Technische Universität München
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Kommunikationsnetze

Fine-grained Isolation and Filtering of Network traffic using SDN and NFV

Raphael Durner, M.Sc.

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr.-Ing. Georg Sigl
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. Wolfgang Kellerer
	2.	Prof. Dr.-Ing. Georg Carle

Die Dissertation wurde am 29.10.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 21.09.2020 angenommen.

Abstract

With increasing importance of computer networks in general, also intentionally adverse activities have increased. In order to protect the integrity of the clients and data in a network, network security concepts have been developed in the last decades. Nevertheless, network security today is mostly enforced on the edge of the network. Thus, connections and packets in the network cannot be checked and security within one network cannot be enforced. An adversary that has breached the boundaries of the network can spread in the network. Large scale incidents like carried out by the WannaCry malware have shown that this is a major shortcoming of today's network security concepts.

With the advent of Software-Defined Networking (SDN) a unique possibility arises to resolve this issue. SDN can be used to provide fine-grained filtering in the network. With SDN the control of the network and the network view is centralized. The header of the network packets can be used directly to isolate different virtual networks. This enables a connection level isolation within the network. However, the filtering capabilities of this SDN approach are limited by the capabilities of the SDN hardware devices. Thus, Network Function Virtualization (NFV) is used in addition to provide filtering options such as stateful and application layer filtering.

Several challenges arise:

SDN is not yet a mature technology. However, it is integral part of the envisioned security architecture. Therefore, in this thesis at first security issues that arise in the SDN architecture are shown and selected security improvements are introduced.

Due to its roots in a security project, virtual network isolation in terms of access is quite straight forward in SDN networks. Nevertheless, access isolation is not sufficient as an overload of one virtual network leads to contention in other networks if physical links are shared.

The NFV concept utilizes IT virtualization technologies for network processing purposes. In contrast to classical workloads in this field, Virtualized Network Functions (VNFs) are quite different as they require high throughput of network packets. This work analyzes performance

issues of the employed hardware and introduces improvements for a better performance of the VNFs.

VNFs run in a centralized manner utilizing the data center like Network Function Virtualization Infrastructure (NFVI). The presented architecture features filtering using NFV. Thus all traffic must be routed through the NFVI which causes detours. An approach is analyzed that reduces the load in the network using SDN offloading. One main challenge in this area is to identify those flows that are worth offloading.

Kurzfassung

Mit der zunehmenden Bedeutung von Computernetzen im Allgemeinen haben auch die Angriffe auf diese Netze zugenommen. Um die Integrität der Teilnehmer in einem Netz und die dort gespeicherten Daten zu schützen, wurden in den letzten Jahrzehnten Netzsicherheitskonzepte entwickelt. Trotzdem wird die Netzsicherheit heutzutage meist nur am Rand des Netzes durchgesetzt. Daher können Verbindungen im Netz nicht überprüft und die Sicherheit innerhalb eines Netzes nicht durchgesetzt werden. Ein Angreifer, der diese Sicherheitsmaßnahmen am Rand des Netzes überwunden hat, kann sich frei im Netz ausbreiten. Groß angelegte Vorfälle wie die WannaCry-Malware haben gezeigt, dass dies ein wesentlicher Mangel heutiger Netzsicherheitskonzepte ist.

Mit dem Aufkommen von Software-Defined Networking (SDN) entsteht eine neue Möglichkeit, diese Herausforderung zu lösen. In dieser Arbeit zeigen wir, wie SDN verwendet werden kann, um eine feinkörnige Filterung im Netz bereitzustellen. Mit SDN wird die Kontrolle über das Netz und die Netzsicht zentralisiert. Der Header der Netzpakete kann direkt verwendet werden um verschiedene virtuelle Netze zu isolieren. Dies ermöglicht eine Isolation auf Verbindungsebene innerhalb des Netzes.

Die Filterfunktionen dieses SDN-Ansatzes sind jedoch durch die Funktionen der SDN-Hardwaregeräte beschränkt. Daher wird zusätzlich Network Function Virtualization (NFV) verwendet, um Optionen wie zustandsbehaftete Filterung und Filterung der Anwendungsschicht bereitzustellen.

Hieraus ergeben sich mehrere Herausforderungen:

SDN ist noch keine ausgereifte Technologie. Es ist jedoch integraler Bestandteil der gezeigten Sicherheitsarchitektur. Daher werden in dieser Arbeit Sicherheitsprobleme, die in der SDN Architektur auftreten, aufgezeigt und ausgewählte Sicherheitsverbesserungen vorgestellt.

Aufgrund der Wurzeln von SDN in einem Sicherheitsprojekt ist die Isolierung virtueller Netze in Bezug auf den Zugriff in SDN Netzen ohne große Herausforderungen möglich. Die Zugriffsisolation ist jedoch nicht ausreichend, da eine Überlastung eines virtuellen Netzes zu

Konflikten in anderen Netzen führt, wenn physische Netzverbindungen gemeinsam genutzt werden.

Das NFV-Konzept nutzt IT-Virtualisierungstechnologien zur Bereitstellung von Netzfunktionen. Im Gegensatz zu klassischen Workloads in diesem Bereich haben VNFs andere Anforderungen an die Infrastruktur, da sie einen hohen Durchsatz von Netzpaketen erfordern. Diese Arbeit analysiert Leistungsprobleme der genutzten Hardware und führt Verbesserungen der Leistung der VNFs ein.

VNFs werden zentral mithilfe einer Infrastruktur bereitgestellt, die einem Rechenzentrum ähnelt. Die vorgestellte Architektur bietet mit dieser Hilfe eine Filterung an. Daher muss der gesamte Datenverkehr über die Infrastruktur geleitet werden, dies verursacht Umwege. Wir zeigen und analysieren einen Ansatz, der die Netzlast mithilfe von SDN Offloading reduziert. Eine Hauptherausforderung in diesem Bereich besteht darin, diejenigen Flüsse zu identifizieren, die es wert sind, auf die SDN Geräte umgelenkt zu werden.

Contents

1 Introduction 1
1.1 Requirements . . . 4
1.2 Research Challenges . . . 5
1.3 Contributions . . . 6
1.3.1 Software Defined Networking . . . 6
1.3.2 Network Function Virtualization . . . 6
1.3.3 SDN and NFV Offloading . . . 6
1.3.4 Thesis Outline . . . 7

2 Background and Related Work 9
2.1 Software Defined Networking . . . 9
2.2 Network Function Virtualization . . . 11
2.3 SDN and NFV . . . 11
2.4 Related security architectures in literature . . . 11
2.5 Commercial solutions . . . 13
2.6 Summary . . . 14

3 Securing SDNs 15
3.1 Attack vectors in Software Defined Networks . . . 16
3.1.1 Rogue SDN Application . . . 16
3.1.2 Man in the Middle Attack to the Control Connection . . . 17
3.1.3 Rogue SDN Controller . . . 18
3.1.4 Spoofing Attack . . . 18
3.1.5 Denial of Service Attacks . . . 19
3.2 Related Work . . . 21
3.2.1 Security Analyses . . . 21
3.2.2 Attacks and Countermeasures . . . 22
3.2.3 Summary . . . 24

3.3 Securing the Control Plane of Software Defined Networks 24
3.3.1 Measurement Setup 25
3.3.2 Measurement Results and discussion 26
3.4 Securing Software Defined Networks against DoS attacks 27
3.4.1 Detection and Mitigation of DoS attacks against SDNs 28
3.4.1.1 Detection 28
3.4.1.2 Counter Measures 30
3.4.2 Evaluation of the approach 31
3.4.2.1 Abstracted Simulation 31
3.4.2.2 Analytic Evaluation of the Detection Performance 32
3.4.2.3 Simulation Results 33
3.5 Summary and Discussion 35

4 Isolation in SDNs **37**
4.1 Isolation in OpenFlow enabled Networks 39
4.1.1 Related Work 40
4.1.2 Fine-grained virtual networks with OpenFlow 41
4.2 Performance Isolation in OpenFlow enabled Networks 42
4.2.1 Related Work 43
4.2.2 Background on Queuing Disciplines 45
4.2.3 Measurement Setup 46
4.2.3.1 Experiment Setup 46
4.2.3.2 Investigated Switches 48
4.2.4 Measurement Results 48
4.2.4.1 Priority Queuing 48
4.2.4.2 Queuing Disciplines with Guaranteed Bandwidth 50
4.2.4.3 PQ and TCP's Retransmission Behavior 52
4.2.4.4 Inverted Priority Queuing 53
4.3 Summary and Discussion 54

5 Performance of Security VNFs **55**
5.1 Server Memory Architecture 57
5.1.1 NUMA Architecture 58
5.1.2 Cache Hierarchy 58
5.2 Related Work 59
5.2.1 NUMA Architecture 59
5.2.2 LLC Interference Management 60

5.3 Impacts of the NUMA architecture to VNF performance 62
5.3.1 Methodology 62
5.3.1.1 Scenario and Test Environment 63
5.3.1.2 Minimal VNF 63
5.3.1.3 Function Chain Implementation 64
5.3.1.4 Measuring CPU Load with Polling 64
5.3.2 Evaluation 65
5.3.2.1 NUMA Impact - Minimal VNF 65
5.3.2.2 NUMA Impact - Function Chain 67
5.3.2.3 Impact of Cache Exhaustion 67
5.4 NFV Last Level Cache Scheduler 70
5.4.1 NFV MANO 71
5.4.2 Cache Allocation Technology 72
5.4.3 Memory Access Model 72
5.4.4 Optimal Scheduler Design 75
5.4.4.1 Optimization objective 75
5.4.4.2 Algorithm 75
5.4.4.3 Example Run 76
5.4.4.4 Optimality Discussion 78
5.4.5 Evaluation 79
5.4.5.1 Experiment Design 80
5.4.5.2 Monitoring 80
5.4.5.3 Transient phase of the LLC 80
5.4.5.4 Scheduler Convergence 81
5.4.5.5 Scheduler Gain 82
5.5 Summary and Discussion 84

6 SDN Offloading 87
6.1 Related Work 89
6.1.1 SDN Offloading 90
6.1.2 Elephant flow detection 91
6.1.3 Traffic Classification 92
6.2 Offloading with the first packet 92
6.2.1 Gathering the training-data 93
6.2.2 Features 95
6.2.3 Machine Learning Algorithms 96
6.3 Sampling based approach 98

6.3.1 Baseline Algorithm 98
6.3.2 Table restricted approach Sample+ 99
6.4 Evaluation of the Machine Learning approach 100
6.4.1 Data Sets 100
6.4.2 Data Labeling 101
6.4.3 Feature selection 103
6.4.4 Merging infrequent nominal values 103
6.4.5 Algorithm evaluation 105
6.4.6 Classification Complexity 108
6.4.7 Temporal stability 109
6.5 Evaluation of the sampling approach 111
6.6 Summary and Discussion 112

7 VNF NIC Offloading 117
7.1 Related Work 119
7.1.1 Software network functions 119
7.1.2 Stateful Hardware and Hybrid Network Functions 120
7.1.3 NIC Offloading 121
7.2 Background 121
7.2.1 Stateful Load Balancers 121
7.2.2 Intel Flow Director 122
7.3 Implementation 123
7.4 Assessing utilization 124
7.5 Performance Evaluation 126
7.5.1 Maximum Throughput 127
7.5.2 Utilization 129
7.6 Discussion 131

8 Conclusion and Outlook 133
8.1 Summary 133
8.2 Outlook 134

Bibliography 135

Acronyms 149

List of Figures 153

List of Tables 159

Chapter 1

Introduction

Digital services and especially the internet have changed society and economy fundamentally. Many aspects of life and work, like for example communication, trading and entertainment already depend on digital services. Upcoming trends like IoT, Industry 4.0 or automated driving will increase the dependency even more. Thus attacks can have more and more severe consequences and network security becomes even more important.

At the time of the introduction of the internet and the development of its main protocols, network security was not considered. During that time the number of users was very low and nobody could foresee the growth in importance nor the technological developments since then. This explains why some of the very basic protocols like ARP, TCP, or IP have no security built in.

In order to overcome these issues, firewalls and other security network functions were introduced. These functions are used to filter the traffic (allow some packets while dropping others) based on security rules that are provided by the security administrator as Access Control List (ACL). Though these security functions are a mere workaround and do not solve the underlying shortcomings completely. To make things worse, firewalls are nowadays mostly deployed between different network segments. Consequently, packets within one network segment can not be filtered and only connections and packets leaving and entering the network are filtered.

Figure 1.1 shows a widely used approach for a security architecture in an enterprise network: The perimeter gateway firewall is filtering the packets between the external network, i.e. the internet, and the private, local network. The perimeter gateway firewall can be realized in conjunction with the gateway to the external network. The gateway as such forwards all packets that are exchanged between the external and the private network. This approach has the drawback, that an adversary that has breached the boundaries of the network, can spread in the network without any countermeasures by the network administrator.

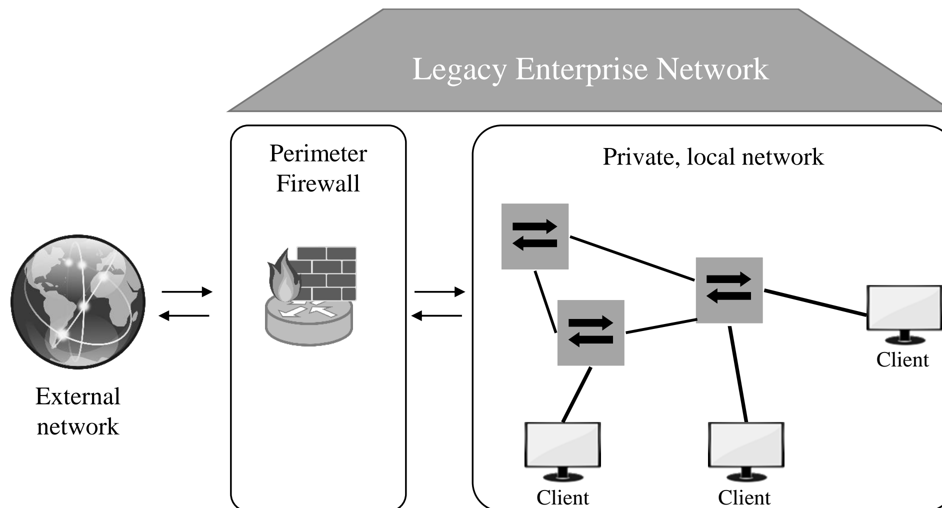


Figure 1.1: Conventional legacy security architecture of an enterprise network

As a result, new security vulnerabilities like the smb vulnerability used by the WannaCry worm [10] can only be resolved with security updates for all clients in the network. Especially for embedded devices like printers, Smart-TVs or IoT devices is this a tedious duty and sometimes even impossible if the device is no longer supported by the vendor.

This is especially urgent in enterprise networks as these networks have to deal with a wide range of different clients that are needed by the employees. Further the security demands of enterprise networks are higher compared to home or provider networks.

Fortunately, the advent of Software-Defined Networking (SDN) provides a unique possibility to overcome these issues: SDN centralizes the control of the network and therefore provides means to filter all packets in the whole network from a central perspective. Centralization is reached by separating the control from the data plane. The control plane is logically centralized in one or more software entities refereed as SDN- or Network controller(s) [11]. The data plane is realized by simple switches with low complexity that forward the packets according to rules given by the controller. Communication between control and data-plane is realized using a communication protocol. One of the first appearances of this concept was presented with the network architecture Ethane [12], that lead to the most prominent SDN protocol OpenFlow [13]. One main advantage of this concept is the centralized visibility and control of the network. In contrast to legacy networks that are controlled in a distributed manner, SDN provides means to control the network centrally. This enables an improved network security as packets can be filtered without using middle boxes, such as perimeter gateway firewalls using the SDN devices. Moreover SDN switches can also provide filtering in the network as they connect all hosts. Furthermore the visibility is increased, which can lead to improvements in terms of attack detection.

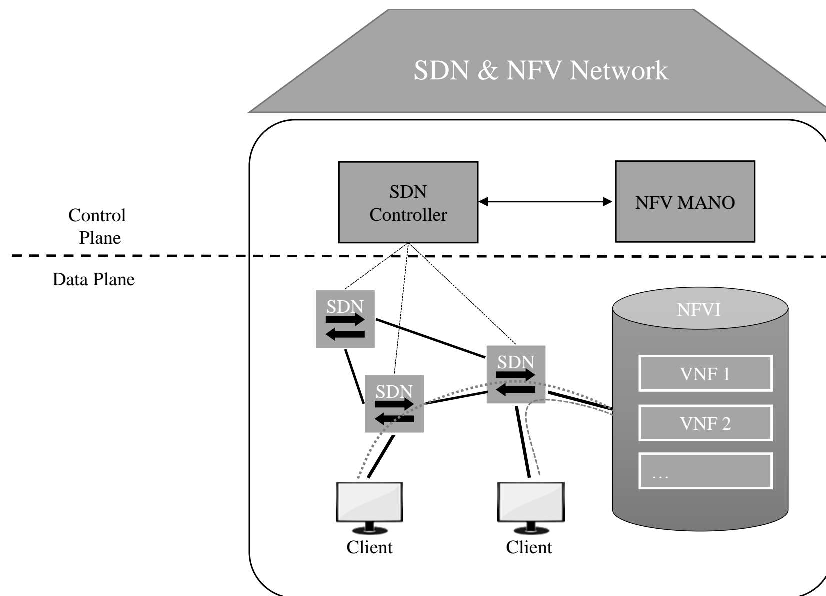


Figure 1.2: SDN/NFV enabled security architecture with fine-grained isolation

In addition Network Function Virtualization (NFV) [14] provides a novel way to implement and deploy security network functions: With the move to NFV, network functions are realized in software using commodity hardware. This reduces costs and increases the flexibility, as virtualization techniques can be used. Further, compute, storage and networking resources are virtualized. Network functions are no longer deployed on fixed appliances but are deployed as Virtualized Network Functions (VNFs) utilizing the virtualized resources. NFV greatly increases the flexibility of the network functions. The network functions can be scaled and migrated to fit to the demands of the network traffic. Furthermore new network functions can be deployed and developed faster. In terms of network security filtering options open up that were not feasible in the past due to performance restrictions. For example application layer firewalls or Intrusion Detection Systems (IDSs) requires substantial compute resources that can be provided more easily using NFV.

As a drawback NFV and SDN also open up security threats. For instance the virtualization introduced by NFV gives up on physical isolation of the security instances. Also SDN introduces new attack vectors: E.g., the centralization of the control plane introduces a single point that gives an adversary many possibilities if he manages to take it over.

Based on these enablers our aim is to develop an SDN/NFV security architecture for enterprise networks as shown in Figure 1.2. In contrast to legacy network we aim to filter traffic not only at the boundaries of the network but also in the network. SDN provides the necessary tools to isolate traffic on a connection level. Further the architecture shall provide means to filter the traffic on a stateful or application level. Due to the large amount of traffic and the large number of connections that must be filtered, the performance of the security

solution can be an issue. The architecture shall use the NFV paradigm in order to provide the necessary performance. With NFV security network functions can be deployed in Network Function Virtualization Infrastructure (NFVI) flexibly based on the network demand. The functions of NFV Management and Orchestration (MANO) framework control the VNFs and other software functions in the NFVI.

1.1 Requirements

Thus, the architecture shall fulfill the following requirements:

R1 Isolation of virtual networks The first requirement is the effectiveness of the isolation between the different virtual networks, in terms of access and performance. Access isolation hinders adversarys from spreading in the network and performance isolation prohibits that overloading of one virtual network by an adversary has adversarial effects to other virtual networks in the same physical network. The architecture shall support an omni-present fine grained access control throughout the network that is a new measure for securing campus and enterprise networks within the boundaries of the network. The solution aims at providing isolation between connections, both in terms of access and in terms of performance.

