PhD-FSTC-2014-06
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 25/03/2014 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Stefan HOMMES
Born on 12 January 1981 in Daun (Germany)

FAULT DETECTION AND NETWORK SECURITY IN SOFTWARE-DEFINED NETWORKS WITH OPENFLOW

Dissertation defense committee:

Dr. Thomas Engel, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Radu State
University of Luxembourg, Luxembourg

Dr. Alexander Clemm
Cisco Systems, San Jose, USA

Dr. Ulrich Sorger, Chairman
Professor, University of Luxembourg, Luxembourg

Dr. Thorsten Herfet
Professor, Saarland University, Germany

To Christa and Lia...

Abstract

Due to the rigid architecture of most switches and routers, which provide functionality only for a certain application scenario, the flexibility of deploying new network functions is limited. The advent of programmable networks, which is described as *Software-Defined Networking (SDN)*, allows the extension and control of networks based on a *flexible control plane*, which is based on software and acts as a network operating system with network applications running on top of it.
In this thesis we focussed on SDN based on the concept of the *OpenFlow* protocol. In order to deploy such networks in operational environments and datacentres, the challenges concerning *network management* are still lacking a sufficient analysis and are further investigated in this thesis, which examines the reliability and maintainability of SDN, as well as new security issues that are introduced with this architecture. The second contribution of this thesis is to provide solutions to some of the addressed challenges, with a focus on fault detection and network security.
With regard to fault detection, we discuss the information content and monitoring aspects of flow entries that are located on the network devices, but are managed from the network controller. This involves applying methods from information theory to determine faults and attacks by observing the logical topology, and correlation facilities to determine errors that relate to the data plane.
In network security, current approaches mostly rely on security appliances that are deployed at different locations in the network. We analyse the extend to which SDN can be leveraged to provide new ways of thwarting network attacks, and investigate the possibilities for controller-based packet inspection to detect malicious communications in the network. This includes the extraction of hidden communication patterns originating from a stealthy backdoor.
The freedom of extending controller software to meet new network service requirements comes at a high cost. Since the reliability of the network must be assured, tools are required to debug and test the software after each alteration step. We propose a solution that instruments network applications with additional code for logging purposes, guaranteeing certain correctness properties. In combination with a database system, our framework can be leveraged to allow network debugging or anomaly detection.

Contents

Acknowledgments

After writing this thesis and my research at the University of Luxembourg, I want to express my gratitude for people that supported me during this period in different ways. First of all, I want to thank Professor Dr. Thomas Engel for his generous support and for giving me the opportunity to be a member of his research group. His constructive and positive remarks have been inspiring and created a pleasant working atmosphere. Next I would like to thank Dr. Radu State for his support and the interesting discussions we have had. I am also grateful for his trust in me in finding my way through the whole process of my PhD. Thirdly, I want to thank Dr. Alexander Clemm for his advice based on industry insights, which added important contributions to my work. I also thank Professor Dr. Ulrich Sorger and Professor Dr. Thorsten Herfet for participating in the jury of my dissertation defence.
Special thanks go to Dr. Christian Franck and Dr. Andreas Zinnen, for their valuable advice and discussion concerning machine learning techniques and implementation. I also owe a great debt to Dominic Dunlop for proofreading my thesis and papers. Last but not least, I want to thank my colleges and collaborators in the SECAN-LAB team for providing an environment that was enjoyable to be a part of.
Finally and most of all, I want to express my gratitude to my parents and family for their support and understanding during this time. This goes especially to my father, who is no longer with us. I am deeply grateful to my lovely wife Christa and my daughter Lia, for being so supportive of my work and patient.

Chapter 1

Introduction

With the advent of telecommunication technologies and distributed computing, computer networks acting as an interface between them have become more and more significant in today's world. The technologies involved allow data transport through heterogeneous networks from a sender to a receiver, with the process of packet forwarding being unimportant to the end user. The increase of new application scenarios such as home offices and cloud computing means that computer networks have emerged as one of the new infrastructures that are indispensable for the private and industrial sector. As a result, the requirements of such networks with regards to flexibility and configuration have greatly increased.

The functionality of computer networks was until recently defined by the network devices that they contained, which had the specifications set by their vendors. While this was acceptable when computer networks were mainly used to give access to mail services and web sites, recent technologies require a greater degree of configuration. For instance, a cloud provider needs to be able to dynamically allocate resources for an enterprise, requiring reconfiguration of the network. While some restrictions concerning configuration could be resolved by network virtualisation solutions (e.g. multiple switches in one physical switch), the natural evolution of networks requires network programmability based on software.

With the introduction of *Software-Defined Networking (SDN)*, computer networks are defined by software applications in the same way that personal computers are defined by their operating system and installed programs. For instance, a network service can define a routing component, which communicates with network devices over a control channel. Similarly, a different interface might allow business applications to communicate with network services in order to configure the network to the requirements of the customer.

One of the first architectures that emerged around SDN was centred on the concept of the *OpenFlow* protocol [1], which was developed by a research group in Standford,

and is based on the previous work by Martin Casado on Ethane [2] and SANE [3]. It specifies a clear separation of *control* and *data* planes. While the control plane is moved to a centralised network controller, the data plane remains on the network devices to take advantage of their specialised hardware, which allows packet forwarding at wire speed. The communication between control plane and data plane is realised through the OpenFlow protocol. It handles packet streams on the basis of *flows*, and uses control messages in order to manage flows on network devices.

While this freedom allows new application scenarios, *network management* in software-driven networking is a largely unexplored territory. Because the initial proposals in SDN did not consider practical problems that occur when SDN is applied in operational environments, this thesis will shed some light on the topic. In accordance to the ISO *Telecommunications Management Network model (FCAPS)*, network management can be separated into fault, configuration, accounting, performance, and security management. Our work focusses primarily on the fault and security management aspects of OpenFlow-based SDN.

1.1 Problem Statement and Objectives

This thesis focusses on the challenges in network management that need to be tackled when confronted with SDN. We base our analyses on the following two assumptions:

- Which new challenges must be tackled in Software-Defined Networking (SDN) compared to traditional network design?
- What are the advantages of a programmable network as regards network management?

More precisely, we focus on the implications of OpenFlow-based SDN for the following categories:

Fault Management The current solutions in fault management relate to techniques that were developed to cope with a traditional network design. While some of these prove also to be applicable in SDN, others must be replaced by new methods. Because the SDN architecture that is based on a programmable control plane, it is possible to create network monitoring applications that collect information and make decisions based on a network-wide view. This enables centralised event correlation on the network controller, and allows new ways of mitigating network faults.

Network Security The concept of a centralised control plane, together with the control channel which is used to exchange information with network devices, introduces new security issues that need to be characterised. From an attacker's perspective, the network controller is attractive due to its important role, and so requires specific protection mechanisms. This is an example of a case where network security solutions are more an aspect

of the application and less dependant on specialised hardware solutions. The scope of such concepts, and of concrete application scenarios, still lack a complete understanding in the research community.

1.2 Outline of this Thesis

The thesis is organised as follows:

- In chapter 2, we give an introduction to the concept of programmable networks, and highlight the basic principles of an OpenFlow-based SDN. This is followed by a summary of the current state of research in areas that are closely related to our work.
- In chapter 3, we address the challenges and missing pieces that are required for the deployment of SDN in operational networks. These comprise fault, configuration and security aspects.
- In chapter 4, we describe the concept of network monitoring based on network flows and a centralised network controller. We discuss how variations in the logical topology can be detected through the application of information theory. Based on a real OpenFlow testbed, we discuss the detection of a Denial-of-Service (DoS) attack and the implications for the network devices and network controller.
- In chapter 5, the potential of controller-based packet inspection is shown for two application scenarios. Firstly, we analyse communication data on the flow level in order to identify network attacks in both on-line and off-line modes. Secondly, the knocking sequence used to enable a stealthy backdoor is discovered by using machine learning techniques.
- In chapter 6, we describe the concept of a centralised logging solution for flow-based communication data. In combination with a database system, the flow records can be used for network debugging and root-cause analysis of network faults. In order to modify the source code to provide such functionality, we leverage a technique for automated source code extension that guarantees specific correctness properties.

Chapter 2

Background and State of the Art

In this chapter, we present an introduction to the concept of Software-Defined Networking. This includes the history of programmable networks and an overview of related standards that have emerged recently. One of these, the OpenFlow protocol, has played a large part in promoting the concept of SDN, resulting in a significant impact on both research and industry. We discuss the basic principles of OpenFlow, and present the current state of research that is closely related to our work.

2.1 Definition

In the literature and within industry, the concept of programmable networks or Software-Defined Networking (SDN) is still evolving, with different developers having different visions of its scope. The Open Network Foundation (ONF) [4], a user-driven organisation with the goal of fostering the development of SDN, describes it as:

> "The physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices."

The term *programmability* relates to the flexibility of the control plane, which is directly programmable in software and includes a control channel to exchange data with network devices. An application interface (API) allows applications to communicate with the control plane in order to access and modify network services. The consequence of this new design principle is increased flexibility in network management. By analogy to mobile phones, this evolution can be compared with the shift from phones with restricted functions to the flexibility of smartphones with an operating system and the ability to install various applications. The concept of traditional computer networks was based on network devices which contain both data and control planes. In order to distinguish these from the SDN paradigm, we refer to them as *non-programmable networking (non-SDN).*

2.2 Historical Development

The paradigm of programmable networks pre-dates to the work of [5, 6, 7], and was vigorously pursued in the mid-1990s. Research efforts centred around programmable networks, and were divided between two schools of thought [8]. The first school, which can be described by the term *active networking (AN)*, constituted of a large group of projects and was founded by DARPA[1]. The general approach is characterised by the very flexible and dynamic manner in which services within the network can be deployed [9, 10]. For instance, a single packet can be used to modify network services to allow a maximum of flexibility in network configuration.

In contrast, the *Open Signalling (OpenSig)* communities proposed the idea of having standardised interfaces to allow the communication with network devices, and were pursued by the IEEE P 1520 initiative [11]. Routers and switches have open access Application Programming Interfaces (APIs) in order to allow third parties to develop software for them. Despite research in the OpenSig community centring around programmable networks [12, 13, 14], the industry did not adapt to the idea of programmable networks with a standardized API. This resulted in the landscape of proprietary protocols and closed-box devices that characterises modern networks.

2.3 Architectures and Interfaces

A variety of architectures from both industry and research communities have evolved in parallel since the inception of SDN. Those from industry are mainly proprietary and based on their own understanding of SDN, although some have been standardised or at least discussed in working groups of the Internet Engineering Task Force (IETF[2]). The open-source community centres mainly on the Open Network Foundation (ONF).

In order to accelerate innovations in a vendor-neutral manner and to determine standards for SDN, the Linux Foundation has established the **OpenDaylight (ODL)**[3] collaborative project. This open source framework is currently supported by 29 members from industry, for instance Cisco, IBM, Juniper and NEC. The ODL project is based on existing interfaces and protocols (e.g. OpenFlow and NetConf [15]), and can currently be considered as the most promising step towards the development of SDN.

In the following we present the different approaches that have emerged around SDN, followed by a timeline in Figure 2.2 and comparison in Table 2.1.

[1]Defense Advanced Research Projects Agency (DARPA): The military development and research agency of the United States Department of Defense.

[2]The IETF is an organisation that focusses on the development of new internet techniques and standards.

[3]http://www.opendaylight.org/

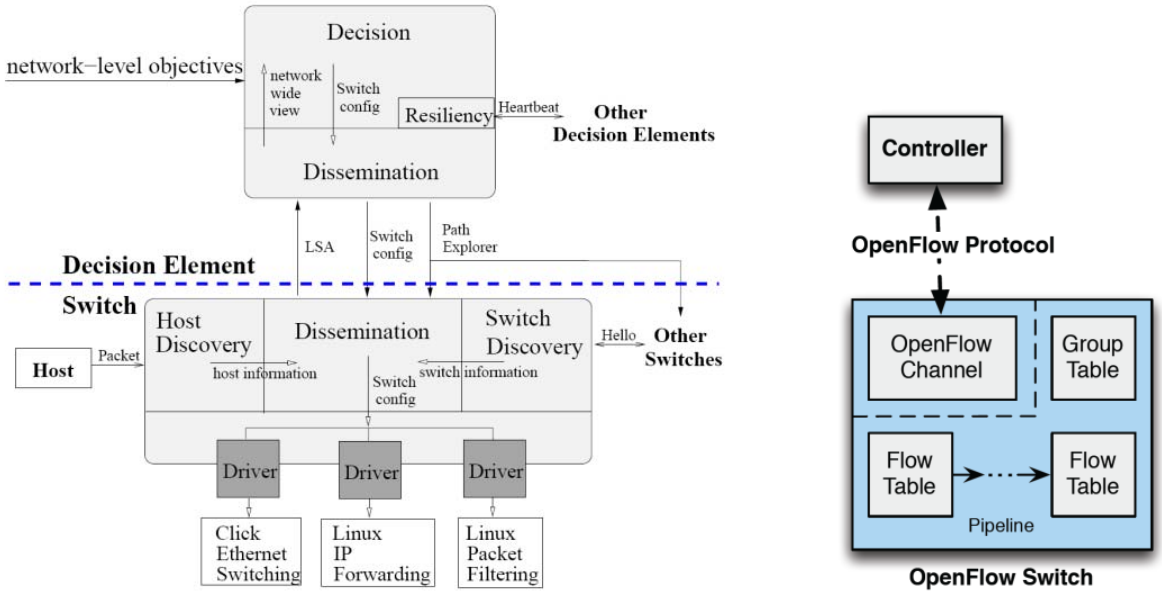


Figure 2.1: The architecture of a) Tesseract and b) OpenFlow. Whereas in OpenFlow most of the control functionality is centralised, Tessteract places more intelligence in the switch.

Routing Control Platform (RCP) The work of RCP [16] was fostered by AT&T Labs in order to overcome limitations of path selection in the Internet. Each RCP is assigned to an Autonomous System (AS), with the Inter-AS protocol exchanging routing information between them. Within an RCP, Border Gateway Protocol (BGP) is used as the control protocol to communicate with the routers. These contain a data plane based on IP forwarding tables, which provide only a small subset of the match fields in OpenFlow.

Tesseract The design of Tesseract [17] is based on a 4D architecture [18], that incorporates *decision*, *dissemination*, *discovery* and *data* planes as shown in Figure 2.1 (a). While the decision plane acts more as a network controller with a network-wide view, the dissemination plane connects the decision plane with the network devices in order to exchange information such as switch configuration, path exploration and Link State Advertisement (LSA). The switches incorporate a discovery plane in order to detect hosts and switches, and forward packets based on the data plane. Tesseract takes a clean-slate approach and allows direct control of a computer network by separating the decision logic (e.g. route computation) from the underlying protocols. This approach can also be leveraged to deploy centralised control policies in the network. Compared to the OpenFlow architecture shown in Figure 2.1 (b), the network devices contain only the flow table and simplify the requirements for *commodity hardware*.

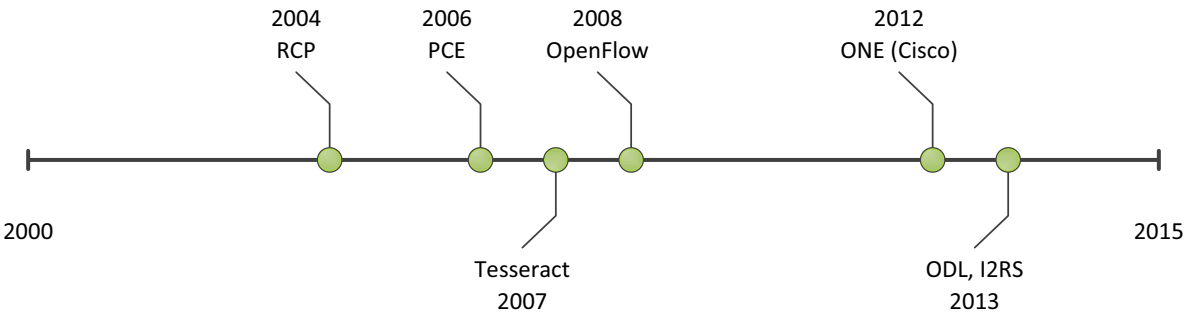


Figure 2.2: Timeline of the historical development of SDN.

Forwarding and Control Element Separation (ForCES) The goal of the ForCES working group (RFC 5810 [19]) is to develop a protocol for Software-Driven Networking. It shows some similarities to the OpenFlow protocol, but differs in the way that the data plane is structured. Whereas OpenFlow is based on flow tables, ForCES uses *LFBs* (Logical Functional Blocks) in order to set data paths and to create packet flows.

ONF and OpenFlow The advent of SDN was greatly facilitated by the development of OpenFlow [1], an open protocol that is used to communicate between a centralised network controller and network devices. It was initiated by the work of Martin Casado in **Ethane** [2], **SANE** [3] and the Clean Slate Project[4] at Stanford University in 2008. OpenFlow was successfully deployed at their campus network using commodity hardware switches, containing only the data plane, with the control plane moved to the network controller. The specification of OpenFlow is defined and developed by the ONF[5].

Path Computation Element (PCE) A PCE is a software component that is used to calculate a path through a Multiprotocol Label Switching (MPLS) network. This approach results in network devices in which only the path computation component is removed from the control plane so that it can be centralised in the network. While PCE provides less functionality and flexibility than OpenFlow, its deployment in Wide Area Networks (WANs) is less difficult for operators, since only the edge routers need to be modified [20]. PCE has been standardized by the IETF, for instance in RFC 4655 [21].

Interface to the Routing System (I2RS) I2RS [22] is a working group at the IETF, which has support from a number of vendors. It keeps both data and control planes in the network devices, but proposes an interface which allows communication with control planes that can be located outside the local network. This allows interaction with routing protocols from an external software-driven instance.

[4]http://cleanslate.stanford.edu/
[5]https://www.opennetworking.org/

Architecture	Interfaces	Centralisation	Standardisation	Industry support
RCP	BGP, Inter-AS	Medium	-	-
Tesseract	Multiple	Medium	-	-
-	ForCES	High	RFC 5810	-
ODL	Multiple	Medium	-	All major vendors
ONF	OpenFlow	High	-	All major vendors
PCE	PCEP	Low	RFC 4655	-
-	I2RS	Low	IETF Draft	All major vendors
ONE	onePK	Low	Proprietary	Cisco

Table 2.1: Architectures and interfaces for SDN.

Open Network Environment (ONE) The management of SDN requires several APIs, for instance one providing a control channel to the network devices while a second manages data exchange between network services and external applications. Cisco's ONE [23] architecture and the One Platform Kit (OnePK) interface integrate these different APIs into a single solution. It allows the deployment of network applications not only on a dedicated network controller as proposed in OpenFlow, but also on the network devices themselves. This precludes the use of commodity hardware, but allows for a more flexible partition of functionality between the data and control planes.

Apart from the described architectures, the general design of SDN is still evolving. For instance, the **Fabric** [24] approach combines network programmability with the insights of MPLS in order to enhance SDN. Packets that enter or leave a network are processed by an ingress/egress edge switch, which is controlled by an edge controller and can provide network services such as security, filtering and isolation. The task of packet forwarding across a network is realized over "network fabrics". This separation allows both fabric and edge switches to evolve separately without influencing each other.

A term that is closely related to SDN is *network virtualization*, which proposes emulating hardware in software to provide a solution, for instance to define multiple virtual switches. These can be realised, for example, with Open vSwitch[6], which is based on open-source software and widely used in operational scenarios. The concept of network virtualisation can also be used as a basis for the development of SDN solutions. This has attracted increased interest since the company Nicira, which was founded by an initiator

[6]http://openvswitch.org/

of the SDN movement, was acquired by VMware in 2012. VMware NSX, a network virtualisation platform, can be considered as a product that merges network virtualisation with the ideas behind SDN.

2.4 OpenFlow-based SDN

This thesis focuses on the OpenFlow-based SDN paradigm. This seems reasonable, since most ongoing research and open-source software is centred around it. As a result, it is increasingly beeing deployed in production environments. For instance, Google has used the concept of an OpenFlow-based SDN for their internal backbone network called *G-Scale* [25], which is a major international network. The experience of Google was very promising, allowing a rapid rich feature deployment and simplified network management. In industry, most vendors of network equipment provide network devices with OpenFlow support, for instance Cisco, Hewlett Packard [26], Juniper [27], and NEC. The architecture of SDN with OpenFlow is shown in Figure 2.3 and comprises three layers.

Infrastructure Layer This layer is used for packet forwarding based on OpenFlow-supported network devices. Specialised hardware, including memory to store flow-related information, is used for this purpose, allowing rapid packet processing. Typical network devices that belong in this layer are *switches* and *routers*. A switch typically operates at the ISO[7] link layer (L2), whereas a router can also process network layer (L3) information, which allows packet forwarding beyond Local Area Networks (LANs). Since the functionality of an OpenFlow-based SDN is defined by a software-driven network controller, we refer to all network devices as switches for the sake of simplicity.

Control Layer The network controller, often referred to as the *network operating system*, comprises the control plane and has a centralised view of the network. This can be leveraged to perform forwarding decisions that are based on the status of all connected network devices. The network controller communicates with network devices through the OpenFlow *southbound* interface. A second type of API, the *northbound* interface, is used to exchange data between the network services and business applications. Typical examples of network services include routing, packet switching (L2) and MPLS. In addition, multiple network controllers can be deployed in a network for different reasons, for instance to avoid a single point of failure or to separate tasks.
Inspired by the first open-source network controller, NOX [28], written in C++, research institutions and companies have developed network controllers for a wide range of purposes. In the open-source community, typical controllers are POX [29] (Python), Beacon[8] (Java) and Floodlight[9] (Java). Latter is supported by the company Big Switch Networks, who market also a commercial network controller, the Big Network Controller.

[7] OSI model: ISO/IEC 7498-1:1994
[8] https://openflow.stanford.edu/display/Beacon/Home
[9] http://www.projectfloodlight.org/floodlight/

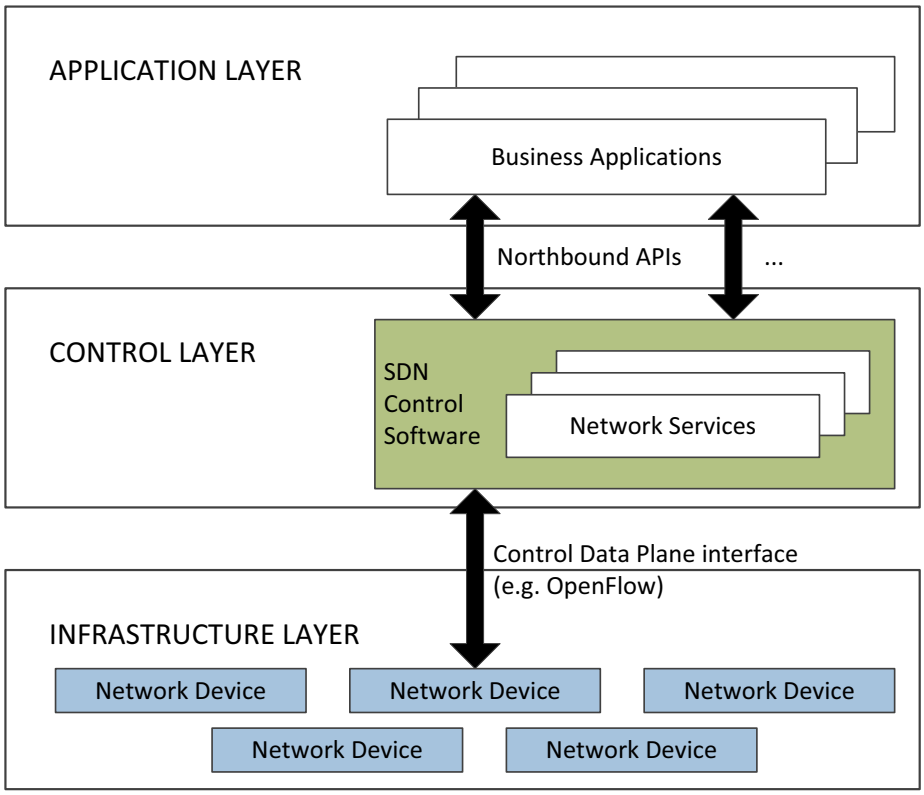


Figure 2.3: Three-layer design of a Software-Defined Architecture (Source: ONF [32]).

Application Layer The application layer allows business applications to modify and influence the way the network behaves in order to provide network services to customers. This requires the definition of an API, to allow third-party developers to build and sell network applications to the network operator. The development of such an API has not yet been addressed by the ONF, but is required in order to guarantee interoperability between a business application and network controllers from different suppliers.
Frenetic [30] is a network programming language that can be used to define such high-level policies. It provides a high-level abstraction of network functions in order to hide low-level packet processing from the programmer. The project has recently proposed a new Python-based language called Pyretic [31]. This provides an abstract packet model, and an algebra of high-level policies and network objects.

2.5 The OpenFlow Protocol

Interest in SDN has increased greatly since the development of the OpenFlow protocol, currently at version 1.4.0 and managed by the ONF [33]. While the core functionality was completely implemented in version 1.0, more recent developments include support for IPv6 and MPLS.

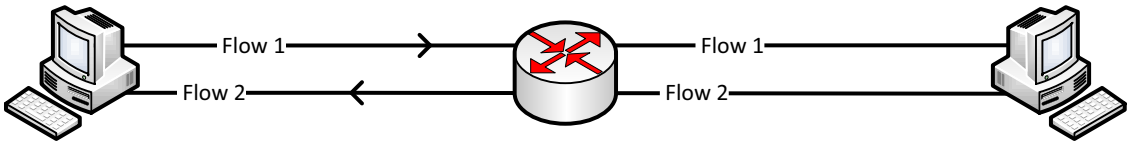


Figure 2.4: Flow-based communication between two participants.

The basic principles of the OpenFlow protocol for packet forwarding are explained in the following.

2.5.1 Network Flows

The OpenFlow protocol is based on the concept of *network flows* to monitor and control traffic within a network. Network flows improve scalability by increasing network speeds and are currently supported by flow-enabled devices from all major vendors. There are several definitions of the term *flow* in the literature. The IP Flow Information eXport (IPFIX) working group within the IETF [34, 35] describes an *IP flow* in this manner:

> "A flow is defined as a set of IP packets passing an observation point in the network during a certain time interval. All packets belonging to a particular flow have a set of common properties."

For instance, a communication between two entities result in two flows as shown in Figure 2.4. The typical properties of a flow include source and destination IP address, source and destination port, and the type of protocol (TCP, UDP, ICMP, ...).

2.5.2 Flow Table

In order to forward packets based on the concept of flows, each OpenFlow-based switch contains one or several *flow tables* used to store pre-defined *flow entries/rules*. For instance, the Media Access Control (MAC) address can be used to identify hosts in a switched network as shown in Figure 2.5. Each flow entry contains the following components:

```
Match Fields | Priority | Counters | Instructions | Timeouts | Cookie
```

For all incoming packets that arrive on a switch, a flow table look up is performed by matching the packet header against the *Match Fields* of all flow entries. Each OpenFlow-based switch supports the match fields shown in Table 2.2, which can either contain a specific value or wildcard (*).

Match Fields
Ingress Port
Ethernet source address
Ethernet destination address
Ethernet type
IPv4 or IPv6 protocol number
IPv4 source address
IPv4 destination address
IPv6 source address
IPv6 destination address
TCP source port
TCP destination port
UDP source port
UDP destination port

Table 2.2: Supported match fields for an OpenFlow-enabled switch (version 1.3.1).

In dependance of the OpenFlow version and network devices, additional match fields might be available, for instance for Virtual Local Area Network (VLAN), MPLS or ARP. Flow entries that contain wildcarded fields are known as *macroflows*, while flows without wildcards and a finer granularity are *microflows*. This concept allows the range of packets that can be matched to a particular flow entry to be defined. If a packet can be matched to several flow entries, the entry with the most precise match is chosen (*Priority*).

On finding a match, the associated *Instructions* are applied to the packet. Typical actions are *forward*, *drop*, *modify* and *enqueue*. While the last of these will also forwards the packet, it is put in a queue where the forwarding behaviour can be configured, for instance to account for Quality of Service (QoS) requirements. As well as specifying packet forwarding actions, instructions can also *modify* the packet header (e.g., to push an MPLS header), if the function is supported by the device.

The lifetime of a flow entry in a flow table is specified by the flow expiration time (*Timeouts*), which is further defined by an *idle* timeout and *hard* timeout. The idle timeout specifies the time after which a flow entry will be removed when it has matched no packets, while the hard timeout defines when a flow entry will be removed regardless of the number of packets it has matched.

The respective statistics, which are defined in the *Counters* field, contain the number of received packets/bytes, the flow duration (seconds and nanoseconds), and the reason for removal. In addition to the counters that are defined per flow, there are counters per flow table (e.g. number of active entries), per port and queue. The usage of the *Cookie* field is not defined, but can be used by the controller for various purposes, for instance

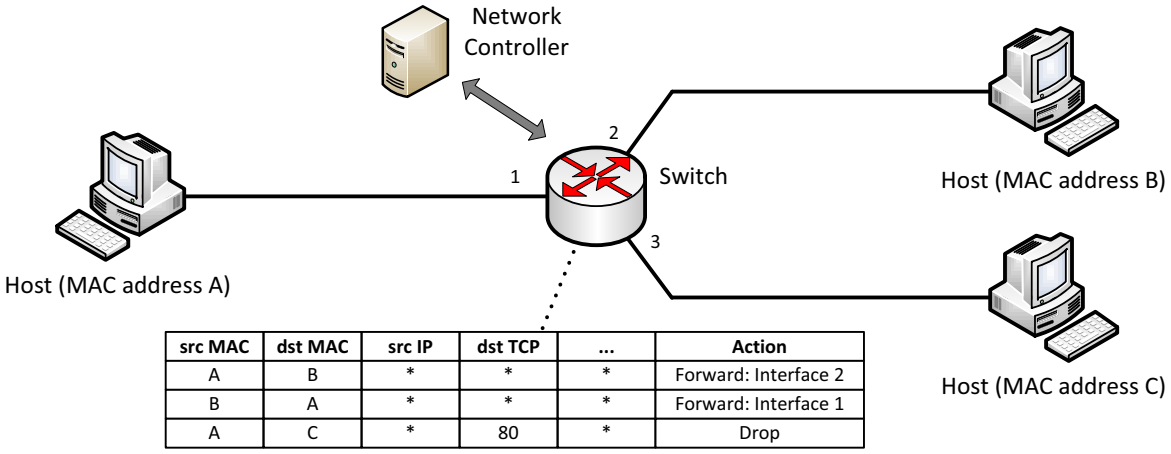


Figure 2.5: OpenFlow network with an instruction set consisting of three flow entries. While packets between host A and host B are forwarded, web traffic from host A to host C is dropped.

as a flag or identifier.

