Investigating current state Security of OpenFlow Networks

Focusing on the control-data plane communications

M.L. Pors

FOX IT part of nccgroup

TU Delft Delft University of Technology

Investigating current state Security of OpenFlow Networks

Focusing on the control-data plane communications

by

M.L. Pors

to obtain the degree of Master of Science, completing the studies
Computer Science (Software Technology) &
Electrical Engineering (Telecommunications & Sensing Systems)
at the Delft University of Technology,
to be defended publicly on Thursday July 6, 2017 at 13:30.

Student number:	4008847	
Project duration:	August 22, 2016 – July 6, 2017	
Thesis committee:	Dr. ir. F. A. Kuipers,	TU Delft, supervisor
	Francisco Dominguez Santos,	Principal IT Security Expert Fox IT, supervisor
	C. Doerr,	TU Delft

FOX IT part of nccgroup

TU Delft Delft University of Technology

Preface

This report is the result of my Master Thesis Research, with which I hope to complete my Double Degree in Computer Science and Electrical Engineering. In this research, I have investigated the security of software-defined networking (SDN) where I have chosen to focus on the control-data plane communications, or the so-called control channel. This research is done at the IT security company *Fox IT*. For them, it is important that this report can serve as a reference guide to security challenges in SDN for both own employees and others, such that this knowledge can be used when working with SDN in the future.

For me, the field of SDN and deep-dive into OpenFlow turned out to be a perfect fit given my interest in network architectures and protocols, while at the same time I was able to work on an ever more relevant topic: security. One of my personal goals for this thesis research was to create a solid combination between theory (one of my strengths) and practice (the direction I wanted to grow in). In my opinion, I succeed in this. One of the parts I loved most during this year was the 'hacking' and 'breaking'. I want to thank my supervisor Francisco for giving me the freedom to form this research to accomplish my goals and encouraging me to perform even more attacks just to see what is possible. Without this encouragement, I am sure it would have resulted in a pretty boring thesis.

M.L. Pors
Delft, June 2017

Summary

As originally defined, Software-Defined Networking (SDN) refers to a network architecture where the forwarding state in the data plane is managed by a remote control plane decoupled from the former. As a result, network devices become simple forwarding elements, while the complete control logic is moved to an external entity, the so-called controller. The network becomes programmable through software applications running on top of this controller that interacts with the underlying data plane devices over the so-called control channel. On this channel a specific protocol is used, of which OpenFlow is the best known example.

While the programmability of the network offers new possibilities for security solutions, the security of the network itself is still a challenge which has obstructed the total take-over of SDN in enterprise networks. Three security challenges which are specific SDN threats are: (1) there is a lack of trust mechanisms between the controller and applications, meaning that malicious applications can easily be deployed on the network, (2) there can be attacks on and vulnerabilities in the controller and a compromised controller may compromise the entire network, and (3) the network can be attacked over the control channel, making it possible to exploit the communication with the controller.

In this research, we have focused on the third problem and investigated the security of the control channel. Specifically, we have limited our research to OpenFlow networks and investigated four different SDN controllers: Ryu, ONOS, OpenDaylight and HPE VAN. We looked at the security of the control channel from an attacker's point of view and searched for possible vulnerabilities from three angles.

First, we investigated the possibilities for attackers who are in possession of a direct connection towards a controller or network device and are able to (mis)use the OpenFlow protocol. Using self-written scripts, we are able to find and identify OpenFlow connections and even perform impersonation attacks. By impersonating a switch towards a controller, an attacker is able to directly influence the controller's view of the network which could influence the controller's decisions. By impersonating a controller towards a switch, an attacker is able to alter the functionality of the switch, influencing traffic forwarding. This gives possibilities for attacks like eavesdropping or man-in-the-middle attacks.

Second, we investigated whether the deployment of authentication methods serves as a countermeasure against the attacks found above. We did this by getting hands-on experience with Secure Sockets Layer and Transport Layer Security (SSL/TLS) in SDN. Using one-way authentication where only the controller needs to authenticate itself, we see that an attacker is still able to impersonate switches. With two-way authentication, we see that the control channel is completely protected. However, there are still challenges which arise, one of which is the potential tedious key and certificate management which discourage network architects to deploy SSL/TLS.

Last, we tried to attack the network while being a regular host. While we are not able to directly create OpenFlow traffic, there are other types of messages which generate traffic towards the controller. We have created an attack which uses the Address Resolution Protocol (ARP) to generate such traffic, to that extent that we are able to flood memory of the network switches and even accomplish denial-of-service by flooding the controller with OpenFlow traffic which it needs to process.

Given our findings, there are three recommendations we give to improve the security of the control channel: network architects (1) must set up an isolated control channel, such that attackers don't have the possibility to access the control channel and misuse OpenFlow, (2) should always deploy two-way SSL/TLS to protect the control channel and (3) should try to reduce the amount of traffic on the control channel.

To find correct solutions for the third recommendation, extra research is needed. Also, it will be important for network architects to investigate the future of the SDN architecture which will probably influence the security of SDN. While the scope of this research was rather broad, one limitation is that it is unknown how relevant it stays due to the fast changes in the SDN world. The main development which will influence the security of the lower part of the network (and thus the control channel) is the programmability of the data plane, adding intelligence to the network devices. This will probably contribute to solutions for the reduction of the communication between the data plane and controller.

Contents

1

Introduction to Software-Defined Networking

1.1. Towards Software-Defined Networks

The complexity of IP networks stems from three key issues. First, these networks are vertically integrated. The control plane, that decides how to handle network traffic, and the data plane, that simply forwards traffic according to the decisions made by the control plane, are bundled inside the networking devices. Second, we have a management layer where network policy is actually defined with the use of software services, such as services to remotely monitor and configure the control functionality. In the past, different vendors offered proprietary solutions with specialized hardware and control programs, such that network operators would have to acquire and maintain these different solutions and corresponding specialized teams. Last, it turned out that within the network there was a lack of in-path functionalities, which could be added by again new specialized components and middleboxes like firewalls and intrusion detection systems. As a result, to express the required high-level network policies, network operators need to configure each individual network device separately using vendor-specific and low-level commands.

Network complexity has been reducing flexibility and hindering innovation and evolution of the networking infrastructure. As a response, the wish arised to break the vertical integration, separating the network's control logic from the underlying routers and switches. The first systems following this wish date from 2006. In this year, Secure Architecture for Network Enterprise (SANE) [1] was proposed, where network connectivity is mediated by a single protection layer overseen by a logically centralized controller. In addition, the same writers proposed Ethane [2] in 2007, a security management architecture where simple flow-based switches work below a central controller managing admittance and routing of the flows.

As originally defined, Software-Defined Networking (SDN) refers to a network architecture where the forwarding state in the data plane is managed by a remote control plane (or, *the controller*) decoupled from the former [3]. The rise of the paradigm started when researchers defined the first protocols / application programming interfaces (APIs) for the communication between the controller and the data plane. The most notable example of such an API is OpenFlow (OF) [4], which originates from 2008. SDN and OpenFlow started as academic experiments, but in 2011 the Open Networking Foundation (ONF) [5] was founded, with the main goal of promotion and adoption of SDN through open standards development. ONF has been funded by (amongst others) Google, Facebook and Microsoft.

Nowadays, most vendors of commercial switches include support of the OpenFlow API in their equipment and the first SDN/OpenFlow deployments have arised. One of the first and largest of these deployments is Google's B4 Wide Area Network (WAN) [6], which has been running since 2010. B4 is a great example for the potential that comes with SDN to dramatically simplify network management and enable innovation and evolution. Namely, in B4, standard routing protocols and centralized traffic engineering are supported simultaneously. In this network the traffic engineering service drives links to near 100% utilization, where WAN links typically are provisioned to 30-40% utilization.

Despite the potential SDN brings and the numerous researches towards SDN solutions, there are still key

challenges which obstruct the adoption of SDN in enterprises [3, 7, 8]. Examples of these challenges are scalability, flexibility and security. In this research the focus will be on the security challenges. Namely, besides that we need to know how to implement threat detection and mitigation, it turns out that the decoupled control layer introduces a new collection of security challenges. For example, what happens when the controller doesn't function well? A compromised controller may compromise the entire network.

Before investigating security challenges of SDN, it is important to have a complete understanding of the paradigm itself. Therefore, this chapter functions as a literature survey towards SDN in general. We will look into the architecture of SDN itself, as well as into the details of the control layer and its interfaces. Most principles explained in this chapter will be referred to in the next chapters. In Chapter 2 we will start diving into SDN security in general, before focusing on particular problems and research questions in the rest of this research.

1.2. What is Software-Defined Networking?

Originally stated, a Software-Defined Network is a network where the forwarding state in the data plane is managed by a decoupled, remote control plane. [3] defines an SDN as a network architecture with the following four pillars:

1. The control and data planes are *decoupled*, which results in network devices becoming simple (packet) forwarding elements.
2. Forwarding decisions are flow-based, instead of destination-based. In the SDN/OpenFlow context, a flow is a sequence of packets between a source and a destination, which (thus) all receive identical service policies at the forwarding devices.
3. The control logic is moved to an external entity, the so-called SDN controller or Network Operating System (NOS).
4. The network is *programmable* through software applications running on top of the NOS that interacts with the underlying data plane devices.

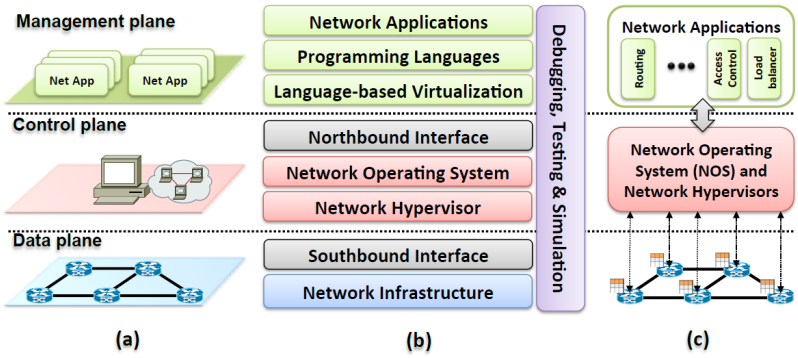

Figure 1.1: Software-Defined Networks in (a) planes, (b) layers, and (c) system design architecture [3]

Given these pillars, the SDN architecture can be viewed in different perspectives, show in Fig. 1.1. As stated, the data plane will contain simple forwarding elements, to which different hosts are connected. The forwarding elements use information in their so-called *flow tables* to find out where to forward packets to. The control plane is the layer where all calculations are done and decisions are made. The controller has knowledge of the network topology, performs routing and informs the forwarding elements in the layer below by updating their flow tables. This is done using the southbound interface, using a protocol such as OpenFlow [4]. Just as in traditional networks, the top layer is the management plane. In the SDN context this layer contains network applications, which are used to be the 'network brains' [3]. They implement the control-logic that will be translated into commands to be installed in the data plane, dictating the behavior of the forwarding devices. The translation process is done by the controller. Examples of network applications are mobility management, traffic monitoring and load balancing.

The software-defined network forwarding functionality is intuitively explained in [7]. The process starts when a switch, say S_1, receives the first packet of a new flow from a host (the sender). At this moment, S_1 checks for a flow rule for this packet in his personal flow table. If a matching entry is found, the instructions associated with the specific flow entry (e.g. update the counter, action set or metadata) are executed and packets are forwarded towards the receiver. If no match is found in the flow table, the packet may be forwarded to the controller over a secure channel. The controller then executes the routing algorithm and adds a new forwarding entry to the flow table of S_1 and to each of the relevant switches along the flow path. Finally, S_1 forwards the packet to the appropriate port to send the packets to(wards) the receiver.

As is clear, the controller can add, update and delete flow entries. It can do this *reactively* and/or *proactively*. The example above is one of a reactive flow instantiation, where the controller only acts in response to received packets from switches. In the proactive case, flow tables are populated beforehand like it is done in a typical routing table in current networks. Proactive flow tables eliminate any latency induced by consulting a controller on every flow, because most of the times, the flows are already installed before communication is initiated. Although both types of flow instantiation are possible, one should always keep in mind that the eventual behavior of the controller is completely determined by its implementation.

1.3. Concepts related to SDN

One who is doing research on software-defined networking will almost always come across other new (networking) concepts. Currently in the enterprise world, people get more and more interested in concepts like *network automation*, *network virtualization* and *software-defined infrastructure*, while not always knowing the difference. Coming with this trend, well-known 'networking bloggers', such as Matt Oswalt [9], try to clarify the differences and relations between these concepts. Because this research is performed in an enterprise environment (*Fox IT*), we will shortly expand on the differences between SDN and the concepts named above.

Network automation

Firstly, software-defined networking is not the same as network automation. An example of network automation is the automatically processing of configuration changes in a consistent manner - an important challenge in current (and future) networks. The industry is in agreement that network automation should be pursued in *all* future networks. While SDN is not network automation by design, is does *introduce possibilities* for automation by creating new programmable possibilities for network management and configuration methods.

Software-Defined Infrastructure (SDI)

Secondly, SDN is sometimes seen as an implementation of software-Defined Infrastructure (SDI). However, in first instance SDI is not related to the actual network while SDN is. SDI is focused on providing efficient IT services by servicing workloads automatically by the most appropriate resources and is used for analytics, mobile, social and cloud environments / applications. To provide these solutions, it could be the case that the SDI *uses* a software-defined network, but it could also rely on other network architectures.

SDN and Network Function Virtualization (NFV)

Lastly, SDN is sometimes mistaken for network virtualization. A well-known virtualization concept is Network Function Virtualization (NFV) [10], a concept born October 2012. Here, the idea is to decouple network functions (NFs) from the physical devices on which they run. A NF is a functional block within a network infrastructure that has well-defined external interfaces and well-defined functional behavior, such as DHCP servers and firewalls. In short, one wants to virtualize network functions as an alternative to using high-cost, purpose-built appliances.

NFV and SDN have a lot in common since they both advocate for a passage towards open software and standard network hardware. They may also be highly complimentary, which means that combining them in one networking solution may lead to greater value. However, they are different concepts, aimed at addressing different aspects of a software-driven networking solution. NFV aims at decoupling NFs from specialized hardware elements while SDN focuses on separating the handling of packets and connections from overall network control. Also, the research in NFV is driven by different telecom service providers and is formalized by the European Telecommunications Standards Institute (ETSI). The comparison of the SDN and NFV concept is summarized in Fig. 1.2.

Issue	NFV (Telecom Networks)	Software Defined Networking
Approach	Service/Function Abstraction	Networking Abstraction
Formalization	ETSI	ONF
Advantage	Promises to bring flexibility and cost reduction	Promises to bring unified programmable control and open interfaces
Protocol	Multiple control protocols (e.g SNMP, NETCONF)	OpenFlow is de-facto standard
Applications run	Commodity servers and switches	Commodity servers for control plane and possibility for specialized hardware for data plane
Leaders	Mainly Telecom service providers	Mainly networking software and hardware vendors
Business Initiator	Telecom service providers	Born on the campus, matured in the data center

Figure 1.2: Table comparing the SDN and NFV concepts. [10]

1.4. The SDN controller

With the separation of the control- and data plane, network switches become simple forwarding devices and the control logic is implemented in a *logically centralized* controller or network operating system (NOS). The controller or NOS follows the same functionality as traditional operating systems, to provide abstractions, essential services and common APIs to developers. The control platform abstracts the lower-level details of connecting and interacting with forwarding devices and is therefore a critical element in an SDN architecture. In this section we will look into the control layer and its connection to the data plane in full detail.

There are a variety of core controller functions that every controller should provide [3]. First, these are functions like program execution, I/O operations control and communications, following base services of an operating system. Second come essential network control functionalities that network applications may use in building its logic, such as topology, statistics and shortest path forwarding. Last, a controller needs to support security mechanisms to provide basic isolation and security enforcement between services and applications. For instance, a controller needs to make sure that rules generated by high priority services should not be overwritten with rules created by lower priority applications.

1.4.1. Controller architecture

Although the controller is always logically centralized, it is important to emphasize that this programmatic model does not postulate a physically centralized system. Current controller implementations follow a *centralized* or *distributed* architecture [3].

When having a centralized controller, we have a single entity that manages all forwarding devices of the network. Naturally, it represents a single point of failure and may have scaling limitations. Most centralized controllers have been designed as highly concurrent systems to achieve high throughput required by enterprise class networks and data centers.

A distributed controller can be a centralized cluster of nodes or a physically distributed set of elements. The first alternative can offer high throughput for very dense data centers, while the second can be more resilient to different kinds of logical and physical failures. Distributed controllers need to deal with data consistency. Controllers support a weak or strong consistency model. Naturally, a strong consistency needs a lot of updates to keep all nodes informed at all times, which has an impact on the system performance. In systems with weak consistency it could be the case that, for a period of time, distinct nodes read different values for a same property. Distributed controllers communicate with each other over the so-called *east-/westbound interface*, following the names of the interfaces between the different layers.

1.4.2. The north- and southbound interface

As can be seen in Fig. 1.1, the control layer is connected to the management plane (the network applications) via the northbound interface and to the data plane via the southbound interface. Controllers differ in what kind of APIs they support on these interfaces. At the northbound interface, most controller implementations support all kind of APIs as long as they follow the constraints of the architectural style REST (Representational State Transfer) [11]. An example of an RESTful protocol is HTTP. Further elaboration on REST lies out of the scope of this research. An interested reader is adviced to start at [11].

The southbound interface is where the communication between the controller and forwarding devices of

the SDN architecture is defined. The standard communication interface which is supported by almost every controller is OpenFlow (OF) [4]. Currently OpenFlow versions up to 1.5 exist, while version 1.3 (v0x04, or OF1.3) is seen as the most important version. It differs which version is supported by different controllers and forwarding devices.

While OpenFlow efficiently manages flows and determines how packets are forwarded through the network, it does not provide the configuration and management functions necessary to allocate ports or assign IP addresses. For this, configuration protocols are designed, which are usually combined with OpenFlow. Two of such protocols are OF-Config [12] and the Open vSwitch Database Management Protocol (OVSDB) [13, 14]. As the names suggest, OF-Config was designed to apply to all OpenFlow implementations (on both physical and virtual switches), while OVSDB was designed specifically to manage Open vSwitch implementations. A third management protocol is NETCONF [15]. As a last example there is OpFlex [16], which is a protocol which focuses on the translation of network policies to actions inside networking devices.

1.4.3. Control channel architecture

Most of the time it is said that the communication between the controller and data plane is transferred over a *control channel*. There are two different ways of handling this communication (i.e. handling the OpenFlow connections). The connections can either use a dedicated control network separated from other network traffic, or the OF packets can be transmitted over the same physical network as the actual network traffic. In this scenario, control traffic may be sent through other OF switches. The first method is referred to as a *out-of-band* control channel and the latter as a *in-band* control channel, depicted in Fig. 1.3.

As will be elaborated on in Chapter 3, the used control channel affects the security of the controller-switch communication. Shortly, *out-of-band* control is less vulnerable to attacks but more expensive because of the need for separate cabling and interfaces.

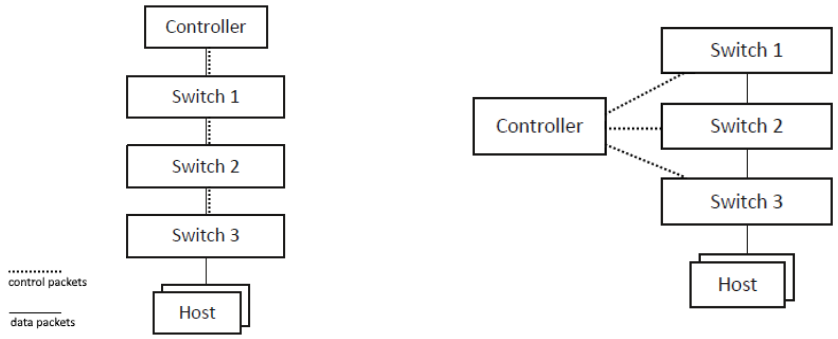

Figure 1.3: Example of a small network using a in-band control channel (*left*) or a out-of-band control channel (*right*). In the in-band case, control and data packets are transported over the same physical connections.

1.4.4. Advances in controller implementations

Nowadays, a variety of controller implementations exists, which follow different architectures and are written in different programming languages. Table 1.1 shows a list of the most well-known SDN controllers, with their architecture and programming language. To support the global idea to *open* standards in networking, almost all controllers are Open Source software. For a more elaborate list of SDN controllers, one should consult [3].

The first highly popular OpenFlow controller was NOX [17]. Although it is not heavily implemented or used because of shortcomings with its implementation and development environment, it was the initial spark for SDN. NOX used to support both C++ and Python, but nowadays the Python support is provided by its successor POX [18], which was built as a friendlier alternative. Another friendly SDN controller that uses Python is Ryu [19], which is well-known particular in university environments, because it is relatively easy to setup an SDN in Python and Ryu supports the OpenFlow 1.3 version (and higher).

The next big step in open source controllers came with Beacon [20]. Beacon is known as a well written and organized SDN controller, written in Java and is integrated into the Eclipse IDE. Beacon's successor is Floodlight [21]. As Ryu, Floodlight supports the OpenFlow 1.3 version. Attractive about Floodlight is that it has a large number of features that can be added to create a system that best meets the requirements of a specific organization, and that it has both a web based and a Java based GUI. Beacon, Floodlight and Trema

Controller	Architecture	Prog. Language	OF support
NOX [17]	centralized	C++	v1.0
POX [18]	centralized	Python	v1.0
Ryu NOS [19]	centralized	Python	v1.{0,2,3,4,5}
Beacon [20]	centralized multi-threaded	Java	v1.0
Floodlight [21]	centralized multi-threaded	Java	v1.{0,3}
Trema [22]	centralized multi-threaded	Ruby, C	v1.0
Onix [23]	distributed	Python, C	v1.0
ONOS [24]	distributed	Java	v1.{0,3,4,5}
OpenDaylight [25]	distributed	Java	v1.{0,3}

Table 1.1: Overview of the architecture and programming language of some well-known SDN controllers

[22] (a controller for Ruby) are known to be centralized multi-threaded controllers. They target specific environments such as data centers and try to explore the parallelism of multi-core computer architectures.

