through the APC agent. As we can configure to publish the network events immediately or set a buffer and linger time based on message bus system, this enables the network application to become decomposable from the runtime of a SDN controller and run in an application sandbox (Section 4.3.3). Furthermore, as the message bus system can push the network events in real time to avoid the API abuse as discussed in Section 4.2.3.

4.3.2 Policy engine

The policy engine consists of high-level policies and a policy parser. Its purpose is to define the resource control of each network application as well as the access control of the network application from the high-level. The high-level policy can configure the resources such as CPU number, memory, storage allocated to a network application. This information will be sent to the resource allocator (Section 4.3.3) by the parser. The service permission configuration in the high-level polices will be passed by the parser to the authorization module (Section 4.3.4). In the high-level policy, we can define the access control for the data exchange between the network applications and the controllers(DPC and APC), and the resource usage by the network application. We provide a non-exclusive permission sets in Table 4.1 because the permission design is outside scope of this work. More discussion of the permission set design can be found specifically in the works [144], [70] and [22].

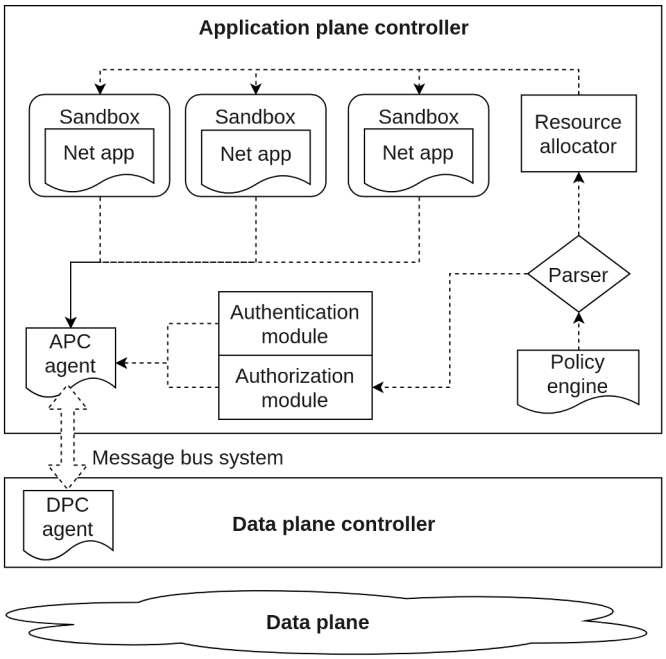


Fig. 4.9 High-level view of the APC

The policy includes access to the service of each network application with the parameters readable services and writable services. The readable services indicate from which service this application can receive the network events. If this application is allowed to send data into a service, it should be identified in the writable service permission. The service `PacketInMsg`, for instance, means the permission set for allowing network application to receive the *packet_in* message. `PacketOutMsg` is used to authorize and send the *packet_out* message. `ProactiveFlowEntry` is used to enforce the proactive flow entry, i.e., *flow_mod* message in OpenFlow. `ReactiveFlowEntry` is used to insert the reactive flow entry via *packet_out* message. `DeviceInfo` and `LinkInfo` provide the host information and switch connection information respectively. `Statistics` permission is used to report the traffic passing through an OpenFlow switch. `Payload` permission means to read the payload of a packet and `Configuration` is used to configure the controller setting.

An example of high-level policy defined in YAML can be found in Figure 4.3.2. In the policy example we can find the configuration of the resource usage of the network application. For instance, the CPU configured as 0.2 means that it can be configured from a part of CPU usage, e.g., 20% of CPU usage, to the full cores of the server. More than CPU resource control, the memory usage and the networking to the controller's host can be configured in the high-level policy, i.e., memory and networking respectively. The networking false means that this application cannot rout to the controller host by default and all the data exchange should pass through the APC agent. In this configuration, network application app1 is authorized to

Permission	Service
READ/WRITE	Hello
READ/WRITE	Error
READ	PacketIn
WRITE	PacketOut
READ	FlowRemoved
READ/WRITE	DeviceInfo
READ/WRITE	HostInfo
READ/WRITE	ProactiveFlowEntry
READ/WRITE	ReactiveFlowEntry
READ	Statistics
READ	Payload
READ/WRITE	Configuration

Table 4.1 Non-exclusive permission sets for Authorization module

get the data from the services DeviceInfo, LinkInfo, Statistics, and ProactiveFlowEntry, and can write new flow rule (ProactiveFlowEntry).

We define a non-exclusive access control list for *SENAD* architecture as it is out of scope of this chapter. More discussions of permission set can be found in [144], [70] and [22]. For example, `FlowRemoved` in Table 4.1 is sent to the controller by the switch when a flow entry in a flow table is removed. It can happen when a timeout occurs, either due to inactivity or hard timeout. An idle timeout happens when no packets are matched in a period of time. A hard timeout happens when a certain period of time elapses, regardless of the number of matching packets. `DeviceInfo` and `LinkInfo` provide the host information and switch connection information respectively. `Statistics` reports the traffic passing through an OF switch. `Payload` means to read the payload of a packet. `Configuration` is used to configure the controller setting like determining how much of a packet will be shared with the controller [3].

The access control configured in the *SENAD* architecture is dynamic, which means the Authorization module can update the access control on the fly. Hence, the access control can be managed in the *SENAD* architecture instead of scanning functions in the controller as [120] proposes. As the permission sets are configured in this way, the access control can be dynamic and interactive. This means the access control can be updated on the fly.

Furthermore, the RESTful API, based on web service, acts as a server to many clients. Figure 4.11 shows the comparison of the data delivering system between web service-based and message bus-based(or message queue) northbound interface. For the web service-based northbound interface, each network application request should pass through to the controller,

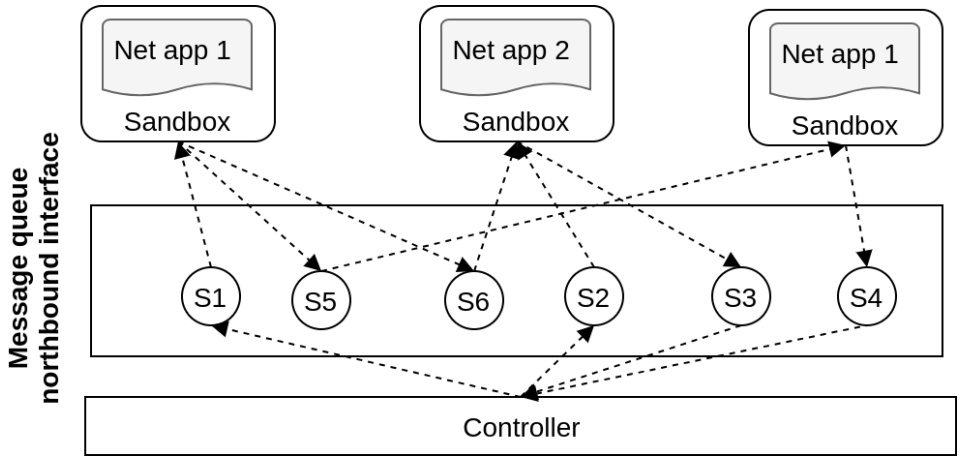
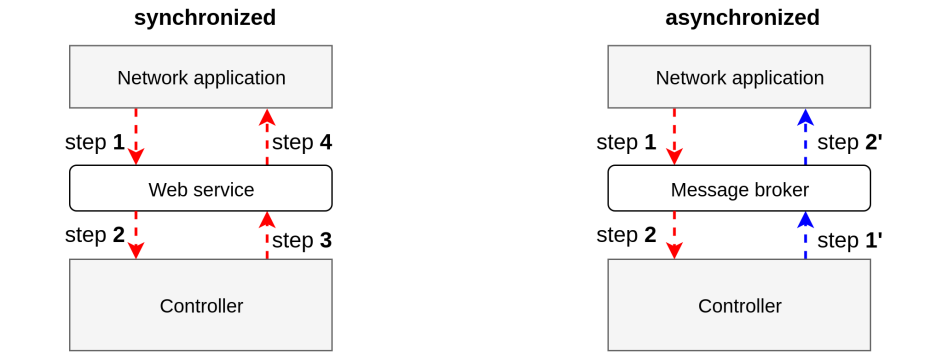


Fig. 4.10 Message bus-based northbound interface services interaction

and then the controller queries the data base or requests the data plane to find the data. Even the requests amongst the network applications, the request should pass through the controller. This model increases the loading to process the unnecessary requests (the requests between two network applications). In this model, if the server fails, the client must take the responsibility to handle the error. Similarly when the server is working again, the client is responsible to resend the request. If the server gives a response to the call and the client fails to receive the request, then the operation is lost. In this case, it is easy for the client to lose some events occurring in the network. Moreover, as it is hard to control the number of client calls on the web service, a huge demand on one server at a given time can result in server shut down (i.e., the DoS/DDoS attack). Although we can expect an immediate response from the server, we still handle asynchronous calls by using callback to request the service in some cases. A message queue like RabbitMQ, ActiveMQ, or Apache Kafka, can have different and more fault-tolerant results. If the server fails, the message persists in the queue (even in machine shutdown). When the server is working again, the server receives the pending message. If the server gives a response to the call and the client fails, the message also persists. For the message queue, we can decide how many requests the server handles by avoiding over requests. Although the data from the server is not synchronous as it is event-based API, we can still implement/simulate synchronous calls by keeping the data in local cache. Moreover, the controller can be free from the requests between the network applications as it can be done by the Service Broker to mediate the requests in the application plane.



(a) Web service-based northbound interface

(b) Message bus-based northbound interface

Fig. 4.11 Web service-based vs message bus-based northbound interface

4.3.3 Application sandbox and resource allocator

Differing from the monolithic SDN controller, the network application is deployed in a sandbox, which isolates the resources and process from the runtime of a SDN controller. The resource allocator allocates the resource to the application sandbox based on the policy passed from the policy engine. One or several business logics to manage the SDN-enabled network can be deployed in a sandbox as the network application. The interaction between the applications or between the controller and network application is also constrained by the policy defined in the high-level policy (permission set), which is controlled by the APC agent with the authentication and authorization modules (Section 4.3.4). Even more, the resource allocator provides the resource usage monitoring service, which posts the three levels of resource usage on the dashboard: (1) the system-level resource usage, (2)the JVM-level resource usage, and (3)the application sandbox-level resource usage for detecting the DoS attack caused by the resource exhaustion attack. The resource monitoring service is used to track and recognize consumption patterns that may violate resource utilization policies imposed upon the application. The system-level monitoring metric includes the totality of the controller process threads and priority, CPU usage, memory usage, and network traffic (download/upload). The application-level monitoring includes the allocation of CPUs, memory, and storage to the network application. The JVM-level monitoring service reveals the Java-based controller, e.g., Floodlight, resource consummation on the JVM runtime. This information should be kept traceable and updating for detecting the abnormal operations from the network applications.

