Fakultät IV
Institut für Softwaretechnik und Theoretische Informatik
Security in Telecommunications

Technische Universität Berlin

Bachelor Thesis

A Feasibility Study of SDN Teleportation in P4Runtime

Konrad Yuri Gbur

November 28, 2018

1st Examiner:	Prof. Jean-Pierre Seifert
2nd Examiner:	Prof. Thomas Zinner
Supervisor:	Kashyap Thimmaraju

Acknowledgments

I sincerely thank my supervisor Kashyap Thimmaraju for supporting my thesis. The research would have not been possible without his knowledgeable input and without him keeping me on the right track. Further, I thank the Security in Telecommunications chair of Prof. Jean-Pierre Seifert for providing a workplace and the necessary coffee.

I also want to thank Brian O'Connor from the Open Networking Foundation for initiating the idea for the topic of my thesis. I thank Robert Krösche, Kashyap Thimmaraju and their co-authors for providing the background of my research by showing the significance of switch identification Teleportation.

Last but not least, I thank my parents for proofreading my English writing and their encouragement.

Contents

1 Introduction **6**
1.1 Thesis Goals 7

2 Background **8**
2.1 Covert Channels 8
2.2 Software-Defined Networking 8
2.3 SDN Teleportation 10
2.3.1 Switch Identification Teleportation 11

3 Threat Model **13**

4 Related Work **14**

5 P4Runtime **15**
5.1 Preliminaries 15
5.1.1 Workflow in P4Runtime 17
5.2 P4Runtime Handshake 19
5.2.1 P4Runtime Handshake in the Specification 20
5.2.2 Uncovering the P4Runtime Handshake in ONOS 20
5.2.3 Network Configuration in Mininet 24
5.2.4 Controller-side Switch Authentication 26

6 P4Runtime Teleportation **28**
6.1 Switch Identification Teleportation with P4Runtime 28
6.1.1 Testing Switch Identification Teleportation in ONOS 29
6.2 Further research 30

7 Conclusion **32**

References **33**

List of Abbreviations **37**

Appendix **38**

List of Figures

1 Topology of an SDN network. 9
2 Message sequence pattern for the OpenFlow handshake. 10
3 Message sequence pattern for switch identification Teleportation in OpenFlow. 12
4 Workflow of programming a target switch with P4. 16
5 Example for message definition in protobuf for a student class. 17
6 Basic example for a service definition with an unary and a server streaming RPC 18
7 Workflow of P4Runtime and gRPC communication. 19
8 Python class to define a topology with one switch. 21
9 Beginning of the pcap for one switch. 22
10 Beginning of a stripped ONOS log with enabled gRPC message log. . . . 24
11 Wireshark capture with HTTP/2 encoding 25
12 Network configuration sent to the ONOS web GUI. 26
13 Message sequence pattern for the P4Runtime Handshake. 27
14 Message sequence pattern for switch identification in P4Runtime. 31
15 Example definition of headers, metadata and parser in $P4_{14}$. 40
16 Basic pipeline of a P4 program. 41

Abstract

Software-defined networking (SDN) is a new paradigm which introduces flexibility into building networks by logically centralizing the control plane. Thus, making it easier and faster to innovate in the respective planes. However, the centralized control plane also introduces novel vulnerabilities. For example, the centralized control plane, as it is a shared resource, can be used to create covert channels. Covert channels in SDN can be realized via SDN Teleportation: a data plane switch sends messages to the control plane which in turn delivers messages to another switch, thus bypassing data plane security mechanisms, and as a result, allowing the two switches to communicate.
In this thesis I investigate the feasibility of a specific SDN Teleportation technique, called switch identification Teleportation, in the state-of-the-art SDN protocol P4Runtime. To show if this attack is feasible, I examine the P4Runtime handshake to uncover the mechanism identifying switches by the controller. Furthermore, I introduce first steps towards mapping the OpenFlow handshake with P4Runtime.

Zusammenfassung

Software-defined networking (SDN) ist ein neues Paradigma welches einen flexibleren Ansatz für das Erstellen von Netzwerken ermöglicht. Die Kontrollebene (control plane) wird zentralisiert und macht es somit einfacher und schneller, Neuerungen und Innovationen in das Netzwerk einzupflegen. Auf der anderen Seite entstehen durch die Trennung von Datenebene (data plane) und Kontrollebene neue Schwachstellen, welche zur Erstellung verdeckter Kanäle (covert channel) genutzt werden können. "SDN Teleportation" verwendet die Kontrollebene um einen verdeckten Kanal zu realisieren. Durch das Ausnutzen der SDN Protokollnachrichten zur Kommunikation, können Sicherheitsmechanismen in der data plane umgangen werden.
In dieser Arbeit zeige ich ob ein verdeckter Kanal durch "switch identification Teleportation" auch im P4Runtime Protokoll existiert. Dafür beginne ich mit der Untersuchung des P4Runtime Handshake, inklusive der vorhandenen Identifikationsmechanismen für Netzwerkswitches. Des Weiteren, schlage ich erste Schritte vor um die Schwachstellen im OpenFlow Handshake auf P4Runtime zu übertragen.

1 Introduction

Despite innovative technologies and new paradigms in the application layer, the way packets are distributed in lower layers has been nearly the same since the IP protocol was introduced. Switches follow pre-set algorithms to route packets to their destination. Thus, the introduction of new routing methods could lead to the necessity to change the whole network infrastructure. This static approach slowed down innovation and the development of new technologies. Furthermore, "existing communication protocols and architectures were unable to meet the increasingly stringent requirements [...] of growing networks" [1]. Hence, demanded performance and dependability were not achievable [2, 1].

One approach, addressing the "ossification" of networks, is software-defined networking (SDN). It divides the network into a data plane which forwards traffic according to flow tables and into a centralized control plane. The control plane contains a protocol specifying the communication to the switches and routing algorithms that are used to populate flow tables. Since the controller is connected to all switches it is able to survey the whole network. Therefore, reactions to events are performed more efficiently. This novel concept of abstraction allows developers to introduce new control plane algorithms quicker while making the network more dynamic and better manageable [1].

However, the division of data and control plane in SDN demands an additional protocol to implement the novel network infrastructure. Non-malicious messages may be used to establish a covert channel which was not intended to be possible in the original protocol design. They can be utilized to co-ordinate attacks or exfiltrate data without detection [3, 1]. This vulnerability in software-defined networks, where data is transmitted via the control plane, was discovered by Thimmaraju et al. [3] who named it "SDN Teleportation". They introduced multiple techniques to realize these covert channels. One of them is an implicit covert channel using mechanisms that show how switches uniquely identify themselves to the controller. Therefore, this technique is called switch identification Teleportation. It was originally introduced as a possibility to realize a rendezvous protocol for malicious switches in the above mentioned article [3]. Krösche et al. [1] describe an approach for using the switch identification method in order to implement a covert timing channel, thereby accomplishing respectable data rates for arbitrary binary files. Through this method the channel becomes more versatile.

Different implementations of the SDN approach have been developed over the years [4]. The leading protocol for the last 10 years has been OpenFlow. Despite being the most established protocol for realizing software-defined networking, it became the target of some critique regarding its flexibility. OpenFlow was developed for switches with fixed behavior. Such a non-extendable switch design was lead by the speed advantages of a hard-coded "silicon-based" switching logic. Therefore, the static switch design in OpenFlow is only able to add entries to tables of well known protocols. The first version of OpenFlow was exclusively supporting Ethernet, VLAN, IPv4 and ACL. McKeown and

Rexford [5] summarize this approach as: “OpenFlow doesn’t really control the switch behavior, it gives us a way to populate a set of well-known tables”. Introducing novel protocols was only possible by adding new header types.

However, the release of the P4 language changed the thinking of switches being non-programmable. The P4 language introduces the possibility to freely program the behavior of a switch. This new feature comprises ways to enhance the static SDN approach. In order to achieve interoperability of different implementations of an SDN protocol using the P4 language, P4Runtime was introduced [6]. Since the interest in P4Runtime has been growing, our aim is to uncover if SDN Teleportation could be realized within this protocol.

1.1 Thesis Goals

The objective of this thesis is to test the feasibility of switch identification Teleportation with P4Runtime. To achieve this goal, I have adopted the following, structured approach:

1. Set up a test environment, emulating a software defined network using P4Runtime with an ONOS controller.
2. Identify the messages comprising the P4Runtime handshake.
3. Understand how P4 switches are identified by the controller.
4. Verify, if the uncovered identification mechanisms can be utilized for switch identification Teleportation.

2 Background

This section introduces the necessary background information to follow the main part of the thesis, i.e., covert channels and SDN Teleportation as introduced by Thimmaraju et al. [3].

2.1 Covert Channels

Covert channels are communication channels not intended in the original design of a system. They can circumvent "[...]security policies, thereby leading to unauthorized information disclosure" [1]. "A covert timing channel uses a temporal or ordering relationship among accesses to a shared resource" [7] to transfer data covertly. It is not always possible to circumvent such covert channels entirely, especially in complex systems. Their significance and imposed threat is based on their bandwidth and on the difficulty to detect them. Thus, the focus of preventing covert channels lies on reducing the data throughput and unveiling the shared resource [8].

2.2 Software-Defined Networking

As already mentioned in the introduction, software-defined networking is a paradigm that divides the network in two planes. One plane contains the data which is forwarded through the network via the switches from one host to another. The other plane contains the communication with a logically centralized controller. Furthermore, the controller is connected to all switches in the network and is responsible to supervise the forwarding behavior of the data plane by populating tables according to its routing algorithms. The controller's knowledge of the whole topology entails an advantage over the classical network design. With this additional information, it can react more efficiently to events in the network. Thus, traffic can be re-routed in the most efficient way when experiencing, e.g., a node failure. It is also easier to implement new routing algorithms as only the controller has to be updated, which then updates the flow tables of the switches accordingly. In comparison, the classical approach required all switches to be updated individually which due to hard-coded behavior was sometimes impossible [9, 3].

An essential part of the SDN approach is a protocol specifying the way switches communicate with the controller. Such a protocol has to contain mechanisms for the inclusion of new switches to the topology (handshake) and for the alteration of a switch's behavior by the controller. All the messages defined by an SDN protocol add up to the control plane. For the following explanation I will use the message definitions of OpenFlow [10] as it is the most common SDN implementation. Also the vulnerability described in Section 2.3

is also based on it. As the main emphasis of this thesis does not lie on SDN in general, I concentrate on introducing the most important parts of the OpenFlow protocol necessary to understand switch identification Teleportation. For the following explanation, I refer to a topology containing one controller (c0) connected to two adjacent switches (s1 and s2) both being connected to one host (h1 and h2) as shown in Figure 1.

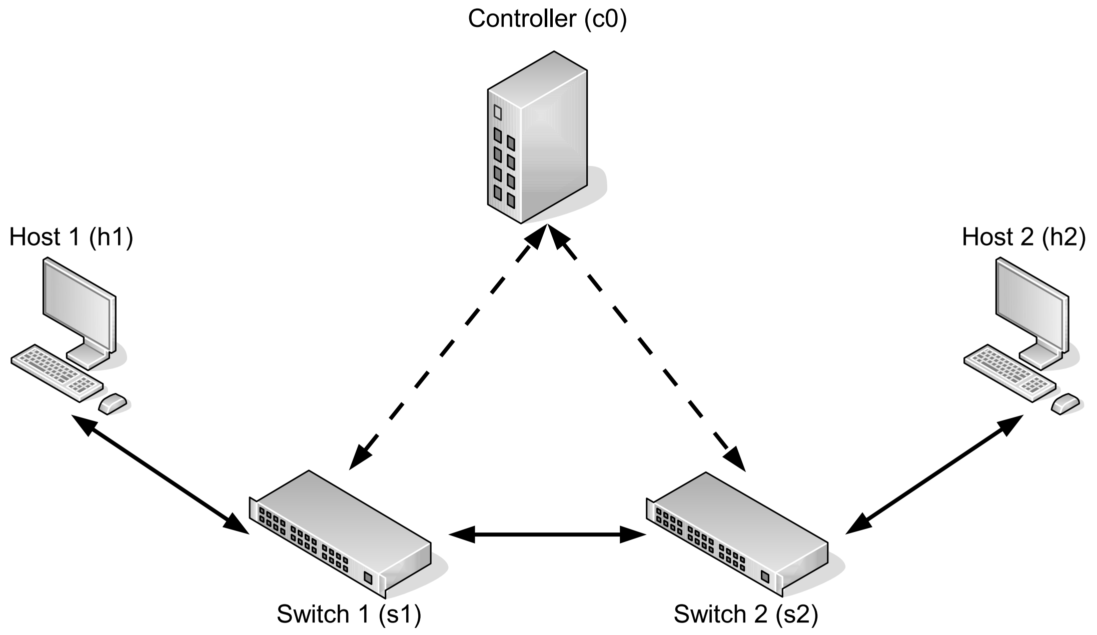


Figure 1: Topology of an SDN network with two switches, two hosts and one centralized controller.