Besides the centralized controllers, Onix [23], ONOS [24] and OpenDaylight [25] are well-known distributed controllers. Although it is more difficult to get started with these controllers, one sees that these controllers are more focused on industrial use. Like Floodlight, both ONOS and OpenDaylight have a number of pluggable modules that can be used to alter it to the needs of an organization. In fact, the two controllers are much alike, but they show a difference in focus [26]. Namely, while ONOS has focused on service providers' needs, OpenDaylight was created to be the 'Linux of networking': one platform to have a very long life and enable people to build a wide range of solutions to solve a wide range of problems. As an example, ONOS has a role as a local controller for AT&T, while AT&T is using the OpenDaylight controller as its basis for its global SDN controller.

The popularity of each controller is indirectly determined by the version of OpenFlow it supports. In practice, we see that controllers which don't support OF1.3 don't have an active community which focuses on further development of the controller. Currently, OpenDaylight is leading the transformation to Open SDN and is regarded as the industry's de facto standard. Its release supports both SDN and NFV and is intended to be scaled to very large sizes.

Specialized implementations

Shortly stated, the controllers in Table 1.1 can be seen as 'general purpose' SDN controllers. During the evolution of SDN, solutions to different identified problems are created. These solutions occur as network applications implemented on top of a controller, extensions to existing controllers (such as NOX) but also as new controllers which specialize in solving one or more existing problems. In the latter category, Rosemary [27] offers specific functionality to guarantee the security and isolation of applications. Fleet [28] is a distributed controller which aims to solve the malicious administrators problem. Well-known responses to shortcomings in the NOX controller are implemented in FortNOX [29] and FRESCO [30]. We will look deeper into these and other specialized implementations in Section 2.3.

Commercial controllers

The controllers elaborated on above are controllers which find their origin in the academic world. Nowadays, development of ONOS and OpenDaylight are supported by the Linux Foundation [31] and Open Network Foundation [5]. Besides this also commercial controllers arise, from companies such as Juniper Networks, Cisco and Hewlett Packard Enterprise (HP(E)). During this research, we experimented with the HPE Virtual Application Networks (VAN) SDN controller[1] [32]. While commercial, this controller can be freely downloaded for trial use. HPE VAN is a controller which has a lot in common with OpenDaylight. This is due to the fact that before focusing on a own controller, HPE contributed a lot of code towards OpenDaylight. Eventually, the HPE VAN controller differs from OpenDaylight in focus, because it focuses on deployment in 'campus' networks. HPE VAN supports OpenFlow 1.3 and also has a web based GUI to configure the controller and load different modules.

[1] Information about and support with this controller was received via direct contact with employees at *HP Enterprise*.

2

Introduction to SDN Security

Software-defined networks encounter a variety of challenges which have obstructed the total take-over of SDN in enterprise networks [7, 8], one of these being security. Current SDN-based security research can be categorized into two branches [8]: (1) research geared towards *protecting the network*, which deals with security configuration, threat detection, remediation and verification in and using SDN and (2) research focusing on providing *security as a service*, elaborating on the development of innovative security capabilities that can be instantiated on-the-fly using SDN. Our research falls in the first category, and the goal of this chapter is to provide an introduction to the security of SDN itself. We will also define our research questions for the remainder of this study.

First, Section 2.1 will explain what the conceptual SDN threat vectors are and give some concrete examples of (general) security challenges. After this, we will start focusing on the security of the control channel, while focusing less on experimental research towards the security of controller applications, management layer and thus the application-control plane. We use Section 2.2 to elaborate on the motivation for this choice, define our research questions and set out the structure for the rest of the report. While our experimental research will focus on the lower part of the network, for completeness, Section 2.3 discusses the studies towards security challenges and solutions for the upper part of the network we've encountered during the literature study.

A note on robustness, resilience and security

In the evolution of the SDN paradigm, three terms have played an important role: *Security*, *robustness* and *resilience*. Before going on, it is important to know the difference between these three concepts. Namely, although these terms lie closely together and have to do with network failures, they are distinct.

First, robustness is a network property. It can be seen as adding 'redundancy' to the network, such that if a failure happens, traffic can be rerouted. Resilience is linked to robustness and is defined as the ability to recover the control logic after a failure [33]. Here, the emphasis is more on the protocols [34] and algorithms [35] in fail-over than the actual added redundancy.

Robustness and resilience can be discussed in context of any type of failure. These failures can also be the result of malicious behavior or attacks. Following this context, security is the *protection against malicious behavior*.

2.1. SDN Threat Vectors

SDNs have two properties which can be seen as attractive honeypots for malicious users and a source of headaches for less prepared network operators [36]. First, the software which gives us the ability to control the network is always subject to bugs and other vulnerabilities. Second, the centralization of the network intelligence in the controller means that anyone with access to the servers that host the control software can potentially control the entire network - making it very attractive for attackers to focus their attacks on the controller. As an specific example, a malicious controller (or application) could be used to reprogram the entire network for data theft purposes in a data center.

The critical security threats of SDN are identified with the use of threat vectors [3, 36], which are shown in Fig. 2.1. Some of these vectors are also common to existing networks, while others are more specific to SDN.

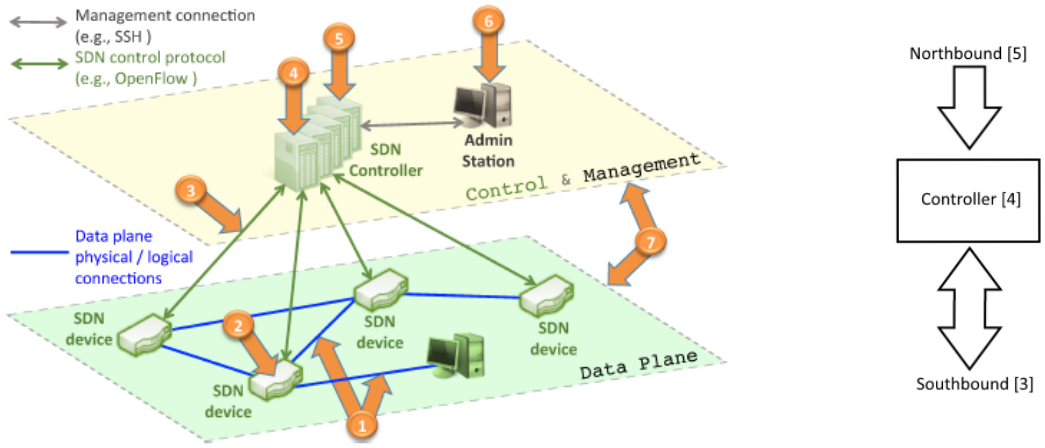

Figure 2.1: Main threat vectors of SDN architectures (*left*) [36] and a simplified version of the threat vectors specific to SDN (*right*).

As will turn out, almost every threat vector can be exploited to reach the controller and damage the network. Below we will shortly discuss all threat vectors. A summary is given in Table 2.1.

The first threat vector consists of forged or faked traffic flows in the data plane, which can be used to attack forwarding devices and controllers. An example is the creation of a denial-of-service (DoS) attack against OF switches and controller resources, which could eventually bring down the controller. Threat vector two stands for attacks on vulnerabilities in switches itself, which could be used to drop or slow down packets in the network. It could even be possible to forge requests to overload the controller or neighboring switches.

Vectors three, four and five are seen as the most dangerous ones, because via these ways network operation could be completely compromised. These vectors are SDN specific, and symbolize attacks on the control plane communications (control channel), attacks on and vulnerabilities in controllers and the lack of mechanisms to ensure trust between the controller and management applications. Trust issues can also arise on the control channel; here between controller(s) and forwarding device(s). From both the northbound- and southbound interface it could therefore be possible to reach the controller via compromised components. Vulnerabilities in controllers (threat vector four) are dangerous because the use of a common intrusion detection system may not be enough to find the exact combination of events that trigger a particular behavior in the controller, and to label this as malicious.

The sixth threat vector symbolizes attacks on and vulnerabilities in administrative stations which, as it is also common in traditional networks, are used in SDNs to access the network controller. Also here, it is possible for malicious users to reach the controller. The last threat vector represents the lack of trusted resources for forensics and remediation, which can compromise investigations and preclude fast and secure recovery modes for bringing the network back into a safe operation condition.

#	Description	Consequences in SDN
1	Forged / faked traffic flows	Open door for (D)DoS attacks
2	Attacks on and vulnerabilities in switches	Potential attack inflation
3	Attacks on the control channel	Communication with the controller can be explored / exploited
4	Attacks on and vulnerabilities in controllers	A compromised controller may compromise the entire network
5	Lack of trust mechanism between controller and applications	Malicious applications can now be easily developed and deployed on controllers
6	Attacks on and vulnerabilities in administrative stations	Potential attack inflation, can eventually reach the controller
7	Lack of trusted resources for forensics and remediation	Negative impact on fast recovery and fault diagnosis

Table 2.1: Overview of SDN security threat vectors. Vectors **3**, **4** and **5** are *SDN specific* threats.

2.1.1. Examples of security challenges

Using the conceptual threat vectors, we can come up with several examples of security challenges. Denial-of-service (DoS) was already mentioned above as something that can initiated from the forwarding devices. Although it is not a SDN specific threat, it is an important security challenge. [37] gives a feasibility study where a so-called fingerprinting method is used as a first step towards a DoS attack in an reactive SDN (where flow tables are filled in response of communication, as explained in Section 1.2). The fingerprinting method is used to measure the delay experienced by the first packet of a flow to infer that the target network is a reactive SDN, after which a DoS can be initiated. Besides this, as stated in [3], every network should also be protected against security issues like spoofing, tampering, repudiation, information disclosure and elevation of privilege. Again, these are no SDN specific threats, but should also be considered in SDN environments.

Another important security concern for the control channel is that the use of encryption here is optional. While stated mandatory in OF1.0, the usage of Secure Sockets Layer (SSL) and Transport Layer Security (TLS) was changed to an optional feature in OF1.3 and higher. The usage of encryption has a large influence on the security of the control channel.

As discussed, threat vector four concerns vulnerabilities in controllers. [27] states that current controllers such as Floodlight, OpenDaylight, POX and Beacon have several security and resilience issues. As an example, a simple malicious action such as changing the value of a data structure in memory can also directly affect the operation and reliability of the controllers.

Attack possibilities can also arise due to network applications. A concrete example is the GUI application for controllers like ONOS and OpenDaylight. This application is accessed via HTTP, where the administrator needs to enter his login credentials. [38] shows how one can retrieve these credentials using a man-in-the-middle (MitM) attack, after which one is capable to login to the controller GUI. Depending on the possible functions the GUI has implemented, in the worst case an attacker can take over the entire network.

Also, extra measures need to be taken into account when we use controllers with a distributed architecture. In an interview [39], Chris Hoff (Chief Security Architect at Juniper Networks) states that the centralized controller architecture with a single controller that configures the network will not be the architecture that makes it to deployment. He expects there to be multiple controllers in heterogeneous environments interacting with lots of other controllers. In this case, one needs to make sure that the communication between the different controllers (the east-/westbound interface) is secured and every controller knows which controller is to be trusted.

Lastly, policy enforcement is widely researched in SDN, as we will see in Section 2.3. Policy enforcement (also know as policy conflict resolution) is not directly a weakness, but could be as soon as it is not implemented well. It will be a direct threat when a malicious application wants to make changes to the eventual forwarding rules, but the controller should also make sure that the policies of 'correct' applications can be carried out in the network at moments that they push their policies on top of each other. A controller needs to make sure that rules generated by high priority services should not be overwritten with rules created by lower priority applications. For clarity, policy enforcement falls into the category of threat vector five.

2.2. Investigating SDN Security

Given our conceptual understanding of SDN security challenges, for this research it was important to define a delimited area which is going to be investigated in full detail. For this decision several things played a role, such as the research environment and outcome of the literature study. In this section, we will define the research questions and associated structure for the remainder of this report.

2.2.1. Research motivation

From Section 2.1 it is clear that the introduction of the separate control plane leads to new security threat vectors, concerning the controller itself as well as its relation (and communication) with network applications and forwarding devices. From a controller's perspective we can describe these new threats in a simplified manner as depicted on the right of Fig. 2.1. Shortly, the controller can be attacked/compromised via the northbound interface, via the southbound interface or via direct attacks or vulnerabilities. This simplified view is used to define our research area.

From the literature study, it became clear that vector five has been investigated the most, resulting in solutions for network verification and policy enforcement, with the goal that the controller can support multiple

network applications and is protected against malicious ones. This trend somewhat conflicts with security analyses presented to date (such as [40]), which focus on control-data plane issues. Besides this, most proposed security solutions presume that there are no network design vulnerabilities with the protocol, thus that the OpenFlow communication is 'safe' - while this is not necessarily so.

Another way to draw a line is as follows: Anomalies at the northbound interface can arise due to incorrect management of desired network functionality, while problems at the control channel and dataplane itself are due to concrete malicious actions which influence the security of the network 'core'. This research is performed with help of the IT Security company *Fox IT*, which houses a large *audits* department where there is a lot of knowledge about networking and networking protocols (in general). To use the available knowledge of the company to its fullest and at the same time provide a study which is useful for different IT security companies and has scientific merit, we want to focus on an research direction which can be approached like we are playing 'attackers'. Namely, while security analyses do focus on the issues on the lower part of the network, they do not focus on the particular ways to perform possible attacks [40]. As follows from the mindset at Fox IT, this is a good way to gain much knowledge about a problem. With this in mind, it is more beneficial to investigate the security of the network core instead of network management.

It turns out that it is less applicable to search towards vulnerabilities in controllers. Vulnerabilities in controllers (and other software programs) can be triggered by changing the value of a data structure in memory. Attackers most of the time find these kind of vulnerabilities by doing 'trial-and-error'. Trying to find new ones in existing controllers will therefore not lead to a structured research. Besides this, because the SDN and its controller are currently 'hot' topics, it could be the case that found vulnerabilities are resolved in next controller updates.

2.2.2. Research questions

From above, it is clear that we will focus on the security of the software-defined network's core. We will perform our investigation in a layered approach. Firstly, we will focus on the control channel and OpenFlow, to see which vulnerabilities arise in the protocol and what kind of attacks are possible when for example a switch is taken over by a malicious user. Examples of attacks we look into are controller hijacking and flow rule modification, both issues which are not yet investigated in full detail [40]. Secondly, we focus on the current state of SSL/TLS in SDN to see to what extent this solves problems. Lastly, we look into the possibilities hosts have to attack an SDN without usage of OpenFlow messages. Here we try to create feasible scenarios for a denial-of-service attack, to get an insight whether this is indeed as easy as literature states.

Concretely, we try to answer the following research questions:

1. Is it possible to misuse the OpenFlow protocol to attack forwarding devices, controllers or the control channel in general and what possibilities lie there for a malicious user who has taken over a network component?

2. Does the usage of SSL/TLS solve problems found in the previous question and is it always feasible as a solution or does it introduce new problems?

3. In what ways can a regular host damage an SDN?

In order to aid in finding answers to the research questions, sub-questions will be formulated at the beginning of each chapter where we focus on a particular question. The layered approach is chosen such that in its totality, this research can be used by security experts (such as those at Fox IT) new in the field of SDN. The overall goal of the research is to provide security experts and network architects a kind of directive for when they are checking the security guarantees of future software-defined networks. It is therefore also a goal to provide some 'best practices' when creating an SDN. Clearly, during this research not only SDN specific problems are investigated but also traditional problems which perhaps will change form when applied to SDN.

In this research, we will not investigate the security of the upper part of the network. For completeness we include some findings of the literature study where we did encounter solutions for problems which arise here. These solutions are discussed below in Section 2.3, mostly for the interested reader. Also, vulnerabilities in the controller itself and eventual problems which arise with controller-to-controller communication (the east-/westboundinterface) will not be investigated.

Solution	Implementation	Vector	Main Purpose
FortNOX (2012) [29]	NOX controller extension	5	Security flow rules prioritization (network verification)
FLOVER (2013) [46]	NOX controller extension	5	Security flow rules prioritization (network verification)
SE-Floodlight (2015) [47]	Floodlight controller extension	5	Multi-app support (incl. network verification)
FRESCO (2013) [30]	API layer on top of (Fort)NOX	4, 5	Providing a framework for security services composition
Rosemary (2014) [27]	(centralized) NOS / Controller	4, 5	Offering security and isolation of applications
Fleet (2014) [28]	(distributed) NOS / Controller	6	Solving the malicious administrator problem

Table 2.2: SDN solutions mentioned in Section 2.3, how they are implemented, which threat vector they target and their main purpose.

2.2.3. Report outline and research approach

The outline for the remainder of this report follows our research questions: We answer the first in Chapter 3, the second in Chapter 4 and the third in Chapter 5. Using our answers, we conclude this report with Chapter 6, where we discuss the implications on and best practices for the security of the SDN control-data plane.

As stated before, in each chapter we will formulate sub-questions to aid us in finding answers to the research questions. The answers to the sub-questions are (mostly) found in an experimental way, performing different attacks. Before focusing on these attacks in detail, we will shortly discuss a *threat model* to clearly state situation and the possibilities (capabilities) of the attacker performing that attack.

Our experiments are performed in both a simulated and a small physical SDN environment. In this first case, we rely on Mininet [41] to setup an SDN network, where the used virtual switches are Open vSwitch [42]. In all our setups, all entities only have an IPv4 address. The physical network is built using the Zodiac FX OpenFlow switch [43]. Network traffic is investigated using Wireshark [44]. The scripts we've written are written in Python and rely on OpenFlow 1.3 [45]. The investigated SDN controllers are Ryu [19], ONOS [24], OpenDaylight [25] and HPE VAN [32]. The controllers are used in their standard implementation, no code is altered. As extra information: for Ryu, we ran the `simple_switch_13.py` module and for OpenDaylight, we installed the `odl-l2switch-switch` and `odl-dlux-all` features which are provided by OpenDaylight itself.

In our report, we will elaborate on the exact experimental setup, attack implementations and used software and hardware per experiment. Also, Appendix C elaborates more on particular used hardware and software in order to give additional explanations for those who are interested to use these products for own experiments.

2.3. Advances in Security Solutions for the Application-Control Plane

Several studies identify and propose solutions for issues that arise at the application-control plane, where the majority focuses on the policy conflict resolution problem [40]. To create a complete view on SDN security for the (interested) reader, we will now discuss such studies in the context of related work. Per solution we shortly elaborate on the problem it tries to solve and how it does this. A summary can be found in Table 2.2.

As stated in Section 1.4, we see that security solutions for the control and management layer occur in different flavors. Firstly, there are network applications that improve security and dependability, thus are implemented as options *on top of* the controller. Secondly, we see solutions that extend controllers, thus are directly implemented *into* existing controllers. Lastly, we also see solutions embedded in completely new controllers.

Two examples of extensions to the NOX controller, both focusing on network verification (also seen as policy conflict resolution), are FortNOX (2012) [29] and FLOVER (2013) [46]. In FortNOX, this extension is called a security enforcement kernel. The main purpose of both solutions is to provide non-bypassable policy-based flow rule enforcement over flow rule insertion requests from OpenFlow applications (for example in response to perceived threats). In other words, it addresses the problem of verifying that the current state of flow rules inserted in a switch's flow table remains consistent with the current network security policy. In the same category, SE-Floodlight (2015) [47] is the implementation of the security enforcement kernel for the Floodlight controller. As with FortNOX and FLOVER, SE-Floodlight focuses on multi-app support. It provides solu-

tions for application co-existence in general, an application permission model, application accountability and privilege separation.

The FortNOX extension is used by FRESCO (2013) [30], an application development framework facilitating the design of sophisticated threat detection and mitigation modules. The goal of FRESCO is to simplify the development of security applications, for which it provides an own script language. With FRESCO it is thus possible to write your own threat detection modules. It also contains some basic reusable modules for security purposes. As as example, [30] presents the implementation of a *reflector net*, which allows OpenFlow network operators to redirect malicious scanners to a third-party remote honeypot.

Another focus category is permission systems for applications. PermOF (2013) [48] is a fine-grained permission system that tries to apply minimum privilege on applications and focuses on solving the problem of lack of trust between the controller and the application. PermOF categorized OpenFlow instructions into 18 permissions and assigns the permissions to applications, which are isolated according to these permissions.

Another example is Rosemary (2014) [27]. Rosemary is a new NOS that tackles security issues which were identified in controllers such as OpenDaylight and POX. Two of these identified problems were that there is no separation for applications from the NOS and that the NOSs assume that network applications are all trustworthy. Rosemary's design solves these problems by explicitly separate the network applications from the trusted computing base of the NOS and monitor applications and resources. Also, network applications must explicitly possess capabilities to access a particular resource of the NOS - it thus works with a permission structure. Rosemary is written in C and supports both the OpenFlow 1.0 and 1.3 specification.

Lastly there is Fleet (2014) [28], a distributed controller which aims to solve the malicious administrators problem. Here, a switch intelligence layer operates on top of switches and mediates communication between switches and the so-called administrator layer, while this layer consists of a set of physically separated machines to which administrators upload their network configurations and a shared data storage system. Inside the administration layer, a voting mechanism using threshold signatures is leveraged. Eventually, an instruction is only accepted when the number of administrators who agree on the instruction reaches a predefined threshold. This threshold will always be larger than the maximum malicious administrators that is allowed in the network.