R2 Stateful and application layer filtering The proposed architecture shall not be limited to stateless filtering, instead we also aim in supporting stateful and application layer filtering. Packet filters can be categorized into stateless, stateful and application layer packet filters. A simple packet filter that can only drop or allow packets based on the header of packets is called stateless. Stateful packet filters are also able to track the state of a network connection and provide filtering capabilities based on the state of connections, e.g., only drop packets from new connections. Further application layer filters support more advanced filtering possibilities, one example would be the filtering of JavaScript content in an Hypertext Transfer Protocol (HTTP) connection. Stateless filtering can be provided by means of SDN. Stateful and application layer filtering can not be done by current hardware SDN switches, instead we propose to use security VNFs.

R3 Performance Both security requirements, fine-grained and application layer filtering of the network traffic, increase the load on the respective VNFs. However, security solutions are only deployed if they can fulfill the performance demand of the users. Consequently the scalability of the proposed solution is another key requirement.

1.2 Research Challenges

The described requirements for an SDN/NFV enabled network security architectures raise several research questions, both in terms of security and in terms of performance.

C1 Secure Operation of SDN By adding new technologies like SDN it has to be ensured that the technology can be operated in a secure way. SDN splits the control from the data plane, this causes new attack vectors that must be investigated. Further, SDN as technology is still in an early stage and even the support of basic security measures is not self-evident. In this work, we show the main security consideration and show improvements to the SDN architecture in order to support Requirement R1 of an increased network security using SDN and NFV.

C2 Isolation of SDN Based on the secure operation of the SDN connections can be isolated in a fine-grained manner. However, isolation in terms of access is not sufficient for all applications as it does not guarantee performance isolation as requested by Requirement R1. Therefore besides access isolation also performance isolation techniques have to be studied. Specifically performance isolation suffers from hardware effects, as the underlying physical hardware is shared due to virtualization.

C3 Performance of security VNFs Using SDN we can provide a fine-grained isolation of the connections, which corresponds to a stateless filtering of network packets. Nevertheless, stateless filtering is not enough to secure the network against attacks that can only be averted using stateful or application layer filtering. Hence in order to fulfill Requirement R2 we utilize the NFV concept in order to provide the resources to use stateful and application layer filtering in the network. However, the move to NFV has its own challenges as now general purpose hardware is used for packet filtering. In this work we show how the performance demands of security network functions can be met nevertheless.

C4 Hardware Offloading The proposed architecture increases the load on the security VNFs as packet filtering is not only provided between different networks but also in the network. Additionally the load in the network is increased as the filtering is not performed by network nodes but by VNFs in the NFVI. NFV features flexibility using hardware virtualization and abstraction. Naturally these concepts counteract hardware usage, as hardware is abstracted away by the virtualization. To overcome these issues we propose new implementation options for hardware offloading in SDN and NFV. First we propose to use offloading of traffic to SDN. Secondly we show how Network Interface Card (NIC) offloading can reduce the load on the VNFs.

1.3 Contributions

The contributions of this thesis can be divided into contributions regarding SDN, contributions with regard to NFV and contributions w.r.t. SDN and NFV offloading.

1.3.1 Software Defined Networking

Security Analysis and Enhancements in SDN In Chapter 3, we revisit the main attack vectors and how they affect a secure operation of an SDN network. We measure and evaluate the overheads in terms of latency that are introduced by encryption of the control plane. Further, we propose a Denial of Service (DoS) attack detection method and evaluate its effectiveness.

Evaluation of the Isolation of SDNs We show how SDN can be used to provide fine grained security in enterprise and campus networks in Chapter 4. As access isolation is not sufficient for providing full isolation between virtual networks, we also shed light on performance isolation in SDN networks. More specifically we compare and evaluate different Quality of Service (QoS) mechanisms on a number of hardware and software switches.

1.3.2 Network Function Virtualization

Performance aspects of the CPU architecture In order to achieve high throughput performance with security VNFs, fast packet processing frameworks can be used. It is known that packet processing performance is very sensitive regarding copying of packets. As modern servers are often built up of multiple CPUs with segregated memory we evaluate the performance penalties resulting from this segregation in conjunction with packet processing frameworks. Additionally we evaluate the effects of cache misses on packet processing in detail in Chapter 5.

Design of a Last Level Cache Scheduler for NFV In order to increase the resource utilization, multiple VNFs are co-located on one single server. Current virtualization techniques do not fully isolate all resources, thus co-location of VNFs causes interference effects. Interference effects can degrade the performance of VNFs in terms of throughput and delay severely. We aim to gather the potential that lies in reduction of the interference due to the shared Last-Level-Cache (LLC) by introducing a novel LLC scheduler in Chapter 5.

1.3.3 SDN and NFV Offloading

Evaluation of SDN Offloading algorithms A solution that combines SDN and NFV is provided in Chapter 6. In this concept, the SDN hardware's properties (line rate throughput but limited programmability) are combined with NFV properties (full programmability but

high resource consumption). Flows can be directed to network functions realized on NFV servers or offloaded via SDN to SDN-based network elements offering line rate hardware forwarding. Specifically we are addressing one challenge of such a combination: identifying those flows that benefit most of the hardware acceleration. We are introducing two approaches: A machine learning approach takes its decision with the first packet of a flow. A fundamentally different approach is using packet sampling for the offloading decision. We are evaluating both approaches in terms of precision, complexity and regarding the metrics of the combined NFV/SDN system.

NIC Offloading Modern packet processing frameworks that are used for NFV, such as the Data Plane Development Kit (DPDK), deliver high performance compared to older approaches. On the other hand, common NICs can provide additional matching capabilities that can be utilized for increasing the performance even further and in turn reduce the necessary server resources. Therefore, we propose the hybrid hardware software approach utilizing the NIC offloading hardware matching capabilities in Chapter 7. The results of our performance evaluations show that the throughput using NIC offloading can be increased by up to 50%, compared to a high performance software-only implementation.

1.3.4 Thesis Outline

The remaining chapters are outlined in the following:

Chapter 2: Background and Related Work In this chapter we first introduce the most important networking concepts SDN and NFV. Further we show related security architectures in literature and commercial solutions that approach the problem of providing fine grained isolation and filtering.

Chapter 3: Securing SDNs In the proposed architecture we are using SDN for providing fine grained virtual networks. Like this the security of SDN is an important condition for the security of the architecture. We analyze main attack vectors of SDN networks and show and analyze selected improvements to the overall SDN security.

Chapter 4: Isolation of SDNs Increased security using virtual networks is based on isolation between the networks. We study the isolation of virtual networks in SDN networks in terms of access and performance isolation.

Chapter 5: Performance of Security VNFs In this chapter we study the performance of security VNFs. We focus on low level effects caused by the memory architecture of modern CPUs. This is especially important as VNFs have a low compute complexity but high demands on I/O performance and the memory access.

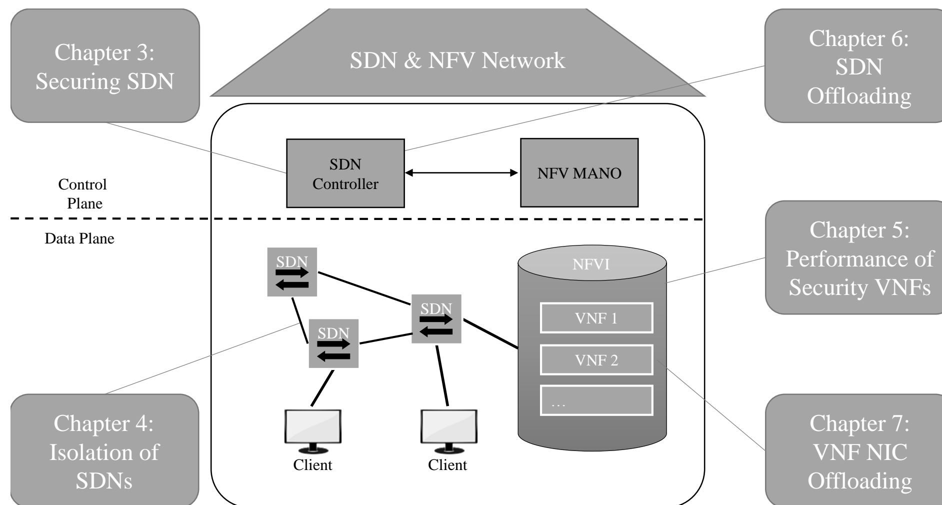


Figure 1.3: Overview of the topics covered in this thesis in relation to different modules of the security architecture

Chapter 6: SDN Offloading This chapter introduces an SDN Offloading approach that aims in reducing the load on the VNFs. As the capacity of SDN hardware tables is not sufficient to offload every connections, we analyze how different offload algorithms affect the offloading performance.

Chapter 7: VNF NIC Offloading An orthogonal approach aiming in an increase of the VNF performance is offloading functions to Smart-NICs. We introduce an approach that can reduce the CPU load significantly using NICs already available in deployed hardware today.

Chapter 8: Conclusion and Future work Finally a conclusion of the thesis that summarizes the findings is provided.

Chapter 2

Background and Related Work

In this thesis a novel security architecture for enterprise and campus networks is proposed. This chapter recapitulates the most important technologies that are used. Further it describes how the requirements described in the previous chapter were approached in the literature. Finally relevant commercial solutions that are sold by network equipment vendors are described.

2.1 Software Defined Networking

Network devices such as switches, routers or middle-boxes contain a control plane and a data plane. The data plane is responsible for processing the packets, while the control plane is responsible for providing more complex, higher level functions. We want to precise this definition by using the example of a IP router. In a router the data plane is responsible for forwarding the packets according to the routing table. The control plane on the other hand is responsible for configuring the routing table. This split is useful as the control plane changes are relatively few compared to the number of packets processed by a network device. Further the control functionality logic is more complex than the data plane logic. In a traditional IP router the control plane provides the routing protocol e.g., OSPF or IS-IS. The data plane however, only implements the relatively simple longest prefix match lookup. Traditionally both planes are co-located in the same system. With the Software-Defined Networking (SDN) architecture the control plane is logically centralized. This facilitates the implementation of new approaches as the network state is more easily accessible. Further the control plane is realized in software referred to as controller. The implementation of new features in one central software instance is also easier than the implementation in a distributed architecture where all devices need to support new features.

Figure 2.1 shows the SDN architecture [11]. The data plane consists of the SDN switches that process and forward the packets according to their forwarding tables. The forwarding

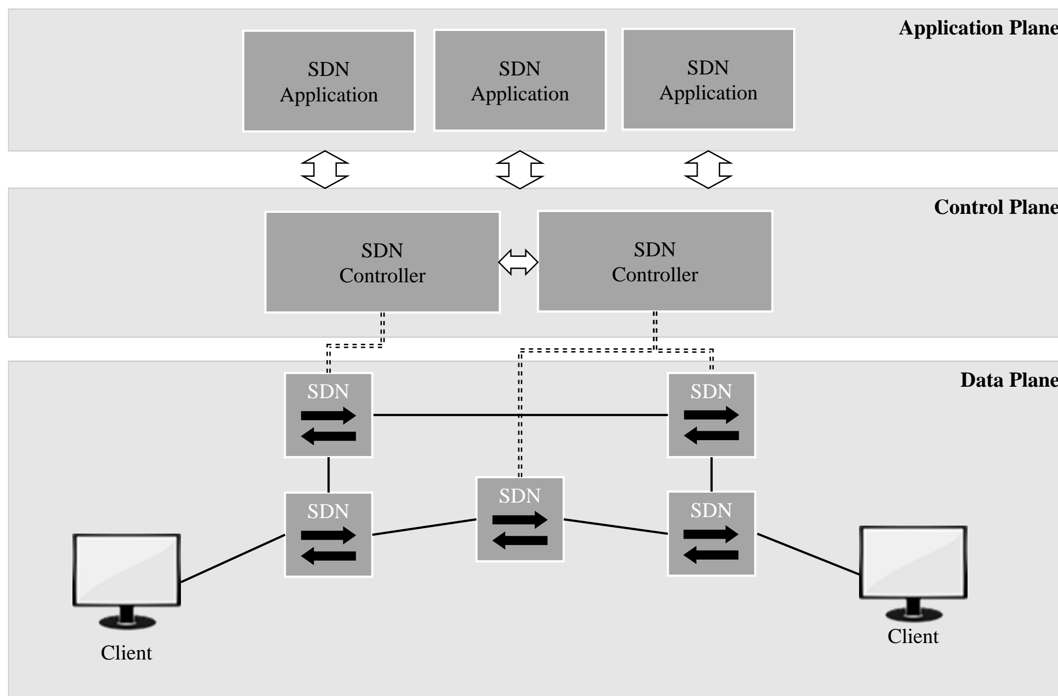


Figure 2.1: Software defined networking architecture

tables are filled by the SDN controllers. Centralization of the control plane does not mean that a single controller is responsible for the complete network, instead multiple controllers share the state between each other. Nevertheless for the network algorithms the distribution is abstracted away using specific approaches [15]. Applications that are specific for a certain network are not implemented in the SDN controller but in the application plane. They access the centralized control plane using the northbound API. The protocol between control and data plane is called southbound API. The best known southbound protocol is OpenFlow [13]. A similar protocol is the ForCES protocol [16]. With OpenFlow packets are processed using a match action semantic for the entries of the forwarding table. The entries are referred to as flows. The match of a flow defines which packets are processed by the entry. It is defined as a set of header fields, e.g. a certain source IP address. Fields from the set can also be partially ignored, e.g. to match an IP subnet. The action defines the steps that should be applied to the packet. This can be e.g. forward the packet to a certain port or change a header field. A full description of the possible matches and actions can be found in the OpenFlow specification [17]. OpenFlow aims at overcoming incompatibilities that arise with networks that use different vendors. Devices from multiple vendors support OpenFlow now, however many only support the mandatory parts of the standard.

The SDN controller provides a centralized point to command the network. This can be used for novel innovations in the field of network security. For instance SDN improves the visibility and control in the network such that filtering within the network is possible.

2.2 Network Function Virtualization

Network functions, such as routers, firewalls or load balancers, are commonly realized using integrated solutions which contain the hardware as well as the software in one box. As a result networks contain a large number of different systems from multiple vendors. With increasing complexity of the networks, these systems are getting more difficult to manage. Furthermore any change or addition of a network service requires the addition of new systems that have to be placed, powered and maintained. Thus changes in the networks are getting increasingly difficult.

As a relief ETSI proposed Network Function Virtualization (NFV) in a white paper [14] in 2012. NFV is designed to enable network functions using commodity servers and IT virtualization techniques. Different services can be run using a unified NFV environment. NFV is complementary to SDN as SDN is focused, even though not limited, to forwarding and routing, while NFV aims in virtualizing more complex network functions such as Deep Packet Inspection (DPI), Network Address Translation (NAT) or Virtual Private Network (VPN) gateways. Both concept can be combined or used exclusively. NFV promises an increase in flexibility as services can be added without change of the infrastructure. Further the services can be scaled up and down as required by the network conditions. Additionally, standard components, such as commercial off-the-shelf servers, promise a reduction of investment costs and energy consumption if compared to traditional hardware appliances.

2.3 SDN and NFV

SDN and NFV do not rely on each other but can be used in combination beneficially. For instance NFV favors rapid scaling and deployment of Virtualized Network Functions (VNFs) to provide middle box functionalities such as firewall, NAT or VPN gateways. Several VNFs are connected to realize more complex use cases in service function chains. Due to this approach routing must be more adaptable and flexible. SDN is one possible approach to meet this requirement, it can provide flexible routing mechanisms to steer the traffic as necessary.

2.4 Related security architectures in literature

The architecture proposed in this thesis promises fine-grained isolation and protection of networks using SDN and NFV. In the following related work that takes steps towards the same goal as in this thesis is presented.

End-host based approaches such as Shield [18] or distributed firewalls [19], filter the network traffic in the network stack of the end-host. As these approaches require the modification of the end-host they are difficult to manage and are not a reasonable measure to integrate fine-grained isolation into the network security concepts as no software can be installed on some devices such as printers. Furthermore if an adversary can gain full access to the end-host he is able to circumvent all measures taken directly on the end host. Thus end-host based solutions can not fulfill Requirement R1.

Ethane [12] opened the path towards SDN and OpenFlow. A centralized controller is checking the first packet of a flow and setups the path in the network if the flow is allowed by network policy. The novelty of ethane's approach is to use the full header for the identification of the virtual network. As such it fulfills requirement R1 in a similar manner as we propose. However, it does not consider performance isolation in detail, as we do in chapter 4. Furthermore, it is lacking the integration of stateful and application layer filtering as asked by requirement R2.

Resonance [20] is a network architecture for access control, that is building upon OpenFlow. Hosts are authenticated using a web-portal using a stateful registration procedure. They can provide fine grained isolation using OpenFlow, however in contrast to our work they do not isolate on connection level but on host level using MAC addresses. The work also concentrates on the access control and isolation and does not consider firewalling as required by R2.

ROFL [21] uses routing protocols to provide a fine grained isolation of flows. The authors propose to extend routing tables by adding ports to the routing entries. In order to block certain entries they propose to use special blocking entries in the routing table. Although they can provide flow level isolation in routed networks with this approach, it is difficult to integrate different transport protocols into the approach.

A combination of SDN and NFV is provided by VNGuard [22]. Similar to the approach that is discussed here, they are also using SDN to provide network isolation. Their focus lies on the virtual firewalls that are implemented using NFV. In this manner they are fulfilling the requirements R1 and R2. Many of the contributions described in this thesis can also be applied to the VNGuard architecture. For example network function offloading could enhance the performance of the virtual firewalls.

Even though all requirements were approached in related work, this thesis enhances the related work by improving upon the state of the art solution as detailed in Section 1.3. Detailed related work is also listed in each chapter respectively.

2.5 Commercial solutions

As described in Chapter 1, organizations and companies are facing security problems due to a diverse IT landscape. In order to solve this issues a number of commercial solutions came up.

Identity Service Engine (ISE) [23] is Cisco's solution to support fine-grained network access security. It supports fine grained access control for the devices in the network, different access profiles for the devices can be defined in a centralized manner. The devices can be authenticated using the RADIUS protocol. Isolation between the different networks is achieved using an additional header to identify the network membership. As such it support the requirements R1 and R2, moreover as a commercial solution it also provides more mature methods for the management of the access policies. On the other hand the header limits the number of possible virtual networks to 64K, while our approach directly uses the network header and as such the number of virtual networks is practically not limited. Furthermore the header used for network virtualization is proprietary and as such the solution is not a good fit for multi-vendor networks.

Aruba ClearPass [24] is the respective solution from HP Enterprise. Its focus is on the management of network access. Unknown devices can register via a captive portal and can be authenticated by a built-in certificate authority. Similarly, the security company Forescout [25] offers a solution that can be integrated with not clearly defined devices from other vendors. For both solutions the isolation is achieved with legacy VLAN using the 802.1Q standard [26]. Further they support the integration with 802.1X [27] supporting devices, which supports the authentication at the edge of the network. Through the use of 802.1Q and, together with 802.1X a complete isolation from the first hop is possible. On the other hand the number of possible VLANs is limited to 4096 due to the length of the header. This means Requirement R1 is fulfilled to some extent. Furthermore VLANs are often used for different purposes, like e.g., prioritization of voice packets. Thus the use of VLANs requires manual coordination if different systems are involved. R2 can be supported by using conventional firewalls between the virtual networks with limited flexibility as the solution does not provide any means for setting up the virtual networks in the physical network.

None of the above solutions can provide an isolation on the flow level as it can be offered through SDN. Furthermore the number of virtual networks is limited, as all commercial solutions add headers for the identification of the virtual network membership.

2.6 Summary

In this chapter firstly the main technology building blocks SDN and NFV were introduced. They provide new possibilities for realizing network security, by enhancing programmability and flexibility of the network. The available commercial solution can also provide some level of isolation. This shows that there is a need for isolation in enterprise networks. Though, current solutions are limited in flexibility. E.g., most solutions can only provide a quite small number of virtual networks due to technological restrictions.

The analysis of the related work shows that some approaches with similar goals exist in literature. Though several challenges in the field of performance of the solution and the security of SDN remain open. In the following chapters it is shown how the research challenges were approached individually.