OpenFlow Handshake In OpenFlow a TCP connection to the controller is typically initiated by the switch as shown in Figure 2. Transport Layer Security (TLS) can be utilized for further authentication and for encryption of following messages. With the connection established, the switch can continue sending the necessary OpenFlow messages. These consist of a *Hello* message sent to the controller which the controller answers with its own *Hello* message. These two messages are used to determine which OpenFlow versions are supported and which version will be used in subsequent messages. Following the negotiation of the version to be used, the controller sends a *Features-request* message and the switch answers with a *Features-reply.* This *Features-reply* contains information and settings from the switch including a *Datapath ID* (DPID) which is needed to uniquely identify switches. The handshake is typically completed with a modification of the flow rules (*Flow-mod*) sent by the controller [1].

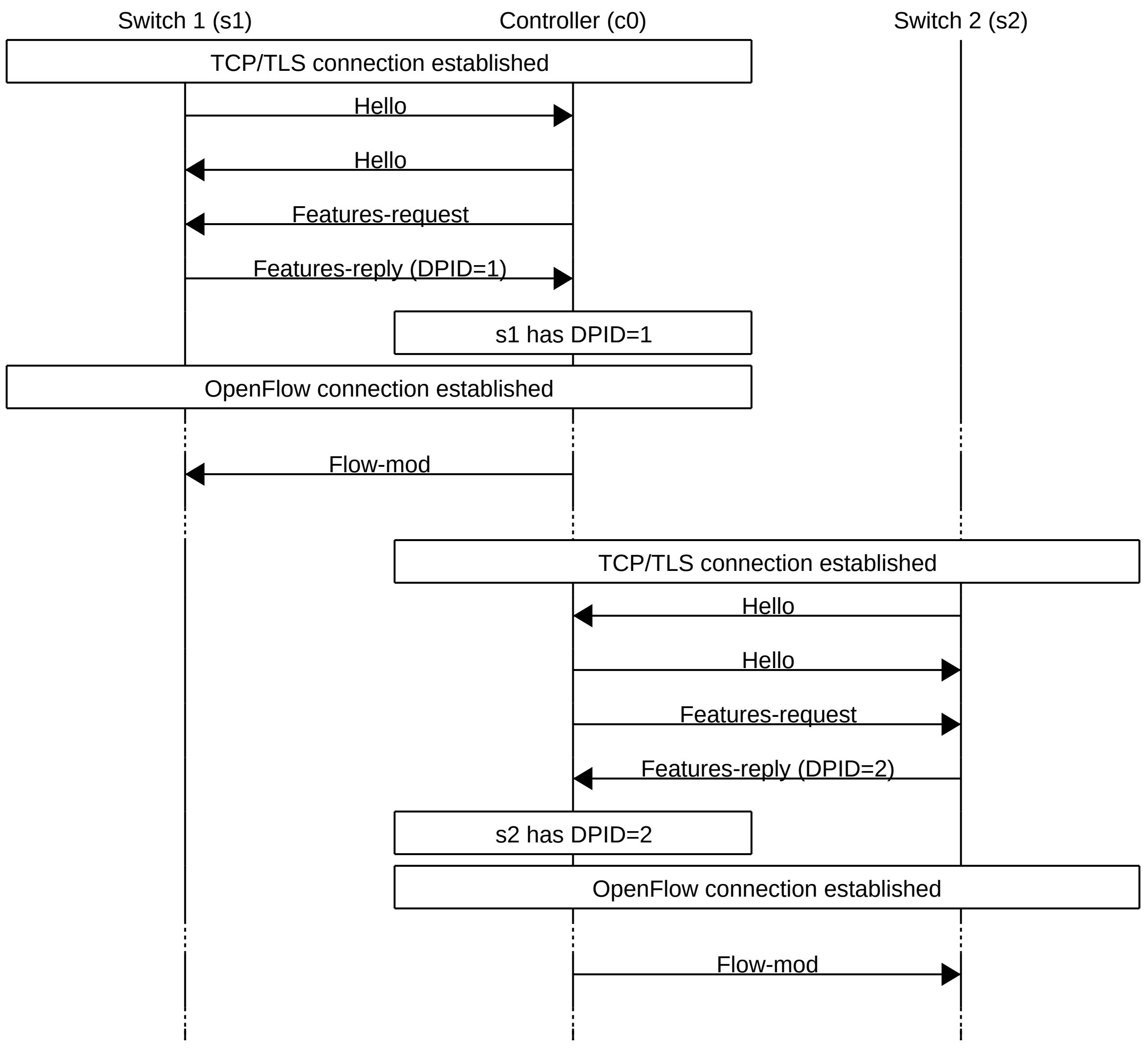


Figure 2: Message sequence pattern for the OpenFlow handshake according to [11].

2.3 SDN Teleportation

SDN Teleportation as introduced by Thimmaraju et al. [3] is the vulnerability of utilizing protocol messages as a covert channel. Teleportation targets "[...]the outsourcing and consolidation of control over multiple data plane elements" [3]. In OpenFlow the controller acts as a "reactor" to events triggered by messages from switches, timeouts or the network operator. According to such an event, messages are sent to one or multiple switches. Finally a switch processes incoming messages. These three stages are the essential functionality of the SDN approach in OpenFlow and they are important for modeling SDN Teleportation. The described functionality leads to two different types of Teleportation channels. The information can be actually embedded in the messages sent which is called explicit Teleportation. The information can also be transferred via an implicit

Teleportation channel. Such a channel uses timing and ordering of messages or the state of an shared resource which can vary over time.

Thimmaraju et al. [3] introduce three Teleportation techniques which have been shown to work in OpenFlow. The first one is called flow (re-)configuration. It is an implicit technique, using the controller's reaction to data plane events and the reconfiguration of the underlying network by the latter. The second one is out-of-band forwarding, an explicit Teleportation channel exploiting how a controller forwards packages when connected to multiple switches. The third one is switch identification Teleportation, an implicit channel utilizing the unique identification of switches to the controller. In this thesis I will focus on the last technique.

2.3.1 Switch Identification Teleportation

Switch identification Teleportation is a covert timing channel utilizing the controller as the shared resource. As mentioned in the introduction of the handshake, OpenFlow uses a *Datapath ID* (DPID) to identify switches. To be functional it is necessary that all DPIDs in a OpenFlow network are unique. This core feature is the target of switch identification Teleportation. The controller has to enforce that all DPIDs transmitted in the handshakes are only used once. If the same DPID is used twice the following behavior can be observed.

For the first switch s1 there are no deviations from the normal handshake since its DPID has not yet been assigned. After the first switch is connected the second switch s2 tries to connect with the same DPID, e.g., 42. Receiving a *Feature-reply* message with DPID=42 from s2 the controller can respond in different ways:

- It can deny the second switch to connect and terminate the TCP connection. This behavior is shown in Figure 3 and is typically the behavior for Open Network Operating System (ONOS) controllers [11, 12].
- The controller can disconnect s1 and complete the handshake with s2.
- The controller accepts both switches and distinguishes them via different *Role-request* messages.

Either way the behavior of the controller can be used for Teleportation [3]. Take, for example, the message pattern as shown in Figure 3: When s1 is already connected and s2 tries to connect with the same DPID, s2 gets rejected. Thus, information is implicitly teleported from s1 to s2. When s1 is not connected s2 gets connected and a different, easily distinguishable state is transmitted. These two states can be utilized to sent arbitrary binary data from s1 to s2 as shown by Krösche et al. [1]. The authors were able to achieve a throughput of 20 bits per second even under high controller load with an accuracy of about 90%. This amount of throughput could lead to a possible exfiltration of an 2048 byte RSA-key in about 13 minutes.

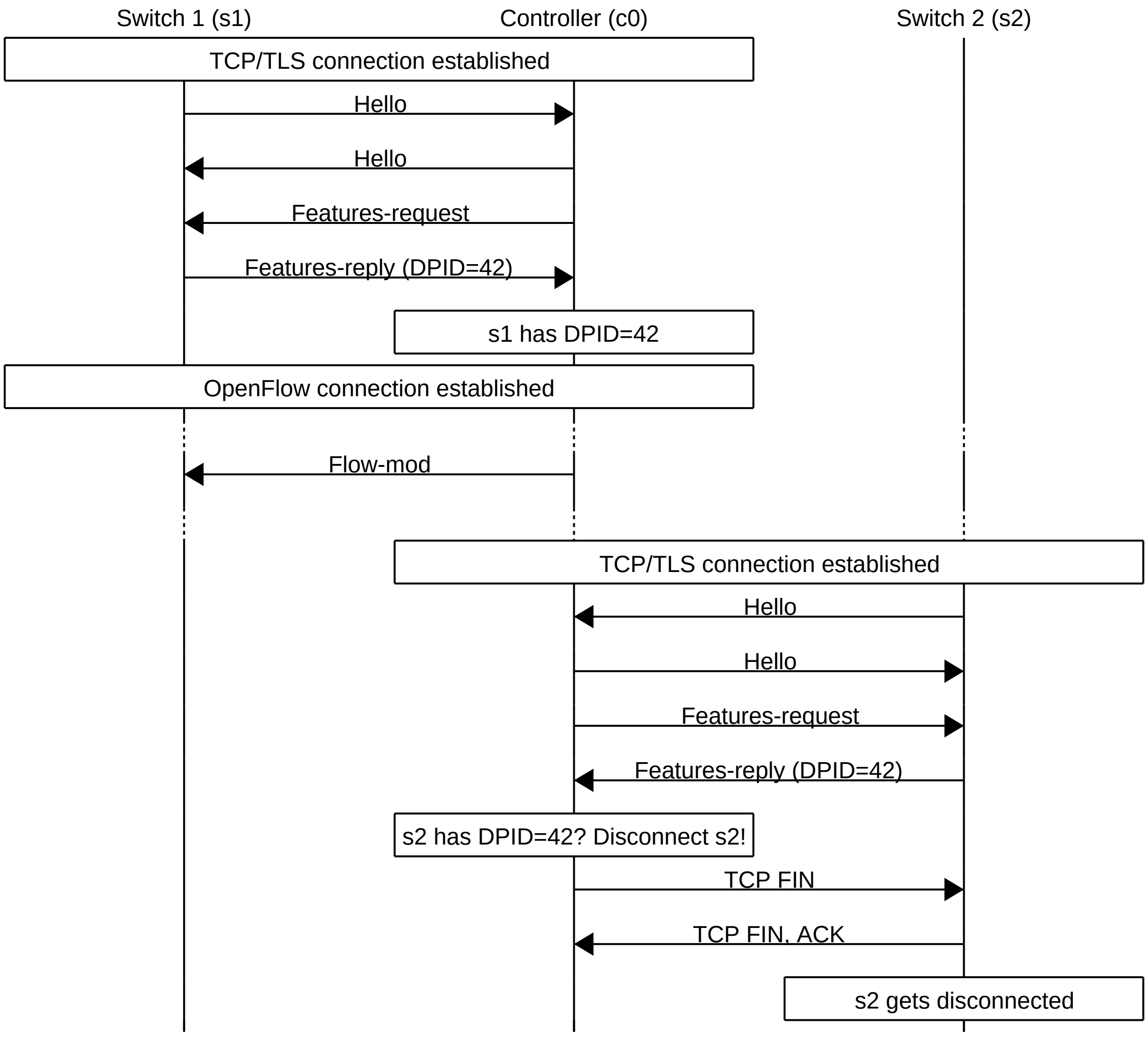


Figure 3: Message sequence pattern for switch identification Teleportation in OpenFlow according to [11].

3 Threat Model

We assume that switches in P4Runtime can be malicious, without any restrictions on what a malicious switch can do. For example, it should be able altering its own device identification, as well as sending messages to the controller at any time. Compromising the switch is beyond the scope of this thesis. However, previous work has shown how an OpenFlow switch [13] can be attacked by exploiting parsing vulnerabilities, compromising the supply chain [14] or utilizing a backdoor [15]. Additionally, we do not place any restrictions concerning the attacker's background, i.e., if he is an administrator with intentional access or if he gains access through other vulnerabilities present on the devices. The objective of the switches is to communicate covertly and to remain undetected in the network while, for example, transmitting private keys, exfiltrating confidential data or performing an attack coordination. These objectives may have to be achieved in the presence of network security mechanisms.

In general, the network topology is not under the control of the attacker. Switches can be logically and physically separated from another. However, it is important that the switches have a connection to the same controller. If there are multiple physical controllers involved it is necessary that they appear to the switches as the same logical controller. The IP address of the controller is known by the switches, e.g., via prior connections.

On the contrary, the controller is a trusted entity. Therefore, all actions performed by the controller are executed according to their definition. The channels which the controller uses to talk with the switches are reliable but not necessarily secure. This threat model is based on SDN Teleportation as described by Thimmaraju et al. [3] and Krösche et al. [1]. Based on the findings in our research, we discuss changes to this threat model in Section 6.

4 Related Work

This thesis was mainly inspired by the papers from Thimmaraju et al. [3] and Krösche et al. [1] which for the first time contain the concepts and the ideas of SDN Teleportation. Deeper insight into OpenFlow, examined in these two papers, can be found in [9, 10]. Further research about security in SDN-based Cloud Systems is available in [13]. The paper by Hu et al. [16] examines covert storage channels in SDN and detection mechanisms.