OpenFlow-based switches can be classified by whether they operate in an *OpenFlow-native* or in *hybrid* mode. Whereas a native mode switch can be deployed only in wholely OpenFlow-based scenarios, a hybrid switch contains an additional control plane and allows a non-SDN operating mode. This is leveraged through additional flow tables, where each switch port can operate either in hybrid or native mode. Such a network device can be considered as a combination of two switches in one, and increases flexibility and possible deployment scenarios for network operators.

2.5.3 Flow Installation

The OpenFlow specification defines control messages, which exchange information between network devices and the network controller. The communication type can either be *Controller-to-switch*, *asynchronous* or *symmetric*. For instance, `Hello` messages (symmetric) are exchanged upon connection start-up between the switch and controller, in order to inform each other about the highest supported OpenFlow version. To determine if the network controller or network devices are reachable in the network, an `Echo` request can be sent whereas the sender must return an `Echo` reply (symmetric).

Apart from such status messages, the process of flow installation can be considered to be the core function of the OpenFlow protocol. Each new packet that arrives on the switch is mapped against the flow entries in the flow table. If there is a match, the corresponding action is executed. If no match is found, the switch sends a `Packet-in` message (asynchronous) containing a copy of the packet[10] to the controller. Depending

[10]The first 128 bytes by default.

on the controller program and an analysis of the packet header fields, the controller can install a corresponding flow entry in the flow table of the switch by sending a `FlowMod` message (Modify-State). Thereafter, all subsequent packets that belong to the flow entry are forwarded on the switch without controller intervention until the flow entry expires. The `FlowMod` message also allows flow/group entries to be modified and deleted, switch port properties to be set, and the setting of a flag for a flow entry, enabling the switch to acknowledge each flow removal by sending a `FlowRemoved` message containing flow statistics (Counters).

In order to build a `FlowMod` message, the following code snippet from the submodule `l2_learning.py` from the POX [29] network controller presents an example of the required commands:

```
...
171| msg = of.ofp_flow_mod()
172| msg.match = of.ofp_match.from_packet(packet, event.port)
173| msg.idle_timeout = 10 # seconds
174| msg.hard_timeout = 30 # seconds
175| msg.actions.append(of.ofp_action_output(port = port))
176| msg.data = event.ofp
177| self.connection.send(msg)
...
```

In line 172, the header fields from the packet that is encapsulated in the `PacketIn` message are copied to the match fields of the new flow entry to be installed. Later lines set flow entry specific parameters, including the interface over which the packet is forwarded on the switch (line 175). In line 177, the `FlowMod` message is sent to the controller.

2.5.4 Application Scenarios and Limitations

The initial concept of OpenFlow envisaged a single network controller. This is preferable in LANs, bringing the advantages of speed and easier management. A multiple-controller mode allows the assignment of network segments or tasks to different network controllers, which has advantages for scalability and reliability. In order to avoid conflicts in a multi-controller environment, FlowVisor [36], a special controller is used to communicate with the network devices and delegates network *slices* to each network controller. For instance, one network can be used for both operational and research traffic without either influencing the other. The current design of OpenFlow does not address such an applications scenario in WANs, and requires further development. A first step in this direction is **DISCO** [37], which enables communication between multiple SDN controllers in a WAN topology.

One of the most important tasks of a network controller is to update the flow tables on all switches in the network. This can be achieved by choosing between two flow installation concepts. In the *reactive* approach, a flow entry is only installed as a response to a respective `Packet-in` message. This keeps the flow table overhead low, since the number of stored flow entries is always adapted to the current traffic. Nevertheless, the first packet of each flow needs to be buffered in the switch until the controller has replied with a corresponding action. This introduces a forwarding delay that is not tolerable for some applications, for instance in high speed trading. However, such a delay is inherent in the conceptual design of OpenFlow, which requires the involvement of the network controller in the flow installation process.

In a *proactive* mode, the controller populates the flow entries ahead of time for all traffic that might be forwarded by the switch. This results in a low latency, since no `Packet-in` messages need to be sent to the network controller. But it requires that the network operator can predict upcoming traffic, and might result in large flow tables when confronted with a large network. In order to reduce the number of flow table entries in this case, the concept of macroflows can be useful. Since the different levels of granularity concerning flow entries are a major advantage of OpenFlow, the proactive mode maybe seen as limiting this potential.

2.6 State of the Art: SDN with OpenFlow

The emerging concept of Software-Defined Networking (SDN) has attracted considerable interest following the development of OpenFlow [1], which is the basis for most ongoing research in the area, and is also supported in commercial switches. In the following section we highlight related work that is centred around network management with OpenFlow.

2.6.1 Fault Detection and Mitigation

The failure of a link in a network can result in disconnected hosts or network separation. For network devices based on OpenFlow, a broken link can result in a break in the control channel used to communicate with the network controller. For *in-band* control, the control traffic is sent over the same network as the transport data traffic, while *out-of-band* control, a separate network is used to forward control messages to the network controller.

For both types, *restoration* and *protection* techniques have been proposed. Restoration attempts to select replacement resources only after a failure has occurred, whereas the resources for protection techniques are already reserved before a failure occurs. We discuss such protection and restoration concepts for in-band and out-of-band control.

The work of [38] addresses the problem of fast failure recovery for in-band OpenFlow networks, and is further explained based on the topology of Figure 2.6 (a + b). To illustrate the restoration of control traffic, assume that switch `C` is initially reachable by

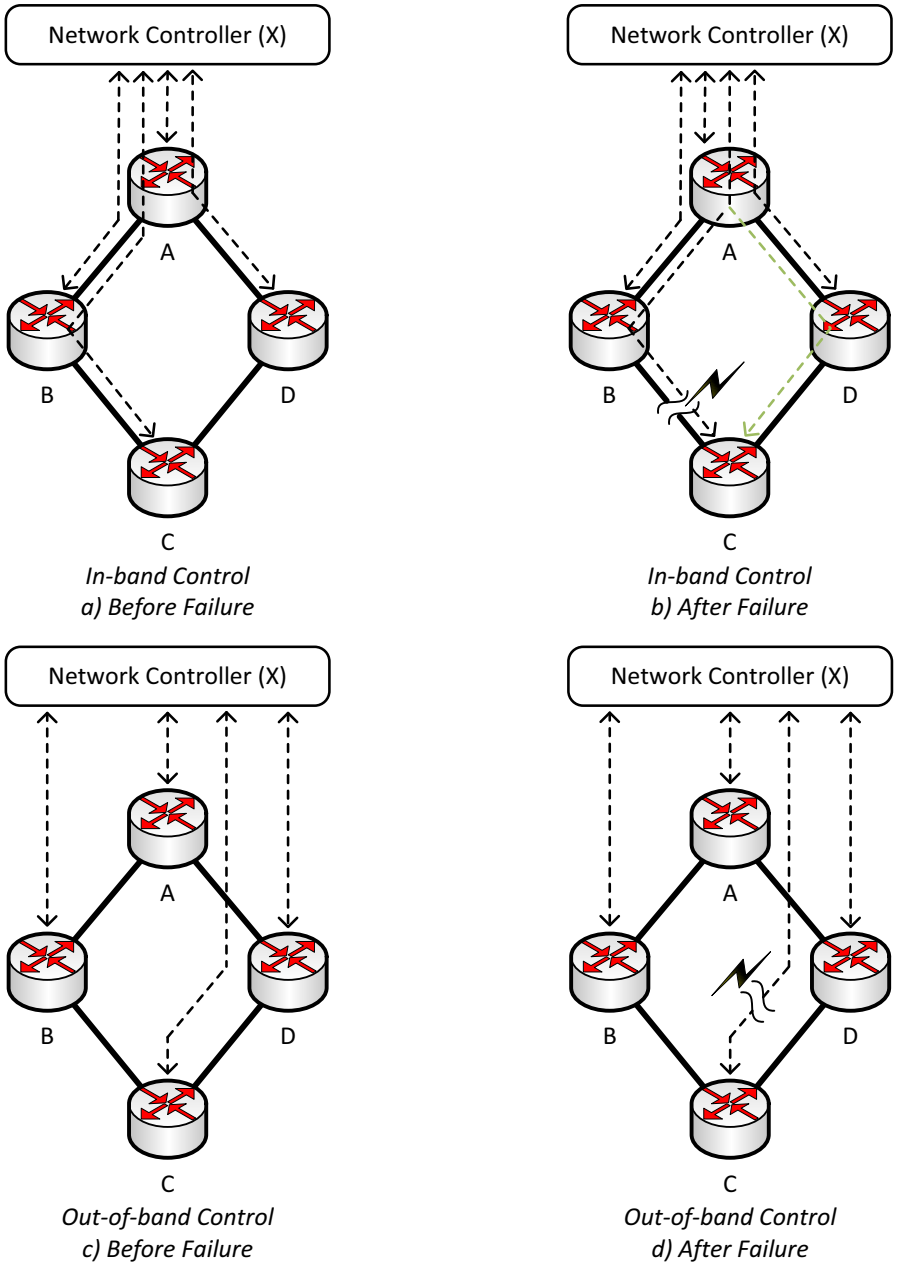


Figure 2.6: Consequences of an interruption of in-band (b) and out-of-band (d) control channels in OpenFlow. Whereas the control traffic in (b) can be re-routed over the link `XADC`*, the failure of the link* `XC` *in (d) activates a failure mode on the switch (see subsection 3.1.2).*

the controller over the path `XABC`. An interruption of the link `BC` will disconnect switch `C` from the network controller, which can detect the interruption of the control channel through the expiration of time-outs (e.g. `Echo` request), or through port-status messages on adjacent switches (e.g. port down on switch `B`). In order to re-route the control traffic to switch `C` over the alternative path `XADC`, the network controller can update the flow

tables of switch A and D. Since the flow table of the disconnected switch C cannot be accessed, the authors suggest that this switch should flood control traffic out to all ports by default. As a result, the switch can always reach the controller over both links (CB and CD), but in a fault condition, either switch B or D is forwarding packets whereas the other switch is dropping them.

For the protection scheme, the *GroupTable* feature in the OpenFlow specification is leveraged to allow the network devices themselves to respond to a link failure. A *group entry* in the GroupTable consists of four components:

```
Group Identifier | Group Type | Counters | Action Buckets
```

Each group entry is assigned to a certain number of *action buckets*, which contain a set of actions. In order to use a group entry to respond to network failures, the *fast failover* group type was chosen. This allows the specification of an action order, which executes the first item if a particular monitored port is in the active state. If the port becomes inactive due to a fault, the second item of the action order is chosen and controller traffic is re-routed over an alternative path. For instance, we consider again our example topology with a broken link BC. The protection scheme would require the following action buckets to be assigned on switch C (pseudo code):

```
1. link to switch B = "active" -> forward to switch B  % normal
2. link to switch B = "inactive" -> forward to switch D  % link failure
```

After re-connecting to the network controller, the flow tables of switch C can be updated and the network can continue operating.

Apart from the recovery methods for the control channel, restoration and protection schemes for the data traffic are discussed in [39, 40] for the case of out-of-band control (see Figure 2.6 (c + d)). Due to the fact that no control messages are transported on the data network, the procedures are simplified compared to the in-band scenario. For instance, restoration after a link failure BC requires that the relevant flow table entries are updated by using add/modify/delete commands in order to establish the alternative link XADC to reach C. This requires that the network topology is known by the network controller, or requires the use of a spanning tree protocol [41]. For the protection schemes, an alternative path can be pre-computed and established by installing appropriate flow entries in the network devices. As before the advantage is a faster reaction time, but at the cost of additional memory (e.g. larger flow table), and a slightly lower performance. As an alternative, the GroupTable concepts can be applied in order to build a protection scheme for the data traffic, as described for the in-band scenario.

Another important aspect of fault detection and mitigation is the failure recovery time, — the elapsed time between the occurrence of a fault and the moment the network is back in operational mode. Experiments described in [38, 39] determined that only the

protection technique can recover OpenFlow networks within 50 ms. The rationale is that no action from the network controller is required, because the switch can directly react to the failure. In the restoration technique, the controller must communicate with the network devices, which takes longer and makes the method less suitable for large-scale networks with a large number of flows.

In order to detect a broken link in a timely manner, different network management protocols can be used for connectivity monitoring. For instance, an inactive switch port can be detected by a Loss of Signal (LOS) failure event. The detection of a broken path between two hosts can be detected by Bidirectional Forwarding Detection (BFD), which is based on a simple *Hello* protocol and defined in RFC 5880 [42]. An alternative is to use the Link Layer Discovery Protocol (LLDP) [43], but this puts a high load on the network controller and limits scalability, since such monitoring messages must be processed at a high frequency. The authors of [44] propose an extension to the OpenFlow specification in order to deploy decentralised network monitoring, including packet generation and processing on the network devices. They demonstrate their approach for fault management of a variant of MPLS, and show that failure recovery within 50 ms is possible.

In case of a link failure, the flooding of notifications of such an event to all network devices creates unnecessary traffic in the network. In order to reduce the number of such messages to a minimum, the authors of [45] propose informing only the affected switches about link failures. The required algorithm is deployed on the network devices themselves, and allows a faster recovery time compared to a controller-based notification scheme. Like the previous concept, it requires additional functionality to be added to an OpenFlow-based switch and so reduces the possibilities for using commodity hardware.

2.6.2 Measurement and Monitoring

Network monitoring informs the operator about the current status of the network, and is the basis for fault detection algorithms. This requires appropriate sensors in the network, for instance switches that provide statistics about flow entries.

NetFuse [46] is intended to detect and mitigate traffic overloading, which can occur for a variety of reasons, for instance due to a distributed denial-of-service (DDoS) attack or as a result of a scheduled backup. In order to detect such sudden traffic surges, NetFuse is installed as an additional layer between the network devices and network controller to process OpenFlow control messages (e.g. `PacketIn`, `FlowMod`, `FlowRemoved`). In order to detect traffic overloading, a multi-dimensional flow aggregation algorithm is used to identify suspicious flows. These are further managed by an adaptive control mechanism, which modifies the control rules on the switches to better handle the traffic overload.

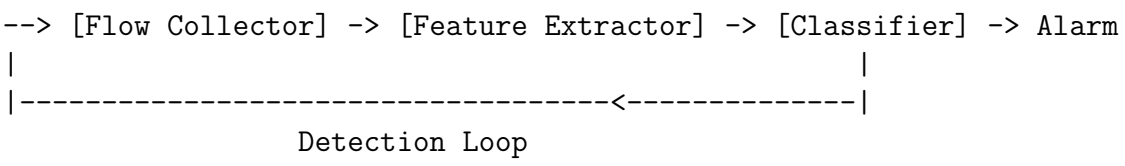
Based on the OpenFlow control messages `Packet-in` and `FlowRemoved`, **FlowSense** [47] computes link utilisation by means of an application that runs on top of the network

controller. This simplifies deployment compared to approaches that require an additional layer (e.g. NetFuse [46]). Since the measurement is only updated after the receipt of a `FlowRemoved` message, a flow rule that contains wildcarded fields in combination with a long expiration time leads to a delayed result. For proactive flow installation policies, a `Packet-in` message might never occur, meaning that active polling would be required to detect the condition. The main application scenario is therefore to analyse microflows with a short flow expiration time.

The work of **OpenSketch** [48] addresses a software-defined traffic measurement architecture. This consists of a simple design for the data plane, which can be implemented on commodity hardware, while data analysis functionality located on the control plane. Based on the idea of *sketches*, which are compact data structures, a more flexible collection of flow-related parameters (e.g. counter) is possible compared to the OpenFlow specification. In combination with the OpenSketch controller and sketch library, new measurement algorithms can be developed, allowing automated configuration of switches in accordance to the requirements of the measurement application.

The installation of additional flow entries for monitoring purposes is addressed in [49]. This scheme is possible because the OpenFlow specification supports multiple stages of flow entries. The network controller can read the respective counters periodically, and can detect *Heavy Hitters*[11] through a hierarchical heavy hitters algorithm. The strength of this approach lies in its use of commodity hardware and its low switch overhead.

The detection of DDoS attacks based on the analysis of flow statistics by neural networks is considered in [50]. The approach was implemented on the NOX controller platform and comprises the following steps:



Firstly, the flow statistics from one or multiple switches are extracted at certain time intervals by the flow collector. In a second step, the feature extractor extracts and identifies the relevant traffic features that indicate a DDoS attack. For instance, the growth rate of single flows in one direction is an indicator of the beginning of such an attack. Based on *unsupervised learning* that leverages Self-Organizing Maps (SOM), the network traffic is classified as normal or abnormal (i.e. attack). The number of switches involved in monitoring can be adapted to new topologies, but an increased number of switches creates additional overhead due to the increased number of control messages.

[11] Also referred to as elephant flows: These are flows that are responsible for the largest number of bytes per second compared to the remaining flows.

The direct measurement of the flow entry related counters is necessary in order to build a traffic matrix (TM), which contains the traffic volume between the origin and destination pairs in a network. But such requests increase the load on a switch and this is undesirable in a network consisting of multiple switches and a large number of flows. **OpenTM** [51] describes strategies in order to determine which switch to query along a flow path in order to reduce overhead. The optimal estimation results are a trade-off between load and accuracy. The authors propose an even distribution for queries of flow counters among all switches in the network, incorporating routing information from the network controller for this purpose.

The authors of [52] describe how P2P traffic may be classified based only on network layer features. This means that the privacy issues associated with deep packet inspection (DPI) are not encountered, and no specialised hardware is needed, since the analysis can be accomplished on the network controller. After determining the relevant network layer features that can differentiate P2P from non-P2P traffic, two classifiers were implemented as an application for the NOX controller and evaluated on a publicly-available dataset.

2.6.3 Debugging, Verification and Testing

The detection of network faults is a major concern for every network operator and relies mainly on human expertise. In order to reduce fault detection time, new methods are required for SDN architectures. For network *debugging*, an approach that incorporates both the network controller (software) and network devices (firmware/hardware) is required to find the root cause of an error. Ndb [53] addresses this requirement. The formal *verification* of controller software can determine whether certain correctness properties (e.g. termination of loops) are guaranteed and is provided by SOFT [54]. The *testing* of computer networks is required in order to assure that all network components work as expected, and should comprise a test scenario that is as close as possible to the operational situation. This issue is addressed by ATPG [55], which generates test packets, whereas OFRewind [56] provides a facility to replay captured scenarios.

OpenFlow allows the use of commodity hardware, requiring a correct implementation of the OpenFlow agents on all network devices. The ONF has defined a test specification with a series of test cases [57] in order to determine the conformance of an OpenFlow-enabled switch implementation (version 1.0.1). This is especially useful for vendors in order to avoid software bugs during the development phase of a new switch model. A similar approach is taken by **OFTest** framework [58], which is part of the Project Floodlight[12] and allows the development of platform-specific test scenarios.

SOFT [54] addresses interoperability testing in order to assure a correct implementation on all switches in the network. By using symbolic execution, all possible paths in the program (firmware) are exercised and the behaviour of each network device is

[12] http://www.projectfloodlight.org/floodlight/

State Layer	Code Layer
Policy	
	Application
Logical View	
	Network Hypervisor
Physical View	
	Network OS
Device State	
	Firmware
Hardware	

Table 2.3: State and code layers of the SDN stack [59]. In case of an error-free network, the state layers can be mapped correctly to every other layer (equivalence). Otherwise, an error can be localised between layers that deviate from each other, and can be identified in the intervening code layer.

determined. Subsequently, a cross-check between the network devices using a constraint solver reveals inconsistencies among input sets. This allows network operators to detect implementation errors before deployment and guarantees error-free orchestration of network devices from different vendors.

A systematic approach to troubleshooting SDN is given in [59], which proposes a workflow that separates the SDN stack in *state* and *code* layers, as shown in Table 2.3. The state layers represent the network configuration, while the code layers describe the mapping between two state layers. The methodology allows automatic detection of the code layer where the fault is located. Normal network behaviour results in an equivalence between all the state layers, such that each layer can be mapped to every other layer. In the case of a network fault, the error can be localised to the code layer lying between state layers that are not equivalent. Once the error has been located, its cause can be determined by using the techniques that are discussed in this subsection.

Like a debugger in software engineering, the *network debugger* **ndb** [53] helps an operator to determine the root cause of software faults (or *bugs*) in the network. Each switch sends *postcards*, which are triggered whenever a packet containing information about the matched flow entry arrives at a switch. The postcards from all switches are collected at a centralised collector, which can be requested to determine the packet traces that relate to a *packet breakpoint*. For instance, in order to investigate a reachability error for a packet sent from host A to host B, the following request can be used:
Packet A → B, not reaching B.
The response is similar to the output of `traceroute`[13], and lists all hops with the corre-

[13]The Linux traceroute tool tracks the route packets take through an IP network on their way to a given host.

sponding matched flow entries including those for the switch where the forwarding error is occurring. This allows the programmer to find errors not only in the controller software itself, but also in network devices and the data plane.

A different approach is to actively generate test packets in order to detect connectivity failures. While this can be realised with tools such as `ping`[14] or `traceroute`, it is not feasible to test all possible links or forwarding rules. **Automatic Test Packet Generation** (ATPG) [55] defines a model of the router configuration, which is leveraged to generate an appropriate set of packets to test either the links or the forwarding rules. After sending the packets to the network, a fault location algorithm is used to identify the type of error before further analysis by the network administrator.

OFRewind [56] provides a record and replay debugging facility for SDN, as shown in Figure 2.7. It is inserted as a proxy between the network controller and network devices, and allows the interception and modification of control messages for record or replay purposes. In order to store user traffic, *data stores* controlled by the OFRewind component are connected to the switches. Different replay modes allow different test scenarios, for instance to receive only the control messages sent to the network controller for debugging purposes (ctrl mode). The framework is light enough in weight to be deployed in production networks.

VeriFlow [60] provides a more focussed view of data plane verification. It adds an additional layer between the network controller and network devices to check new rules in real time before they are deployed in the network. The rules of the network are partitioned into a set of equivalence classes (ECs), where similar forwarding actions are located in the same EC. These are stored in tree data structures, which describe the forwarding graphs of packets in the network. Since new rules affect only certain ECs, they are analysed by VeriFlow by running queries based on *invariants* in order to detect whether new rules violate them. Such invariants can be set up through an API, and cover a wide range of different network conditions, for instance reachability, loop-freeness and consistency.

Anteater [61] also focusses on data plane analysis. It represents the collected network topology and forwarding information bases (FIBs) from the network devices as boolean functions. In combination with user-defined network invariants (e.g. loop-free forwarding), a boolean satisfiability problem (SAT) solver performs the analysis. Using this method, the authors were able to detect 23 bugs in a campus network. The advantage of such a data plane related mechanism is that the real network behaviour can be analysed without needing to model different routing protocols, so simplifying the procedure. Also, unlike control plane based analysis, it allows the detection of router software bugs.

The **Header Space Analysis** (HSA) [62] framework focusses on detecting errors

[14] This tool sends an ICMP ECHO_REQUEST to a host in order to test network reachability.

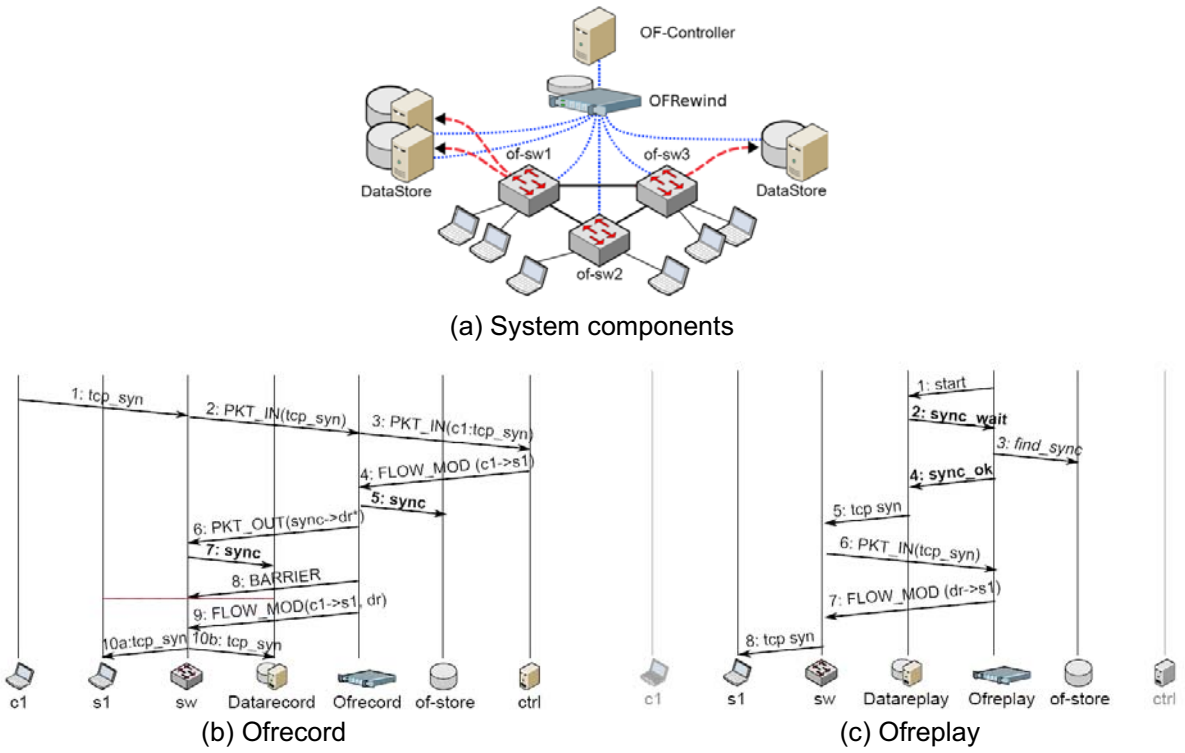


Figure 2.7: System components of OFRewind [56] (a). A TCP SYN request from host c1 to s1 can be recorded with Ofrecord (b), and replayed again with the Ofreplay mechanism (c) were the request (5:tcp syn) is sent from the Datareplay storage system instead of from c1.

such as reachability failures, forwarding loops, traffic isolation, and leakage problems. Each packet header is represented as a point in a *header space*, while network devices and ports are modelled in the *network space*. When a packet traverses the network, it is transformed from one point in network space to other point(s) elsewhere in network space. The separation of network traffic is realized through *slices* (see FlowVisor subsection 2.5.4), which are represented as a subset of the network space. Network devices, such as switches and routers, can further be modelled through appropriate transfer functions. In combination with header space algebra and analysis, which are available in the *Hassel* library, this allows the detection of violations and the corresponding failures. While such a framework is able to locate the source of an error (e.g. inconsistent routing tables), it cannot determine the reason why the error has occurred.

Testing of new controller applications requires an OpenFlow testbed and the proper configuration of controller and switches. Since only a small number of people have access to such resources, networks consisting of virtual machines (VMs) are an attractive alternative. Due to the huge memory overhead that is needed for setting up a large network, scalability extends to only a handful of switches. A more lightweight type of

virtualisation is possible with the **Mininet** [63] network emulator, which allows network prototypes to be tested on a standard laptop. In order to create a new network, the following command gives an example of a test on a virtual network with a tree topology of depth 2, a fanout of 8 and a network controller running NOX:

```
mn --switch ovsk --controller nox --topo tree,depth=2,\\
fanout=8 --test pingAll
```

All links, hosts, switches and controllers are emulated, and shell commands allow the use of basic network tools (e.g. `ping`). Mininet allows developers to execute and test new controller programs in various network topologies. The limitations of this approach stem from the fact that the performance of the virtualisation machine is a bottleneck when subjected to high loads, and also from the speed of packet forwarding, which is of complexity $\mathcal{O}(n)$, compared to $\mathcal{O}(1)$ for a table lookup in a hardware switch based on Ternary Content Addressable Memory (TCAM) (see subsection 4.4.1). Due to its widespread usage, Mininet is the most common emulator for SDN and OpenFlow.

Network debuggers are necessary in order to detect errors when software faults do not make themselves readily apparent. **NICE** [64] can be used to test controller programs to discover violations of correctness properties caused by bugs in the controller software. Model checking is used to describe the network topology and to explore the state space, which incorporates the controller, the switches and the hosts. In order to reduce the size of the input space, event handlers on the controller are executed automatically by a symbolic engine, which generates test inputs and injects packets into the network. The NICE prototype was evaluated on three real OpenFlow applications that were written in Python for the NOX controller. A typical example is a bug in the MAC-learning switch application, which should send packets to a particular device even when that device moved to a new location in the network. If a hard time out notification is missed, all packets are still delivered to the old location and the corresponding flow entries are not updated with the new location parameters. The tool allows the detection of such design flaws and implementation bugs and is publicly available.

2.6.4 Security

The architecture of SDN allows the realisation of new security concepts that were not possible in non-SDN networks. For instance, each network device can be configured to block packets in a similar manner to a network firewall. Although intrusion detection has traditionally required expensive hardware solutions, the work of [65] demonstrates how anomaly detection algorithms can be adapted to OpenFlow-based networks implemented in the network controller. On the other hand, the architecture of SDN provides a new play ground for attackers and requires adequate protection mechanisms. For instance, FortNOX [66] provides a security mechanism that protects the flow installation mechanism against adversaries.

The work of [65] addresses OpenFlow anomaly detection, and argues that it should be moved from the core to the home network (close to the user) in order to obtain better detection results. The authors adapt four existing algorithms for use with OpenFlow, including the detection of scanning infections on hosts [67], rate-limiting in the face of infection [68], anomaly detection using maximum entropy [69] and an anomalous value detector [70]. Since the processing load is distributed to the home users, this approach reduces processing requirements at the Internet Service Provider (ISP) and reduces costs. In addition, the deployment of detection algorithms on the network controller does not influence the performance of packet forwarding on the data plane.

The **FRESCO** [71] framework allows the deployment of security services for OpenFlow. It comprises a script language that allows the development of security services based on an API and a library consisting of 16 re-usable modules, which provide basic functionality. The FRESCO framework itself is implemented as an application built on the NOX controller.