4.3.4 Authentication and Authorization modules

The APC agent is used to exchange data between the DPC and the network applications. As every data exchange, including the exchange between the network applications and the exchange

```
- appname: app1
  cpus: 0.2
  memory: 128 MB
  storage: 100 MB
  networking: false
  readableServices:
   - DeviceInfo
   - LinkInfo
   - Statistics
   - ProactiveFlowEntry
  writableServices:
   - ProactiveFlowEntry
```

Fig. 4.12 YAML-based policy example for deployment in *SENAD* architecture

between a network application and DPC, should pass through the APC agent. In this case, the APC agent co-works with the authentication module and the authorization module to provide access control service. We adopt the password-based authentication and the authorization is based on the rules parsed from the policy engine.

4.3.5 The workflow of *SENAD* architecture

The workflow of this architecture is summarized as the following:

- Define the high-level policy by the network administrator.
- The Parser will translate the high-level policy to Resource Allocator and the Authorization modules.
- Resource allocator deploys the application in the sandbox under the constraint of policy.
- Controller APIs exchange the OpenFlow messages between the DPC and APC.
- Authorization module authorizes every message exchange between the DPC and APC.

Hence, the resource control is deployed by the resource allocator, which executes policy passed from the parser and deploys the application sandbox with the configured rule. The authentication and authorization modules collaborate with the APC controller agent to verify every data request from DPC to APC.

4.4 Implementation

We implemented the prototype based on *SENAD* architecture with the SDN controller Floodlight. The Floodlight controller here functions as the DPC in our architecture, and we adopt the

message bus system Apache Kafka to play the role as APC. Apache Kafka supports reading and writing streams of data like a messaging system. The data publisher and the data subscriber are called Kafka producer and Kafka consumer respectively. The Kafka producer generates the data and then pushes it to the Kafka broker (instead of to consumer directly), and then the broker sends the data to the data consumer. The Kafka producer and Kafka consumer exchange the data by registering a common *topic*. This means only the producer and consumer who share the same Kafka topic can exchange the data. In this case, the DPC agent is implemented by a Kafka producer who pushes the data to the *PacketInMsg* topic and a Kafka consumer who receives the data from the *PacketOutMsg* topic. The network application can register the *PacketInMsg* topic as a data consumer for the receiving of the *packet_in* message. Once the *packet_in* messages terminates the processing in the network application, the network application can send the results, e.g., network rules, to the specified topics. For instance, if the result is sent to the *PacketOutMsg* topic, the DPC will receive it and then interpret it into the OpenFlow entry.

For authenticating and authorizing the data exchange, the truststore of Apache Kafka is configured to authenticate and authorize the topic requested by a producer/consumer, i.e., the network application in our case. The Kafka server will check the permissions of the consumer and the producer to determine which one can read/write the data from a topic. In this case, only the authorized service provider/receiver can push/fetch the data to/from the APC agent. The rules in the authorization module is parsed from the high-level policy (Section 4.12). Figure 4.14(c) depicts the ACLs translated from the high-level policy to the truststore of Apache Kafka. In this case, Figure 4.15 shows the access control of the APC agent. In our configuration, the network application can only access to the service *DeviceInfo* but cannot read the controller configuration (permission *Configuration*), so the application can receive the *DeviceInfo* service data(source IP/MAC and destination IP/MAC), but is refused to connect to *Configuration* service by replying `UNKOWN TOPIC`.

The configurations in the policy engine including CPU, memory, and storage will be parsed to the resource allocator, which applies the resource allocation on the application sandbox. The container technology is a merged paradigm for the service deployment such as Docker, which uses `cgroup` (control group) and `namespace` in the Linux kernel-level to isolate the resource of the container from the host. Hence, we leverage the Docker container as the network application sandbox in the *SENAD* architecture. Apache Mesos uses Linux *cgroups* to provide isolation for CPU, memory, I/O and file system and provides applications (e.g., Hadoop, Marathon, Kafka) with APIs for resource management. In this case, Apache Mesos works as the resource allocator in collaborating with Marathon to control the resource usage of the network application sandbox, i.e., Docker container. When the high-level policy is

Prototype components	Implementation
DPC	Floodlight(v1.2)
DPC agent	Kafka producer/consumer
APC agent	Kafka broker
Authentication module	Kafka truststore
Authorization module	Kafka truststore
Resource allocator	Marathon, Sigar
Application sandbox	Docker container
High-level policy	YAML
Parser	Java application
Network simulator	mininet

Table 4.2 Prototype implementation

written in YAML, the parser translates it in JSON for allowing Marathon to deploy the network application in the sandbox(Docker) as depicted in Figure 4.14(b). Notably, the networking option in the policy of Figure 4.14(a) is parsed as the firewall rules in the `iptables` to disable the routing from the Docker container to the APC. The Java package Sigar is used to monitor the resource usage, including CPU, memory, and traffic, of the network applications and post it on the a Javascript-based dashboard to visualize the resource consuming. The components of our prototype implementation are resumed in Table 4.2. More implementation details will be explained as follows.

4.4.1 Controller agent

In an OpenFlow SDN, when a switch receives a packet on a port, it will try to match the packet to a flow entry in the switch's default flow table. If the switch cannot locate a flow that matches the packet, it will by default send the packet to the controller as a *packet_in* for closer inspection and processing. In Floodlight, we create a new module and override the receive command to listen to the *packet_in* message with the API `OFMessage.getType()`. As shown in Figure 4.13, when this agent receives the *packet_in* message, it can process the payload of this packet, including the layer two data like packet type Ethernet or ARP, layer three data like IPv4 or IPv6, even the layer four type like TCP or UDP, and then sends to the *packet_in* consumer(s). We implemented the controller agents by using Kafka producer and Kafka consumer to send and receive messages respectively.

Apache Kafka is an open-source, distributed and real-time streaming data processing platform developed by the Apache Software Foundation written in Scala and Java. It adopts the publish/subscribe system to retrieve the records; in this respect, it is similar to a message queue

```
@Override
public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx)  {

        switch (msg.getType()) {

           case PACKET_IN:

               Ethernet eth = IFloodlightProviderService.bcStore.get(cntx,
                               IFloodlightProviderService.CONTEXT_PI_PAYLOAD);
               IPv4 ipv4 = (IPv4) eth.getPayload();

               /* extract the data(payload) from the packets */
               PacketInProducer.sendMsg( /* send data for packet-in consumers */ );
```

Fig. 4.13 Code example for controller API agent implementation in Floodlight

messaging system. Kafka can run as a cluster on one or more servers. The Kafka cluster stores streams of records in categories called `topics`. The service publisher and subscriber in Kafka are called Producer and Consumer respectively.

- **Kafka Producer** allows an application to publish a stream of records to one or more Kafka topics in the message broker. The service provider deployed in the network application functions has the role of the Kafka producer to publish data to the topics in service broker.
- **Kafka Consumer** allows an application to subscribe to one or more topics and fetch the data from a Kafka message broker. Hence, the service receiver of a network application acts similarly the role of a Kafka Consumer to obtain the updated data from the service broker.

The advantages of choosing Kafka are (1) it uses the publisher/subscriber system. This means it is a message-driven message bus system and can exchange the OpenFlow messages in real time without request/reply model as server-client model. (2)Apache Kafka is based on TCP. In this case, the network applications can be decoupled completely from the runtime of the SDN controller and be deployed in a sandbox.

4.4.2 Policy Engine

The example of high-level policy in YAML is shown in Figure 4.12, which can configure the resource control, access control and network routing. The resource control consists of CPU, memory and storage. The access control is identified by the permission to write and read the service provided by the SDN controller or the network application. The network routing is blocked by default between the network application and the controller, i.e., networking:false in policy. Figure 4.14 depicts the parser translating the high-level policy to the Resource Allocator (Marathon) and Service Broker (Kafka) from the policy in Figure 4.12.

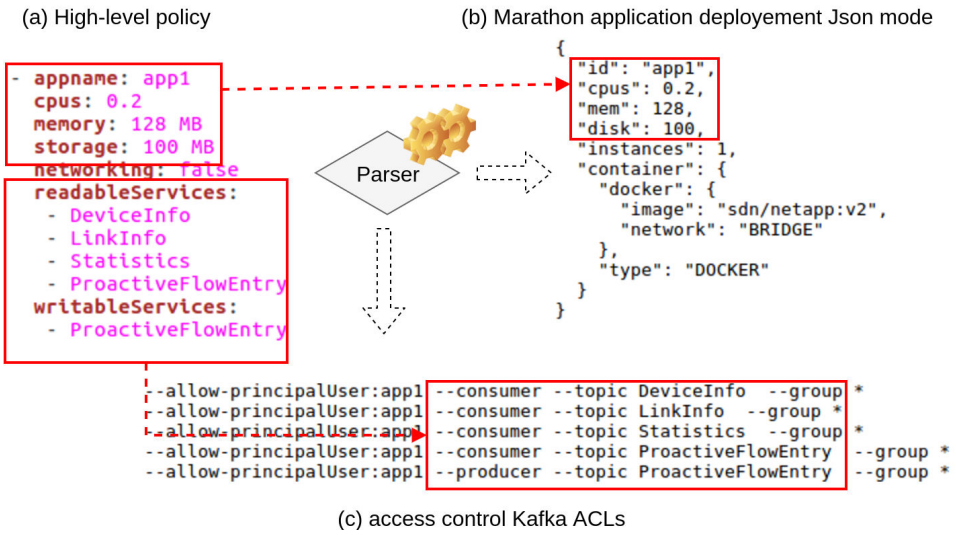


Fig. 4.14 *SENAD* parser implementation

The resource control policy will be sent to the resource allocator in json format if Marathon is used to deploy the network application. The parser needs to extract the access between the services to the message broker of Kafka as ACLs (Access Control List). The options networking will be passed into `iptables` as a rule to control the routing between the controller and the network application. The role will be used in other network applications in case of of the need to allocate the priority of flow entries.

4.4.3 Authentication and Authorization modules

The truststore of Apache Kafka is configured to authenticate the topic requested by a user, i.e., network application in our case. The Kafka server authenticates the network applications(password-based) and checks the permissions of the consumer (service receiver) and the producer (service provider) to decide which can read/write the data from a service topic. In this case, only the service provider/receiver is authorized to register a service with the certificate can push/fetch the data from the server. The service permission is centralized to configure in the Kafka's message broker, i.e., the APC agent in our implementation. The automatic topics creation should be disabled so that the network application will not create a topic to send/receive data without the validation of the Authorization module. Figure 4.15 shows the access control authorized by the APC agent: (a) is the firewall rule configuration in the `iptables` of controller host, which disables the native external API access (we configure the default port 8080 to 8082) but only opens the port 9092 for data exchange via the service broker(Kafka). Hence, (b) shows that network application cannot access to the SDN controller through RESTful API and always waits the response. The (c) is the data exchange through the APC agent. As in

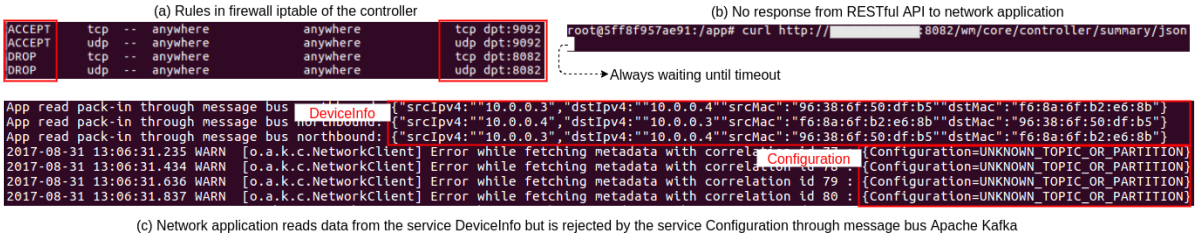


Fig. 4.15 Access control provided by the APC agent with Authorization module

our configuration, the network application can only access to the service `DeviceInfo` but cannot read the controller configuration(permission Configuration), so that the application can receive the `DeviceInfo` service data(source IP/MAC and destination IP/MAC), but is refused to connect to Configuration service.