P4 and P4Runtime are still in development. Further information about them can be found on the website of the P4 Language Consortium [17], in the specification [18] and in the github for the P4Runtime project [19]. P4Runtime uses the gRPC Network Management Interface (gNMI) based on the google Remote Procedure Call protocol (gRPC) [20] and on protocol buffers (protobuf) [21, 22].

Nevertheless, there are some studies about security in the P4 language. Liu et al. [23] implement a tool to verify data plane behavior for devices using P4. Kabasele Ndonda et al. [24] introduce a network intrusion detection system applied to P4. The papers by Agape et al. [25, 26] explain general threats for SDN with P4. To the best of our knowledge, this is the first scientific work studying covert channels in P4Runtime.

5 P4Runtime

Announced in October 2017, P4Runtime is an SDN protocol that appears to be gaining traction in industry [6] and academia [27]. With increasing interest in this protocol, more organizations might choose to adopt P4 and P4Runtime. Thus, security aspects are going to be more important. This section introduces the components of P4Runtime as well as our test environment. Moreover, we present the P4Runtime handshake and its identification mechanisms.

5.1 Preliminaries

P4Runtime was announced by Google and Barefoot Networks [6] as one approach to standardize a communication protocol between a controller and P4 switches. P4 is a domain specific language to program the packet-processing pipeline of a switch. It is an attractive architecture to be used with SDN as it overcomes the static OpenFlow approach. Since the P4 language is not the main topic of this thesis, I will not go into greater detail about its mechanics. However, having a basic knowledge of the underlying concepts will be helpful in understanding P4Runtime.

P4 language The name P4 comes from the original paper introducing it as "Programming Protocol-independent Packet Processors" [28]. It targets the already mentioned hard-coded behavior of classical switches by giving a programmer the possibility to freely define the forwarding behavior of the data plane. This includes full control over the packet-processing pipeline in the data plane, i.e., parsing, matching and performing actions on matched packets. This allows the P4 programmer to introduce non-standard or proprietary protocols into the pipeline.
P4 programs are compiled by the P4 compiler to create two artifacts. It generates the data plane runtime and an application programming interface (API), which both are loaded to a switch as shown in Figure 4. The API provides the control plane with access to data plane objects such as flow tables. However, the implementation of the control plane is not in the scope of P4. Compiling the P4 programs allows for target independence: one can write a single P4 program and compile it to an FPGA switch, ASIC NIC, NPU or simply an x86 software switch. With the compiler taking care of device-specific details, it is possible to reconfigure switches in the field by simply updating the program, recompiling it and loading it to the switch [29, 30].

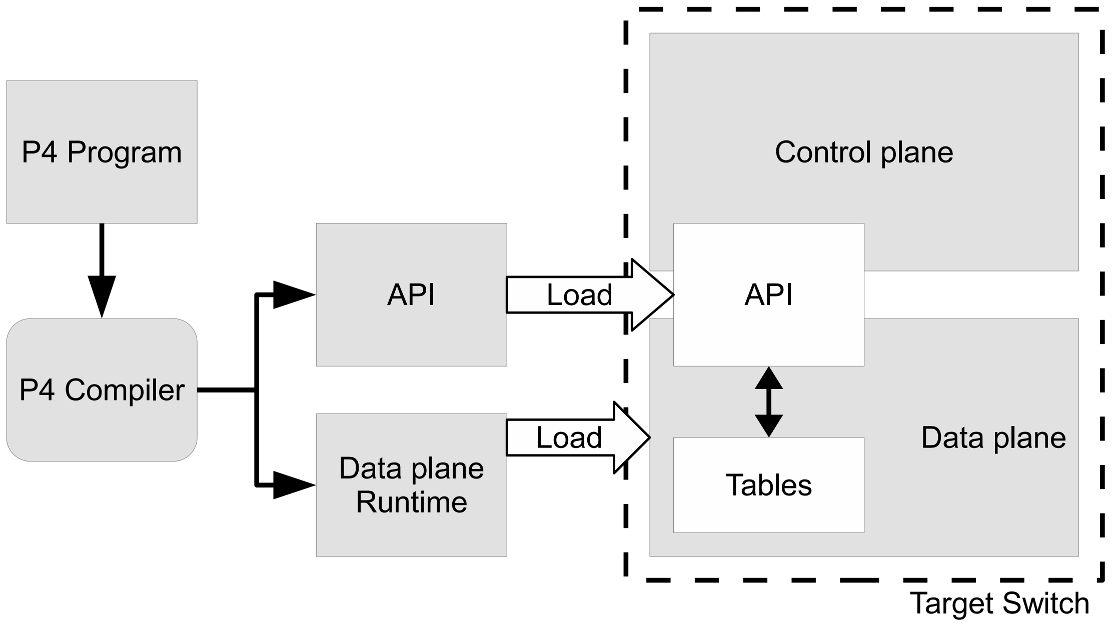


Figure 4: Workflow of programming a target switch with P4 according to [29]

As the P4 language itself already supports the control and data plane separation in its design, the outsourcing of the control part is generally possible. However, P4 does not specify which protocol is used to realize the communication between controller and switch. This makes interoperability between different implementations of an SDN protocol using P4 challenging. Therefore, the P4 Working Group introduced the idea of a uniform specification for a protocol which solves possible compatibility issues. P4Runtime is such an approach; it uses protocol buffers (protobuf) [21] to define messages and Google's remote procedure calls (gRPC) [20] to transfer the data [6].

Protobuf Protobuf is a markup language for serializing structured data. It is close to the extended markup language (XML) as it also uses key-value pairs, but its goals are different. Protobuf in its raw binary form is optimized for computational processing and not for human readability. Therefore, encoded protobuf messages are only meaningful if the message definitions are available. However, parsing of messages is faster and the encoded files are smaller than a comparable XML representation [22].

A `.proto` file consists of *message* definitions. It contains various basic data types like *bool*, *int32*, *float*, *double* and *string* or nested message definitions which can be used as data types as well. The *enum* type is a special data type used if only a list of predefined values should be possible for a key. The number associated with each key is a unique identifier used in the binary encoding. Each field in a message definition has to have one

```
message Student {
        required string name = 1;
        required int32 matrikelnr = 2;
        optional string email = 3;

        enum PhoneType {
                MOBILE = 0;
                HOME = 1;
        }

        message PhoneNumber {
                required string number = 1;
                optional PhoneType type = 2 [default = MOBILE];
        }

        repeated PhoneNumber phones = 4;
}
```

Figure 5: Example for message definition in protobuf for a student class.

of three modifiers. *Required* fields have to be provided with a value otherwise the message will be regarded as "uninitialized" and will raise an exception while parsing. *Optional* fields may or may not be set. If no value is set, a preset default value is inserted for standard data types or an own default value should be set for composite types. *Repeated* fields can be used any number of times including zero. In the binary format the order of the repetitions is preserved similar to a dynamic array [22, 31]. An example protobuf definition is shown in Figure 5.

gRPC Google Remote Procedure Calls (gRPC) is a protocol utilizing protocol buffers to send program states to a remote server which then executes some calculations and returns the results back to the client. The protocol takes the native protobuf *message* type to create structures which then can be used in the gRPC messages. In addition to the native protobuf types, gRPC adds a *service* structure that defines the RPC messages. There are four different types of service methods. The most simple one is an unary call where client and server both send a single request and response. The other three types are streaming RPCs: server streaming, client streaming or bidirectional streaming [32]. Figure 6 shows a basic service definition for gRPC.

5.1.1 Workflow in P4Runtime

This section introduces the workflow of P4Runtime according to the specification [18]. It should be mentioned that this specification was not available when I started my research. In later sections it will primarily be used to verify results.

The messages for P4Runtime are defined in the `p4runtime.proto` file [33] which is used to generate the gRPC client and server stubs. The controller contains the gRPC

```
service HelloService {
        rpc SayHello (HelloRequest) returns (HelloResponse){
        }
        rpc ManyHellos (HelloRequest) returns (stream HelloResponse){
        }
}

message HelloRequest {
        string greeting = 1;
}
message HelloResponse {
        string greeting = 1;
}
```

Figure 6: Basic example for a service definition with an unary and a server streaming RPC according to [32].

client and the switch contains the gRPC server. Since RPC communication is always established by the client (controller), the switch can not initiate TCP connections. This is a fundamental difference to the OpenFlow protocol. However, P4Runtime switches need a possibility to contact the controller in order to send, for example, *Packet-in* messages. To ensure this functionality the `p4rtuntime.proto` specifies a bidirectional *StreamChannel* message type. This stream is also utilized to indicate session liveness and to perform *MasterArbitration*.

A *MasterArbitration* is necessary as P4Runtime supports multiple controllers. Moreover, a controller in P4Runtime can be embedded in the target switch or it can be remote in order to realize an SDN approach. Different controllers can either be used as redundancy to achieve high availability or to split control plane tasks among them. For example, one controller is responsible for routing whereas the other one is responsible for statistics. Furthermore, embedded and remote controllers can be combined. An embedded device generally has a lower latency to the switch, thereby making it preferable for time-critical tasks.

A P4 program is compiled into a target-specific device configuration and into the P4Info metadata which specifies the entities on a switch accessible by the controller. Both artifacts generated by the P4 compiler are installed in the switch (gRPC server) via the *SetForwardingPipelineConfig* RPC as shown in Figure 7. The entities specified in the P4Info metadata can be accessed and altered via the *Read* and *Write* RPC messages during run time [18].

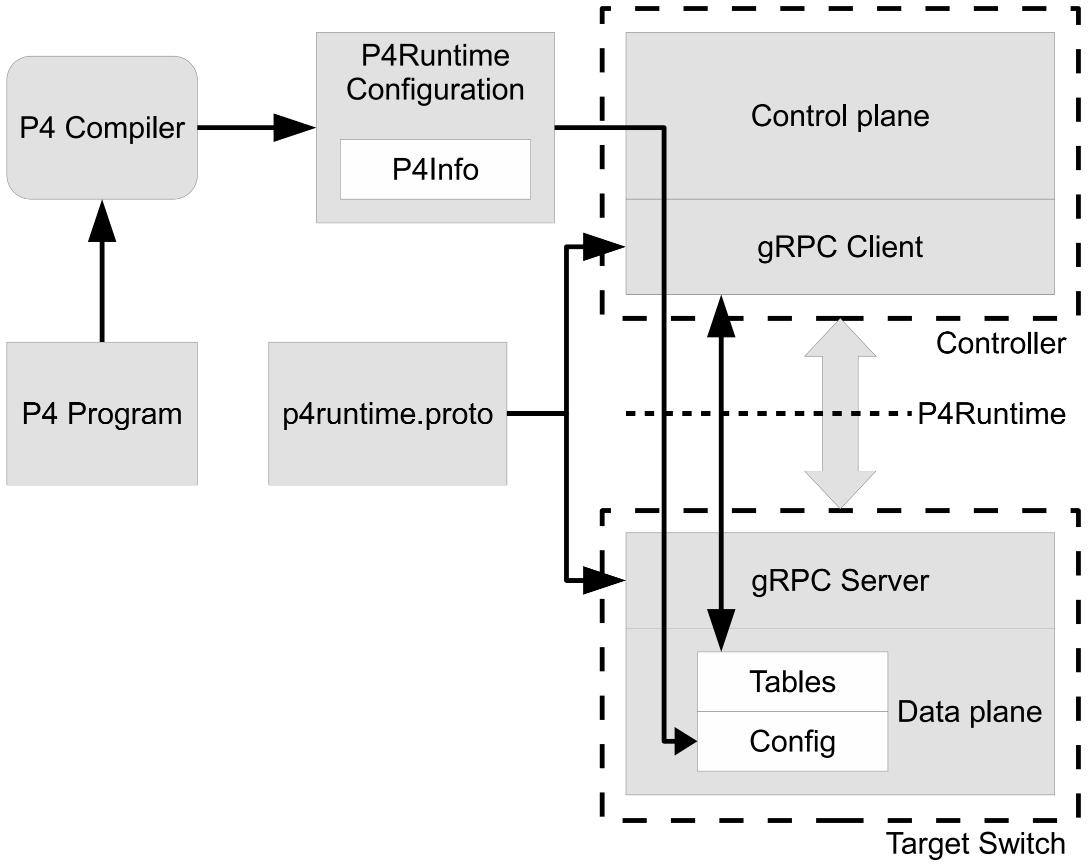


Figure 7: Workflow of P4 Runtime and gRPC communication for one switch and one controller according to [18]

5.2 P4Runtime Handshake

In this section we examine the P4Runtime handshake between controller and switch. Therefore, we present the handshake related messages mentioned in the P4Runtime specification [18]. Furthermore, we collect all the communication between a P4 switch and a controller in an emulated network, thereby gaining insight into a specific P4Runtime implementation. At last, we combine these insights and the specification to uncover the following details:

- Who initiates the P4Runtime connection?
- Which messages are sent to the controller by the switch?
- Which messages are sent to the switch by the controller?
- How are switches identified by the controller?
- Which messages are present in the implementation but do not belong to P4Runtime?