Chapter 3

Secure Operation of Software Defined Networks

The upraise of Software-Defined Networking (SDN) is an opportunity to enhance network security. The fine-grained security architecture presented in this work relies on the isolation made available through SDN. On the other hand novel attack vectors are introduced via SDN and some attack vectors that also exist in legacy networks continue to exist. Although network security has received increased attention in the last years, security is still largely ignored in many novel network concepts and not considered from the beginning. This also holds for the SDN concept.

Our goal is to design an security architecture for fine grained access control. SDN is used as main virtualization technology in the network and can be used to provide isolation between the virtual networks. We present this in Chapter 4. Consequently SDN is a crucial technology in the overall architecture. Thus, in this chapter we are studying security concerns. Attack vectors and countermeasures in SDN networks necessary to operate the network securely are described.

The contributions presented in this chapter are as follows:

First we introduce main attack vectors in an SDN network. SDN differs to legacy approaches by centralizing the control plane. On the one hand the centralization causes some novel attack vectors, on the other hand these attack vectors can also be mitigated easier using the increased flexibility of such a software based solution. Further some vectors are inherited from legacy networks as many protocols and approaches are still used as well in SDN.

As the control plane is no longer co-located with the data plane devices, the centralized controllers must be connected to the SDN devices using a control plane data plane connection. This opens an crucial attack vector by making Man in the Middle (MitM) attacks to the control plane data plane connection possible. Fortunately this can be mitigated using Transpor Layer

Security (TLS). However the cost of encryption was not clear before, therefore we explore the cost of TLS encryption in terms of delay.

Further we introduce a detection and mitigation algorithm for Denial of Service (DoS) in SDN. This attack can cause failures of the switches, the controller or the control plane connectivity. We propose a tailored statistical detection approach as well as a lightweight countermeasure. We evaluate the detection by simulation and an analytical approach. Throughout this evaluation, we highlight the trade-off between detection speed and adaptability and show a way to tune the solution analytically.

Section 3.1 introduces the main attack vectors in an SDN network. Section 3.2 gives an overview on related work on attack vectors and countermeasures in SDN networks. Further Section 3.3 and Section 3.4 introduce and analyze important countermeasures that are necessary for a secure operation of SDN networks. Finally Section 3.5 concludes the chapter and summarizes the main findings.

This chapter is partially based on measurements results regarding the encryption of the SDN control plane presented in [1]. Further the approach for a DoS countermeasure was presented in [2]. The summary of attack vectors in SDN networks was not published before, except Section 3.1.5 which was presented as part of [2].

[1] R. Durner and W. Kellerer. “The cost of security in the SDN control plane.” In: *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT) - Student Workshop*. 2015.

[2] R. Durner, C. Lorenz, M. Wiedemann, and W. Kellerer. “Detecting and mitigating denial of service attacks against the data plane in software defined networks.” In: *IEEE Conference on Network Softwarization (NetSoft)*. 2017.

3.1 Attack vectors in Software Defined Networks

First, in this section we categorize the different attack vectors in an SDN network. We only show attack vectors that are specific for SDN networks even though some vectors exist in a similar fashion in legacy networks. Figure 3.1 describes the main attack vectors in an SDN network.

3.1.1 Rogue SDN Application

In addition to introducing a split between control and data plane, SDN also features the introduction of SDN applications. These apps can extend the functionality of the network

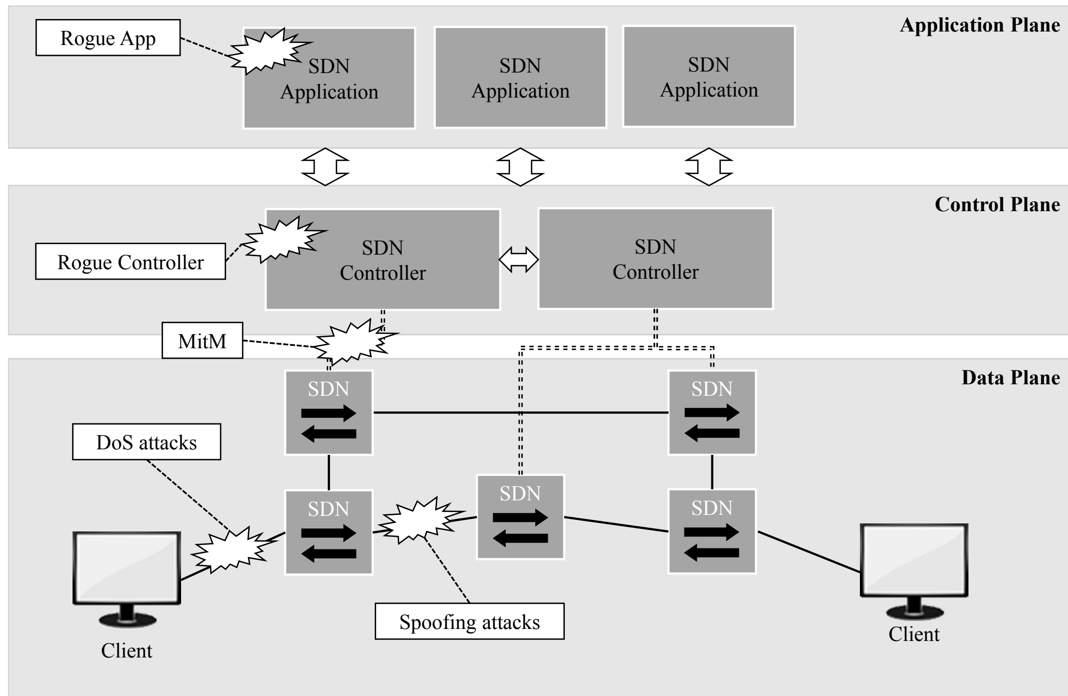


Figure 3.1: Attack vectors in SDNs. Attack vectors can be grouped into attacks from the data, control and application plane.

in order to support very specific use-cases. According to the Open Networking Foundation SDN applications are connected to the controller via a northbound API. In order to support a wide range of use-cases northbound APIs of common controllers like, Ryu, OpenDaylight or ONOS provide rich functionality to the applications, e.g. the APIs allow to write OpenFlow rules. In general this can allow a rogue SDN application to circumvent the security policies of the network. Additionally legitimate applications could also harm the network by adding rules that are in conflict with the security policy or conflict with rules from other apps.

3.1.2 Man in the Middle Attack to the Control Connection

With SDN the control plane is logically centralized in an SDN controller and the forwarding devices, the SDN switches, are controlled remotely using a protocol such as OpenFlow. If an adversary succeeds to break into the control connection using a MitM attack, the adversary can control the switch(es) in the same manner as the controller. To make things worse the adversary can even modify the messages send to the controller and mask his attack using forged messages.

3.1.3 Rogue SDN Controller

The SDN controller is connected with the SDN switches over the network. We refer to the network that carries this control traffic as control network. The control can either be operated in-band or out-of-band. Using out-of-band control, the control network is either a separate physical or virtual network. The deployment and operation of this network causes an overhead to the network operator. With an in-band control this overhead can be avoided as the control network is the same network as the data plane network.

If the adversary can gain access to the control network in either cases, it can launch a rogue controller. Without other precautions the rogue controller can connect to the switch as primary or secondary controller. As a result the adversary can reroute flows or undermine the security policies of the network by allowing connections between hosts that should be isolated.

Many popular controllers, such as ONOS [28], are realized distributed to enhance reliability and scalability. The control plane consists of multiple distributed controllers that share the state in different manners. The different possible realizations [15] have in common that network state is shared between the instances. If an adversary succeeds in connecting a rogue controller to the legit instances, he can inject manipulated network state and undermine network security in this way.

3.1.4 Spoofing Attack

Spoofing attacks are an unresolved issue in legacy networks, the adversary impersonates a communication partner, e.g. by spoofing its IP or MAC address. Usually such an attack is used to become MitM, which can then be used for other attacks such as eavesdropping. Spoofing attacks are possible due to a lack of authentication. In general every communication is vulnerable to spoofing if its communication partners are not authenticated using cryptographic methods.

Also functionalities of SDN networks are subject to this vulnerability. Specifically the topology discovery process is not authenticated per se. Clients are discovered using their communication which is based on the Ethernet and the Internet Protocol (IP) protocol. An adversary can easily spoof the addresses of these protocols and consequently manipulate network state also in an SDN. Furthermore the links between the nodes in the network are discovered using Link Layer Discovery Protocol (LLDP). Each switch is commanded by the controller to send an LLDP packet with an identifier that is specific for each switch. Other switches in the network receive the packets and forward it to the controller. By comparing the send and the receive location, the controller can then detect links in the network. On the

other hand an adversary can easily produce fake links by injecting LLDP packets according to his will. This can then be used for launching a MitM attack to other hosts in the network.

3.1.5 Denial of Service Attacks

If an adversary has already access to an SDN device, he can launch well known DoS attacks, such as SYN flooding, against the controller. On the other hand SDN introduces an attack vector that is not existing in legacy networks: DoS attacks against the data plane. DoS attacks against the data plane use the reactive mode for attacking. This attack class was first described in 2013 by [29] and [30].

SDN switches process traffic according to the entries in their forwarding tables which are set by a logically centralized controller. SDN offers two major modes of operation – *proactive* and *reactive*. In the former case the controller presets all forwarding rules according to the configuration of the networking applications which provide the networking functionality, e.g. switching or routing. Packets that do not match any entry in the forwarding table are dropped by the networking element.

In reactive setups, on the other hand, a table miss results in a query to the controller. In the controller, the networking applications can make a decision based on a global view of the network's state. Then, they are able to enforce a network policy, e.g. routing, by individually forwarding packets, sending out packets, or setting up forwarding rules. The most prominent SDN protocol allowing both modes of operation is OpenFlow (see [13]) which also enables any hybrid approach with proactive and reactive elements.

Figure 3.2 shows the typical behavior of a reactive SDN setup:

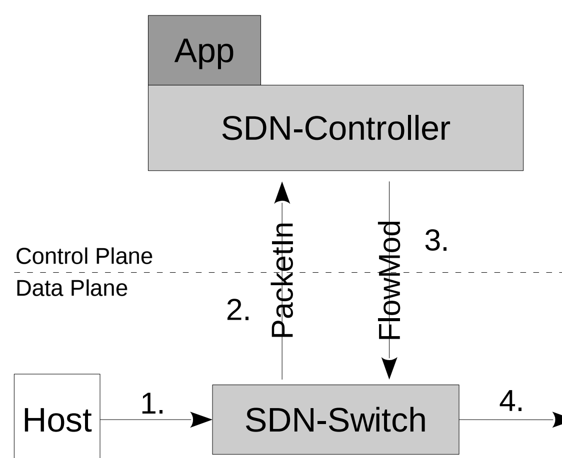


Figure 3.2: Normal behavior of a reactive SDN.

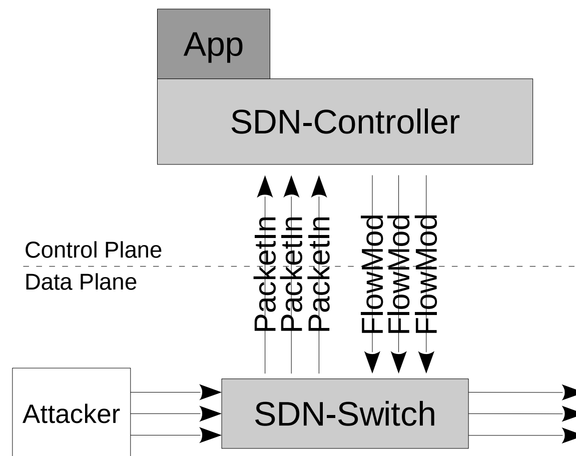


Figure 3.3: DoS attack against a reactive SDN.

1. A host sends a packet for a new connection which reaches an SDN switch. Then, the switch performs a lookup in its forwarding table.

2. Since the packet belongs to a new flow which is unknown to the switch, the packet is encapsulated into a *PacketIn* message and sent to the controller.

3. In the controller the PacketIn is processed by an SDN application which provides networking functionality like switching or routing. The applications decision may include sending a *FlowMod* message which installs a rule in the forwarding table. Also, the switch may be instructed to forward the original packet through any of its ports.

4. If the controller application has set up a rule in the switch to handle the flow, further packets belonging to that flow will be processed in the fast forwarding hardware without the need for additional communication with the controller.

As previously seen in Figure 3.2, upon the incoming of an unknown flow the switch typically consults the controller for further decisions by encapsulating the first packet of the flow into a PacketIn message. Then, the controller can inspect the packet, make a forwarding decision, and set new flow rules in the switch using FlowMod messages if necessary.

There are different ways for a controller to handle this decision making, especially regarding the granularity of flow definitions. For instance, flows could be setup using the quintuple Source IP, Destination IP, Source Port, Destination Port, Protocol or just Layer-2 addresses. An adversary knowing about the controller's decision making and flow rule setting behavior is able to craft packets that trigger those modifications of the forwarding table. For the switch these packets appear as unseen flows and therefore, are handled by the controller resulting in an increasing number of forwarding rules in the switching tables as depicted in Figure 3.3.

The adversary continues triggering additions to the flow table causing the table to fill up. The specific impact of a switch with full switching tables is not generally defined and highly depends on the model. Typical behavior includes the dropping of older switching entries or ignoring new rule setting requests. Especially hardware devices rely on a limited forwarding table in order to be able to process packets at line rate. On the other hand software tables have a nearly unlimited capacity, though a large table can also cause performance degradation in this case [31].

3.2 Related Work

Security in SDN was studied widely in the past years. This chapter of the thesis describes the main attack vectors and countermeasures necessary in order to operate SDN networks securely. Thus we also concentrate on this subset of SDN security research. The presented related work is structured in the following fields: Security Analyses and Attacks to SDN networks and its countermeasures.

3.2.1 Security Analyses

Systematic security analyses aim in showing weaknesses of SDN and OpenFlow in a general way. On the other hand they do not propose improvements or countermeasures in detail.

One of the first comprehensive security analyses of SDN and OpenFlow specifically was published by Klöti et. al. [32]. The authors analyze security using STRIDE Method [33]. The paper shows that SDN is vulnerable towards DoS attacks against the flow table of the switches. Furthermore, it also analyses information disclosure due to flow aggregation by the SDN controller.

Another analyses studies vulnerabilities of OpenFlow [34] and especially highlights the importance of transport encryption for the connection between the SDN switches and the controller. Specifically the paper highlights the need for authentication of both switches and controller in the SDN network.

Further Schehlmann et. al. [35] weight the security risks of SDN networks against the potential improvements to security. The study compares SDN networks to conventional networks. It is concluded that the increased risks that are introduced by SDN can be solved using existing security concepts and approaches such as authentication. On the other hand authors claim that SDN can improve security in the network substantially.

3.2.2 Attacks and Countermeasures

There are quite a number of works that study specific attacks to SDN networks and their countermeasures.

An attack that can compromise cloud systems from the inside of the Virtual Machine (VM) is severe as the adversary can gain access to many other systems running in the same cloud. Such an attack is possible by exploiting weaknesses in software switches [36]. The attack exploits weaknesses in the packet parsing functionality of the software switch to compromise the hypervisor. This can be used in a second step to compromise the controller and finally also other VMs and services running on the same cloud environment.

A first feasibility study for DoS attacks on an SDN network was done in [37]. Authors show that the granularity of rule aggregation can be detected using time differences in the Round Trip Time (RTT) of connections. Further a successful DoS attack in a test bed shows the feasibility of the attack and its required attack time.

In [38] an approach for the detection of distributed DoS attacks against the controller is presented that relies on detecting deviations from a normal distribution of PacketIns in terms of destination addresses. An attack is indicated by a significant growth of new flows to a single host compared to the normal situation where new flows reach hosts evenly distributed. Therefore, an attack is indicated by a lower entropy calculated over a window of PacketIns. The authors used windows of size 50 and viewed five consecutive entropy values below a threshold to be an attack resulting in a sample of only 250 PacketIns. The emulation results look promising offering a detection rate between 95 to 100%, although most parameters like arrival distribution and network settings remain unclear. Nevertheless, they evaluated their approach using the destination address as fixed and the source address as varied parameter. The more parameters an adversary can shuffle the higher the entropy of the attack packets will be. As is, an adversary who can address the whole subnet under supervision is likely able to circumvent the detection completely. Therefore, it remains unclear whether this approach can be scaled to scenarios with manifold variable header fields, e.g. if the adversary controls a virtual machine in a cloud data center. Further, the approach does not yield information to quickly apply countermeasures against the attack. Additionally, the authors did not consider scenarios where new services are started on a host and publicly announced resulting in a lower entropy for a short time since numerous external clients start to use this service.

A very notable approach called FlowRanger is provided by [39]. The basic idea is to classify PacketIns using a trust-level metric and enqueueing them with different priorities. This leads to a faster processing of PacketIns triggered by trusted hosts and a higher probability of untrusted PacketIns being dropped in high load situations. A host's trust level may be adapted over time due to its behavior. Although, invented for mitigating DoS attacks against

the controller, this approach also helps to reduce the impact of attacks against the data plane. Since fewer malicious PacketIns are processed by the controller, this also reduces the rate of FlowMods. Nevertheless, FlowRanger reduces the attack's impact without removing its root cause – the initial triggering of PacketIns by an adversary. Also, with sufficient resources granted to the controller the approach becomes less effective since also the low priority queues are processed fast. However FlowRanger might be a suitable supplement to the work presented in this Section 3.4 since it helps to reduce an attack's impact before its detection due to its different focus.

In [40] a technique is proposed to minimize the impact of a DoS against the controller and the switch tables by optimizing rule expiration and rule aggregation in the switches. These measures lower the impact of attacks flooding the flow tables by reducing the overall resource usage without tackling the attacks' root cause. Additionally, the reduction of the expiration time could increase the load of the controller and may add delays to flows which timeout prematurely. Nevertheless, this approach is complementary to our efforts and could help in building a robust and efficient system.

Further, [41, 42] propose FloodGuard, an approach that tries to anticipate the behavior of the controller as well as the applications and set up rules in the switches proactively. These rules try to reduce the amount of PacketIn events and therefore restrict the abilities of an adversary to be successful. Occurring PacketIns are cached and served using rate limiting to further reduce the impact of an attack. As a side effect this approach causes unfavorable delays due to the caching of packets.

In [43] a mechanism is proposed to safely remove entries from full flow tables. The ratio of PacketIns and FlowMods is supervised and if the table is going to be full, rules are removed using a least-frequently-used scheme. As a disadvantage the approach causes potentially high load on the switches due to aggressive usage of OpenFlow's statistical features.

A different kind of attack is a MitM to the controller switch connection [44]. The work describes an attack to the OpenDaylight controller using ARPspoofing in detail. Authors show that also large SDN projects like the OpenDaylight do not consider security as a primary goal.

SDN controllers are extendable by SDN applications that are connected to the controller using the northbound API. FortNox [45] and SE-Floodlight are security kernels that are implemented in the controller. Their main goal is to restrict access to the northbound API in order to prevent applications from circumventing security requirements. Besides authenticating applications they also aim in avoiding rule conflicts that can arise if multiple applications are used.

In most cases multiple SDN controllers are used for reliability reasons. Byzantine fault tolerance is an approach that can improve security of a control plane consisting of multiple controllers as well. Sakic et.al [46] show that 2m + n + 1 controllers are sufficient to tolerate m malicious and n faulty controllers.

Finally many SDN controllers also lack protection against replay attacks in the data plane. This can be exploited to compromise network visibility [47]. The authors show how forged LLDP packets can be used to create fake links. Further it is shown that host locations can be faked using MAC Spoofing. As a countermeasure the authors propose authentication for LLDP packets and hosts.