Retrospect on solution suitability

The solutions in Table 2.2 are not the only proposed security solutions, but give a proper overview of the different kind of investigated solutions. In the early days of the SDN paradigm itself, solutions to shortcomings of the controller were mostly developed on top of or into the NOX controller. Namely, in a list with developed security network applications presented in [3], 14 of the 20 solutions are deployed onto the NOX controller - among which we find FortNox, FLOVER and FRESCO. These solutions will probably mostly serve as interesting proof-of-concepts but less probably be deployed in enterprises, because the NOX controller is not frequently used in enterprise networks due to its implementation shortcomings (see Section 1.4).

Without looking at the correctness of the solutions, a lot is unknown about the eventual suitability. This is due to the fact that we have to wait and see what kind of controller implementations break through to become truly 'usable' for the industry. Are the solutions for NOX correct concepts that can be deployed on newer controllers? Besides this, stated is that OpenDaylight is leading the evolution in the SDN industry, but up to date no particular security analysis is present about the security deployability of the controller. While Rosemary covers OpenDaylight, this solution mostly focuses on the isolation of applications and results in a centralized controller, which is known to represent a single point of failure.

Last, for eventual releases of solutions in commercial controllers, one needs to investigate to which extent enterprises such as HP actually use academical proposals. For the HPE VAN case, it is known that the controller is closely related to OpenDaylight. Here, solutions and concepts could probably be re-used, but it may be less suited in commercial controllers developed far away from the academic world.

3

Investigating the Control-Data Plane

In Section 2.2 we defined our research questions, of which the first one focuses on the control-data plane in detail. Possible misuses in this lower part of a software-defined network are controller hijacking and flow rule modification. In this chapter we target 'simple' (and unencrypted) SDN implementations and we will investigate the possibilities for malicious activities when we misuse the OpenFlow protocol or when a particular network component has been taken over. More concretely, we want to find answers to the following questions:

- Is it possible, using the OpenFlow (OF) protocol, to impersonate a switch or controller towards the original controller, and if so, to what extent is it possible to retrieve information from or give information to the controller in order to influence the network behavior?
- Is it possible, using the OpenFlow protocol, to impersonate a controller towards forwarding devices, and if so, to what extent is it possible to influence the behavior of these forwarding devices?
- What are the possibilities for malicious network devices?
- In what way can we defend ourselves against the problems stated above?

In this chapter, the answers to the first two questions are retrieved in an experimental way using an attacker's point of view. Besides this, we focus on gaining insights into the OF protocol itself to get a grip on the particular communication between the controller and forwarding devices, which we do in Section 3.1. Our experiments are discussed in Section 3.2 to Section 3.4. The possibilities for malicious network components are discussed in Section 3.5. In Section 3.6, we end with our conclusions of the total investigation as well as a dive into solutions to the found vulnerabilities. But before this all, we elaborate on the used threat model in this chapter.

Threat model

When describing all attacks in this chapter, we use an attacker's point of view. We assume that the attacker is in possession of a direct connection towards the controller or switch (s)he want to attack, or has the needed information/tools to set up such a link. As will be elaborated on in Section 3.2, for this one needs an IP-address and TCP-port. The investigated networks don't make use of TLS/SSL. When having the connection towards a controller or switch, the attacker is able to follow the OpenFlow protocol - and has correct knowledge to do so. For our experiments, the attacker doesn't need physical access to the devices (s)he wants to attack.

3.1. Investigating the OpenFlow Protocol

To get a broad view of the OpenFlow protocol and how communication using this protocol works, we focused on the OpenFlow protocol specification [45] and 'simple' OpenFlow communication. For the latter, we can simply rely on a virtual machine (VM) with Wireshark [44], Mininet [41], Open vSwitch [42] and the Ryu controller [19] installed - such VM's are easy to find as soon as one searches the Internet for 'SDN Ryu Tutorial'.

OpenFlow Message Name	Value	Message Type
OFPT_HELLO	0	Symmetric message
OFPT_ERROR	1	Symmetric message
OFPT_ECHO_REQUEST	2	Symmetric message
OFPT_ECHO_REPLY	3	Symmetric message
OFPT_FEATURES_REQUEST	5	Controller/Switch message
OFPT_FEATURES_REPLY	6	Controller/Switch message
OFPT_GET_CONFIG_REQUEST	7	Controller/Switch message
OFPT_SET_CONFIG	9	Controller/Switch message
OFPT_PACKET_IN	10	Asynchronous message
OFPT_PACKET_OUT	13	Controller/Switch message
OFPT_FLOW_MOD	14	Controller/Switch message
OFPT_MULTIPART_REQUEST	18	Controller/Switch message
OFPT_BARRIER_REQUEST	20	Controller/Switch message
OFPT_ROLE_REQUEST	24	Controller/Switch message

Table 3.1: A collection of OpenFlow Message Types and their corresponding values, which are used and encountered during our experiments. Note that omitted replies have a value $request + 1$.

Using Wireshark to monitor the network traffic while running Mininet with a topology with a single switch (with an arbitrary number of hosts) connected to the `simple_switch_13.py` example of the Ryu controller gives you direct insight in the handshake procedure between switch and controller as well as the filling of the switch's flow table which in this case happens in a *reactive* manner. For clarification, `simple_switch_13.py` implements a controller which functions as a traditional 'learning switch' which uses OF1.3 (=**0x04**). Below we will elaborate on the insights we gained investigating the protocol.

OpenFlow messages

The first step is to understand the different OF messages that exist - all listed in [45]. In Table 3.1 we listed the OF messages which are important for and used in the remainder of this study.

As can be seen, there are three different types of messages. The *symmetric* messages can come from both switches and the controller. As an example, the switch sends a `ECHO_REQUEST` towards the controller to check the connection after which the controller answers with an `ECHO_REPLY`. The *controller/switch* messages are used for concrete controller-to-switch communications. Important to note is that the controller is always the entity which initiates this kind of communication. For example, it could thus not be that a switch sends a `FEATURES_REQUEST`. Lastly, there are only three different *asynchronous* messages, which occur asynchronously. The `PACKET_IN` is the most important: most of the cases it is a packet which is sent from the switch to controller when the switch didn't find a match in its flow table(s).

In Table 3.1 also the value of a specific message is listed, which is the actual value we see in the OF header when we investigate the packets with Wireshark. Using Wireshark, one can totally dissect the OF messages to see what is what - knowledge which is leveraged when forging OF messages. For example, the `HELLO` message {04 00 00 08 6e 3d be f1} (hexadecimal representation) can be dissected as follows:

Bytes	04	00	00 08	6e 3d be f1
Meaning	OF version 1.3	message type (`HELLO`)	length (8 bytes)	transaction ID

Table 3.2: Dissection of a OFPT_HELLO message.

The OpenFlow handshake

The complete handshake procedure can be found in [45], which is the same for every OF version. Shortly the handshake is as follows: It starts with a `HELLO` message from both sides. The overall understanding is that these messages arrive simultaneously. In experiments, it differs between controllers whether they send a `HELLO` before or after they have seen a `HELLO` from the connecting side. In all cases, after the controller has seen the `HELLO` from the connecting switch, it (also) sends a `FEATURES_REQUEST`. The switch replies then a `FEATURES_REPLY`.

Officially, these three messages are enough for the handshake procedure, but in practice some more communication takes place between the switch and controller before the controller knows enough to update its known network topology or send the first `FLOW_MOD` to the switch. We elaborate more on different handshake procedures in Section 3.2. Another interesting observation we have made during the investigation is that a switch also accepts (and correctly responds after) other messages than the expected `FEATURES_REQUEST` from the controller after its initial `HELLO`. This is leveraged in Section 3.3.

Difference in switch and controller behavior

As is already somewhat clear from the elaboration above, the behaviour of the switches and controller differ in their response to different OF messages. As an example, a (general implemented) controller instance doesn't know how to react to a `FEATURE_REQUEST`, while a forwarding device will send a `FEATURE_REPLY`. Again, controller-to-switch communication is always initiated by the controller. This behaviour difference can be used to identify particular OF connections, which we will investigate in Section 3.2.

Another OF message type which creates different responses from the different devices - and thus will be used in our investigation - is the `ERROR` message. As specified in [45], there are a lot of different `ERROR` messages. It turns out that the forwarding devices most of the time don't know what to do with errors, because they should be the devices that send these messages to the controller. The controller on the other hand, can trigger particular functionality when receiving an `ERROR`.

3.2. Identifying OpenFlow connections

Leveraging the OpenFlow handshake gives us several attack possibilities. As a first step, SDN controllers could be identified by the used sequence of OF messages. Or, particular OF messages could be connected to particular functionality (or applications) of a controller. Knowledge about the type and capabilities of the SDN controller gives attackers valuable information for planning their attacks. Below we show that we are able to find, identify and differentiate OF connections.

Finding possible OpenFlow channels

An attacker can connect to an OF device by setting up a socket towards the device's IP-address and an open TCP-port. We assume that the IP address of the OF device is known, after which we can search for a suitable TCP-port to connect to. In general, both standard (virtual) switches and controllers have an open TCP port which is dedicated to listen to incoming OF traffic. For our investigated controllers, this is `6633` or `6653`. For Open vSwitch, this is usually `6634`.

One can also use the `nmap` [49] package to scan for open TCP connections on a particular IP address. This way, we can easily find the open TCP connections of which probably some are dedicated to OF traffic. However, `nmap` isn't able to identify the particular TCP traffic, and will display an open TCP port to host an unknown service, as can be seen in Fig. 3.1. Although the service is unknown, the open TCP ports are a good starting point for an attacker.

Identifying OpenFlow connections

The identification of the OF connections is done using the Python script `controller_checker.py`, shown in Appendix A. This script makes use of the *pyof* library [50] to create fake OpenFlow traffic. The *pyof* library is a Python library which is able to generate most of the basic OpenFlow 1.3 messages. More information on this library can be found in Appendix C.1.

```
ubuntu@sdnhubvm:~[00:49]$ nmap 192.168.253.129 -p 1-65535

Starting Nmap 6.40 ( http://nmap.org ) at 2016-12-19 00:49 PST
Nmap scan report for 192.168.253.129
Host is up (0.00028s latency).
Not shown: 65532 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
6633/tcp open  unknown
6634/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 8.05 seconds
```

Figure 3.1: `Nmap` scan of 192.168.253.129. In our setup, this IP hosts a controller instance and one Open vSwitch. Expected is to find the controller on port `6633` and the switch `6634`, given the standard configuration.

```
ubuntu@sdnhubvm:~[04:09]$ sudo python3 ./attacks/v1.3/controller_checker.py 192.168.253.129 6634
[0, 2, 6]
OpenFlow connection with switch, version OF1.3
ubuntu@sdnhubvm:~[04:09]$ sudo python3 ./attacks/v1.3/controller_checker.py 192.168.253.129 6633
[0, 5, 24, 20]
OpenFlow OF1.3 connection with OpenDaylight controller
```

Figure 3.2: Results from `controller_checker.py`, which identifies a OpenDaylight controller instance and an Open vSwitch.

`controller_checker.py` works as follows: We start by creating a connection with the attacked device via a socket connection using `<IP,TCP>`. Then, we make use of the fact that in almost all OpenFlow communications (as is clear from Section 3.1), the controller is the entity that initiates a message exchange. The script is able to respond with correct (expected) OF responses to different received OF messages. While doing this, we store the message type of all received OF messages, up to the moment that we believe to have received enough information to compare this received message sequence with handshake sequences which are known for different controllers. These known handshake message sequences are found with the use of Wireshark and stated in the `check_message_sequence()` method. This method is used to perform the comparison step, after which we receive our output.

For the identification of a connection to an OF supporting switch, the approach above is altered a bit. `controller_checker.py` is extended to also send a `FEATURE_REQUEST`, an OF message that by default originates from a controller, . There are two moments that a this message can be send: (1) directly after the `HELLO` or (2) after the reception of a `ECHO_REQUEST`. The switch expectingly reacts with a `FEATURE_REPLY` following the standard OpenFlow handshake specified in [45]. This message can directly be used to conclude that we connected to a switch.

In our implementation we chose to wait until we 'heard' from the other side before sending our next message after our `HELLO`. In this way, we could detect what kind of OpenFlow connection is expected: In case we receive a `FEATURE_REQUEST`, we are probably talking to a controller, in case of a `ECHO_REQUEST` it is probably a OpenFlow switch.

Our script is capable of identifying an OF1.3 switch and the Ryu, ONOS, OpenDaylight and HPE VAN controllers. As an example, Fig. 3.2 shows the result of the script on the `<IP,TCP>` combinations we have found in Fig. 3.1. Here, before displaying the result, we print the message type list of the received OF messages.

Our findings are that OpenDaylight is the most recognizable controller, because it is the only controller that sends a `ROLE_REQUEST` (value 24) and `BARRIER_REQUEST` (value 20) at the beginning of the handshake. In fact, the other controllers never send a `ROLE_REQUEST`. The `BARRIER_REQUEST` is sometimes seen in HPE VAN. Also worth mentioning is that Ryu directly sends a `SET_CONFIG`, even before receiving additional information from the connecting switch, while in the other controllers this message is only seen after receiving all needed additional information.

Attack limitations

Given this experiment, two things need to be kept in mind when expanding the attack. First, although the script is a success, tests are only done with clean builds (or simple implementations) of the four controllers. It could easily be the case that in newer builds, or implementations which also run a couple of network applications, the total handshake procedure alters because the controllers need more or different information from the switch. This directly influences the suitability of the script. Second, for four controllers, it is easy to maintain the list of used handshake procedures, but this list needs to be expanded to other controllers (and maybe their versions) as well. Also, it could be that two different controllers use the same message sequence, making it impossible to identify them without additional information.

In earlier versions the script was also capable of identifying the OF version which was used on the other side. This feature was implemented after the discovery of [51], where a similar script was presented to identify OF communication and did already support the identification of the OF version. This was accomplished by holding a response `HELLO` back for a moment to check the OF version the attacked `<IP,TCP>` initiates, after which we set this same OF version in our responses. As stated before, some controllers initiate a `HELLO` as soon as the socket is open, while for example ONOS only communicates after receiving a `HELLO`. In the newer version of the script we always initiate the communication by sending a `HELLO` with version 1.3. Because all four controllers support this version, this does not lead to issues.

3.3. Abusing controller-switch communication

Having found and identified OF connections, it is time to look whether we can use simple OF messages (Table 3.1) to impersonate valid network components to influence the behavior of switches and/or the controller. To this extent, we firstly created scripts to 'attack' a chosen `<IP,TCP>` with messages like `HELLO`, `FEATURE_REQUEST`, `FEATURE_REPLY`, `ERROR` and `FLOW_MOD`. Eventually, we used the knowledge about message responses and handshakes to write a script to impersonate a switch towards the different controllers.

Implementation description

All scripts created for the attacks described above were created with use of the *pyof* library [50] (Appendix C.1). The script for the `FLOW_MOD` attack is shown in Listing 3.1. This attack can be performed in two ways: Sending a hard-coded 'drop all' packet, or creating a custom `FLOW_MOD` with use of the *pyof* library (which in Listing 3.1 is also a 'drop-all' flow). For the custom message, it can be seen that almost all sent values are set to 0, except the *Match* command. Also important is that the `FLOW_MOD` doesn't contain an *Action*, which defines what will be done after the match is found in the flow table. A `FLOW_MOD` without an *Action* is accepted and will result in an empty action in the flow table, which corresponds with the 'drop' action. (For the complete specifications of a `FLOW_MOD`, see [45].) The priority of this `FLOW_MOD` can easily be set higher by changing the priority parameter to a higher value.

```
#!/usr/bin/env python3
import argparse, socket
from pyof.v0x04.symmetric.hello import Hello
from pyof.foundation.basic_types import Pad, UBInt8, UBInt16, UBInt32, UBInt64
from pyof.foundation.constants import UBINT32_MAX_VALUE
from pyof.v0x04.controller2switch.flow_mod import FlowModCommand, FlowModFlags, FlowMod
from pyof.v0x04.common.flow_match import Match, MatchType

# OMITTED - Code to process command-line arguments

if attacked_ip != None and attacked_port != None:
    # create a hardcode 'drop all' FlowMod bytestring
    if hardcoded:
        data = b'\x04\x0e\x00@\xe4\xa7PN\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
                 \x00\x00\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00
                 \x00\x00\x00\x01\x00\x04\x00\x00\x00\x00\x04\x00\x08\x00\x00\x00\x00'

    # create an FlowMod message with the use of the pyof library
    else:
        # Empty Match, with MatchType '1' and four empty oxm_fiels.
        match = Match(  match_type = MatchType.OFPMT_OXM,      length = UBInt16(4),
                        oxm_field1 = UBInt8(0),                 oxm_field2 = UBInt8(0),
                        oxm_field3 = UBInt8(0),                 oxm_field4 = UBInt8(0)
                    )
        # All unimportant fields are set to zero. XID is arbitrary.
        OFP_NO_BUFFER = UBINT32_MAX_VALUE
        flowmod = FlowMod(  xid = 44,
                            cookie = UBInt64(0),                cookie_mask = UBInt64(0),
                            table_id = UBInt8(0),               command = FlowModCommand.OFPFC_ADD,
                            idle_timeout = UBInt16(0),          hard_timeout = UBInt16(0),
                            priority = UBInt16(0),              buffer_id = OFP_NO_BUFFER,
                            out_port = UBInt32(0),              out_group = UBInt32(0),
                            flags = FlowModFlags.OFPFF_SEND_FLOW_REM,
                            match = match
                          )
        data = flowmod.pack()               # create bytestring

    # Start the attack
    print("FlowMod attack start towards", attacked_ip, ":", attacked_port)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((attacked_ip, attacked_port)) # Open Socket
    s.send(Hello().pack())                  # Identify ourselves with an HELLO message
    print(s.recv(1024))                     # Receive a HELLO back
    s.send(data)                            # Send the FLOW_MOD
    print(s.recv(1024))                     # Await reaction.
```

Listing 3.1: Core of the Python script for a FLOW_MOD attack. Some parts are omitted or adapted in this listing for readability. *The complete code is delivered with this master thesis.*

A primary goal of the small attacks was to see what the response of the different OF entities is and whether these reactions always correspond with the expectations. Our gained knowledge was then used to implement a OF switch impersonator, which can successfully contact the controllers such that we reach the 'main mode' or are found in the network topology. The code of this script can be found in Appendix B. As can be seen there, we also start sending more complicated messages such as `MULTIPART_REPLY`'s. For this, we mostly send hard-coded messages, because the *pyof* library doesn't support the construction of `MULTIPART_REPLY`'s (yet). Also, our used version of the library contains some minor errors. For example, we had to correct the implementation for the *multipart_reply*, *multipart_request* and *get_config_reply*. The hard-coded messages are constructed with use of observed OF messages in Wireshark.

3.3.1. Attacking the switch

Whether a switch accepts an OF message from other switches depends on the control channel architecture (explained in Section 1.4.3). For the outcomes we discuss below, our setup was created with standard Mininet [41]. This means, we have an *out-of-band* control channel, such that switches are not supposed to talk OF to each other. All contacted switches thus expect that if we are talking to them that we are a controller.

Shortly, important observations from the experiments are (some already mentioned before):

- When receiving a `HELLO`, the switch responds with a `HELLO` and will eventually send an `ECHO_REQUEST` to check the connection. After this initial `HELLO` the switch accepts every kind of message.
- When receiving a `FEATURE_REQUEST`, the switch will send a `FEATURE_REPLY` (as expected). Effectively, responses on different requests (such as different `MULTIPART_REQUEST`'s) can be used to receive packets which can successfully be reused when impersonating a switch.
- When receiving an `ERROR`, the switch will send an `ERROR` back.
- The `FLOW_MOD` attack successfully adds a 'drop all' flow into the switch's flow table.

Given our observations, it seems that a switch directly trusts an incoming OF connection to be a trustworthy controller. It shows that this way, attacks like flow rule modification are easily possible. During the first experiments, the impersonated controller is always the *master* controller. We also investigated the possibilities of sending a `FLOW_MOD` when we are a *slave* controller - which could be the case when we have multiple controllers in the network and have taken over one. For this we adapted the code from [52]. It turns out that when we send a `FLOW_MOD` from a controller which at the switch is recognized as a *slave*, the switch will respond with an `ERROR` with reason `BAD_REQUEST`. However, the switch does accept multiple controllers to have the *master* role at the same time, which will make the attack possible again.

3.3.2. Attacking the controller

At first glance, it turns out there are less possibilities for 'attacking' a controller. Namely, from Section 3.1 it can be concluded that switches generally don't initiate communication towards the controller. Retrieving important information by faking particular requests from the controller is thus not directly a possibility. Another idea is to impersonate a colleague controller. However, communication is then shifted to the east-/westbound interfaces [3], where OpenFlow is not employed. As an example, as soon as the controller receives an OF message (here a `HELLO`), it directly assumes a switch wants to connect and it responds with a `FEATURES_REQUEST`. When we try to send a `FEATURES_REQUEST` ourselves (thus impersonate a controller), communication dies out. Further investigation of the east-/westbound interface lies outside the scope of this research.

One possibility that shows perspective is trying to influence the behaviour of the controller by sending `ERROR` messages. From our investigation it turns out that the controller does react to an `ERROR`, even though our constructed message with use of *pyof* is seen as a malformed package in Wireshark. As proof-of-concept, we adapted the Ryu [19] controller such that a error response method installs a 'drop all' flow in the flow table of all connected switches. Indeed, we see that this functionality is triggered when an `ERROR` is received. We need to note however, that the response to an `ERROR` is completely determined by the implementation of the controller - and that the probability that actual controllers will have such drastic response functionality is zero. An attacker could investigate what is seen as the most severe type of error message to possibly trigger the most decisive response.