These details will determine the approach for switch identification Teleportation in P4Runtime.

5.2.1 P4Runtime Handshake in the Specification

According to the P4Runtime specification [18], opening the *StreamChannel* is the first step of the handshake. It is immediately followed by a *StreamMessageRequest*, populating the `device_id`, `role_id` and the `election_id` in the *MasterArbitrationUpdate* field. Depending on the values received, the switch responds with one of the following messages.

- If the `device_id` is unknown to the P4Runtime server, it terminates the *StreamChannel* with a `FAILED_PRECONDITION` error.
- If the `election_id` equals the one of an already connected controller with the same `device_id` and `role_id`, the *StreamChannel* is terminated with an `INVALID_ARGUMENT` error.
- If the maximum number of clients with the same `device_id` and `role_id` is exceeded, the *StreamChannel* is terminated with a `RESOURCE_EXHAUSTED` error.
- If the given values are valid, the switch adds the controller to a list of known clients and returns a *StreamMessageResponse*.

Next, in order to achieve an operational forwarding pipeline, the P4Runtime target has to be configured via the *SetForwardingPipelineConfig* RPC. The *SetForwardingPipelineRequest* also contains the `device_id`, `role_id` and the `election_id` as introduced in the *StreamMessageRequest*. If the `device_id` is not recognized or the `role_id` does not match any role negotiated via the *StreamChannel*, the switch returns an `INVALID_ARGUMENT` error.

However, both RPCs used in the P4Runtime handshake need the `device_id` of a switch in advance. The exchange of the necessary configurations is not within the scope of the P4Runtime specification [18].

5.2.2 Uncovering the P4Runtime Handshake in ONOS

In the following, I present our approach to uncover the entire P4Runtime handshake implementation of an ONOS (Open Network Operating System) [12] controller and of bmv2 (behavioral model version 2) [34] software switches. I set up a test environment with a program to create emulated networks called mininet [35]. I used the provided virtual machine (VM) by ONOS [36] which already contains the software described.

In order to uncover the messages exchanged during the handshake, I captured the packets sent between the controller and the switch by using wireshark [37]. However, unlike OpenFlow, for which a wireshark dissector is available, dissecting P4Runtime is non-trivial due to the inherent nature of P4Runtime, i.e., its usage of gRPC and of protobuf (see Section 5). When I started my research I used the VM released in February 2018. Due to helpful updates in the ONOS controller I switched to the one from August 13th, 2018 [36]. The same holds true for the wireshark versions. With the new Version (2.6.4), one could use the HTTP/2 dissector to decode at least gRPC header information [38]. Further information on the usage of the test environment will be found in the appendix or by visiting the ONOS wiki [36].

To study the the ONOS P4Runtime handshake, we started with one switch and one controller. This eliminated noise when collecting packet traces and also simplified the analysis. Hence, we modified the default mininet script that came with the VM by removing the two hosts (Figure 8) and by modifying the gRPC port to a static value of our choice. In this case we chose port 55551. For the second switch, deployed later on, we chose port 55552.

```python
# Own topology with 1 switch.
class MyTopo(Topo):

    def __init__(self):
        "Create custom topo."

        # Initialize topology
        Topo.__init__(self)

        # Add switch
        switch = self.addSwitch('s1')
```

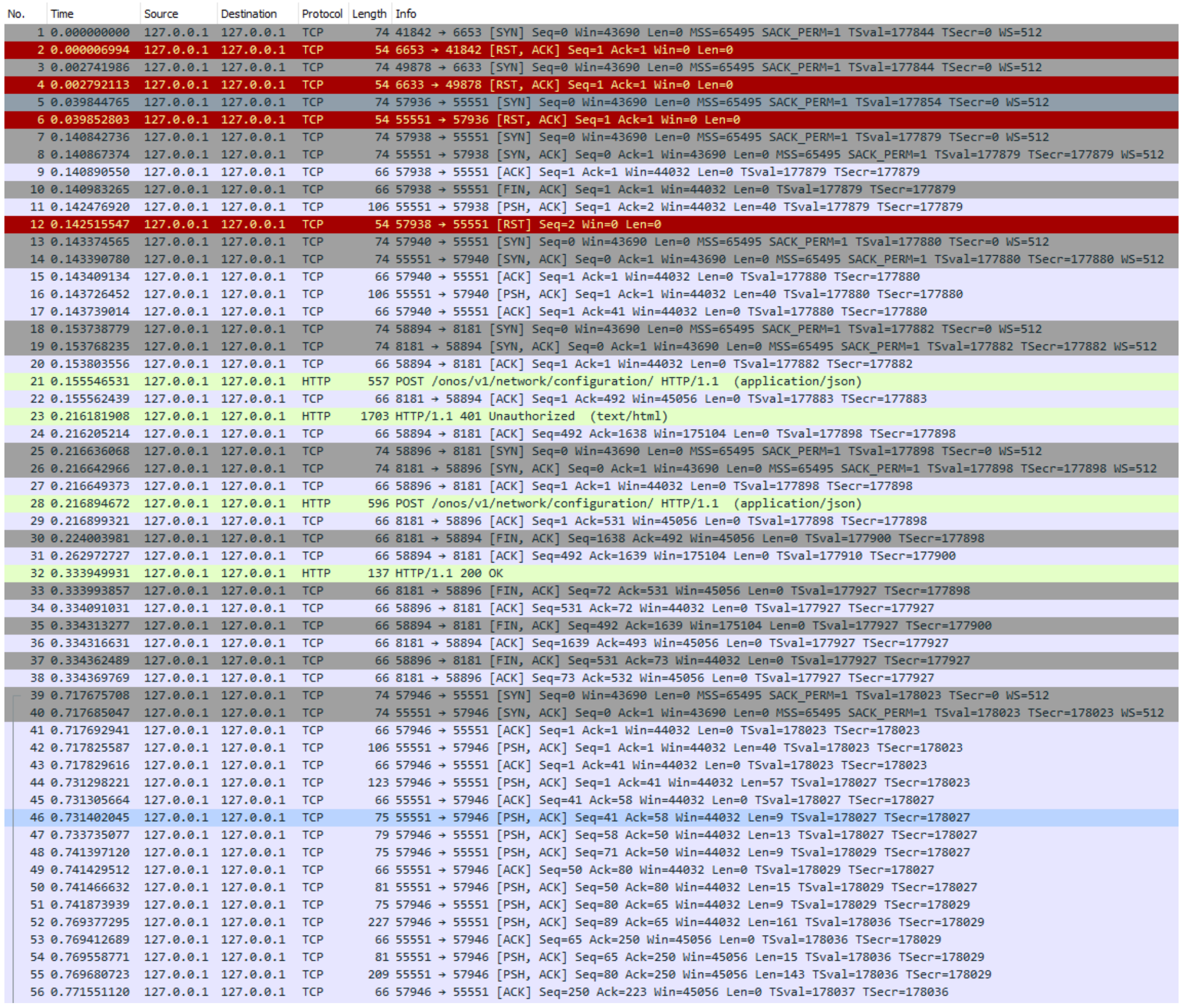
Figure 8: Python class to define topology with one switch.

Collecting and Analyzing the Network Traffic To minimize the size of the wireshark packet capture (pcap), I execute the following steps:

1. Start ONOS.
2. Establish a connection to the ONOS CLI (Command Line Interface) after finishing the initialization.
3. Start the wireshark capture.
4. Create a switch by starting the mininet script.
5. Terminate the wireshark capture approximately 15 seconds after the switch is shown in the ONOS CLI to ensure all relevant packets involved in the handshake are captured.

The workflow above avoids capturing ONOS CLI related communication. The first packets from the capture are shown in Figure 9. The complete pcaps are available at the following link `https://tubcloud.tu-berlin.de/s/9tYFjpfQ255nkww`.

The capture shows TCP SYN packets to ports 6653, 6633 and 55551. We can eliminate the first two ports as they are well known OpenFlow ports. These messages are present



```
No.  Time           Source     Destination  Protocol Length Info
   1 0.000000000   127.0.0.1  127.0.0.1    TCP        74 41842 → 6653 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177844 TSecr=0 WS=512
   2 0.000006994   127.0.0.1  127.0.0.1    TCP        54 6653 → 41842 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
   3 0.002741986   127.0.0.1  127.0.0.1    TCP        74 49878 → 6633 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177844 TSecr=0 WS=512
   4 0.002792113   127.0.0.1  127.0.0.1    TCP        54 6633 → 49878 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
   5 0.039844765   127.0.0.1  127.0.0.1    TCP        74 57936 → 55551 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177854 TSecr=0 WS=512
   6 0.039852803   127.0.0.1  127.0.0.1    TCP        54 55551 → 57936 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
   7 0.140842736   127.0.0.1  127.0.0.1    TCP        74 57938 → 55551 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177879 TSecr=0 WS=512
   8 0.140867374   127.0.0.1  127.0.0.1    TCP        74 55551 → 57938 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177879 TSecr=177879 WS=512
   9 0.140890550   127.0.0.1  127.0.0.1    TCP        66 57938 → 55551 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=177879 TSecr=177879
  10 0.140983265   127.0.0.1  127.0.0.1    TCP        66 57938 → 55551 [FIN, ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=177879 TSecr=177879
  11 0.142476920   127.0.0.1  127.0.0.1    TCP       106 55551 → 57938 [PSH, ACK] Seq=1 Ack=2 Win=44032 Len=40 TSval=177879 TSecr=177879
  12 0.142515547   127.0.0.1  127.0.0.1    TCP        54 57938 → 55551 [RST] Seq=2 Win=0 Len=0
  13 0.143374565   127.0.0.1  127.0.0.1    TCP        74 57940 → 55551 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177880 TSecr=0 WS=512
  14 0.143390780   127.0.0.1  127.0.0.1    TCP        74 55551 → 57940 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177880 TSecr=177880 WS=512
  15 0.143409134   127.0.0.1  127.0.0.1    TCP        66 57940 → 55551 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=177880 TSecr=177880
  16 0.143726452   127.0.0.1  127.0.0.1    TCP       106 55551 → 57940 [PSH, ACK] Seq=1 Ack=1 Win=44032 Len=40 TSval=177880 TSecr=177880
  17 0.143739014   127.0.0.1  127.0.0.1    TCP        66 57940 → 55551 [ACK] Seq=1 Ack=41 Win=44032 Len=0 TSval=177880 TSecr=177880
  18 0.153738779   127.0.0.1  127.0.0.1    TCP        74 58894 → 8181 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177882 TSecr=0 WS=512
  19 0.153768235   127.0.0.1  127.0.0.1    TCP        74 8181 → 58894 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177882 TSecr=177882 WS=512
  20 0.153803556   127.0.0.1  127.0.0.1    TCP        66 58894 → 8181 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=177882 TSecr=177882
  21 0.155546531   127.0.0.1  127.0.0.1    HTTP      557 POST /onos/v1/network/configuration/ HTTP/1.1  (application/json)
  22 0.155562439   127.0.0.1  127.0.0.1    TCP        66 8181 → 58894 [ACK] Seq=1 Ack=492 Win=45056 Len=0 TSval=177883 TSecr=177883
  23 0.216181908   127.0.0.1  127.0.0.1    HTTP     1703 HTTP/1.1 401 Unauthorized  (text/html)
  24 0.216205214   127.0.0.1  127.0.0.1    TCP        66 58894 → 8181 [ACK] Seq=492 Ack=1638 Win=175104 Len=0 TSval=177898 TSecr=177898
  25 0.216636068   127.0.0.1  127.0.0.1    TCP        74 58896 → 8181 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177898 TSecr=0 WS=512
  26 0.216642966   127.0.0.1  127.0.0.1    TCP        74 8181 → 58896 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=177898 TSecr=177898 WS=512
  27 0.216649373   127.0.0.1  127.0.0.1    TCP        66 58896 → 8181 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=177898 TSecr=177898
  28 0.216894672   127.0.0.1  127.0.0.1    HTTP      596 POST /onos/v1/network/configuration/ HTTP/1.1  (application/json)
  29 0.216899321   127.0.0.1  127.0.0.1    TCP        66 8181 → 58896 [ACK] Seq=1 Ack=531 Win=45056 Len=0 TSval=177898 TSecr=177898
  30 0.224003981   127.0.0.1  127.0.0.1    TCP        66 8181 → 58894 [FIN, ACK] Seq=1638 Ack=492 Win=45056 Len=0 TSval=177900 TSecr=177898
  31 0.262972727   127.0.0.1  127.0.0.1    TCP        66 58894 → 8181 [ACK] Seq=492 Ack=1639 Win=175104 Len=0 TSval=177910 TSecr=177900
  32 0.333949931   127.0.0.1  127.0.0.1    HTTP      137 HTTP/1.1 200 OK
  33 0.333993857   127.0.0.1  127.0.0.1    TCP        66 8181 → 58896 [FIN, ACK] Seq=72 Ack=531 Win=45056 Len=0 TSval=177927 TSecr=177898
  34 0.334091031   127.0.0.1  127.0.0.1    TCP        66 58896 → 8181 [ACK] Seq=531 Ack=72 Win=44032 Len=0 TSval=177927 TSecr=177927
  35 0.334313277   127.0.0.1  127.0.0.1    TCP        66 58894 → 8181 [FIN, ACK] Seq=492 Ack=1639 Win=175104 Len=0 TSval=177927 TSecr=177900
  36 0.334316631   127.0.0.1  127.0.0.1    TCP        66 8181 → 58894 [ACK] Seq=1639 Ack=493 Win=45056 Len=0 TSval=177927 TSecr=177927
  37 0.334362489   127.0.0.1  127.0.0.1    TCP        66 58896 → 8181 [FIN, ACK] Seq=531 Ack=73 Win=44032 Len=0 TSval=177927 TSecr=177927
  38 0.334369769   127.0.0.1  127.0.0.1    TCP        66 8181 → 58896 [ACK] Seq=73 Ack=532 Win=45056 Len=0 TSval=177927 TSecr=177927
  39 0.717675708   127.0.0.1  127.0.0.1    TCP        74 57946 → 55551 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=178023 TSecr=0 WS=512
  40 0.717685047   127.0.0.1  127.0.0.1    TCP        74 55551 → 57946 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=178023 TSecr=178023 WS=512
  41 0.717692941   127.0.0.1  127.0.0.1    TCP        66 57946 → 55551 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=178023 TSecr=178023
  42 0.717825587   127.0.0.1  127.0.0.1    TCP       106 55551 → 57946 [PSH, ACK] Seq=1 Ack=1 Win=44032 Len=40 TSval=178023 TSecr=178023
  43 0.717829616   127.0.0.1  127.0.0.1    TCP        66 57946 → 55551 [ACK] Seq=1 Ack=41 Win=44032 Len=0 TSval=178023 TSecr=178023
  44 0.731298221   127.0.0.1  127.0.0.1    TCP       123 57946 → 55551 [PSH, ACK] Seq=1 Ack=41 Win=44032 Len=57 TSval=178027 TSecr=178023
  45 0.731305664   127.0.0.1  127.0.0.1    TCP        66 55551 → 57946 [ACK] Seq=41 Ack=58 Win=44032 Len=0 TSval=178027 TSecr=178027
  46 0.731402045   127.0.0.1  127.0.0.1    TCP        75 55551 → 57946 [PSH, ACK] Seq=41 Ack=58 Win=44032 Len=9 TSval=178027 TSecr=178027
  47 0.733735077   127.0.0.1  127.0.0.1    TCP        79 57946 → 55551 [PSH, ACK] Seq=58 Ack=50 Win=44032 Len=13 TSval=178027 TSecr=178027
  48 0.741397120   127.0.0.1  127.0.0.1    TCP        75 57946 → 55551 [PSH, ACK] Seq=71 Ack=50 Win=44032 Len=9 TSval=178029 TSecr=178027
  49 0.741429512   127.0.0.1  127.0.0.1    TCP        66 55551 → 57946 [ACK] Seq=50 Ack=80 Win=44032 Len=0 TSval=178029 TSecr=178027
  50 0.741466632   127.0.0.1  127.0.0.1    TCP        81 55551 → 57946 [PSH, ACK] Seq=50 Ack=80 Win=44032 Len=15 TSval=178029 TSecr=178027
  51 0.741873939   127.0.0.1  127.0.0.1    TCP        75 57946 → 55551 [PSH, ACK] Seq=80 Ack=65 Win=44032 Len=9 TSval=178029 TSecr=178029
  52 0.769377295   127.0.0.1  127.0.0.1    TCP       227 57946 → 55551 [PSH, ACK] Seq=89 Ack=65 Win=44032 Len=161 TSval=178036 TSecr=178029
  53 0.769412689   127.0.0.1  127.0.0.1    TCP        66 55551 → 57946 [ACK] Seq=65 Ack=250 Win=45056 Len=0 TSval=178036 TSecr=178029
  54 0.769558771   127.0.0.1  127.0.0.1    TCP        81 55551 → 57946 [PSH, ACK] Seq=65 Ack=250 Win=45056 Len=15 TSval=178036 TSecr=178029
  55 0.769680723   127.0.0.1  127.0.0.1    TCP       209 55551 → 57946 [PSH, ACK] Seq=80 Ack=250 Win=45056 Len=143 TSval=178036 TSecr=178029
  56 0.771551120   127.0.0.1  127.0.0.1    TCP        66 57946 → 55551 [ACK] Seq=250 Ack=223 Win=45056 Len=0 TSval=178037 TSecr=178036
```