4.4.4 Resource Allocator and Application Sandbox

Container technology is a merged paradigm for the service deployment such as Docker, which uses cgroup (control group) and namespace in Linux kernel-level for isolating the resource of the application sandbox from the host. We test the effect of the same malicious application running directly on the SDN controller and in a container. The application contains a malicious loop, which keeps inserting data in a HashMap. Figure 4.16 shows that when this malicious loop runs directly on the controller runtime (JVM for the controller Floodlight), the SDN controller drops the packet until it cannot service the data plane (mininet) any more, i.e., drop completely the packets that the controller receives. The memory of JVM is configured as 512M, and the service is completely stopped after three minutes of launching the attack. Fortunately, the runtime of network application sandbox, contains a component called containerd-shim which sits between containerd and runC in Docker runtime for isolating the network application from the OS resource. Hence, when the same codes runs on the sandbox generated by the Policy Engine app1 (CPU 0.5 and 128 MB memory), the service provided by the controller to the data plane can be delivered successfully without being affected as shown in Figure 4.17.

4.4.5 Resource monitoring dashboard

We developed a Java program based on Apache Mesos APIs(API /tasks.json) and Java library Sigar for resource usage monitoring. We integrate the system-level monitoring results and network application resource allocation in the dashboard of the SDN controller Floodlight (/ui/index.html) that dynamically updates results. Figure 4.18 shows the three level resource usage monitoring: (i)system-level, (ii)JVM-level(Floodlight is Java-based), and (iii)application-

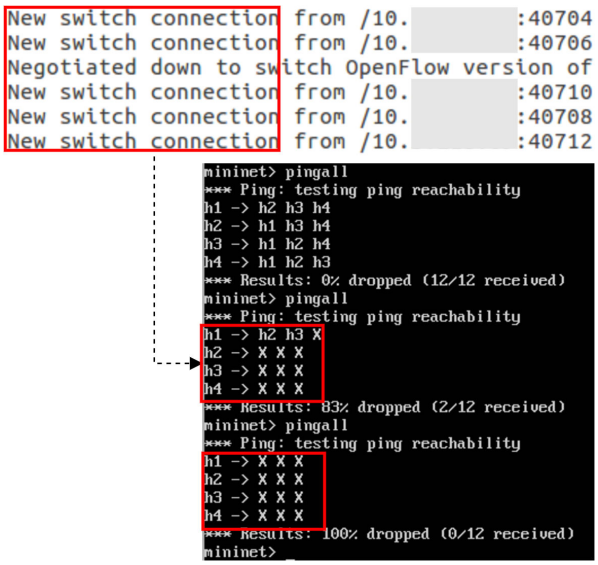
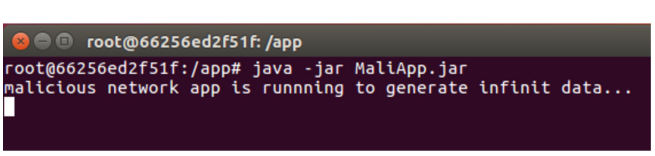
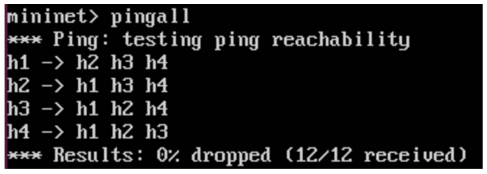


Fig. 4.16 Service delivering failure of the controller under attack



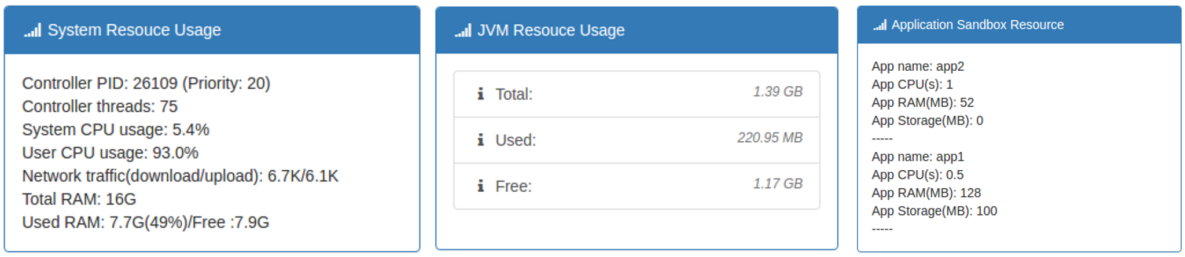
(a) Malicious application is running in the Docker container app1 generated by the Policy Engine



(b) Service delivery successfully of the SDN controller Floodlight to mininet under attack

Fig. 4.17 Service delivering successfully of the controller with sandbox protection

level. The system-level results include the controller PID with priority on the operating system, the treads running on the controller, the system CPU usage and user CPU usage, network traffic(download/upload), and RAM usage. The difference between system CPU usage and the user CPU usage is the time spent in user space or kernel space. User CPU time is time spent on the processor running your program's code (or code in libraries); system CPU time is the time spent running code in the operating system kernel on your program's behalf. The JVM-level is the overall JVM memory allocation (Figure 4.18 (b)). The application-level is the resource allocation(CPU, memory, and storage) to each network application deployed in the sandbox(Docker here).



(a) System-level resource usage (b) JVM-level resource usage (c) Application-level resource usage

Fig. 4.18 Resource usage monitoring dashboard of the SDN controller Floodlight

4.5 Network application deployment based on *SENAD* architecture

We tested the effect of the same malicious Java-based application running directly on the SDN controller and in a container. The application contains a malicious loop, which keeps inserting data in a HashMap. Figure 4.16 shows that when this malicious loop runs directly on the controller runtime (JVM for the controller Floodlight), the SDN controller drops the packet until it cannot service the data plane (mininet) any more, i.e., drop completely the packets that the controller receives. The memory of JVM is configured as 512M, and the service is completely stopped after three minutes of launching the attack. Fortunately, when the same codes run on the sandbox generated by the policy engine app1 (CPU 0.5 and 128 MB memory), the service provided by the controller to the data plane can be delivered successfully without being affected as shown in Figure 4.17.

Moreover, we implement a network application based on *SENAD* for evaluating its feasibility. We use this network application detect the priority-bypassing attack. For understanding the priority-bypassing attack, we explain first the flow rule selection process in the OpenFlow-based SDN, i.e., priority-based mechanism, and then exploit the vulnerabilities of this mechanism. The attack is used to exploit the priority-based mechanism in OpenFlow is called "priority-bypassing" attack in this thesis.

4.5.1 Introduction to priority-based mechanism in OpenFlow

The priority-based mechanism in OpenFlow is the flow rule selection process in an OpenFlow-based SDN. As SDN decouples the control plane from the data plane and becomes an external entity. The data plane consists of forwarding devices that perform a set of elementary operations. The forwarding devices have well-defined instruction sets (e.g., flow rules) used to take actions on the incoming packets (e.g., forward to specific ports, drop, forward to the controller, rewrite some headers, etc). These instructions are defined by southbound interfaces, ie., OpenFlow in

our use case, and are installed in the forwarding devices by the SDN controller. Forwarding devices are programmed by control plane elements through well-defined southbound interface embodiments.

The applications plane is the set network applications that leverage the functions offered by the northbound interface to implement network control and operation logic in control plane. The applications called network applications include the applications such as routing, firewalls,load balancers, monitoring, and so forth. The policies are ultimately translated to southbound-specific instructions that program the behavior of the forwarding devices. Despite the great benefits of OpenFlow-based SDN, it introduces many technical challenges for the rules applying such as policy conflict and policy inconsistency [109] [132] due to the `set` attribute for actions. The security threat identified here is due to the `priority` attribute in the flow rule. As a flow rule is broadly defined by a set of packet field values acting as a match (filter) criterion and a set of actions (instructions). Packet match fields are extracted from the packet. Packet match fields that are used for table lookups depend on the packet type, and typically include various packet header fields, such as Ethernet source address, IPv4 destination address or ingress port etc. The packet is matched against the table and only the highest priority flow entry that matches the packet must be selected. A flow rule selection example is as following.

Example. Existing two rules R_1 and R_2 in controller.

- $R_1 :\ if(src\_ip = 10.0.0.1)\ A = drop,\ P = higher$
- $R_2 :\ if(src\_mac = 11:11:11:11:11:11)\ A = allow,\ P = lower$

A and P mean *action* and *priority* respectively. The controller receives a *packet_in* message from $Host_A$. The configuration of $Host_A$:

- $IP = 10.0.0.1$
- $MAC = 11:11:11:11:11:11$

Once the SDN controller receives a *packet_in* message from $Host_A$, the controller will choose one rule to apply between R_1 and R_2, because the packet from $Host_A$ matches against at the same time R_1 and R_2 (R_1 matches against its IP and R_2 matches against the MAC address). But the two rules will execute two different actions, one is to drop the packet and another is to allow the packet. The controller will apply R_1 to the *packet_in* message for it has a higher

priority than the other. This is the priority-based mechanism. The nature of this mechanism is never changed from OpenFlow 1.0 [1] to OpenFlow 1.5 [2].

In OpenFlow 1.0, the header fields is 12-tuple; until OpenFlow 1.5, the header fields becomes 44-tuple. Because of the popularity and success of OpenFlow, most of the SDN controllers adopt it for southbound communication [101] [93] [35] [15] [95]. Under these conditions, the priority-based mechanism becomes more vulnerable. We will conduct the threat analysis in the next section.

4.5.2 Priority-bypassing attack

The northbound interface in SDN is still short of protocol to build up the communication between control layer and the network applications in applications layer. For example, Phillip Porras et al. proposed the role-based source authentication to recognize by default three authorization roles among those agents that produce flow rule insertion requests [109]. The first role is that of human administrators, whose rule insertion requests are assigned the highest priority. Second, security applications are assigned a separate authorization role. Flow insertion requests produced by security applications are assigned a flow rule priority below that of administrator-defined flow rules. Finally, non-security-related network applications are assigned the lowest priority. Roles are implemented through a digital signature scheme, in which FortNOX is preconfigured with the public keys of various rule insertion sources. If a legacy network application does not choose to sign its flow rules, those rules are assigned the default role and priority of a standard OpenFlow application [109] [110]. This design for assigning priority is legitimate. In order to enforce a flow rule in a current SDN controller, such as Floodlight, OpenDayLight and ONOS, they allow the system administrator via curl, a popular command-line tool for transferring data using various protocols, to interact with the REST API and to insert the policies into controller through digesting the rules inserted, remotely or locally, in the network apps with southbound interface OpenFlow [35] [101] [95]. We discuss about the security threat with this kind of usage scenario.