Our largest attack is the impersonation script `switch_impersonator.py`, shown in Appendix B. With this, we were directly successful in impersonating an Open vSwitch towards the Ryu, ONOS and OpenDaylight

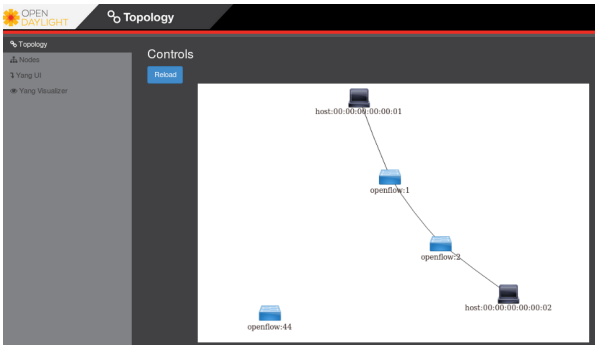

Figure 3.3: Result of impersonating an instance of Open vSwitch towards the OpenDaylight controller, such that the impersonated switch (`id:44`) is found in the network topology. The other part of the network in the topology is set up via Mininet.

controller. For the impersonation towards the HPE VAN controller, we needed to construct more complicated OF messages which wasn't supported by the *pyof* library. At first, we weren't successful, because HPE VAN asks us for a `TABLE_FEATURES` response which we could not easily fake. Reusing such a response from a Mininet Open vSwitch was in this case not straightforward, because it contains the features of all its 254 tables distributed over multiple packets. It is not possible to set the number of tables of Open vSwitch to a lower value. In a later setup we found out that Open vSwitch can also respond with an `ERROR` message with type `BAD_REQUEST` and code `BAD_TYPE`. Reusing this response made the impersonation attack towards HPE VAN also successful.

The success of `switch_impersonator.py` can be checked in two ways. For the Ryu controller, debug output shows that after entering *config mode* after the exchange of the `HELLO` messages, the controller eventually switches to *main mode*. With the other controllers, we see that the impersonated switch is included in the network topology view which is accessible via their GUI application. The result of such inclusion in the network topology can be seen in Fig. 3.3, where we attacked the OpenDaylight controller.

Attack limitations

As seen, the first consequence of a successful impersonation attack is that the controller thinks that this switch is part of the network and thus has a incorrect view of the network topology, which is already a serious issue. However, there are limitations on what an attacker can do from this point. While an attacker can send `ERROR` messages towards the controller to attack as described above, it is in its initial state impossible to communicate to other network devices in the network. This is because the controller doesn't know about connections between the impersonated switch and these other network devices. For the identification of connections between switches, the controller doesn't rely on OpenFlow, but uses the Link Layer Discovery Protocol (LLDP). To build a larger fake network connected to the controller, an attacker thus also needs to have knowledge of this protocol in order to (mis)use it. In case the attacker impersonates multiple switches, this protocol is only needed when the attacker wants to connect these switches - it is also possible to flood the controller with multiple, unconnected impersonated switches.

3.4. Possibilities in Physical Networks

To get some insights into the attack possibilities in physical networks, we experimented with our own small physical setup for which we used the Zodiac FX OpenFlow Switch [43]. With use of a single Zodiac switch, one can create a small SDN which can connect up to three hosts and is connected to a single controller. More information on working with the Zodiac switch can be found in Appendix C.3.

Fig. 3.4 shows the two setups we have worked with. The setup on the left matches with the setup used in the simulated environment, where the attacker is directly connected to the controller. This setup is used to redo our experiments with the impersonation script, expecting the same results. The setup on the right is a new situation. The Zodiac switch allows for the ports to be configured to match a particular VLAN, such that we can state whether the ports are part of the OpenFlow network or just *native*[1] ports. The connection towards the controller is always part of a native VLAN, while hosts part of the network are connected via the OpenFlow VLAN (here $H1$ and $H2$). We stated that the connected to the attacker is a native port, such that we

[1] We use the term *native* because this is used in the Zodiac FX switch documentation.

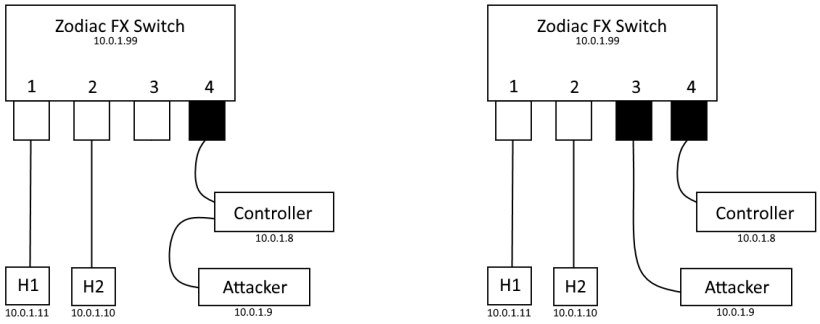

Figure 3.4: The two experimental setups we used while using the Zodiac FX Switch, including the IP-addresses of all entities. The *white* ports are ports over which the switch is configured to expect traffic part of an OpenFlow network, while the *black* ports are configured to be so-called native ports. On the *left*, the attacker is directly connected to the controller. This setup matches the setup used in the simulated environment. On the *right*, the connection to the controller goes via the switch over two native ports.

could see whether we could (1) impersonate another controller towards the switch or (2) attack the controller via the switch.

As expected, when the attacker is directly connected to the controller, it is possible to perform switch impersonation. It turns out it is also possible to perform this attack via the switch, where the only role for the switch is to forward the traffic between the attacker and controller. We do see that the controller thinks that the Zodiac switch and impersonated switch are not connected, while this is the case. The controller doesn't make use of the fact that the packets travel through the Zodiac switch, but needs additional information from the switches or via LLDP to come to this conclusion.

We were not able to impersonate a controller and perform flow rule modification. This is because we weren't able to setup a connection due to the lack of open TCP-ports on the Zodiac switch - and this capability is one of the start conditions to perform our attacks. From this, it is even more clear that the feasibility of our attacks is mostly defined by the ability to connect to the entities we want to attack and that the standard open TCP-ports of controllers is a vulnerability we can abuse.

3.5. Malicious Network Components

Besides attacks which we can perform via scripts, impersonating certain network components, security issues will arise when network components are taken over by attackers. First, a malicious controller can take over the entire network, given it has the master role. Because this follows naturally, we will not further look into this. Instead, we elaborate on our investigation towards the possibilities for malicious switches, which is mostly performed with use of [53][2]. We did not perform experiments.

There are several ways to create (or obtain) malicious network components. Most straightforward, switches can be taken over directly by accessing it physically or by exploiting some vulnerability in the switch. As soon as an attacker has access to a network component, it can leverage OpenFlow to influence the network's behaviour, just like we have done in Section 3.3. To what extent this is possible, depends on the used control channel architecture, which we will discuss separately.

3.5.1. Attack possibilities on the in-band control channel

First we look into the attack possibilities when the network has an in-band control channel. An example scenario is given in Fig. 3.5 on the left. Here, not only the data packets between the hosts $H1$ and $H3$ will pass through the malicious switch, but the malicious switch will also see all the control data which is intended to or originates from Switch 3, $S3$. Thus, an attacker will be able to attack both the data plane and the control channel. [53] states that the severeness of the threat of a compromised switch depends on whether the attacker can only modify the flow table of the compromised switch or (s)he can perform a control channel hijack, where the malicious switch believes that the attacker is the correct controller it should listen to.

In the case of flow table modification, the attacker only accesses the flow table of the malicious switch. First, an attacker could focus its attack on the data plane. As an example, it gives him(/her) opportunities

[2] [53] contains some errors regarding the explanation of the in-band and out-of-band control channel. The correct explanation can be found in Section 1.4.3

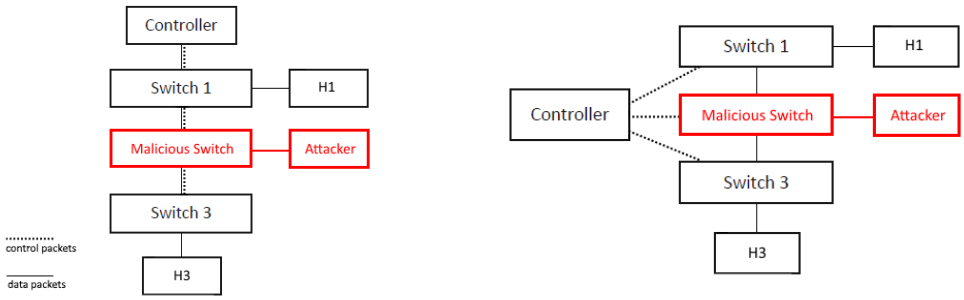

Figure 3.5: Example scenarios of a network with a malicious switch, where the network has a in-band (*left*) or out-of-band (*right*) control channel. The connection between the compromised switch and the attacker can be physical or virtual.

to setup the flow table to perform eavesdropping - where the data is duplicated towards the attacker. In this way, the attacker should be able to retrieve all data which is sent between $H1$ and $H3$. More severely, the attacker could setup a man-in-the-middle attack, where all data which travels through the malicious switch is sent only to the attacker, which then can modify the packets before sending it back to the malicious switch to continue the normal route. Given these two attacks, the attacker could also focus on the control data which travels through the malicious switch. Looking at $S3$, the attacker could not only eavesdrop control data, but also edit the control data to for example edit the state of $S3$ known by the controller.

When the attacker is capable of impersonating a controller and hijack the control channel, a larger portion of the network can be influenced by sending control data towards the malicious switch. In our example, the attacker could be able to reach $S3$ by sending control data destined for $S3$ via the malicious switch, which was already used to forward these kind of packets. In this case, the possibilities for the attacker are larger then in the case of the man-in-the-middle scenario, because now the attacker can initiate control data, while otherwise it can only adapt control data which travels by.

A last possible attack is topology spoofing. The attacker could send fake control data towards the malicious switch stating it is a new part of the network. The fake control data could state that there is only one other switch connected to the malicious switch, but also that there is a sequence of switches because of the in-band architecture. The malicious switch doesn't know better than to forward these messages to the controller, which as a result will have a wrong view of the network topology.

Note that, in general, the influence of the attacker depends on the logical location of the malicious switch. As an example, in Fig. 3.5 the malicious switch is not used to receive control data for $S1$, thus the attacker could never reach this switch. Shortly, in this topology, the closer a malicious switch is to the controller, the larger the influence of an attacker which pretends to be the controller.

3.5.2. Attack possibilities on the out-of-band control channel

There are far less possibilities for an attacker as soon as the network has an out-of-band control channel, such as on the right in Fig. 3.5. Attacks that are possible are attacks targeted at the data plane. Namely, using flow table modification, it is still possible to configure the flow table of the malicious switch to perform eavesdropping or a man-in-the-middle attack, which influences the traffic of data packets between $H1$ and $H3$.

Control data attacks, on the other hand, are not possible any more. In the out-of-band control channel, control data travels over dedicated lines, such that no control data will travel via (or through) the switches. Therefore, the attacker will not have access to the control data for $S3$, which was the case in the in-band scenario. In the case of a hijacked control channel, the attacker can only target the malicious switch and not others, sharply decreasing the attacker's options. Neither can the attacker perform topology spoofing, because this is also achieved by sending control data through a switch towards the controller. Because in this scenario, the malicious switch will not accept control data which for or from other switches, the faked control data will never reach the controller.

3.6. Conclusions

In this chapter we have investigated the possibilities for the abuse of the control-data plane by misusing the OpenFlow protocol, particular on the moment that the controller and a network device want to establish a connection (the OpenFlow handshake). In an experimental way, we have shown that it is possible to impersonate one (or more) switches to influence the controller's view of the network topology and that a simple script is enough to adapt a switch's flow table to drop all traffic. For both these attacks, we discussed the attack limitations and sketched possibilities for attack expansions. We also elaborated on the possibilities for an attacker when (s)he is in possession of a compromised network device. With usage of OpenFlow, flow tables can be adapted to enable eavesdropping or even the modification of data packets (the latter with use of a man-in-the-middle attack).

Our work is relatable to [54], which gives a security analysis of the OpenFlow protocol and shows possibilities for denial-of-service and information disclosure. However, [54] doesn't focus on the attack possibilities when misusing the OpenFlow handshake and other particular OpenFlow messages, which we have done.

Our experiments show similarities with the work of Gregory Pickett [51], which we encountered halfway our experiments. Pickett presents a toolbox that can be used to attack SDN in several ways. First, the tools *of-check* and *of-enum* are able to find and identify OpenFlow services and controllers. Second, *of-switch* is able to impersonate an OpenFlow switch and *of-flood* is able to flood an OpenFlow controller in order to disrupt the network or even bring it down. Picket's work is mostly used as inspiration. We expand on his attacks by also using the OpenFlow handshake to identify different type of controllers and perform flow rule modification towards switches. Furthermore, while the impersonation attack of Pickett makes use of OpenFlow 1.0, we use OpenFlow 1.3, and we conduct our experiments with different controllers, namely Ryu, ONOS, HPE VAN and OpenDaylight - while Pickett used OpenDaylight and Floodlight.

In our threat model, we assume that an attacker is able to create a direct link towards the entity (s)he wants to attack. Countermeasures against the attacks discussed in this chapter should be focused on preventing the attacker to create such a link and protecting the control channel. Firstly, clear from Section 3.5, the network seems to be automatically protected against attacks to the control channel as soon as an out-of-band control channel architecture is used. In this case, malicious components can only attack the data plane, because no control data travels through switches. Secondly, authentication methods and encryption should be applied, such as SSL/TLS. In this way, an attacker can not simply impersonate a network device any more. Also with encryption, while eavesdropping may still be possible, the data an attacker retrieves is still protected and could not be misused. The current state of the deployment of authentication and encryption methods in SDN will be investigated in the next chapter.

4

Securing the Control-Data Plane

As stated in Section 3.6, one (obvious) security measure to protect communication is to use authentication and encryption. A well-known example of such a measure is the usage of Secure Sockets Layer and/or its successor Transport Layer Security, referred to as SSL/TLS. But, as a known security challenge in an SDN using OpenFlow, using SSL/TLS is set to an *optional* feature since OF1.3, which counteracts the general deployment of such measures in new networks. In this chapter we will investigate the current state deployment of authentication and encryption, using SSL/TLS as a central example. To this extent, we will focus on the following questions:

- How does SSL/TLS work and what problems does it solve?
- Are there new challenges that arise when providing authentication and encryption and how does this relate to traditional networking?
- What kind of alternatives to SSL/TLS are there?
- What is the current state deployment of the different methods in current popular controllers?

Naturally, we start with an introduction to SSL/TLS in Section 4.1. In Section 4.2, we elaborate on simple experiments we performed to gain insight in SSL/TLS deployed in a simple SDN with the Ryu controller. The core of this investigation is found in Section 4.3, where we will elaborate on the solved problems and new challenges in SDN where SSL/TLS is deployed. While SSL/TLS is our main example, for the interested reader different alternatives are also shortly discussed here, such as proposals found in [55–57]. Last, Section 4.4 comes with a retrospect on the current-state deployment of the discussed authentication and encryption methods as well as an overall discussion.

4.1. Introduction to SSL/TLS

Secure Sockets Layer and its successor Transport Layer Security are two protocols used for a secured communication channel. Because they are both much alike, they are mostly referred together as SSL/TLS[1]. Shortly, the secured channel is created in two steps. We start with the SSL/TLS handshake which provides the authentication step. During this handshake, encryption keys are exchanged. These keys are then used to encrypt en decrypt the exchanged messages, such that the overall channel is secured.

SSL/TLS is known as a *public key system* [58], where two keys are used: one for enciphering and one for deciphering a text. More concretely, SSL/TLS is an *asymmetrical cipher system*. It relies on both a secret key and a public key. Namely, the cipher key is made public while the deciphering key is kept secret. In other words, everyone is permitted to encipher a text, but only a few people can retrieve the original text. The precise en-/decryption algorithms themselves lie outside the scope of this research - an interested reader can start with [58] (Chapter 6).

The SSL/TLS handshake is used for both the authentication step as well as the distribution of the encryption keys. For the authentication, we have two possibilities. In *one-way* authentication, only the 'server'

[1] Officially, TLSv1 is the successor of SSLv3. It is not the same, but it is backward compatible.

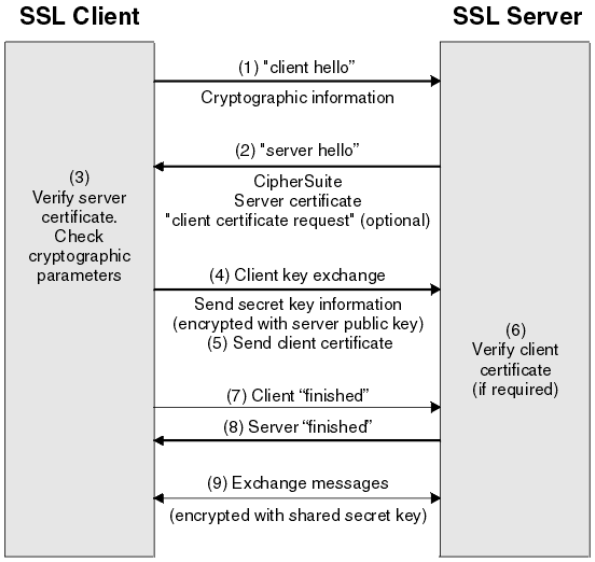

Figure 4.1: Overview of the SSL/TLS handshake as presented in [59]. In the SDN context, the controller can be seen as the server and the connected switches as clients.

needs to identify itself. In such scenarios, clients most of the time identify themselves by entering a password (when trying to log into the server). In *two-way* authentication, also the 'client' needs to identify itself. In the context of SDN, the controller serves as the server and the different connected switches are clients. Most controllers which support TLS, such as Ryu and OpenDaylight, work with two-way authentication.

The overview of the SSL/TLS handshake is given in Fig. 4.1, which is elaborately explained in [59][2]. Shortly, key messages are the certificates and the client key exchange. Every system that wants to correctly identify itself has a certificate, which is signed by a so-called Certification Authority (CA) which is known to be trusted. This certificate also holds the public key of the system. In Fig. 4.1, the client verifies the server certificate and then uses the public key retrieved from the certificate to start the key exchange, which contains encrypted secret information (a future shared secret key). The server on the other hand, is the only one that can decrypt this secret information by using its own secret key, after which it can retrieve (or calculate, if needed) the shared secret key. After the handshake is finished, further exchange of messages is encrypted with the shared key. In the context of SDN, these future messages are OpenFlow messages.

4.2. Deploying SSL/TLS in a simple SDN

To gain insight into SDN deployed with SSL/TLS, we investigated a simple network which used the Ryu controller using SSL/TLS. For this, we relied on tutorials [60, 61]. For a system with Open vSwitch [42], Mininet [41] and Ryu [19], it is first explained how to initialize the so-called Public Key Infrastructure (PKI) which is needed to run SSL/TLS to create, sign and manage the digital signatures. After this, we can generate the secret key and certificate for the controller(s) and switch(es) using this PKI. As a final step, both the switches and the controller also need a reference to the CA of the PKI, which is used to verify other certificates.

Experiment description

Setting up both the controller and switches with the private key, certificate and reference to the CA, we can successfully set up a simple Mininet network with two hosts connected to one switch, which is connected to a Ryu controller. For clarity, we show the necessary command to run Ryu with the correct parameters in Fig. 4.2. Eventually, one can check whether the SSL connection was setup successfully by using the `ovs-vsctl show` command in Mininet. Our result is shown in Fig. 4.3. The `ovs-vsctl` library is also used to connect Open vSwitch to the generated key and certificate, as explained in [60, 61].

Using this simple SDN, we investigated the network traffic using Wireshark on two locations. First, retrieving messages on the switch-controller link (the control channel), one can investigate the SSL/TLS handshake by decoding the messages not as OpenFlow but as SSL traffic. Investigating this traffic, we could extract that

[2]A clear video explanation is available at `https://www.youtube.com/watch?v=n_d1rCXNrx0` (Accessed: 20-02-2017)

```
ryu-manager <ryu_controller.py>
        --ctl-privkey <private_key.pem>
        --ctl-cert    <certificate.pem>
        --ca-certs    <ca-file.pem>
        (--verbose)
```

Figure 4.2: Running the Ryu controller with SSL, one needs to give the private key, certificate and CA reference as parameters.

```
mininet> sh ovs-vsctl show
fd344adc-503b-4bad-814d-c7db8ae924dd
    Bridge "s1"
        Controller "ssl:192.168.253.130:6633"
            is_connected: true
        fail_mode: secure
        Port "s1"
            Interface "s1"
                type: internal
        Port "s1-eth1"
            Interface "s1-eth1"
        Port "s1-eth2"
            Interface "s1-eth2"
    ovs_version: "2.0.2"
mininet>
```

Figure 4.3: The `ovs-vsctl show` command inside Mininet shows that the switch is successfully connected to the controller over SSL.

the used public key system is RSA used with AES (the Advanced Encyption Standard) - both explained in [58]. Also we see that after the handshake, Wireshark isn't able to convert the encrypted data into OpenFlow messages. One can use this as a check that the channel between the controller and switch is secured/protected.

Secondly, we looked into retrieved messages between one of the hosts and the switch. This data can be generated by initiating a ping between the two hosts. Looking at this data, we see that these packets aren't encrypted. This result isn't unexpected, because we haven't deployed SSL/TLS on the data plane, but solely on the control channel.

Discussion

Simply deploying SSL/TLS on the control channel leaves the dataplane itself unprotected, as shown in Fig. 4.4. This gives opportunities towards attackers. On the other hand, as seen in Section 3.3, for an SDN the most severe attacks are focused on the control channel (and OpenFlow), which is protected in this way. The problem of an unprotected dataplane could be solved by also setting up SSL/TLS between the hosts and switches.