Figure 9: Beginning of the pcap for one switch.

because mininet first attempts to make an OpenFlow connection. This is an environmental artifact and does not affect our investigation.

The packets with port 55551 are of interest to us as it is the port configured for the gRPC server in mininet. We observe that the gRPC server (switch) is contacted by the client (controller) even before it has received the switch configuration information. Hence, we see the switch respond with FIN packets. Further study is necessary to identify the cause for this behavior.

Next, we observe a network configuration JSON file (`netcfg.json`) exchanged via HTTP to the controller's standard port (8181) for the ONOS web GUI [39]. Due to an unauthorized first attempt, the `netcfg.json` is transmitted twice . The authentication credentials are: "onos:rocks" which are the default credentials for the ONOS web GUI. As

HTTPS is not applied, the credentials are transferred in plain text which is not advisable from a security point of view.

The `netcfg.json` is not part of the P4Runtime protocol. Yet, it belongs to the P4Runtime workflow in ONOS and contains relevant parameters of the switch. I will discuss it in more detail at the end of this section (5.2.3).

Analyzing gRPC Communication At this point in time, the P4Runtime development team included the possibility to parse the human-readable representation of the gRPC messages into the ONOS log. Later on, wireshark introduced the update to partially dissect gRPC header information. Even though the content is still not decodable in wireshark, the messages can be matched to the ones displayed in the ONOS log. The updated VM from August 2018 enabled us to analyze the handshake further. Figure 11 shows the wireshark pcap and Figure 10 shows the beginning of an ONOS log. Only relevant log messages are shown here for brevity.

The first message printed is a *DummyService/SayHello* message. This message type is repeated frequently in the log. Although the specification [18] states that the stream is sufficient to indicate session liveness, the ONOS implementation examined introduces this dummy message to probably detect liveness in a “heart beat” fashion. I will include it in the handshake since it is an integral part of the ONOS implementation. The `.proto` definition of the dummy service, found in the ONOS github [40], defines this message as an unary type with empty request and response.

The *StreamChannel* is opened as described in Section 5.1.1, and it is used to send an arbitration message. This arbitration message should be the *StreamMessageRequest*, but the ONOS log only shows the `election_id`. Neither the `device_id` nor the `role_id` are printed. The switch responds with an answer to the arbitration message. Presumably, this message is the expected *StreamMessageResponse*. Similar to the request, the `election_id` is the only id printed. Additionally, a status message confirms that the controller is indeed the master controller.

After establishing the stream, the controller sends the *SetForwardingPipelineConfig* message containing the P4Info metadata and the device configuration. This message is comparable to the *flow-mod* message of an OpenFlow controller. Therefore, it completes the handshake. When the pipeline is set the next *MasterArbitration* message is already initiated. Both log and pcap messages reflect the messages of the P4Runtime specification. The actual P4Runtime handshake can now be reconstructed for the ONOS implementation. However, at this point we do not know how the controller receives the `device_id` of a switch. To reveal the entire workflow in ONOS, we examine the `netcfg.json` discovered earlier.

```
SENDING GRPC MESSAGE
[device:bmv2:42:p4runtime-ClientKey{deviceId=device:bmv2:42, serverAddr=127.0.0.1,
    serverPort=55551, p4DeviceId=0}]
dummy.DummyService/SayHello:

SENDING GRPC MESSAGE
[device:bmv2:42:p4runtime-ClientKey{deviceId=device:bmv2:42, serverAddr=127.0.0.1,
    serverPort=55551, p4DeviceId=0}]
p4.v1.P4Runtime/StreamChannel:

SENDING GRPC MESSAGE
[device:bmv2:42:p4runtime-ClientKey{deviceId=device:bmv2:42, serverAddr=127.0.0.1,
    serverPort=55551, p4DeviceId=0}]
p4.v1.P4Runtime/StreamChannel:
arbitration {
  election_id {
    low: 1
  }
}

RECEIVED GRPC MESSAGE
[device:bmv2:42:p4runtime-ClientKey{deviceId=device:bmv2:42, serverAddr=127.0.0.1,
    serverPort=55551, p4DeviceId=0}]
p4.v1.P4Runtime/StreamChannel:
arbitration {
  election_id {
    low: 1
  }
  status {
    message: "Is master"
  }
}

SENDING GRPC MESSAGE
[device:bmv2:42:p4runtime-ClientKey{deviceId=device:bmv2:42, serverAddr=127.0.0.1,
    serverPort=55551, p4DeviceId=0}]
p4.v1.P4Runtime/SetForwardingPipelineConfig:
```

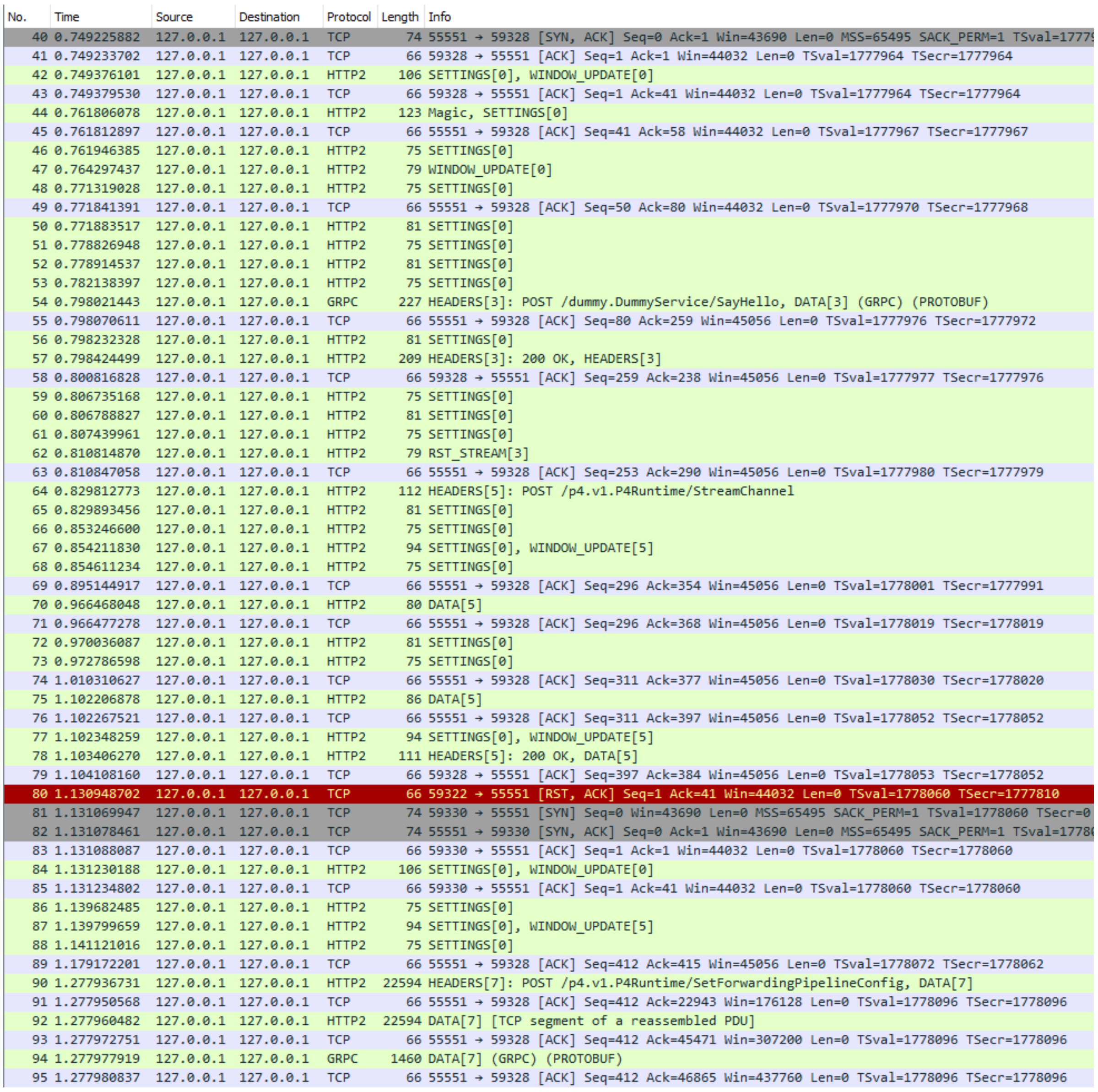
Figure 10: Beginning of a stripped ONOS log with enabled gRPC message log.