[1] Priority description OpenFlow 1.0: *Packets are matched against flow entries based on prioritization. An entry that specifies an exact match (i.e., it has no wildcards) is always the highest priority. All wildcard entries have a priority associated with them. Higher priority entries must match before lower priority ones. If multiple entries have the same priority, the switch is free to choose any ordering. Higher numbers have higher priorities.*

[2] Priority description in OpenFlow 1.5: *The packet is matched against the table and only the highest priority flow entry that matches the packet must be selected. The counters associated with the selected flow entry must be updated and the instruction set included in the selected flow entry must be applied. If there are multiple matching flow entries with the same highest priority, the selected flow entry is explicitly undefined. This case can only arise when a controller writer never sets the* `OFPFF_CHECK_OVERLAP` *bit on flow mod messages and adds overlapping entries.*

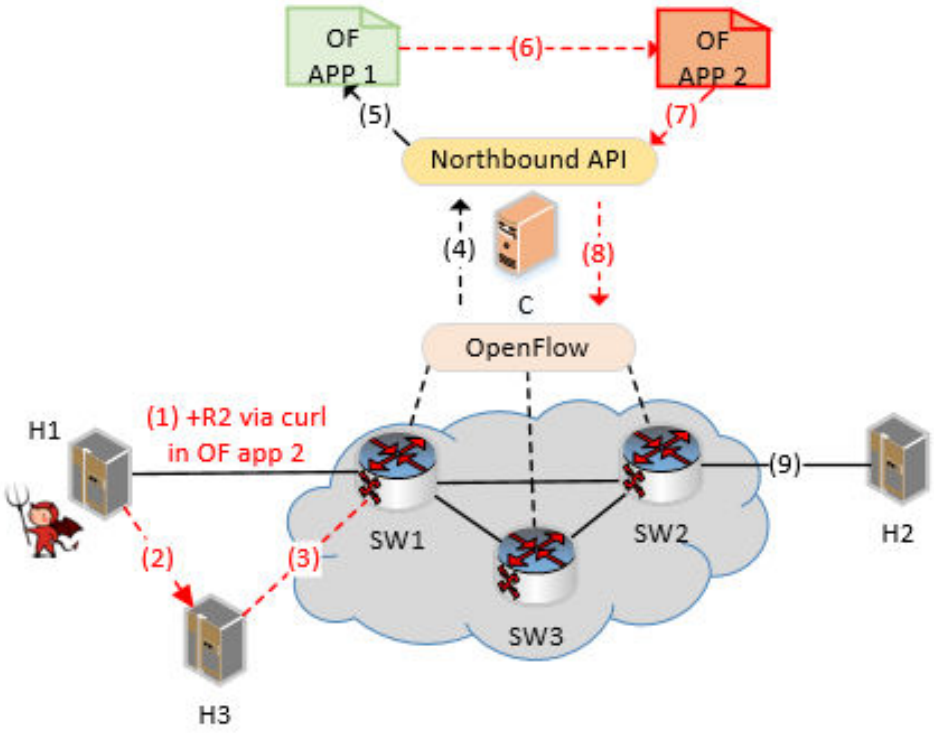


Fig. 4.19 Priority-bypassing attack model

Moreover, network application which is provided by a third-party developer can be installed on the applications plane. Most controllers encourage anyone who has an interest in this area to implement a network application by releasing open APIs [35] [43] [101] [15] [93]. This trend can be seen in the industry; for example, a network application store that has been launched by HP [5]. This makes it extremely difficult to verify the rules from network applications, ultimately leading the priority-based mechanism harder to control. We test the priority-bypassing attack in the controller Floodlight with the network configuration as following.

Scenario 1: IP/MAC-passing and VLANs-crossing by priority-passing attack.

Figure 4.19 is the attack model and the notation is as following:

- C ⇒ The SDN controller
- SW1 ⇒ OF switch 1
- SW2 ⇒ OF switch 2
- SW3 ⇒ OF switch 3
- H1 ⇒ Host 1(attacker), connecting with $port_2$ of SW_1

H1	$IP = 10.0.0.1$
H2	$IP = 10.0.0.2$
H3	$IP = 10.0.0.3$
R1	if(src_ip=10.0.0.1, dst_ip= 10.0.0.2) A=drop P=higher
R2	if(src_ip=10.0.0.3, dst_ip= 10.0.0.2) A=allow P=lower

Table 4.3 Configurations for IP-passing attack

- H2 $\Rightarrow$ Host 2(target), connecting with $port_2$ of SW_2
- H3 $\Rightarrow$ Host 3 (the host mutated from attacker)
- R1 $\Rightarrow$ A rule from administrator, with higher priority
- R2 $\Rightarrow$ A malicious rule from attacker, with even the lowest priority

The configurations of each role in this scenario is shown in Table 4.3.
The attack can be launched as the following steps:

1. Add a new rule R2, through a compromised network app or a compromised host via curl, even with a lowest priority, i.e., R2 : if(src_ip=10.0.0.3,dst_ip= 10.0.0.2) A=allow P=lowest
2. The attacker modifies IP to 10.0.0.3(H3).
3. The attacker sent a packet from IP=10.0.0.3(H3) to IP=10.0.0.2(H2).
4. When the packet arrive the OF switch, it is a new and unknown packet and the switch send to the controller for asking which rule to apply.
5. The controller search the rule from the highest priority to the lowest.
6. R1 doesn't match against for the IP address was no longer 10.0.0.1(H1) but 10.0.0.3(H3). The controller pass the request to the next rule.
7. R2 matches against this packet(IP=10.0.0.3) and be applied.
8. A *packet_out* message from controller to switch for applying R2 allows this packet(action of R2 is allow).
9. This packet is sent to the destination H2.

H1	$MAC = 11:11:11:11:11:11$
H2	$MAC = 22:22:22:22:22:22$
H3	$MAC = 33:33:33:33:33:33$
R1	if($src\_mac = 11:11:11:11:11:11$) A=drop P=higher
R2	if($src\_mac = 33:33:33:33:33:33$) A=allow P=lower

Table 4.4 Configurations for MAC-passing attack

H1	$MAC = 11:11:11:11:11:11$
H2	$MAC = 22:22:22:22:22:22$, $vlan = 2$
H3	$MAC = 33:33:33:33:33:33$
R1	if($MAC = 11:11:11:11:11:11$) A=set_vlan_id=1 P=higher
R2	if($MAC = 33:33:33:33:33:33$) A=set_vlan_id=2 P=lower

Table 4.5 Configurations for VLANs-crossing attack

This attack cannot be simply resolved by binding MAC or defining the white-list because the attacker can launch the attack after the machine has been authenticated. For example, in Linux, the attacker can use the command "`ifconfig eth0 hw ether 02:01:02:03:04:08`" to modify the MAC address easily. The `eth0` is name of the Ethernet card, the `02:01:02:03:04:08` is the new MAC address. The experiment proves that the attacker can pass the rules of higher priority successfully by modifying the MAC address. Moreover, the attacker can launch the attacker from the virtual machines, which have their own virtual IP addresses and virtual MAC addresses, the MAC-binding rules cannot match against the *packet_in* message from these virtual machines. In the scenario of modifying MAC address, the configurations for each role is in Table 4.4.

The attack can be launched by following the same steps in the last scenario with the configuration in Table 4.4. We successfully reach H2 by modifying the MAC from $MAC = 11:11:11:11:11:11$ to $33:33:33:33:33:33$ with the inserted lowest priority R2 from a compromised host or malicious network app.

Another potential attack caused by priority-passing is to cross the VLANs. The configuration for this network is in Table 4.5. H1 and H2 should not be in the same VLAN. H1 should be in vlan1 but H2 in vlan2. The R2, which is the rule from the attacker, will set the packet from $MAC = 33:33:33:33:33:33$ to vlan2, even it should belong to vlan1 because the packet in

fact is from the H1. The attacker might cross any VLAN even to the sensitive ones in this way [3].

Scenario 2: DoS by egressing actions The priority-based problem can cause DoS by following the same principle. R1 is the rule from administrator and R2 is the malicious rule. In this scenario, the administrator allows the packet from H1 to H2 with R1, but the attacker inserts a rule(R2).

- R1: if(src_ip = 10.0.0.1, dst_ip = 10.0.0.2) A=allow, P=higher
- R2: if(src_ip = 10.0.0.1, dst_ip = 10.0.0.2) A=set_src_ip = 11.0.0.1, P=lower

In OpenFlow, the action set for egress processing is initialized at the beginning of egress processing with an output action for the current output port, whereas the action set for ingress processing starts empty, i.e., in this scenario, the action of R2(lower priority) is executed firstly and R1(higher priority) secondly for the action set is egress processing. The action of R2 set the source IP to another sub-network(11.0.0.0) and causes the communication to 10.0.0.0 break. We successfully interrupted the communication between two hosts by a rule with lowest priority(0) in Floodlight, even the system administrator has defined a rule with highest priority to allow the communication between these two hosts.

4.5.3 Priority-attack test on the SDN controller Floodlight

Here we describes how the attack affects the controller Floodlight. Figure 4.20(a) is the configuration of attacker, which has original IP address 10.0.0.1 . In Figure 4.20(b), we can see that the victim is configured as IP 10.0.0.2. The flow rule in Figure 4.20(c) defined by the administrator, with high flow rule priority 3000, for dropping the packet with source IP 10.0.0.1(ipv4_src) to the destination with IP 10.0.0.2(ipv4_dst). Hence, the connection between attacker and victim is always suspended (Figure 4.20(d)). Unfortunately, if the attacker inserts a flow rule, even with low priority 10(Figure 4.21(d)). With this malicious rule, it allows the packet from IP 10.0.0.123 (any non-existing IP or configuration) to the destination IP 10.0.0.2. By following this, the attacker changes its IP (or any configuration to match this header in this flow rule) to 10.0.0.123. As the attacker's IP is no longer 10.0.0.1, so the adm-rule will not be applied on the packets sent by the attacker; instead, the malicious-rule will applied on it. The action defined in the malicious-rule is to output the packet, so we can find in Figure 4.21(e), the connection between the attacker and victim is rebuilt.