3.2.3 Summary

From the given security analysis and the related work we can see that the main attack vectors introduced with the SDN paradigm are twofold. Firstly the split of data plane and control plane introduces a new powerful MitM attack. Secondly with SDN the network is managed more autonomously following higher level policies, while legacy networks are still often managed with lots of manual interaction. This is especially true if reactive SDN paradigm is considered. On the other hand the increased automation enables DoS attacks. Thus in the following we are focusing on these two attacks. In Section 3.3 we shed light on the encryption of the control plane that can prevent MitM attacks. In Section 3.4 we study the detection of DoS attacks in SDN networks.

3.3 Securing the Control Plane of Software Defined Networks

In order to operate an SDN securely it is essential that the control plane is not corrupted either by a rogue controller or a MitM attack. Therefore it is necessary to authenticate the controller to the switch and vice versa and furthermore to encrypt the communication between controller and switch. Both can be provided by the TLS protocol. In fact for OpenFlow connections TLS encryption is recommended by the specification.

Thus we analyze the TLS support in the OpenFlow eco-system. In particular, we implemented a performance measurement tool for encrypted OpenFlow connections, as there is non available. The results show that security comes at an extra cost and hence further work is needed to design efficient mechanisms taking the security-delay trade-off into account.

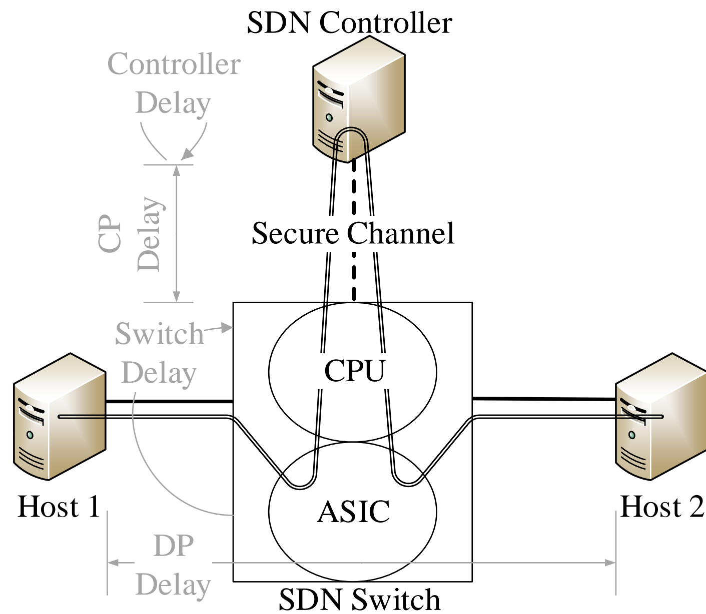


Figure 3.4: Measurement setup: The SDN Controller is used as a relay, thus no rules are installed in the hardware table.Thus all packets have to traverse the shown path.

3.3.1 Measurement Setup

We investigated the packet-in delay that occurs for the first packet of a flow in reactively managed SDN networks. In this case the first packet of each connection is forwarded to the controller. The controller setups the path in the network upon this packet, i.e. installs the corresponding rules in the SDN switches, and outputs the packet using packet out. As a result the delay to and from the controller directly affects the setup delay of new connections.

We developed a measurement setup specifically to measure this delay. The measurement setup is shown in Figure 3.4. Packets are send from Host 1 to Host 2 via an SDN switch that is controlled by an SDN controller running on a VM. In general we always measured round-trip times, directly at Host 1.

We are verifying the effects of encryption to the OpenFlow performance with an experiment using different hardware and software SDN switches: An NEC PF5240, a Pica 8 P3290, a Pica 8 P3297 and the software switch Open vSwitch.

Beforehand, we measured the delay of packets with matching flows (DP Delay) and the delay from switch to controller (CP Delay) separately. The CP Delay was directly measured at the controller machine NIC. The DP Delay was measured accordingly using round trip times.

In the measurements our controller acts as a relay. On a packet-in the controller replies with an appropriate packet-out message but no forwarding rule is inserted.

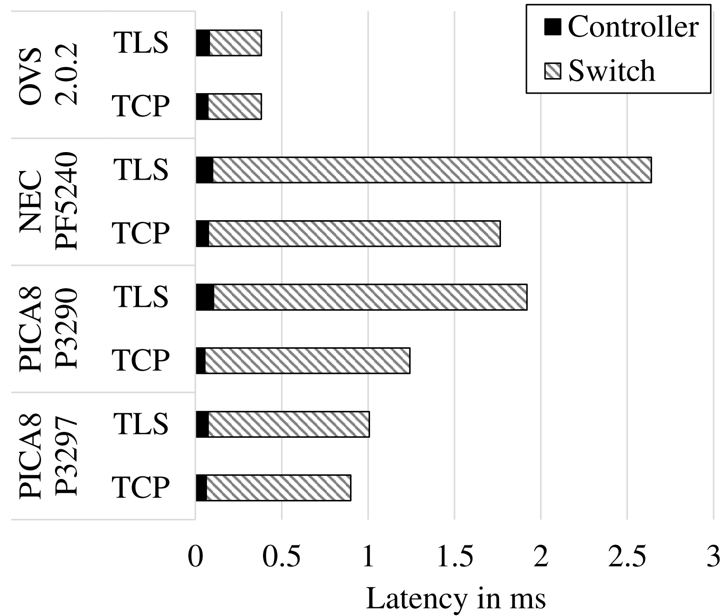


Figure 3.5: Latency added to the first packet by switches and controller using TCP and TLS

At Host1 we measured the round trip-time of packets using this setup. The additional switch processing delay for the first packet of a flow is then determined out of the DP, CP and controller delays subtracted from the end to end delay. In general the total delay is dominated by the switch delay. More detailed results for the different devices are shown in the next section.

3.3.2 Measurement Results and discussion

We did independent measurements for TLS and TCP for the different switches, the results are shown in Figure 3.5. We conducted 1000 measurements for each result. This leads to confidence intervals <0.05 ms of all measured latencies that are omitted in the figure. As can be seen the switch adds by far the dominant part to the complete latency. The delay of the controller in comparison is small. Both PICA8 Switches run PICOS, however the P3297 has a more powerful CPU than the P3290, therefore latencies are smaller in general. Specifically the TLS overhead for the P3297 is much smaller with less than 1 ms latency as it has hardware acceleration for encryption built in the CPU. The results of Open vSwitch supports this observation, as also low latencies and low overhead were measured and the Intel CPU is more powerful than the ones of the switches. In contrast to that, the packet-in delays of the NEC and the P3290, without hardware acceleration, differ significantly if encryption is used or not.

The delays of the controller differs, as for some switches the payload of the respective packet is not sent to the controller along with the packet-in message. This can be seen at the

delay consumed at the controller side for the NEC and the P3290 are larger than the OVS and the P3297. This different behavior of course also influences the time consumed of the switches.

In general it can be seen that the delays are quite significant with at least 1 ms for the hardware switches, this is equivalent to a distance of 200 km in terms of propagation delay. Thus it can be seen that delay sensitive flows should be set-up in a pro-active manner, as re-active operation of SDN requires signaling to the controller which delays flow setup.

3.4 Securing Software Defined Networks against DoS attacks

The uprise of SDN as a paradigm that separates the data from the control plane introduces new challenges in network security. Especially, in modern cloud environments where an attacker can get access to the network by simply renting a virtual machine, DoS attacks pose a serious threat.

Forwarding tables of SDN switches have limited memory capacities. Typically, switches rely on Content Addressable Memory (CAM) that performs table lookups at line rate. Especially, Ternary Content Addressable Memory (TCAM) is expensive and therefore very small, ranging in the region of 1k to 2k entries. But also, regular and cheaper Binary Content Addressable Memory (BCAM) is limited to only a couple of 100k entries. An attacker with the ability to remotely trigger flow handling modifications that add entries to the flow tables can cause a DoS by exhausting the switches memory. Some works extend the available table size by using a combination of software and hardware flow tables [48, 49]. If the attacked devices make use of such techniques, the severity of the DoS is reduced, as software tables allow far more entries. Although, also software tables suffer from performance penalties for big table sizes, e.g. Open vSwitch uses a linear search to handle wildcard rules [31]. To deal with this threat we propose a tailored statistical approach for the detection of such an attack.

Therefore, our main contributions are:

- A problem specific detection mechanism for attacks on the data plane, that is more comprehensive than existing approaches.
- A lightweight counter measure to stop attacking flows with one or very few flow rules.
- A novel evaluation by analytic means and simulation.

a	**b**	**c**
0	0	0
0	0	1
0	0	0
0	0	0
1	0	0
0	0	0
0	1	0
0	0	0
0	0	0

a	**b**	**c**
0	0	0
0	0	1
0	0	0
0	0	1
2	0	0
0	0	0
0	2	0
0	0	0
0	0	0

. . .

a	**b**	**c**
0	0	1
0	0	1
0	0	1
0	0	1
10	0	1
0	0	1
0	**10**	1
0	0	1
0	0	1

Table 3.1: Simple example showing the growth of a counter table over time for ten consecutive packets with three header fields **a**, **b**, **c** where **c** is varied while **a** and **b** remain fix.

3.4.1 Detection and Mitigation of DoS attacks against SDNs

In this section, we first present a novel statistical detection approach specifically tailored to the problem presented in section 3.1.5. Secondly, we introduce a novel lightweight method to mitigate a detected attack with only small restrictions to the networks' functionality.

3.4.1.1 Detection

The general idea for the detection of DoS attacks against the data plane aims at localizing the fixed header fields of the attacking flow. These impose a regularity that is not observed in normal traffic since PacketIn events are just seen once upon flow establishment. The approach uses a table of counters with the different header fields as columns. The table is regularly, i.e. in fixed time intervals, inspected statistically and the maximum entry is abnormally large in case of an attack. To normalize the table size the header fields are hashed by a uniformly dispersing function with fixed output size. The digest of an input determines the row where to increment the counter. During an attack the entries which correspond to fixed fields of the attacking flow grow very fast and are used for the detection.

Table 3.1 shows a simplified example with three header fields a, b and c where the latter is varied. The columns represent the different header fields, while the rows are accessed using the hashed values of the particular header field. After a couple of PacketIns the fixed fields of the attacking flow are clearly distinguishable from the varied fields.

For further explanations we formalize the necessary terms as follows. $\mathcal{H}$ is the set of all header fields considered by the detection algorithm, e.g., $\mathcal{H}$ ={src_mac, dst_mac, src_ip, dst_ip, proto, src_port, dst_port, ...}. A packet p is characterized by a set of tuples (h, v) of header fields $h \in \mathcal{H}$ where h acts as a key and v as value, e.g., p = {(src_mac, 11:22:33:44:55:66), (dst_mac, 66:55:44:33:22:11), (src_ip, 1.2.3.4), ...}. The

Algorithm 1: Book-keeping of seen PacketIn messages.

```
  Input: Counter table T
1 while true do
2     receive PacketIn and unwrap packet p;
3     forall h ∈ {h|h ∈ 𝓗 ∧ (h, v) ∈ p} do
4         S ← hash(v_{h,p});
5         (c, v_{h,p−1}) ← T_{(S,h)};
6         T_{(S,h)} ← (c + 1, v_{h,p}));
7     end
8 end
```

concrete value of a field h in the packet p is denoted by $v_{h,p}$. Using these terms, a hash function is defined as

$$hash : v_{h,p} \rightarrow \mathbb{N}_0^{<|hash|}$$

where $|hash|$ is the size of the hash function's image set. The table T is a matrix of the dimension $\mathbb{N}^{|hash|,|\mathcal{H}|}$.

We used the 32Bit FNV-1a hashing function (see [50]) folded to an output size of 16Bit by applying an XOR operation of the upper half to the lower half of the hash sum. The chosen hash function is designed to be fast while having a low collision rate which is evenly distributed itself. This results in 2^{16} rows in the counter table.

Every incoming PacketIn is unwrapped and the included packet is handled by the method shown in Algorithm 1. For each field its value is hashed. The hash sum is now used as an index in the table where the counter is incremented. Additionally, the corresponding fields of the most recent packets are stored with the counter for later usage by the mitigation routine upon a detected attack. The employed FNV-1a hash function is designed to have a low collision rate. Thus we can assume that storing of one field is enough in practice.

Since the application of the hash function can be bounded to the largest field size and the field updates run in $O(1)$, the overall update time is in $O(|\mathcal{H}|)$. The table may require quite a large amount of memory. Depending on the implementation the memory consumption can be bound to

$$O(|hash| \cdot (|counter| + |field|) \cdot |\mathcal{H}|)$$

where $|counter|$ is the size of the counter (in Bytes) and $|headerfield|$ is the size of the structure holding the most recent header field value. If the table is statically allocated it would require exactly this amount of memory. Some implementations allow a dynamic allocation at run time which optimizes memory consumption at the cost of slower data access and allocation overhead. We consider a statically sized table to be the preferable approach since the usage

Algorithm 2: Attack detection and mitigation.

```
   Input: Counter table T, time interval t_W
 1 P ← {};
 2 for every t_W seconds do
 3 |   for h ∈ H do
 4 |   |   for S ∈ N_0^{<|hash|} do
 5 |   |   |   (c, v_{h,p}) ← T_{S,h};
 6 |   |   |   if c > θ_m then
 7 |   |   |   |   P ← P ∪ {(h, v_{h,p})};
 8 |   |   |   end
 9 |   |   end
10 |   end
11 |   if P ≠ ∅ then
12 |   |   Block all packets that match headers in P;
13 |   end
14 |   reset T;
15 end
```

of hash functions distributes the counter updates evenly. Therefore, there should not be too many untouched fields with counters equaling zero.

As seen in Algorithm 2, the detection routine runs independently of the book keeping on a regular basis and statistically evaluates the counter values. The table is evaluated every fixed time interval t_W. The counter is evaluated for every header h and every hash value S, i.e. for every field in the table. If the value of a counter is higher than a predefined threshold θ_m, an attack is indicated and the corresponding field $v_{h,p}$ is appended to list P.

The detection algorithm has a complexity of $O(|hash| \cdot |\mathcal{H}|)$. In conjunction with the data collection seen above, the overall detection is lightweight. Especially, since it can be executed concurrently and thus, no stalling of the packet pipeline is necessary.

3.4.1.2 Counter Measures

After detecting an attack the set P contains the fixed headers of the attacker. From this set it is easy to craft a flow rule that matches the fields from P while treating the others as wildcards. If one header or more headers is in the set with different values, multiple rules have to be installed. For this the existing rule with the singular headers from P has to be copied and for each value of one header a rule has to be created. This could for example be the case if the attacker leverages a bot net and as a result more than one IP-Source address has to be blocked. When installed as a low prioritized dropping rule in the switch it is now able to handle further attacking packets at line rate and without additional interaction with the controller. The impact

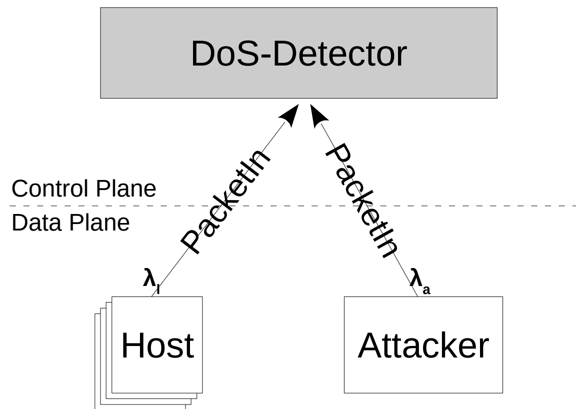


Figure 3.6: Abstracted simulation of the detection system.

on the network is negligible since only regular hosts falling in the detected flow characteristics are affected by the mitigation which is considered unlikely concerning our attack model.

3.4.2 Evaluation of the approach

This section describes the abstracted simulation method which was used to validate the detection algorithm. Besides the simulation results, an analytic evaluation of the *false positive* and *false negative* probabilities is provided which can be used to determine the correct parameters of the detection algorithm.

3.4.2.1 Abstracted Simulation

In order to evaluate the detection method we built a simulation based on the widely used `OMNeT++` framework [51]. One feature of our simulation is that we did not simulate on a data plane level, but only on a control plane level. The abstracted view is shown in Figure 3.6: In the simulated system a host causes a new PacketIn when a new connection is started, i.e. with the first packet. Afterwards, all packets which belong to the same flow would be handled in hardware (i.e. the data plane) in the real system and are not simulated. This greatly reduces the number of simulated packets, while retaining all important effects of the attack. In our simulation we used for the legitimate users negative exponentially distributed arrivals with an expected mean inter arrival time T_l, which correspond to an arrival rate $\lambda_l = \frac{1}{T_l}$. The attacker arrival rate is called λ_a. Other important parameters are the window size t_W, i.e. the time between two consecutive runs of the detection algorithm and the detection threshold θ_m. The parameters are also summarized in Table 3.2.

Normal Traffic Arrival Rate	λ_l
Attack Arrival Rate	λ_a
Window time	T_W
Number of Hosts	H
Max value threshold	θ_m

Table 3.2: Parameters of simulation and Analysis

3.4.2.2 Analytic Evaluation of the Detection Performance

The detection mechanism of our approach labels the state of the system as *under attack* if the table maximum $m_T = max(T_{S,h})\,\forall S, h$ exceeds some threshold θ_m. The threshold should be low enough to detect attacks, but it should not raise an alarm if no attack is attempted. As usual, we call these false alarms False Positive (FP) while undetected attacks are False Negative (FN). The threshold could be set empirically by just trying different thresholds and measuring the effects.

In this section, we try to give a more systematic approach for determining the threshold. The expected m_T corresponds with the maximum expected collisions of a header value. For example, if all headers of the incoming connections of one host are uniformly distributed, except for the source IP, this results in a high value in the corresponding table entry s, h and therefore this entry $T(s, h)$ dominates $m_s = max(T_s)$, the maximum of row s. For this system, we can compute the probability $P_{m_s}(n)$ of the maximum of row T_s with the help of the Erlang distribution: The Erlang Cumulative Distribution Function (CDF) $F_{n,\lambda}(x)$ describes the probability of n events occurring in a certain time interval x with $0 \leq X \leq x$, if the events are exponentially distributed in time with a rate λ.

$$F_{n,\lambda}(x) = 1 - e^{-\lambda \cdot x} \sum_{i=0}^{n-1} \frac{(\lambda \cdot x)^i}{i!}$$

For our case the interval is always $0 \leq X \leq t_W$. For a fixed event rate, the probability of more than n events is:

$$P_{m_s}(n) = 1 - F_{n,\lambda}(t_W) = e^{-\lambda \cdot t_W} \sum_{i=0}^{n-1} \frac{(\lambda \cdot t_W)^i}{i!}$$

The probability of n events is then:

$$p_{m_s}(n) = P_{m_s}(n+1) - P_{m_s}(n)$$

$p_{m_i}(n)$ corresponds with the probability of a value of n for row m_s if the entry s, t of the counter table is hit by the repeated header field, which is the source IP in our case. Now, our algorithm determines maximum of the table m_T. The network consists of H hosts which

	Normal Traffic	**10% Rate Attack**
Arrival Rate	$\lambda_l = 1\,s^{-1}$	$\lambda_a = 10\,s^{-1}$
θ_m	20.00	20.00
H	100	100
	Simulation	
Mean max	11.40	52.58
Maximum max	18.00	84.00
	Theory	
Expected max	11.41	50.00
FP Propability	0.035%	0.035%
FN Propability	-	0.000048 %

Table 3.3: Exemplary simulation results for a network with 100 Hosts

generate flows statistically independent, therefore the cumulative probability of m_T is:

$$P_{m_T}(n) = (P_{m_s}(n))^H = (1 - F_{n,\lambda}(t_W))^H$$

and the corresponding probability density is:

$$p_{m_T}(n) = P_{m_T}(n+1) - P_{m_T)}(n)$$

From this density the expected maximum can be derived with:

$$E(m_T) = \sum_{n=0}^{\infty} n \cdot p_{m_T}(n)$$

The results shown in Figure 3.7 support this theoretic model, the simulation fits the theoretical results very well, i.e. the analytical expected value matches the simulated mean.