5.2.3 Network Configuration in Mininet

The captured `netcfg.json` is shown in Figure 12. Belonging to the key `p4runtime`, there is the localhost IP address, a `deviceKeyId`, the preset gRPC port and a `deviceId`. There are two identification values for a switch according to the key names. The first one is comprised of the string "`p4runtime`" and the key of the device. The second one is a numerical value containing zero.

To determine how the two values are processed by ONOS, we searched for their usage in the ONOS P4Runtime source code [40]. The class `P4RuntimeControllerImpl.java` identifies all the switches via the `deviceKeyId`. However, in the `P4RuntimeClientImpl.java` class, the `deviceId` is used to construct the gRPC messages, e.g., the *SetForwardingPipelineConfigRequest*. Therefore, the `deviceId` is equivalent to the `device_id` introduced in the P4Runtime specification.

I repeated the wireshark capture using two switches to analyze the behavior of the two ids for different devices. The second `netcfg.json` contained `p4runtime:device:bmv2:2`



```
No.  Time         Source     Destination Protocol Length Info
  40 0.749225882  127.0.0.1  127.0.0.1   TCP        74 55551 → 59328 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=1777
  41 0.749233702  127.0.0.1  127.0.0.1   TCP        66 59328 → 55551 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=1777964 TSecr=1777964
  42 0.749376101  127.0.0.1  127.0.0.1   HTTP2     106 SETTINGS[0], WINDOW_UPDATE[0]
  43 0.749379530  127.0.0.1  127.0.0.1   TCP        66 59328 → 55551 [ACK] Seq=1 Ack=41 Win=44032 Len=0 TSval=1777964 TSecr=1777964
  44 0.761806078  127.0.0.1  127.0.0.1   HTTP2     123 Magic, SETTINGS[0]
  45 0.761812897  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=41 Ack=58 Win=44032 Len=0 TSval=1777967 TSecr=1777967
  46 0.761946385  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  47 0.764297437  127.0.0.1  127.0.0.1   HTTP2      79 WINDOW_UPDATE[0]
  48 0.771319028  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  49 0.771841391  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=50 Ack=80 Win=44032 Len=0 TSval=1777970 TSecr=1777968
  50 0.771883517  127.0.0.1  127.0.0.1   HTTP2      81 SETTINGS[0]
  51 0.778826948  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  52 0.778914537  127.0.0.1  127.0.0.1   HTTP2      81 SETTINGS[0]
  53 0.782138397  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  54 0.798021443  127.0.0.1  127.0.0.1   GRPC      227 HEADERS[3]: POST /dummy.DummyService/SayHello, DATA[3] (GRPC) (PROTOBUF)
  55 0.798070611  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=80 Ack=259 Win=45056 Len=0 TSval=1777976 TSecr=1777972
  56 0.798232328  127.0.0.1  127.0.0.1   HTTP2      81 SETTINGS[0]
  57 0.798424499  127.0.0.1  127.0.0.1   HTTP2     209 HEADERS[3]: 200 OK, HEADERS[3]
  58 0.800816828  127.0.0.1  127.0.0.1   TCP        66 59328 → 55551 [ACK] Seq=259 Ack=238 Win=45056 Len=0 TSval=1777977 TSecr=1777976
  59 0.806735168  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  60 0.806788827  127.0.0.1  127.0.0.1   HTTP2      81 SETTINGS[0]
  61 0.807439961  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  62 0.810814870  127.0.0.1  127.0.0.1   HTTP2      79 RST_STREAM[3]
  63 0.810847058  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=253 Ack=290 Win=45056 Len=0 TSval=1777980 TSecr=1777979
  64 0.829812773  127.0.0.1  127.0.0.1   HTTP2     112 HEADERS[5]: POST /p4.v1.P4Runtime/StreamChannel
  65 0.829893456  127.0.0.1  127.0.0.1   HTTP2      81 SETTINGS[0]
  66 0.853246600  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  67 0.854211830  127.0.0.1  127.0.0.1   HTTP2      94 SETTINGS[0], WINDOW_UPDATE[5]
  68 0.854611234  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  69 0.895144917  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=296 Ack=354 Win=45056 Len=0 TSval=1778001 TSecr=1777991
  70 0.966468048  127.0.0.1  127.0.0.1   HTTP2      80 DATA[5]
  71 0.966477278  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=296 Ack=368 Win=45056 Len=0 TSval=1778019 TSecr=1778019
  72 0.970036087  127.0.0.1  127.0.0.1   HTTP2      81 SETTINGS[0]
  73 0.972786598  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  74 1.010310627  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=311 Ack=377 Win=45056 Len=0 TSval=1778030 TSecr=1778020
  75 1.102206878  127.0.0.1  127.0.0.1   HTTP2      86 DATA[5]
  76 1.102267521  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=311 Ack=397 Win=45056 Len=0 TSval=1778052 TSecr=1778052
  77 1.102348259  127.0.0.1  127.0.0.1   HTTP2      94 SETTINGS[0], WINDOW_UPDATE[5]
  78 1.103406270  127.0.0.1  127.0.0.1   HTTP2     111 HEADERS[5]: 200 OK, DATA[5]
  79 1.104108160  127.0.0.1  127.0.0.1   TCP        66 59328 → 55551 [ACK] Seq=397 Ack=384 Win=45056 Len=0 TSval=1778053 TSecr=1778052
  80 1.130948702  127.0.0.1  127.0.0.1   TCP        66 59322 → 55551 [RST, ACK] Seq=1 Ack=41 Win=44032 Len=0 TSval=1778060 TSecr=1777810
  81 1.131069947  127.0.0.1  127.0.0.1   TCP        74 59330 → 55551 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=1778060 TSecr=0
  82 1.131078461  127.0.0.1  127.0.0.1   TCP        74 55551 → 59330 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=1778
  83 1.131088087  127.0.0.1  127.0.0.1   TCP        66 59330 → 55551 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=1778060 TSecr=1778060
  84 1.131230188  127.0.0.1  127.0.0.1   HTTP2     106 SETTINGS[0], WINDOW_UPDATE[0]
  85 1.131234802  127.0.0.1  127.0.0.1   TCP        66 59330 → 55551 [ACK] Seq=1 Ack=41 Win=44032 Len=0 TSval=1778060 TSecr=1778060
  86 1.139682485  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  87 1.139799659  127.0.0.1  127.0.0.1   HTTP2      94 SETTINGS[0], WINDOW_UPDATE[5]
  88 1.141121016  127.0.0.1  127.0.0.1   HTTP2      75 SETTINGS[0]
  89 1.179172201  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=412 Ack=415 Win=45056 Len=0 TSval=1778072 TSecr=1778062
  90 1.277936731  127.0.0.1  127.0.0.1   HTTP2   22594 HEADERS[7]: POST /p4.v1.P4Runtime/SetForwardingPipelineConfig, DATA[7]
  91 1.277950568  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=412 Ack=22943 Win=176128 Len=0 TSval=1778096 TSecr=1778096
  92 1.277960482  127.0.0.1  127.0.0.1   HTTP2   22594 DATA[7] [TCP segment of a reassembled PDU]
  93 1.277972751  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=412 Ack=45471 Win=307200 Len=0 TSval=1778096 TSecr=1778096
  94 1.277977919  127.0.0.1  127.0.0.1   GRPC     1460 DATA[7] (GRPC) (PROTOBUF)
  95 1.277980837  127.0.0.1  127.0.0.1   TCP        66 55551 → 59328 [ACK] Seq=412 Ack=46865 Win=437760 Len=0 TSval=1778096 TSecr=1778096
```

Figure 11: Wireshark capture with HTTP/2 encoding enabled for the gRPC port 55551.

as `deviceKeyId` and again zero as `deviceId`. It seems that the first identification is sufficient for the ONOS implementation to distinguish two devices. In the following, the term `onos_id` refers to the numerical part of the `deviceKeyId`. It is also deducible from the P4Runtime specification that multiple P4 switches can have the same `device_id`. The latter is used to refer to different switching devices on the same P4Runtime switch (e.g., multi-ASCI line card) not the switch itself [18].

Since the `netcfg.json` is not part of P4Runtime, it could be provided by another entity than the switch. As long as the ONOS web GUI credentials are known, anyone can add

```
{
"devices": {
        "device:bmv2:1": {
                "generalprovider": {
                        "p4runtime": {
                                "ip": "127.0.0.1",
                                "deviceKeyId": "p4runtime:device:bmv2:1",
                                "port": 55551,
                                "deviceId": 0
                        },
                        "bmv2-thrift": {
                                "ip": "127.0.0.1",
                                "port": 32906
                        }
                },
                "piPipeconf": {
                        "piPipeconfId": ""
                },
                "ports": {},
                "basic": {
                        "driver": "bmv2"
                }
        }
}}
```

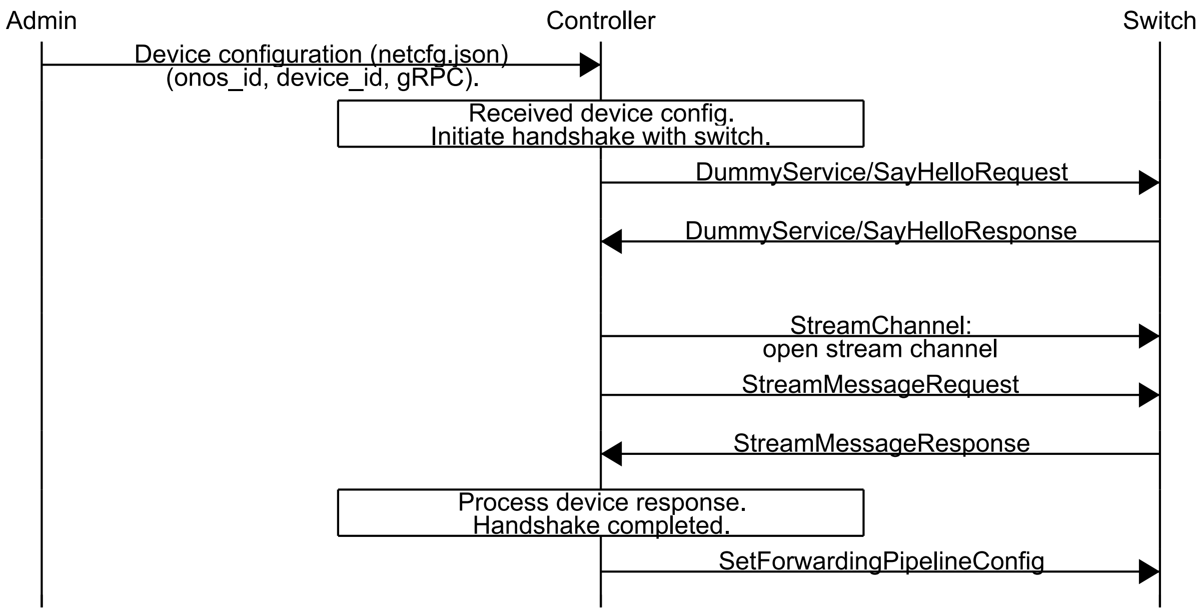
Figure 12: The network configuration sent to the ONOS web GUI.

new configurations to the controller. This is another fundamental difference compared to the way OpenFlow handles new switches. We assume that the `netcfg.json` is provided by a network administrator or by a switch with access to the web GUI. Figure 13 shows the message pattern for the reconstructed P4Runtime handshake implementation in ONOS. We include the `netcfg.json` as it is an integral part of the P4Runtime workflow in ONOS.

The VM provided by ONOS uses default credentials to push device configurations. However, this should not be assumed in general. When using HTTP, the authentication is executed in plain text and the credentials could be potentially eavesdropped. Since an attacker has no control over the topology, intercepting traffic should generally be impossible. Even if we were not able to adjust the examined ONOS version to use TLS, encryption would solve the issues of wiretapping, providing a secure authentication.

5.2.4 Controller-side Switch Authentication

The uncovered handshake reveals a lack of verification if a controller is communicating with the right switch. According to the specification, the switch has to ensure that the `device_id`, `role_id` and `election_id` received in the *StreamMessageRequest* are valid. The controller trusts the switch configuration it obtains, e.g., the `netcfg.json` in ONOS. Thus, the *StreamChannel* is immediately opened without further authentication of the switch.



Admin
Controller
Switch
Device configuration (netcfg.json)
(onos_id, device_id, gRPC).
Received device config.
Initiate handshake with switch.
DummyService/SayHelloRequest
DummyService/SayHelloResponse
StreamChannel:
open stream channel
StreamMessageRequest
StreamMessageResponse
Process device response.
Handshake completed.
SetForwardingPipelineConfig

Figure 13: Message sequence pattern for the P4Runtime Handshake.

Another questionable behavior of the ONOS controller is storing the `netcfg.json` for a second switch, containing an already allocated `onos_id`. We will analyze this behavior in more detail in the following section.

6 P4Runtime Teleportation

In this section, we convey the mechanisms of switch identification Teleportation in OpenFlow to P4Runtime. Therefore, we recapitulate the necessary steps from Section 2.3 and we highlight the differences between P4Runtime and OpenFlow discovered in the previous section. In order to be suitable for Teleportation in P4Runtime, we have to adjust the threat model (Section 3). Finally, we introduce one approach to realize switch identification with the ONOS P4Runtime workflow and present concepts for other approaches in future research.