[3] set_vlan_id in OpenFlow 1.0 or push-VLAN in OpenFlow 1.5

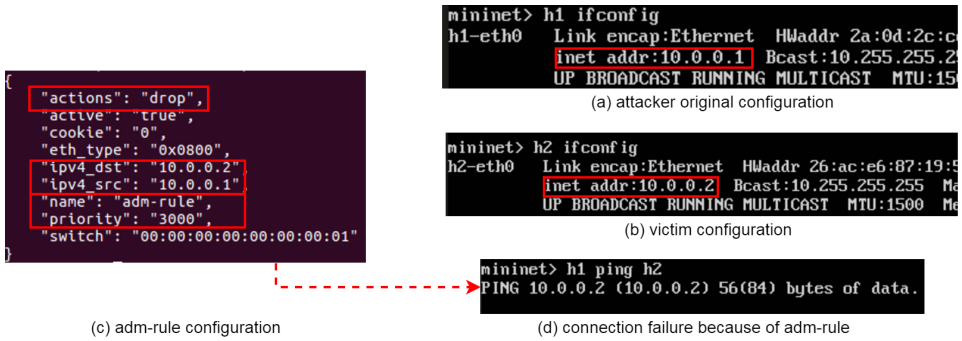


Fig. 4.20 Initial configuration and connection before priority-bypassing attack

Fig. 4.21 Priority-bypassing attack success on the controller Floodlight

4.5.4 *SENAD*-based attack detector application

We use a machine with Intel i7 CPU and 16G memories as the testing platform. The attack detector is deployed in a Docker container through Marathon framework as the framework described in Section 4.4. The algorithm 1 is used to detect the malicious host migration as in Scenario 1 IP-bypassing attack in Section 4.5.2. The detection process is as the following steps:

1. Get the flow rule list as `l`
2. Get the *packet_in* message
3. Update the device information with IP and Mac address pair
4. Replace the IP in header field in `l` with `mac` as new rule list `l'`
5. Check whether there is the same head field in `l'`
6. If find the same head field in `l'`, compare the action sets
7. If the action sets different but the priority the same, send a WARNING message

```
Result: Return a new flow entry or warning message
while listen to packet_in message do
    map_d ← device information with IP and Mac address mapping;
    l_o ← original flow rule list;
    l_n ← update flow rule list according to device information map_d ;
    if more than two rules with the same head field but different actions in l_n then
        if the same priority then
            send WARNING message;
        else
            update conflict rule;
        end
    else
        do nothing
    end
end
```

Algorithm 1: The algorithm to detect priority-bypassing attack

8. If the action sets different but the priority different, update the rule with the new attacker configuration

Hence, Figure 4.22 shows how this detector works. Figure 4.22(a) is the attacker new configuration and (b) the victim configuration. Without the attack detector, these configuration allows the attacker to connect with the victim as we tested in Section 4.5.3. But the Figure 4.22(c) is the new rule generated automatically by the attack detector. Notably, the detector updates the adm-rule automatically with the new attacker configuration (IP 10.0.0.123). Once the new rule is enforced to the data plane, the malicious connection from the attacker will be quarantined as shown in (d).

4.6 Performance evaluation

We evaluated the *SENAD* architecture performance in delivering the *packet_in* message received by the controller Floodlight until it passes the message to the network application in a sandbox(Docker) by progressively installing one network application to ten applications running at the same time. The configuration of the network application sandbox is a CPU of 0.2, a memory of 128 MB, and storage of 100 MB. The northbound processing time , i.e., latency, for delivering the message when the controller receives the *packet_in* message until arriving at the network application is shown in Figure 4.23. The figure shows that the latency can be kept under five milliseconds on the long range.

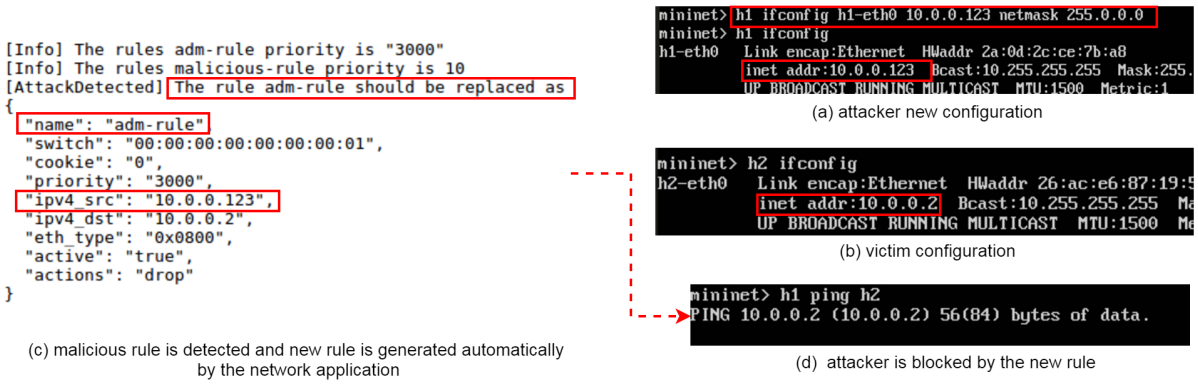


Fig. 4.22 Priority-bypassing attack protection successfully by the message queue-based detector

4.7 Discussion

We discuss the related work of the priority-bypassing attack and detection here. VeriFlow [23], FLOVER[134], NetPlumber[108], and FlowGuard [57] are real time policy verification tools. VeriFlow proposes to slice the OpenFlow-based network into equivalence classes to efficiently check for reachability violations. FLOVER checks the OpenFlow configuration for security violations using Yices SMT solver. NetPlumber is a checking tool based on HSA that utilizes a dependency graph between flow entries to incrementally check for loops, black holes, and reachability properties. These work resolve the traditional network configuration issues for the data plane of SDN. The priority-bypassing problem occurs more in the OpenFlow-based SDN. VeriFlow and NetPlumber exist between the control layer and data plane layer. Our solution(priority-bypassing detector as network application) resolves the problem in the application layer. FlowGuard is a firewall framework which examines dynamic flow updates to detect firewall policy violations. We implement our detector as a single network application for easier deployment.

FlowChecker [50] and HSA [107] are off-line configuration analysis tools. FlowChecker uses binary decision diagrams to test the configurations within a single flow table and uses model checking to verify security properties. HSA verifies the data plane correctness by modeling the network as a geometric model to discover reachability violations, forwarding loops, and traffic isolation. Anteater [53] offers a static analysis approach to diagnose network configuration problems. NICE tool applies model checking to explore the state space of the entire system — the controller, the switches, and the hosts with symbolic execution of event handlers. In general, these works take several seconds or a few hours to run. Our attack detector can find the attack in real-time as the proposed northbound interface supports event-driven.

FortNOX and SE-Floodlight resolve the policy conflict caused by the `set` action of OpenFlow in the control layer. Instead, we point out the vulnerability of the `priority` field in the

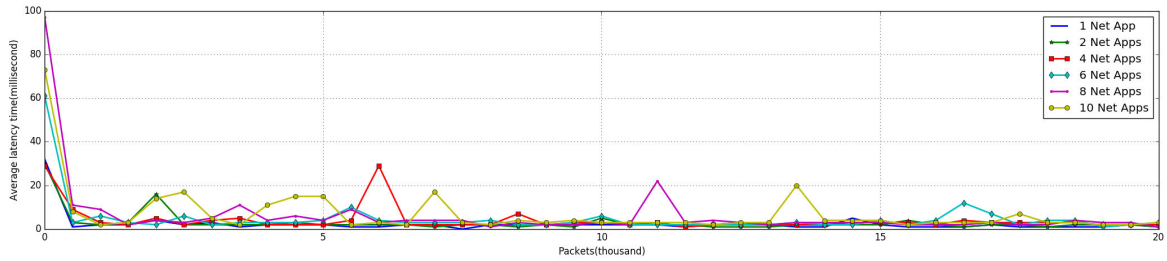


Fig. 4.23 *SENAD* architecture *packet_in* message delivering time

flow rule and resolves the policy conflict caused by priority-passing of OpenFlow. FlowTags is used to verify the policies inconsistency among the network applications; however, the priority-passing attack is a security problem rather than priority field in the flow rule [132] as the attacker can modify the self-configuration crossing different layers.

In the Policy Engine, the approle field is used for allocating the `priority` field in the flow rule. This can be our future work to optimize the priority allocation amongst different the network applications. We attempt also to improve the performance of the message queue-based northbound interface with unikernel system, technically speaking OSv or Rumprun, to allow the message broker to execute in the system kernel-level without passing through all the OS stack.

4.8 Conclusion

This work attempts to provide a secure network application deployment architecture for SDN. We propose the novel SDN architecture, *SENAD*, to deploy network applications. This architecture adopts a the message bus system as the northbound interface for delivering the OpenFlow messages from the data plane to the network applications in real time. The network administrator fist defines the high-level policies in the Policy Engine, and then the Parser will translate these policies to the Resource Allocator, the Authentication Module and the Authorization module. Based on the policies, the Resource Allocator allocates the resources of the application sandbox to separate the resources of each network application from the controller's runtime. The Authentication Module and Authorization Module will authenticate and authorize the message exchange between the APC and DPC. The preliminary results show that this approach can protect the controller against the DoS attack caused by the resource exhaustion attack, command injection, code injection, and illegal service calling. Furthermore, we implemented a network application in *SENAD* architecture to detect the priority-bypassing attack, which

is a security issue of the priority-based mechanism in OpenFlow and causes the rule-based policies fail, including IP-passing, MAC-passing, VLANs-crossing as well as DoS. With the increase of the header fields in OpenFlow, it gives attackers more opportunities to launch this attack; hence, this attack deserves careful attention. The preliminary result proves that the *SENAD* architecture can be used to deploy the network application in the current existing SDN controller and then detect the malicious host migration or flow rule enforcement. On the long range, the latency of this architecture keeps around 5 ms for delivering the *packet_in* message from the data plane to the network application.

Chapter 5

Conclusion

In this thesis, we conduct a SDN controller specific security analysis for exploiting the vulnerabilities of the SDN controller. By following this analysis, we found one of the major security issues in SDN architecture is the malicious network application injection. For this reason, we propose a security-enhancing layer (SE-layer) with two prototypes (prototype I *Controller SEPA* and prototype II *Controller DAC*) to secure the data exchange between the network applications and the SDN controller through the external APIs. More than providing the AAA control, this approach is controller-independent, that means this SE-layer can be applied to the SDN controllers including Floodlight, OpenDaylight, ONOS and Ryu with less than 0.5% performance overheads. A network secure architecture for application deployment is also introduced to protect the SDN controller against the DoS attack caused by resource exhaustion attack or malicious command injection through the internal APIs.

3D approach analysis on the security of the SDN controller. The 3-dimensional is to evaluate the security of the SDN controller. In this work, we conduct a 3-dimensional controller specific security analysis and resume the nine security principles identified from the listed attacks. The three dimensions are the essential components of a SDN controller, the characteristics provided by the controller and the Microsoft STRIDE model. We conclude that the southbound interface, the northbound interface and core services are the three weakest components of the SDN controller. The southbound interface suffers from the data-to-control spoofing packets and flow flooding. Attackers tamper core services of the controller via the northbound interface. The characteristics centralization and off-the-shelf make the SDN controller vulnerable from DoS and elevation of privileges(mainly due to the injection of malicious application to the controller's run-time) respectively. Moreover, the SDN controllers lack still the security mechanisms like connection verification, application-based access control, and data-to-control traffic control. Finally, we resume 9 security principles for the SDN controller and check the security of the current SDN controllers with these principles. We

found that the SDN controllers ONOS and OpenContrail which show relatively more secure than the others according to our analysis methodology.