Although the control channel itself is secured, there are still possibilities for attacks. As an example, using the generated Open vSwitch private key and certificate it was possible to extend our script to setup an SSL connection with the controller and then impersonate a switch (`switch_impersonator.py`, line 75-82 in Appendix B). It turns out that the (simple) Ryu controller does not check whether a certificate is already used by another switch, allowing us to impersonate a switch over SSL. In our case, we could directly re-use the private key and certificate because we had generated it. To retrieve this information in other ways, one could use the retrievable TLS handshake. Namely, given that in our deployment TLS works with RSA and AES, it is not impossible to retrieve the private key - it would be if the so-called Diffie-Hellman protocol (explained in [58]) was used. Another example is that it is possible to prick through the SSL encryption using an enhanced Wireshark. Following the 'wiki'-page of Wireshark [62], it is possible to decrypt SSL traffic as soon as Wireshark is running with `GnuTLS` and `Gcrypt`[3].

As will also be elaborated on in Section 4.4, the attacks named above are more attacks on SSL/TLS instead of attacks on SDN in general. One should see the deployment of SSL/TLS as a security layer around the network itself. As soon as we can break this layer, the vulnerabilities discussed in Chapter 3 (such as impersonation possibilities) will arise. Given the scope of this research, we will not (further) investigate attacks on the SSL/TLS protocol itself.

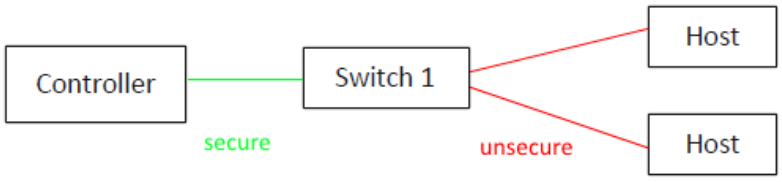

Figure 4.4: Result of deploying SSL/TLS on the control channel in an out-of-band architecture. Not unexpected, the dataplane itself stays unsecured/unprotected.

[3]The OS of our virtual machines didn't support these packages, such that we couldn't verify this statement.

4.3. Securing the Control-Data Plane

The main goal of using SSL/TLS is to protect the network against attacks discussed in Chapter 3, such as eavesdropping, topology spoofing and impersonation attacks. It turns out that the capabilities of the attacker are largely determined by two factors: (1) how SSL/TLS is used to protect control-data plane communication and (2) whether the network has an in-band or out-of-band architecture [53]. Below we will elaborate on these two factors, look into the problems SSL/TLS solves, but also discuss the challenges which come with it. After this, we will shortly present some alternatives to secure the control channel.

4.3.1. Consequences of SSL/TLS

SSL/TLS provides protection with the use of authentication and encryption. A network can effectively be protected against eavesdropping solely because of the encryption - it doesn't matter whether the communicating systems are malicious or not, one can not decrypt the information. For this, it doesn't matter whether SSL/TLS is used with one- or two-way authentication. However, this does matter for topology spoofing and impersonation attacks. When SSL/TLS is deployed with two-way authentication, the network will be protected against both these attacks. But, with one-way authentication, an attacker can still create bogus switches, making topology spoofing completely possible and impersonation to a certain level. Namely, while it is not possible to impersonate a controller (and influencing a switch's flow table, for example), impersonated switches can alter a controller's view of the network or be used to flood the controller with traffic.

While the out-of-band architecture does protect the control channel by itself as discussed in Section 3.5, the control channel architecture does also determine to what extent data is protected when SSL/TLS is deployed (only) between the controller and switch. Given the in-band architecture, data also travels over channels which are used for control messages. Theoretically, the data which travels over these channels is protected, reducing the locations in the network where for example eavesdropping is possible. In the out-of-band architecture, all data is unprotected, as was seen in Section 4.2 and Fig. 4.4.

There are also new challenges that arise when deploying SSL/TLS. First, while not in the scope of this research, vulnerabilities in and attacks at the protocol itself should not be forgotten. Second, the added traffic due to the TLS handshake will influence the performance of the network. Third and most important, it does increase the complexity of the network. This has mostly to do with key management. Although this is not an unknown problem (it also arises in traditional networks), it can scale up to a large problem when we have a lot of switches, or even multiple controllers. Since OpenFlow in principle uses two-way authentication, network administrators must create device-specific certificates, sign all of them and finally distribute the certificates to the devices [53]. This tedious management may tempt these administrators to skip some parts of the process or let them decide using one-way authentication or no TLS at all.

When it is decided not to use SSL/TLS at all, sometimes the network type can provide some protection. In secured networks (such as data centers) the lack of TLS is feasible. Namely, access to the network components is difficult, making it harder to take them over physically or remotely. In campus-style or office networks where devices are directly accessible, not using SSL/TLS can become a serious security vulnerability [55].

4.3.2. Alternatives to SSL/TLS

While SSL/TLS is the recommended option for the control channel security stated by the OpenFlow specification, a couple alternatives are proposed in [55–57]. We will shortly discuss these studies in context of related work, to create a complete view on control channel security for the (interested) reader. We note that all proposed protocols lack a critical analysis, but providing this lies outside the scope of this research.

As first alternatives, [55] proposes Secure Shell (*SSH*) and *IPSec*, both known Internet security systems which can be adapted to suit SDN. Both protocols create a communication tunnel (where data is encrypted) and differ from TLS in terms of the setup procedure. The most important reason for these alternatives was to address configuration problems for network operators which arise with TLS. For these protocols, the administrator doesn't have to build all the infrastructure to support it, which is the case for the PKI with SSL/TLS. Although these two protocols are only discussed briefly and deployment of the protocols is stated as future work in [55], in their analysis it is already shown that both IPSec and SSH also have their vulnerabilities.

[56] contains the detailed proposal for *Identity-Based Cryptography* (IBC). The focus of this alternative is the simplification of the setup process and a solution for multidomain SDN and a wireless data plane. We did not investigate how IBC implements a solution for the latter, but solely focused on the setup process, because there it directly competes with SSL/TLS. Similar to TLS, IBC requires a trusted authority, here to act as a

Private Key Generator (PKG). But, in TLS the CA (certified authority) is used to generate the public and private key pairs, whereas the PKG of IBC generates only the private keys and the public keys will be derived from the identity of the user (here, the switch). With SDN, controllers can act as PKGs for the switches that are located within their domains. This reduces the needed storage and management of the public keys. The key exchange process knows a speed up and network scalability is improved because with IBC two communicating parties do not need to obtain each other's public key from the CA to derive the session key.

Last, [57] proposes the *Off-the-Record* (OTR) protocol. Shortly, this protocol is used to protect the network 'instant-session' communications and does so by using *forward encryption* and *hidden identity* (or *deniable*) *authentication*. Forward encryption is that messages are only encrypted with temporary per-message keys, in this case AES keys negotiated using the Diffie-Hellman key exchange (see [58]). Important is that compromise of any long-lived keys does not compromise any previous conversations. Concretely in SDN, one sees that every short communication between switch and controller is encrypted with a different key. Deniable authentication is that messages don't have digital signatures such that after a conversation is complete, anyone is able to forge a message to appear to have come from one of the participants in the conversation, assuring that it is impossible to prove that a specific message came from a specific component. But, within a conversation itself, the recipient can be sure that a message is coming from the component they have identified; the exchange of identity documents is implemented inside the channel, which effectively prevents it from being intercepted by a third party. Given these principles, challenges regarding the security and management of long-term key storage, certificate exchange and the requirement of a PKI, are solved.

4.4. Conclusions

In this chapter, we introduced SSL/TLS as a solution for the security of the control-data plane and investigated which problems this solves and what new challenges arise. Using simple experiments and analysis, we can conclude that SSL/TLS, given that it uses two-way authentication as advised in most controllers, protects the control channel against all attacks discussed in Chapter 3. However, we see that attacks on the data plane are still possible. Thus, for a completely secure network, administrators need also to invest in data plane security.

After our experiments, we stated that there are still possibilities for attackers to penetrate the SDN, given that the SSL/TLS protocol itself is first successfully attacked. Again, while SSL/TLS provides protection, one should not forget that the protocol is merely a security layer over the control channel. SSL/TLS does not solve (or fix) the vulnerabilities with OpenFlow itself. Naturally, vulnerabilities towards a network will arise as soon as SSL/TLS is not configured securely. Fortunately, companies specialized in IT security - such as Fox IT - invest in research into how to securely configure the protocol[4].

The biggest challenge which hinders the general deployment of SSL/TLS in current SDNs is the key management which can grow tedious, especially when two-way authentication is used (which we actually want) and we also want to protect the data plane. The alternatives discussed in Section 4.3.2 all focus on simplifying the setup process and thereby the key management process. However, these alternatives are not (yet) found in existing controllers, thus the expectation is that SSL/TLS will remain the default solution by OpenFlow.

In conversation with *HP Enterprise* regarding the HPE VAN controller [32], we gained some insight in the current state deployment of security in SDN. For the HPE VAN case, the company sees that in larger networks the management of the switches is done with use of the out-of-band architecture, having the control channel completely seperated from the operational network. For the authentication, HPE VAN uses Keystone, while for the encryption SSL is used. In practice, not a lot of networks use the PKI but rely on self-signed certificates. The usage of TLS is supported, but gives extra overhead, while the combination of the out-of-band architecture, SSL and Keystone is regarded as sufficient secure, also by HP itself. Still, we need to note that the usage of self-signed certificates instead of a PKI is a possible security risk. In our discussion with HP, no details were shared on the protection of the operational network.

Because controllers like Ryu, OpenDaylight and ONOS are open source controllers, it was not straightforward to get current state deployment insights for these controllers. However, using our experiments we can conclude that here mostly the direct use of TLS (and thus the required PKI) is advertised and supported. To invest in the general deployment of control channel protection, one could investigate whether deploying only (simple) SSL could be sufficient to secure the channel (such as is done with HPE VAN). This could be a separate research, were one also needs to dig deeper into the particular vulnerabilities of the SSL protocol itself. For us, this lies outside the scope of this research.

[4] Confirmation that indeed Fox IT does this, is given by an employee.

5

The Villainous Host

The last step of our layered approach to investigate SDN security focuses on the host. From Section 2.2 followed the concrete question: 'In what ways can a regular host damage an SDN?' To answer this question, we investigated whether different kind of attacks can be performed towards an SDN without directly leveraging the OpenFlow protocol, such as:

- The so-called *fingerprinting* attack, where timing analysis is used to identify OpenFlow networks or even particular SDN controllers.
- An attack leveraging the Address Resolution Protocol (ARP) and the possibility to use this protocol to perform table flooding and denial-of-service (DoS) attacks.
- Other 'traditional' attacks, such as the *packet-in-packet* attack or misuse of IGMP (Internet Group Management Protocol) messages.

The implementation and results of the different attacks are discussed in Section 5.1, Section 5.2 and Section 5.4, following the order above. We will also shortly discuss several defence mechanisms in Section 5.3. This chapter ends with a retrospect in Section 5.5. But first, we elaborate on the used threat model.

Threat model

In this chapter, we take the position of a villainous host. This host is connected to an OpenFlow supporting switch part of an SDN. The host does not posses a connection to the controller and is not able to leverage the OpenFlow protocol. For clarification, the setups used in our experiments (both in the simulated and physical environment) are displayed in Fig. 5.1 and Fig. 5.2.

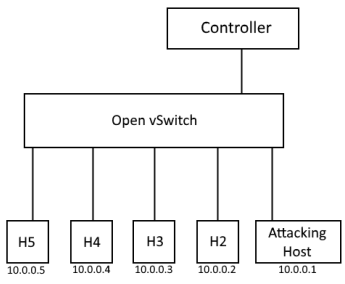

Figure 5.1: Setup for our attacks in the simulated environment, including the IP addresses of the hosts. Open vSwitch and the hosts are created by and accessible via Mininet.

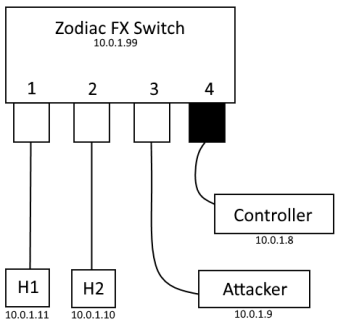

Figure 5.2: Setup for our attacks in the physical network using the Zodiac FX switch, including the IP-addresses of all entities. The *white* ports are configured to expect traffic part of an OpenFlow network.

5.1. SDN Fingerprinting

In fingerprinting attacks, timing analysis is done to identify an SDN in general or particular SDN controllers. The feasibility study of [37] states that when an SDN functions reactively and installs flow rules in response to received packets from switches, this results in additional delay. In other words, one can encounter two different response times: (1) the response time when a flow is already present in de flow table, T_1, and (2) the response time when there is the additional flow setup time, T_2. Seeing such different response times in a network could hint towards the existence of an SDN.

A challenge for attackers is to collect enough data for 'common' values for T_1 and T_2 and to surely distinguish the two response times given random noises. As a response to [37], [63] presents a real-world evaluation. They relied on the packet-pair dispersion[1], which turns out to be a stable feature over time and little affected by the size of the network. They found that indeed the delay introduced by flow rule installation provides an effective distinguisher for an attacker to identify whether packets are only processed on the data plane or triggers an interaction with the controller, which is said to be relatively slow. The controller of their testbed was Floodlight (v0.9).

The study of [64] focuses the timing analysis on the `idle_timeout` and `hard_timeout` values given to installed flows, which are then leveraged to calculate control plane processing-times to identify different SDN controllers. Their approach can successfully identify the controllers OpenDaylight, Floodlight, POX, Ryu and Beacon. This is done by only using ICMP (ping) traffic, which would mean that such an attack can be performed by every 'simple' host.

In the next sections, we elaborate on our investigation towards the feasibility of timing analysis given the standard implementation of our used controllers: Ryu, ONOS, OpenDaylight and HPE VAN. To do this, we performed a simplified version of the timing analysis studies explained above. In short, we (1) retrieved the response time of a first ping between two hosts connected via one switch, given that no flows exist to correctly forward this traffic and (2) checked what the default values of `idle_timeout` and `hard_timeout` are of the flows which are added because of this initiated ping traffic. We will discuss our results for the experiments run with in the simulated and physical network we worked with.

5.1.1. Timing analysis using Open vSwitch

The results of our experiments in the simulated network are shown in Table 5.1. The table shows two things: (1) Response times which include flow installation time and (2) default timeout values.

Focusing on the ping response times, we see that already in this simplified case we can distinguish the different controllers. It is also clear that these values are significantly larger than the response times of the following pings, which in this case lie around 0.05 ms for all controllers. ONOS gives us by far the largest response time. The response time of the HPE VAN controller is the smallest, but in this case there is *no* flow installed. Namely, all traffic reaches a flow with `action=output:NORMAL`, which means that the packets are processed using the normal (traditional switch) pipeline. Here, the response time only contains additional delay due to ARP (Address Resolution Protocol) traffic.

We need to note that all registered response times in Table 5.1 are the response times of a first communication (ping) between two hosts and thus also contains the exchange of ARP messages, naturally influencing the reponse time. But, given that the response time for HPE VAN is without flow rule installation but with ARP exchange, we can conclude that the flow rule installation creates a larger (distinguishable) delay.

[1] The term *dispersion* is commonly defined as the time interval between the first bit of the first packet and the first bit of the second packet of the pair.

Controller	**Response time first ping** (ms)	`idle_timeout` (s)	`hard_timeout` (s)
Ryu	5.998	0	0
ONOS	12.287	0	0
OpenDaylight	0.372	1800	3600
HPE VAN	0.238	n/a	n/a

Table 5.1: Timing analysis results for the four used controllers in their standard implementation where the switch is Open vSwitch (averaged over 10 performed attacks). For the timeout values, 0 means that the values are not set in the flow rule and the flow entry will never be removed (we did this to be consequent with [64]). For the HPE VAN case, no flows are installed, thus the timeout values are not applicable. This also explains the relative quick first ping response time.

We also tried to figure out whether we could identify the controllers by their default timeout values, where HPE isn't applicable because no flows are installed. We see that only OpenDaylight set the timeout values, which means that we can not identify all controllers. [64] presents the default timeout values for the five controllers they worked with. Comparing their values with our observations, we see that in their case there are also controllers with the same default timeout values - for example Floodlight and Beacon with `idle_timeout` = 5 and `hard_timeout` = 0. Besides this, they found the same values for the Ryu controller, but different values for OpenDaylight (namely 0). The latter can be explained by stating that [64] worked with the 'Hydrogen' version of OpenDaylight, while we worked with 'Helium', a newer version.

Lastly, we want to mention that while ONOS does not define the timeout values, we see that installed flows are deleted rather quickly. This is accomplished by an active statement from the controller to delete the flow entries (a `DELETE_STRICT` message).

5.1.2. Timing analysis in the Zodiac FX network

The results of our experiments in the physical network with the Zodiac FX switch are shown in Table 5.2. Because the timeout values only depend on the used controller, here we only focus on the flow installation times. We also excluded HPE VAN, because it doesn't install flows.

As expected, the overall performance of the controllers is the same as in the simulated network. But during the experiments we also learned more about the controllers. Namely, because the Zodiac switch is a slower device than the virtual switch (Open vSwitch), there are also differences in the response time when there is no flow installed (i.e. when there is a flow match). This difference can be explained by the fact that all three controller install different kind of flows (different `match_fields` [45]), influencing the time needed to perform flow matching. Concretely, OpenDaylight installs flows where the switch needs to match on source and destination MAC address, with Ryu we match on source port and destination MAC address and ONOS wants matches on all three. Using Table 5.2, we conclude that matching on three properties is more costly than on two, and that matching on source port is probably more costly than on (source) MAC address.

As in Table 5.1, the registered response time in Table 5.2 contains the exchange of ARP messages. In this experimental setup, we wanted to find out whether it was also possible to record response times including flow installation but excluding the ARP exchange. During this investigation, we found out that the exchange of ARP messages plays a vital role in the installation of the flows. Namely, while excluding ARP traffic by saving exchanged information at the hosts, we saw that OpenDaylight doesn't install flows and communication between two hosts fails. It turns out that the installation of flows is triggered by ARP traffic and not ICMP traffic (which actually follows naturally when following the traditional way of networking).

5.1.3. Timing analysis challenges

Given our experiments and the findings of primarily [63, 64], we can state that timing analysis can be used to identify SDN in general and to a certain extent specific SDN controllers. However, there are some challenges for this approach. The biggest challenge is that one needs to collect enough (and correct) data to create a complete view of the behavior which identifies particular controllers. For this, [63, 64] made a good start, but question is how long this data stays 'valid' or correct. As an example, while the experiments in [64] are performed in 2016, we find different values for the OpenDaylight controller a year later. This suggests that processing-times depend on different versions of controllers or on the used hardware. Besides this, networks could rely on customized controllers - as an example, network administrators can easily edit the default timeout values a controller assigns to flows. This all could mean that the collected data is not usable.

One can also argue that focusing on processing-times and timeout values isn't enough to identify controllers. As seen in the experiments, the default timeout values are similar for different controllers. While this limits the search space, we still need to apply more techniques to decide which one it is. This is cited in

Controller	**Response time when installing flow** (ms)	**Other response time** (ms)
Ryu	9.04	1.05
ONOS	12.9	1.35
OpenDaylight	1.60	0.873

Table 5.2: Timing analysis results for used controllers in their standard implementation using the Zodiac FX switch (averaged over 10 performed attacks). We did not include HPE VAN because this controller doesn't install flows. We also include the response time when no flow is installed, because they clearly differ per used controller. The averaged values for ONOS could be influenced, because while working with this controller and the Zodiac FX switch we encountered difficulties (see Appendix C).

[64], but there is not elaborated on which techniques. In the future, it could turn out that particular network functions (applications) have a large influence on the performance of different controllers, which could suggest that focusing on this could identify them. The techniques discussed here solely focusing on data-control plane.

When we let go of the desire to identify SDN controllers and focus solely on identifying SDN in general, the challenges named above are less applicable. In all cases, the processing-time is clearly larger than in a traditional network. This will be enough information for an attacker to change his(/her) attack plan and assume (s)he is dealing with an SDN. Besides this, an attacker can still perform 'traditional' attacks to bring down an SDN, as will be clear from the next sections.

5.2. Misusing ARP for Table Flooding and Denial-of-Service

Two known attack possibilities to networks in general are table flooding and accomplishing denial-of-service (DoS). DoS is a classic type of attack where one tries to accomplish the complete lack of reaction of a service by flooding the service with requests. In the context of SDN, table flooding is focused on filling the flow table with different flow rules up to the moment the table is full. Goal is to influence and eventual completely bring down performance. Naturally, table flooding can be used to accomplish a DoS attack.

Denial-of-service was already mentioned as a known security challenge in SDN in Chapter 2. Now, we focus on effectively accomplishing this attack. In [65], this is done via two ways: (1) attacking the switch's flow table and (2) attacking the control plane bandwidth. When the switch's flow table is full and consequently cannot install new rules, it sends an `ERROR` message to the controller with the error code `TABLE_FULL` and drops the packets. Shortly, as soon (and as long) as the switch's flow table is full, it cannot forward packets for which it doesn't find a match.

To attack the control plane, one needs to send traffic in such a way that a lot of `PACKET_IN`s are generated towards the controller. In this case, packet loss can occur in two ways [65]. First, given the switch's limited output queue size, packets are lost as soon as this queue is full. Second, congestion may introduce latency in the control channel which can result in the switch receiving responses from the controller too late, at a moment that the buffered packet is already discarded. The impact of the attack is mostly local to the switch. However, it could also be the case that the switch succeeds in overwhelming the controller in general.