6.1 Switch Identification Teleportation with P4Runtime

The switch identification Teleportation approach which we want to apply to P4Runtime utilizes the identification of switches to the controller. In OpenFlow two switches could send the same DPID in the *Feature-Reply* message of the handshake. An OpenFlow controller enforces unique DPIDs for switches; therefore information could be transmitted by distinguishing if another switch with DPID=x is already connected or not.

However, as shown in the previous section, P4Runtime switches can not initialize gRPC communication to the controller. The handshake in P4Runtime is initiated by the controller via opening the *StreamChannel*. Therefore, the switch identification Teleportation approach we examine is not directly applicable to P4Runtime. Yet, we can apply the approach to the ONOS implementation. An ONOS controller identifies devices by the `onos_id` received in the `netcfg.json`. If a switch is able to push its own `netcfg.json` to the controller it can force the latter to open a *StreamChannel*.

Altered Threat Model We have to alter the threat model at this place: Switches can not necessarily send messages to the controller at any time. The only connection not initiated by the controller is the HTTP request containing the `netcfg.json`. Therefore, a switch needs to be able to send this device configuration to push its `onos_id` to the controller. Thus, the credentials for the web GUI have to be known by the switch.

The controller is still considered to perform all actions according to their specification. However, switch configurations pushed to the controller are, as already mentioned, trusted. A connection is established as long as parsing the `netcfg.json` does not raise any errors.

6.1.1 Testing Switch Identification Teleportation in ONOS

In this section, we examine the effects of introducing a second switch with the same `onos_id` to the controller. As already shown, the `onos_id` is the unique feature used by ONOS to identify switches. Hence, the `device_id` will be neglected in the subsequent section.

We use the following steps to unveil the controller's behavior when receiving a `netcfg.json` with duplicated entries:

1. Start ONOS, ONOS CLI and wireshark as described in Section 5.2.2.
2. Create first switch with `onos_id`=42 by starting the mininet script.
3. Create second switch after approximately 10 seconds, also with `onos_id`=42.
4. Terminate the first switch after roughly another 10 seconds and wait some time before terminating the wireshark capture.

With the collected data, we can investigate two things: How does the ONOS controller react to a duplicated `onos_id`? Is the stored `netcfg.json` of the second switch used again after the first switch is unreachable? In order to do so, we examine the ONOS CLI, the mininet console, the ONOS log and the wireshark capture.

CLI: In the ONOS CLI only one switch is shown as connected at any time during the experiment. It is impossible to distinguish which one, as the `onos_id`s are equal and the gRPC port is not indicated in the device info.

Mininet: Although the mininet consoles do not show any information whether a switch gets connected or not, they show that both are running with the expected gRPC ports.

Log: The ONOS log does not show any conclusive log entry for the relevant timespan after the configuration was pushed to the controller. Thus, the handshake for the second switch is not initiated at this point. A few seconds after the first switch gets disconnected, the controller completes the entire P4Runtime handshake with `onos_id`=42 to the second switch.

Pcap: The wireshark capture shows that the `netcfg.json` for the second switch is pushed to the controller successfully. After establishing the HTTP connection, some TCP connection to the gRPC port gets initialized. However, there are no gRPC messages visible and all the connections are reset. After stopping the first switch, the pcap shows that the *StreamChannel* breaks. It also confirms that the handshake to the second switch is completed as indicated by the ONOS log.

The controller, not connecting to the second switch, allows for switch identification Teleportation, by distinguishing the connection status of the first switch. When the first switch is connected, the second one does not receive the *DummyService/SayHello* message and no *StreamChannel* is opened. Respectively the handshake is completed if the first switch is not connected. With this differentiation of two states, information can be teleported implicitly.

We go one step further and integrate the uncovered reconnecting behavior into the covert channel. As shown, the controller trusts and stores a `netcfg.json` even if it contains duplicated parameters. Thus, it connects to switches that clearly had invalid configurations at some point. Switches only need to push the duplicated `onos_id` once in order to realize a Teleportation channel. Figure 14 shows the messages involved for switch identification Teleportation in the ONOS P4Runtime implementation. Information is teleported implicitly by comparing the time it takes until the *StreamChannel* is opened to the second switch.

We have shown that our approach of switch identification Teleportation, directly derived from the one in OpenFlow, is indeed feasible in the ONOS implementation of P4Runtime. However, our threat model requires the ability to push the `netcfg.json` to the controller. This requirement is unlikely to be true in a real-world scenario, since it would presumably implicate that the credentials for the controller are known. Thus, an attacker would already have control over the network, probably eliminating the necessity of a Teleportation channel.

6.2 Further research

My thesis was focused on the P4Runtime handshake and on one specific switch identification technique. It would have been worthwhile to examine the P4Runtime handshake further in order to invent novel types of switch identification Teleportation. This was out of the scope of this work but will be examined in future research. We assume that even if the controller initiates all the connection it should still be possible to utilize the ids in the *MasterArbitration* for switch identification Teleportation. One attempt would be to examine the controller's reaction to the different error messages sent by the switch.

Other types of Teleportation could be affected by the differences of OpenFlow and P4Runtime, too. An adjusted threat model may be needed as well. Nevertheless, the concept of Teleportation should still be applicable. Finding a Teleportation technique that is exclusively based on P4Runtime and not on a specific implementation would be a considerable threat for the P4Runtime protocol.

Even if a switch is not able to initiate gRPC connections, it could use the stream channel messages to contact the controller. Thimmaraju et al. [3] introduce other types of Teleportation, e.g., flow (re-)configuration Teleportation, which uses the controller's reaction to *Packet-in* messages. Since a P4Runtime switch is also able to send *Packet-in* messages over the bidirectional *StreamChannel*, flow (re-)configuration Teleportation is likely to succeed.

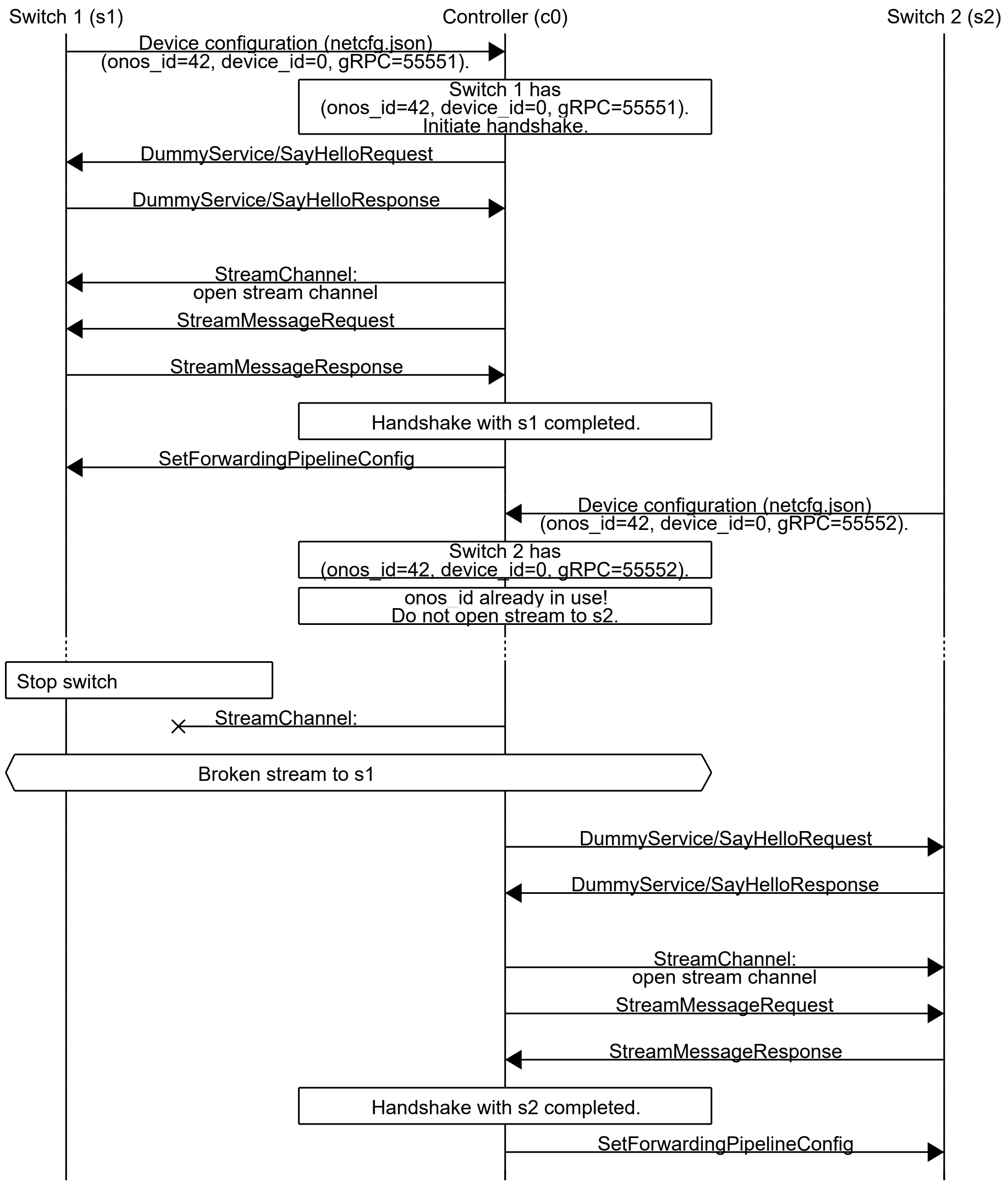

Switch 1 (s1)
Controller (c0)
Switch 2 (s2)
Device configuration (netcfg.json) (onos_id=42, device_id=0, gRPC=55551).
Switch 1 has (onos_id=42, device_id=0, gRPC=55551). Initiate handshake.
DummyService/SayHelloRequest
DummyService/SayHelloResponse
StreamChannel: open stream channel
StreamMessageRequest
StreamMessageResponse
Handshake with s1 completed.
SetForwardingPipelineConfig
Device configuration (netcfg.json) (onos_id=42, device_id=0, gRPC=55552).
Switch 2 has (onos_id=42, device_id=0, gRPC=55552).
onos_id already in use! Do not open stream to s2.
Stop switch
StreamChannel:
Broken stream to s1
DummyService/SayHelloRequest
DummyService/SayHelloResponse
StreamChannel: open stream channel
StreamMessageRequest
StreamMessageResponse
Handshake with s2 completed.
SetForwardingPipelineConfig

Figure 14: Message sequence pattern for a realization of switch identification Teleportation in P4Runtime using reconnecting of devices.

7 Conclusion

In this thesis we studied the P4Runtime handshake from the specification and implementation with ONOS, and finally conducted a first feasibility test of switch identification Teleportation with P4Runtime.

Our analysis of the P4Runtime handshake and our Teleportation test show that unlike in OpenFlow, the controller initiates the handshake with the switch in P4Runtime. This prevents a naive approach of switch identification Teleportation with P4Runtime. However, if we assume that the controller receives the necessary switch configuration, switch identification Teleportation can succeed.

Nonetheless, our security analysis of the P4Runtime handshake uncovered the fact that the current specification as well as the examined ONOS implementation of P4Runtime do not authenticate the switch they connect to. As we have seen a similar lack of authentication with OpenFlow [41], we are in contact with the ONOS security team and the P4Runtime specification team to address the issue. Covert communication remains a threat to this novel SDN approach and should be in the center of future research and investigation in this field.

References

[1] R. Krösche, K. Thimmaraju, L. Schiff, and S. Schmid, "I DPID it My Way! A Covert Timing Channel in Software-Defined Networks," *IFIP Networking 2018*, 2018.

[2] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet Impasse through Virtualization," *IEEE Computer*, vol. 38, no. 4, pp. 34–41, April 2005.

[3] K. Thimmaraju, L. Schiff, and S. Schmid, "Outsmarting Network Security with SDN Teleportation," *Proc. IEEE European Security & Privacy (S&P)*, 2017.

[4] M. Raza, S. Sivakumar, A. Nafarieh, and B. Robertson, "A Comparison of Software Defined Network (SDN) Implementation," *Procedia Computer Science*, vol. 32, pp. 1050–1055, 2014.

[5] N. McKeown and J. Rexford, "Clarifying the differences between P4 and OpenFlow," visited 2018-05-02. [Online]. Available: https://p4.org/p4/clarifying-the-differences-between-p4-and-openflow.html

[6] L. Vicisano and A. Bas, "Announcing P4Runtime," visited 2018-10-11. [Online]. Available: https://p4.org/api/announcing-p4runtime-a-contribution-by-the-p4-api-working-group.html

[7] M. Bishop, *Computer Security: Art and Science.* Addison Wesley, November 2002.

[8] Prof. Claudia Eckert, "IT-Sicherheit," 2014, De Gruyter, 9th Edition.