With the help of this probability we can also get the FP probability for a given threshold θ_m, as this is the probability to reach a value of more than θ_m:

$$P_{FP} = 1 - P_{m_T}(\theta_m) = 1 - (1 - F_{\theta_m,\lambda_l}(t_W))^H$$

On the other hand, an attacker with the rate λ_a is not detected with the probability indicated by the FN rate:

$$P_{FN} = P_{m_T}(\theta_m) = 1 - F_{\theta_m,\lambda_a}(t_W)$$

As we assume only one attacker $H = 1$ for P_{FN}.

3.4.2.3 Simulation Results

We evaluated our system with a network of legitimate $H = 100$ hosts. We simulated 10 repetitions for each setting with a duration of 1000 s per run.

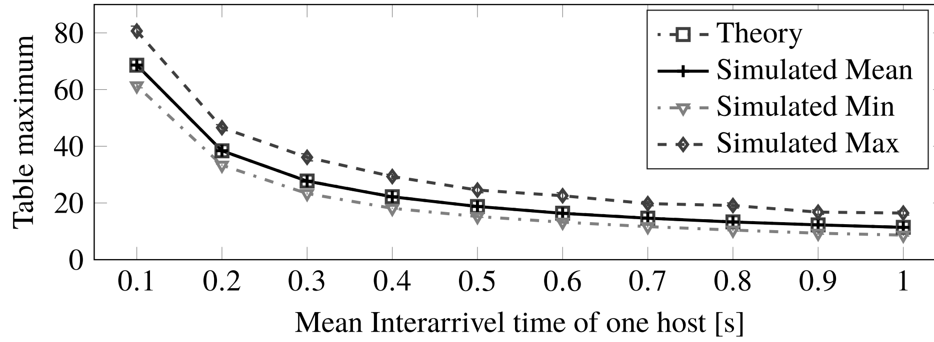


Figure 3.7: Results of the maximum table value for different inter arrival times.

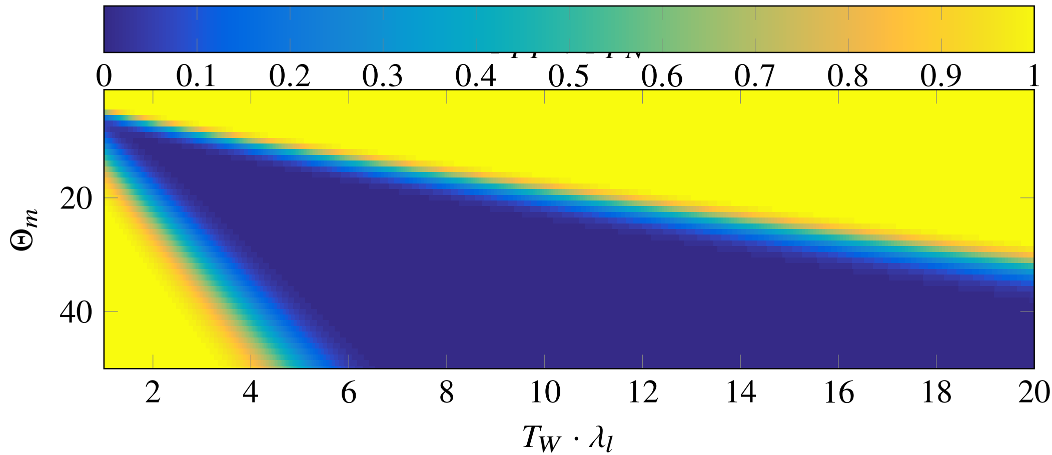


Figure 3.8: Working region of detection algorithm, with $\lambda_a = 10 \cdot \lambda_l$ and $H = 100$

Table 3.3 shows a comparison of the behavior without and with attack, in this simulation the maximum did not exceed a value of 18 so we did not have any false positives for the given threshold. As can be seen from the results, if an attack adds only 10% additional load to the system, it can be detected easily. Even with this relatively small additional load, the expected maximum value is about five times higher than the normal traffic. This large difference results in very small FN and FP probabilities. A 10% increase means that the table of the switch is filled 10% faster than usual, or if we take timeouts into account the table has 10% more entries. Usually, this comparably small increase should not affect the system's behavior.

Figure 3.7 shows the behavior of the max value for different traffic intensities. The simulation results match almost exactly the theoretic forecast. It can be observed that the table maximum is highly dependent on the traffic rate of one host. Therefore, it can be necessary to set the detection threshold according to the specific network environment.

Figure 3.8 shows the sum of the false negative and false positive probabilities. When the sum is close to 0 (darker) the algorithm is performing well. The lighter upper right region is caused by a high false positive probability the lower left by false negatives. For a given arrival

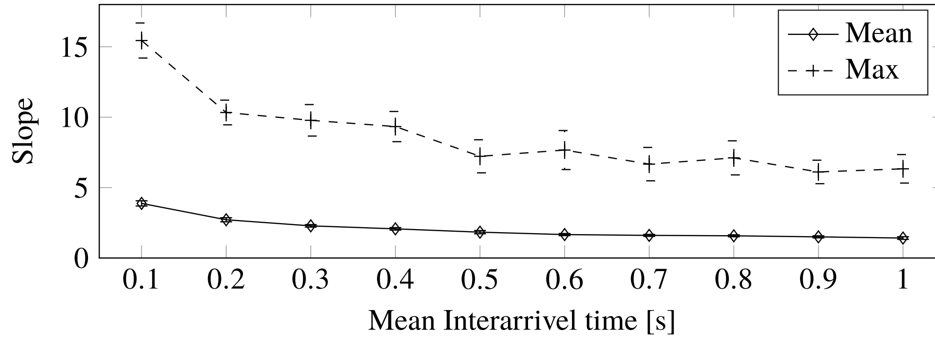


Figure 3.9: Results for the slope for different inter arrival times

rate a small detection window is beneficial as this improves the detection speed. Although if the product of window time and rate of legitimate hosts is big we have a very broad detection range, i.e. we have a big region where we can choose a good threshold while sacrificing the detection speed. On the other hand for a small product and consequently a small threshold the system is very sensitive for changes in the arrival rate of the users as this can cause false positives. As described in Algorithm 2, the maximum value is not the only detection metric. Also the difference between two consecutive maximas, the slope is considered. The results shown in Figure 3.9 show that the slope is not affected as much of the inter arrival time. On the other hand the results are much less stable and therefore are less dependable. Therefore the slope can be used if the protected network is largely unknown.

3.5 Summary and Discussion

In this chapter we introduced main attack vectors in SDN networks. Some attack vectors like attacks using rogue controller or a rogue app are inherent to SDN. Although they are not unprecedented as they are common for extensible software systems. One example are mobile operating systems and their applications. As the applications should be developed from a large number of developers, in order to increase the diversity and number of applications, some developers might not be trust worthy. In the network context the controller takes the role of the operating system and SDN apps take the role of mobile apps. Solutions are on the one hand a strict examination of the apps before they are made accessible to the users [52]. On the other hand solution like the SDN security kernel FortNox [45] that are restricting the access rights of the applications are viable solutions.

Further the network operator should always respect security best practices like authentication and encryption of the control connection. Even though the evaluation of the delay measurements indicate the importance of CPU power for a good OpenFlow control plane performance of a switch, specifically using encryption. We found that the software solution

Open vSwitch adds the lowest packet-in delay, this could give implications to future network device designs: Hardware acceleration support for encryption should be added to future SDN switches to keep delays small. Currently the OpenFlow testing tools do not support TLS, although our results show that encryption may have a noticeable impact on control plane performance.

Finally attacks that have been there in legacy networks, like e.g. spoofing or DoS attacks keep existing in SDN. Fortunately these issues can be solved more easily than in legacy networks, as additional software components can be employed. We developed and analyzed such a countermeasure aiming against DoS attacks. Our work concentrates on attacks which aim to overflow the hardware tables of SDN switches. The attacker causes a high number of PacketIn messages by changing header fields. Our proposed detection approach is based on the observation that an attacker cannot change all header fields. This allows us to identify the attack. Our approach uses a table with the header fields as columns and hashes of the header fields as rows. If an attack occurs the table entries corresponding to the unchanged fields grow tremendously. After identifying the attack, we propose to use an OpenFlow rule which drops further attack packets. Our evaluation shows that the algorithm can detect attacks reliably and with low false positive probability with the correct parameters. Using the proposed formulas it is possible to analytically determine a good choice of these parameters. For example for a mean arrival rate λ_l of 100 connections per second and a window time T_W of 40 ms, the detection threshold Θ_m should be between 15 and 40. The choice of these parameters yields a detection error that is lower than 5%.

Chapter 4

Isolation in Software Defined Networks

Network virtualization enables the use of multiple virtual networks using one physical network. The virtualization should provide networks to the hosts such that it matches the functionality and performance of multiple physical networks. Thus the networks must be isolated from each other.

Isolation in terms of security firstly means connectivity, respectively the circumvention of connectivity. In general Internet Protocol (IP) networks are designed to connect clients rather than prevent connections. However, not all clients in a network have good intentions. Therefore from a security perspective it is desirable to only allow as much connectivity in the network as strictly necessary for the respective purpose. To achieve this, the clients are logically grouped, e.g. in employee clients and guest clients. Further, connections are only allowed within one group, i.e. the groups must be isolated from each other. Thus, isolation between clients in multiple virtual networks is advisable and the separation in multiple networks is necessary.

Using multiple physical networks for each user group can provide the necessary isolation. Due to the overhead in terms of costs and overall complexity this is often not desirable. The task of network virtualization is therefore to provide an isolation that is equivalent to the isolation provided by physical networks.

In many companies and organization nowadays the network virtualization is provided by the IEEE 802.1Q standard [26]. It supports virtual networks by adding an extra header to the Ethernet frame header, a so called VLAN tag, which is then used by the switches and routers to enforce the isolation. The use of VLAN tags is well established though it is often tedious and error prone as the VLAN configuration of the network nodes is maintained mostly manually. Furthermore 802.1Q standard [26] itself does not include any authentication mechanisms for the clients. This means that, the virtual networks are configured interface based and everyone

that has access to the configured interface physically, can connect to the virtual network(s) configured on this interface.

Another problem arises due to the limited number of VLANs possible, 802.1Q provides only 4096 virtual networks, which is not sufficient in networks with many tenants, e.g., in cloud networks. Though this restriction is avoided often with the use of overlay networks like VXLAN [53]. VXLAN encapsulates the packet within an User Datagram Protocol (UDP) packet, this additional header can be used for tagging. The drawback of this approach is that the payload per packet is reduced due to the limited Maximum Transmission Unit (MTU) size.

Both protocols do not provide performance isolation between the virtual networks. Thus an adversary in one virtual network can overload the physical network and cause performance issues up to a complete denial of service in other virtual networks that share the same physical network.

In this chapter we show how Software-Defined Networking (SDN) networks can provide isolation. OpenFlow was developed based upon Ethane [12], that also introduces new filtering capabilities in the network. Thus it is also fairly straight forward to achieve isolation in OpenFlow. Nevertheless we want to introduce the basic concepts isolation in OpenFlow networks by defining virtual networks in general and how they can be realized using OpenFlow in Section 4.1. We use this capabilities for an approach that realizes fine grained filtering using SDN presented in Section 4.1.2. Further performance isolation between different virtual networks can be realized in Section 4.2. Performance isolation is more difficult to achieve in practice, thus the main contributions described in this chapter lay in this field. We show how different OpenFlow devices perform in this case and what is necessary to improve upon it. Finally, in Section 4.3 we conclude the chapter.

The performance isolation of OpenFlow switches that is described in this chapter was presented first in [3]. The detailed discussion regarding isolation using OpenFlow in Section 4.1 was not published before, however the general concept of isolation using SDN was demonstrated in [4, 5].

[3] R. Durner, A. Blenk, and W. Kellerer. “Performance study of dynamic QoS management for OpenFlow-enabled SDN switches.” In: *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*. 2015, pp. 177–182. DOI: 10.1109/IWQoS.2015.7404730.

[4] S. Gebert, T. Zinner, N. Gray, R. Durner, C. Lorenz, and S. Lange. “Demonstrating a Personalized Secure-by-Default Bring Your Own Device Solution Based on Software Defined Networking.” In: *28th International Teletraffic Congress (ITC 28)*. 2016, pp. 197–200. DOI: 10.1109/ITC-28.2016.133.

[5] B. Pfaff, J. Scherer, D. Hock, N. Gray, T. Zinner, P. Tran-Gia, R. Durner, W. Kellerer, and C. Lorenz. "SDN/NFV-enabled Security Architecture for Fine-grained Policy Enforcement and Threat Mitigation for Enterprise Networks." In: *Proceedings of the ACM SIGCOMM Posters and Demos*. 2017, pp. 15–16. DOI: 10.1145/3123878.3131970.

4.1 Isolation in OpenFlow enabled Networks

In this section, we will focus on isolation in terms of connectivity. Within this scope, isolation for virtual networks can be defined as the ability of the network operator or management system to deny or allow the connectivity between different clients. Clients can be physical clients or virtual machines or even more fine granular applications using the network. Furthermore the virtual network consists of a subset of the links of the physical network. In packet switched networks this means that the forwarding nodes in the network (switches, routers etc.) need to be able execute two main tasks:

A) Detect the membership of a packet to a virtual network.

B) Accept or deny sending and receiving a packet on a link depending on the membership of the link and packet to the virtual network.

These tasks must be executed for incoming packets and for outgoing packets after the forwarding decision as only then the outgoing link is known. In the case of 802.1Q [26] task A is realized with the VLAN tag that is added to the Ethernet header. This header is evaluated by the forwarding nodes. Task B is realized with port memberships, if a port is member of a VLAN corresponding packets can be sent and received with this port.

SDN and especially OpenFlow isolation can be used as follows: The basic concept of the OpenFlow packet pipeline is the match action semantic. Incoming packets are checked against the rule table, if a rule matches the packet the defined actions are performed. The match part of a rule can consist of arbitrary header fields like e.g. Ethernet addresses, IP addresses or also higher layer fields like TCP and UDP ports. The possible actions include forwarding to a port and dropping. This pipeline of OpenFlow supports all necessary tasks to implement isolation of virtual networks. Through matching, a node can detect a membership of a packet to a virtual network. The action can then be either drop or forward depending on if the link is a member of the respective virtual network. Dropping received packets from links that are not part of the virtual network can be realized by adding rules that contain the input port as part of the match. In contrast to other widely used approaches like 802.1Q and VXLAN, OpenFlow

does not rely on additional headers in order to map a packet to a virtual network. Instead the complete header can be used for that purpose. Additionally many OpenFlow capable devices provide hardware support for these tasks.

This native support is a direct consequence of the history of OpenFlow. OpenFlow was introduced 2008 [13] aiming for more flexible and innovative networks in general. Though the basic concept was already introduced with Ethane [12]. Ethane was introduced to improve visibility and security in enterprise networks. It introduces a controller that holds the network policy, all first packets are presented to the controller and the connection is only allowed if it is allowed by the policy. This largely improves manageability, as the policy is centralized in one location, furthermore this also improves security as the Ethane architecture provides means to control the connectivity between hosts. OpenFlow as a standard, has inherited these isolation features of Ethane.

To summarize, OpenFlow can provide the tools for network virtualization and isolation between the virtual networks. Though OpenFlow does not specify how the virtual networks should be managed and which header fields should be used to identify the virtual network.

4.1.1 Related Work

Besides the isolation that is supported by the OpenFlow architecture, also the management of the virtual networks has to be provided. Especially in more complex scenarios with lots of virtual networks and also networks that include rewriting of headers, the isolation can be violated.

In order to avoid inconsistencies in the network like e.g. conflicting rules or routing loops formal methods were proposed. FlowChecker [54] was one of the first proposed approaches to solve this issues: Networks are represented as state machines where the state represents the header fields and the location of a packet. Network devices may change the state by changing some header fields, changing the location of a packet or removing the packet from the network. Using this framework the isolation of virtual networks can be verified. Other works also included other network protocols like Multi Protocol Label Switching (MPLS) into the verification [55] and increased the speed of the verification to support real-time verification[56, 57]. Furthermore Policy Graph Abstraction [58] is a framework that supports simpler high level policies and additionally supports service chains composed of multiple middle boxes.

Fresco [59] introduces a scripting language that can be utilized to develop SDN security applications while guaranteeing isolation and conflict free network rules.

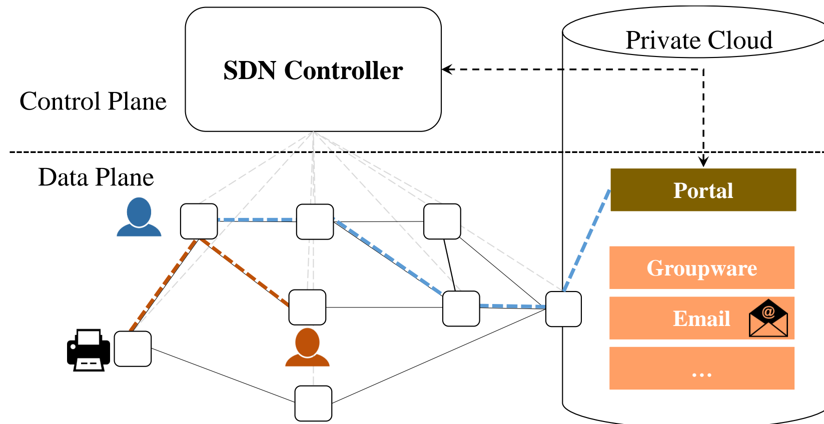


Figure 4.1: Virtual networks are provisioned for each service. By default the user has only access to the portal.

An orthogonal approach are SDN Hypervisors, this concept was presented first with FlowVisor [60]. With this approach each of the virtual networks can use its own SDN Controller, while the SDN Hypervisor has to guarantee the isolation between the networks. The main advantage of this approach is that different virtual networks can be tailored to different needs of the virtual networks using multiple SDN controllers. A more detailed description of this field of network virtualization can be found in a recent survey by Blenk et. al [61].

4.1.2 Fine-grained virtual networks with OpenFlow

The proposed architecture aims at achieving a fine grained isolation in order to reduce attack surface as much as possible. The isolation is designed to use a Virtual Network (VN) for every user service relation. More concrete, five tuples are used as identifier for the VN, as most services require bidirectional communication, we also use the five tuple with switched direction in the same VN. An example for a VN for the user 10.0.0.25 with port 23560 to access a web server at 10.0.1.60 on port 80 using TCP would include all packets with the five tuples (TCP, 10.0.0.25,23560, 10.0.1.60, 80) and (TCP, 10.0.1.60, 80, 10.0.0.25, 235600) in the VN. The concept is sketched in Figure 4.1. The picture shows how SDN can be used to allow the packets of the specific VN only on the necessary physical links between the user and the service.

In practice this is implemented as follows: By default each user can only access the portal. The portal can be used to enable services like printing, groupware or email. If a service is enabled at the portal the controller is commanded to allow the provisioning of the VN using the Northbound API. When the user tries to connect to the service the first packet of the respective connection is forwarded to the SDN controller. The SDN controller checks if

the VN is allowed and provisions the virtual network upon a positive result. This two step approach is necessary as the user port is only chosen when the user actually starts the service on his device, furthermore this also reduces the number of necessary rules as only rules that are actually used are configured in the tables of the devices. If rules are not used anymore they are removed automatically using soft timeout functionality of the switches. We envision to use a private cloud system in order to host the portal and other company services such as Email or Groupware.

The functionality in the SDN controller was implemented as an application running in the ONOS Controller in the Demos [4, 5]. An implementation using the Northbound API was not possible, as the Northbound API does not provide access to packet-in messages. This approach has the drawback that the solution is not easily portable to other controllers.

The portal can be used to implement access levels for user groups, e.g. an ERP system that should be only available for certain users. Further the portal can as well be used to map organizations business processes, e.g. approvals of superiors and logging into the solution. As a summary the presented VN approach provides a rich toolbox that can make enterprise networks more secure and manageable. It can be seen that isolation in OpenFlow networks can be achieved without large modifications. On the other hand performance isolation as shown in the next section is more difficult to achieve in practice.