A protection against IP scanning-based attacks is described in **OpenFlow Random Host Mutation** [72]. This is based on the idea that static IP addresses are an easy target for attackers, but can be avoided through proactive techniques that change the hosts' IP addresses over time (a Moving Target Defence (MTD)). In the case of OpenFlow, the real IP address is retained by the host but replaced by a virtual IP address which is frequently reassigned to the network devices by the network controller. This requires an address translation mechanism, and the assurance of constraints such as mutation unpredictability. The approach was evaluated using MiniNet against an external network scanner (`NMAP`[15]) and a worm attack.

FortNOX [66], a new security policy enforcement kernel, improves the security of the flow installation procedure in OpenFlow. This can be exploited by adversaries in order to take over a network and is not well protected in OpenFlow. FortNOX requires digital signatures for authorisation. Depending on the application, these define the priority level of a flow rule in the flow table. In addition, FortNOX detects when flow rules circumvent existing security policies. This is possible not only in case of overlapping IP ranges, but also if a rule sets a new packet header and the modified packet could reach a destination that is otherwise blocked by an existing rule.

2.7 Summary and Limitations

The current research in OpenFlow-based SDN has already tackled some questions that must be addressed when OpenFlow can move from being a research testbed into operational deployment. As discussed in this chapter, fault management concentrates mainly on detecting and correcting the interruption of links, which can affect not only the data

[15] http://nmap.org/

traffic, but also the communication with the controller. In order to propose new fault detection schemes for SDN, the relevant network monitoring techniques for this architecture must be explored. Several approaches to network measurement and monitoring are based on inserting an additional layer, between the network devices and network controller. This allows inspection of the OpenFlow control messages, for instance to identify traffic overloading. Because such an additional layer deviates from the original OpenFlow architecture and requires more hardware, analysing flow entries on the network controller becomes more attractive. This can be reached by deploying monitoring algorithms running on top of the network controller. The analysis of flow entry statistics can be used to reveal traffic characteristics and to build a complete picture of the network by incorporating inputs from all network devices. For software debugging, a network debugger can trace packets in the network. While formal verification may be used to determine the correctness of flow rules, the existing approaches do not consider the instrumentation of the controller software to enable code debugging. This is required in order to generate detailed network snapshots, and to inspect function parameters under certain network conditions. From a security perspective, the potential of OpenFlow to realise new security schemes has not been well explored. The first steps have been taken by modifying and deploying existing anomaly detection algorithms for the SDN architecture. Analysing packets within a `Packet-in` message on the network controller has not yet been addressed, but could be used to identify and classify malicious communication before flow installation. While several of the benefits of SDN for network security have been investigated, new attack vectors targetting the centralised network controller and the unprotected control channel have not yet been fully addressed.

Chapter 3

Towards Network Programmability with OpenFlow

The advent of programmability in computer networks overcomes current limitations stemming from hardware restrictions and proprietary software. Compared to traditional networks, programmability increases flexibility and enables the deployment specific solutions that could not previously because limitations imposed by vendors. While network operators benefit from this new freedom, new challenges concerning network management arise. In this chapter we discuss these and give an overview of the solutions that are proposed in this thesis.

3.1 Challenges

Current solutions in *network management* were designed for a network model where software is mainly used in the form of firmware for embedded systems. This applies particularly to switches and routers, which have traditionally been non-extensible and non-programmable network devices containing control, data and management planes. With the recent concepts of Software-Defined Networking and OpenFlow, the functionality of such devices can now be based around software that is located on a centralised network controller. This increases flexibility, since new functionality depends on the user's choice of network applications instead of device specifications defined by vendors.

While network programmability allows the deployment of new techniques that were not possible with the old design schema, software-defined network architectures demands new solutions for network management. We discuss in the following network management aspects, that are required for an operational usage of SDN.

3.1.1 Network Operating System

The use of a centralised network controller allows control of all network devices from a single point. Since most of the control plane functionality of each device is moved to the controller, several previously separate control plane operations can be executed on a single machine. This reduces the processing power required by the switches and routers, and also simplifies hardware requirements, so allowing the usage of *commodity hardware*, which is used mainly for packet forwarding on behalf of more specialized hardware, and to exchange control messages with the network controller by way of the OpenFlow protocol. The network controller itself hosts a *network operating system*, that allows the installation of network applications that meet the requirements of the operator.

Such applications can be separated into network services, which define the functionality of the network controller software, and business applications that are used interact with them.

Network Services The network controller, in combination with the chosen network services, allows the forwarding behaviour of packets in the network to be customised. This is achieved through actions that can be applied to each packet, based on rules located on the network devices (see subsection 2.5.2). Typical examples of network services are to provide layer 2 switching capabilities or Multiprotocol Label Switching (MPLS) to a network. Through the increased flexibility of a software-based network controller, even sophisticated network management concepts such as *multi-tenancy networks* [73] can be realized. This requires additional mechanisms to identify tenants in the network due to the problem of overlapping IP addresses. Latter can be solved by processing Address Resolution Protocol (ARP) requests on the controller, and by deploying forwarding rules that are based on tenant MAC address to allow separation of hosts.

Business Applications This type of application is software that is used to access network services, and allows customers to configure network services that are customised to their use cases. This concept allows software companies to develop software for various application scenarios. In order to access the network services, it is required to define an interface that is open and accepted by a broad community.

Early implementations include Project Floodlight's JSON/REST[1] API, which allow users to develop applications that interact with the controller. The API uses an Uniform Resource Identifier (URI) to access different controller functions, which can be modified through GET/POST/PUT/DELETE methods with appropriate arguments. For instance, the following code allows a flow entry to be added to switch 1 in order to forward packets from port 1 to port 2. The first command inserts the flow entry on switch 1, while the second command is used to activate:

[1] http://www.openflowhub.org/display/floodlightcontroller/Floodlight+REST+API

```
curl -d '{"switch": "00:00:00:00:00:00:00:01", "name":"flow-mod-1",
"priority":"32768", "ingress-port":"1","active":"true",
"actions":"output=2"}' http://<controller_ip>:8080/wm
/staticflowentrypusher/json

curl http://<controller_ip>:8080/wm/core/switch/1/flow/json;
```

Even though such an API is an inherent requirement of SDN, it still needs to be defined. Definitions can be developed by the ONF, or by industry, and may later be standardised by the Internet Engineering Task Force (IETF). The core idea is that applications can be deployed on a network in a similar way that an "app" can be installed on a smartphone.

OpenFlow was initially proposed with *one* centralized network controller, but there might be scenarios that require the use of a *multiple* controller environment, for instance, by partitioning customers between different controllers that provide different services. In this case, the installation of flow entries on a switch must be in accordance to the authorisation of each controller and should not influence the packet forwarding resulting from other controllers' policies. This was the rationale for the development of FlowVisor [36], an additional controller that is located between the network devices and operational controllers in order to assign *slices* of the network to each of them. For instance, it allows the coexistence of research and operational traffic in the same network, but means that only one network infrastructure needs to be maintained.

3.1.2 Control Channel

Reliable communication between the network controller and the network devices must be guaranteed in order to reduce the risk of a network failure. If the communication channel is broken, data and control planes are separated, leading to unpredictable network behaviour as shown in Figure 3.1. In such a case, the switch enables one of two *fail modes* that further depend on the type of hardware and its placement in the network topology.

The standard reaction of the switch in the case of a broken control channel is to enable the *fail secure mode*. The switch keeps forwarding packets based on the installed flow entries, but does not update the flow table. All packets that do not match the existing flow entries are dropped. Since flow entries expire automatically, in order to avoid deletion, the hard time out should be set to a value that is higher than the time it takes to fix the control channel.

In the case of a hybrid switch, which, as well as supporting SDN, also operates in a non-SDN manner through an existing control plane, the *fail standalone mode* is applicable. This disables the OpenFlow-based forwarding mode, and activates the vendor's firmware and configuration instead. For instance, the switch can then forward packets based on a L2 learning strategy, which maps each source MAC address to the physical

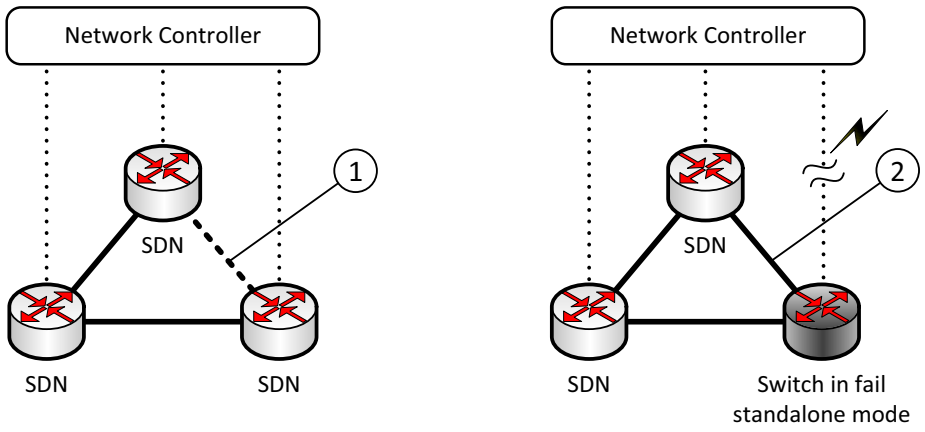


Figure 3.1: Implication of an interruption of the control channel: although the controller has avoided loops in the network by disabling redundant links (1), a switch that enables "fail standalone mode" might enable the link due to limited knowledge of the network and the absence of spanning tree protocol (2).

port. The appropriate mode should further be chosen based on the location of the switch in the network topology. We now consider *critical* and *non-critical* placements in the network topology as important for the selection of the appropriate fail mode.

Critical We consider a switch to be critical if there are no alternative routes available that lead to a certain destination. An example would be the only switch that is connected to a router in order to forward traffic to the internet. In such a case, the fail standalone mode should be activated in order to establish new flows and to provide internet connectivity to all hosts.

Non-Critical Such a scenario applies if a switch failure can be completely mitigated by re-routing all flows over alternative routes to the destination. In this case, the fail secure mode is applicable and only packets that match the existing flow entries are forwarded. Even if all flow entries are expired and the switch stops forwarding packets, the connection is still guaranteed by the alternative routes, but the available bandwidth might be reduced.

Apart from the network devices, the network controller itself needs to implement countermeasures when confronted with a broken control channel. If the fail mode and configuration of the switch concerned is known by the fault management engine on the controller, the logical topology of the network can be adapted in accordance to the new situation. For instance, in order to avoid loops in a L2 network when a switch has enabled links due to a fail mode, the controller can re-run the spanning tree protocol (STP). If the network topology is known by the controller, it can update the flow installation policy and adapt to the fail mode of the switch. The optimal reaction to a broken channel also

depends on the type of network device (switch or router), the network topology and the supported management protocols. The requirements of the network after failure, and the range of countermeasures available, define the optimal strategy to handle such a situation.

Another important aspect of the control channel is the type of physical topology that is used to connect the network devices with the network controller. There are two basic ways that this can be done are *in-band* and *out-of-band* control. The former uses the network that is controlled by the switch to reach the network controller, while the latter has a separate network for this purpose.

In-band In this scenario, the traffic of the controller and hosts are forwarded across the same network, introducing several attack vectors that can be exploited by an intruder, because of the controller access that it grants to each host in the network. An attacker can compromise the controller with a denial-of-service (DoS) attack, which results in a performance drop and possible breakdown of the controller. A more sophisticated attacker who obtains access to the controller might want to install malicious software or manipulate packet forwarding. By altering the forwarding mechanism, user traffic could easily be replicated and forwarded to the attacker's machine by installing a suitable flow entry, so enabling communication sniffing or man-in-the-middle attacks. The replication of packets is difficult to detect since such a modification is not obvious, and so may go unrecognised by the network operator. Nevertheless, in-band control has its attractions, since no additional network need be built and maintained and no dedicated switch ports are required. However, it increases the complexity of the flow table mechanisms needed to handle the controller-related flow entries. It also raises questions about how to boot an OpenFlow-based SDN with in-band control (see subsection 3.1.5).

Out-of-band In out-of-band control, an additional network physically separates the controller traffic from the network which is used by the hosts. This simplifies the switch implementation, since no flows that relate to the controller communication need be considered by the flow tables. Moreover, excessive switch traffic cannot interfere with the controller traffic, so increasing reliability. From a security perspective, in contrast to the in-band solution an attacker located on a host cannot impersonate a switch or connect to the controller. This also increases confidentiality, since sniffing of controller traffic is impossible and no information about the network topology can be revealed. Of course, physical access to all switches must be secured, since otherwise all advantages concerning security are lost. An open question still concerns the network infrastructure that is required for such out-of-band communication. Considerations include the type of network topology and the redundancy level for links.

Apart from the described control channel strategies, a *hybrid mode* could lead to a higher degree of fault tolerance. For instance, all network devices could be connected to the network controller over an out-of-band control channel. If a link to a switch is broken, the network device could switch to in-band communication in order to continue

in operating mode. Apart from the additional cost of maintaining a separate network for out-of-band control, the hybrid control channel delivers all the discussed advantages but allows also resilience against faults (e.g. link failures).

3.1.3 Software Debugging

The network controller and the related software require special attention in regard to *software engineering* and reliability. For instance, a software bug can lead to a failure of the network controller, which has an immediate effect on the connections of all network participants. Such coding errors can occur due to software modifications, which are inevitable in order to adapt to new customer specifications or to extend software with new features. If the error is detected by the programmer, the affected code can be fixed or an older version can be redeployed. In order to avoid errors in controller software before deployment, programmable networks require special attention to *debugging* and *testing*.

The complexity of the network controller software is often reduced by following an event-driven programming paradigm, where a single event can trigger the code execution in several programs. For instance, suppose a new packet arrives at a switch and cannot be matched to the existing flow entries (*table mismatch*). This results in a `Packet-in` event on the controller, which might in turn trigger several functions in the controller software in order to install a flow entry on the switch. Such an event-based architecture requires an appropriate debugger that is capable of tracing such events in order to determine faults related to bugs in the controller code.

Additional tools are also required for software testing, which comprises all methods needed to validate or verify software. A framework to test network applications needs to emulate the network devices and OpenFlow protocol, in order to provide a realistic test environment if a testbed is not available. In addition, a packet generator is required in order to emulate packets that traverse the network.

3.1.4 Configuration

Computer networks consist of a series of network devices, where each device requires a configuration specific to the network and the application scenario. Typical configuration data includes general settings such as IP address and network mask, as well as device- and protocol-specific parameters. These can either be manually edited by command line interface (CLI) scripting, or through the use of management protocols. The rationale for such a protocol is provided by the fact that it quickly becomes infeasible for an operator to read or edit settings manually on all network devices.

The protocol OpenFlow focusses only on the flow-based forwarding mechanism, and provides no solution for the issue of device configuration. To fill this need, the ONF has specified the *OpenFlow Management and Configuration Protocol (OF-Config)* [74], which is a companion to the OpenFlow protocol, and allows the remote configuration

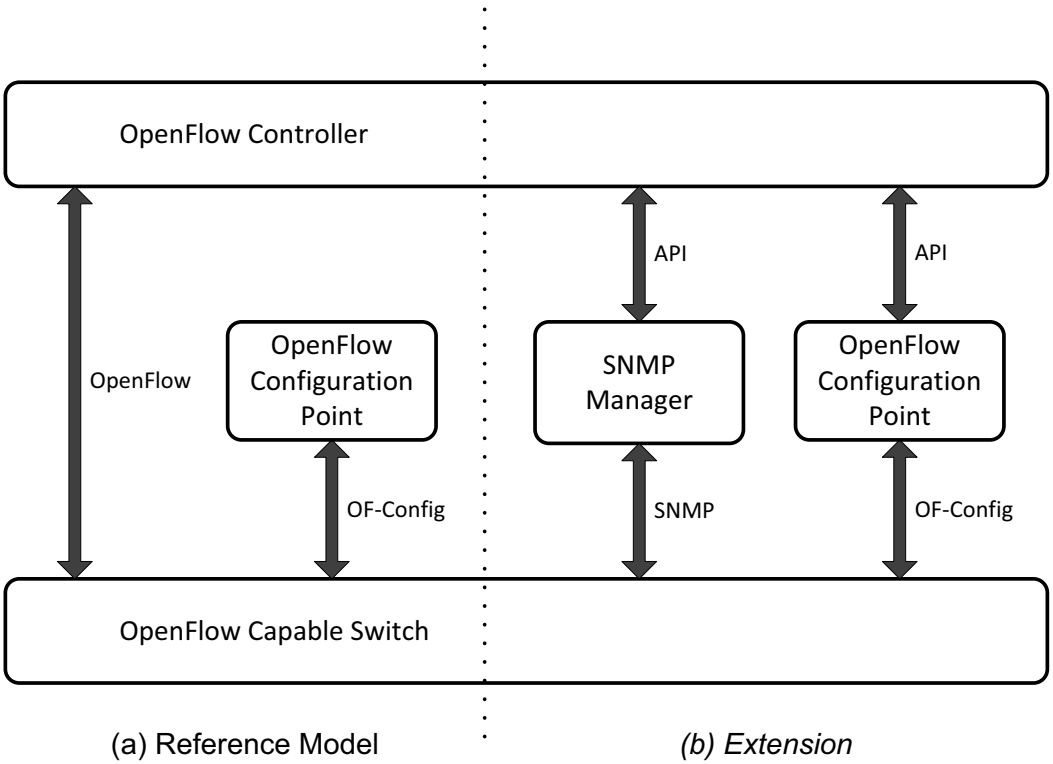


Figure 3.2: Proposed extension of the OpenFlow reference mode. This incorporates APIs for the network controller in order to access MIBs over SNMP, and to configure the switch through OF-Config.

of network devices from *configuration point(s)* as shown in Figure 3.2. The OF-Config protocol allows the following settings to be configured on OpenFlow-enabled switches:

- Assignment of one or more controllers to a switch.
- The modification of interface properties.
- Configuration of certificates for secure communication to the controller.
- Determination of switch capabilities.
- Configuration of tunnelling protocols.

The OF-Config protocol is based on *NetConf* (RFC 6241 [15]), published in 2006 as the transport protocol. It uses a generic data modelling language called YANG (RFC 6020 [75]), which represents data structures in an Extensible Mark-up Language (XML)-based data encoding. YANG can be used to model the configuration and state data of network elements.

The following example shows the OF-Config syntax of the controller class that describes related parameters on the switch:

```
<controller>
  <id>Controller3</id>
  <role>master</role>
  <ip-address>192.168.2.1/26</ip-address>
  <port>6633</port>
  <local-ip-address>192.168.2.129</local-ip-address>
  <local-port>32768</local-port>
  <protocol>tcp</protocol>
  <state>
    <connection-state>up</connection-state>
    <current-version>1.2</current-version>
    <supported-versions>
      <version>1.2</version>
      <version>1.1</version>
    </supported-versions>
  </state>
</controller>
```

OF-Config has the potential to help operators to remotely configure their network devices, although the protocol is still awaiting implementation by vendors. Without the ability to configure network devices using a suitable protocol, network monitoring becomes cumbersome without an automated approach. In order to help the administrator with this task and to find the root cause of system faults and errors, most network devices send a notification using *Simple Network Management Protocol (SNMP)*. Because of the lack of existing monitoring solutions for OpenFlow-based scenarios, the adoption of SNMP seems reasonable due to its widespread use by other network devices, which means that it can simplify the integration of OpenFlow based switches into existing network infrastructure.

SNMP is defined in RFC 1157 [76] and is the most common protocol for network monitoring and fault handling. All participating network devices, which are defined as *network elements*, can be polled from the administrator's *network management station (NMS)* to read or change settings if authorised. To structure and store data in network elements, SNMP uses a tree structure that is defined as a *Management Information Base (MIB)* [77]. Such a tree can be considered to be a virtual database and contains managed objects that are accessible through a unique object identifier (OID) that is identical on all network devices.

The advantage of the standardised OID is that a single request is answered by all network devices that support such an object. In the case of a huge network, polling over SNMP creates additional network load and is often not necessary when the network is healthy. This has resulted in MIBs that send a status message only in case of a fault, for instance the alarm MIB, which is defined in RFC 3877 [78]. MIBs originating either

from industry or other organisations are defined for several areas by the IETF. A typical example is the MIB-II, which contains information about the TCP/IP protocol (RFC 1213 [79]) and is supported by many network devices.

Even though SNMP allows new values to be set in a MIB, it is rarely used for configuration due to the inconvenient procedures that are required to achieve this. A specific MIB for OpenFlow-based networks has yet to be developed, but is required in order to allow operators to manage such networks using SNMP. The features that must be supported by such a MIB are similar to those specified for OF-Config.

The following example shows a small part of a possible structure of a MIB that could be used to store information about an OpenFlow switch:

```
[Object Name]           [Object Identifier]
mib-OF                   1.3.6.1.3.1
  controller             1.3.6.1.3.1.1
    main                 1.3.6.1.3.1.1.1
      adress             1.3.6.1.3.1.1.1.1
      port               1.3.6.1.3.1.1.1.2
    backup               1.3.6.1.3.1.1.2
      adress             1.3.6.1.3.1.1.2.1
      port               1.3.6.1.3.1.1.2.2
  suppVersion            1.3.6.1.3.1.2
  ...
```

For instance, the object `suppVersion` contains the supported version of OpenFlow on the switch and could be defined as:

```
suppVersion OBJECT-TYPE
      SYNTAX  FLOAT
      ACCESS  read-only
      STATUS  mandatory
      DESCRIPTION
              "The OpenFlow version that is supported
              by the switch."
      ::= { 1.2 }
```

As discussed above, the OF-Config or SNMP protocols can be used to access configuration data on a switch. The network controller itself might also need to request network parameters which are accessible only over SNMP or OF-Config. For instance, CPU load and memory consumption data might be needed for load-balancing applications. It is therefore necessary to define APIs that allow an exchange of configuration data between the network controller to the configuration points (OF-Config) or network management station (SNMP), as shown in Figure 3.2. For instance, the controller could send a request

containing the appropriate OID (e.g. CPU load) to the network management station, which triggers the request over SNMP to the switch, in order to request the data from the MIB. Such a concept requires that the controller and involved devices are modified in order to support the API. This can be fostered through a standardisation of such APIs in order to increase acceptance by developers and vendors.

3.1.5 Bootstrapping

Bootstrapping concerns the mechanisms that start a system from an initial state and bring it into operational mode. For OpenFlow-based networks, this can be formulated as a problem of how to establish communications between the network devices and the network controller. Necessary preconditions are that the network controller is running and that all required applications are installed and started. Communication over the OpenFlow protocol requires that network controller and network devices can be identified by a unique IP address and corresponding network mask, with which the network devices can register themselves at the controller in order to enable operational mode. As well as being required when a network is started for the first time, bootstrapping is relevant to switches that have lost communication to the controller. This requires that the switch must automatically reboot and attempt to reconnect to the network controller.

Before booting a network for the first time, a network technician must configure the network devices. In order to simplify the process and to avoid mistakes, specific configuration files can be defined for each network device, and be made accessible over a Trivial File Transfer Protocol (TFTP) server [80]. The drawback is that such a configuration requires the intervention of an expert, which is costly for the operator. In addition, it complicates the replacement of broken network devices when such an expert is not available. When network management protocols are used for configuration (see subsection 3.1.4), manual modification can result in conflicts and must be avoided.

Automated bootstrapping mechanisms are techniques that allow a network device to retrieve its configuration settings without accessing files from a configuration server. This requires an automated management of IP addresses in the network, for instance by deploying a Dynamic Host Configuration Protocol (DHCP) server. The assignment of IP addresses, and the establishment of a connection with the DHCP server, is especially challenging if the switch communicates with the network controller using in-band mode (see subsection 3.1.2), as requires that a switch establishes a path through adjacent switches in order to reach the controller. The authors of [81] describe an automated bootstrapping approach that is based on DHCP, and which can be deployed as a service on a dedicated machine, or run on the network controller in order to provide IP addresses to the network devices.

An automated technique is also required to inform network devices about the IP address of the network controller. This can be achieved by using the OF-Config protocol,

as described in subsection 3.1.4. An alternative approach is to extract the IP address with the aid of *network discovery protocols*. These are broadcast messages that allow network devices to inform neighbouring devices about their existence and also provide some device-specific parameters to them. Such discovery protocols are used by the NOX [28] controller, which contains a module that leverages the *link layer discovery protocol (LLDP)* in order to explore the network topology. This requires the implementation of LLDP agents on all network devices in the network.

3.1.6 Faults

Fault management encompasses all methods that are used to detect, isolate and correct errors in computer networks. Minimizing fault detection time is mandatory for network operators in order to reduce the cost of a network failure. As a single point of failure the network controller presents a risk since a failure affects all network devices. In order to increase the reliability of the network, a backup controller needs to be provided in order to take control responsibility in case of an incident. Such a solution requires synchronisation in real-time between the two network controllers in order to mirror the current network state and to reduce the downtime after a failure to a minimum. The information synchronised concerns the network controller and network services.

In addition to the deployment of one or more backup controllers for increased reliability, dedicated controllers for different tasks can be deployed in the network. This separation minimises the risk that a fault in one network application will affect the network as a whole. Nevertheless, such separation creates additional complexity if applications require to communicate with each other but are running on different controllers.

Beyond the network controller itself, faults on network devices must be detected in order to mitigate their consequences. This can be achieved by using a forecasting system that is based on the analysis of *syslog* messages in order to predict certain system events [82]. Historical system logs can also be used to detect future anomalies by generating signatures from previous alert conditions [83]. The network controller, due to its centralised position, allows the collection of system events from the network for network monitoring purposes. The broad range of different events (e.g. error codes, packet metadata) can be analysed in a fault correlation engine in order to detect failures that are characterised by several low-level events.

A first approach is to consider switches and routers as *probes* that propagate error messages to the network controller. Such a passive mode requires that the switch is still capable of sending such an event. For instance, a network device can send an error message using the OpenFlow protocol in case of a fault. In the case of a failure of the device or a broken control channel, the network controller cannot be informed about the event. An active mode avoids this by polling network devices periodically for status updates. This is implemented in OpenFlow by sending *echo requests* between switches and

network devices, with a time-out signalising that either the control channel or network device is broken.

Because network devices in SDN have no control plane, errors are limited to the data plane, reducing the number of possible faults compared to non-SDN. A typical error on the data plane is related to the flow table of a switch. In non-SDN, a switch's Content Addressable Memory (CAM) stores, for each source MAC address the physical port through which the device may be reached. An attacker can exploit this behaviour by sending packets with randomly-spoofed source MAC addresses in order to fill the table with useless entries (*MAC flooding attack*). A full table forces the switch into fail-safe mode, which forwards all packets out to all interfaces. This allows an attacker to sniff traffic from all hosts that are connected to that switch. In SDN, a similar attack is possible. By consuming all allocated memory that is reserved for the flow table, an attacker can prevent new entries from being installed on the switch, hindering packet forwarding and also affecting the network controller to some extent, because it has to tackle an increased number of `PacketIn` requests and corresponding flow installation messages. In addition, a flaw or misconfiguration of the flow installation policy can lead to a similar fault and must be avoided through extensive testing.

Another goal of fault management is to make networks more *autonomous*, for instance making them able to react automatically to failures of *security appliances* (e.g. firewall). In non-programmable networks, this requires manual intervention by an administrator in order to deploy a backup system as shown in Figure 3.3. In SDN, this can be automated by updating the flow entries on the respective switches in order to re-route traffic to a security appliance. This limits interruptions for users or network services to a minimum. Before this can be done, the failure must be detected. This can be accomplished either by polling the IDS or by receiving an appropriate error message. In addition, the network controller can analyse the statistics of flow entries that are used to forward traffic over a security appliance. An unchanging number of received packets/bytes or a flow that reaches the idle time-out can be considered to be an indicator that the communication is broken.

3.1.7 Security

Network security is mandatory for any network operator in order to thwart attacks that originate from attackers located on the internet or within the operator's network (i.e. insider attacks). Such threats are typically countered by deploying different kinds of security appliances in the network, as shown in Figure 3.3. The mechanisms of such systems have some common structure. After receiving packets at the network interface by enabling *promiscuous mode*, the header and payload are analysed and compared with known *signatures* that are defined as rules in a configuration file. In the case of a rule match, the action that is applied to the packet depends on the type of security appliance. For sensors such as *Intrusion Detection Systems (IDSs)* that operate in passive mode, a

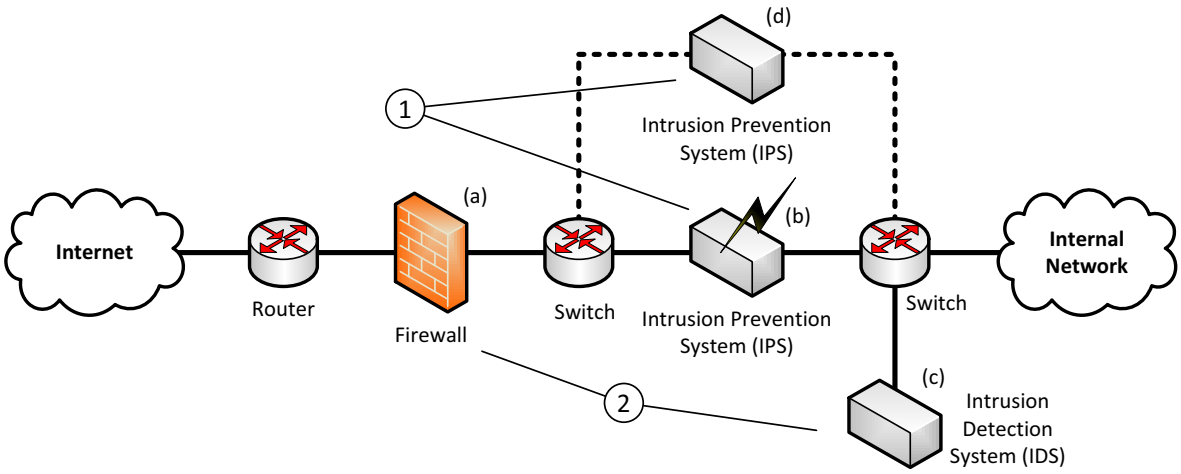


Figure 3.3: Example of network security in non-SDN. (1) Security appliances (a, b, c, d) cannot adapt to failures automatically by activating an alternative device. (2) No synchronisation between Firewall and IDS rule sets, increasing the chance of misconfiguration. Moreover, detected attacks cannot activate countermeasures in adjacent systems, for instance by firewall reconfiguration.

triggered rule results in a message that is written to a log file, describing the action together with its flow information for later inspection. Intrusion Prevention Systems (IPSs) or firewalls operate in an active mode, which allows packets to be blocked in order to prevent them from reaching the target host. Due to hardware limitations, a single device can handle only a limited number of packets per second. This leads either to unmonitored traffic in the network, or dramatic costs increases as additional devices are deployed.