The security-enhancing layer. The security-enhancing layer exists between the control plane and the application plane. We address the app-to-control security issues with focus on five main attack vectors: data tampering, illegal function calling, malicious scanning, and API abuse. Based on the identified threat models, we develop a light-weight plug-in, which is called *Controller SEPA*, by using RESTful API to defend SDN controller against malicious network application operations. Specifically, *Controller SEPA*, a controller-independent SE-layer, can provide the services including network application authentication, authorization, isolation, and information protection. Unlike OpenDaylight and ONOS which offer user-based or role-based coarse-grained access control, *Controller SEPA* provide a more fine-grained access control. The preliminary results show that *Controller SEPA* create negligible latency (0.1% to 0.3%). Furthermore, we exploit the vulnerability of the static permission control to the SDN controller with the four permission categories: READ, ADD, UPDATE and REMOVE on four open source SDN controllers, including OpenDaylight, ONOS, Floodlight, and Ryu. We found that malicious network application still can infect the SDN controllers which are even hardened by the static permission control. Therefore, we extend this SE-layer prototype I (*Controller SEPA*) to prototype II *Controller DAC*, which can protect the SDN controller against the API abuse of the network application with Dynamic Access Control System. In our implementation, *Controller DAC* requires low deployment complexity as *Controller SEPA* for securing SDN controllers, and most of time its operation is independent from the underlying SDN controller. The preliminary experimental results show that *Controller DAC* can prevent SDN controllers from API abuse through external APIs with less than 0.5% performance overhead. Hence, this SE-layer generates a low performance overhead on the SDN controller but the overall controller's security has been enhanced with low deployment complexity.

***SENAD*: The security-enhancing architecture.** The *SENAD* architecture addresses the attacks through the internal APIs, such as DoS caused by the resource exhaustion attack or malicious command injection. This architecture split the SDN controller in (1) the data plane controller (DPC) and (2) the application plane controller (APC) and adopts the message bus system as the northbound interface for delivering the OpenFlow messages from the data plane to the network applications in real time. The DPC is dedicated to communicate the date plane and interpret the OpenFlow messages and the APC is secured by design for installing the third party network applications. The network administrator fist defines the high-level policies in the Policy Engine, and then the Parser will translate these policies to the Resource Allocator, the Authentication Module and the Authorization module. Based on the policies, the Resource Allocator allocates the resources of the application sandbox to separate the resources

of each network application from the controller's runtime. The Authentication Module and Authorization Module will authenticate and authorize the message exchange between the APC and DPC. Additionally, for testing this novel architecture, we exploited an attack, called the priority-bypassing attack, due to the priority-based mechanism in the OpenFlow-based SDN. This attack can insert the malicious rules with lowest priority to manipulate the entire SDN network and make all the rules with higher priorities fail. Hence, we deployed a network application based on the *SENAD* architecture to detect the priority-bypassing attack and update the rule automatically once the attack is found. On the long range, the OpenFlow *packet_in* message processing time in the *SENAD* architecture can be maintained around 5 ms.

Perspectives. The SE-layer prototype I *Controller SEPA* is used to authorize the request of the network application via the external APIs. The prototype II *Controller DAC* even accounts the malicious request. However, determining a real malicious request is a non-trivial work because some over requests may be caused just by human operation error. The over strict verification on the requests may generate the false positive signals. In this case, a smarter algorithm implemented with artificial intelligence could be applied to improve the detection of the API abuse attack. As the internal northbound interfaces bring the security issues such as malicious code injection or resource exhaustion attack, this motivates us to propose a new SDN architecture by replacing the northbound interfaces between the network applications and the SDN controller by the message bus system, a real-time and message-driven system. As discussed in Chapter 4, we split the SDN controller in DPC and APC, and then secure the APC by design for protecting against the malicious application injection and provide the real-time message service to the network applications. However, the processing time in the Figure 4.23 is non-linear (Zig-Zag). The non-linear results are because we configure a buffer space (16 bytes) and linger time (1 millisecond) for increasing the throughput of the Kafka producer. The zero buffer space and zero linger time can reduce the latency; however, the throughput will be compromised. The high scalability provided by the message bus allows us to deploy the network service flexibly in the SDN-enabled network. Hence, the performance tuning, i.e., optimizing the latency and the throughput with different network service deployment, would be our future work when using the message bus system in the SDN architecture. Even more, unikernel as an emerging technologies by providing a lighter approach to isolate the network application runs directly on a hypervisor or hardware without an intervening OS such as Linux or Windows [2]. This approach can avoid the attacker to attack the SDN controller through the vulnerabilities or bugs from the host of a SDN controller. Hence, unikernel could be one of possibility to securely deploy the network applications in the SDN architecture. The study and test for the feasibility of a unikernel-enabled SDN could be one of our future works.

Publication list

1 Yuchia Tseng, Zonghua Zhang and Farid Naït-Abdesselam, "SRV: Switch-based rules verification in software defined networking,"2016 IEEE NetSoft Conference and Workshops (NetSoft), Seoul, 2016, pp. 477-482.

2 Yuchia Tseng, Zonghua Zhang and Farid Naït-Abdesselam, "ControllerSEPA: A Security-Enhancing SDN Controller Plug-in for OpenFlow Applications,"2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Guangzhou, 2016, pp. 268-273. (Outstanding Student Paper Reward)

3 Yuchia Tseng, Montida Pattaranantakul, Ruan He, Zonghua Zhang and Farid Naït-Abdesselam, "Controller DAC: Securing SDN controller with dynamic access control,"2017 IEEE International Conference on Communications (ICC), Paris, 2017, pp. 1-6.

4 Montida Pattaranantakul, Yuchia Tseng, Ruan He, Zonghua Zhang, and Ahmed Meddahi, ” A First Step Towards Security Extension for NFV Orchestrator”, 2017 ACM, SDN-NFVSec (Security in Software Defined Networks & Network Function Virtualization)

5 Yuchia Tseng, Farid Naït-Abdesselam and Ashfaq Khokhar, “SENAD: Securing Network Application Deployment in Software Defined Networks”, 2018 IEEE International Conference on Communications (ICC).

6 Yuchia Tseng, Farid Naït-Abdesselam and Ashfaq Khokhar, “A Comprehensive 3D Security Analysis of a Controller in Software Defined Networking”, 29 April 2018, in Wiley Security and Privacy.

References

[1] Advait Dixit, Fang Hao, Sarit Mukherjee, T. V. Lakshman, and Ramana Rao Kompella. Elasticon: an elastic distributed sdn controller. In *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 17–27, Oct 2014.

[2] Anil Madhavapeddy and David J. Scot . Unikernels: The Rise of the Virtual Library Operating System. URL http://unikernel.org/files/2014-cacm-unikernels.pdf.

[3] Flowgrammable.org. Flowgrammable. URL http://flowgrammable.org/sdn/openflow/message-layer.

[4] He Li, Peng Li, Song Guo, and Amiya Nayak. Byzantine-resilient secure software-defined networks with multiple controllers. In *2014 IEEE International Conference on Communications*, ICC '14, pages 695–700, June 2014. doi: 10.1109/ICC.2014.6883400.

[5] HP . SDN Applications , . URL https://www.hpe.com/emea_europe/en/networking/applications.html.

[6] HP . 8200 ZL switch series, . URL http://h17007.www1.hp.com/us/en/networking/products/switches/HP_8200_zl_Switch_Series.

[7] HP . SDN controller architecture, . URL http://h17007.www1.hpe.com/docs/networking/solutions/sdn/devcenter/06_-_HP_SDN_Controller_Architecture_TSG_v1_3013-10-01.pdf.

[8] Jeremy M. Dover . A denial of service attack against the Open Floodlight SDN controller. URL http://dovernetworks.com/wp-content/uploads/2013/12/OpenFloodlight-12302013.pdf.

[9] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. OFX: Enabling OpenFlow Extensions for Switch-Level Security Applications . In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1678–1680, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2810120. URL http://doi.acm.org/10.1145/2810103.2810120.

[10] Kostas Pentikousis, Yan Wang, and Weihua Hu. MobileFlow: Toward software-defined mobile networks. *IEEE Communications Magazine*, 51(7):44–53, 2013. ISSN 0163-6804. doi: 10.1109/MCOM.2013.6553677.

[11] M.M. Othman Othman and Koji Okamura. Hybrid control model for flow-based networks. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, COMPSACW '13, pages 765–770, July 2013. doi: 10.1109/COMPSACW.2013.13.

[12] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J. Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Jennifer Rexford, Cole Schlesinger, Alec Story, and David Walker. Languages for software-defined networks. In *IEEE Communications Magazine*, pages 128–134. IEEE, 2013.

[13] Network Working Group . A Border Gateway Protocol 4 (BGP-4) . URL https://tools.ietf.org/html/rfc4271.

[14] Niels L.M. van Adrichem, Benjamin J. van Asten, and Fernando A. Kuipers. Fast recovery in software-defined networks. In *Proceedings of the 2014 Third European Workshop on Software Defined Networks*, EWSDN '14, pages 61–66, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6919-7. doi: 10.1109/EWSDN.2014.13. URL http://dx.doi.org/10.1109/EWSDN.2014.13.

[15] NOX Repo . POX . URL http://www.noxrepo.org/pox.

[16] Open Network Lab . ONOS Application Permissions . URL https://wiki.onosproject.org/display/ONOS/ONOS+Application+Permissions.

[17] Paulo Fonseca, Ricardo Bennesby, Edjard Mota, and Alexandre Passito. A replication component for resilient openflow-based networking. In *2012 IEEE Network Operations and Management Symposium*, NOMS '12, pages 933–939, April 2012. doi: 10.1109/NOMS.2012.6212011.

[18] Pica8 . Pica8 3920. URL http://www.pica8.org/documents/pica8-datasheet-64x10gbe-p3780-p3920.pdf.

[19] Takayuki Sasaki, Adrian Perrig, and Daniele E. Asoni. Control-plane Isolation and Recovery for a Secure SDN Architecture. In *Proceedings of Second IEEE Conference on Network Softwarizartion Workshop Sec-VirtNet 2016*, NetSoft '16, 2016.

[20] TechTarget . SDN controller (software - defined networking controller) , . URL http://searchsdn.techtarget.com/definition/SDN-controller-software-defined-networking-controller.

[21] TechTarget . Data Plane , . URL http://searchsdn.techtarget.com/definition/data-plane-DP.

[22] Xitao Wen, Bo Yang, Yan Chen, Chengchen Hu, Yi Wang, Bin Liu, and Xiaolin Chen. SDNShield: Reconciliating Configurable Application Permissions for SDN App Markets. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '16, pages 121–132, June 2016. doi: 10.1109/DSN.2016.20.

[23] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13,

pages 15–28, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482630.

[24] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN '10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863133.1863136.

[25] AMQP ORG. Advanced Message Queuing Protocol. URL http://www.amqp.org/.

[26] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. In *PADL*, pages 235–249, 2011.

[27] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1477-0. doi: 10.1145/2342441.2342451. URL http://doi.acm.org/10.1145/2342441.2342451.

[28] Andreas Voellmy, Junchang Wang, Yang R. Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486030. URL http://doi.acm.org/10.1145/2486001.2486030.

[29] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC '11, pages 29–29, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2002181.2002210.

[30] Andrei Bondkovskii, John Keeney, Sven van der Meer, and Stefan Weber. Qualitative comparison of open-source SDN controllers. In *2016 IEEE/IFIP Network Operations and Management Symposium*, NOMS '16, pages 889–894, April 2016. doi: 10.1109/NOMS.2016.7502921.