4.2 Performance Isolation in OpenFlow enabled Networks

The previous section showed that SDN and especially OpenFlow can provide isolation in terms of connectability. On the other hand this is not the full scope of isolation as the different virtual networks still interfere w.r.t. the performance. E.g. if a link is shared between two virtual networks one virtual network can exhaust the full link and thus cause outages in the other virtual network. With the emergence of SDN, a new flexible network operation and control is possible. Such a flexible operation and control demands the realization of well established Quality of Service (QoS) concepts. SDN promises to provide a powerful way to introduce QoS concepts in today's communication networks. Using SDN and OpenFlow for realizing new resource management, existing studies, such as [62–64], have shown that applications benefit from a fast and frequent change of network resource allocations, which we call dynamic QoS management. For instance, for progressive video streaming applications, [63] showed that changing the queue assignment of video flows based on the currently buffered playtime avoids stalling. Stalling has a high negative impact on perceived application quality of network users.

Prior research on dynamic QoS management in SDN-based networks, e.g., [62–64], has neglected the diversity of existing hardware, i.e., switch diversity. However, in existing work [65, 66], it was shown that switches from different vendors show different behaviors, e.g., for flow installation times or for different order of OpenFlow rule operations. Ignoring such switch diversities may lead to significant performance degradations in SDN networks and can cause performance interference between virtual networks. Accordingly, the diverse behavior has to be taken into account when designing SDN applications in general, and when designing SDN applications based on dynamic QoS management in particular.

In the following, we study effects on network traffic when applying dynamic QoS management in OpenFlow-based SDN networks. We measure the impact on TCP network flows when changing the queue assignment of multiple flows at runtime. Our measurement study is done for different configurations of two fundamental quality of service concepts, namely priority queuing and bandwidth guaranteeing, which are deployed on three switches. The switches are the NEC PF5250, the P3290 from PICA8, and the software switch Open vSwitch (OVS) [67]. Furthermore, as OVS is implemented in software, it provides detailed information on the used queuing disciplines and the used queue implementations, e.g., First-In-First-Out (FIFO) queues or Stochastic Fairness Queuing (SFQ) queues. Using the OVS behavior as reference allows us to draw conclusions about the used queue implementations of the hardware switches, which is only rarely publicly available.

4.2.1 Related Work

Previous work on providing QoS guarantees using OpenFlow can be partitioned in three categories. First, studies deploying dynamic QoS in an SDN environment [62, 63, 68]. Second, studies on switch diversity [65, 66, 69, 70]. Third, research on network performance resulting from QoS with OpenFlow-enabled switches [71, 72].

The OpenFlow-assisted Quality of Experience Fairness Framework (QFF) provides user-level fairness for adaptive-video streaming such as Dynamic Adaptive Streaming over HTTP (DASH). QFF guarantees QoS in the network in order to provide users suitable and more stable bandwidths. As DASH also utilizes long living TCP flows, we expect the appearance of the same effects as we show in this paper with the use of QFF. The study [62] only considers effects on the video application, while we measure effects in the network.

[64] introduces a QoS controller prototype that guarantees end-to-end QoS by routing the flows based on performance requirements. With the arrival of a new flow, the controller calculates the resource allocation and installs the necessary rules with QoS guarantees. Although the controller prototype is examined using one hardware and one software switch, the focus

of the analysis lies on the QoS control framework and not on the effects that these guarantees cause when they are applied dynamically.

VMPatrol [68] employs a framework that limits the used portion of the available bandwidth when migrating Virtual Machines (VMs) in a cloud environment, for the sake of the productive traffic. The conclusion is that QoS mechanisms can decrease the adverse impact of VM migrations on other network flows. This is an important use case for dynamic QoS with OpenFlow, as many SDN researchers currently focus on virtualized environments.

The study [63] examines bandwidth management depending on the currently buffered playback time of YouTube progressive video streams. The study also measures and compares the impact of different queuing strategies. Different queuing strategies are found to lead to varying buffered playback times of the videos. In contrast to our study, [63] does not use OpenFlow and does not compare different switches. Additionally, [63] focuses on resource management, while we concentrate on detailed measurements of different dynamic QoS settings.

Tango [65] is a switch probing engine that aims at countering problems that result from switch diversity. In contrast to our approach, Tango analyzes the diversity regarding flow installation and manipulation timing in the control plane, while we focus on the effects that rule changing has on ongoing flows in the data plane. However, we believe that our observations can also be integrated in a framework such as Tango.

Recent studies [66, 69, 70] analyze the diversity of switches regarding the SDN control plane behavior. For instance, [66, 70] analyze rule installations in combination with barrier replies that should confirm the rule installation and show that some switches confirm the installation of rules prematurely. The barrier reply is sent out before the OpenFlow rules are actually active. Additionally, the relationship between the number of flow rules and data plane performance, as well as the impact of rule priorities is measured. [69] measures the processing capabilities of hardware and software switches for specific OpenFlow message rates.

[72] investigates data-rate guarantees equipped at the Pica8 P3290 switch. The influence of short living UDP flows, also called bursts, to a long living UDP flow is measured. The main result is that bandwidth guarantees of the P3290 switch are not fully enforced against such bursts. Therefore, isolation between the data flows can not be guaranteed. While this work focuses on UDP traffic, we investigate the behavior of TCP traffic and use a dynamic resource management instead of static QoS, which does not change over the course of the experiment.

The study [71] examines the OpenFlow metering feature. The metering feature limits the data rate of a flow to a desired data rate. In this study, meters are applied to TCP

flows on a bottleneck link. The authors observe that bursts of packets are dropped, which wastes resources and causes TCP to leave the congestion avoidance state and regress to the slow start state frequently. Although the setup is quite similar to our setup, the study is limited to measurements based on the network emulator Mininet and does not evaluate real hardware. Besides only static QoS setups are studied, while we analyze the dynamic behavior. Nevertheless, to our best knowledge, this study is most closely related to our analysis.

4.2.2 Background on Queuing Disciplines

As OpenFlow is only an API for the forwarding of packets, differences between switches may show different behaviors with respect to QoS. In this section, we explain queuing techniques that are used by the studied switches. OpenFlow claims to be vendor neutral, but as OpenFlow is only an API for the forwarding of packets, differences between switches may show different behaviors w.r.t. QoS. In this section, we explain queuing techniques that are used by the studied switches.

Classless Queuing Disciplines Classless queuing disciplines forward traffic neutrally. Neutrally has different meanings for packets and for flows, which lead to different implementations:

- First-In-First-Out (FIFO) queues have a very simple structure. The item that was first put into the queue leaves it first. In the case of a full queue, incoming packets are dropped. FIFO is fair in the sense of individual packets, as every packet is processed equally.

- Stochastic Fairness Queuing (SFQ) [73], in contrast, is a queuing mechanism that tries to ensure fairness between several flows. The basic structure is shown in Figure 4.2. In order to provide fairness, incoming packets are enqueued to several queues. The selection of the queue is based upon a hash of the source and destination IP plus the source port. Packets are dequeued using the round-robin scheme between the queues. This mechanism results in a high probability that flows are processed fairly, although the fairness is not deterministically guaranteed. Consequently, the mechanism is called Stochastic Fairness Queuing.

Classful Queuing Disciplines In contrast to the previous disciplines, Classful Queuing Disciplines forward packets according to a scheduling algorithm.

- Priority Queuing (PQ) uses queues with different priorities. The packets of the queues are dequeued according to their assigned priority, i.e., a queue is only served if no queue with higher priority has packets.

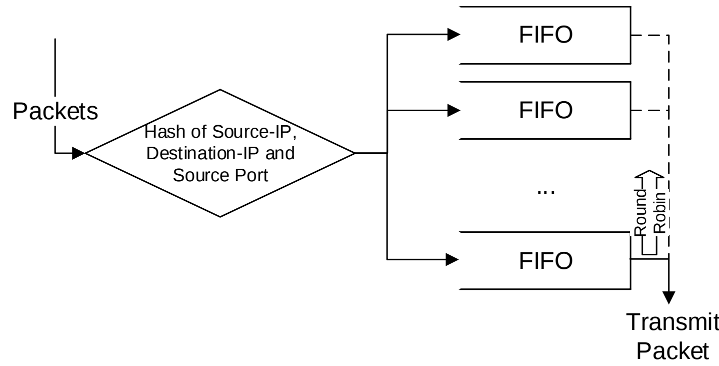


Figure 4.2: Stochastic Fairness Queuing Algorithm

- Weighted Fair Queuing (WFQ) guarantees a specific, configured minimum bandwidth for each queue. The remaining unused bandwidth is shared between the queues.
- Hierarchical Token Bucket (HTB) is used by OVS for providing bandwidth guarantees (BG). While HTB affects the rate at which packets are sent from each queue, the type of the underlying queue affects the order of the sent packets. This underlying queuing discipline can be, for example, FIFO or SFQ.

4.2.3 Measurement Setup

In this section, we describe the measurement setup and procedure and some background on queuing disciplines. We designed an experiment that examines dynamic QoS support of OpenFlow specifically for TCP Flows. We probe and compare different queuing techniques on multiple switches.

4.2.3.1 Experiment Setup

As shown in Figure 4.3, two physical hosts act as TCP traffic source and TCP traffic sink. The hosts are connected via the SDN switch. An additional host runs the SDN Floodlight controller, which is connected to the SDN switch. The Round Trip Times (RTTs) in the measurement network are <1 ms. The application *iperf* [74] is used in version 2.0.5 for the generation of the TCP traffic. The hosts run Ubuntu Linux 14.04 and Linux kernel version 3.13. The data flows are recorded with *tcpdump* [75] and analyzed afterwards. We use TCP CUBIC and disable the features segmentation offloading and TCP metrics save.

As QoS mechanisms are only useful if multiple flows interact with each other, we use two TCP flows that are routed via a single bottleneck link. This link's data rate is limited to 10 MBit/s, while the other link is unrestricted and has a maximum rate of 1 GBit/s. The

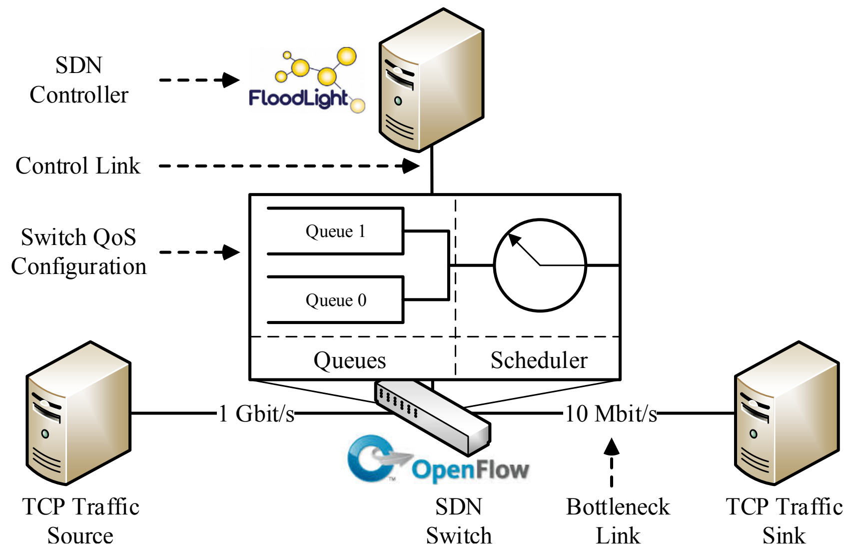


Figure 4.3: OpenFlow-based testbed setup. Queue 0 (q_0), Queue 1 (q_1), and Scheduler are configured for each experiment accordingly.

bottleneck causes that packets are enqueued at the switch. The OpenFlow rules that are necessary for the experiments are set via Floodlight's REST-API [76].

The course of the experiment is presented in Figure 4.4, It is divided in three stages. At time $t = 0$, both TCP flows are started and forwarded over the same queue (q_0). As TCP provides fairness, they share the bandwidth approximately equally. At time $t = T1$, one flow is rerouted via a different queue (q_1). Depending on the settings of the queues, one flow has more bandwidth than the other in the second stage ($T1 < t < T2$). In the third stage, from time T2 onward, the flows are again forwarded together in q_0. Each of the stages has a duration of 30 s, which means that T1 is 30 s and T2 60 s after the start of the run.

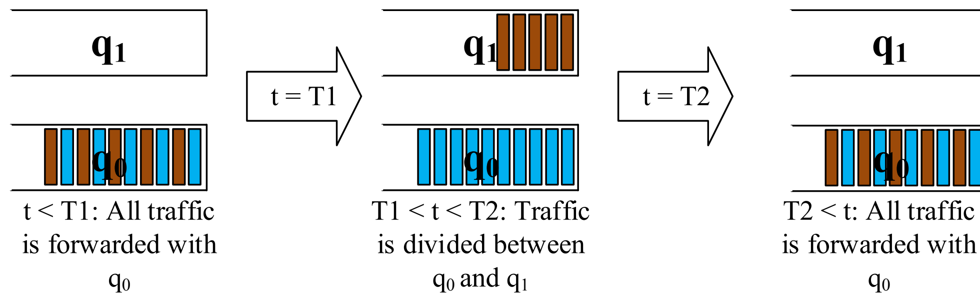


Figure 4.4: Time sequence of the experiments

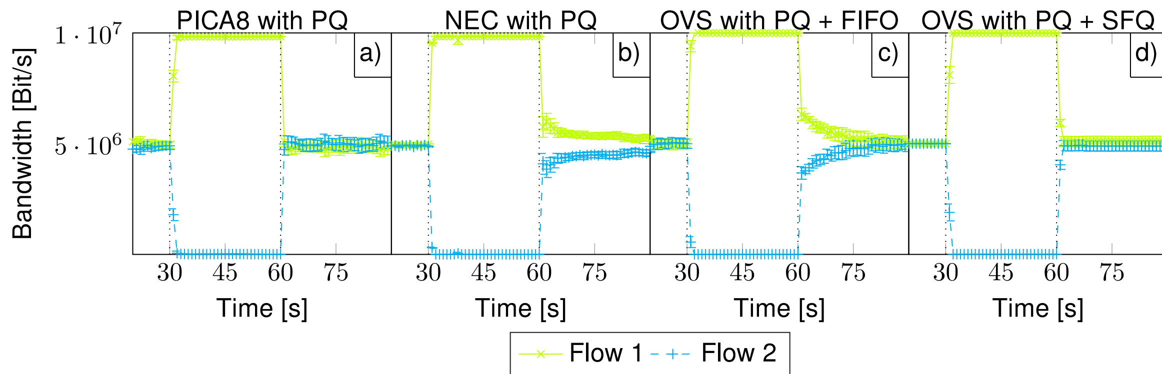


Figure 4.5: Measured mean bandwidth over time using different switches and priority queuing. 95% confidence intervals for the mean values are added to all plots.

4.2.3.2 Investigated Switches

In our studies, we investigate two hardware switches from different vendors, namely an NEC PF5240 and a PICA8 P3290, and the OVS [67], which can run on commodity hardware.

The NEC switch supports multiple QoS techniques, while only PQ and WFQ are studied in this paper. WFQ is used to provide Bandwidth Guarantees Each queue has a length of 64 packets. The P3290 is a so called bare metal switch and supports different network operating systems. In our setup, the P3290 runs PicOS in version 2.1.5 in OVS-mode. In the OVS-mode, Open vSwitch is running on top of PicOS and provides the OpenFlow interface. Regarding the configuration of the queues, priorities and a predefined minimum and maximum bandwidth can be set. The OVS runs version 2.1.0 on the traffic sink. The advantage of OVS is that we can change the settings of the queues with the Linux application TC [77].

4.2.4 Measurement Results

The following figures show the results of 50 runs. We studys the bandwidth of the flow over time and the sum of the duplicated packets for intervals with a length of 10 s around T1 and T2. We examined Queues using PQ and Bandwidth Guarantees.

4.2.4.1 Priority Queuing

Figure 4.5 shows the results for PQ. In the first stage, both flows are in q_0 and share the bandwidth equally. q_1 is set to have a higher priority than q_0, therefore Flow 2 is depleted very fast after T1. After T2, Flow 1 is again together in q_0 with Flow 2. For the PICA8 switch, we can observe that Flow 2 recovers pretty fast. This can be explained as follows: In the first stage, Flow 1 via q_1 prevents the submission of packets from Flow 2 via q_0. In the meantime, as the source of Flow 2 still tries to reestablish the connection, these packets are enqueued at

q_0 but can not be forwarded and are not dropped. After T2 both Flows are forwarded via q_0 and thus these packets of Flow 2 are flushed to the sink. The sink acknowledges these packets, which causes the congestion window and, thus, the bandwidth of Flow 2 to increase quickly.

The number of duplicated packets at the sink, that is shown in Figure 4.6 confirms the observation. Packets of Flow 1 are duplicated at both switchover points (T1, T2). At time T1 the packets in q_1 pass packets that are enqueued in q_0 and cause the sink to request the slower packets again, which results in duplicated packets. This means that short after T1 packets from both queues are sent, in contrast to the rest of the second stage. At time T2 packets from q_0 are submitted again, at this time some retransmission packets from Flow 2 are enqueued that were sent during the second stage and appear at T2 as duplicates.

The NEC Switch shows a different behavior in the third stage. When Flow 1 is rerouted to q_0 at T2, the average bandwidth of Flow 2 grows fast first, until it again reduces. The time until fairness between the two competing flows is regained is longer than 30 seconds. The number and distribution of duplicated packets are similar to the results with the PICA8 switch, although fewer packets are duplicated at T1. On the other hand, duplicated packets of Flow 1 are measured at T2. A detailed analysis shows that these packets were stuck in the queue since T1. The very steep ascent of Flow 1 at T1 causes this behavior.

For comparison, we have examined OVS with two different classless queuing disciplines, namely SFQ and FIFO, which handle the transmission of packets out of q_0 and q_1. The FIFO queues are configured with a length of 64 packets. The SFQ queues have a length of 128

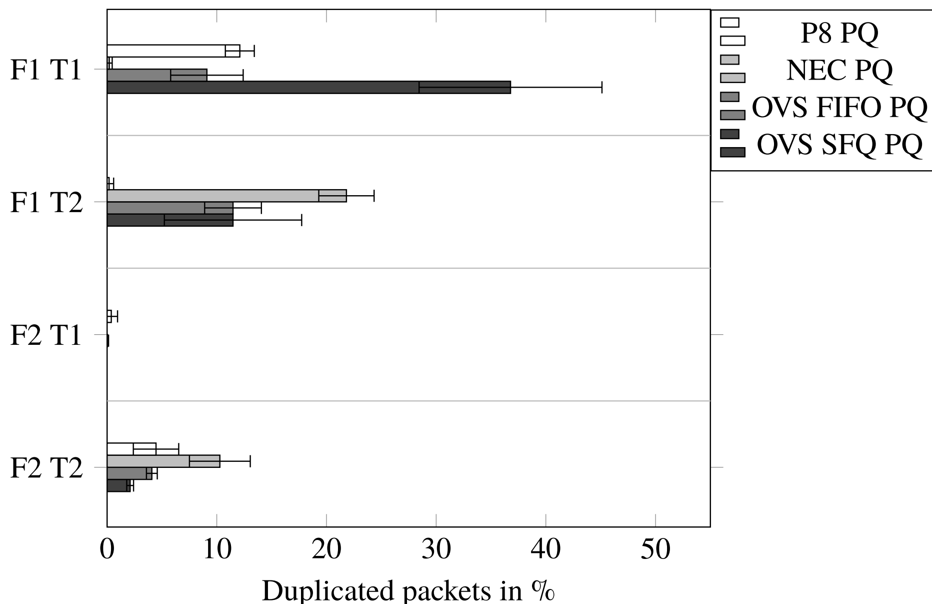


Figure 4.6: Mean number of duplicated packets of Flow 1 (F1) and Flow 2 (F2) in an interval of 10 s around T1 and T2. 95% confidence intervals are included. Measurement at the sink of the flows.