Although an IDS can identify an attack, it cannot block it. Since updating the rule sets of a firewall or IPS takes time and may not be feasible when confronted with an ongoing attack, the network cannot be protected properly in such a situation. By altering the forwarding action of the flow entry from *forward* to *drop*, OpenFlow-based switches can be leveraged to block traffic and hence can be considered as a *distributed firewall*. This allows the network controller to react to the attack in real time by stopping the traffic from the adversary either at the perimeter of the network, or at the closest switch, in order to isolate the attacker.

Apart from analysing traffic on dedicated security appliances, the network controller itself can inspect packets to a certain extent. This can be achieved because the first packet of any flow that cannot be matched to existing flow entries, is sent to the network controller encapsulated in a `Packet-in` message. The packet header information is extracted in order to determine the relevant parameters, which are used to install a new flow entry for this type of packet. The same information about the packet can further be processed by network services and business applications. For instance, *anomaly*

detection algorithms or *white-lists* can be used to determine whether a packet contains malware or has originated from a malicious source. If the result is positive, the switch can block the traffic and prevent such packets from entering the network. A second option is to realise multi-layer security concepts, for instance to forward packets from unknown sources over an alternative route through the network which is separated from sensitive communication data. Developing such applications is again dependant on the northbound API, which is responsible for ensuring that packet header information are accessible from business applications.

3.2 Aspects Covered in this Thesis

The previous section has been intended to give a comprehensive overview of the potential and open questions concerning an OpenFlow-based SDN architecture. Such challenges must tackled in order to put the idea based on a research concept into an operational deployment that is chosen by operators. In this thesis we propose OpenFlow protocol-based solutions to some of these issues.

3.2.1 Network Monitoring and Flow Analysis (Chapter 4)

The detection of faults in OpenFlow-based networks requires an adequate monitoring solution. In addition to the events that are specified by the OpenFlow protocol, network *flows* are an important source of information that can be processed on a centralised control plane. We are interested in determining the flow related parameters that can be considered as appropriate and supportive for anomaly detection or traffic classification. Network flows can also be leveraged to determine the logical topology of a network. This requires the definition of a metric that can be used to analyse the traffic distribution accross the whole network. This information can further be used by network services, for instance to realise a load-balancing application or to detect network failures.

Critical errors can also occur at the switch, with consequences for packet forwarding. The relevant mechanism in this case is the flow table, which defines the relationship between packet header fields and physical ports. Since the number of entries per flow table is limited, attackers may attempt to fill the table with useless entries (MAC flooding attack) in order to enable fail-safe mode, which allows the sniffing of normally hidden traffic. Even though mitigation techniques exist in non-SDN, the consequences of such an attack in OpenFlow-based networks have yet to be investigated. We analyse the extent to which a full flow table can influence the network device and network controller.

3.2.2 Controller-Based Intrusion Detection (Chapter 5)

From a security perspective, SDN allows new ways to defeat attacks by providing increased monitoring capabilities. One aspect concerns the packet inspection that can be realised on the network controller itself. This is reasonable since control messages that

encapsulate the first packet of a new flow are sent to the controller. By analysing such packets, no additional communication overhead is incurred, and the range of such an inspection can cover all protocol layers in order to determine relevant traffic features.

A goal of this thesis is to determine the potential for such controller-based packet inspection through the presentation of typical application scenarios. The inspection mechanism could be realized as network services, for instance a security application that provides on-line protection against a certain type of attack. The information extracted from a particular flow could also be used to apply different policies in order to allow different levels of security. For instance, traffic for a specific host could be routed over a different path from the remaining traffic.

3.2.3 Network Debugging (Chapter 6)

The concept of SDN requires new tools for network debugging and root-cause analysis. This is because all network events are processed in a centralised control plane, which is further separated into different network services. One of the main tasks is the management of network flows, which is realised by the configuration of the network devices and the installation of flow entries. In order to provide complete tracking of a flow entry to the responsible network services, we present a tool that can provide such functionality. It provides an automated approach in order to guarantee reliability, maintainability and scalability.

3.3 Conclusion

The concept of SDN is a new paradigm for controlling and managing computer networks. Even though the flexibility with such an infrastructure is much greater than with non-SDN, the network management categories summarised by FCAPS (see chapter 1) still exist and require appropriate solutions for the SDN scenario. In this chapter we have discussed several challenges that need to be addressed when operators deploy SDN in their operational networks. For instance, bootstrapping of such networks is more difficult than for non-SDN, since each network device is completely dependent on the network controller. The same applies for the control channel, which can either be realised in in-band or out-of-band mode. Due to the centralised control plane, fault management is simplified, since all network events can be processed at a central location on the network controller. Debugging such networks still lacks appropriate tools. Any solution must consider the functionality of the network services, and the events that are triggered from the network devices. From a security perspective, the network controller can apply different policies in order to differentiate the forwarding behaviour of packets in the network. This information can also lead to concepts that dynamically incorporate different IDSs in a flow path, in order to analyse particular communications and to adapt the level of security to the network load.

Chapter 4

Network Monitoring and Flow Analysis

In this chapter, we address the potential of network monitoring based on the concept of network flows. We discuss the relevant flow features, the different levels of flow aggregation, and the potential for fault detection based on a network-wide view. Due to the vulnerability of the flow table, which can store only a limited number of entries, we discuss and show the implications for a DoS attack on a testbed consisting of OpenFlow enabled network devices. Such an attack can be detected by analysing variations in the logical topology, using techniques from information theory that can run as a network service on the network controller.

4.1 Flow Monitoring

The approach of capturing all packets in a network for forensic analysis has been used for decades with much success [84]. However, with current multigigabit speeds, this technique can overwhelm storage capacity. Therefore, the extraction of particular flow properties on the network devices has become popular as an alternative, greatly decreasing the amount of data captured but offering metadata information about the communication between two chosen entities. The analysis of such flow records is also used for digital forensics as to determine the root cause of faults and attacks. Several approaches have been proposed by the industry for the collection and storage of network flows, for instance cFlow (Juniper), NetStream (Huawei) and the *NetFlow*[1] architecture.

4.1.1 NetFlow

The extraction of flow information with NetFlow was originally developed by Cisco, and the current version, 9, is an open standard defined in RFC 3954 [85]. It allows aggregated

[1] http://www.cisco.com/go/netflow

views of network traffic for monitoring purposes.

Universal properties of a flow are called *flow keys* and generally comprise the following components:

{source IP, destination IP, source port, destination port, IP protocol}

Additional fields can be added to allow finer-grained flow definitions. Monitoring flows with NetFlow is a two-step approach: *flow exporting* and *flow collection*. Routers or switches are typically used as flow exporters and are responsible for the metering process, which incorporates four steps:

The flow exporter captures the packet header (1) from each packet and marks each packet with a timestamp (2). The *sampling* and filtering process (3) is used to reduce CPU load and memory consumption by filtering out unwanted information and applying a variety of sampling strategies. Those can either be applied at packet level, considering each packet independently, or at flow level, where packets belonging to a particular flow are processed.

The sampling techniques vary from systematic packet sampling to random sampling. In systematic sampling, packet selection is based on a time interval or fixed number of packet arrivals. Random sampling is often realized with a n-in-N sampling schema, meaning that n packets are selected out of a series of N packets, for instance 1-in-100 or 1-in-1000. The actual network traffic characteristics are estimated from the sampled measurements. Finally, step (4) triggers the updating module, which writes the latest flow status into the flow cache of the flow exporter. A flow record is exported to the flow collector whenever a flow terminates. The flow collector stores the record and allows further analysis, for instance with tools such as `flow-tool`[2] or `nfdump`[3].

NetFlow was developed for non-SDN architectures. In the following we discuss the potential of OpenFlow flow entries for network monitoring on a centralised network controller, which can process such information to create a complete picture of the current network state, and allow network services to adapt and react to that picture.

4.1.2 Flow Aggregation in OpenFlow

The conceptual design of the flow table in OpenFlow (see subsection 2.5.2) allows the level of flow aggregation to be modified. Aggregation defines the range of packets from network traffic that is assigned to a flow entry in the flow table. The granularity of packets belonging to a particular flow entry can be influenced by the configuration of the match fields. For instance, in order to forward every incoming packet on a particular network

[2]http://www.splintered.net/sw/flow-tools/
[3]http://nfdump.sourceforge.net/

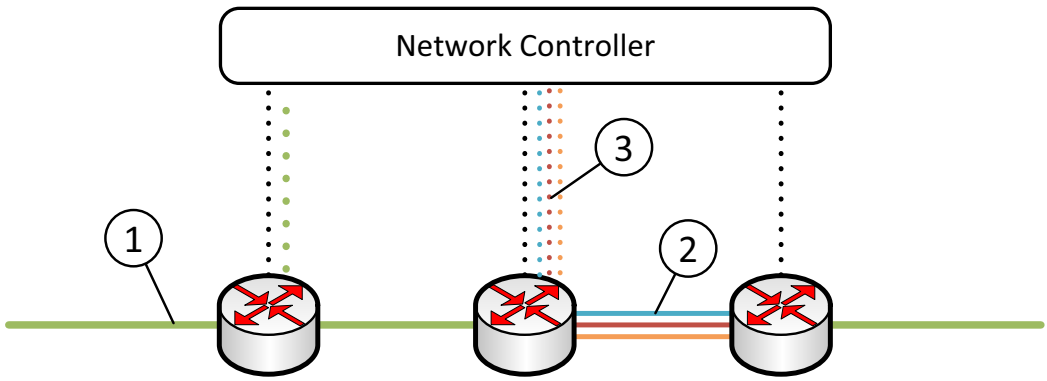


Figure 4.1: Different levels of flow aggregation: OpenFlow allows to dynamically split a macroflow (1) into several microflows (2), which allows finer-grained packet forwarding control and flow statistics, which can be separatly accessed and analysed from the network controller (3).

device to a certain destination host, the source IP address of the flow entry is wildcarded.

Since such a flow aggregation can be controlled by the network controller in real time, several existing types of application can benefit from this:

- Network security: Each switch can act as an access control mechanism for incoming packets, allowing attack-related network traffic to be blocked.
- Load balancing: The forwarding path of packets in the network can be adapted to the current network load.

Apart from such flexibility in packet forwarding, *network monitoring* can profit from inserting flow entries that act as *probes*. As shown in Figure 4.1, a macroflow can be split into three microflows in order to retrieve finer-grained statistics for the network controller. Such splitting can be realised on any network device, for instance at ingress and egress switches in order to determine whether traffic has been altered by a man-in-the-middle attack between the two.

Flow statistics can be requested (polled) for a single flow entry by sending a message to each network device in order to retrieve the applicable information. As an alternative, the statistics can be requested for multiple flow entries in an aggregated form (`OFPMP_AGGREGATE`), or to determine the current status of the flow table (`OFPC_TABLE_STATS`).

One drawback of polling network devices is the increased traffic overhead, especially when faced with a large number of network devices and high update frequency. This can be avoided through the use of `FlowRemoved` messages, where the switch sends a notification to the network controller after removal of each flow entry.

Such a `FlowRemoved` message includes information about:

- the number of packets.
- the number of bytes.
- the flow duration (sec and nsec).
- the reason for removal (removed or expired).

4.2 Flow Features

The flow statistics can provide insights into traffic classification, network security and anomaly detection. Several metrics can be calculated from the flow entries for this purpose:

Packets per Flow The number of received packets per flow entry can be leveraged to determine the type of traffic. For instance, a typical `ping` request is indicated by a new flow entry showing only a very low number of packets. For the detection of a denial-of-service (DoS) attack, which attempts to send an enormous number of packets to a chosen host in order to interrupt or suspend services, similar patterns to the `ping` request can be identified. In order prevent the attacker from being identified, the source IP address is forged — a technique called *IP address spoofing*. Such attack attempts are also flow entries that contain only very few packets per flow, since each sent packet with a spoofed IP results in a new flow and new flow entry.

In order to determine a threshold for classification, [50] proposes the calculation of the median value based on the assumption that normal flows contain a higher number of packets. This is shown in Equation 4.1, where n is the number of flows on a particular switch, and $X = (x_1, ..., x_n)$ is a vector containing the number of packets per flow. The latter must be in ascending order before the median can be computed, so that $x_1 \leqslant x_2 \leqslant ... \leqslant x_n$.

$$\mathrm{md}(X) = \begin{cases} x_{(n+1)/2} & \text{if } n \text{ is odd,} \\ \dfrac{x_{n/2} + x_{(n+2)/2}}{2} & \text{otherwise.} \end{cases} \tag{4.1}$$

The median value of X for a particular switch can be used in order to make an initial evaluation of whether flows are suspicious and should be analysed.

Flow Duration Network flows can be categorised by the total time for which a each flow entry was active in the flow table of a switch before removal. This *flow duration* allows a separation into *short-lived* and *long-lived* flows. The latter are either well-behaved, having a steady rate (e.g. video streams), or bursty with a fluctuating rate — typical for Peer-to-Peer (P2P) traffic.

The relationship between the flow duration d, the idle time-out i, and hard time-out h, with $i < h$, can be used to identify one of the following states:

- $d = i$: The flow either terminated due to a lack of transmitted data, or originates from a DoS attack or network scan.
- $i < d < h$: The flow was terminated by the originating sender.
- $d = h$: The flow entry was removed because of a hard time-out, indicating that the connection is probably still active. In such cases, the removed flow entry is subsequently replaced by a similar one. In order to reduce the frequent installation of similar flow entries, the flow expiration time should be increased for long-lived flows in order to reduce the switch to controller traffic to a minimum.

Reason for Removal The reason for removal indicates whether the flow was removed due to expiration or by the network controller.

Similarity If the number of flow entries with similar destination IP/MAC addresses but different source IP addresses exceeds a certain number for a given interval, it is most probably due to IP address spoofing, indicating an ongoing attack or network scan.

Pair Flows The percentage of pair-flows compared to the total number of flows can indicate a DoS attack or scanning attempt. A pair-flow is defined as two flows that have the same communication protocol, and where the source IP of flow A is equal to the destination IP of flow B and vice versa. The growth or the gradient of the ratio between the total number of flows and pair-flows is an indicator that may signal the start of an DoS attack.

4.3 Centralised Flow Monitoring

As described in the previous section, different metrics can be calculated from network flows to assist in anomaly detection. Nevertheless, flow entries from a single network device contain only limited information about anomalous activities in a network.

By incorporating the information from all network devices, variations in the logical topology can be detected, which can be the result of:

- A DoS attack or network scan.

- A failure of a switch due to a hardware defect or attack.
- A fault in the load-balancing algorithm.
- An interruption of a link, which results in re-routing of flows.
- A fault in the flow installation routine or policy.

In order to determine such variations in the topology of the network, the number of *active* flow entries provides an indicator of the current load distribution. We can use the aforementioned `FlowRemoved` messages to retrieve the current status of flow entries from all network devices for monitoring purposes. The flow properties can be stored in a database system, as described in chapter 6, in order to allow network services to access them.

4.3.1 Distance Functions

The difference between two probability distributions in information theory can be measured using the the *Hellinger distance* or *Kullback-Leibler divergence*, the latter is also known as *relative entropy*.

We consider to two discrete probability distribution, $P = (p_1, ..., p_n)$, and $Q = (q_1, ..., q_n)$.

The Hellinger distance [86] is defined as:

$$H(P,Q) := \frac{1}{\sqrt{2}}\sqrt{\sum_{i=1}^{n}(\sqrt{p_i} - \sqrt{q_i})^2} \tag{4.2}$$

The Kullback-Leibler divergence [87] is defined as:

$$K(P,Q) := \sum_{i=1}^{n} p_i \cdot \log_2 \frac{p_i}{q_i} \tag{4.3}$$

4.3.2 Variations in the Topology

We can measure the variation in the underlying logical topology by comparing two probability distributions $P^{(1)} = (p_1^{(1)}, ..., p_n^{(1)})$ and $P^{(2)} = (p_1^{(2)}, ..., p_n^{(2)})$ from different times.

In order to calculate the probabilities, let a_i^t be the number of active flows for a certain instant in time t on switch i, where n is the total number of switches.

We estimate the time-instant probability $p_i^{(1)}$ and $p_i^{(2)}$, that a new flow is installed on a particular switch, where $t \neq t'$, as:

$$p_i^{(1)} = \frac{a_i^t}{\sum_{j=1}^{n} a_j^t} \quad \text{and} \quad p_i^{(2)} = \frac{a_i^{t'}}{\sum_{j=1}^{n} a_j^{t'}} \tag{4.4}$$

The tracking of the distances of Equation 4.2 and Equation 4.3 over time allows to monitor the network. The rationale is that, for normal traffic with no anomalies, the distances should be small, while anomalies that lead to changes in the logical topology are expected to increase the distance.

4.3.3 Experimental Setup

In order to validate our approach in a realistic scenario, we used a testbed consisting of 8 TP-LINK TL-WR1043ND v1.8 wireless routers with the topology shown in Figure 4.2. This commodity hardware was updated with new Pantou[4] firmware. This modified firmware supports OpenFlow and OpenWrt[5], which is a common Linux-based operating system allowing embedded devices to route network traffic. It incorporates version 1.0 of OpenFlow, and version 10.03.1 (Backfire) of OpenWrt.

The network controller, POX, ran on a Intel i5 PC (2.50 GHz x 4 cores) with 8 GB of RAM and an Ubuntu 12.04 64-bit OS. Two additional hosts with an Ubuntu 12.04 64-bit OS were used to demonstrate different attack and fault scenarios in the network. All network components supported gigabit Ethernet. Due to an implementation issue concerning OpenFlow in the controller firmware, the communication between switch and controller was unstable and resulted in numerous error messages "socket error: broken pipe". After contacting the development site for the controller, we were able to find a workaround by sending `Echo` requests from the controller to the switches. This could be achieved by choosing the betta branch of POX and adding the command `openflow.keepalive --interval=5`.

The following experiments are based on the controller program `l2_learning.py`[6], which provides layer 2 switching capabilities to the network. It enables the controller to build a layer 2 table, which maps the source MAC address to the switch port. If the destination MAC address of an incoming frame is unknown and not included in the table, the switch floods it out to all ports. If the switch port for the MAC address can subsequently be determined by the reply, a flow entry is installed on the switch in order to forward subsequent frames without involving the network controller.

[4] http://www.openflow.org/wk/index.php/Pantou_:_OpenFlow_1.0_for_OpenWRT
[5] https://openwrt.org/
[6] https://github.com/noxrepo/pox/blob/betta/pox/forwarding/l2_learning.py

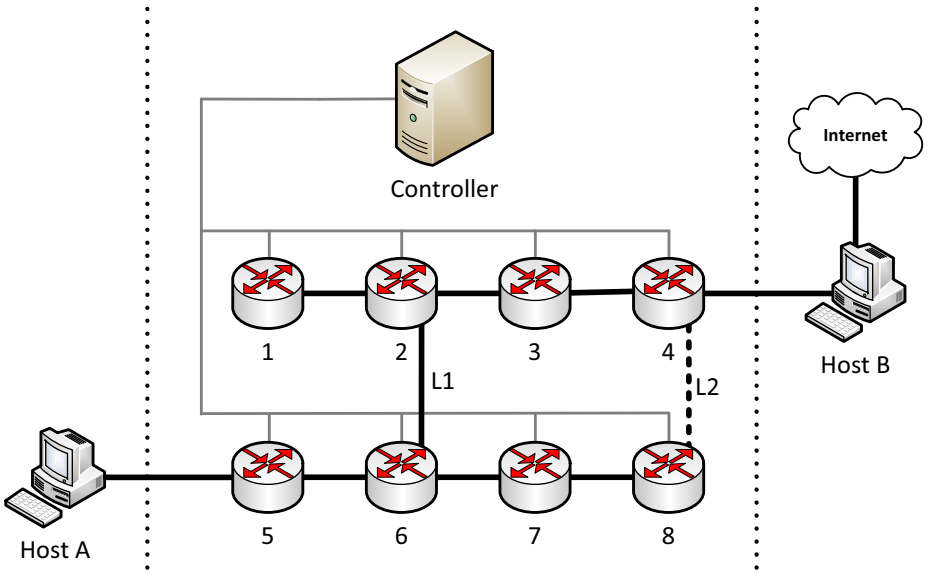


Figure 4.2: Testbed topology of a LAN consisting of two hosts and switches supporting OpenFlow (1-8), which are controlled by a single network controller (POX).

4.4 Denial-of-Service Attacks

The need for protection of the network controller against attacks from within or outside a network is obvious, due to its import role in the network. However, the network devices in OpenFlow can also be targeted by attackers, and so require a protection strategy in order to avoid disturbances in packet forwarding. In the following we address the vulnerabilities in the flow tables in OpenFlow for a Denial-of-Service (DoS) attack, and afterwards discuss detection based on variations in the logical topology, as described in the previous chapter.

4.4.1 Vulnerabilities of the Flow Table

A typical attack against a switch in non-SDN is the *MAC flooding attack*. Each switch contains a table with a limited number of entries that is used to map the binding of the MAC address to the corresponding input port. An attacker who sends packets with a spoofed source MAC address can fill the flow table with useless entries, which makes the switch enter fail-safe mode. The switch then operates simply as a hub and floods incoming packets out to all network ports. This is exploited by the attacker, who can sniff all the traffic that is forwarded by the switch.

The same kind of attack can also be targeted against an OpenFlow-based network device. In contrast to non-SDN, this impacts both the network controller and the network devices. For each incoming packet that cannot be matched to an existing flow entry, a `Packet-in` message is sent to the network controller. This increases the traffic on the

control channel, which can lead to a reduced controller performance. For the network devices, all related packets from the attack must be buffered until the network controller responds with a `FlowMod` message in order to install a flow entry. If the maximum *input buffer* space is reached, this can lead to dropped packets, which should be avoided. The implications of such an attack on the network devices also depends on the implementation of the flow tables that are used for SDN with OpenFlow.

Table Size

We can categorise flow table according to the type of memory that is used to store flow entries. The most common types of memory used in network devices are Binary Content Addressable Memory (BCAM) and Ternary Content Addressable Memory (TCAM):

BCAM This kind of memory is used to store fine-grained flow entries, where all match fields contain a specific value and only an exact match (0 or 1) is possible. For instance, this type of memory is used to store the binding between MAC address and port number in non-SDN switches. Since this type of memory is cheaper, the maximal number of flow entries supported on a switch is much higher when compared to TCAM. For instance, a NEC Univerge PF5820[7] switch with native OpenFlow support can store more than 80,0000 layer 2 flows (BCAM), but only 750 flows when using the 12-tuple OpenFlow match fields (TCAM).

TCAM This type of memory is used to store macroflows, which can contain wildcarded match fields. The rationale is that such memory must determine which of several flow entries matches an incoming packet most closely. Following the match process, the flow entry with the highest priority is chosen and the corresponding action is applied to the packet. Due to the fact that the memory architecture of a TCAM is more complex, the power consumption is increased compared to a BCAM. Since TCAM is a rather expensive type of memory, current switches can store only on the order of a thousand flow entries.

Based on our testbed, we can determine the number and type of flow tables by connecting to the network devices (TP-LINK). As displayed in the following, the first table, `0: hash2` is used to store up to 32,768 fine-grained flow entries, while the table `1: linear` uses a memory structure similar to TCAM and allows up to 100 wildcarded flow entries.

```
root@OpenWrt:~# dpctl dump-tables unix:/var/run/dp0.sock
stats_reply (xid=0x4f67d5c2): flags=none type=3(table)
 2 tables
  0: hash2   : wild=0x00000, max= 32768, active=0
               lookup=329284, matched=61823
  1: linear  : wild=0x3fffff, max=   100, active=0
               lookup=267461, matched=584
```

[7] http://www.necam.com/Docs/?id=ba0dadc4-f253-4a8a-b27a-a791378f9acf

Particularly for table 1, the maximum number of flow entries can be reached in a very short time if the flow installation policies require the installation of macroflows, as these need wildcarded fields. The maximum number of flow entries that can be installed per second, R, is defined by the flow table size n, and by the time-out t determining the duration for which a flow entry occupies memory in the flow table.

$$R = \frac{n}{t} \tag{4.5}$$

For table 0, the default POX time-out of 10 s limits R to 3,276 flow entries per second.

Packet Delay

Since table flooding is a major threat to an OpenFlow network, an attacker can exploit this vulnerability by using a DoS attack. There is a wide spectrum of DoS attack types targeted at particular goals, for instance to consume resources or to alter a configuration. Different attacks focus on different attack targets (e.g. application layer or operating system) and the weaknesses that they exploit. Since our goal is to fill the flow table to the maximum, we concentrate on attacks that aim to create new flow entries.

For our experiment, we used the network topology of Figure 4.2 with host `A` as the attacker and host `B` as the victim machine. Both were able to communicate with each other over link `L1` and the network devices 5-6-2-3-4. In order to simulate background traffic in the network, host `A` could access the internet over host `B` and generate web traffic using Hypertext Transfer Protocol (HTTP) requests.

In order to accomplish the DoS attack, we wrote a Python[8] script that sent packets with randomly-assigned destination ports (0-65535) over User Datagram Protocol (UDP). This forced the controller to install a separate flow entry for each destination port, and resulted in a large number of flow entries and `Packet-in` messages.

An important parameter of a flooding attack is the packet size. Whereas smaller packets aim to suspend services on the destination machine, a larger packet size is used to exhaust the available bandwidth. For our testbed we determined that host `A` could send approximately 143,000 packets per second for a payload size of 40 bytes (82 bytes per frame), and 100,000 packets per second for a payload size of 1,024 bytes (1,066 bytes per frame).

Because the delay between sent packets should be minimised in order to increase the impact on the target machine, we observed that the number of installed flow entries on the network devices depended on the *delay* that was added between a sequence of sent

[8] A high-level programming language.

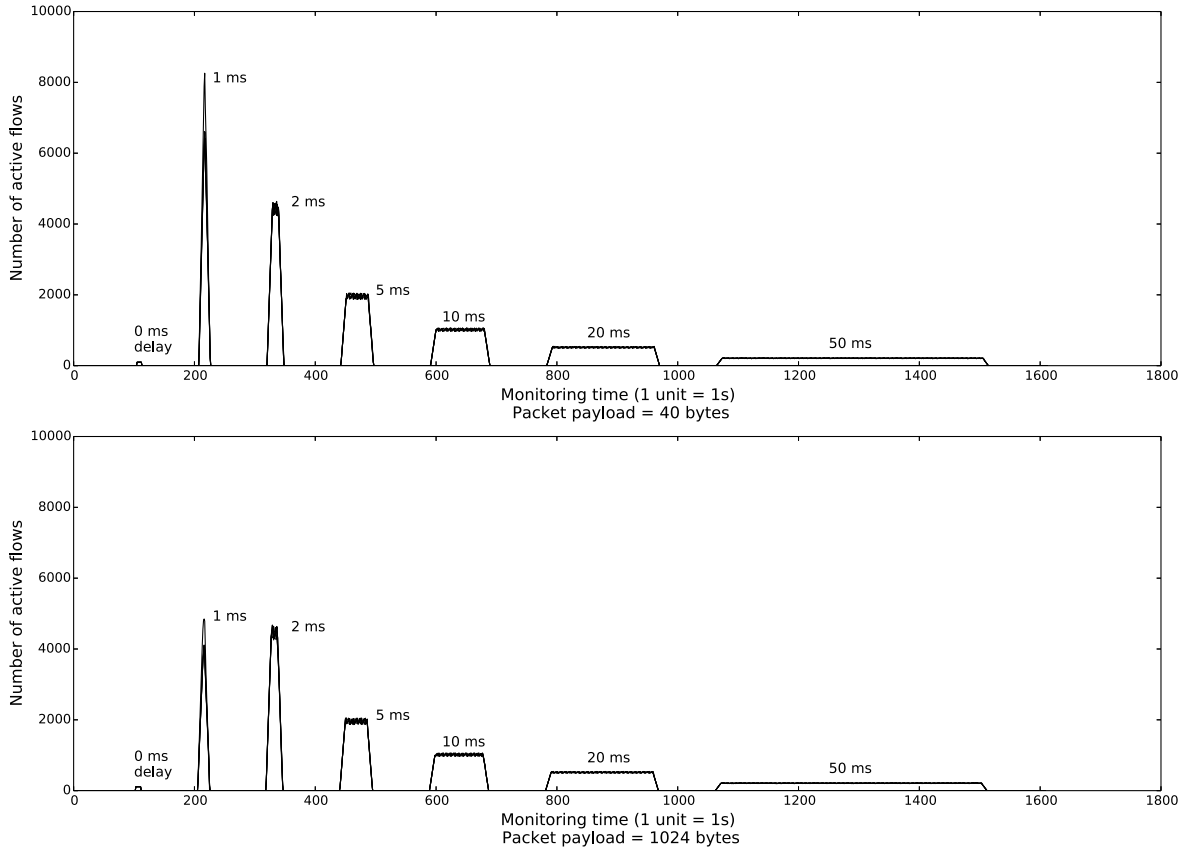


Figure 4.3: Number of active flows during UDP flooding attacks, each with 10,000 packets and 40/1024 byte payload. The delay between packets for each sequence is 0, 1, 2, 5, 10, 20, and 50 ms.

packets. In order to determine this relation between the number of active flows and packet delay, we run an experiment to measure the number of active flows on switch 5, since it was the closest to the attacker. We performed seven attacks, each with 10,000 packets, adding delays of 0, 1, 2, 5, 10, 20, and 50 ms between the sent packets in attacks one to seven respectively. As shown in Figure 4.3, the number of installed flow entries on all network devices is maximal when a delay of 1 ms is added between successive packets (see monitoring time = 210).

The explanation is that, when there is no delay, the overloaded network device was not able to handle the large number of unmatched packets, and was able only to install a few flow entries (see monitoring time = 110). In addition, a larger packet size of 1,024 bytes further reduced the number of flow entries when compared to a smaller packet size of 40 bytes. This is because the input buffer of the switch has only a limited storage capacity, determined by the number of packets multiplied by the payload.

Results

Based on our previous observations, we chose a packet delay of 1 ms for the following experiments in order to set the number of installed flow entries to the maximum. We executed 20 attacks with each 10,000, 50,000, and 200,000 packets and varying payload. The durations for each of the attacks were 12 s, 58 s, and 233 s respectively.