In [65], the attacks are performed on a network connected to the NOX controller. [66] presents the DDoS vulnerability in Floodlight. Next, we investigate the feasibility of both table flooding and DoS for our investigated controllers. In Section 2.1 it is stated that, corresponding with threat vector one, forged traffic provides an open door for (D)DoS attacks. Therefore, we approached our attack using forged traffic, in our cased spoofed ARP (Address Resolution Protocol) messages. As explained in the next subsection, the properties and used values in ARP messages were precisely what was needed for our attacks.

5.2.1. Table flooding

As explained before, the table flooding attack corresponds with the attempt to add as much as possible flow rules to the switch's flow table with the goal to completely fill the table. A flow rule is different from another as soon as it is different on a single `match_field`. Following the OpenFlow specification [4], there are 39 different `match_fields`. This suggests a lot of possibilities for an attacker to create different flows. However, it turns out that the standard implementations of our investigated controllers only use a couple of these fields. As already shortly mentioned in Section 5.1.2, Ryu only uses the source port (`in_port`) and destination MAC address (`eth_dst`) for matching. For added flows, OpenDaylight uses both destination and source MAC address (`eth_src`).

To create new flows, it is needed to alternate these fields, which is possible with use of (spoofed) ARP messages. Shortly, the Address Resolution Protocol is used by hosts to translate IP addresses to MAC addresses. An ARP request can be seen the question: "Does anyone know who IP$_{dst}$ is? Please tell IP_{src} located on MAC$_{src}$." after which the ARP response states "IP$_{dst}$ is at location MAC$_{dst}$!" At this moment, the source host can couple an IP address to a MAC address. At the same time, the destination host has also identified a (new) MAC address. In a spoofed ARP message, one is able to send ARP requests with any (random) value for the source IP and/or MAC address, which results in the receiver believing that this these addresses actually exist. In a successful attack, this results in the switch adding the appropriate flow rule(s) to support traffic from and towards a fake MAC address.

Attack implementation

Our attack is performed with the script displayed in Listing 5.1. In just a couple of steps we are able to create both a random IP and MAC address and use these in a custom ARP message, constructed with *scapy* [67]. More information on *scapy* can be found in Appendix C.2. In the last line of the code, there can be seen that we introduce a timing gap between two send ARP messages, to influence the rate we attack the network.

```
from scapy.all import *
import random, time, argparse

# OMITTED - Code to process command-line arguments

""" Returns a random IP adress in range: (1-254).(0-254).(0-254).(0-254) """
def randomIP():
  return (str(random.randint(1,254)) + "." +
          ".".join(str(random.randint(0,254)) for _ in range(3)))

""" Returns a random MAC adress """
def randomMAC():
  return ":".join(gen_hex(2) for _ in range(6))

""" Helper function, which creates a hexidecimal number of a given 'length' """
def gen_hex(length):
  return ''.join(random.choice('0123456789ABCDEF') for _ in range(length))

# Attack!
if arguments.ip != None and arguments.size != None:
  for x in range(0,arguments.size):
    rand_ip = randomIP()
    rand_mac = randomMAC()
    if (rand_ip != arguments.ip):
      sendp(Ether(src=rand_mac)/ARP(op="who-has",hwsrc=rand_mac, psrc=rand_ip, pdst=arguments.ip),
            iface=arguments.iface)

      # A gap between ARP messages (and thus PACKET_IN messages)
      # makes sure the controller doesn't experience denial-of-service
      time.sleep(arguments.gap)
```

Listing 5.1: Core of the Python script for the performed attack using spoofed ARP messages. Some parts are omitted for readability. *The complete code is delivered with this master thesis.*

Achieving table flooding

For our first experiments, we used our simulated setup as depicted in Fig. 5.1 and focused on the behaviour of the controllers to predict the feasibility of our table flooding attack. For this, we looked at what happens inside the switch, because that is dictated by the controller. We see that OpenDaylight performs completely according to the expectation and installs two new flow rules per spoofed ARP message to support traffic in two ways between the attacked host MAC address and the spoofed MAC address. ONOS behaves similarly, installing two flows per spoofed ARP message, but table flooding attack is impossible for this controller because it rather quickly sends a `DELETE_STRICT` message to the switch to remove the installed flows, such that the flow table never really gets filled. Due to the implementation of Ryu, it fluctuated when new flows were installed, because as information at the controller grows, not every new ARP message seems to trigger a `FLOW_MOD`. Lastly, in the case of HPE VAN there are no rules installed at all because all traffic reaches the flow with `action:output:NORMAL`.

Table flooding is thus most feasible in a network with the OpenDaylight controller. A next important factor that influences the success of the attack is the used switch. In the simulated version, using Open vSwitch, we were able to flood but not completely fill the flow table and influence the network performance. Even with more than 5000 flows installed, the performance of ping (ICMP) traffic isn't significantly slower. Following the specification of Open VSwitch, it turns out that it can support over a million flow rules, which isn't easy to flood.

[65] names two switches with a limited capacity which thus could be more easily flood-able. The HP 5406z1 switch can store (only) 1500 flow rules, and the CpQD OpenFlow 1.3 Software Switch can store up to a maximum of 4096 flow rules. From our experiments with Zodiac FX switch (setup shown in Fig. 5.2), it turns out that this switch can only store 300 flows. In this setup, we have seen in Wireshark that indeed the switch

Number of installed flows	Ping response time (ms)
5 (initial state)	1.153
67 (after 30 spoofed ARP messages)	1.267
300 (after 150 spoofed ARP messages)	1.876

Table 5.3: Ping response time values for a network with a Zodiac FX switch connected to OpenDaylight. The values are averages for 100 pings, given that needed flows for communication are already installed.

sends an `ERROR` message and drops traffic as soon as the table is full. We also see that, when the table is partially full, this influences network performance. We show this with the response time results in Table 5.3. As expected: The more flows the switch needs to match on, the larger the response time.

5.2.2. Denial-of-service

While from our observations it follows that not in all cases we can achieve table flooding, we see that the ARP traffic *always* causes traffic towards the controller. As an example, the ONOS, OpenDaylight and HPE VAN controllers use ARP traffic to create a network view which is displayed in their GUI. We could thus easily use the ARP traffic to try to flood the controller with `PACKET_IN`s. Therefore, it was possible to also use the code of Listing 5.1 to try achieving denial-of-service.

Because in case of the Zodiac FX switch, the switch is relatively quickly flooded (we only need 150 spoofed ARP messages), we only investigated this attack in the simulated environment. Given our setup (Fig. 5.1), the attacker always sends the spoofed ARP messages towards $H2$. We then looked into the performance of the network from the point of view of the other hosts. Eventually, we investigated (1) performance of the GUI, (2) performance when a flow doesn't exist and (3) performance when a flow does exist.

The network topology GUI applications from both ONOS, OpenDaylight and HPE VAN struggle with displaying a network with over a 1000 hosts. As an example, Fig. 5.3 shows the GUI of HPE VAN after an attack. Given the three controllers, the GUI of OpenDaylight performs the worst and eventually crashes first. While crashing the GUI isn't an important result, we can directly conclude that using spoofed ARP messages is a successful way to perform topology spoofing. Further research could investigate whether this influences the decision making of the controller (or network administrators which configure the controller).

In our script, we can define the waited time between each send ARP message. By default this gap is 0.5 ms, in which case all the controllers can handle the traffic and the other hosts do not experience additional delays in their communication. By minimizing this gap, we see that network performance gets influenced. Namely, `PACKET_IN` messages are processed at the controller in a *first in first out* matter. Thus, when there is not yet a flow between two hosts, this flow is only installed after all other packets are processed at the controller. Using this fact, we see that at a moment that an large attack is performed, we can achieve denial-of-service between other hosts. This result is given in Fig. 5.4. Here, we see that we experience dropped packets and a huge delay for the packet which does arrive.

We also see that network performance is influenced when flows do exist. Namely, the switch still needs to try to match all arriving ARP messages in the flow table, delaying the moment that a flow match for real traffic is performed. Using OpenDaylight, we see that ping (ICMP) traffic experiences a delay around 0.2 ms, while without the attack it was 0.05 ms. This reduced network performance is temporary.

We could not achieve denial-of-service when the HPE VAN controller is used. This is not unexpected, because in its standard implementation, ARP traffic is solely forwarded towards the controller for the GUI application, while the switch itself is stated to process the traffic like a 'traditional' switch. But also here, we see an additional delay while the attack is performed. We shortly investigated a Mininet setup with a switch *not* connected to a controller to see whether we then also have this additional delay (as is expected). Indeed, we see that for a period of time, ping traffic between h_3 and h_4 does experience an additional delay (0.150 ms instead of 0.05 ms) while the link between h_1 and h_2 does experience complete denial-of-service.

5.3. Defence Mechanisms against Data-Control Plane Saturation

Achieving DoS by flooding the control plane with `PACKET_IN` messages, such as is achieved using the spoofed ARP messages, is known as a *data-to-control plane saturation* attack. While for example [68] names some modules in OpenDaylight and ONOS which solve both DoS and authentication problems, also specialized solutions for such data-to-control plane saturation attacks are designed. In the context of related work, below

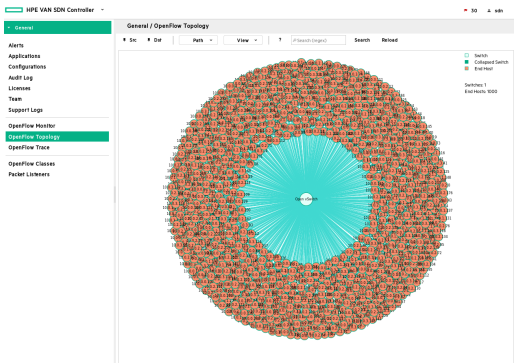

Figure 5.3: HPE VAN controller GUI after an attack with over 1000 spoofed ARP messages, where it mistakenly thinks these hosts are real and part of the network.

```
mininet> h4 ping h3 -c5
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=11026 ms

--- 10.0.0.3 ping statistics ---
5 packets transmitted, 1 received, 80% packet loss, time 4017ms
rtt min/avg/max/mdev = 11026.882/11026.882/11026.882/0.000 ms, pipe 5
```

Figure 5.4: Proof of dropped packets and increased delay when two hosts try to communicate while the attacker performs the attack with spoofed ARP messages.

we will shortly discuss such solutions.

One of the most well-known solutions which focuses on anomalous control plane floods is AVANT-GUARD [69]. The system is presented as a security extension to the OpenFlow data plane, where added intelligence makes that there is less communication needed between the control plane and data plane. Most important of the system is the so-called connection migration module, which can identify attack traffic by verifying the TCP handshake of each new flow and will only allow traffic from sources that will complete the TCP handshake. The writers of AVANT-GUARD admit that the system is limited to fight against DoS attacks based on IP spoofing and not attacks based on UDP and ICMP.

FloodGuard [70], presents a defence system with a proactive flow rule analyzer and a data plane cache and is said to be a solution independent of the protocol of the attack traffic. It's main motivation is the philosophy that pre-installing flow rules into the data plane and discarding all other table-miss packets would resolve the complete security problem, but that it is unrealistic due to the dynamics of network policies. Eventually, the proactive flow rule analyzer makes sure that, during attacks, the major functionality of the network infrastructure is enforced by deriving proactive (and thus not reactive) flows with use of dynamic application tracking. The data plane cache is added to migrate the table-miss packets to, before sending them to the controller in a limited rate in form of `PACKET_IN` messages. This means that also attack traffic will reach the controller, but in this way no benign packets are lost.

Weakness of both AVANT-GUARD and (parts of) FloodGuard is that they are implemented into the OpenFlow switches, which means that success/strength of the systems depends on whether all switches have the system implemented. As a response, solutions which are implemented on the control plane are FlowRanger [71] and FlowDefender [72]. FlowRanger uses a trust-based mechanism to evaluate the likelihood that `PACKET_IN` requests are from attacking sources and prioritize them into multiple buffer queues with different priorities. Naturally, requests from trusted sources are arranged in high priority queues and are served faster than requests in low priority queues. FlowDefender is somewhat more complicated, adding four modules that implement an attack detector, packet filter, flow rule manager and a table-miss engineering module. The modules are implemented as applications on the Ryu controller.

While out of the scope of this research, as future works one could investigate the success of our attack with spoofed ARP messages when different defence mechanisms are deployed in the network. Expectation is that the attack is still possible when AVANT-GUARD is deployed, because the system is stated to be limited to fight against DoS attacks based on IP spoofing. For the other solutions, in the evaluation of the other solutions there are no concrete experiments towards defending against ARP spoofing in particular. Also, besides the specialized defence mechanisms, [68] names some modules in OpenDaylight (Defense4All and AAA) and ONOS (Security-Mode ONOS) which solve both DoS and authentication problems. Experiments will show whether these modules also defend against spoofed ARP messages.

5.4. Other attacks

While performing timing analysis as discussed in Section 5.1 is a new attack approach which focuses on particular SDN properties, using spoofed ARP messages is a traditional attack which is also shown successful

in traditional networks. There are more attacks which were successful in traditional networks of which it is interesting to see whether they also work in SDNs. Below, we will discuss two of such attacks, both suggested by employees at Fox IT.

First, instead of using ARP messages to flood the network/controller, there is suggested to leverage the Internet Group Management Protocol (IGMP(v3)) or Multicast Listener Discovery (MLD(v2)). Shortly, IGMP is used to establish multicast group memberships in traditional networks and is an integral part of IP multicast. Normally, it is used for one-to-many networking applications such as online streaming video and online gaming. MLD can be seen as the equivalent of IGMP, where IGMP is used on IPv4 networks and MLD uses IPv6. For an attack, one can leverage IGMP 'membership requests' and flood a switch with these messages. However, the use of multicast is normally not supported in SDN. As an example, [73] presents code for custom-made IGMP support in the POX controller. Because of this, we decided that further research towards IGMP lies outside the scope of this research. We didn't look into MLD because IPv6 excluded from all experiments in this research.

The second suggested attack is the 'packet-in-packet' approach. Shortly, this is a piggy-backing technique that allows attackers to hide malicious packets inside packets that are permitted on the network [74], in which way for example firewalls can be evaded. In the context of SDN, an example of a successful packet-in-packet attempt is to send an OpenFlow (e.g. `FLOW_MOD`) message encapsulated in a `PACKET_IN` message, which would result in the encapsulated message (thus, the `FLOW_MOD`) to be executed/processed. Or, one could investigate whether one can break the controller by changing the packet hierarchy. In our setting, *scapy* [67] allowed us both to experiment with the packet hierarchy and the construction of encapsulated OpenFlow messages. When sending such messages, we focused on how `PACKET_IN`s and its data are treated. In the definition of the SDN controllers it is stated that they work with deserialization, where the controller can extract layer per layer what kind of message the encapsulated data is and whether there is important header information. This property is used in the processing of ARP message, for example.

In the networks with our four controllers in their standard implementation, we have not found any way to leverage the packet-in-packet principle. When sending direct or encapsulated OpenFlow messages, we see that in the cases that these messages are encapsulated in `PACKET_IN` message as its data, this data is not further used. This is because the controllers are not implemented to expect OpenFlow messages, thus don't have modules to leverage this data. When receiving packets with incorrect hierarchy (e.g. Ethernet inside IP instead of IP inside Ethernet), the packets are encapsulated in a `PACKET_IN`, but - as in the case of the OpenFlow messages - during the processing of the data, the controller implementation fails to use the data properly (as expected) and simply discards the message.

5.5. Conclusions

In this chapter, we looked into the possibilities for a regular host to damage an SDN without directly leveraging the OpenFlow protocol as is done in Chapter 3. Concretely, we experimented with timing analysis, spoofed ARP messages and a 'packet-in-packet' approach to see whether these attacks are suitable. We have found that simple timing analysis focused on the processing time of controllers gives enough information to an attacker to decide whether a target network is an SDN or not. However, the identification of specific controllers gives a number of challenges, such that there is need of additional techniques to be successful in this. Given the investigated traditional attacks, we have found that ARP spoofing can lead to denial-of-service in case of the Ryu, ONOS and OpenDaylight controller. The gravity of the table spoofing attack mostly depends on the used SDN supporting switch. We have seen that it was quite easy to flood the Zodiac FX switch, making the installation of new flows impossible.

The performed timing analysis directly follows the ideas presented in [37, 64] and had the first purpose to serve as a feasibility study for our four investigated controllers. It also provide us of more insight into the workings and functionality of the controllers without investigating the code of the controllers. To our knowledge, this is the first study which leverages ARP to perform table flooding and denial-of-service. Another way to perform data-to-control plane saturation is with the use of SYN packets, as stated in [75]. We relied on ARP because these packets were easy to construct with the use of *scapy* and the misuse of ARP has the additional consequence that we perform topology spoofing.

It is important to invest in security solutions for the found vulnerabilities in this chapter. Some possible solutions are already mentioned in Section 5.3, and further research in this direction will be mentioned as future works in the next chapter, where we will review all our findings of this research.

6

Improving SDN security

In this research, we have investigated the current state security of a software-defined network's core, focusing on the security of the control-data plane communications: the control channel. In the previous three chapters we presented all the details of our performed attacks and findings in order to answer the research questions we presented in Section 2.2. Now our investigation is complete, we use this concluding chapter to present compact answers to these research questions and derive best practices to secure the control channel the most. We also elaborate one more time on the importance of the protection of the control channel. We end this research with a look into the future of SDN and its security, also providing future works we think can (and needs to) be done as a next step in the field of SDN security.

6.1. Protecting the Control Channel

Since its introduction, there has been controversy over the security of software-defined networking - a fact that is still addressed in recent articles about SDN potential [76]. On one hand, SDN introduces opportunities for more sophisticated security solutions, such as implementations for automated malware quarantine [77], while on the other hand, new challenges arise for the security of the network itself and its controller. As an example, [78] names the programmability of SDN controllers *a double-edged sword*: "Engineers can install security applications on the controller's northbound interface to open up new ways to apply security policies, but those applications can reprogram the network through the controller - hackers can trick engineers into installing compromised applications."

As done in the introduction to SDN security in Chapter 2, the security challenges for SDN can be categorized over its different layers. Challenges which arise with the implementation and deployment of network applications (threat vector *five*) can be seen and investigated separately from the challenges for the security of the network itself (threat vector *three* and *four*). In other words, one should build an SDN networking environment where the security of the network is not dependent on SDN applications being free from vulnerabilities [79]. A lot of networking applications (which could be security solutions) presume that there are no network design vulnerabilities in the network they are deployed in. Following this motivation, it is thus beneficial to focus on the security of the network's core, providing such independent secure network environment.

In the context of SDN, a secure network environment starts with a secure, protected controller. Leaving network applications out of the equation, protecting the controller is done by focusing on the communications towards the controller - securing the control channel. Because of this, in this research we have focused on the security of the control channel. Summarizing, we have investigated the current state security of a control channel where the OpenFlow protocol is used, to see what potential vulnerabilities this results in for the entire network. We have approached this from an attacker's point of view, and tried to perform two different kind of attacks: (1) attacks where an attacker directly (mis)uses the OpenFlow protocol and (2) attacks where an attackers is not able to use this protocol. We also investigated to what extent the deployment of SSL/TLS provides protection of the control channel.

6.1.1. Answering the research questions

The research questions we are able to answer are, as stated in Section 2.2:

1. Is it possible to misuse the OpenFlow protocol to attack forwarding devices, controllers or the control channel in general and what possibilities lie there for a malicious user who has taken over a network component?

2. Does the usage of SSL/TLS solve problems found in the first question; and is it always feasible as a solution or does it introduce new problems?

3. In what ways can a regular host damage an OpenFlow Network?

In Chapter 3, we investigated the first question. We have found that with the appropriate knowledge of the OpenFlow protocol, an attacker is able to create scripts to identify OpenFlow channels and impersonate a OpenFlow supporting switch. With our scripts in their current form, we are able to perform topology spoofing and flow rule modification. More advanced attackers could extend the impersonation script to perform more severe attacks. Another source that introduces more attacks is the presence of malicious network devices. In short, the implication for security is that an OpenFlow network is vulnerable to different man-in-the-middle attacks, topology spoofing, eavesdropping, flow rule modification or even controller hijacking, which are all possible to perform via the control channel.

The protection of the control channel done via the employment of SSL/TLS was investigated in Chapter 4. Using our experiments, we see that two-way authentication protects the control channel against all attacks named above. However, we see that the control channel and data channel function separately, such that SSL/TLS on the control channel leaves the data channel unprotected. Thus, for a completely secure network, administrators need also to invest in data plane security. This possibly contributes to an even more complex key management process, which is stated to be the main point why SSL/TLS is not standard deployed in current SDNs.

Last, Chapter 5 investigated attack possibilities without leveraging the OpenFlow protocol. Here, it turns out that the Address Resolution Protocol (ARP) can be misused to generate a lot of traffic on the control channel to attack the controller. The scripts provided in this research can be used to perform topology spoofing (without impersonation), table flooding and denial-of-service, an example of a data-to-control plane saturation attack. In this case, using SSL/TLS doesn't solve our problems. A solution to the problems here should focus on reducing the needed control traffic.

6.1.2. Control channel best security practices

Given the answers to our research questions, we can give a three recommendations in order to secure an OpenFlow network and its control channel the most. Concretely, when setting up an SDN, one should:

1. Set up an isolated control channel. Most advocated is the use of an out-of-band control channel, naturally separating the control and data channel;

2. Use two-way SSL/TLS between all switches and controllers, securing the control channel;

3. Try to reduce the amount of traffic on the control channel.

The first two recommendations follow directly from our investigation and don't need extra research for future implementations. The isolation of the control channel can be realized during the first stages of creating the network and should be part of the network architecture. SSL/TLS is supported by all investigated controllers, but network administrators should invest in the correct setup and operable key management – two challenges which comes with SSL/TLS. It is important to implement two-way authentication, because with one-way authentication an attacker can still create bogus switches in order to perform topology spoofing and flood the controller with traffic.