[9] Open Networking Foundation, "Software-Defined Networking (SDN) Definition," visited 2018-10-01. [Online]. Available: https://www.opennetworking.org/sdn-definition/

[10] Open Networking Foundation, "OpenFlow Specifications," visited 2018-10-21. [Online]. Available: https://www.opennetworking.org/software-defined-standards/specifications/

[11] Kashyap Thimmaraju, "SDN Teleportation: Exploit the OpenFlow Handshake," 2018, visited 2018-11-17. [Online]. Available: https://wiki.onosproject.org/download/attachments/12422167/onos-brigade-workshop-openflowhandshake.pdf?version=1&modificationDate=1526041574410&api=v2

[12] "Open Network Operating System (ONOS)," visited 2018-10-06. [Online]. Available: https://onosproject.org/

[13] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldman, and S. Schmid, "Taking control of SDN-based cloud systems via the data plane," *Proc. ACM Symposium on Software Defined Network Research (SOSR)*, 2018.

[14] B. Snyder, "Snowden: The NSA planted backdoors in Cisco product," *InfoWorld*, May 2017, visited 2018-11-18. [Online]. Available: https://www.infoworld.com/article/2608141/internet-privacy/snowden--the-nsa-planted-backdoors-in-cisco-products.html

[15] T. Yeh, "Netis Routers Leave Wide Open Backdoor," *TrendMicro, Security Intelligence Blog*, August 2014, visited 2018-11-18. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/netis-routers-leave-wide-open-backdoor/

[16] Y. Hu, X. Li, and X. Mountrouidou, "Improving Covert Storage Channel Analysis with SDN and Experimentation on GENI," *National Cyber Summit*, vol. 16, pp. 7–9, 2016.

[17] P4 Language Consortium, "p4.org," visited 2018-04-19. [Online]. Available: https://p4.org

[18] The P4.org API Working Group, *P4Runtime Specification*, 1st ed., October 2018, visited 2018-11-09. [Online]. Available: https://s3-us-west-2.amazonaws.com/p4runtime/docs/v1.0.0-rc3/P4Runtime-Spec.pdf

[19] "P4Runtime GitHub," visited 2018-08-31. [Online]. Available: https://github.com/p4lang/PI

[20] "GRPC A high performance, open source universal RPC framework," visited 2018-05-07. [Online]. Available: https://grpc.io/

[21] "Protobuf github," visited 2018-05-07. [Online]. Available: https://github.com/google/protobuf

[22] "Protocol Buffers, Developer Guide," visited 2018-10-11. [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview

[23] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "p4v: practical verification for programmable data planes," *Proc. 2018 Conference of the ACM Special Interest Group on Data communication (SIGCOMM'18)*, pp. 490–503, August 2018.

[24] G. Kabasele Ndonda and R. Sadre, “A Two-level Intrusion Detection System for Industrial Control System Networks using P4,” *International Symposium for ICS & SCADA Cyber Security Research 2018 (ICS-CSR 2018)*, 2018.

[25] A. A. Agape, M. C. Dănceanu, R. R. Hansen, and S. Schmid, “Exposing Security Issues in P4 Programmable Software Defined Networks,” 2017.

[26] A. A. Agape, M. C. Dănceanu, R. R. Hansen, and S. Schmid, “Charting the Security Landscape of Programmable Dataplanes,” *arXiv:1807.00128*, June 2018.

[27] I. Martinez-Yelmo, J. Alvarez-Horcajo, M. Briso-Montiano, D. Lopez-Pajares, and E. Rojas, “ARP-P4: A Hybrid ARP-Path/P4Runtime Switch,” *Proc. 2018 IEEE 26th International Conference on Network Protocols (ICNP)*, September 2018.

[28] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming Protocol-Independent Packet Processors,” *Proc. 2014 ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 87–95, July 2014.

[29] The P4 Language Consortium, *$P4_{16}$ Language Specification*, 1st ed., May 2017, visited 2018-11-03. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[30] The P4 Language Consortium, *$P4_{14}$ Language Specification*, 1st ed., May 2017, visited 2018-11-03. [Online]. Available: https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

[31] “Protocol Buffers Basics: Python,” visited 2018-10-11. [Online]. Available: https://developers.google.com/protocol-buffers/docs/pythontutorial

[32] “gRPC Concepts,” visited 2018-05-02. [Online]. Available: https://grpc.io/docs/guides/concepts.html#service-definition

[33] “GitHub p4runtime.proto,” visited 2018-10-21. [Online]. Available: https://github.com/p4lang/p4runtime/blob/8b3b71cfc4f6d328cb8c1d790ae07803ba89d258/proto/p4/v1/p4runtime.proto

[34] “Behavioral Model 2 GitHub,” visited 2018-08-31. [Online]. Available: https://github.com/p4lang/behavioral-model

[35] “Mininet,” visited 2018-08-31. [Online]. Available: http://mininet.org/

[36] C. Cascone, “P4 Runtime Support in ONOS,” 2018, visited 2018-08-31. [Online]. Available: https://wiki.onosproject.org/display/ONOS/P4Runtime+support+in+ONOS

[37] "Wireshark," visited 2018-05-07. [Online]. Available: https://www.wireshark.org/

[38] Netsil, "HTTP/2 and gRPC - The Next Generations of Microservices Interaction," visited 2018-10.18. [Online]. Available: https://blog.netsil.com/http-2-and-grpc-the-next-generation-of-microservices-interactions-aff4ffa6faed

[39] "The ONOS Web GUI," visited 2018-08-31. [Online]. Available: https://wiki.onosproject.org/display/ONOS/The+ONOS+Web+GUI

[40] "ONOS GitHub," visited 2018-08-13. [Online]. Available: https://github.com/opennetworkinglab/onos

[41] "OpenFlow-CVE-2018-1000155," May 2018, visited 2018-11-18. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-1000155

[42] V. Gurevich, G. Brebner, J. Tonsing, and B. Pfaff, "P4 Introduction," 2016, visited 2018-09-03. [Online]. Available: https://www.youtube.com/watch?v=U3Mn6o2j4zQ

List of Abbreviations

ACL	Access Control List
API	Application Programming Interface
bmv2	behavioral model version 2
CLI	Command Line Interface
DPID	Datapath ID
gNMI	gRPC Network Management Interface
gRPC	Google Remote Procedure Calls
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
JSON	JavaScript Object Notation
lpm	longest-prefix match
ONOS	Open Network Operating System
pcap	packet capture
protobuf	Protocol Buffers
RPC	Remote Procedure Call
RSA	Rivest Shamir Adleman
SDN	Software-Defined Networking
TCP	Transmission Control Protocol
TLS	Transport Layer Security
VLAN	Virtual Local Area Network
VM	Virtual Machine
XML	Extended Markup Language

Appendix

How to start the test environment

The following steps are used o initialize the setup. You will find a more detailed introduction in the ONOS wiki for P4 Runtime support in ONOS [36].

1. Start up the virtual machine provided and export the necessary applications for ONOS.

   ```
   $ export ONOS_APPS=drivers.bmv2,proxyarp,lldpprovider,hostprovider,fwd
   ```

2. Start ONOS

   ```
   $ cd ~/onos
   /onos$ bazel run onos-local -- clean debug
   ```

3. Access the ONOS CLI from a second terminal and check if the correct applications are installed

   ```
   /onos$ onos localhost
   onos> apps -s -a
   ```

4. Include the decoded gRPC messages into the ONOS log

   ```
   onos> cfg set org.onosproject.grpc.ctl.GrpcControllerImpl enableMessageLog true
   ```

5. Initialize the mininet topology from a third terminal with bmv2 switches and a remote controller

   ```
   $ sudo -E mn --custom $BMV2_MN_PY --switch onosbmv2 --controller remote
   ```

   for the original script with default topology or use the following to start the custom switches.

   ```
   $ sudo -E mn --custom /home/sdn/bmv2_myTopo_switch<number>.py --topo mytopo --
       switch onosbmv2 --controller remote
   ```

6. From the second terminal with the ONOS CLI check if the switch was connected to the controller

   ```
   onos> devices
   ```

Additional P4 information

The following explanation is based on the first language iteration $P4_{14}$ but most of the concepts are analogous to the ones in $P4_{16}$. Keywords and syntax can differ between both versions.

P4 programs work as a pipeline containing four elements:

- A parser converting inbound packets into a representation with access to header fields and implemented protocols.
- A sequence of multiple match and action tables which processes the parsed data.
- A metadata bus transferring information between match and action tables.
- A deparser converting the processed packets back into a serialized packet.

To implement a switch, usually an ingress and an egress pipeline is used with a management unit in between that controls scheduling, queuing and different relations between input and output for tasks like flooding, broadcasting or unicast forwarding.

To generate the four parts of a pipeline, a P4 program consists of three main sections. First, there is a data declaration which defines the data types used in the metadata of the pipeline as well as the data types used for the parsed representation of packets. Data types are declared in “C-type”structures. The next part of the program generates the behavior of the parser and deparser section of the pipeline. It uses an *extract* function to parse packets and it selects the next parsing step. When a parser reaches the control function *ingress* the parsing is completed and the pipeline will continue with the match and action tables. P4 supports a variety of primitive actions like no_op, drop, modify_field, and many others. Thus, a programmer is enabled to write compound actions containing primitive and already implemented actions. All actions can access the packet fields in the deparsed representation or in the metadata fields defined earlier. The table declaration consists of a *reads* section in which the fields that will be possibly matched are listed. A match might be an exact or a so called longest-prefix match (lpm). Furthermore, a list of possible actions is defined for a table. However, one has to be aware that an entry is supposed to have only one associated action from this list. A basic overview about how to write definitions in P4 is shown in Figure 15.

Own Code

In order to strip down the ONOS log, I wrote a script matching entries in the log with the packets in the wireshark capture. The source code can be found at `https://tubcloud.tu-berlin.de/s/GnNK7qenHNfJGAX`.

```
header_type ethernet_t {         // header definition for ethernet
        fields {
                dstAddr          : 18;
                srcAddr          : 18;
                etherType        : 16;
        }
}

header_type ipv4_t { ... }       // header definition for IP version 4
header_type ipv6_t { ... }       // header definition for IP version 6

header_type ingress_metadata_t { //metadata definition
        fields {
                ingress_port     : 9;
                packet_length    : 16;
        }
}

header ethernet_t ethernet;
header ipv4_t ipv4;
header ipv6_t ipv6;
metadata ingress_metadata_t ingress metadata;

parser start {
        extract(ethernet);       // extract the ethernet packet
        return select(latest.etherType){        // select according to therType field
                0x0800           : parse_ipv4;   // parse IP version 4 next
                0x86DD           : parse_ipv6;   // parse IP version 6 next
                default          : ingress;      // done parsing,
        }
}

parser parse_ipv4 { ... }
parser parse_ipv6 { ... }

action route_ipv4(dst_port, dst_mac, src_mac) {
        modify_field(ethernet.dst_addr, dst_mac);
        modify_field(ethernet.src_addr, src_mac);
        ...
}

table ipv4_host {
        reads {
                ipv4.dst_addr    : exact;
        }
        actions{
                route_ipv4;
                drop;
        }
        size : HOST_TABLE_SIZE
}
```

Figure 15: Example definition of headers, metadata and parser in $P4_{14}$ [42]. $\{\ldots\}$ denotes a section where content was cut to improve readability of the code.

The VM containing the my custom mininet scripts is available at `https://tubcloud.tu-berlin.de/s/JpwGZekmigXYz63`.

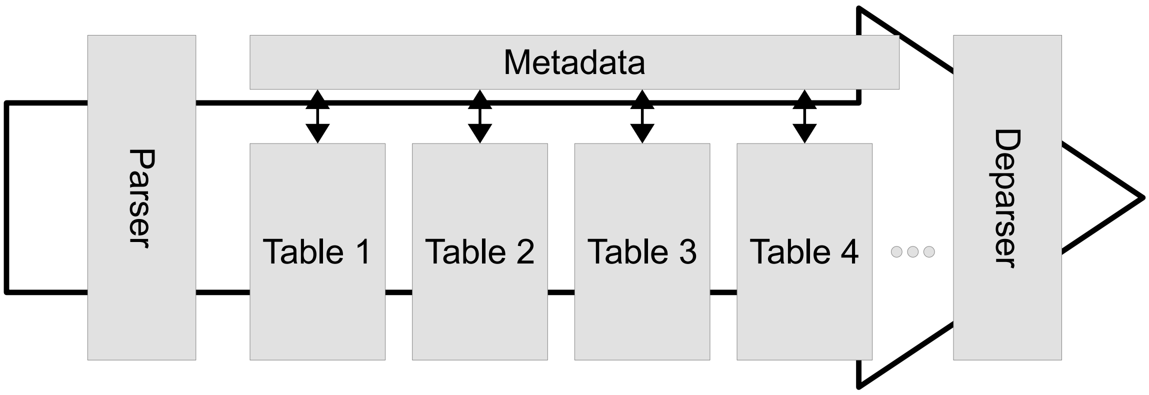


Figure 16: Basic pipeline of a P4 program according to [42]