We modified the flow installation routines of the POX controller in order to install flows either in flow table 0 or in 1. For flow table 1, which is limited to 100 entries, it was possible to reach the maximum number of flow entries in a very short time. Subsequently, additional flow installation attempts were blocked by the network device with the following error message:

```
[00-23-20-6a-9d-56 13] Error: type: OFPET_FLOW_MOD_FAILED (3)
[00-23-20-6a-9d-56 13] Error: code: OFPFMFC_ALL_TABLES_FULL (0)
```

For flow table 0, the maximum number of flow entries is shown in Figure 4.4, and varies around a value of 8,000, independent of the duration of the attack. We determined by experiment that host A sent around 850 packets per second to host B. Since the maximum number of flow entries per second that can be installed is defined by $E_r = 3,276$, we increased the impact of the attack by connecting a second attacker simultaneously performing the same flooding attack as host A to switch 5. This did not increase the maximum number of flow entries and kept varying around 8,000 active flow entries.

We believe that the bottleneck is the limited size of the input buffer. After its capacity is filled with buffered packets that cannot be forwarded due to a missing reply from the network controller, the network devices drop additional packets and thus avoid the installation of additional flow entries.

We observed that, during the attack, the CPU load of the switches was increased but did not reach 100 %. A manual inspection of active flows on all involved switches showed that switch 5 was suffering the most, while the switches 6-2-3-4 were less affected by the attack. This is shown in Figure 4.4 for the number of active flows, which is approximately 2,000 entries lower in case for the attack duration of 50,000 and 200,000 packets. We assume that, because the performance was heavily impacted on switch 5, some flows never reached the next hop. Furthermore, we observed that the attack with 200,000 packets made switch 5 crash and reboot in some rare cases.

4.4.2 Attack Detection

In order to detect the aforementioned DoS attack on the network controller, we calculated distances, as proposed in section 4.3. Both the Hellinger distance and the Kullback-Leibler divergence are shown in Figure 4.5. While the start and particularly the end of most attacks are indicated by peaks in both distance measures, the Hellinger distance shows a greater level of noise during non-attack periods.

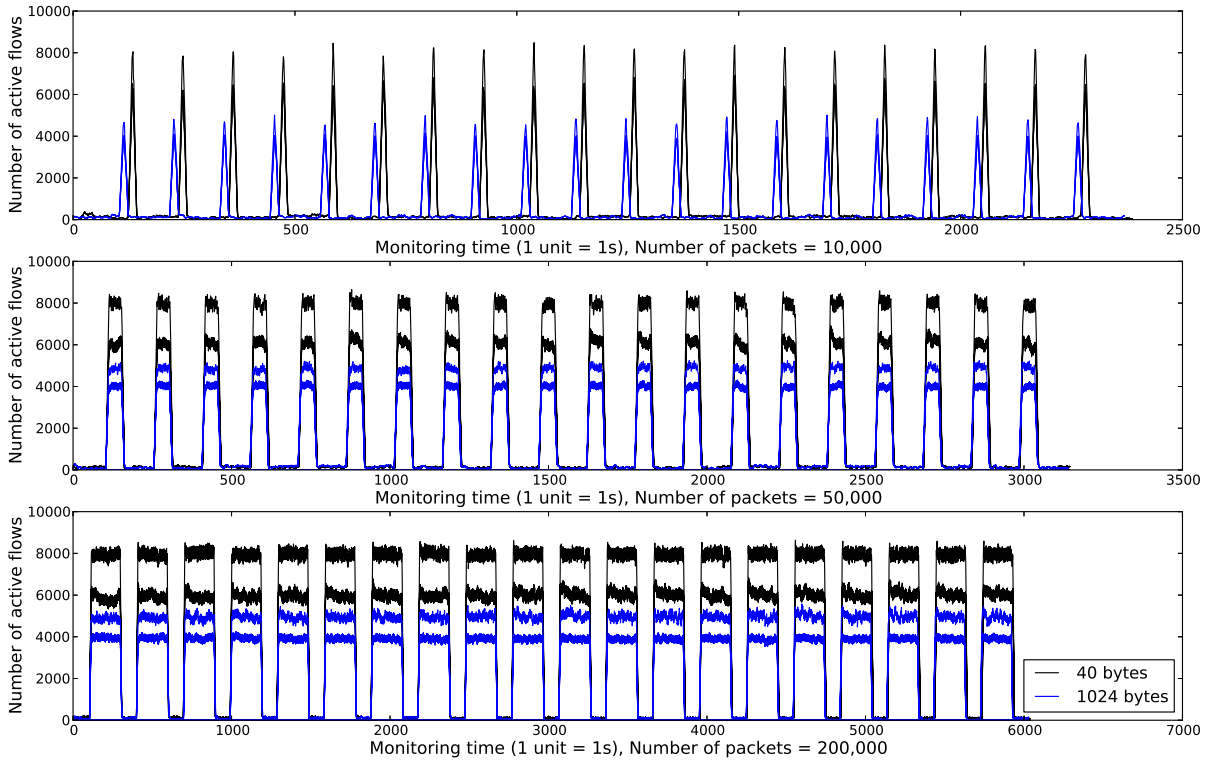


Figure 4.4: Number of active flows for 20 UDP flooding attacks with varying number of packets and payload.

Control Charts

To automatically detect an attack using the computed distances, we used *control charts* [88], which are deployed in statistical process control to determine if a process variable is within certain limits. An example of a control chart is shown in Figure 4.6. A human operator can observe the current and past trend of the data and is notified if a control limit is exceeded, as indicated by a red point.

Both limits, the upper control limit (UCL) and a lower control limit (LCL), are defined in Equation 4.6 and allow the chart to adapt its limits to a rise or fall of the process variable (i.e. distance):

$$\begin{aligned} \text{UCL} &= \mu + k\sigma \\ \text{LCL} &= \mu - k\sigma \end{aligned} \tag{4.6}$$

where μ denotes the mean of the historical data, and σ denotes the standard deviation of the past data. The constant k is defined as a factor of the standard deviation σ, where $k = 3$ is an accepted standard in industry. Alternative values of k can be chosen using

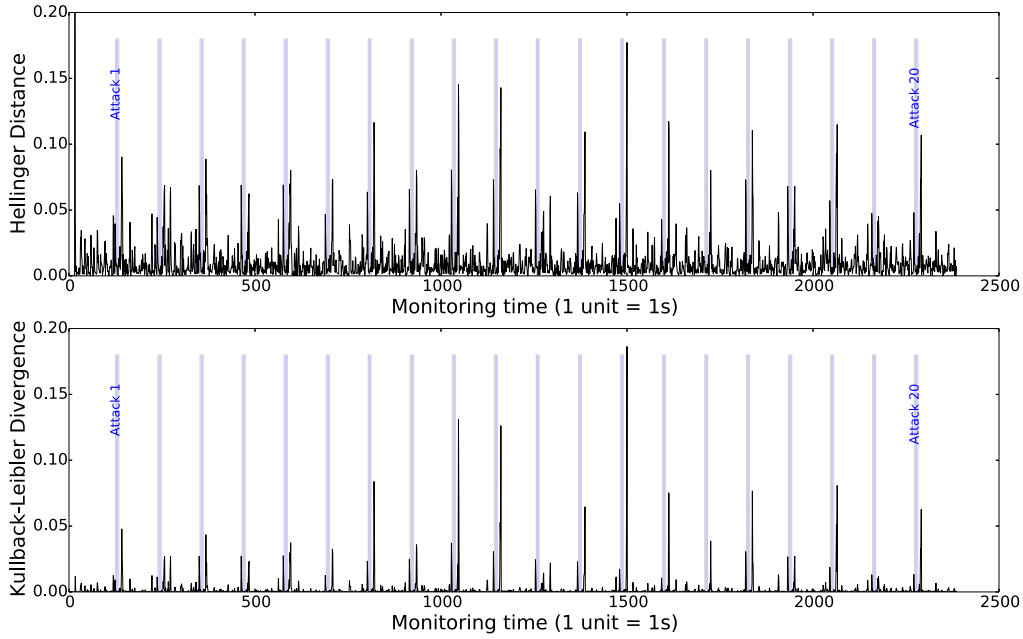


Figure 4.5: Hellinger distance and Kullback-Leibler divergence for an attack sequence with each 10,000 packets.

the Lucas and Saccucci [89] tables. We used a sliding window of fixed size that was used to calculate the the historical data.

Accuracy Measure

In order to determine the performance of the control charts for classification, we can assign each classified data point as either a *true positive* (tp), *true negative* (tn), *false positive* (fp) or *false negative* (fn). Since the factor k influences the classification results by altering the Upper Control Limit (UCL) of the control chart, we use *Precision* and *Recall*, two statistical measures that show the quality of a binary classification test.

The recall (R) and the precision (P) are defined as:

$$R = \frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fn}} \tag{4.7}$$

$$P = \frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fp}} \tag{4.8}$$

These measures can be plotted as Receiver Operating Characteristic (ROC) curves, which are graphical plots for a classifier system in which the discrimination threshold is

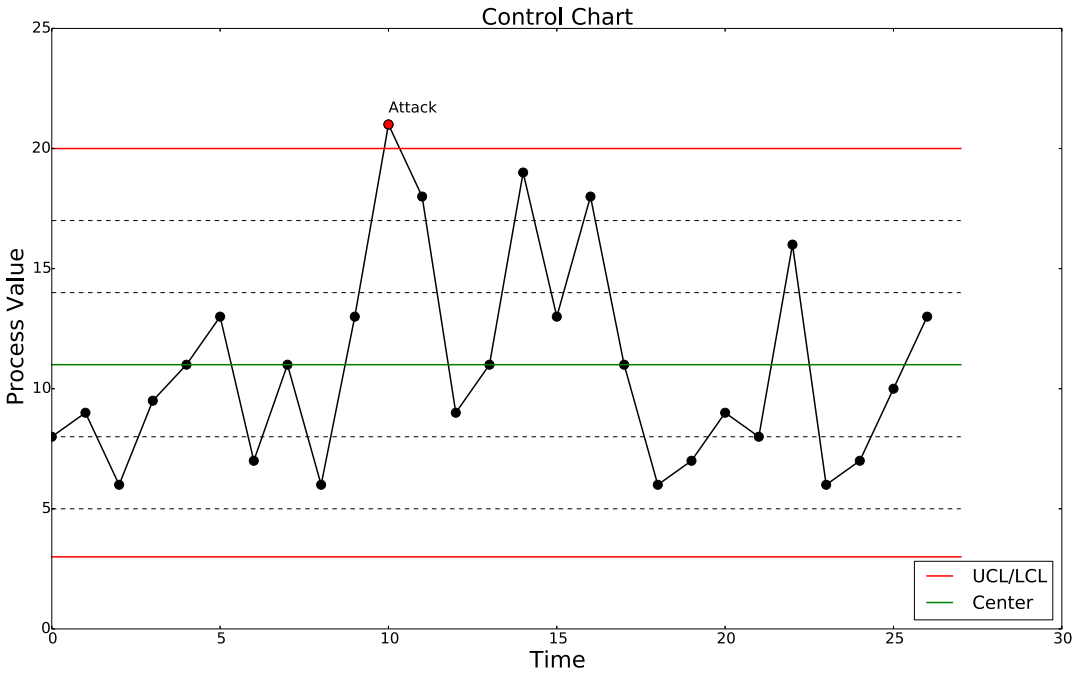


Figure 4.6: Control chart with a distance above the upper control limit (UCL), which was classified as an attack.

varied. The x axis shows $1 - precision$, and the y axis the recall or sensitivity. In our case, the factor k is used for the discrimination threshold and varies from $0.01, ..., 7\sigma$.

As shown in Figure 4.7, the number of monitoring points covered by an attack period depends on the monitoring frequency. Since the definition of recall and precision requires the number of true positives and false negatives to be known, the number of monitoring points within an attack period must be reduced to a single data point. The rationale for this is that, otherwise, monitoring points below the UCL within an attack period are counted as false negatives. For instance, the left control chart shown in Figure 4.7 would result in a recall rate of approximately 33 %, even though the attack was detected by exceeding the upper limit.

In order to reduce the number of monitoring points to a single point per attack, we processed the data with the following techniques:

1. Moving Average Filter: This filter is used to remove noise and smooth the data points.

2. Non-Maxima Suppression: This technique determines the local maxima of data points in a certain range, and reduces the total number of distances. The range of data points that are taken into account can be controlled by a parameter, which defines the number on either side to use in the comparison.

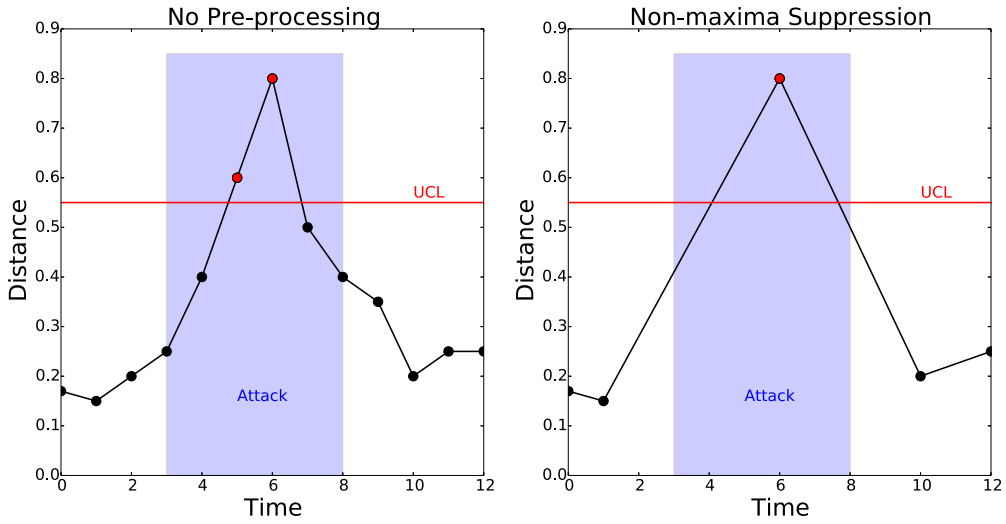


Figure 4.7: Example of a control chart with no pre-processing, and a reduced number of data points after using Non-Maxima Suppression, which shows a single data point per attack period.

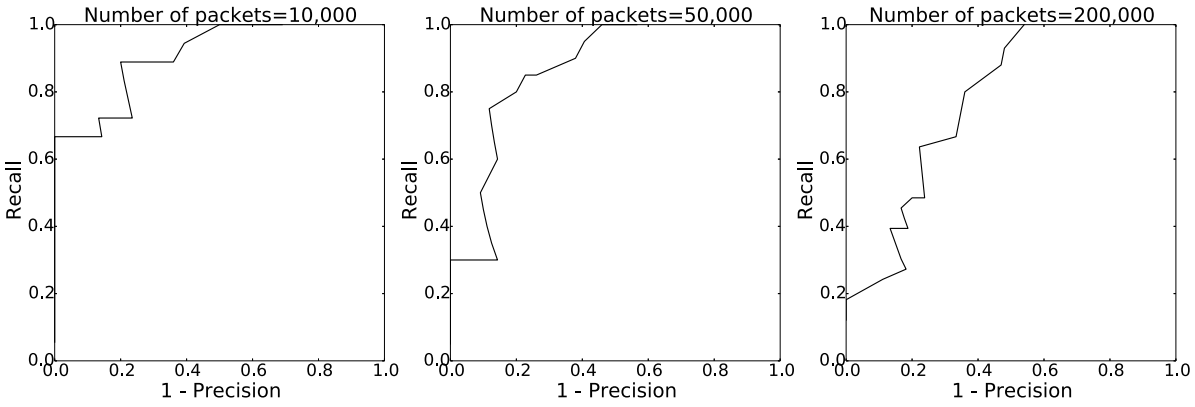


Figure 4.8: ROC curves of the Kullback-Leibler distance for 20 attacks with different number of packets.

After applying steps 1 and 2, the reduced number of data points is suitable for classification with control charts. As shown in the ROC curves in Figure 4.8, the performance is quite promising for the attacks with 10,000 and 50,000 packets, but decreases for the attack with 200,000 packets. The rationale for this is, that, for a longer attack duration, the number of data points per attack is always greater than one and the second and subsequent points are likely to be below the UCL, and so classified as false negatives.

4.4.3 Variations of the Traffic Load

While sudden changes of the traffic load in a network may indicate an attack, they can also be the result of a broken communication link, which can fragment a network, making one or several subnets no longer reachable. To mitigate such errors, switches in the higher tiers of a network are usually redundant and can redirect affected traffic over an alternative link. Since loops within a switched network cannot be tolerated because they can result in packets becoming stuck in the loop forever, the spanning tree protocol (STP) [41] avoids this kind of situation by preventing traffic from traversing duplicated links.

As well as being informed about broken links, the network operator must be notified when there is a burst of traffic or if the routing paths have changed. While a traffic burst might be caused by a scheduled backup of a database system, similar behaviour would also ne seen for an insider attack attempting to download internal documents to a host. Based on the centralised flow monitoring approach, we were able to detect such variations by tracking the distance with control charts.

In order to force traffic re-routing in our experimental network, we created a loop by adding a second link `L2`. Based on the topology of Figure 4.2, we used `spanning_tree.py`[9] controller program to keep only one of links `L1` or `L2` active. In order to allow packet switching on layer 2, the program uses the `l2_learning.py` submodule. At the outset, both links `L1` and `L2` were physically connected in the topology and host `A` was connected to the internet over host `B`. Following the disabling of `L1` or `L2` to simulate a link failure, the spanning tree algorithm activated the alternative link to take over the communication. Over a period of 30 minutes with simulated background traffic, we simulated 10 link failures, each resulting in a new spanning tree that enabled either `L1` or `L2`. The sudden deviations in the distribution of the network traffic could be detected with our approach, which showed a greater distance both when a link went down and after activating an alternative link. The results based on a classification with control charts showed a high recall rate of 90 %, combined with a precision of 70 %.

4.5 Protection Scheme

As discussed in the previous section, protection against DoS attacks in OpenFlow-based SDN is important since they can affect such a network in several ways:

- Network performance is decreased due to the huge traffic burst.
- Packet forwarding on network devices is limited due to full flow tables.
- The performance of the network controller is decreased, because it has to cope with a large number of `Packet-in` messages.

[9] https://github.com/noxrepo/pox/blob/betta/pox/samples/spanning_tree.py

An ongoing DoS attack can be detected either by tracking the Kullback-Leibler divergence, or by calculating relevant traffic features, as shown in section 4.2. In order to prevent DOS attack-related traffic from entering the network, the *flow installation policy* should adapt to such incidents. In order reduce the number of `Packet-in` messages, wildcarded flow entries should be installed in order to drop the traffic at the switch closest to the attacker. For the DoS attack scenario of subsection 4.4.2, the following flow entry (pseudo code) on switch 5 could be used to block the traffic:

```
src IP = "*" | dst IP = "IP address host B" | action = "drop"
```

The drawback of such a rule is that all traffic, whether good or bad, targeted to the victim host is blocked. Nevertheless, this defence mechanism can be automatically activated over a suitable network service on the network controller without requiring the manual installation of a specific firewall rule.

4.6 Conclusion

In this chapter we have discussed the potential for network monitoring in OpenFlow. Its flexibility in creating microflows and macroflows allows different levels of flow aggregation to be defined, these can be leveraged to inject flow entries as probes in real time. For instance, it allows finer-grained flow statistics to be retrieved for flows that need to be analysed due to security concerns. In the same way, different features can be extracted in order to determine traffic properties of interest. By varying the current number of active flows, we have used a realistic testbed to show the implications of a DoS attack for the flow table of network devices. In addition, we could detect the attack by measuring the variations in the logical topology using different distance functions. In order to determine the degree of similarity between probability functions at different points in time, we compared two distance functions, which we classified with control charts. We also evaluated the success of the control chart approach by determining recall and precision for different parameter settings.

Chapter 5

Controller-based Intrusion Detection

This chapter addresses network security concepts that take advantage of a programmable network based on OpenFlow. We introduce an approach for packet inspection on the network controller based on a network-wide view. This allows the flow metadata to be processed, leading to the implementation of of a network firewall as a network service on the network controller, allowing the detection of network attacks in both on-line and off-line modes. In a second application scenario, we address the detection of hidden communication patterns that are used to activate a stealthy backdoor in the network. The contribution at this chapter has been published in [90], [91] and [92].

5.1 Intrusion Detection

Network security depends mainly on two important factors: the *network protocols* that are used to communicate between entities and the *network topology* [93]. The topology determines which hosts can be reached over a particular link in the network, and can be controlled through flow installation policies located on the network controller. This is reasonable, since the controller's network-wide view allows routing decisions to be made based on knowledge of the topology, which includes the number of active links and the costs involved in reaching a destination.

Network protocols, they can be split in two categories. The basic communication protocols (e.g. Internet Protocol (IP)) are used to exchange data between hosts and do not depend on the underlying network infrastructure. The second category consists of protocols that are used for network management. Since some of these are standardised, the majority are vendor-dependent and allow network operators to control and configure their network equipment. Although vulnerabilities in the basic communication protocols are well-explored [94], network configuration protocols still lack important security features. This allows intruders to abuse them to modify the network configuration, and also

to gain insights into the network configuration by sniffing network traffic. Since some network management protocols are no longer required in SDN, the possibility for such attacks is reduced.

The detection of and protection against network attacks is the responsibility of security appliances, which are placed at different positions in the network topology. There exists a wide variety of commercial products, all offering functionality based on the following approaches, which work at the packet level.

Firewall A network firewall permits or refuses data packets to enter or exit the network perimeter by matching packets to a user-defined rule set. Stateless firewalls analyse each packet independently without incorporating information about similar packets that belong to the same flow. In contrast, stateful firewalls keep track of each session (i.e. TCP and UDP) and reduce the need for processing power by avoiding expensive lookups for each packet. While the first generation of firewalls operated only at the network layer, modern firewalls also incorporate the application layer to allow fined-grained rules such as the definition of allowed protocols and services.

Intrusion Detection System (IDS) Such a sensor or monitoring system is usually deployed within a network to protect against insider attacks. All ingress traffic is analysed using a variety of techniques, although signature-based methods prevail among anomaly detection methods. After an ongoing attack or attempt to circumvent security measures has been identified, an alert is raised. However, countermeasures are not provided for, and require the deployment of additional security appliances.

Intrusion Prevention System (IPS) Although similar to an IDS, which raises an alert in case of an incident, an IPS is connected in-line with the network traffic and is able to stop particular flows by blocking such connections. An IPS must be able to cope with changes in the traffic load, and also have the ability to show some resilience against network attacks, for instance a DoS against a host machine. IPSs themselves are often directly targeted by attackers in order to separate parts of a network in case of a link failure or to enable fail-safe mode.

All the described systems allow system events to be exported to a *log file*. This is possible not only for the triggered rules, but also to construct a history of all connections that occurred within a certain period of time.

5.2 Packet Inspection

As described in chapter 4, the flow entries located at the flow table of a network device can be leveraged for network monitoring. While this allows certain flow properties

and statistics to be retrieved, it lacks the possibility of determining more detailed information about the transmitted packets itself (e.g. protocols involved or payload). Such information is desirable because it can be used by network operators in order to make better routing decisions. For instance, it allows traffic prioritisation or the application of different security policies for different senders.

A solution to the problem of retrieving such packet-based information is to analyse the packet encapsulated within each `packet-in` message of the OpenFlow protocol. Such a message is sent for each flow that cannot be matched to a switch's existing flow entries. It contains a copy of the first packet, which is used by the network controller in order to determine the appropriate parameters for the flow installation process. The same information can additionally be used on the network controller in order to realise new means of intrusion detection.

The structure of a `packet-in` message with an encapsulated packet is shown in Figure 5.1. The example packet was sent from a programmable logic controller (PLC), which is used in automation processes to control machinery. The communication between PLCs, sensors, and actuators[1] requires protocols that support real-time mode (e.g. ProfiNet[2]). Since all parameters of the protocol stack are accessible from the network controller, network applications can use this information to define more sophisticated network policies than can be realized in non-SDN networks. In order to secure such PLC communication in the network, the following policy could be defined:

1. Extract the MAC address of the sender S (`Siemens_13:38:7a`[3]).
2. Check if S uses the ProfiNet protocol.
3. Determine the Quality of Service (QoS) required by S (e.g. by using a whitelist).
4. Determine if S is authorised to access services at the destination host (i.e. port).
5. Calculate a path to destination depending on the QoS requirements of S.
6. Install flow entries with the appropriate action (i.e. drop or forward) on the network devices.

Going beyond inspection of the packet header, the network payload can reveal important hints about the type of traffic. The payload can either be inspected in order to determine information about proprietary protocols, and to avoid the infection of the destination host in case of malicious content. The degree of payload inspection possible with a single packet depends on the transport layer protocol. In case of UDP, a single packet might contain enough information to make an assumption about the content or

[1] Converts energy into motion, for instance an electromechanically operated valve.
[2] http://www.profibus.com/
[3] The first part of the MAC address was replaced by the manufacturer prefix.

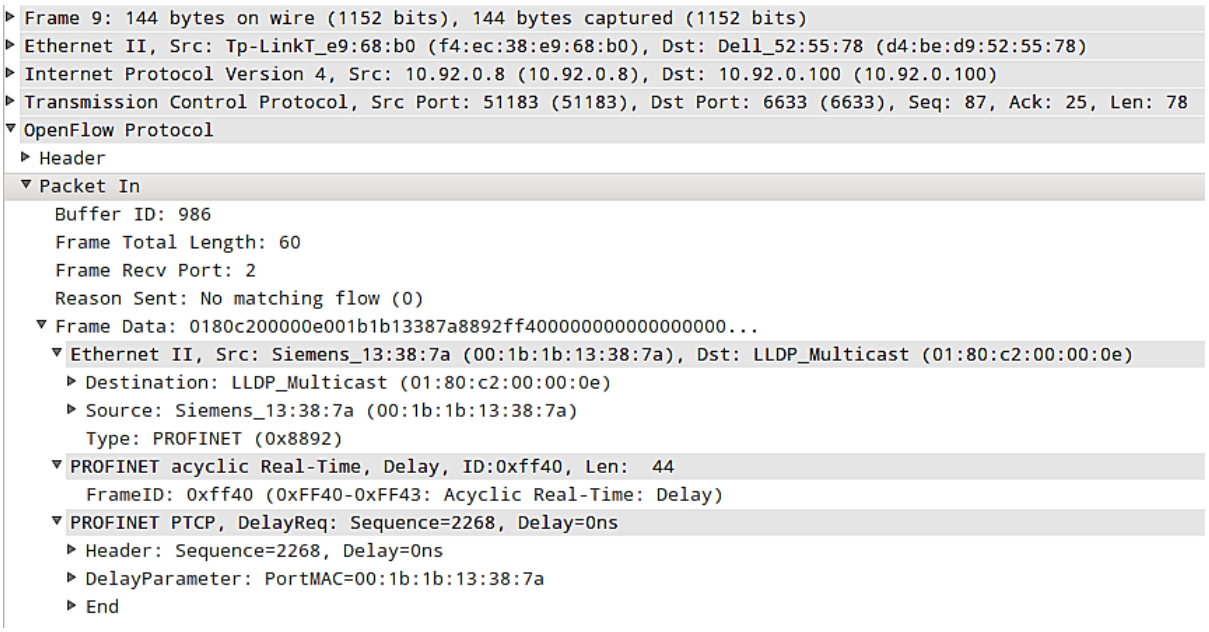


Figure 5.1: An example of the protocol stack of a `packet-in` *message in OpenFlow. It allows the inspection of the first packet of a new flow on the network controller, which can be used to define finer-grained security policies.*

purpose of the message. For TCP-based connections, a single packet is unlikely to be meaningful, since more packets are necessary in order to perform TCP message assembly.

In order to demonstrate the feasibility of our approach, we have propose two scenarios for an OpenFlow-based SDN, as shown in Figure 5.2. They combine the global network view of the controller with the ability to have network services that can operate at packet level. In the first scenario, we focus on a network firewall application that is used to determine the number and type of attacks that a network is exposed to. This is realised by storing flow-based communication data on the network controller, which requires (1) extracting the relevant packet properties (e.g. IP address, protocol) from the `packet-in` message, and (2) determining the type of attack using firewall signatures, where the triggered rule is described by an ID.

Since the number of stored records can become very large in an operational network, we propose an automated approach to process this data in order to detect network attacks. More precisely, we are interested in identifying attacks that result in a block of similar records in a log file. In order to detect such events, we present two approaches: *on-line* and *off-line* detection. While the on-line mode allows real-time detection and protection and could be realized as a network service on the network controller, the off-line mode allows the inspection of archived log files. Both modes are further discussed in section 5.3.

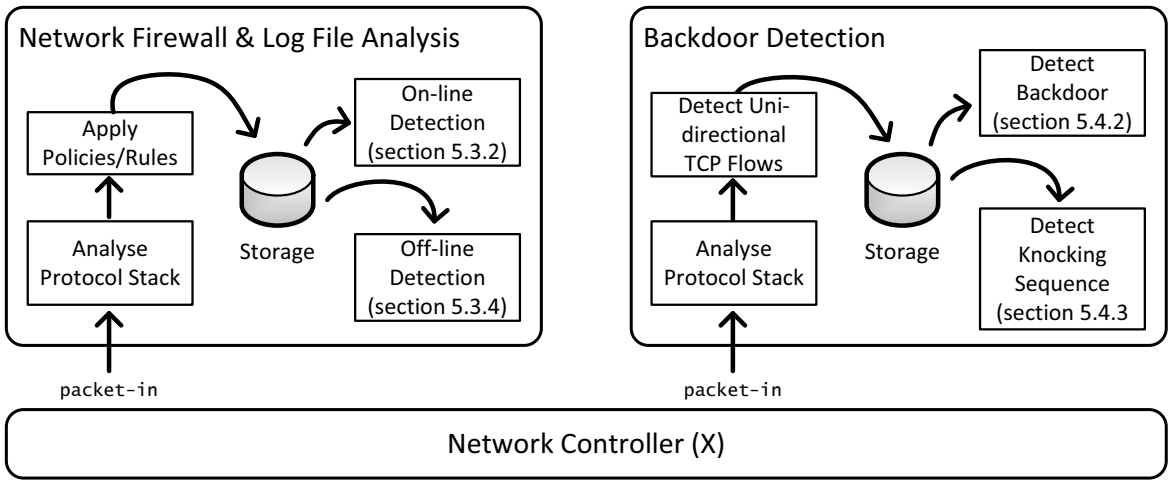


Figure 5.2: Two scenarios for controller-based intrusion detection, which analyses the packets encapsulated in a `packet-in` *message.*

A second scenario focusses on the detection of packets that are part of the knocking sequence, required to activate a stealthy backdoor. Currently, there is no approach that can tackle such an attempt, due to the difficulty in detecting the relevant packets. Since each new port knock results in a message that can be processed on a OpenFlow-based network controller, we can determine the ports involved by analysing such uni-directional flows. In order to determine the port sequence, we apply an algorithm developed for rare association rule mining. Both approaches are described in section 5.4.