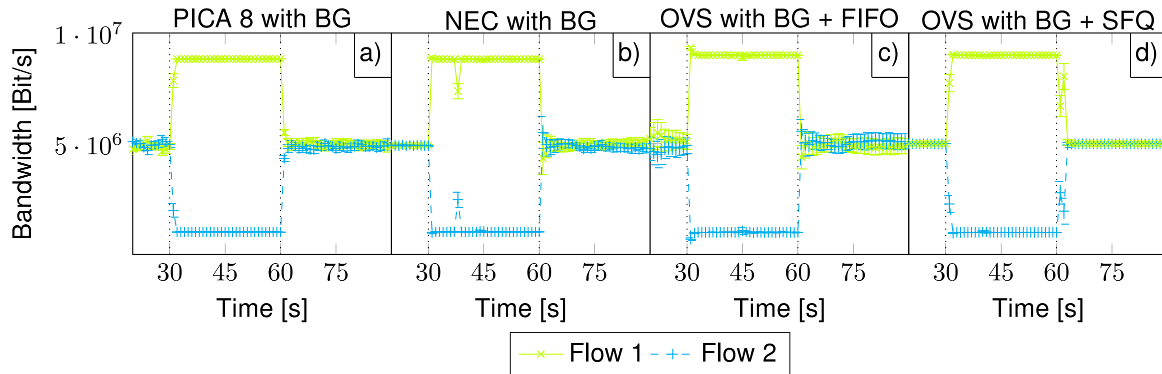


Figure 4.7: Measured mean bandwidth over time using different switches and bandwidth guarantees settings. 95% confidence intervals for the mean values are added to all plots.

packets, which is fixed at compile time. Until T2 the differences between FIFO and SFQ are marginal. After T2, fairness is established very fast and accurate, if SFQ is used. This is the expected behavior as SFQ tries to ensure fairness between all flows. However, the results for FIFO are quite different: The time until the two flows have again approximately the same bandwidth is in the range of 20 to 30 seconds. We can observe a higher number of duplicated packets when OVS is used at both switchover points. Especially SFQ causes a high number of duplicated packets. This can be justified with the bigger size of the queues.

From the results we can see that only OVS with SFQ behaves as one would naively expect. In a network with different switches, the discovered diversities will degrade the performance of QoS-oriented SDN applications, if they are not prepared accordingly. Duplicated packets need to be taken into account when deploying and using QoS mechanisms, as they cause a waste of bandwidth.

4.2.4.2 Queuing Disciplines with Guaranteed Bandwidth

One of the key features of SDN is Network Virtualization. Virtual Networks can be allocated to different queues. Queues receive a data rate guarantee and, therefore, provide network resource isolation. In the following, we investigate queues with performance guarantees. The basic setup stays the same, although now, the bandwidth in the second stage ($T1 < t < T2$) is to be shared according to a 1:9 ratio. The sum of the guaranteed rate equals the link speed. This results in 1 Mbit/s guarantee for q_0 and 9 Mbit/s for q_1. The NEC switch supports WFQ, while the other switches can employ bandwidth guarantees for their queues.

Using the PICA8 switch, after T1 the bandwidth of Flow 2 decreases very fast to the configured 1 Mbit/s. After T2, fairness is reestablished quickly. The number and distribution of duplicated packets is comparable to the PQ experiment.

In the second stage the bandwidth guarantees are violated for a short instance using the NEC product. The drop is visible for approximately 6% of the measurements always 8 seconds after T1. This regularity indicates some switch fault. As no packets of Flow 2 are accumulated during stage 2, fairness is established more steadily compared to the PQ case. Regarding the duplicated packets, one can see the same behavior as for the PICA8 switch, although more packets are duplicated in total.

The bandwidth results for the experiments with OVS and SFQ are close to ideal: at T1 the bandwidth of Flow 1 increases in less than two seconds to the predefined level and at T2 fairness is reestablished very fast. OVS and FIFO shows a different run of the bandwidth. Especially interesting is the behavior at T2: the bandwidth of Flow 2 overshoots Flow 1 and afterwards both flows oscillate around the fair distribution. The number of duplicated packets is much higher, when compared to the hardware switches.

Summarizing the results, we can see that the guarantees, as they do not cause the depletion of one flow, do not harm the overall connectivity of the network. Vendor dependencies for this setting are less distinctive compared to priority queuing, although with bandwidth guarantees none of the flows can use the full bandwidth of the link, which might be desired for some applications.

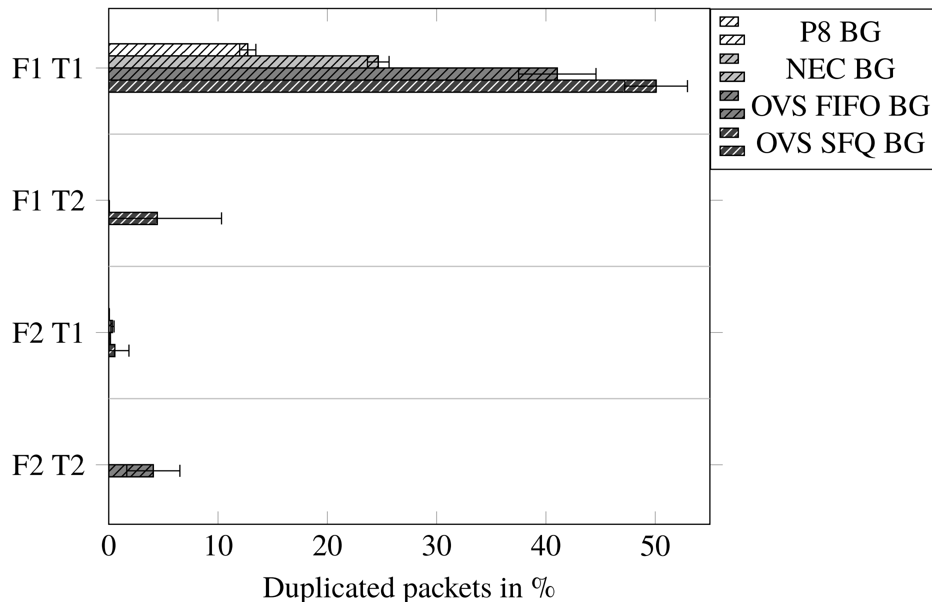


Figure 4.8: Mean number of duplicated packets of Flow 1 (F1) and Flow 2 (F2) in an interval of 10 s around T1 and T2. 95% confidence intervals are included. Measurement at the sink of the flows.

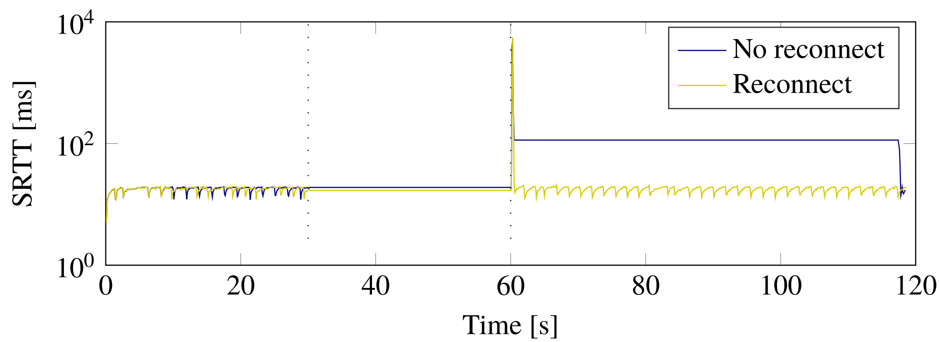


Figure 4.9: Smoothed round trip time of Flow 1 from two different measurements using priority queuing on the NEC Switch

4.2.4.3 PQ and TCP's Retransmission Behavior

The results show that Priority Queuing with TCP flows can lead to the depletion of the flow in the lower priority queue. As TCP always interprets packet loss as congestion, the complete depletion causes the source to throttle its sending rate to almost zero. When a TCP flow is depleted, only some retransmissions are sent in an exponentially increasing interval starting with 1 s according to the retransmission timeout (RTO) timer definition in RFC6298 [78]. If we choose $T2 - T1 = 30$ s Flow 2 is blocked for 30 seconds and retransmission are sent 1 s, 3 s, 7 s and 15 s after T1. This results in an RTO of $2^4 = 16$ s at the end of the blocked section. When the transmission restarts at $t = 60$ s, RTO is not reset immediately. Instead RTO is computed out of RTT measurements at the source, which are averaged out and result in the so called Smoothed Round Trip Time (SRTT).

SRTT of Flow 2 is plotted for two different runs in Figure 4.9. Until T1, SRTT moves around about 20. Between T1 and T2, no packets are transmitted, therefore RTT measurement is not possible and SRTT is constant. At time T2, the packets that were stuck in q_0 are flushed to the sink, which acknowledges these packets. This leads to some very high RTT measurements at the source and, thus, a very large SRTT. When no further packets are lost, SRTT gets small again pretty fast as more and more RTT measurements after T2 are successful. On the other hand, if the reconnect after T2 fails, due to packet loss, this results in a subsequent connection interruption. The TCP stack of the source waits RTO until it sends another reconnect. As SRTT is still high, RTO is also big and it can take many seconds until the TCP flow restarts again. In the pictured measurement it took 60 seconds until the connection was finally reestablished.

Altogether this can lead to an interruption of the TCP connection. In the worst case, the application running the TCP connection does not restart the connection on its own and a user intervention is necessary. This means that in case of PQ starvation of flows should be avoided

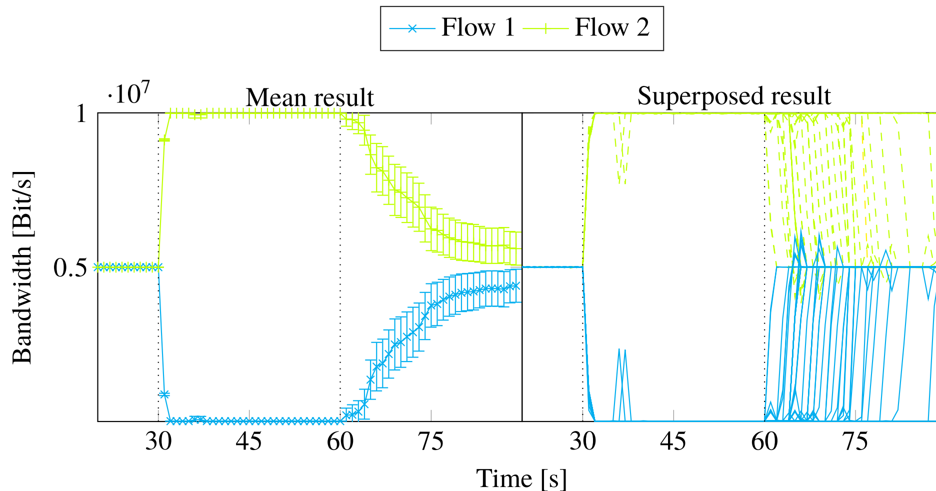


Figure 4.10: Bandwidth of 50 runs using OVS employing inverted PQ and SFQ

or at least kept short. An active notification to the source host from the switch at T2 could avoid the problem.

4.2.4.4 Inverted Priority Queuing

This behavior has a major influence on the result with "inverted" priorities. "Inverted" priorities means that first both flows are routed over the queue with the higher priority, then one flow is routed over the lower priority queue until both flows are again together on the higher priority queue. We performed experiments with inverted PQ for all switches, as the results are all comparable with some minor differences, we only show the results for OVS with SFQ in Figure 4.10. The mean result gives the impression that the time until fairness is reestablished is very long. But in this case the superposed single results give a better insight. These plots show that if Flow 1 is reconnected, fairness is restored pretty fast. But the time when Flow 1 is reconnected is very divergent. Between T1 and T2 no packets can be transmitted from Flow 1, the source tries to reestablish the connection with retransmissions. In contrast to the previous experiments, these packets are still retained also after T2. Therefore the only way that Flow 1 reconnects is a retransmission packet after T2, although, as Flow 1 was disconnected fairly long, the frequency of retransmissions is very low. The link is still well loaded by Flow 2, therefore the probability is high, that one or more of these packets are dropped, which can stop the TCP connection again for many seconds. As this is a non-deterministic process each experiment is different. Summarizing this it must be noted that the queuing approach regarding PQ makes a major difference.

4.3 Summary and Discussion

In this chapter we show how SDN and more specifically OpenFlow can provide isolation for virtual networks. We study isolation on two different levels: Firstly isolation must prohibit access between different hosts that are not in the same virtual network. We discuss the general fitness of OpenFlow regarding access isolation. Further we show how a fine-grained access control using VNs can be realized.

Secondly the performance of the slices must be decoupled as well, i.e. depletion of the bandwidth resources in one virtual network must not influence the performance of other virtual networks. Performance isolation can be realized using dynamic QoS mechanisms in SDN networks. Although SDN and, in particular, OpenFlow claims to provide a standardized interface, the existing diversity of OpenFlow-enabled switches leads to varying behavior for the same QoS mechanisms. Existing work on realizing QoS mechanisms and on OpenFlow switch diversity has neglected the impact of dynamic QoS mechanisms on network traffic.

Accordingly, in this chapter, we study switch diversity while deploying dynamic QoS mechanisms for TCP flows in an OpenFlow-based network. Our measurement results for two fundamentally different QoS techniques, namely priority queuing and bandwidth guaranteeing, show severe performance variations for two hardware and one software switch. Using priority queuing, the different switches lead to different, even unfair, bandwidth shares between TCP flows. Furthermore, the use of priority queuing can interrupt TCP connections, which can result in nondeterministic flow behavior. Besides, our results for the software switch show that different queue implementations, i.e., FIFO queues or SFQ queues, also impact the network performance. In case of bandwidth guarantees that can be used for performance isolation between virtual networks, one hardware switch violates the configured bandwidth guarantees. This means even though OpenFlow is standardized, performance isolation can not be guaranteed for all devices. Instead for a real deployment the fitness must be determined for each device. More specifically we saw that rescheduling of queues due to QoS requirements can cause duplicates and performance degradation. SFQ can reduce this issues, though it is not supported by the employed hardware switches. Thus, a realization depends on the used devices, which evidently hinders flexibility.

Chapter 5

Performance of Security Virtual Network Functions

In the previous chapter it was described how Software-Defined Networking (SDN) can be used to provide fine-grained isolation on a connection level. This kind of isolation provides stateless filtering as the network packets are separated based on their header fields, without considering connection state or payload of the packets. However according to requirement R2, the security architecture presented in this work shall also provide stateful and application layer filtering of packets. The architecture uses Network Function Virtualization (NFV) in order to provide the necessary performance, fulfilling requirement R3. Further NFV also provides increased flexibility as it supports scaling of the network functions depending on the network traffic.

One main challenge in the field of NFV is the performance and more specifically lack of performance guarantees of the NFV environment. In contrast to traditional solutions, hardware and software are no longer developed together, but the Virtualized Network Functions (VNFs) should run on commodity hardware. One issue is the large number of packets that have to be processed. For simple VNFs, like for example a load balancer, the complexity of processing one packet is very low. If a hash table is used, it can be as low as O(1). Because of that, other effects which can be usually neglected gain more importance. E.g., [79] shows that shuffling a packet processing workload between cores can reduce the packet rate significantly, as the data locality is not kept.

In this chapter the memory architecture of modern CPUs with regard to packet processing is evaluated. VNFs are I/O intensive applications as they mostly do not make many computations but rather handle a lot of data. This data is first the packets that need to be loaded to the CPU to be changed or accessed by the CPU. Secondly stateful or higher layer network functions that need to trace connections like e.g. stateful firewalls or application layer gateways need

to access the data corresponding to the processed packets. The data can be for instance a connections state. Consequently the memory architecture of the hardware performing all these operations is very important.

The server market is dominated by Intel, e.g. it is reported that Intel has a market share of 98% of all the server CPU market in 2018 [80]. As also the second in market AMD uses an x86 architecture this architecture can be assumed to be dominant for NFV as well.

One important concept of this architecture is the Non Uniform Memory Access (NUMA), that is used to support multiple CPU chips in one system. On the other hand this architecture causes performance penalties due to copy operations between the CPU chips.

Further, CPU caches are used to improve the access times to memory that is needed regularly for the execution of a program. As the CPU caches take a large share of the chip area, they are very costly. As a result, most modern chip designs share the largest cache, the Last-Level-Cache (LLC), between multiple cores.

In this chapter we take steps towards quantifying the efficiency of NFV regarding packet copying overhead at hardware level. Additionally we aim to gather the potential that lies in reduction of the interference due to the shared LLC. We can show which factors influence the gain of LLC scheduling in NFV deployments. We propose a scheduler which optimally allocates the LLC in order to reduce the maximum CPU utilization of all VNFs.

The content of this chapter is partially based on an analysis of performance impacts of CPU memory architectures presented in [6]. This work was written in close collaboration between the first two authors. Nevertheless respective core contributions can be identified: The core contribution of C. Sieber in the work was towards a novel metric that can be used for quantifying the efficiency of a network function. This contribution is not used in this thesis. The core part of the author of this thesis was on the measurements regarding overheads in the NUMA architecture and especially overheads caused by cache exhaustion as presented in this chapter.

Furthermore the work on LLC interference and its management was presented first in [7].

[6] C. Sieber, R. Durner, M. Ehm, W. Kellerer, and P. Sharma. “Towards Optimal Adaptation of NFV Packet Processing to Modern CPU Memory Architectures.” In: *Proceedings of the 2Nd Workshop on Cloud-Assisted Networking (CAN)*. 2017, pp. 7–12. DOI: 10.1145/3155921.3158429.

[7] R. Durner, C. Sieber, and W. Kellerer. “Towards Reducing Last-Level-Cache Interference of Co-located Virtual Network Functions.” In: *28th International Conference on Computer Communication and Networks (ICCCN)*. 2019.

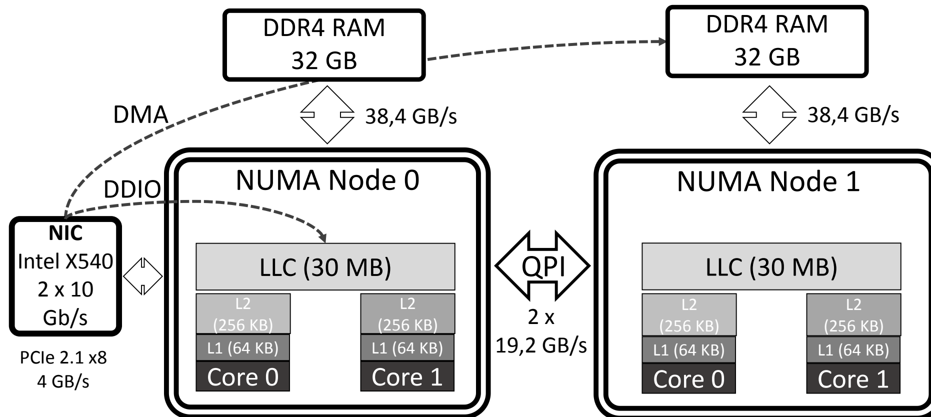


Figure 5.1: NUMA architecture with DDIO for the NIC-local NUMA node. DDIO allows direct access to the local LLC. DMA is used for packets destined to the remote node. Each Core has a L1 and L2 cache, the LLC is shared between all cores on one node. A Quick-Path Interconnect (QPI) bus with 2x19.2 GBps connects the nodes.

The structure of this chapter is as follows. First, in Section 5.1 we introduce the NUMA architecture and its cache hierarchy. Section 5.2 introduces related work on VNF performance using the NUMA Architecture and LLC interference management. Then we study performance impacts of the NUMA architecture to VNF performance. More specifically, we introduce a measurement methodology and show measurement results quantifying the overheads associated with different VNF placements. The results show that cache exhaustion and cache interference degrade the performance of VNFs. As a relief, we outline and evaluate the design of an optimal LLC scheduler for static VNFs in Section 5.4. Finally we summarize and discuss the findings of this chapter in Section 5.5.