To find correct solutions to reduce the amount of traffic on the control channel, extra research is needed. Some solutions which try to achieve this are already presented, such as AVANT-GUARD [69] and FloodGuard [70]. These solutions are discussed shortly in Section 5.3, but deep investigation towards such solutions was outside the scope of this research. Besides solutions against control-data plane saturation attacks, network architects should think about possible control channel vulnerabilities which can occur when deploying certain network applications on top of the controller. With the installation of every application which generates packets from the data plane to the control plane, one should check whether this can be misused and if so, whether this possible misuse can be mitigated (for example by filtering). If this is not the case, such an application introduces a direct vulnerability to the system and should not be implemented in the network[1].

[1] This recommendation is done by Ronald van der Pol, network innovation advisor at SURFnet.

6.2. The Future of SDN Security

While we are able to present three recommendations which directly contribute to the security of an OpenFlow network, it will probably even more important for network architects to investigate the future of the SDN architecture, which will influence the security of SDN. The software-defined networking paradigm is a relative new concept which is still evolving, which is seen in different ways. First, we have the development and availability of many different controllers - and as seen, the security of the network is influenced a lot by the security of the controller. Second, while the main concept of SDN will stay the same, the used SDN architecture changes in order to support more. An example is an suggested extended SDN architecture to support Network Function Virtualization (NFV) [80]. Third, we see that the developments in the SDN security field plays a more central role in the development of SDN in general. A concrete outcome is the rise of communities like [81], where security solutions are presented such that it is possible to contribute to these solutions; of which one is Security-mode ONOS.

The main limitation of this research is that it is unknown how relevant it stays due to the fast changes in the SDN world. As an example, the used implementations of the controllers could be replaced with improved ones which protect the network from our identified vulnerabilities. However, we did focus on ONOS and OpenDaylight which are expected to be controllers we will eventually end up with[2]. Also, in our research we assume the data plane to be as simple as possible, while in current developments we see that focus lies on improving and expanding the data plane. This was already used in early security solutions such as AVANT-GUARD [69], but more significant is the rise of the programming language P4 [82] which can be used to program the data plane. Also, using NFV in SDN solution contributes to the functionality and intelligence of the data plane. In the future, it could be so that on the more sophisticated SDN switches, table flooding isn't possible any longer - or that a solution to this problem is easy programmable.

Eventually, the scope of this research was rather broad, investigating the current state of research towards SDN security in general, identifying possible research areas and gaining an insight into the security of the control channel from two different angles. Researchers should take the developments of the SDN architecture into account in order to find research directions which are still relevant at that moment (and in the future).

6.2.1. Recommendations for future work

Taking this complete research and the prospects of the future of the SDN paradigm into account, we finish with some recommendations for future work. These examples will or (1) continue on (and expand) the presented attacks or (2) investigate the security of particular (future) SDN implementations:

1. Expand the impersonation attack to see whether it is possible to create fully functional (controllable) bogus switches which make more severe attacks, such as controller hijacking, possible;
2. Investigate whether it is possible to misuse the Link Layer Discovery Protocol (LLDP) in combination with OpenFlow, in order to virtually connect impersonated switches to other impersonated or genuine network devices;
3. Investigate whether it is also possible to flood a switch's *group-table*[3];
4. Investigate the security of the ONOS or OpenDaylight controller in full detail, also focusing on the code and security solutions for these controllers. While this is already done in other studies (such as [68]), this will stay a relevant topic as long as the controllers are in development.
5. Investigate whether the combination of SDN and NFV can mitigate current security challenges for SDN.

The future works presented above are demarcated research topics which focus on security problems in detail. Here, one will look at the network from an attacker's point of view, following the approach we took in this research. We disregarded future works having a defensive angle, focusing on concrete security solutions and defence mechanisms. However, this doesn't mean that such work isn't recommended. As an example, in our research we didn't find a direct solution to prevent control-data plane saturation with use of ARP. Future work could investigate whether the solutions we discussed shortly can protect the network against our attack, or create a new one. Eventually, complete secure networks can only be constructed with knowledge from both the attacker's and defender's sides.

[2] This is expected by Ying-Dar Lin, IEEE fellow and ONF Research Associate, also writer of [80].

[3] In this research, we never elaborated on group-tables. Shortly, group-tables (introduced in OpenFlow 1.3 [45]) are sort-like to flow-tables and make it possible to state multiple actions for a particular match.

Bibliography

[1] M. Casado, T. Garfinkel, A. Akella *et al.*, "SANE: A protection architecture for enterprise networks," *Proceedings of the 15th conference on USENIX Secuity Symposium - Volume 15*, 2006.

[2] M. Casado, M. Freedman, J. Pettit *et al.*, "Ethane, taking control of the enterprice," *Proceedings of the 2007 conference on Applications, technologies, achitectures and protocols for computer communications*, pp. 1–12, 2007.

[3] D. Kreutz, F. M. V. Ramon, P. Verissimo *et al.*, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, no. 1, pp. 14–76, 2015.

[4] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, no. 2, pp. 69–74, 2008.

[5] "Open Networking Foundation," https://www.opennetworking.org/, 2014, [Accessed: 30-09-2016].

[6] U. Hölze, S. Jain, A. Kumar *et al.*, "B4: experience with a globally-deployed software defined wan," *Proceedings of the ACM SIGCOMM 2013 conference*, pp. 3–14, 2013.

[7] S. Sezer, S. Scott-Hayward, P. Chouhan *et al.*, "Are we ready for SDN?" *Communications Magazine, IEEE*, no. 7, pp. 36–43, 2013.

[8] S. Taha Ali, V. Sivaraman, A. Radford, and S. Jha, "A Survey of Securing Networks Using Software Defined Networking," *IEEE Transactions on Reliability*, no. 3, pp. 1086–1097, 2015.

[9] M. Oswalt, "BLOG: Keeping it Classless - Perspectives on the Intersection of Networking and Software Development," https://keepingitclassless.net/, [Accessed: 15-12-2016].

[10] R. Mijumbi, J. Serrat, J.-L. Gorricho *et al.*, "Network Function Virtualization: State-of-the-Art and Research Challenges," *IEEE Communications Surveys and Tutorials*, no. 1, pp. 236–262, 2016.

[11] T. Fredrich, "What is REST?" http://www.restapitutorial.com/lessons/whatisrest.html, [Accessed: 15-12-2016].

[12] Open Networking Foundation, "OF-CONFIG 1.2 - OpenFlow Management and Configuration Protocol Specification," 2014.

[13] B. Pfaff and B. Davie, "The Open vSwitch Database Management Protocol," https://tools.ietf.org/html/rfc7047, 2013, [Accessed: 15-12-2016].

[14] M. Oswalt, "SDN Protocols part 3 - OVSDB," https://keepingitclassless.net/2014/08/sdn-protocols-3-ovsdb/, 2014, [Accessed: 15-12-2016].

[15] J. Schönwälder, M. Björklund, and P. Shafer, "Network Configuration Management using NETCONF and YANG," *IEEE communications Magazine*, pp. 166–173, 2010.

[16] M. Oswalt, "SDN Protocols part 4 - OpFlex," https://keepingitclassless.net/2014/09/sdn-protocols-4-opflex-declarative-networking/, 2014, [Accessed: 15-12-2016].

[17] A. Tootoonchian, S. Gorbunox, Y. Ganjali *et al.*, "On controller performance in software-defined networks," *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services Hot-ICE'12*, pp. 13–18, 2013.

[18] M. McCauley, "POX," http://www.noxrepo.org/, 2012.

[19] Nippon Telegraph and Telephone Corporation, "Ryu Network Operating System," http://osrg.github.com/ryu/, 2012.

[20] D. Erickson, "The Beacon OpenFlow controller," *Proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in software defined networking HotSDN'13*, pp. 13–18, 2013.

[21] "Floodlight is a Java-based OpenFlow controller," http://www.projectfloodlight.org, 2012.

[22] Y. Takamiya and N. Karanatsios, "Trema OpenFlow controller framework," https://github.com/trema/trema, 2012.

[23] T. Koponen, M. Casado, N. Gude *et al.*, "Onix: a distributed control platform for large-scale production networks," *Proceedings of the 9th USENIX conference on Operating systems design and implementation OSDI'10*, pp. 1–6, 2010.

[24] "ONOS: Open Network Operating System," http://onosproject.org/, 2013, [Accessed: 02-02-2017].

[25] "OpenDaylight: A Linux Foundation Collaborative Project," http://www.opendaylight.org, 2013, [Accessed: 02-02-2017].

[26] L. Hardesty, "ONOS joins the Linux Foundtaion, becoming an OpenDaylight Sibling," https://www.sdxcentral.com/articles/news/onos-joins-the-linux-foundation-becoming-an-opendaylight-sibling/2015/10/, 10 2015, [Accessed:30-10-2016].

[27] S. Shin, V. Yegneswaren, P. Porras *et al.*, "Rosemary: A robust, secure, and high-performance network operating system," *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[28] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending SDNs from malicious administrators," *Proceedings of the 3rd Workshop on HotSDN '14*, pp. 103–108, 2014.

[29] S. Shin, P. Porras, V. Yegneswaran *et al.*, "A security enforcement kernel for openFlow networks," *Proceedings of the 1st Workshop on HotSDN '12*, pp. 121–126, 2012.

[30] ——, "FRESCO: Modular composable security services for software-defined networks," *Internet Society NDSS*, 2013.

[31] "The Linux Foundation," https://www.linuxfoundation.org/, [Accessed: 30-09-2016].

[32] Hewlett Packard Enterprise, "HPE VAN SDN Controller 2.7.18 Release Notes," 2016.

[33] B. van Asten, N. van Adrichem, and F. Kuipers, "Scalability and Resilience of Software-Defined Networking: An Overview," 2014.

[34] N. van Adrichem, F. Kuipers, and B. van Asten, "Fast Recovery in Software-Defined Networks," *Proceedings of the European Workshop on Software Defined Networking*, 2014.

[35] N. van Adrichem, F. Iqbal, and F. Kuipers, "Backup rules in Software-Defined Networks," *Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2016.

[36] D. Kreutz, F. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," *Proceedings of the second ACM SIGCOMM workshop on HotSDN'13*, pp. 55–60, 2013.

[37] S. Shin and G. Gu, "Attacking Software-Defined Networks: A First Feasibility Study," *Proceedings of the 2nd ACM SIGCOMM Workshop on HotSDN '13*, 2013.

[38] M. Brooks and B. Yang, "A Man-in-the-Middle Attack against OpenDayLight SDN Controller," *Proceedings of the 4th Annual Conference on Research in Information Technology (RIIT)*, 2015.

[39] S. Kerner, "Is SDN Secure?" http://www.enterprisenetworkingplanet.com/netsecur/is-sdn-secure.html, [Accessed: 30-09-2016].

[40] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, pp. 1–7, 2013.

[41] "Mininet - A Instant Virtual Network on your Laptop," mininet.org, [Accessed: 15-12-2016].

[42] Linux Foundation, "Open vSwitch," openvswitch.org, [Accessed: 15-12-2016].

[43] Northbound Networks, "Zodiac FX OpenFlow Switch," https://northboundnetworks.com/products/zodiac-fx, [Accessed: 10-05-2017].

[44] "Wireshark," https://www.wireshark.org/, [Accessed: 15-12-2016].

[45] Open Networking Foundation, "OpenFlow Switch Specification Version 1.3.2 (V0x04)," 2013.

[46] S. Son, S. Shin, V. Yegneswaren *et al.*, "Model checking invariant security properties in OpenFlow," *Communications (ICC), 2013 IEEE International conference on*, pp. 1974–1979, 2013.

[47] P. Porras, M. Fong, V. Yegneswaren *et al.*, "Securing the Sofware-Defined Network Control Layer," *NDSS '15*, 2015.

[48] X. Wen, Y. Chen, C. Hu *et al.*, "Towards a secure controller platform for OpenFlow application," *Proceedings of the 2nd ACM SIGCOMM Workshop on HotSDN '13*, pp. 171–172, 2013.

[49] "NMAP," https://nmap.org, [Accessed: 15-12-2016].

[50] T. K. project, "Pyton-Openflow low level library to parse and create OpenFlow messages," https://github.com/kytos/python-openflow.git, [Installable via PIP3. Accessed: 15-12-2016].

[51] G. Pickett, "Abusing Software Defined Networks," *DefCon 22, Las Vegas 2014*, 2014.

[52] Y. Tseng, "SDN-Work: multicontrol master-slave," https://github.com/TakeshiTseng/SDN-Work/tree/master/MultiControl/ms, [Accessed:15-12-2016].

[53] M. Antikainen, T. Aura, and Särelä, "Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch," *Secure IT Systems*, 2014.

[54] R. Kloti, V. Kotronis, and P. Smith, "OpenFlow: A Security Analysis," *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2013.

[55] D. Samociuk, "Secure Communication Between OpenFlow Switches and Controllers," *Proceedings of the 7th International conference on Advances in Future Internet (AFIN)*, 2015.

[56] J. Lam, S.-G. Lee, H.-J. Lee, and Y. Oktian, "Securing SDN Southbound and Data Plane communication with IBC," *Hindawi Publishing Corporation Mobile Information Systems*, 2016.

[57] X. Fan, Z. Lu, L. Ju, and D. Mu, "The research on security SDN south interface based on OTR protocol," *Proceedings of the 16th International Symposium on Communications and Information Technologies (ISCIT)*, 2016.

[58] J. van der Lubbe, *Basic methods of cryptography*. VSSD and Cambridge University Press, (1998) 2005.

[59] IBM Knowledge Center, "An overview of the SSL or TLS handshake," https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm, [Accessed: 20-02-2017].

[60] "SSL on Open vSwitch and ovs controller," https://github.com/mininet/mininet/wiki/SSL-on-Open-vSwitch-and-ovs-controller, [Accessed: 20-02-2017, Last edited: 29-04-2014].

[61] "Ryu documentation: Setup TLS Connection," http://ryu.readthedocs.io/en/latest/tls.html, [Accessed: 20-02-2017].

[62] "Wireshark and SSL (Wikipage)," https://wiki.wireshark.org/SSL, [Accessed: 20-02-2017].

[63] R. Bifulco, H. Cui, G. Karame, and F. Klaedtke, "Fingerprinting Software-Defined Networks," *Proceedings of the 23rd International Conference on Network Protocols*, 2015.

[64] A. Azzouni, O. Braham, N. Trang *et al.*, "Fingerprinting OpenFlow controllers: the first step to attack an SDN control plane," *arXiv:1611.02370 [cs.NL]*, 2016.

[65] R. Kandoi and M. Antikainen, "Denial-of-Service Attacks in OpenFlow SDN Networks," *IFIP*, 2015.

[66] N. Solomon, Y. Francis, and L. Eitan, "Floodlight OpenFlow DDoS," https://www.slideshare.net/YoavFrancis/floodlight-openflow-ddos, 2013, [Accessed: 30-03-2015].

[67] "Scapy," www.secdev.org/projects/scapy/, [Accessed:30-03-2017].

[68] R. Arbettu, R. Khondoker, K. Bayarou, and F. Weber, "Security Analysis of OpenDaylight, ONOS, Rosemary and Ryu SDN Controllers," *IEEE*, 2016.

[69] S. Shin, V. Yegneswaren, P. Porras *et al.*, "AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks," *Proceedings of the 2013 ACM conference on Computer and Communications Security, CCS'13*, 2013.

[70] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks," *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[71] L. Wei and C. Fung, "FlowRanger: A Request Prioritizing Alogirthm for Controller DoS Attacks in Software Defined Networks," *Proceedings of the IEEE International Conference on Communications*, 2015.

[72] S. Goa, Z. Peng, B. Xiao *et al.*, "FloodDefender: Protecting Data and Control Plane Resources under SDN-aimed DoS Attacks," *Proceedings of the IEEE International Conference on Computer Communications*, 2017.

[73] A. Craig, "GroupFlow," https://github.com/alexcraig/GroupFlow, 2014, [Accessed: 30-03-2017].

[74] T. Goodspeed, S. Bratus, R. Melgares *et al.*, "Packet in Packets: Orson Welles' in-band signaling attacks for modern radios," *Proceedings of the 5th USENIX Conference on Offensive Technologies, WOOT' 11*, 2011.

[75] Y. AFek, A. Bremler-Barr, and L. Shafir, "Network Anti-Spoofing with SDN Data Plane," *Technical Report, Paper version accepted to INFOCOM 2017*, 2017.

[76] T. Kridel, "How SDN Reduces Network Risks in Campus Settings," https://www.govtechworks.com/how-sdn-reduces-network-risks-in-campus-settings/#gs.ZDSuxRo, 2017, [Accessed: 30-05-2017].

[77] Open Networking Foundation, "SDN Security Considerations in the Data Center," https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-security-data-center.pdf, 2013, [Accessed: 30-05-2017].

[78] A. Lim, "Security Risks in SDN and Other New Software Issues," https://www.rsaconference.com/writable/presentations/file_upload/sec-r01_security-risks-in-sdn-and-other-new-software-apps_copy1.pdf, 2015, [Accessed: 30-05-2017].

[79] R. Chua, "Interview with Phil Porras: Lack of Secure controller Hurting OpenFlow?" https://www.sdxcentral.com/articles/interview/phil-porras-openflow-secure-controller/2012/07/, 2012, [Accessed: 30-05-2017].

[80] Y.-D. Lin, P.-C. Lin, C.-H. Yeh *et al.*, "An Extended SDN Architecture for Network Function Virtualization with a Case Study on Intrusion Prevention," *IEEE Network*, 2015.

[81] "SDNSecurity," http://www.sdnsecurity.org/, [Accessed:30-05-2017].

[82] "Programming the Data Plane in P4," p4.org, [Accessed: 30-05-2017].