5.3 Analysis of Communication Records

We show in the following how anomaly detection can be leveraged with SDN in order to identify abnormal communication patterns. In order to perform such an experiment with realistic data, we used the firewall records from a dataset generated by Checkpoint[4] firewall, which is used by an Internet Service Provider (ISP) in order to protect internet-connected hosts against network attacks.

The storage of such firewall records can serve different purposes:

- Improvement: The configuration of security appliances must be adapted to the specific network environment and security regulations. By analysing the triggered warnings or alerts of an Intrusion Detection System (IDS) in a log file, firewall rules can be updated in order to stop attackers at the network perimeter, so preventing future intrusions.

[4]http://www.checkpoint.com/

- Assessment: Network operators can determine the number and type of attacks that a particular network is exposed to. This can result in adapting configuration rules for the observed attack types, or in the deployment of additional security infrastructure.

- Forensics: If a company or institution has been targeted by a professional attacker, it is important to determine the kind of attack and compromised hosts. Since log files can also be used as an evidence in a lawsuit, the correct deployment and configuration of logging facilities (e.g. ensuring accurate timestamps) is mandatory.

The auditing of large data files faces the challenge of the ever-increasing volumes of data to be processed, and thus requires both processing facilities and conceptual solutions to cope. In incident response scenarios, a human analyst is required to process a large quantity of log data in order to find suspicious activities and correlate them with additional pieces of evidence. In many cases, this incident response is triggered off-line after some additional facts have been determined. Performing on-line incident detection is very difficult. For example, the many application-specific log formats require deep domain-specific knowledge in order to properly configure existing rule-based event correlation engines. Secondly, either a precise misuse model has to be provided, or efficient detection algorithms are required.

Since the analysis of such files is a very tedious task and often neglected, most attack attempts are never known to the operators. By using a centralised network controller in OpenFlow, connection-related information can be determined on the network controller by targetted security policies and analysed by a dedicated application in real time. We address these issues in our approach and propose a general framework that has been implemented for the specific case of firewall log files.

In the first phase, a sliding window over the log file compares successive windows by applying a set of appropriate distance functions. The observed time differences are then analysed with statistical quality process control rules for on-line detection. In order to analyse archived log files, we propose the concept of label propagation for off-line detection. However, the described methods are general, and can be applied to many other types of log formats. Since many log formats include similar types of data (e.g. IP address, timestamps and objects) and comply with the generic information model described in the seminal paper [95], a simple data type-based conversion should be sufficient to allow our methods to be applied.

5.3.1 Window-Based Approach

We consider all accepted or dropped connection records from a firewall log file as data that must analysed. The typical attributes and values from an example firewall record are shown in Table 5.1. We consider the attributes a_1 to a_9 to be important for anomaly detection.

Attribute	Example	Metric
Number	4237	-
Date	16May2011	-
Time	0:05:11	-
Type	Log	-
Source Port (a_1)	49954	J
Rule (a_2)	298	S
Current Rule Number	298-Standard	-
Information	-	-
Product	VPN-1 Power/UTM	-
Interface (a_3)	eth-s1/s1p1c3	S
Origin (a_4)	IP1220-Gare1	S
Action (a_5)	Drop	S
Service (a_6)	694	S
Source IP (a_7)	192.168.8.183	J
Destination IP (a_8)	192.168.8.255	J
Protocol (a_9)	UDP	S
Rule Name	-	-
User	-	-

Table 5.1: Example of a Checkpoint firewall log file record. Relevant attributes are assigned to distance metrics (S defined in Equation 5.1 and J defined in Equation 5.2) for detecting outliers in a series of records.

In an initial step, we bundle log file records into chunks, or *windows*, for two reasons: firstly, to reduce the amount of data, since the number of firewall log file records produced by a company or ISP can be huge due to the large number of network attacks and blocked connections; and secondly, to incorporate the dependencies from temporally-related events, since a typical attack attempt (e.g. network scan) often results in a series of blocked connections and log file records which all relate to the same attacker.

Each window w can be built in two different ways. A window can contain either a fixed number of records, or all records that occurred within a certain time period. Since the time of occurrence is not considered to be important in our approach, we chose to use a fixed-length window. This also avoids the empty windows that can occur during attack-free intervals, or during periods having no traffic.

To determine whether a window is anomalous or normal, we calculate the distance measures between two consecutive windows w_i and w_{i+1}.

Distance Measures

We used the Kullback-Leibler divergence, as defined in Equation 4.3, to calculate the distance between two probability distributions $P = (p_1, ..., p_n)$, and $Q = (q_1, ..., q_n)$. Since the Kullback-Leibler divergence is not *symmetric*, we defined the distance $S(P, Q)$ as:

$$S(P, Q) := K(P, Q) + K(Q, P) \tag{5.1}$$

If both distributions are exactly the same, we find that $S(P, Q) = 0$.

We choose the *Jaccard similarity coefficient* to give an estimation of the similarity between two sets A and B. It is defined by the size of the intersection divided by the size of the union of both sets.

$$J(A, B) := \frac{|A \cap B|}{|A \cup B|} \tag{5.2}$$

If both sets are identical, we find that $J(A, B) = 1$.

Histogram

In order to compare two consecutive windows, the values from each attribute (e.g. "TCP") in the first window are compared with the corresponding attribute in the second. Attributes differ in data type (e.g. number or string) and range (e.g. source port and protocol). Therefore, we either calculate the distance $S(.,.)$ or the Jaccard similarity coefficient $J(A, B)$.

To calculate the distance $S(.,.)$, we consider the attribute in the first window as a discrete probability distributions $P^{(1)} = (p_1^{(1)}, ..., p_n^{(1)})$, and the attribute in the following window as a discrete probability distributions $P^{(2)} = (p_1^{(2)}, ..., p_n^{(2)})$.

The probabilities $p_i^{(1)}$ and $p_i^{(2)}$ denote the frequency of occurrence of each value in the histogram.

The distance $S(P^{(1)}, P^{(2)})$ is not suitable for an attribute that can have a large number of different values (e.g. IP address), due to the large number of different bins in the histogram. For such attributes, we compared the similarity using the Jaccard similarity coefficient. The set A denotes all values of an attribute in the first, and the set B the respective values from the second window.

Distance Between Windows

In order to describe the similarity between two windows with a single value, we define a distance d:

$$d = \sum_{(P,Q)\in\mathbb{V}} S(P,Q) + \sum_{(A,B)\in\mathbb{W}} J(A,B) \tag{5.3}$$

with $\mathbb{V} = \{(C_2, D_2), (C_3, D_3), (C_4, D_4), (C_5, D_5), (C_6, D_6), (C_9, D_9)\}$,

where C_i is the probability distribution of attribute i in the first window and D_i the probability distribution of attribute i from the second window,

and $\mathbb{W} = \{(E_1, F_1), (E_7, F_7), (E_8, F_8)\}$,

where E_i is a set consisting of all values of attribute i in the first window and F_i a set consisting of all values of attribute i from the second window.

A description of all attributes is shown in Table 5.1. A sequence of distances d is calculated by comparing successive windows. We now describe an approach that allows the analysis of such a sequence with control charts in order to detect outliers.

5.3.2 Control Charts

The characteristic and trend of a series of distances d (see Equation 5.3) from consecutive windows can give information about attacks, misconfigured networks or other types of network failure. In order to detect outliers and trends in such data, control charts (see section 4.4.2) are an important tool in analysing a sequence of values and detecting if certain limits are exceeded. They are deployed in statistical process control, and are used to determine if a process variable is within certain limits. The advantage of control charts is that a human operator can observe the current and past trends of the process value.

As shown in Figure 4.6, the chart consists of a centre line, the upper control limit (UCL) and a lower control limit (LCL). The centre line is the mean of the historical data, and is updated each time a new process value is added.

Since each connection recorded in the log file can last over several windows, we smooth each distance d using the formula of Equation 5.4 in order to retrieve a new distance $\tilde{d}$. The smoothing constant λ describes the weight of the current distance compared to past distances $\tilde{d'}$, where $\lambda = 1$ only weights the current, and $\lambda = 0$ only weights past data.

$$\tilde{d} = \lambda d + (1-\lambda)\tilde{d'} \tag{5.4}$$

Rule	Description
1	Any point above $k\sigma$ or below $-k\sigma$
2	2 out of the last 3 points above $(k-1)\sigma$ or below $-(k-1)\sigma$
3	4 out of the last 5 points above $(k-2)\sigma$ or below $-(k-2)\sigma$
4	8 consecutive points above or below mean

Table 5.2: Definition of WECO *rules [96], where k is a factor of standard deviation σ.*

We chose the distance $\tilde{d}$ as the process value that is further observed with the control chart. Besides rules generating an alarm when the control limits (LCL/UCL) are exceeded, further control rules can give additional information about the process state. There are several-well known rules, known as the Western Electric Company Rules (WECO) [96]. The most common are presented in the Table 5.2. The WECO rules increase the sensitivity in detecting trends or drifts of the control variable, but also increase the number of false alarms. Therefore, an alarm raised by an out-of-control signal must be considered carefully before activating any rules on a new dataset.

5.3.3 Experimental Results

Our industrial partner, the Post Telecom S.A. of Luxembourg, provided us with two datasets from a Checkpoint firewall cluster, which is used as an internet firewall for their internet-connected servers. The dataset contains a smaller sample dataset and a dataset that represents the traffic from a normal working day (24 h). The dataset statistics are presented in Table 5.3. The experiment was done with all WECO rules enabled and a fixed window size of $n = 100$. For both datasets, the numbers of activated WECO rules are shown in Table 5.4.

The following records are a summary of suspicious and anomalous connection attempts to the company network:

Results for Dataset 1

Records similar to the following occurred 73 times[5] and was identified by WECO rule 1 and 2:

```
"27536" "16May2011" "0:47:50" "Log" "sip_any" "298" "298-Standard"
"" "VPN-1 Power/UTM" "eth-s2/s2p2c1" "⋆⋆⋆" "Drop"
"sip_any" "⋆⋆⋆" "⋆⋆⋆" "udp" "" ""
```

The reason for such connection attempt can either be a badly configured VoIP configuration, or a fraudulent use of Session Initiation Protocol (SIP). In this specific case,

[5]The content of certain sensitive fields has been replaced with "⋆⋆⋆" for data protection reasons.

	Dataset 1		Dataset 2	
Number of events	49,550		1,016,811	
Time period	1h 39 min		24h	
Date	15.05.2011		05.06.2011	
Data size	10.4 MB		196.7 MB	
Unique values	total	$n = 100$	total	$n = 100$
Source port	21,696	70.7	62,982	84
Rule	34	7.1	52	6.4
Interface	26	7	32	6.4
Origin	2	1.9	2	1.5
Action	4	2	5	2
Service	3,293	19.2	65,535	23.6
Source	2,657	47.8	28,727	42.1
Destination	449	34.1	4,000	26.4
Protocol	5	3	5	2.9

Table 5.3: Dataset statistics: The lower part shows the number of unique values for each attribute in the the dataset and the average for a window size of $n = 100$.

WECO rules	Dataset 1	Dataset 2
1	10	1111
2	4	1300
3	4	1521
4	8	1540

Table 5.4: Number of times WECO *rules were activated for dataset 1 and dataset 2.*

many SIP Register requests occurred without success, and both the firewall and control chart identified this anomaly correctly.

Records like the following occurred 72 times and was identified by WECO rule 1:

```
   "38023" "16May2011" "1:10:41" "Log" "4561" "298" "298-Standard"
"" "VPN-1 Power/UTM" "eth-s2/s2p2c1" "⋆⋆⋆" "Drop"
"telnet" "⋆⋆⋆" "⋆⋆⋆" "tcp" "" ""
```

The telnet service is known to be less secure then SSH, because it lacks integrity and confidentiality protection. Inbound telnet traffic is a clear sign of malicious activity and was correctly identified by the system.

Records like following occurred 119 times and were identified by WECO rules 1 and 2:

```
   "45284" "16May2011" "1:27:50" "Log" "X11" "298" "298-Standard"
"" "VPN-1 Power/UTM" "eth-s2/s2p2c2" "⋆⋆⋆" "Drop"
"MySQL" "⋆⋆⋆" "⋆⋆⋆" "tcp" "" ""
```

In this case, an attempt to access an internal database server is being made. Since inbound access to database servers is a clear violation of deployed access control mechanisms, such an incident is highly indicative of potential data theft.

Results for Dataset 2

Records like the following occurred 157,000 times and were identified by WECO rules 1, 2, 3 and 4:

```
   "230236" "6Jun2011" "7:54:34" "Log" "36446" "301" "298-Standard"
"" "VPN-1 Power/UTM" "eth-s2/s2p2c2" "⋆⋆⋆" "Drop"
"26870" "⋆⋆⋆" "⋆⋆⋆" "tcp" "" ""
```

The huge volume of attempted SMTP connections triggered these four WECO rules. Manual inspection of the concerned address showed that it was listed as a *comment spammer* in Project Honey Pot[6].

The greatest challenges would be to evaluate the performance of our approach when a large-scale labelled ground truth dataset is available. We could not perform this task, because such labelled dataset does not exist, though our approach could be used for such a purpose. It might perform an initial labelling, followed by a human-driven validation. Although we focussed specifically on firewall logs, the proposed method should be applicable to many types of log, since the individual data entries share common features (timestamp, IP addresses, actions, ...).

5.3.4 Label Propagation

In addition for the detection of attacks in communication records in real time on the network controller, off-line detection methods are required. For instance, after an incident has occurred, the available log files must be analysed for traces that might point to the attack. Unfortunately such log files are often neglected, since their analysis is difficult due to the huge amount of data (e.g. one million records per day).

Automated approaches to detect anomalous events with *supervised learning* techniques require a labelled dataset. Such a fully-labelled training set is not available in most cases due to the high cost associated with the labelling process and privacy issues.

[6] http://projecthoneypot.org

We propose an approach to classify log file records by using label propagation, a *semi-supervised learning* technique that requires only a small subset of labelled data which acts as a source that pushes out labels to the unlabelled data.

Semi-supervised learning can be used to reduce the cost of the labelling process since only a small set of labelled data is needed. The key to this technique is that both unlabelled data and labelled data are used for the training. Thus, the unlabelled data is used to improve learning accuracy, which is very useful since unlabelled data is usually abundant.

Algorithm

We review the label propagation algorithm described in [97]. The concept of label propagation is shown in Figure 5.3.

We denote the total number of data points of a dataset as:

$$X = (\underbrace{x_1, ..., x_l}_{\text{labelled}}, \underbrace{x_{l+1}, ..., x_{l+u}}_{\text{unlabelled}})$$

which consist of a number of labelled data points l and a number of unlabelled data points u.

The labelled data is described as $(x_1, y_1)...(x_l, y_l)$ with $Y_L = \{y_1...y_l\}$ as the class labels. Labelled data item x_i is known to belong to one of two classes C.

The unlabelled data is described as $(x_{l+1}, y_{l+1})...(x_{l+u}, y_{l+u})$ with $Y_U = \{y_{l+1}...y_{l+u}\}$ as the class labels.

Estimating the class labels Y_U from X and Y_L can be described as a transductive learning process [98]. The main idea behind this mechanism is that unlabelled data items should be labelled with class labels by taking into account similarities to already-labelled data items.

If we consider all labelled and unlabelled data points as nodes in a fully connected graph; the labels from each node propagate to neighbouring nodes according to their proximity. The calculation of the weight between two nodes i, j is shown in equation 5.5 and based on a distance d_{ij}. Nodes with closer Euclidean distances should have similar class labels and thus a higher weight. The attribute σ is used to control the weight.

$$w_{ij} = \exp\left(-\frac{d_{ij}^2}{\sigma^2}\right) \tag{5.5}$$

We can calculate a probabilistic transition matrix T based on the distance from all

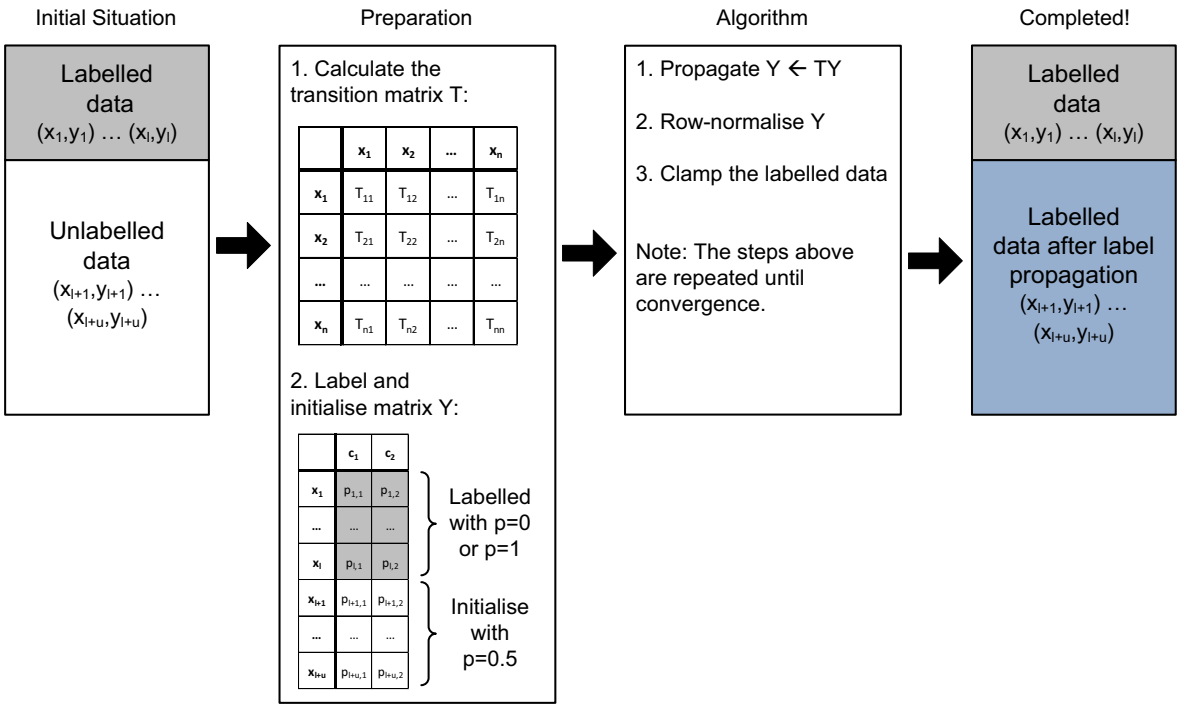


Figure 5.3: Concept of label propagation; the preparation step requires the calculation of the transition matrix T and labelling some windows by assigning them to the different classes (c_1 or c_2). The algorithm is executed until it converges and assigns the initially unlabelled windows to their respective classes.

nodes compared to each other, which defines the probability of jumping from node i to j. The size of the matrix is $(l+u) \times (l+u)$.

$$T_{ij} = \frac{w_{ij}}{\sum_{k=1}^{l+u} w_{kj}} \tag{5.6}$$

In addition, we define a label matrix Y of size $(l+u) \times C$, containing the probability of each data point belonging to class c_1 or c_2.

The label propagation algorithm consists of three steps that are repeated until the label matrix Y converges:

1. Propagate $Y \leftarrow TY$
2. Row-normalise Y
3. Clamp the labelled data

After multiplying the two matrices in step 1, we propagate the probabilities of the labelled data to the unlabelled data. After row normalisation in step 2, we reset the probabilities of the labelled data to the appropriate value (0 or 1). This is necessary in order to retain persistent elements in the label matrix (clamping). In the next run, the labelled data acts as a source and the unlabelled data is updated.

Steps 1, 2 and 3 in the algorithm are computationally equivalent to a matrix multiplication, which is of $\mathcal{O}(n^3)$. In our case the Y matrix has n rows and two columns and thus the computational complexity of steps 1, 2 and 3 is only $\mathcal{O}(n^2)$. The outer loop of this algorithm (until convergence) is shown to be bounded in [97] and our own practical experience is in line with this result.

Iterative Labelling

Label propagation can reduce the manual effort required since only a small part of the dataset need be labelled by an expert. But since a typical firewall log file consists of a huge number of records, the labelled part can still be uncomfortably large for manual labelling. Our iterative method reduces the amount of manually-labelled data needed.

The concept of iterative labelling is shown in Figure 5.4. Firstly, we separate the log file into a subset and label $X_{L(1)}$ with the support of an expert. We then use the label propagation algorithm to label $X_{U(1)}$. We repeat the process, choosing $X_{L(2)} = X_{L(1)} + X_{U(1)}$, and use the label propagation algorithm again until the log file is fully labelled.

Firewall Log Files

Even though label propagation is not limited to a specific application scenario, we demonstrate its use in detecting anomalous records of firewall log files. Due to the huge amount of logged connections and the temporal dependencies (e.g. in a network scan), we split records into windows and determine the distance as described in section 5.3.1. This is used by the label propagation algorithm to represent the degree of similarity between data points. We consider two classes C, either normal or anomalous, because we wish to classify malicious windows in the log files.

To reduce the amount of labelled data required in large datasets, we split the data into several parts which are labelled in several steps. After the first part has been labelled with the label propagation algorithm, the resulting labelled data can be leveraged as an input to label the remaining data of the next part. This steps are repeated until the whole dataset is fully labelled.

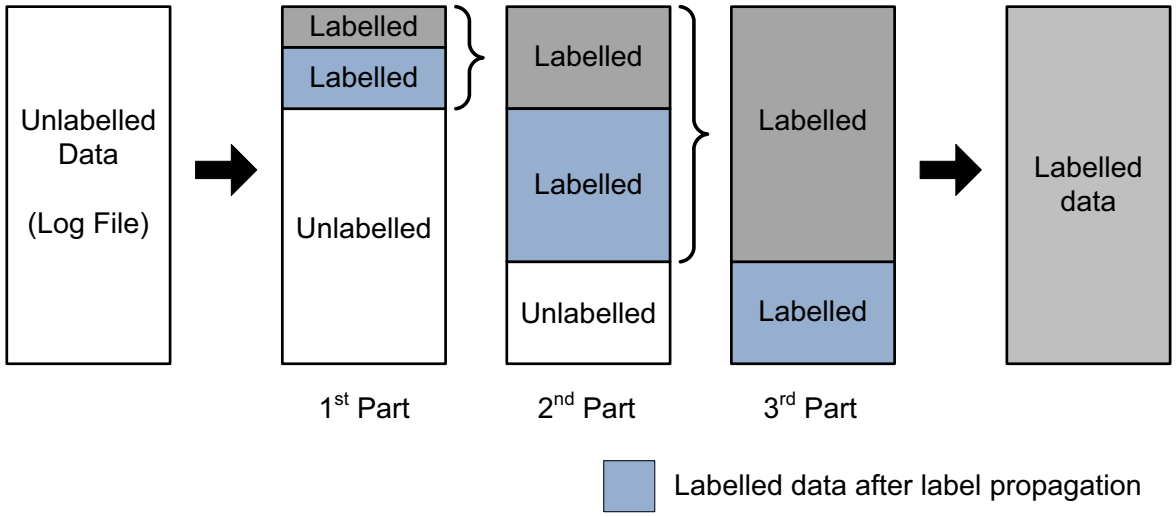


Figure 5.4: Concept of iterative labelling; only a small amount of labelled data is initially used by the label propagation algorithm, reducing the cost of manual labelling for large datasets.

5.3.5 Experimental Results

For the validation of the label propagation algorithm, we used the Checkpoint firewall dataset 1, as described in Table 5.3. The experiments were run on a dual-core 2.8 GHz Intel PC with 4 GB of RAM. The calculation of the transition matrix T is the most time-consuming operation if the window size is small compared to the number of records in the dataset. We chose a fixed window size of $n = 100$, since most attacks result in a number of logged records that are in that range. The dataset was further split into three parts, since the ratio between the amount of labelled and unlabelled data has an impact on the classification. We determined by experiment a ratio of 1 to 5 to be sufficient.

First Part

This part consisted of 50 windows in total, where we manually labelled five anomalous and five normal windows. For the 40 unlabelled windows, both classes were initialized with a probability of 0.5. After applying the label propagation algorithm, we detected one window which was labelled with a probability of 0.99 as anomalous and occurred as a block of 116 records:

```
   "2128" "16May2011" "0:02:15" "Log" "60270" "166" "166-Standard"
"service_id:  http" "VPN-1 Power/UTM" "eth-s2/s2p2c2" "⋆⋆⋆" "Accept"
"http" "⋆⋆⋆.bb.netbynet.ru" "⋆⋆⋆" "tcp" "" ""
```

Investigating this case we looked into the log and correlated the source IP with a well-known list of public HTTP proxies. Such proxies are often used to hide the identity

of web users and while this can be legitimate in order to protect privacy in less democratic countries, in our case the incident was more probably a brute-force authentication breaking attack targetting a Content Management System (CMS).

Second Part

This part consists of 50 labelled windows from the previous step and added 200 unlabelled windows. The algorithm classified three windows as anomalous: these included two different types of records. The following window was identified with a probability of 0.74 and occurred as a block of 327 records of the following type:

```
   "11141" "16May2011" "0:16:37" "Log" "35961" "166" "166-Standard"
"service_id:  http" "VPN-1 Power/UTM" "eth-s2/s2p2c2" "⋆⋆⋆" "Accept"
"http" "⋆⋆⋆.bb.netbynet.ru" "⋆⋆⋆" "tcp" "" ""
```

The interpretation of this record is similar to the record from the first part. The next window was identified with a probability of 0.73 and occurred as a block of 60 records:

```
   "12182" "16May2011" "0:18:21" "Log" "2364" "298" "298-Standard"
"" "VPN-1 Power/UTM" "eth-s2/s2p2c1" "⋆⋆⋆" "Drop"
"telnet" "⋆⋆⋆" "⋆⋆⋆" "tcp" "" ""
```

Any telnet traffic is highly suspicious and is a clear sign of attack.

Third Part

The final part took the 250 labelled windows from the previous step, and attempted to label the remaining 246 windows. It classified one windows as anomalous with a probability of 0.96, containing a block of 73 similar records:

```
   "27842" "16May2011" "0:48:32" "Log" "5071" "298" "298-Standard"
"" "VPN-1 Power/UTM" "eth-s2/s2p2c1" "⋆⋆⋆" "Drop"
"sip_any" "⋆⋆⋆.calpop.com" "⋆⋆⋆" "udp" "" ""
```

We looked into the log and the incoming traffic was sourced by a machine located at a popular hosting provider. The many SIP packets resulted from a Voice over Internet Protocol (VoIP) scan looking for open SIP proxies.

Convergence

We also looked at the convergence properties of the label propagation algorithm. We evaluated the differences between successive label matrices Y using the squared sum of all elements. The difference D can be described using Equation 5.7, where the probabilities

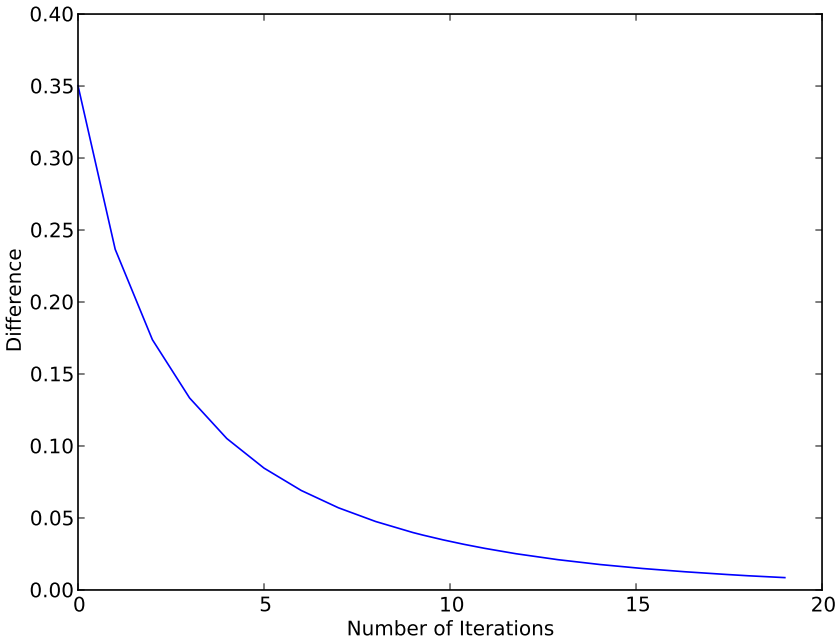


Figure 5.5: Convergence of the label propagation algorithm (section 5.3.4) for the second part (50 labelled windows, 200 unlabelled windows, window size $n = 100$).

of matrix Y after the first iteration are given by $p_{i,j}^{(1)}$, and the probabilities after the second iteration by $p_{i,j}^{(2)}$.

$$D = \sum_{i=1}^{l+u} \sum_{j=1}^{2} (p_{i,j}^{(1)} - p_{i,j}^{(2)})^2 \tag{5.7}$$

The evolution of the label propagation algorithm for the dataset during the second part (50 labelled, 200 unlabelled) is displayed in Figure 5.5. This shows that 10 iterations are sufficient to obtain convergence. This result empirically validates the convergence properties established in [97].

5.4 Hidden Communication Patterns

The following application scenario focusses on the detection of hidden communication patterns used by adversaries to activate a backdoor on a compromised host in the network. The detection of stealthy backdoor communications is challenging because of a lack of proven methodologies and targeted tools. A typical backdoor makes it possible for an attacker to connect at will and perform actions on compromised machines. While such backdoors operate through a shell command line interface operated over TCP ports

opened on a compromised machine, stealthy backdoors will avoid detection by enabling temporary shells for short periods. This is done by promiscuously intercepting network traffic and activating a remote shell when a given sequence of predefined IP packet patterns occurs.

The best-known backdoor that implements this technique is cd00r [99], where a series of successive IP packets is used to trigger a remote shell activation. The underlying principle is quite simple. Several IP packets are sent to the compromised machine. The port is activated by a defined sequence of TCP SYN packets, followed by the chosen port number that is then used to access the compromised host. If this sequence of predefined destination ports occurs, then a basic shell is opened on the machine. Consequently, this shell runs over a port only during the time interval in which the attacker is interacting with the machine.