[31] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM '13, Hong Kong, China, August 2013.

[32] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating SDN Application Failures with LegoSDN. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 22:1–22:7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3256-9. doi: 10.1145/2670518.2673880. URL http://doi.acm.org/10.1145/2670518.2673880.

[33] Bela Ban. Design and implementation of a reliable group communication toolkit for java. Technical report, 1998.

[34] Ben Pfaff, Justin Pettit, Keith Amidon, Martín Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Notes Proceedings of the 8th ACM Workshop on Hot Topics in Networks*, HotNets-VIII, 2009.

[35] Big Switch. Floodlight. URL http://www.projectfloodlight.org/.

[36] Carolyn J. Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535862. URL http://doi.acm.org/10.1145/2535838.2535862.

[37] Centec Networks. V350VCentec open SDN platform. URL http://www.valleytalk.org/wp-content/uploads/2013/04/Centec-Open-SDN-Platform.pdf.

[38] Changhoon Yoon, Taejune Park, Seungsoo Lee, Heedo Kang, Seungwon Shin, and Zonghua Zhang. Enabling security functions with SDN: A feasibility study. *Computer Networks*, 85:19 – 35, 2015. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/j.comnet.2015.05.005. URL http://www.sciencedirect.com/science/article/pii/S1389128615001619.

[39] Christian Banse and Sathyanarayanan Rangarajan. A Secure Northbound Interface for SDN Applications. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 834–839, Aug 2015. doi: 10.1109/Trustcom.2015.454.

[40] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 1–13, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto.

[41] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 217–230, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103685. URL http://doi.acm.org/10.1145/2103656.2103685.

[42] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol and Anja Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1477-0. doi: 10.1145/2342441.2342443. URL http://doi.acm.org/10.1145/2342441.2342443.

[43] David Erickson. The beacon openflow controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491189. URL http://doi.acm.org/10.1145/2491185.2491189.

[44] David Jacobs. Addressing SDN security challenges means securing the controller, 2017. URL http://searchsdn.techtarget.com/tip/Addressing-SDN-security-challenges-means-securing-the-controller.

[45] Diego Kreutz, Fernando M. V. Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *CoRR*, abs/1406.0440, 2014. URL http://arxiv.org/abs/1406.0440.

[46] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043560. URL http://doi.acm.org/10.1145/2043556.2043560.

[47] Diogo M. F. Mattos, Lyno H. G. Ferraz, and Otto C. M. B. Duarte. AuthFlow: Authentication and Access Control Mechanism for Software Defined Networking, 2016.

[48] Djamila Bendouda, Abderrezak Rachedi, and Hafid Haffaf. Programmable architecture based on Software Defined Network for Internet of Things: Connected Dominated Sets approach. *Future Generation Computer Systems*, 80:188 – 197, 2018. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2017.09.070. URL http://www.sciencedirect.com/science/article/pii/S0167739X17314590.

[49] Dongting Yu, Andrew W. Moore, Chris Hall, and Ross Anderson. Authentication for Resilience: The Case of SDN. *Security Protocols XXI*, pages 39–44, 2013.

[50] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '10, pages 37–44, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0093-3. doi: 10.1145/1866898.1866905. URL http://doi.acm.org/10.1145/1866898.1866905.

[51] Fábio Botelho, Alysson Bessani, Fernando M. V. Ramos, and Paulo Ferreira. On the design of practical fault-tolerant SDN controllers. In *Proceedings of 3rd European Workshop on Software Defined Networks*, EWSDN '14, 2014.

[52] Guang Yao, Jun Bi, and Luyi Guo. On the cascading failures of multi-controllers in software defined networks. In *2013 21st IEEE International Conference on Network Protocols*, ICNP '13, pages 1–2, Oct 2013. doi: 10.1109/ICNP.2013.6733624.

[53] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, and P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0797-0. doi: 10.1145/2018436.2018470. URL http://doi.acm.org/10.1145/2018436.2018470.

[54] Haopei Wang, Lei Xu, and Guofei Gu. OF-GUARD: A DoS Attack Prevention Extension in Software-Defined Networks. URL https://www.usenix.org/sites/default/files/ons2014-poster-wang.pdf.

[55] Haopei Wang, Lei Xu, and Guofei Gu. Floodguard: A dos attack prevention extension in software-defined networks. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 239–250. IEEE Computer Society, 2015. ISBN 978-1-4799-8629-3. URL http://dblp.uni-trier.de/db/conf/dsn/dsn2015.html#WangXG15.

[56] Hitesh Padekar, Younghee Park, Hongxin Hu, and Sang-Yoon Chang. Enabling Dynamic Access Control for Controller Applications in Software-Defined Networks. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, SACMAT '16, pages 51–61, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3802-8. doi: 10.1145/2914642.2914647. URL http://doi.acm.org/10.1145/2914642.2914647.

[57] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. Flowguard: Building robust firewalls for software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 97–102, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2989-7. doi: 10.1145/2620728.2620749. URL http://doi.acm.org/10.1145/2620728.2620749.

[58] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 59–72, Oakland, CA, 2015. USENIX Association. ISBN 978-1-931971-218. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kim.

[59] Ijaz Ahmad, Suneth Namal, Mika Ylianttila, and Andrei Gurtov. Security in software defined networks: A survey. *IEEE Communications Surveys Tutorials*, 17(4):2317–2346, Fourthquarter 2015. ISSN 1553-877X. doi: 10.1109/COMST.2015.2474118.

[60] Internet Engineering Task Force (IETF) . Requirements for MPLS Transport Profile (MPLS-TP) Shared Mesh Protection, . URL https://tools.ietf.org/html/rfc7412.

[61] Internet Engineering Task Force (IETF) . Network Configuration Protocol (NETCONF) , . URL https://tools.ietf.org/html/rfc6241.

[62] Internet Engineering Task Force (IETF) . The Transport Layer Security (TLS) Protocol, . URL https://www.ietf.org/rfc/rfc4346.txt.

[63] Internet Engineering Task Force (IETF) . Improving TCP's Robustness to Blind In-Window Attacks , . URL https://tools.ietf.org/html/rfc5961.

[64] Internet Engineering Task Force (IETF) . SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains, . URL https://tools.ietf.org/html/draft-yin-sdn-sdni-00.

[65] Internet Engineering Task Force (IETF). The Locator/ID Separation Protocol (LISP) , . URL https://tools.ietf.org/html/rfc6830.

[66] Internet Engineering Task Force (IETF). The Open vSwitch Database Management Protocol, . URL https://datatracker.ietf.org/doc/rfc7047/.

[67] Internet Engineering Task Force (IETF). The TCP Authentication Option , . URL https://tools.ietf.org/html/rfc5925.

[68] Internet Engineering Task Force (IETF). Recommendations for Transport-Protocol Port Randomization , . URL https://tools.ietf.org/html/rfc6056.

[69] Jeremy M. Dover . A switch table vulnerability in the Open Floodlight SDN controller. URL http://dovernetworks.com/wp-content/uploads/2014/03/OpenFloodlight-03052014.pdf.

[70] Jiseong Noh, Seunghyeon Lee, Jaehyun Park, Seungwon Shin, and Brent B. Kang. Vulnerabilities of network OS and mitigation with state-based permission system. *Security and Communication Networks*, pages n/a–n/a, 2015. ISSN 1939-0122. doi: 10.1002/sec.1369. URL http://dx.doi.org/10.1002/sec.1369.

[71] John Sonchack, Jonathan M. Smith, Adam J. Aviv, and Eric Keller. Enabling Practical Software-defined Networking Security Applications with OFX. In *Proceedings of 2016 Annual Network and Distributed System Security Symposium*, NDSS '16, 2016.

[72] Juniper. Opencontrail architecture document. URL http://www.opencontrail.org/opencontrail-architecture-documentation.

[73] Juniper. Opencontrail. http://www.opencontrail.org/, 2016. Online available.

[74] Juniper Networks. Contrail architecture. URL http://www.juniper.net/us/en/local/pdf/whitepapers/2000535-en.pdf.

[75] Kapil Dhamecha and Bhushan Trivedi. SDN Issues - A Survey. *International Journal of Computer Applications*, 73(18):30–35, July 2013. Full text available.

[76] Kevin Benton, L. Jean Camp, and Chris Small. OpenFlow Vulnerability Assessment. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 151–152, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491222. URL http://doi.acm.org/10.1145/2491185.2491222.

[77] Kevin Phemius, Mathieu Bouet, and Jeremie Leguay. DISCO: Distributed SDN controllers in a multi-domain environment . In *2014 IEEE Network Operations and Management Symposium*, NOMS '14, pages 1–2, May 2014. doi: 10.1109/NOMS.2014.6838273.

[78] KulCloud Inc Ltd. OpenMUL. http://www.openmul.org/, 2016. Online available.

[79] M. M. Othman Othmanand Koji Okamura. Securing distributed control of software defined networks. *International Journal of Computer Science and Network Security*, 13 (9), 2013.

[80] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test Openflow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228312.

[81] Matthew Monaco, Oliver Michel, and Eric Keller. Applying Operating System Principles to SDN Controller Design. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 2:1–2:7, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2596-7. doi: 10.1145/2535771.2535789. URL http://doi.acm.org/10.1145/2535771.2535789.

[82] Michael Jarschel, Thomas Zinner, Tobias Hossfeld, Phuoc Tran-Gia, and Wolfgang Kellerer. Interfaces, attributes, and use cases: A compass for SDN. *IEEE Communications Magazine*, 52(6):210–217, June 2014. ISSN 0163-6804. doi: 10.1109/MCOM.2014.6829966.

[83] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. Sphinx: Detecting security attacks in software-defined networks. In *Proceedings of 2015 Annual Network and Distributed System Security Symposium*, NDSS '15. The Internet Society, 2015. URL http://dblp.uni-trier.de/db/conf/ndss/ndss2015.html#DhawanPMM15.

[84] Moreno Ambrosin, Mauro Conti, Fabio De Gaspari, and Nishanth Devarajan. Amplified distributed denial of service attack in software defined networking. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security*, NTMS '16, pages 1–4, Nov 2016. doi: 10.1109/NTMS.2016.7792432.

[85] Moreno Ambrosin, Mauro Conti, Fabio De Gaspari, and Radha Poovendran. LineSwitch: Efficiently Managing Switch Flow in Software-Defined Networking while Effectively Tackling DoS Attacks. In *The 10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '15, Feb 2015.

[86] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008. ISSN 0146-4833. doi: 10.1145/1384609.1384625. URL http://doi.acm.org/10.1145/1384609.1384625.

[87] NEC. Trema. https://trema.github.io/trema/, 2016. Online available.

[88] NECProgrammable. Flow UNIVERGE PF5820. URL http://www.nec.com/en/global/prod/pflow/images_documents/ProgrammableFlow_Switch_PF5820.pdf.

[89] Network Working Group. Generalized Multi-Protocol Label Switching (GMPLS) Architecture, . URL https://buildbot.tools.ietf.org/html/rfc3945.

[90] Network Working Group. Path Computation Element (PCE) Communication Protocol (PCEP), . URL https://tools.ietf.org/html/rfc5440.