A

Code: controller_checker.py

```
#!/usr/bin/env python3
"""
With this script we want to check whether an open TCP port [discovered with nmap],
is used for OpenFlow communication. For this we rely on the OpenFlow handshake, which
in case of controller connection is:
connection -> hello(contr, att); hello(att, contr) -> ftrequest(contr, att) -> ....
in case of switch connection is:
connection -> hello(sw, att) ; hello(att,sw) + ftrequest(att,sw) -> ftreply(sw, att)

In this version (2.0), we are able to find out which controller it is,
looking at the reaction after a we send ftreply(att,cont).
We only support detection of OF1.3 and detection of ONOS, ODL, Ryu and HPE VAN.
"""

import socket
import argparse
import pdb
from pyof.v0x04.common.header import Type
from pyof.foundation.basic_types import UBInt8
from pyof.v0x04.symmetric.hello import Hello
from pyof.v0x04.controller2switch.features_reply import FeaturesReply
from pyof.v0x04.controller2switch.features_request import FeaturesRequest
from pyof.v0x04.controller2switch.common import MultipartTypes
from pyof.v0x04.controller2switch.get_config_request import GetConfigRequest
from pyof.v0x04.controller2switch.flow_mod import FlowMod
from pyof.v0x04.asynchronous.error_msg import ErrorType, BadRequestCode, ErrorMsg

# Process command-line arguments
argParser = argparse.ArgumentParser(description='Check for an given IP-port combination whether it is used
      for OpenFlow 1.3 communication, and if so, which controller is used.')
argParser.add_argument('ip', type=str, help='checked IP address')
argParser.add_argument('port', type=int, help='check port')
argParser.add_argument('--debug', type=int, default=0, help='Set to 1 if you want to debug!')
argParser.add_argument('-v','--version', action='version', version='%(prog)s is at version 2.0')
arguments = argParser.parse_args()

# We only support OF1.3 (all supported by RYU, ONOS, ODL and HPE)
allowed_version = 4
version = 4

""" used for output """
def version_to_str(version):
    if version == 1:
        return "OF1.0"
    if version == 2:
        return "OF1.2"
    if version == 4:
        return "OF1.3"

""" return the integer value of a part of the bytestring message """
```

```
50 def get_value(bytestring):
51     return int.from_bytes(bytestring, 'big')
52
53 """ Given a bytestring, returns whether we get a OpenFlow message and if so,
54     the message_type and other information what comes with it.
55     Simplified version of the method in <imp_hybrid.py> [V1.3.1]
56     Output:
57     1. Boolean saying wether we received a valid OF message
58     2. message_type of OF message / None when invalid
59     3. extra info: parameter dictionary when OF message / explanatory string when invalid
60     4. message residu in case of piggybacked OF messages / None when invalid
61 """
62 def check_message(rec_msg):
63     if len(rec_msg) < 8:
64         return False, None, "received message to small for processing.", None
65
66     msg_version = rec_msg[0]                # should be an legite OpenFlow version
67     msg_type = rec_msg[1]                   # OF message type (integer-form)
68     msg_length = get_value(rec_msg[2:4])    # should be validated against 'adressed length'
69     msg_id = get_value(rec_msg[4:8])        # transaction ID
70
71     msg = rec_msg[:msg_length]              # the message we are going to evaluate here
72     msg_rest = rec_msg[msg_length:]         # if not empty the input contained multiple messages
73
74     # length assertion with advertised length.
75     if len(msg) != msg_length:
76         return False, None, "length assertion failed", None
77
78     # version assertion (first time we adapt the list with allowed versions)
79     if msg_version != allowed_version:
80         return False, None, "not supported version", None
81
82     # Only in the MULTIPART_REQUEST case we need to extract extra information
83     if msg_type == Type.OFPT_MULTIPART_REQUEST.value:
84         mtp_type = get_value(msg[8:10])
85         return True, msg_type, {'trans_id':msg_id, 'mtp_type':mtp_type}, msg_rest
86
87     # Basic case (transaction_id comes in handy when we need to send responses)
88     else:
89         return True, msg_type, {'trans_id':msg_id}, msg_rest
90
91
92 """ Checks whether given sequence of received OF messages matches with expected sequences
93     Output: String which explains the outcome of the check.
94     (V2.0: the parameter (OF) version is not yet used and has always the value 4 (OF1.3))
95 """
96 def check_message_sequence(expected_component, version, messages):
97     print(messages)
98
99     if expected_component == "switch":
100         if messages == [Type.OFPT_HELLO.value,
101                         Type.OFPT_ECHO_REQUEST.value,
102                         Type.OFPT_FEATURES_REPLY.value]:
103             return "OpenFlow connection with switch, version " + version_to_str(version)
104         else:
105             return "Unable to identify OpenFlow connection: Unexpected sequences of OF messages."
106
107     else:   # expected_component == "controller"
108         if messages[0:3] == [Type.OFPT_HELLO.value,
109                              Type.OFPT_FEATURES_REQUEST.value,
110                              Type.OFPT_ROLE_REQUEST.value]:
111             return "OpenFlow " + version_to_str(version) + " connection with OpenDaylight controller"
112
113         if messages[0:3] == [Type.OFPT_HELLO.value,
114                              Type.OFPT_FEATURES_REQUEST.value,
115                              [Type.OFPT_MULTIPART_REQUEST.value, MultipartTypes.OFPMP_PORT_DESC.value]]:
116             return "OpenFlow " + version_to_str(version) + " connection with ONOS controller"
117
118         if messages[0:3] == [Type.OFPT_HELLO.value,
119                              Type.OFPT_FEATURES_REQUEST.value,
120                              Type.OFPT_SET_CONFIG.value]:
```

```
121             if len(messages) < 6 and messages[3] == [Type.OFPT_MULTIPART_REQUEST.value,
122                                                      MultipartTypes.OFPMP_PORT_DESC.value]:
123                 return "OpenFlow " + version_to_str(version) + " connection with Ryu controller"
124
125             elif len(messages) == 6:
126                 mtp_messages = [[Type.OFPT_MULTIPART_REQUEST.value, MultipartTypes.OFPMP_PORT_DESC.value],
127                                 [Type.OFPT_MULTIPART_REQUEST.value, MultipartTypes.OFPMP_DESC.value],
128                                 [Type.OFPT_MULTIPART_REQUEST.value,
129                                  MultipartTypes.OFPMP_TABLE_FEATURES.value]]
130                 # catch case that order of messages is mixed up (although not seen in practice)
131                 if len([m for m in mtp_messages if m in messages[4:7]]):
132                     return "OpenFlow " + version_to_str(version) + " connection with HPE VAN controller"
133
134         return "Unable to identify OpenFlow connection: Unexpected sequences of OF messages."
135
136
137 """ Using a socket, we retreive (and react correctly to) different OF messages,
138     Which are put in the list 'received_list' which is used in the method above,
139     which is called at the moment we expect we can answer what we are dealing with.
140 """
141 def is_openflow(ip, port):
142
143     # if you fail the socket connection, it is not an OpenFlow connection, naturally.
144     # Or we have an TLS connection which is refused.
145     try:
146         s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
147         s.connect((ip, port))
148     except:
149         return "Connection Refused"
150
151     first_message = True                # The first message should always be Hello!
152     received_list = []                  # List containing all (correct) OF messages
153     ready_to_check_controller = False   # As soon as true we go to 'check_message_sequence()'
154
155     # We are not awaiting the controller and initiate Hello
156     s.send(Hello().pack())
157
158     # We want to receive and process messages untill we have an answer
159     while True:
160         rec_msg = s.recv(1024)
161
162         # This loop is to ensure that piggybacked messages are all processed.
163         while len(rec_msg) > 0:
164
165             # note 'info' is a dictionary if is_openflow is True, otherwise a string.
166             is_openflow, msg_type, info, msg_rest = check_message(rec_msg)
167             if is_openflow == False:
168                 return "No OpenFlow connection:" + info         # display reasoning
169
170             # FIRST MESSAGE PROCESSING
171             if first_message and msg_type != Type.OFPT_HELLO.value:
172                 return "No OpenFlow connection: First message should be hello!"
173             elif first_message:
174                 received_list.append(msg_type)
175                 first_message = False
176
177             # SWITCH RESPONSES PROCESSING
178             elif msg_type == Type.OFPT_ECHO_REQUEST.value:
179                 received_list.append(msg_type)
180                 pkt_ft_request = FeaturesRequest(xid=info['trans_id'])
181                 s.send(pkt_ft_request.pack())
182
183             elif msg_type == Type.OFPT_FEATURES_REPLY.value:
184                 received_list.append(msg_type)
185                 return check_message_sequence("switch", version, received_list)
186
187             # CONTROLLER RESPONSES PROCESSING
188             elif msg_type == Type.OFPT_FEATURES_REQUEST.value:
189                 received_list.append(msg_type)
190                 pkt_ft_reply = FeaturesReply(xid=info['trans_id'], datapath_id=44,
191                                              n_buffers=253, n_tables=254,
```

```
192                                         auxiliary_id=0, capabilities=79,
193                                         reserved=0)
194                 s.send(pkt_ft_reply.pack())
195
196             elif msg_type == Type.OFPT_MULTIPART_REQUEST.value:
197                 received_list.append([msg_type, info['mtp_type']])
198                 # In case we receive a PORT_DESC request, it can still be ONOS or RYU.
199                 if info['mtp_type'] == MultipartTypes.OFPMP_PORT_DESC.value:
200                     s.send(b'\x04\x13\x00\x90\x00\x00\x00\x01\x00\r\x00\x00\x00\x00\x00\x00\x00\
    x01\x00\x00\x00\x00V\xf6\xb2\xb8\xc2\x97\x00\x00s1-eth1\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
    x00\x00\x00\x00\x00\x00\x00\x00\x08@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x98\x96\x80\
    x00\x00\x00\x00\xff\xff\xfe\x00\x00\x00\x00>\xf2E#\xe6B\x00\x00s1\x00\x00\x00\x00\x00\x00\x00\x00
    \x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
    x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
201                 else:
202                     ready_to_check_controller = True
203
204             elif msg_type == Type.OFPT_FLOW_MOD.value:
205                 received_list.append(msg_type)
206                 ready_to_check_controller = True
207
208             elif msg_type == Type.OFPT_GET_CONFIG_REQUEST.value:
209                 received_list.append(msg_type)
210                 ready_to_check_controller = True
211
212             elif msg_type == Type.OFPT_ROLE_REQUEST.value:
213                 received_list.append(msg_type)
214                 ready_to_check_controller = True
215
216             # For example BARRIER_REQUEST and SET_CONFIG
217             else:
218                 received_list.append(msg_type)
219
220             # Again: if needed, process the rest before getting new message.
221             rec_msg = msg_rest
222
223         # if rec_msg is all processed, check for controller version if we are ready
224         if ready_to_check_controller:
225             return check_message_sequence("controller", version, received_list)
226
227
228 """ RUNNING SCRIPT """
229 if arguments.ip != None and arguments.port != None:
230     if arguments.debug == 1:
231         pdb.set_trace()
232
233     print(is_openflow(arguments.ip, arguments.port))
```

Listing A.1: Given an IP address and TCP port, this Python script is used to successfully detect OpenFlow connections with OvSwitch or SDN controller (Ryu, ONOS, OpenDaylight, HPE VAN). The identified controllers run with 'standard' implementations. Different controller features can change the used 'handshaking'-sequence, which will influence the success of this script. Some parts (such as the argument-parsing) are omitted in this listing for readability.
The complete code is delivered with this master thesis.

B

Code: switch_impersonator.py

```
#!/usr/bin/env python3
"""
Script to correctly react to received OpenFlow messages in order to impersonate a OF1.3 switch.
Confirmed working for RYU, ONOS and OpenDayLight and HPE VAN
Confirmed working over SSL for RYU
"""

import socket, ssl, time, pdb, argparse
from pyof.v0x04.common.header import Type
from pyof.v0x04.symmetric.hello import Hello
from pyof.v0x04.controller2switch.features_reply import FeaturesReply
from pyof.v0x04.controller2switch.common import ControllerRole
from pyof.v0x04.controller2switch.role_reply import RoleReply
from pyof.v0x04.controller2switch.barrier_reply import BarrierReply
from pyof.v0x04.symmetric.echo_request import EchoRequest
from pyof.v0x04.controller2switch.common import MultipartTypes
from pyof.v0x04.controller2switch.get_config_reply import GetConfigReply
from pyof.v0x04.asynchronous.error_msg import ErrorType, BadRequestCode, ErrorMsg
from pyof.foundation.basic_types import UBInt8

# OMITTED - Code to process command-line arguments

# Requests with empty bodies
empty_requests = [Type.OFPT_HELLO.value, Type.OFPT_ECHO_REQUEST.value,
                  Type.OFPT_FEATURES_REQUEST.value, Type.OFPT_BARRIER_REQUEST.value]
# Messages which don't need a response from the switch
no_response_messages = [Type.OFPT_FLOW_MOD.value, Type.OFPT_SET_CONFIG.value]


""" return the integer value of a part of the bytestring message """
def get_value(bytestring):
    return int.from_bytes(bytestring, 'big')


""" Given a bytestring, return the message_type and other information what comes with it
    Output is 3-tuple: msg_type (int), other_info (dict), msg_rest (int)
        -> The other_info always contains the transaction_id.
        -> extra information is for example needed in case of a RoleRequest / MultipartRequest
        -> msg_rest contains other OF messages if they are 'piggybacked' in rec_msg
"""
def check_message(rec_msg):
    msg_version = rec_msg[0]
    msg_type = rec_msg[1]
    msg_length = get_value(rec_msg[2:4])
    msg_id = get_value(rec_msg[4:8])

    msg = rec_msg[:msg_length]              # the message we are going to evaluate here
    msg_rest = rec_msg[msg_length:]         # if not empty the input contained multiple messages

    # We will always store the transaction ID to be consequent.
```

```
    if msg_type in empty_requests or msg_type in no_response_messages:
        return msg_type, {'trans_id':msg_id}, msg_rest

    # with a ROLE_REQUEST we need also return the requested role and generation_id
    if msg_type == Type.OFPT_ROLE_REQUEST.value:
        req_role = get_value(msg[8:12])
        gen_id = get_value(msg[16:])
        return msg_type, {'trans_id':msg_id, 'role':req_role, 'gen_id':gen_id}, msg_rest

    # For a MULTIPART_REQUEST also the type of request is stored to pick the correct response
    if msg_type == Type.OFPT_MULTIPART_REQUEST.value:
        mtp_type = get_value(msg[8:10])
        return msg_type, {'trans_id':msg_id, 'mtp_type':mtp_type}, msg_rest

    # could be enlarged in future versions, for now 'standard' reaction.
    else:
        return msg_type, {'trans_id':msg_id}, msg_rest



""" IMPERSONATOR """
if arguments.ip != None and arguments.port != None:
    if arguments.debug == 1:
        pdb.set_trace()

    # SETUP SOCKET (With or Without SSL)
    if arguments.privkey != None and arguments.cert != None and arguments.cacert != None:
        print("Impersonation over SSL starts towards", arguments.ip, ":", arguments.port)
        non_ssl_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s = ssl.wrap_socket(non_ssl_s,
                    keyfile=arguments.privkey, certfile=arguments.cert,
                    cert_reqs=ssl.CERT_NONE, ca_certs=arguments.cacert)
        s.connect((arguments.ip, arguments.port))
    else:
        print("Impersonation without SSL starts towards", arguments.ip, ":", arguments.port)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((arguments.ip, arguments.port))

    handshaking_complete = False
    # We send a initiated Hello before getting info from the channel
    s.send(Hello().pack())

    while True:
        # if we are done handshaking we send every 3 seconds a ECHO_REQUEST,
        # sleep(x) is not favorable because it reduces your listening capabilities...
        if handshaking_complete:
            time.sleep(3)
            s.send(EchoRequest().pack())

        # Retrieve message(s) (bytestring) from the controller.
        rec_msg = s.recv(1024)

        if arguments.messages == 1:
            print(rec_msg)

        # we are going to evaluate al OF messages in the received bytestring
        while len(rec_msg) > 0:
            # update rec_msg to the 'rest' of the message (evaluated in next round(s))
            msg_type, info, rec_msg = check_message(rec_msg)
            xid = info['trans_id']

            # reset the response, otherwise we keep sending the last one over and over again
            response = None

            # We already sent an Hello, thus pass and do nothing
            if msg_type == Type.OFPT_HELLO.value:
                pass

            elif msg_type == Type.OFPT_FEATURES_REQUEST.value:
                # This FeaturesReply values are taken from a mininet OVSwitch.
                response = FeaturesReply(xid=xid, datapath_id=44, n_buffers=253,
```

```
                                    n_tables=254, auxiliary_id=0, capabilities=79,
                                    reserved=0)

            elif msg_type == Type.OFPT_BARRIER_REQUEST.value:
                response = BarrierReply(xid=xid)

            elif msg_type == Type.OFPT_ROLE_REQUEST.value:
                # send RoleReply including correct requested role and generation id.
                response = RoleReply(xid=xid, role=info['role'], generation_id=info['gen_id'])

            elif msg_type == Type.OFPT_MULTIPART_REQUEST.value:
                # Responding to a OFPMP_PORT_DESC means reaching 'main mode' in a simple RYU controller.
                if info['mtp_type'] == MultipartTypes.OFPMP_PORT_DESC.value:
                    # hardcoded message saying the switch is connected to one host.
                    s.send(b'\x04\x13\x00\x90\x00\x00\x00\x01\x00\r\x00\x00\x00\x00\x00\x00\x00\x00
                             \x01\x00\x00\x00\x00V\xf6\xb2\xb8\xc2\x97\x00\x00s1-eth1\x00\x00\x00\x00\x00
                             \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08@\x00\x00\x00\x00
                             \x00\x00\x00\x00\x00\x00\x00\x00\x00\x98\x96\x80\x00\x00\x00\x00\xff\xff\xff
                             \xfe\x00\x00\x00\x00>\xf2E#\xe6B\x00\x00s1\x00\x00\x00\x00\x00\x00\x00\x00
                             \x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x00\x00\x00
                             \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
                            ')
                elif info['mtp_type'] == MultipartTypes.OFPMP_DESC.value:
                    # 'standard' OVSwitch reply (hardcoded - 'shorted for display'):
                    s.send(b'\x04\x13\x040\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00Nicira, Inc.
                             \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
                                                ..........
                                                ..........\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
                elif info['mtp_type'] == MultipartTypes.OFPMP_METER_FEATURES.value:
                    # 'standard' OVSwitch reply (hardcoded, especially used towards ONOS)
                    s.send(b'\x04\x13\x00 \xff\xff\xff\xf9\x00\x0b\x00\x00\x00\x00\x00\x00\x00
                               \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
                elif info['mtp_type'] == MultipartTypes.OFPMP_TABLE_FEATURES.value:
                    # create the appropiate ERROR
                    response = ErrorMsg(xid=xid, error_type=ErrorType.OFPET_BAD_REQUEST,
                                        code=BadRequestCode.OFPBRC_BAD_TYPE, data=None)
                # Other MULTIPART_REQUEST's don't get a response (yet)
                else:
                    pass

            elif msg_type == Type.OFPT_GET_CONFIG_REQUEST.value:
                response = GetConfigReply(xid=xid, flags=0x0000, miss_send_len=0xffff)

            else:   # in case we got a message which doesn't need a reply, such as a ECHO_REPLY
                pass

            if response is not None:            # Only send something if we created it.
                s.send(response.pack())

            # For now we say the handshake is over when we have received a SET_CONFIG.
            # Whil this is incorrect for the ONOS case, it doesn't do any harm.
            # FUTURE WORK: Better define when handshaking is complete (differs per controller)
            if msg_type == Type.OFPT_SET_CONFIG.value:
                handshaking_complete = True
```

Listing B.1: This Python script is successfully used to impersonate an OvSwitch towards the four different controllers (Ryu, ONOS, OpenDaylight, HPE VAN) such that we indeed reach the 'main mode' (Ryu) or are found in the GUI topology (ONOS/ODL/HPE VAN). We react hybridly on different received OpenFlow messages. The responses towards MULTIPART_REQUEST's are hardcoded responses which we have seen used by the Mininet OvSwitch - this is because the *pyof* library doesn't support complete MULTIPART_REPLY's. In order to fully connect to the HPE VAN controller, we choose to respond with an `ERROR` instead of a constructed MULTIPART_REQUEST:TABLE_FEATURES response. Namely, creating this is quite some work, because OvSwitch answers with multiple messages because it has 254 tables and it is not possible to set this number lower (for example when experimenting with Mininet) to find such a response. The usage of the `ERROR` was seen in the trace of the communication between OvSwitch and HPE VAN in some settings of Mininet.
In this listing the PORT_DESC response is shortened to increase readability. The complete code is delivered with this master thesis.

C

Elaboration on used Hardware and Software

In this Appendix, we give some extra practical information about *pyof*, *scapy* and the Zodiac FX switch in order to aid people who want to use these libraries or switch in their research or experiments. We also provide direct links on the internet for tutorials or extra information.

C.1. The *pyof* library

The python-openflow (in short, *pyof*) library [50] is a low level OpenFlow messages parser used by the Kytos SDN platform[1]. Using this library, one is able to read OpenFlow packets from an open socket or send different OpenFlow messages. The library uses Python 3 and supports OpenFlow 1.0 to 1.3.

We used the library to impersonate OpenFlow 1.3 traffic. The library was sufficient to create all simple OpenFlow messages, such that we could generate messages needed for the OpenFlow handshake. The recognition of the received OpenFlow messages is done with use of the lists of al `Enum` values for all different OpenFlow messages. It is possible to shift between OpenFlow 1.3 and 1.0 by only changing the *version* parameter in the generated OpenFlow messages, because the structure of the protocol is such that only this is different for most messages - and changing this value is possible in the library.

We have downloaded the library around October 2016. In this version, the library contains some errors and doesn't implement the generation of more complex messages such as different `OFPT_MULTIPART_REPLY` messages. The found errors were small, where incorrect message headers where set. For the missing messages, it is possible to create workarounds or use hardcoded OpenFlow messages (such as presented in our research). The library is still under development, as can be seen by the fact that the library has been updated in March 2017. In this update, we see implementations for some `MULTIPART` messages. Also, we see that our found errors are corrected.

C.2. *scapy*

Scapy is a powerful interactive packet manipulation program which is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more [67]. The library has build-in methods for classical tasks like scanning, tracerouting, attacks or network discovery and can be used as an alternative for *hping*, *nmap* (to a certain extent), *arpspoof* and *arpping*.

In our research, we only used the library to construct our own messages. Custom packets are build following the same hierarchy as packets in the Internet. As example, a correct TCP message is constructed as: `Ether(..)/IP(..)/TCP(..)`. In our research, we mostly created ARP messages: `Ether(..)/ARP(..)`, as can be seen in Listing 5.1.

It is also possible to create packets which do not follow the expected hierarchy, to see whether this leads to vulnerabilities in the network. Also, the current version of *scapy* also allows the definition of OpenFlow messages. These two notions were used when we tried performing the 'packet-in-packet' attack, though this attack was not successful.

[1] For more information, see `https://docs.kytos.io/kytos/`.

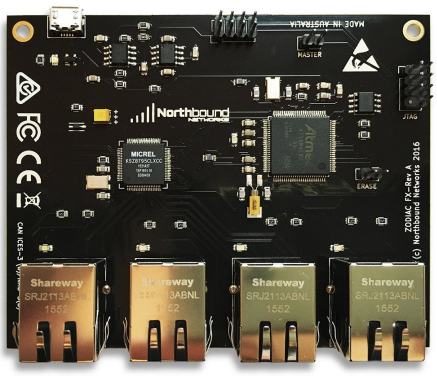

Figure C.1: Picture of the Zodiac FX OpenFlow switch. The ports count up from left to right.

C.3. The Zodiac FX switch

The Zodiac FX OpenFlow switch [43] is developed by Northbound Networks to present an affordable OpenFlow supporting switch which people can use to experiment with SDN at home. With use of a forum[2], starting developers are supported and updated when the new firmware comes out, which is directly downloadable from the forum. There are different tutorials and posts on the internet helping you get started with the switch[3]. The switch can be configured and looked into with use of *minicom*[4]. A useful command in *minicom* is `help`, which lists all possible commands for the switch.

As can be seen in Fig. C.1, the switch counts four ports. The fourth port is always reserved for the connection to the controller, which the switch expect to be on IP-address `10.0.1.8` (but this is configurable). For the other ports, you can configure the ports to support a link part of the OpenFlow network or a native connection. This is done by setting the VLAN type. In this research, we have stated how we configured the ports in the environments setups.

We have worked with firmware $v0.79$. In this case, there is also a GUI available on the IP address of the switch (standard `10.0.1.99`) and it is possible to restart the switch, cleaning the flow table. It is also possible to disable OpenFlow. At that moment, the switch functions as a traditional switch. The newer firmware resolves the security issue that flooded traffic (such as ARP traffic) is also flooding over the connection towards the switch, which makes flooding attacks much easier.

We also experimented with setups with multiple Zodiac switches. We were successful in setting up a network with a out-of-band control channel. Naturally, the Zodiac switch is not advocated for use with an in-band control channel. It is also possible to create a network with a Zodiac switch where OpenFlow is enabled while on connected switches OpenFlow is disabled.

While working with the Zodiac and ONOS as controller we encountered some difficulties. Namely, it not possible to delete or modify flows on the switch. However, the ONOS controller actively sends commands to delete flows. This resulted in the switch having troubles and crash, after which we needed to manually reset the switch.

[2] See `forums.northboundnetworks.com`.
[3] We relied on `http://vk5tu.livejournal.com/55803.html` and `http://adhocnode.com/sdn-raspberrypi-and-zodiacfx/`.
[4] Installable via `sudo apt-get install minicom`.