Until now, detecting such a backdoor was impossible because the specific series of destination ports can easily be changed and extended by the attacker. In the following, we propose a method for detecting such a backdoor, which includes the following contributions:

- A practical approach for detecting stealthy backdoors on an OpenFlow-based network controller.
- We leverage a data mining technique for searching for associations between rare events in order to determine the knocking sequence. We demonstrate its performance and efficiency on several datasets and attack scenarios.
- We propose a methodology to set the parameters (minimum support, confidence and interest) used to search for rare associations.
- We have implemented and assessed our prototype with respect to existing known backdoors.

5.4.1 Concept of Backdoor cd00r

Cd00r demonstrates a hidden Linux backdoor that cannot be detected by network scanners and intrusion detection systems (IDSs). It is not visible in the network socket table until the backdoor is activated by a sequence of TCP, UDP or ICMP packets. Subsequently, a shell or other tools (e.g. `netcat`[7]) can be used to access the compromised machine. This strategy is possible because that cd00r uses the libpcap library to listen on raw sockets. Therefore, applications like `netstat`[8] or `NMAP`[9], which listen on higher-level sockets, cannot detect that the port is being monitored [100]. In addition, many low-end firewalls lack advanced deep packet inspection features and thus do not block the

[7] http://netcat.sourceforge.net/
[8] http://linux.die.net/man/8/netstat
[9] http://nmap.org/

modified and illegally-formed packets used in attempts to connect to the compromised machine [101]. For a network administrator it is therefore very difficult to detect or prevent the use of such backdoors. On the other hand, the backdoor could also provide a secure way to defend a system against attacks, since critical services can be protected by the fact that the port is not visible from the outside.

In our scenario, the activation of the backdoor uses a sequence of TCP SYN packets that must be sent to a list of ports in the correct order (knocking). Furthermore, a hard-coded port is not needed for the connection to the compromised machine since we can choose it dynamically by sending the chosen port after the activation sequence. The following example demonstrates a complete session to activate and communicate with the backdoor using telnet:

```
t="telnet host.victim.com"
$t 999; $t 888; $t 777; $t 666; $t 555; $t 1234; sleep 1; $t 1234
```

The ports 999, 888, 777, 666 and 555 are used to activate, and the port 1234 to define where the shell should be spawned. After waiting for a short time (sleep 1) for the shell to be started on the target machine, we can connect with telnet and have access to the system. Our experimental implementation of the backdoor was achieved using the code of [99]. The original cd00r implementation, which uses a hard-coded listening port, can be found at [102]. The implementation of a similar backdoor, Sadoor, is available at [103]. An advanced methods of port knocking with secret sharing that need a group of people to open ports is proposed in [104].

5.4.2 Detection of Knocking Sequence

The detection of a knocking sequence in network traffic is difficult since no appropriate tools are available. In case of SDN, we can identify the knocking sequence by providing a network application for this purpose. As shown in Table 5.5, an indicator of a port knock is that the response from the compromised target is a TCP RST instead of a TCP SYN/ACK. The tracing of such uni-directional flows to a particular destination IP address allows the knocking ports involved to be determined. Moreover, the port that is subsequently opened by the backdoor can be determined and a flow entry with the action "drop" can be installed in order to prevent the attacker from reaching the destination host.

Although the sequence itself can be determined by tracing the TCP handshake for flows, the order of ports that were used to enable the backdoor is still unknown. This is because each `packet-in` message of the knocking sequence can arrive on the network controller at different times.

We therefore propose in the following an approach that allows such sequences to be found in a large dataset.

Sequence	Flow	Source	Destination	Port	Protocol
1	1	192.168.2.1	192.168.2.2	80 → 777 [SYN]	TCP
	2	192.168.2.2	192.168.2.1	777 → 80 [RST]	TCP
2	3	192.168.2.1	192.168.2.2	80 → 666 [SYN]	TCP
	4	192.168.2.2	192.168.2.1	666 → 80 [RST]	TCP
3	5	192.168.2.1	192.168.2.2	80 → 555 [SYN]	TCP
	6	192.168.2.2	192.168.2.1	555 → 80 [RST]	TCP
4	7	192.168.2.1	192.168.2.2	80 → 1234 [SYN]	TCP
	8	192.168.2.2	192.168.2.1	1234 → 80 [SYN/ACK]	TCP
	7	192.168.2.1	192.168.2.2	80 → 1234 [ACK]	TCP

Table 5.5: An example of a knocking sequence (1,2,3) of the cd00r backdoor in OpenFlow-based SDN, where each port request creates a new flow and `packet-in` *message. Sequence 4 finally connects to the backdoor that is located at port 1234.*

5.4.3 Rare Association Rule Mining

Finding association rules was introduced as a generally applicable method in data mining by Agrawal, Imielinski, and Swami in 1993 [105]. Analysing supermarket transactions was one of the original motivations for finding association rules, since they can reveal information about products that customers usually buy together. Just as information about frequent *itemsets* (e.g. bread, butter and jam) is interesting in some applications, finding *infrequent* itemsets is useful in finding events that rarely occur and may be the result of a failure or an attack. A valid rule can be defined as the combination of itemsets with length k, that appear only seldom in a dataset transaction, but more frequently than suggested simply by coincidence — which would have normally been pruned out when using classical association mining algorithms such as Apriori [105].

The following sections describe the algorithm that discovers association rules among infrequent items and is used to find the knocking sequence of TCP ports for the backdoor described in the previous section.

A TCP port is considered to be a random variable I with event set $\mathbb{E}(I) = \{0, ..., 65535\}$. All recorded port accesses are saved in a database D and are divided into windows w of a defined size n. For each window we get $w = \{i_1, i_2, ..., i_n\}$. If X, Y are itemsets of ports that occur together in a window, the resulting association rule is $X \Rightarrow Y$ with $X, Y \subset \mathbb{E}(I)$ and $X \cap Y = \varnothing$.

The percentage of windows in the dataset that contain the rule can be described by the *support*:

$$\text{supp}(X \cup Y) = P(X \cup Y) \tag{5.8}$$

The ratio of windows that contain X and also contain Y is defined as the *confidence*:

$$\text{conf}(X \Rightarrow Y) = P(Y|X) = \frac{P(X \cup Y)}{P(X)} \tag{5.9}$$

In order to find interesting association rules, the minimum threshold for support (*minSupp*) and confidence (*minConf*) must be defined by the analyst in advance and depends on the dataset. A common problem when finding an infrequent itemset is that, since both sets rarely occur, the minimum support threshold (*minSupp*) has to be set very low. This results in a huge number of possible combinations of items and is known as the *rare item problem* [106]. In general, rare association rule mining requires low *minSupp* but high *minConf* to reduce the overall number of rules.

Since the resulting number of rules may still be very high, uninteresting rules must be filtered out, and we rely on the pruning strategy described by Wu [107]. Based on the assumption that X and Y are independent if $P(X \cup Y) = P(X) \cdot P(Y)$, the interest function is defined as shown in Equation 5.10.

$$\text{Interest}(X, Y) = |\text{supp}(X \cup Y) - \text{supp}(X)\text{supp}(Y)| \tag{5.10}$$

By comparing the interest value with a minimum interest threshold (*minInte*) set by the analyst, this interest function helps to reduce the number of rules and removes itemsets which are independent of each other and are not considered as valid rules. The correlation function in Equation 5.11 determines the dependence of two itemsets X, Y with three possible cases.

$$\text{Correlation}(X, Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X)\text{supp}(Y)} = \begin{cases} 1 & \text{independent} \\ > 1 & \text{positive correlated} \\ < 1 & \text{negative correlated} \end{cases} \tag{5.11}$$

A positive correlation means that the occurrence of one itemset increases the occurrence of the other itemset, while a negative correlation means that the occurrence of one itemset discourages the occurrence of the other. To determine if a found rule is above the minimum confidence threshold (*minConf*), Wu defines $\text{conf}(X \rightarrow Y)$ to be $\text{CPIR}(Y|X)$:

$$\text{CPIR}(Y|X) = \frac{\text{supp}(X \cup Y) - \text{supp}(X)\text{supp}(Y)}{\text{supp}(X)(1 - \text{supp}(Y))} \tag{5.12}$$

	Dataset 1	Dataset 2
Total number of TCP packets/ports t	19,609	50,787
Involved port numbers u	559	804

Table 5.6: Dataset statistics.

Algorithm

The pseudocode of the matrix-based scheme is shown in Algorithm 1. We rely on the description given in [108]. We initially scan the whole dataset and store all port numbers that have a support count greater than *minSupp* in a matrix, $Inf1$[10] (see lines 1 to 4). In lines 5 to 15, we check whether each port in every window is infrequent by checking the $Inf1$ matrix, and, if it is included, map it to a new matrix, *Temp*. If a window in *Temp* contains more than one element, we calculate all possible combinations and increase the support count of each combination in a new matrix, $Inf2$.
In lines 16 to 24, we calculate the interest value (Equation 5.10) for every itemset in $Inf2$ that has a support count greater than one. If the interest is below *minInte*, we remove the itemset from the matrix. For the remaining itemsets, we determine (lines 25 to 32) for each if the correlation (Equation 5.11) is greater than or equal to one and if CPIR (Equation 5.12) is larger than *minConf*. This final step is needed to determine the correct order of the elements in the itemset.

5.4.4 Experimental Results

The experiments were run on a dual-core 2.80 GHz Intel PC with 4 GB of RAM and an Ubuntu 64 bit OS. The backdoor was installed on an Ubuntu VM with the network card running in bridged mode. We evaluated our results on two datasets which contain the complete network traffic between the attacker and the target host. There were five complete connections to the backdoor during the period covered by each dataset. Each connection consisted of a port-knocking sequence, followed by a user-defined port for the shell. After opening the port and connecting to the backdoor, a text document was saved on the target machine and the connection was closed.

To determine the optimal values for *minSupp*, *minConf* and *minInte*, we used an analytical approach based on statistical parameters that were calculated from the dataset beforehand and are shown in Table 5.6. These are the the number of ports involved u, and the total number of ports t in the dataset. The resulting number of windows is defined by $l = t/n$, where the window size n was chosen by experiment. We had promising results with $n = 1,000$, which gave a high probability that the port sequence was completely included in a single window.

[10] *Inf1* denotes an infrequent itemset having a cardinality of 1, *Inf2* a cardinality of 2, etc.

Algorithm 1 Matrix-based scheme (Zhou, Yau [108])

Require: database D, *minSupp*, *minConf*, *minInte*, association rules AR = ∅

scan the database D and find all infrequent 1-itemsets (*Inf1*)
Item ← {a matrix used to store information of all items in D}
Item.index ← the index value of infrequent item in *Inf1*; frequent items are assigned an index value of −1;
Infk ← {matrices used to store support counts of infrequent k-itemsets, where $k > 1$};
scan database D a second time
for each window $w_i \in$ D **do**
  for each item i ∈ window w_i **do**
    if i.index $\neq -1$ // identify infrequent items **then**
      map i.index into *Temp*
    end if
  end for
  if the number of items in *Temp* is greater than 1, **then**
    find all combinations of these values and increase support count of each combination
  end if
end for
for each k-itemset $I \in$ Infk **do**
  if I.count ≥ 1 **then**
    for $\forall$ itemsets $X, Y, X \cup Y = I$ and $X \cap Y = \varnothing$ **do**
      if interest(X,Y)< minInte **then**
        Infk ← Infk-{I};
      end if
    end for
  end if
end for
for each infrequent k-itemset of interest $X \cup Y \in Infk$ **do**
  if correlation$(X, Y) > 1$ && CPIR$(Y|X) \geq$ minConf **then**
    $AR \leftarrow \{X \Rightarrow Y\}$
  end if
  if correlation$(X, Y) > 1$ && CPIR$(X|Y) \geq$ minConf **then**
    $AR \leftarrow \{Y \Rightarrow X\}$
  end if
end for
return AR

If the number of backdoor activations is known, the optimal *minSupp* can be determined by

$$\text{supp} = \frac{\text{number of knocking sequences}}{\text{number of windows } l} \tag{5.13}$$

To determine the optimal *minSupp* when the number of backdoor activations is unknown, we propose the following analytical approach. The probability that a certain port is included in one window is given by:

$$p = P(\text{port} \in \text{window}) = 1 - \left(1 - \frac{1}{u}\right)^n \tag{5.14}$$

where u is the number of ports involved and n the window size.

The probability that a port is in exactly x windows can be calculated as a binominal distribution, as shown in Equation 5.15. In order to simplify the calculation of the permutation, logarithmic values were used in the implementation of the algorithm.

$$p(x) = \binom{l}{x} \cdot \left(1 - \left(1 - \frac{1}{u}\right)^n\right)^x \cdot \left(\left(1 - \frac{1}{u}\right)^n\right)^{l-x} \tag{5.15}$$

In order to evaluate the results for both datasets, we needed to set the correct parameters for *minSupp*, *minInte* and *minConf*. In the first case, we see from Equation 5.13 that, for datasets 1 and 2, the optimal *minSupp* values were approximately 0.3 and 0.1 respectively for a window size of $n = 1,000$. Since the number of activations is unknown in a realistic scenario, we chose *minSupp* by calculating the probabilities of the binominal distribution for different support thresholds from Equation 5.15, giving the results displayed in Figure 5.6. The chosen *minSupp* should be in the range that lies between 0 and the gradient of the probability function.

The correct *minInte* was experimentally determined after choosing *minSupp* of 0.3 for dataset 1 and 0.1 for dataset 2. Table 5.7 displays the number of rules found in the dataset as *minInte* was increased from zero, decreasing rule identification. To reduce the number of uninteresting rules and to assure that two itemsets are not independent, we propose a *minInte* that is approximately 0.17 for dataset 1 and 0.07 for dataset 2.

Based on the different settings of the *minSupp* threshold, the results for both datasets are displayed in Table 5.8 and Table 5.9. We used *Precision* and *Recall*, two statistical measures that determine the quality of a binary classification test. In our scenario, the knocking sequence of the backdoor results in 20 possible rules (e.g {555, 666}, {777, 888}). The recall is defined as the number of correctly found rules divided by the number of all possible rules. The precision defines the quality of the classification and can be

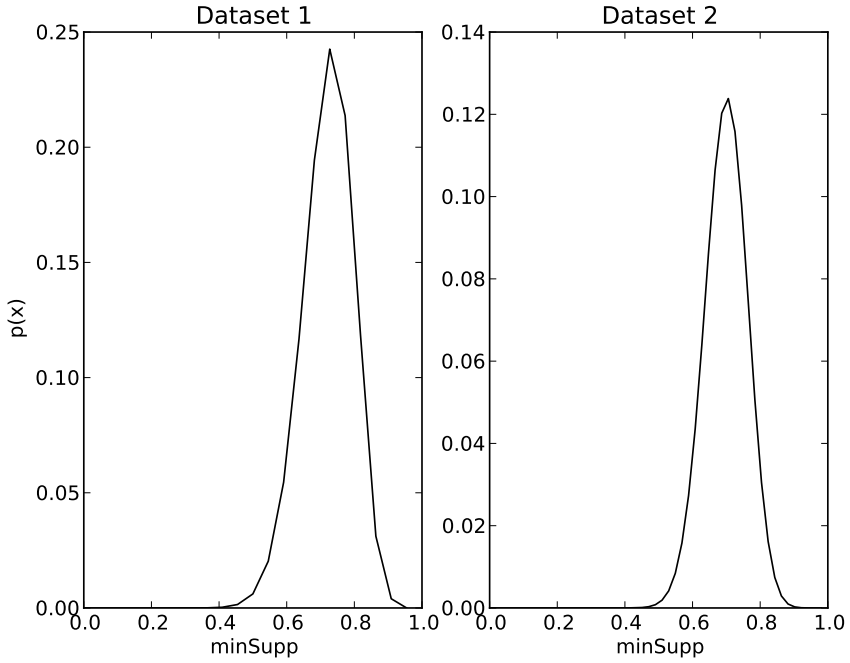


Figure 5.6: Probability that a port is in exactly x windows.

calculated from the number of correctly found rules divided by the number of found rules. The results achieve a recall rate of 70% combined with a high precision of approximatly 80%.

We have shown a practical and efficient approach to detecting port-knocking sequences in network traffic. To the extent of our knowledge, no previous papers have tackled this issue. This might be due to the inherent complexity of the task as well as to the perceived limited deployment of backdoors based on port knocking. In fact, very little is known about the operational management of such backdoors. Since no existing tool detects them, such backdoors may have been in place for a long time without being detected by existing monitoring solutions. We have designed a monitoring tool that relies on data mining methods targeted at identifying sequences of rare events and we have shown that the proposed approach is viable.

We have illustrated the performance of our tool on two datasets generated in a controlled environment, where we could precisely assess the true positive and false negative rates. We are aware that these datasets might not be representative for a typical network. As a matter of fact, we also used larger datasets from large campus networks, but the major issue was that the obtained results could not be easily validated. This would have required us to have forensic access to the suspicious machine, which was impossible from an administrative point of view.

minInte	Dataset 1	Dataset 2
0.00	10437	12847
0.01	10437	12847
0.02	10437	1639
0.03	10437	1639
0.04	10437	287
0.05	10437	287
0.06	1586	14
0.07	1586	14
0.08	1586	0
0.09	1586	0
0.10	1586	0
0.15	205	0
0.20	16	0
0.25	14	0
0.30	0	0
0.35	0	0

Table 5.7: Number of rules detected for different interest values (minInte).

Parameters	Recall	Precision
minSupp = 0.30	(14/20) = 0.70	(14/16) = 0.88
minSupp = 0.50	(14/20) = 0.70	(14/16) = 0.88
minSupp = 0.70	(14/20) = 0.70	(14/17) = 0.82
minInte = 0.17		
minConf = 0.90		

Table 5.8: Results for dataset 1.

Parameters	Recall	Precision
minSupp = 0.10	(14/20) = 0.70	(14/14) = 1.00
minSupp = 0.40	(14/20) = 0.70	(14/22) = 0.63
minSupp = 0.70	(14/20) = 0.70	(14/32) = 0.44
minInte = 0.07		
minConf = 0.90		

Table 5.9: Results for dataset 2.

The major challenge in finding rare association rules in network traffic is that the required low minimum support leads to a huge number of rules. We have proposed a set of simple rules that rely on pruning and a parameter-setting method. An analytical approach is needed to achieve a good understanding of the probabilities that a certain itemset will occur, and of how to set the parameters of the algorithm. If this requirement is fulfilled, the proposed concept is able to find the port sequences of stealthy backdoors. Datasets from a longer time period should achieve even better results, because other infrequently-accessed ports that would otherwise be considered as noise will occur more often so no longer be infrequent.

5.5 Conclusion

In this chapter, we have presented the concept of packet inspection on an OpenFlow-based network controller. This is realised by inspecting encapsulated packets within OpenFlow control messages. Due to the lightweight approach, new security policies can be realised that are based on the whole protocol stack, including payload.

For the analysis of communication records, we have proposed a controller application that consists of a two-phase approach. Firstly, it tags each new communication with an respective ID based on existing firewall rule sets. Afterwards, we proposed a window-based approach and defined distance metrics to differentiate network records in order to detect network anomalies in a large datasets. This can be either realised in an on-line mode based on control charts, or off-line by using label propagation, a semi-supervised learning approach that requires only a limited amount of labelled data and simplifies the finding of relevant log file records after an incidence has occurred.

The analysis of TCP flows on the network controller also allows hidden communication patterns to be identified. These may be used by a stealthy backdoor in order to enable an otherwise hidden port on a compromised host. We have described an approach to detect such a connecting attempt by tracing uni-directional flows. In order to detect the sequences themselves, we used an approach from rare association rule mining, which was implemented and validated on a testbed.

Chapter 6

Logging Facility for Network Services

Network applications that run on a network controller must be designed to meet a high standard of reliability in order to avoid network failures. This requires high standards for software management, for instance when modifying software components to customer requirements. In order to prevent this error-prone task from resulting in network faults, in this chapter we propose a concept for automated source code extension that provides a high degree of reliability, maintainability and scalability. This concept is complemented by a transparent storage solution in order to allow anomaly detection based on flow entries. The contribution at this chapter has been published in [109].

6.1 Logging of Communication Records

Following a major network failure, the network administrator is often forced to perform root-cause analysis. In order to determine the SDN software components that require a deeper investigation, considerable knowledge of the software-based control plane is required. The control plane in SDN is based on several software components, principally the network controller and the network services. Typical events that must be processed originate either from the network devices, or from business applications. From an operator's perspective, such an architecture requires new tools for fault management and network debugging.

After the program components that could be responsible for the fault have been identified, the system events of interest can be analysed. Since some faults occur only under certain circumstances, a storage system is required in order to log parameters for a certain period of time. While some errors can be found by analysing the source code isolated from the network, some faults are only visible in operational mode. We focus on faults that are related to the flow installation routine in OpenFlow, since this is very important for the packet forwarding behaviour of the network. To aid in the

diagnosis of possible problems, we suggest that each new flow entry installed based on a `FlowMod` message, should be stored as a duplicate (*flow record*) in central network storage connected to the network controller (see Figure 5.2).

6.1.1 Requirements

In order to realise such a concept in an operational context, all software components involved in the flow installation process must be modified or extended with the additional code required to provide such functionality. Since each program structure is different, due to the differing application scenarios, source code modification and extension is a tedious and error-prone task. Therefore, the requirements of our approach are to allow:

- Storing of network parameters in a database to enable user-defined tracing, attack and fault detection capabilities; for instance, digital forensics could require complete tracing support, while other scenarios might be more lightweight;
- Transparent storage for the underlying monitoring plane. Since several solutions exist for storing large datasets, we need an approach that can be easily adapted to a particular storage solution. For instance, changing the storage solution from Redis[1] to HBase[2] should be automated and transparent for the end user.
- Automated source code extension of a wide variety of controller programs to instrument controlling code with new functionality without modifying the controller itself. This should be possible with user-defined levels of granularity;

To satisfy these requirements, in this chapter we propose a framework that incorporates an efficient NoSQL[3] database for logging OpenFlow-related network parameters in combination with *graph transformation* [110] for automated source code extension of network controller software.

6.1.2 Application Scenarios

There are several application scenarios that benefit from such a storage concept:

Network Forensics Based on the history of collected flow records, the network operator can reconstruct network attacks or incidents. This is especially important when attack has resulted in an economic loss to the company and traces must be provided (i.e. digital forensics), for example, to law enforcement authorities. In the same way, recall/replay facilities can be provided to investigate the implications of such attacks on the network.

[1] http://redis.io/
[2] http://hbase.apache.org/
[3] http://nosql-database.org/

Software Debugging The functionality of debugging software running on the controller can become increasingly complex and may involve many different program parts. Firstly, it allows to identification of the function call that installed flow entry on a switch. Secondly, logical errors can be detected by defining *safety invariants*, which can be compared with the relevant flow records in the database. For instance, a typical invariant could be defined to require that the source and destination IP addresses of a flow entry are different, as the inverse would indicate a flaw in flow installation routine concerned. The processing of safety invariants on the database entries could be realised by a separate application that is decoupled from the controller in order to avoid performance degradation.

Traffic Visualisation From a monitoring perspective, flow records can be used to visualise the traffic load in the network topology. This information can also be used by load-balancing applications, since the current number of flows on a particular switch can be retrieved from the database. In this way, the controller-to-switch traffic is reduced because it is not necessary to poll network devices for status updates.

Network Snapshots A new application scenario that becomes feasible with SDN is to create *network snapshots*; this is not possible with non-SDN. Such a snapshot includes the state of the network controller and network devices at a particular time. Since the network devices are distinguished by the number and type of flow entries, they can be stored in a centralised storage solution as described in subsection 6.1.4.

6.1.3 Extensions

The replicated flow entries can be extended with additional information in order to provide more detail to the network operator. We propose the following fields as an extension to the fields that are already provided for the flow entries in the OpenFlow protocol (see subsection 2.5.2):

- Unique flow entry identifier (global).
- Switch identifier.
- Name of program that sent the `FlowMod` message, together with the code line number.

The unique flow identifier can be used to trace back suspicious flow entries on a switch to the program that sent the installation message. Since each flow record in the database contains the program name and line in code at the creation time, it is possible to determine the program and function call of the program that installed the flow entry. Additionally, each flow record can be supplemented with flow entry statistics (see subsection 2.5.2) after expiration by using of `FlowRemoved` messages. This can tag each flow record as being in an active or inactive state, allowing centralised flow monitoring (see section 4.3).

Key	Field	Value
0001	match1	matchfield1 value
	...	
	matchn	matchfieldn value
	sid	Switch identifier
	prog	Path of associated program
	lic	Line in code
	time	Timestamp
	rp	Received packets (*)
	rb	Received bytes (*)
	ds	Flow alive time (s) (*)
	dns	Flow alive time fractional part (ns) (*)
	reason	Reason for flow removal (*)
		(* updated after removal)

Table 6.1: Data structure of a flow record in Redis. The match fields are extracted from the `FlowMod` message, and are complemented with additional parameters in order to allow network debugging.

6.1.4 Database Management

Saving flow entries as flow records in a central database is challenging since huge volumes must be inserted in a very short time. A benchmark for the POX controller showed that slightly over 30,000 flows per second can be processed [29] on an Intel i3 PC and 32-bit OS. In order to achieve this high scalability and performance, we propose the use of a NoSQL database system having a key-value store, which is optimised to store many new items, but provides fewer query methods than a relational database system.

For implementation, we chose the open source Redis[4] database. The program's own benchmark utility showed a rate of approximately 150,000 requests per second for the `SET` and `GET` database requests on an Intel i5 PC (2.50 GHz x 4 cores) with 8 GB of RAM and an Ubuntu 12.04 64-bit OS. This can be considered to be sufficient for a single POX controller.

The Redis database uses hashes to map between string fields and string values and is able to represent structured objects:

```
[key]:{'field' -> 'value', 'field' -> 'value'}
```

As shown in Table Table 6.1, each flow record is defined as a hash object that contains parameters as its field values. The second hash object in Table 6.2 is generated for each switch in order to map all associated flow records by using their IDs as field values.

[4]http://redis.io/

Key	Field	Value
switch:1	total	6
	removed	2
	0001	1
	0002	0
	...	

Table 6.2: Example of a switch specific data structure that keeps track of all flow records and their status.

Except for the first two entries, which are reserved as counters for the total number of installed and removed flow entries, the current status of each flow entry is specified by a flag that denotes whether it is active (1) or removed (0). An active state defines a flow entry that is currently in use, whereas a removed flow entry has expired or was uninstalled by the controller. The described object structures provide a convenient way of obtaining information about the network with only a small number of database requests.

6.2 Automated Source Code Extension

In order to relieve the burden on the programmer in manually adding code to existing controller programs to realise such a logging facility, we propose dynamically generating code using graph transformation, as displayed in Figure 6.1. This allows all existing controller software to easily be extended to provide additional functionality without manual intervention by the programmer.

The steps involved are (1) parsing of the source code to derive its abstract syntax tree representation, AST1; (2) executing the graph transformation system on AST1, yielding a second abstract syntax tree AST2, where intermediate graphs contain additional temporal structures; and (3) serialising AST2 to derive the extended source code. The modification of the graph AST1 to the graph AST2 is defined by the *graph transformation rules*, which are presented for an application scenario in section 6.3.

The automated formal and rule-based technique of graph transformation has several advantages compared to other automated approaches such as ATL [111]. These are in line with the listed requirements in subsection 6.1.1 and include the following benefits:

Reliability We achieve a high level of reliability due to the guaranteed formal results and automated analysis techniques for graph transformation concerning the well-formedness of the output. A more detailed description about the correctness of our approach is available at [112]. In addition, the approach is minimally invasive in the sense that the observable behaviour of the the network controller is not affected. This is

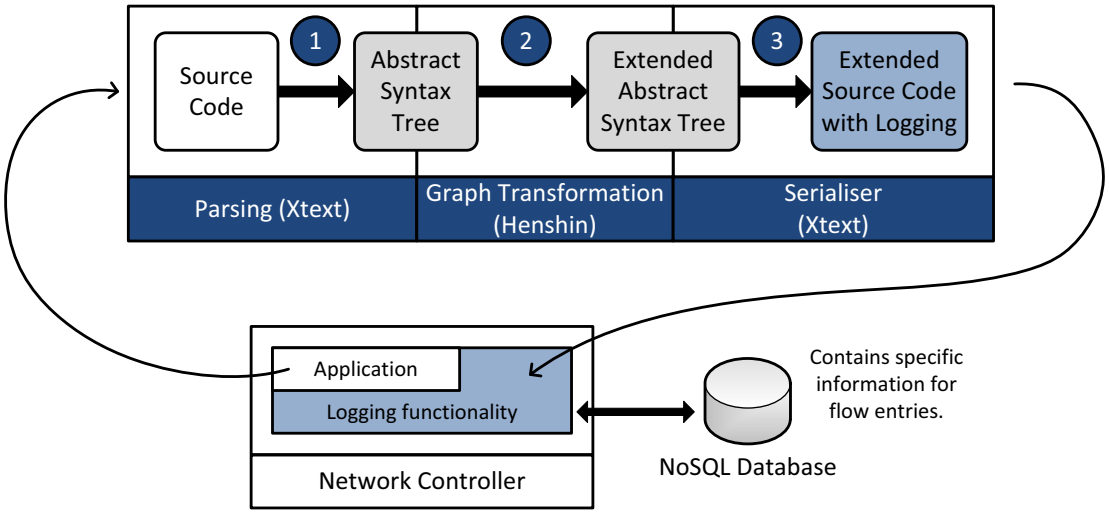


Figure 6.1: Graph transformation to allow automated source code extension of controller programs.

because the existing code is retained and extended only with the necessary code modifications, for instance additional logging instructions.

Maintainability When the network operator modifies the underlying storage solutions, the framework does not require manual changes to the controller source code, but only an adaptation of the transformation rules, which are then used to automatically regenerate the source code.

Scalability The same set of transformation rules is used for a wide range of existing controller software.

Automated source code extension was developed using the Eclipse plug-ins Xtext [113] and Henshin [114], which are both based on the Eclipse Modelling Framework (EMF)[5]. Henshin is a graph transformation environment. We used its graphical user interface for the visual specification of the graph transformation rules and its graph transformation engine for their execution. The parser and serialiser for Python source code were generated using Xtext with the official Extended Backus–Naur Form (EBNF) grammar for Python 3.2.