[91] Network Working Group. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks, . URL https://tools.ietf.org/html/rfc3411.

[92] NoviFlow. NoviSwitch 1248 high performance OpenFlow switch. URL http://www.nvc.co.jp/pdf/product/noviflow/NoviSwitch1248Datasheet.pdf.

[93] NOX Repo. NOX. URL http://www.noxrepo.org/.

[94] NTT. Ryu. https://osrg.github.io/ryu/, 2016. Online available.

[95] ON.Lab. ONOS, . URL http://onosproject.org.

[96] ON.Lab. ONOS Security Mode, . URL https://wiki.onosproject.org/display/ONOS/Introduction.

[97] ON.Lab. Configuring OVS connection using SSL/TLS with self-signed certificates, . URL https://wiki.onosproject.org/pages/viewpage.action?pageId=6358090.

[98] Open Networking Foundation (ONF) . Principles and Practices for Securing Software-Defined Networks, 2015.

[99] Open Networking Foundation (ONF) . Threat Analysis for the SDN Architecture, 2016.

[100] Open Networking Foundation(ONF). Security Foundation Requirements for SDN Controllers, 2016.

[101] OpenDaylight foundation. OpenDaylight: A Linux Foundation Collaborative Project, . URL https://www.opendaylight.org.

[102] OpenDaylight foundation. OpenDaylight: Authentication, Authorization and Accounting (AAA) Services, . URL http://docs.opendaylight.org/en/stable-boron/user-guide/authentication-and-authorization-services.html.

[103] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2989-7. doi: 10.1145/2620728.2620744. URL http://doi.acm.org/10.1145/2620728.2620744.

[104] Montida Pattaranantakul, Ruan He, Ahmed Meddahi, and Zonghua Zhang. Secmano: Towards network functions virtualization (nfv) based security management and orchestration. *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 598–605, 2016.

[105] Montida Pattaranantakul, Yuchia Tseng, Ruan He, Zonghua Zhang, and Ahmed Meddahi. A first step towards security extension for nfv orchestrator. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFVSec '17, pages 25–30, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4908-6. doi: 10.1145/3040992.3040995. URL http://doi.acm.org/10.1145/3040992.3040995.

[106] Peter Loscocco and Stephen Smalleyi. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conference*, 2001.

[107] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 113–126, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[108] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 99–111, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian.

[109] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A Security Enforcement Kernel for OpenFlow Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 121–126, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1477-0. doi: 10.1145/2342441.2342466. URL http://doi.acm.org/10.1145/2342441.2342466.

[110] Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran. Securing the Software-Defined Network Control Layer. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, NDSS '15, February 2015.

[111] Rahamatullah Khondoker, Adel Zaalouk, Ronald Marx, and Kpatcha Bayarou. Feature-based comparison and selection of Software Defined Networking (SDN) controllers. In *2014 World Congress on Computer Applications and Information Systems*, WCCAIS '14, pages 1–7, Jan 2014. doi: 10.1109/WCCAIS.2014.6916572.

[112] Raphael Durner and Wolfgang Kellerer. The cost of Security in the SDN control Plane. In *Proceedings of the ACM CoNEXT Student Workshop*. ACM, 2015.

[113] Rishikesh Sahay, Gregory Blanc, Zonghua Zhang and Hervé Debar. Towards autonomic DDoS mitigation using Software Defined Networking. In *SENT 2015 : NDSS Workshop on Security of Emerging Networking Technologies*, NDSS '15, San Diego, Ca, United States, February 2015. Internet society. URL https://hal.archives-ouvertes.fr/hal-01257899.

[114] Rowan Klöti, Vasileios Kotronis, and Paul Smith. Openflow: A security analysis. In *2013 21st IEEE International Conference on Network Protocols*, ICNP '13, pages 1–6, Oct 2013. doi: 10.1109/ICNP.2013.6733671.

[115] Ryan Beckett, Xuan Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford, and David Walker. An assertion language for debugging sdn applications. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 91–96, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2989-7. doi: 10.1145/2620728.2620743. URL http://doi.acm.org/10.1145/2620728.2620743.

[116] Ryan Hand and Eric Keller . ClosedFlow: Openflow-like Control over Proprietary Devices . In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 7–12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2989-7. doi: 10.1145/2620728.2620738. URL http://doi.acm.org/10.1145/2620728.2620738.

[117] Rishikesh Sahay, Gregory Blanc, Zonghua Zhang, and Hervé Debar. ArOMA: an SDN based autonomic DDoS mitigation framework. *Computers and Security*, 70:482 – 499, September 2017. doi: 10.1016/j.cose.2017.07.008. URL https://hal.archives-ouvertes.fr/hal-01648031.

[118] Sandhya Rathee, Yash Sinha, and K. Haribabu. A survey: Hybrid SDN. *Journal of Network and Computer Applications*, 100:35 – 55, 2017. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2017.10.003. URL http://www.sciencedirect.com/science/article/pii/S108480451730317X.

[119] Sandra Scott-Hayward. Design and deployment of secure, robust, and resilient sdn controllers. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization*, NetSoft '15, pages 1–5, April 2015. doi: 10.1109/NETSOFT.2015.7258233.

[120] Sandra Scott-Hayward, Christopher Kane, and Sakir Sezer. OperationCheckpoint: SDN Application Control. In *Proceedings of the 2014 IEEE 22Nd International Conference on Network Protocols*, ICNP '14, pages 618–623, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6204-4. doi: 10.1109/ICNP.2014.98. URL http://dx.doi.org/10.1109/ICNP.2014.98.

[121] Sandra Scott-Hayward, Gemma O'Callaghan, and Sakir Sezer. SDN Security: A Survey. In *2013 IEEE SDN for Future Networks and Services*, SDN4FNS '13, pages 1–7, Nov 2013. doi: 10.1109/SDN4FNS.2013.6702553.

[122] Sandra Scott-Hayward, Sriram Natarajan, and Sakir Sezer. A survey of security in software defined networks. *IEEE Communications Surveys Tutorials*, 18(1):623–654, Firstquarter 2016. ISSN 1553-877X. doi: 10.1109/COMST.2015.2453114.

[123] SDxCentral. What are SDN Northbound APIs?, . URL https://www.sdxcentral.com/sdn/definitions/north-bound-interfaces-api.

[124] SDxCentral. What are SDN Southbound APIs?, . URL https://www.sdxcentral.com/sdn/definitions/southbound-interface-api.

[125] SDxCentral. SDN Controllers Report, 2015.

[126] Seungsoo Lee, Changhoon Yoon, and Seungwon Shin. The Smaller, the Shrewder: A Simple Malicious Application Can Kill an Entire SDN Environment. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks; Network Function Virtualization*, SDN-NFV Security '16, pages 23–28, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4078-6. doi: 10.1145/2876019.2876024. URL http://doi.acm.org/10.1145/2876019.2876024.

[127] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Proceedings of 2017 Annual Network and Distributed System Security Symposium*, NDSS '17. The Internet Society, 2017. URL https://www.internetsociety.org/doc/delta-security-assessment-framework-software-defined-networks.

[128] Seungwon Shin, Phil Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proceedings of the 2013 Network and Distributed System Security Symposium*, NDSS '13, 2013.

[129] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 413–424, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516684. URL http://doi.acm.org/10.1145/2508859.2516684.

[130] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent B. Kang. Rosemary: A Robust, Secure, and High-performance Network Operating System. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 78–89, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660353. URL http://doi.acm.org/10.1145/2660267.2660353.

[131] Seungwon Shinand and Guofei Gu. Attacking software-defined networks: A first feasibility study. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 165–166, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491220. URL http://doi.acm.org/10.1145/2491185.2491220.

[132] Seyed K. Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NDSI '14, pages 543–546, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/fayazbakhsh.

[133] Soheil Hassas Yeganeh and Yashar Ganjali . Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1477-0. doi: 10.1145/2342441.2342446. URL http://doi.acm.org/10.1145/2342441.2342446.

[134] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Model checking invariant security properties in openflow. In *2013 IEEE International Conference on Communications*, ICC '13, pages 1974–1979. IEEE, 2013. URL http://dblp.uni-trier.de/db/conf/icc/icc2013.html#SonSYPG13.

[135] Srinivas Narayana, Jennifer Rexford and David Walker. Compiling path queries in software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 181–186, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2989-7. doi: 10.1145/2620728.2620736. URL http://doi.acm.org/10.1145/2620728.2620736.

[136] Stephanos Matsumoto, Samuel Hitz, and Adrian Perrig. Fleet: Defending SDNs from Malicious Administrators. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 103–108, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2989-7. doi: 10.1145/2620728.2620750. URL http://doi.acm.org/10.1145/2620728.2620750.

[137] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proceedings of 2015 Annual Network and Distributed System Security Symposium*, NDSS '15, February 2015.

[138] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh , Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle,

Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486019. URL http://doi.acm.org/10.1145/2486001.2486019.

[139] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 351–364, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924968.

[140] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 1–10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-443-0. doi: 10.1145/1592681.1592683. URL http://doi.acm.org/10.1145/1592681.1592683.

[141] Volkan Yazici, Oguz Sunay, and Ali O. Ercan. Controlling a Software-Defined Network via Distributed Controllers. *ArXiv e-prints*, January 2014.

[142] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A survey on software-defined networking. *IEEE Communications Surveys Tutorials*, 17(1): 27–51, Firstquarter 2015. ISSN 1553-877X. doi: 10.1109/COMST.2014.2330903.

[143] Xin Jin. *Dynamic control of Software-defined Networks*. PhD thesis, PRINCETON UNIVERSITY, 2016.

[144] Xitao Wen, Yan Chen, Chengchen Hu, Chao Shi, and Yi Wang. Towards a Secure Controller Platform for Openflow Applications. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 171–172, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491212. URL http://doi.acm.org/10.1145/2491185.2491212.

[145] Yuchia Tseng, Montida Pattaranantakul, Ruan He, Zonghua Zhang, and Farid Naït-Abdesselam. Controller DAC: Securing SDN Controller with Dynamic Access Control. In *Proceedings of the IEEE ICC 2017 Conference*, ICC '17. IEEE, 2017.

[146] Yuchia Tseng, Zonghua Zhang, and Farid Naït-Abdesselam. ControllerSEPA: A Security-Enhancing SDN Controller Plug-in for OpenFlow Applications. In *Proceedings of the IEEE PDCAT 2017 Conference*, PDCAT '16. IEEE, 2016.

[147] Yuchia Tseng, Zonghua Zhang, and Farid Naït-Abdesselam. SRV: Switch-based Rule Verification. In *Proceedings of Second IEEE Conference on Network Softwarizartion Workshop Sec-VirtNet 2016*, NetSoft '16, 2016.

[148] Zehui Wu and Qiang Wei. Quantitative analysis of the security of software-defined network controller using threat/effort model. In *Mathematical Problems in Engineering*, page 11, 2017. doi: 10.1155/2017/8740217.

[149] Zheng Cai. *Maestro: Achieving Scalability and Coordination in Centralizaed Network Control Plane*. PhD thesis, Rice University, Houston, TX, USA, 2012. AAI3521292.