[5]http://www.eclipse.org/emf/

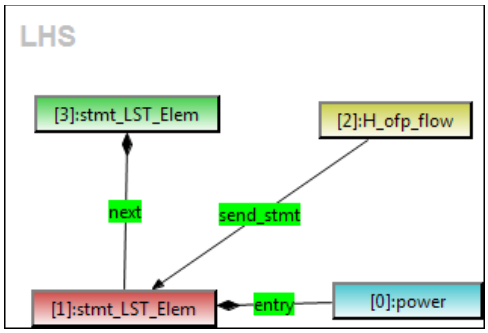


Figure 6.2: Example of graph transformation rule LHS: `extend_ofp_flow_full`

6.3 Proof of Concept: POX controller

In the following we describe a typical application scenario for the concept of graph transformation described above. It allows the logging of flow entries in a NoSQL database system based on the POX network controller (Python) [29]. In order to allow the operator to choose the required level of logging detail, we have developed two rule sets.

The user can choose between a *light* and a *forensic* mode. In the first, a user-defined selection of flow entry parameters is saved in the database; this is useful for archiving or simplistic analysis. In the forensic mode, the functionality of the first mode is extended by including the parameters shown in subsection 6.1.3. Technically, the two rule sets are stored as two transformation units that share a subset of rules. This reuse improves the maintainability of the implementation should additional logging features be incorporated. The source code for the light and forensic mode is available online[6].

In the concrete case of the POX controller submodule `l2_learning.py`, which installs flow entries on the switches in order to allow layer 2 switching, the function call that sends the `FlowMod` message is "`of.ofp_flow_mod()`". The part of the syntax tree that incorporates this function call is shown by the yellow edge (2) in Figure 6.2.

In order to modify the program to log flow entries in the database, the following additional source code is required:

```
1.0| #Change flag to receive flow_remove event
1.1| msg.flags = 1
2.0| msg.cookie = current_cookie #Set the flow identifier
3.0| #Sent flow to switch
3.1| l_scriptName = inspect.getfile(inspect.currentframe())
    # Script name
3.2| l_lineNumber = inspect.currentframe().f_back.f_lineno
    # Get line number
```

[6] git://github.com/shommes/POX.git

```
3.3| l_timestamp = datetime.datetime.now()# Timestamp
3.4| #Add flow entry to switch ---> key: FlowID | Field | Value
3.5| header = of.ofp_header()
3.6| self.dbHandler.addFlowEntry(msg, header, event.dpid, l_scriptName,
    l_lineNumber, l_timestamp)
3.7| #Set flow active --> SwitchID | FlowId | Value = 1
3.8| self.dbHandler.modifySwichEntry(event.dpid, msg.cookie)
4.0| #Increase the flow identifier (cookie)
4.1| current_cookie = current_cookie + 1
```

The source code can be divided into four code snippets, as indicated by the first number on each line. We focus on the third code snippet, which must be executed for each flow installation function call in order to write the flow entry into the database. The new abstract syntax tree in Figure 6.3 incorporates this functionality after the transformation process.

After the required modifications are finished, the syntax tree is transformed back into the Python language.

6.4 Conclusion

In this chapter, we have presented a concept for logging communication records in an OpenFlow-based network. In addition to the advantages it provides for fault detection and network debugging, the centralised network controller can access such flow records in order to provide access to network services. For instance, a load-balancing application could use the distribution of active flow entries in the network to optimise the traffic flow by adapting the flow entries to the current network state. To realise such a storage concept on the network controller, the source code of all existing network services must be modified. In order to reduce the potential for failure when this task is performed by humans, we present an approach based on graph transformation, which can automate the process and guarantees certain correctness properties for the modified programs. We have demonstrated the concept of automated source code extension in order to instrument a Python-based network controller with the functionality required in order to log flow records in a NoSQL database system.

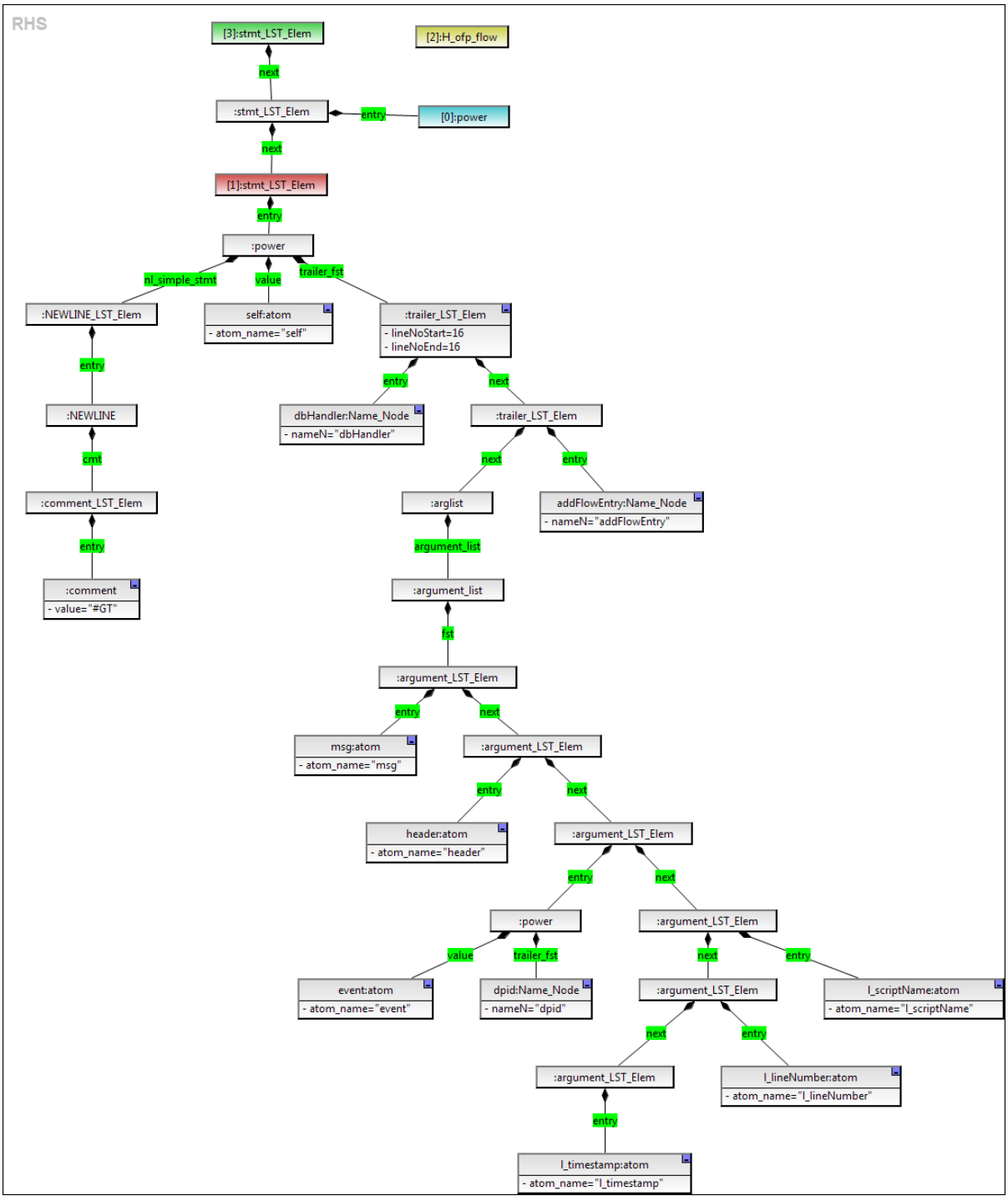


Figure 6.3: Example of graph transformation rule RHS: `extend_ofp_flow_full`

Chapter 7

Summary and Perspectives

7.1 Overall Conclusions

The centralized control plane, a key concept of Software-Defined Networking (SDN) brings many desirable advantages. Firstly, network policies and configuration can be defined at a single location, reducing an otherwise complex management overhead. Secondly, the same applies for fault detection facilities and security concepts, which benefit from centralised processing of all network events.

After discussing the general advantages of such an architecture, in chapter 3 we highlighted the important aspects of OpenFlow-based SDN for a deployment in an operational context. For hybrid networks that employ both SDN and non-SDN network devices, additional network management interfaces are required in order to allow protocols such as NetConf or SNMP to be used.

For network monitoring (chapter 4), the concept of OpenFlow allows flows to be observed at different levels of granularity, enabling the analysis of specific connections by breaking a macroflow into several microflows, which can be recombined after the examination process is complete. While a flow provides certain features that can be leveraged for traffic classification, the network-wide view of the network controller allows traffic variations to be identified based on information from all network devices. In order to determine such variations, we implemented a technique to measure the relative entropy in order to recognise attacks that are targeted at hosts in the network, as well as those that are aimed at the data plane of the network devices (e.g. MAC flooding attacks).

The approach described in chapter 5 concerns intrusion detection, which can be realised as a network service on the network controller. It allows examination of the packet payload and header fields, and can be used for security- and policy-based routing. For instance, the trust level of a sender can be determined by a whitelist or the type of

protocols involved, and can be further modified in accordance with a specific policy. A second example scenario is the examination of the payloads of several packets for malicious activity, with the communication only being allowed if no threat is identified. We demonstrated a similar application scenario by processing flow records originating from a firewall, and were able to identify attacks that were composed of multiple events. Furthermore, we processed packet level information in order to detect hidden communication patterns that are used to activate a stealthy backdoor. By tracking the TCP handshake for a series of flows, we were are able to determine such infrequent combinations using an algorithm from rare association rule mining.

Due to the ability of network services to access the communication records of all network devices, new network applications, such as creating a network snapshot or distributed firewalls based on network devices can be realised. We modified existing controller programs in chapter 6 to store flow entries and related parameters in a database system to allow network services and business applications to access them. To reduce the error-prone task of manually altering existing controller programs to realise such functionality, we proposed and implemented a technique for automated source code extension.

As a consequence, we believe that the key asset of having a software-defined control plane will increase the number of network applications, particularly those that cannot be realised in traditional networks due to their decentralised architecture. For networks with very specific requirements, OpenFlow-based SDN can provide solutions that are simpler and more flexible than is possible with non-SDN.

7.2 Limitations and Open Research Questions

The focus of this thesis as regards SDN is based on the concept of OpenFlow. Other approaches, described in section 2.3, are not covered in this thesis, since we believe that native OpenFlow is better suited for research due to its clear separation of data and control planes. Since OpenFlow enabled network devices are available from multiple vendors, new concepts can conveniently be validated in a testbed. This is not possible with the alternative architectures, since they are either still in a conceptual stage or are not available to the research community.

A second limitation is the focus on software-defined networks with network devices connected to a *single* network controller, as proposed in the OpenFlow specification. We believe that all approaches described in this thesis can also be applied in a multi-controller environment, but that this would simply increase implementation complexity without providing new insights into the concept underlying our work.

Nevertheless, existing research into OpenFlow-based SDN requires additional work. For the control channel, which will probably be realised in an in-bound mode for reasons of cost, further investigation into implementation issues is needed. Since the flow table

of each switch must handle flow entries related to controller communication, these must be managed properly in order to avoid a loss of connectivity to the network controller. Similarly, the bootstrapping of such networks requires more research into the steps required from the network devices, as well from the network controller side.

In addition to intrusion detection through controller-based packet inspection (chapter 5), security appliances such as Intrusion Detection Systems (IDSs) can be deployed in a network. Since only a fraction of the network traffic can be analysed by a single IDS, we propose a new concept, *elastic intrusion detection*. This uses a chain of IDSs, where the order and number of devices depend on the required service or policy. This is an example of *service chaining*, an idea being discussed in the IETF *Service Function Chaining (SFC)*[1] working group. The concept can be separated into the *service chains*, which define the general sequence of involved services, and the *service path*, which describes the actual path through the network and network devices involved.

While service chaining can be used to implement a variety of services, we think that it useful to focus on the specific case of intrusion detection. This allows the dynamic insertion of single or multiple IDSs into a flow path, for instance to adapt the level of security to the current amount of traffic in the network. As second application scenario is initially to forward all packets of a new flow to multiple IDSs, in order to determine whether the connection conforms to existing security policies and hence should be permitted. This increases the level of security, since a combination of IDSs can perform different types of packet inspection, for instance analysing packets with respect to the network, transport and application layers. If the result indicates no threat, the involved IDSs can be removed from the path in order to reduce their load.

The challenges for such a concept lie in determining the IDS service chains, as well as any new communication methodology that might be required in order to allow dependable communication between the IDSs and network controller. In order to avoid manual management of signatures on each IDS, a centralised archive located on the network controller could be used for this purpose. The network controller in SDN supports such a concept since it can influence the type and number of IDSs involved by adapting the packet forwarding rules on the network devices. This makes it possible to reuse existing security appliances, reducing the cost barrier for network operators wishing to implement such a facility in the form of a network service.

[1] https://datatracker.ietf.org/wg/sfc/charter/

List of Figures

List of Tables

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*. New York, NY, USA: ACM, 2007, pp. 1–12.

[3] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: a protection architecture for enterprise networks," in *Proceedings of the 15th USENIX Security Symposium (USENIX-SS'06)*, vol. 15. Berkeley, CA, USA: USENIX Association, 2006.

[4] Open Networking Foundation, "SDN Definition," 2013. [Online]. Available: https://www.opennetworking.org/index.php?option=com_content&view=article&id=686&Itemid=272&lang=en

[5] D. W. Wall, "Messages as active agents," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82)*. New York, NY, USA: ACM, 1982, pp. 34–39.

[6] J. Zanderr and R. Forchheimer, "Preliminary specification of a distributed packet radio system using the amateur band," University of Linköping, Sweden, Tech. Rep., 1980.

[7] J. Zander and R. Forchheimer, "SOFTNET – an approach to high level packet radio," in *AMRAD Conf.*, 1983.

[8] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, "A survey of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 2, pp. 7–23, Apr. 1999.

[9] J. Smith and S. Nettles, "Active networking: one view of the past, present, and future," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 34, no. 1, pp. 4 –18, Feb. 2004.

[10] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A survey of active network research," *Communications Magazine, IEEE*, vol. 35, no. 1, pp. 80–86, Jan. 1997.

[11] J. Biswas, A. A. Lazar, J. F. Huard, K. Lim, S. Mahjoub, L. F. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein, "The IEEE P1520 standards initiative for programmable network interfaces," *Comm. Mag.*, vol. 36, no. 10, pp. 64–70, Oct. 1998.

[12] R. Stadler and B. Stiller, Eds., *Active Technologies for Network and Service Management, 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'99)*, ser. Lecture Notes in Computer Science, vol. 1700. Springer, Oct. 1999.

[13] M. Brunner, B. Plattner, and R. Stadler, "Service creation and management in active telecom networks," *Commun. ACM*, vol. 44, no. 4, pp. 55–61, Apr. 2001.

[14] M. Brunner and R. Stadler, "Management in telecom environments that are based on active networks," *J. High Speed Netw.*, vol. 9, no. 3,4, pp. 213–230, Dec. 2000.

[15] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, *NETCONF Configuration Protocol (IETF RFC 6241)*, The Internet Society, June 2011.

[16] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe, "The case for separating routing from routers," in *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA'04)*. New York, NY, USA: ACM, 2004, pp. 5–12.

[17] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai, "Tesseract: a 4D network control plane," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. Berkeley, CA, USA: USENIX Association, 2007, pp. 27–27.

[18] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 5, pp. 41–54, Oct. 2005.

[19] L. Dong, R. Gopal, and J. Halpern, *Forwarding and Control Element Separation (ForCES) Protocol Specification (IETF RFC 5810)*, The Internet Society, Mar. 2004.

[20] Metaswitch Networks, "PCE – an evolutionary approach to SDN," 2012. [Online]. Available: http://www.metaswitch.com/sites/default/files/metaswitch-white-paper-pce-an-evolutionary-approach-to-sdn.pdf

[21] A. Farrel, J.-P. Vasseur, and J. Ash, *A Path Computation Element (PCE)-Based Architecture (IETF RFC 4655)*, The Internet Society, Aug. 2006.

[22] A. Atlas, J. Halpern, S. Hares, D. Ward, and T. Nadeau, "An architecture for the interface to the routing system (draft)," The Internet Society, 2013. [Online]. Available: https://ietf.org/doc/draft-ietf-i2rs-architecture/

[23] S. Kiran and G. Kinghorn, *Cisco Open Network Environment: Bring the Network Closer to Applications*, CISCO Systems, 2013. [Online]. Available: http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9902/white_paper_c11-728045.pdf

[24] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: a retrospective on evolving SDN," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. New York, NY, USA: ACM, 2012, pp. 85–90.

[25] U. Hoelzle, "OpenFlow @ Google," 2012. [Online]. Available: http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf.

[26] HP Networking, "Software defined networks (SDN)." [Online]. Available: http://h17007.www1.hp.com/us/en/mobile/solutions/tech/sdn.html

[27] Juniper Networks, "OpenFlow Switch Application (OF-APP) for Juniper MX-Series Routers." [Online]. Available: https://developer.juniper.net/shared/jdn/docs/ProgrammableNetworks/OpenFLow_APP_JDN_Overview.pdf

[28] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[29] "About POX," 2012. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[30] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker, "Frenetic: a high-level language for OpenFlow networks," in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO'10)*. New York, NY, USA: ACM, 2010, pp. 6:1–6:6.

[31] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," *USENIX ;login*, vol. 38, no. 5, pp. 128–134, Oct. 2013.

[32] *ONF White Paper: Software-Defined Networking: The New Norm for Networks*, Open Networking Foundation, 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf

[33] *OpenFlow Switch Specification - Version 1.4.0*, Open Networking Foundation, 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[34] J. Quittek, T. Zseby, B. Claise, and S. Zander, *Requirements for IP Flow Information Export (IPFIX) (IETF RFC 3917)*, The Internet Society, Oct. 2004.

[35] B. Claise, *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information (IETF RFC 5101)*, The Internet Society, Jan. 2008.

[36] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "FlowVisor: A network virtualization layer," Deutsche Telekom Inc. R&D Lab, Stanford University, Nicira Networks, Tech. Rep., 2009. [Online]. Available: http://archive.openflow.org/downloads/technicalreports/openflow-tr-2009-1-flowvisor.pdf

[37] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed multi-domain SDN controllers," *CoRR*, 2013.

[38] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Fast failure recovery for in-band OpenFlow networks," in *9th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2013, pp. 52–59.

[39] ——, "Openflow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, no. 6, pp. 656–665, Mar. 2013.

[40] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, "Software defined networking: Meeting carrier grade requirements," in *18th IEEE Workshop on Local Metropolitan Area Networks (LANMAN)*, 2011, pp. 1–6.

[41] *IEEE Standard for local and metropolitan area networks — Media Access Control (MAC) Bridges (IEEE Std 802.1D-2004)*, 2004. [Online]. Available: http://standards.ieee.org/getieee802/download/802.1D-2004.pdf

[42] D. Katz and D. Ward, *Bidirectional Forwarding Detection (BFD) (IETF RFC 5880)*, The Internet Society, June 2010.

[43] *IEEE Standard for local and metropolitan area networks — Station and Media Access Control Connectivity Discovery (IEEE Std 802.1AB-2009)*, 2009. [Online]. Available: http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf

[44] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Skoldstrom, "Scalable fault management for OpenFlow," in *IEEE International Conference on Communications (ICC)*, 2012, pp. 6606–6610.

[45] M. Desai and T. Nandagopal, "Coping with link failures in centralized control plane architectures," in *Second International Conference on Communication Systems and Networks (COMSNETS)*, 2010, pp. 1–10.

[46] Y. Wang, Y. Zhang, V. Singh, C. Lumezanu, and G. Jiang, "Netfuse: Short-circuiting traffic surges in the cloud," in *IEEE International Conference on Communications (ICC)*, 2013.

[47] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "Flowsense: monitoring network utilization with zero measurement cost," in *Proceedings of the 14th International Conference on Passive and Active Measurement (PAM'13)*. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 31–41.

[48] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI'13)*. Berkeley, CA, USA: USENIX Association, 2013, pp. 29–42.

[49] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches," in *Proceedings of the 11th USENIX Conference on Hot topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE'11)*. Berkeley, CA, USA: USENIX Association, 2011, pp. 13–13.

[50] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *35th Conference on Local Computer Networks (LCN'10)*, Oct. 2010, pp. 408 –415.

[51] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic matrix estimator for OpenFlow networks," in *Passive and Active Measurement*, ser. Lecture Notes in Computer Science, A. Krishnamurthy and B. Plattner, Eds. Springer Berlin Heidelberg, 2010, vol. 6032, pp. 201–210.

[52] H. Khan, S. A. Khayam, M. Rajarajan, L. Golubchik, and M. Orr, "Wirespeed, privacy-preserving P2P traffic detection on commodity switches," in *under submission*, 2013. [Online]. Available: http://www.xflowresearch.com/docs/Wirespeed_Privacy-Preserving_P2P_Traffic_Detection_on_Commodity_Switches.pdf

[53] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. New York, NY, USA: ACM, 2012, pp. 55–60.

[54] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT way for Openflow switch interoperability testing," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'12)*. New York, NY, USA: ACM, 2012, pp. 265–276.

[55] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'12)*. New York, NY, USA: ACM, 2012, pp. 241–252.

[56] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: enabling record and replay troubleshooting for networks," in *Proceedings of the USENIX Annual Technical Conference (USENIXATC'11)*. Berkeley, CA, USA: USENIX Association, 2011, pp. 29–29.

[57] *Conformance Test Specification for OpenFlow Switch Specification 1.0.1*, Open Networking Foundation, 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-test/conformance-test-spec-openflow-1.0.1.pdf

[58] Project Floodlight, "OFTest," 2013. [Online]. Available: http://www.projectfloodlight.org/oftest/

[59] B. Heller, C. Scott, M. Nick, S. Scott, W. Andreas, Z. Hongyi, W. Sam, V. Jeyakumar, N. Handigol, M. McCauley, K. Zarifis, and P. Kazemian, "Leveraging SDN layering to systematically troubleshoot networks," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*. ACM, 2013.

[60] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: verifying network-wide invariants in real time," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. New York, NY, USA: ACM, 2012, pp. 49–54.

[61] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with Anteater," in *Proceedings of the ACM SIGCOMM Conference*. New York, NY, USA: ACM, 2011, pp. 290–301.

[62] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: static checking for networks," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9.

[63] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets'10)*. New York, NY, USA: ACM, 2010, pp. 19:1–19:6.

[64] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test Openflow applications," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10.

[65] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 161–180.

[66] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. New York, NY, USA: ACM, 2012, pp. 121–126.

[67] S. Schechter, J. Jung, and A. Berger, "Fast detection of scanning worm infections," in *Recent Advances in Intrusion Detection (RAID'04)*, ser. Lecture Notes in Computer Science, E. Jonsson, A. Valdes, and M. Almgren, Eds. Springer Berlin Heidelberg, 2004, vol. 3224, pp. 59–81.

[68] J. Twycross and M. M. Williamson, "Implementing and testing a virus throttle," in *Proceedings of the 12th USENIX Security Symposium (SSYM'03)*, vol. 12. Berkeley, CA, USA: USENIX Association, 2003, pp. 20–20.

[69] Y. Gu, A. McCallum, and D. Towsley, "Detecting anomalies in network traffic using maximum entropy estimation," in *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement (IMC'05)*. Berkeley, CA, USA: USENIX Association, 2005, pp. 32–32.

[70] M. V. Mahoney, "Network traffic anomaly detection based on packet bytes," in *Proceedings of the ACM Symposium on Applied Computing (SAC'03)*. New York, NY, USA: ACM, 2003, pp. 346–350.

[71] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined networks," in *Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[72] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "OpenFlow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. New York, NY, USA: ACM, 2012, pp. 127–132.

[73] J. Mudigonda, P. Yalagandula, J. Mogul, B. Stiekes, and Y. Pouffary, "Netlord: A scalable multi-tenant network architecture for virtualized datacenters," in *Proceedings of the ACM SIGCOMM 2011 Conference*. ACM, 2011, pp. 62–73.

[74] *OpenFlow Management and Configuration Protocol (OF-Config 1.1.1)*, Open Networking Foundation, 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1-1-1.pdf

[75] M. Bjorklund, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF) (IETF RFC 6020)*, The Internet Society, Oct. 2010.

[76] J. Case, M. Fedor, M. Schoffstall, and J. Davin, *A Simple Network Management Protocol (SNMP) (IETF RFC 1157)*, The Internet Society, May 1990.

[77] K. McCloghrie, D. Perkins, and J. Schoenwaelder, *Structure of Management Information Version 2 (SMIv2) (IETF RFC 2578)*, The Internet Society, April 1999.

[78] S. Chisholm and D. Romascanu, *Alarm Management Information Base (MIB) (IETF RFC 3877)*, The Internet Society, Sept 2004.

[79] K. McCloghrie and M. R. and, *Management Information Base for Network Management of TCP/IP-based internets: MIB-II (IETF RFC 1213)*, The Internet Society, March 1991.

[80] R. Finlayson, *Bootstrap Loading using TFTP (IETF RFC 906)*, The Internet Society, June 1984.

[81] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Automatic bootstrapping of OpenFlow networks," in *19th IEEE Workshop on Local Metropolitan Area Networks (LANMAN'13)*, 2013, pp. 1–6.

[82] A. Clemm and M. Hartwig, "NETradamus: A forecasting system for system event messages," in *Network Operations and Management Symposium (NOMS'10)*, April 2010, pp. 623–630.

[83] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Interactive learning of alert signatures in high performance cluster system logs." in *Network Operations and Management Symposium (NOMS'12)*, 2012, pp. 52–60.

[84] T. V. Lillard, C. P. Garrison, C. A. Schiller, and J. Steele, *Digital Forensics for Network, Internet and Cloud Computing: A Forensic Evidence Guide for Moving Targets and Data*. Syngress, 2010.

[85] B. Claise, *Cisco Systems NetFlow Services Export Version 9 (IETF RFC 3954)*, The Internet Society, Oct. 2004.

[86] E. Hellinger, "Neue Begründung der Theorie quadratischer Formen von unendlichvielen Veränderlichen." *Journal für die Reine und Angewandte Mathematik*, vol. 136, pp. 210–271, 1909.

[87] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[88] S. Wise and D. Fair, *Innovative Control Charting: Practical SPC Solutions for Today's Manufacturing Environment*, ser. Quality and reliability. Asq Quality Press, 1998.

[89] J. M. Lucas and M. S. Saccucci, "Exponentially weighted moving average control schemes: Properties and enhancements," *Technometrics*, vol. 32, no. 1, pp. pp. 1–12, 1990.

[90] S. Hommes, R. State, and T. Engel, "A distance-based method to detect anomalous attributes in log files," in *IEEE Network Operations and Management Symposium (NOMS)*, 2012, pp. 498–501.

[91] ——, "Classification of log files with limited labeled data," in *Principles, Systems and Applications of IP Telecommunications (IPTComm)*, Chicago, USA, Oct. 2013.

[92] ——, "Detecting stealthy backdoors with association rule mining," in *Proceedings of the 11th international IFIP TC 6 conference on Networking - Volume Part II*, ser. IFIP. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 161–171.

[93] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley Publishing, 2008.

[94] B. Guha and B. Mukherjee, "Network security via reverse engineering of TCP code: Vulnerability analysis and proposed solutions," *IEEE Network: The Magazine of Global Internetworking*, vol. 11, no. 4, pp. 40–48, July 1997.

[95] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 222–232, 1987.

[96] Western Electric, *Statistical Quality Control Handbook*. Western Electric Corporation, Indianapolis, Ind., 1956.

[97] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," School of Computer Science - Carnegie Mellon University, Tech. Rep., 2002.

[98] P. P. Talukdar and K. Crammer, "New regularized algorithms for transductive learning," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part II*, ser. ECML PKDD. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 442–457.

[99] FunOverIP, *cD00r knocking backdoor (improved)*, 2011. [Online]. Available: http://funoverip.net/2011/03/cd00r-knocking-backdoor-improved/

[100] G. Hay, "Extending the packet coded backdoor server to netcat relays on relatively high-bandwidth home networks," SANS, Tech. Rep., 2001.

[101] Y. Jonathan, *Use port knocking to bypass firewall rules and keep security intact*, 2005. [Online]. Available: http://www.techrepublic.com/article/use-port-knocking-to-bypass-firewall-rules-and-keep-security-intact/

[102] Phenoelit, *cd00r.c - packet coded backdoor*, 2000. [Online]. Available: http://www.phenoelit-us.org/stuff/cd00r.c

[103] C. M. Nyberg, *Sadoor*, 2013. [Online]. Available: http://http://packetstormsecurity.com/search/?q=sadoor

[104] S. Miklosovic, *PA018 - Term Project - Port knocking enhancements*, 2011. [Online]. Available: http://www.portknocking.org/view/resources

[105] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 1993, pp. 207–216.

[106] B. Liu, W. Hsu, and Y. Ma, "Mining association rules with multiple minimum supports," in *Knowledge Discovery and Data Mining*, 1999, pp. 337–341.

[107] X. Wu, C. Zhang, and S. Zhang, "Efficient mining of both positive and negative association rules," *ACM Trans. Inf. Syst.*, vol. 22, pp. 381–405, July 2004.

[108] Y. S. Koh and N. Rountree, *Rare Association Rule Mining and Knowledge Discovery: Technologies for Infrequent and Critical Event Detection*. Information Science Reference - Imprint of: IGI Publishing, 2009.

[109] S. Hommes, F. Hermann, R. State, and T. Engel, "Automated source code extension for debugging of OpenFlow based networks," in *9th International Conference on Network and Service Management (CNSM)*, Zurich, Switzerland, Oct. 2013.

[110] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann, "Theory of constraints and application conditions: From graphs to high-level structures," *Fundamenta Informaticae*, vol. 74, no. 1, pp. 135–166, 2006.

[111] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, pp. 31–39, 2008.

[112] F. Hermann, S. Hommes, R. State, and H. Ehrig, "Correctness of source code extension for fault detection in OpenFlow based networks," SnT, University of Luxembourg, Tech. Rep. TR-SnT-2014-2, 2014, ISBN 978-2-87971-116-4. [Online]. Available: http://hdl.handle.net/10993/15749

[113] *Xtext - Language Development Framework – Version 2.3*, Eclipse Consortium, 2012. [Online]. Available: http://www.eclipse.org/Xtext/

[114] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced concepts and tools for in-place EMF model transformations," in *Proc. MoDELS'10*, ser. LNCS, vol. 6394. Springer, 2010, pp. 121–135.