5.1 Server Memory Architecture

The x86 architecture is the dominant CPU architecture in data centers. As NFV largely builds upon data center technologies such as compute and network virtualization, it can be expected that NFV environments will also consist of servers using such CPUs. In this market, Intel is the clear leader, thus in the following we concentrate on Intel's architecture and more specifically on the CPUs employed in our test-bed. The testbed consists of multiple Dell PowerEdge R530 servers with Intel Xeon E5-2650 v4 2.2GHz CPUs and Intel C610 chipsets. In the following we describe the memory architecture of x86 CPUs by example of this specific platform.

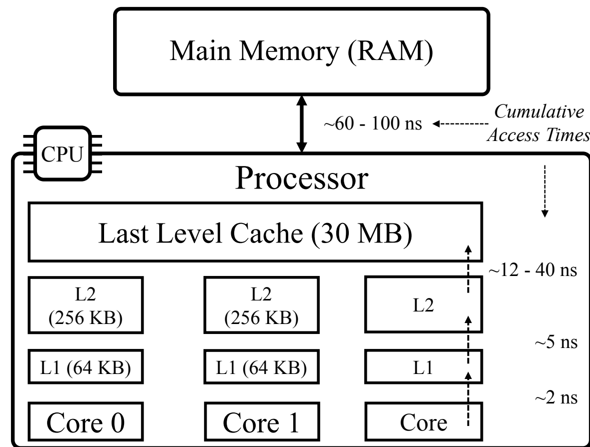


Figure 5.2: Simplified depiction of the Intel Xeon processor's cache hierarchy. Each core is equipped with an exclusive L1 cache (64 KB) and L2 cache (256 KB). A Last Level Cache (LLC) with a capacity of 30 MByte is shared. Access times are cumulative and range from roughly 2 ns if the memory access can be satisfied from the L1 cache to 100 ns if the data has to be fetched from the main memory.

5.1.1 NUMA Architecture

First we describe the NUMA architecture. As the fabrication of large chips is expensive, multi-processor systems are used for high end servers. Each socket houses one processor with multiple cores. The sockets are then connected using high speed bus connections.

Figure 5.1 illustrates the NUMA cache hierarchy and interconnection technologies of the aforementioned test-bed hardware. The servers have two CPUs called NUMA node connected via two bi-directional Intel Quick-Path Interconnect (QPI) lanes with a bandwidth of 19,2 Gbyte/s each. An Intel Ethernet X540 Network Interface Card (NIC) is connected to one NUMA node via PCI-express 2.1 x8. Each physical core is equipped with two exclusive cache levels, an L1 cache of 64 KByte and an L2 cache of 256 KByte. A LLC with a capacity of 30 Mbyte is shared among the CPU cores of a NUMA node. The caches are much faster than the system memory. Therefore if multiple VNFs share the LLC, the performance is degraded compared to the VNF running alone on the processor, due to interference effects. 32 GByte of DDR4 RAM capacity is attached to each node with a bandwidth of 38,4 GBytes/s. We disabled hyper threading on our server to eliminate another source of interference.

5.1.2 Cache Hierarchy

Figure 5.2 illustrates the cache hierarchy of the Intel Xeon processor and the connection to the RAM in more detail. Data from the main memory is accessed in chunks, denoted as *cache lines*, of 64 Bytes. When a CPU core accesses a particular memory location, the caches are checked incrementally starting from L1, through the shared LLC and up to the main memory.

The caches of our CPU are inclusive, i.e., every line that is cached in L1 is also cached in L2 and LLC. The shared LLC cache is 20-way associative, hence data of every memory location can be cached at 20 locations in the LLC cache and each cache way has a capacity of 1536 KBytes.

Intel Data Direct I/O (DDIO) allows NICs to copy packets straight to the LLC-cache of the NIC-local NUMA node, instead of doing the round-trip to the main memory and back to the LLC-cache when the processing application tries to read the packets. However, this is only possible for the NUMA node where the NIC is attached to, not for the remote NUMA node. For the remote node the packets are copied to the RAM using Direct Memory Access (DMA).

5.2 Related Work

This section presents related work performance aspects and improvements of security VNFs. The section is structured as follows: On the one hand related work focusing NUMA architecture is presented. Additionally we introduce work related to LLC interference in NFV environments.

5.2.1 NUMA Architecture

Some works that study the performance of NFV with focus on memory and data locality bottlenecks regarding the NUMA architecture already exist: Authors of [81] study the performance of Data Plane Development Kit (DPDK) in conjunction with single-root input/output virtualization (SR-IOV). SR-IOV is a pass through I/O technology which enables Virtual Machines (VMs) to access the NIC hardware directly. The work shows the sensibility of the performance to the NUMA placement, although the focus of the study is on the performance impact of the number of VMs.

Authors of [82] study the usage of DPDK for very high packet rates. Their results show that a data rate of 100Gb/s can be achieved using a single server. For multi threaded packet processing multi queue NICs are used. Authors evaluate the effects of different queue to core mapping strategies. They evaluate their system in terms of packet drop rate. Cerrato et. al [83] study the performance of VNFs using DPDK and different memory architectures. Their results show that a DPDK-based packet processing system with a high number (>100) of tiny network functions can deliver satisfactory throughput performance, although the experienced delay becomes high. Authors of [84] developed a NUMA aware thread scheduling approach, that reduces the slowdown caused by the NUMA architecture. The authors propose a performance slowdown index based on the inter-socket overhead caused by LLC cache misses.

Banerjee et. al [85] and Dobrescu et al. [86] show the performance impacts on VMs when comparing the VM on the same NUMA node as the NIC versus the remote node. Their results show a high number of cache misses when placing the VM on the remote node. They also provide an approach for determining a NUMA aware placement in virtualized environments. In [87], Kulkarni et al. introduce NFVnice, a framework for scheduling network functions on a server. They look at the problem of how computing resources can be allocated to VNFs using rate-cost proportional fair shares.

In contrast to the works above we concentrate on the performance bottlenecks in the CPU architecture and therefore do not use any virtualization techniques to avoid side effects. In addition we introduce a novel algorithm to quantify the load on a VNF in polling mode.

5.2.2 LLC Interference Management

LLC interference in NFV environments was not widely studied yet, nevertheless there are some works which study LLC interference and aim to reduce LLC interference [88–91]. NFV also emerged from cloud compute concepts, but LLC interference scheduling is studied more in depth with respect to compute cloud environments.

A number of works considers LLC contention effects in compute cloud environments [92–97], i.e. not considering NFV. One main difference is the performance metric employed: this is commonly the completion time of a program. NFVs are different in nature as they are event based, an incoming packet is an event that has to be processed. Consequently a VNF program never completes its task and no completion time metric exists. Though some works can also give indications to the problem we are addressing here and were helpful during our work.

[93] studies performance degradation due to LLC interference. It is shown that programs that have many LLC references degrade stronger while other programs that are more compute intensive are less affected. We cover this diversity as we emulate VNFs with a larger and a smaller working set and show that the same findings are also true in NFV environments.

Heracles [96] reduces contention between batch tasks and event based tasks in order to improve the utilization of shared server resources. Heracles considers different resources such as network bandwidth, memory bandwidth CPU cores and also the LLC. The approach uses Intel Cache Allocation Technology (CAT) in order to isolate between the two types of tasks but does not consider interference between tasks of the same type. In contrast to that we study contention between multiple latency sensitive tasks, in our case VNF.

Another work [92] aims to build a fair LLC scheduler. Fairness is defined such that the performance degradation due to LLC interference is equal for all programs. Authors show that

the degradation is higher for some programs than for others and that the developed scheduler can avoid this effect. Compared to our work there are two main differences: On the one hand a different optimization objective is chosen and, on the other hand, authors focus on compute workloads rather than VNF.

PACMan [97] places VMs on different servers such that the interference between the programs is reduced. The approach first profiles the VMs and then consolidates the VMs on different servers. The approach doesn't consider a controllable LLC like we do and does not optimize online, but uses the VM's profile to optimize the VM placement. As the profile of VNFs strongly depends on the traffic profiling is more difficult in an NFV environment. Nevertheless VNF profiling an placement/LLC optimization would be an interesting extension to our work, which we also consider to study more in deep in the future.

Recently authors of [88] studied the interference effects of co-located VNFs in depth. They consider contention of network I/O bandwidth, CPU, memory and cache. With there measurements they can show that different types of VNF cause different effects. A VNF that only read packets but does not modify them, like a gateway, has a different patterns than a VNF that modifies the packet, e.g. a load balancer. This is in line with our results that show that the cache interference effects depend on the memory access patterns.

A different possibility than CAT that prevents cache interference is page coloring. Authors of [89] study cache coloring to avoid cache interference in an NFV environment. Page coloring enables the separation of the cache in different cache regions as pages with different colors do not contend for the same cache ways. The authors cluster different memory buffer pages to different colors. Different clustering methods are applied: Element-based clustering, groups functions of the same type together, while flow based buffering uses the same page colors for groups of flows. We are studying a different approach that does not cluster but schedules individual VNFs independently and without any knowledge about the VNF implementation.

Veitch et. al [90] showed that Intel CAT can improve the performance of VNFs when a noisy neighbor is present. A noisy neighbor is a VNF or program that evicts cache lines regularly and therefore causes high interference and performance degradation to other VNF co-located on the CPU. The authors study different static CAT configurations and show that the latency of VNFs can be decreased with CAT. In contrast to our work they aim to show the benefits that CAT can give in an NFV environment rather than aiming for an optimal allocation. Nevertheless this work was an important starting point for our work.

The work of Dobrescu et. al [91] studies interference effects of software packet processing systems (i.e. VNFs). The authors show that there exist different types of VNFs that use the cache in different manners. Some VNF types have only a low amount of LLC accesses while others have a high number of accesses. It is shown that these types impact the performance

degradation when they are co-located on one CPU. This strongly supports our findings from Section 5.4.3. On the other hand the authors do not study LLC scheduling, but suggest an orchestration that places the VNF on different servers and CPUs such that the interference is reduced.

ResQ [98] proposes to use CAT in order to enforce performance guarantees. The authors show that CAT can be successfully used to enforce throughput and latency guarantees. A 2-step offline approach with pre-profiled network functions is used. First the network functions have to be profiled using a variety of traffic profiles. In the second step the profiles can be used to improve the placement of the network functions and the allocation of the LLC.

NFV environments that are the target of this work are quite dynamic systems. State of the art has shown that the performance and LLC interference depends on the type of the VNF, the packet rate but also the traffic mix. Thus a LLC scheduler that can work without a priori knowledge of the VNFs and the environment is desirable. Nevertheless none of the above works shows such an scheduler, which we propose in the following. Before doing that we analyze the performance impact of the NUMA architecture to VNF performance.

5.3 Impacts of the NUMA architecture to VNF performance

In the following we evaluate the influence of the memory architecture on the packet receiving performance. We evaluate the effects of the memory architecture to the packet processing performance. The results show that copying packets between the NUMA nodes increases the CPU load drastically and should be avoided if possible. The measurements using an increasing Access Control List (ACL) size in memory show that the effects of CPU cache exhaustion should be considered when designing VNFs. Additionally, we show that the CPU cycles needed for memory access follow the Average Memory Access Time (AMAT) model. Overall performance penalty of copying between NUMA nodes is bigger than accessing the memory at the local socket.

5.3.1 Methodology

In this section we discuss the measurement setup, the implementation of the VNFs and how the CPU load is measured.

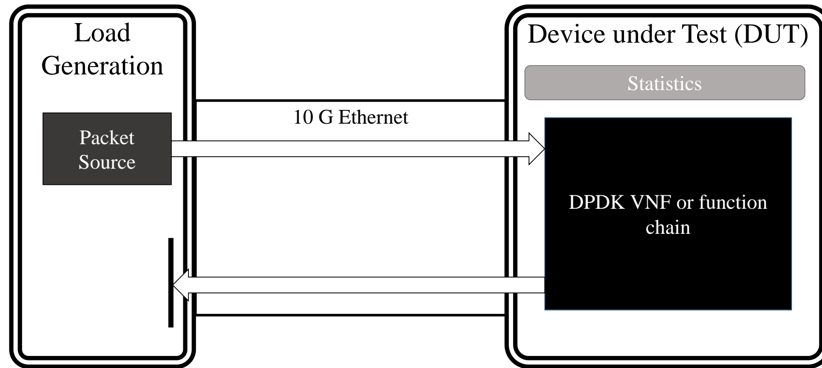


Figure 5.3: Test environment: A PC running a load generator is connected with the DUT, a x86 server. Both devices are connected using one 10G Ethernet link.

5.3.1.1 Scenario and Test Environment

The test environment is shown in Figure 5.3. It consists of the load generation PC which runs DPDK-Pktgen Application and the Device-under-test (DUT), which is one of the servers described in Section 5.1.1 in detail. The PC generates packets with a limit of 2 Mpps and sends it to the DUT with a constant rate. At the DUT the packets are processed from one of the VNFs described in Section 5.3.1 and sent back to the PC. Fast packet processing frameworks address this and other issues to provide high packet rates on commodity servers. We utilize virtual network functions realized with the DPDK [99]. The DPDK is a set of libraries for Linux to facilitate fast packet-processing on common server hardware. It was first introduced by Intel in 2013 and since then has become a Linux Foundation Project with broadening vendor-support.

As we concentrate on CPU metrics in this work, no packets are received by the load generation machine, i.e. they are dropped by the NIC. Our study concentrates on CPU performance metrics, like core utilization and cache hit rates. In consequence, all statistics are gathered directly on the DUT. For all memory and cache related measurements we are using Processor Counter Monitor Tool.

5.3.1.2 Minimal VNF

First we consider a minimal network function that only receives packets and does not do any processing or forwarding. Using this implementation we can show the overhead of the NUMA communication for packet processing without any side effects. The minimal VNF drops all packets after receiving.

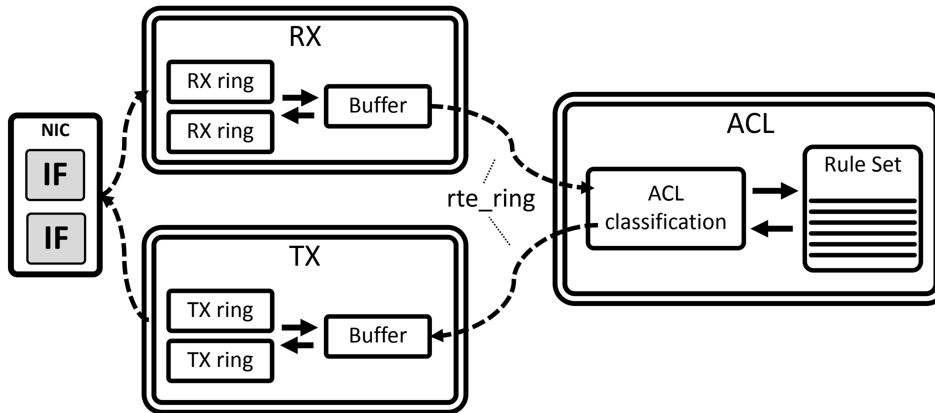


Figure 5.4: The function chain consisting of one RX, one ACL and one TX process. Each process runs exclusively on one core. The RX and TX processes move the packets between the respective buffers and rings. The ACL process checks each packets against the its rule set.

5.3.1.3 Function Chain Implementation

The chain implementation consists of three separate entities as illustrated by Figure 5.4. Our results are based on the use case of a virtualized firewall. A receiving, a sending and a packet classification process, each running exclusively on one CPU core following the DPDK best-practices. Two receiving rings, one for each physical NIC port, are connected to a software switch, which writes the packet pointers to a buffer. The interconnection between the three entities is implemented via a rte_string data structure. A rte_ring is a lock-less, fixed-size queue implementation provided by the DPDK. The ACL classification core matches the received packets against the loaded ACL rules and decides to either forward or drop the packets. The sending core moves the packets to be forwarded to the TX rings of the physical NIC ports.

5.3.1.4 Measuring CPU Load with Polling

One of the techniques that DPDK uses to increase the packet throughput is the change from an interrupt-based packet retrieval to a polling-based packet retrieval. Usually if a packet arrives on the NIC, the CPU is interrupted and the packet is then copied and processed by the kernel of the operating system. DPDK does not use interrupts, instead it checks for packets at the NIC, processes these packets and then checks again for packets in an infinite loop. As a consequence conventional CPU utilization tools do not work as the CPU is always fully utilized by the loop. Because of that we developed an algorithm shown in Algorithm 3 to evaluate the current CPU load.

The main idea is to rely on the cycles reported by the CPU, as they are available without much overhead. During the main processing loop, the cycles are read before and after the

Algorithm 3: Measure CPU Utilization

```
1  OPS, REF = 0
2  cyc_last ← read CPU cycles from register
3  while True do
4  |  cyc_before ← read CPU cycles from register
5  |  REF + = cyc_before - cyc_last
6  |  cyc_last ← cyc_before
7  |  if packets received then
8  |  |  process packets
9  |  |  cyc_processed ← read CPU cycles from register
10 |  |  OPS + = cyc_processed- cyc_before
11 |  end
12 end
```

packet processing to determine the cycles needed for processing, i.e. the OPS. Additionally, the cycles needed for each loop are measured as reference counter REF. Both counters are reported regularly to the monitor. The CPU utilization can then be computed with the following formula:

$$CPU\ Utilization = \frac{OPS}{REF}$$

Using this definition of CPU utilization, a utilization of 100 % is reached with the maximum possible packet rate. As can be observed in the results plotted in Figure 3, the utilization is roughly linearly dependent on the packet rate.

5.3.2 Evaluation

In this section we evaluate the impacts of modern x86 processor architectures on NFV performance. In the first and second part we evaluate the impact of the NUMA architecture with the minimal VNF and the function chain. The third part evaluates the performance degradation when using VNFs with a large working set in the memory, i.e. when the caches of the processor are exhausted.

5.3.2.1 NUMA Impact - Minimal VNF

In the first scenario, the receiver is placed on the same node as the NIC is attached to, denoted as node 0 in the following. In the second scenario, the receiver is placed on the remote NUMA node, in the following denoted as node 1. Figure 5.5 illustrates the receiver's core utilization for packet rates from 0 to 1 million packets per second for both scenarios. Confidence intervals are not visible as the measurements showed little variation.

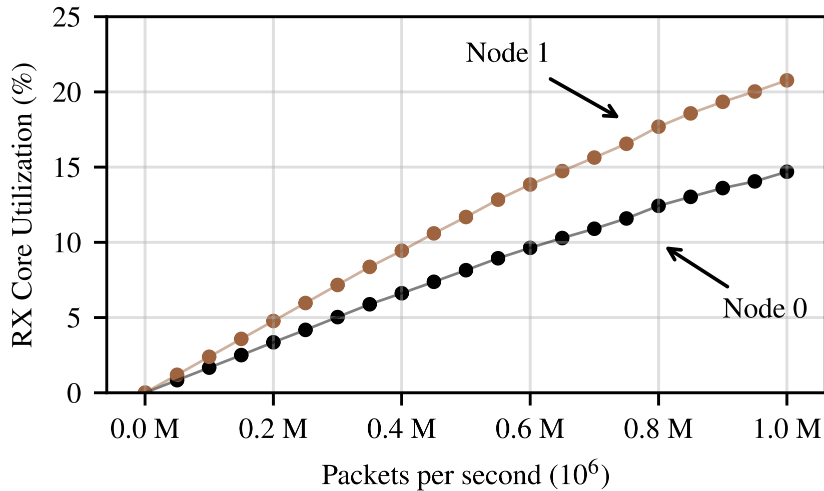


Figure 5.5: Simple Receiver: Utilization of the core for increasing packet rates and two different placements (node 0 and node 1). The nic is attached to node 0. A 41 % increase in utilization is observed when the receiver is executed on node 1 (20.8 % utilization compared to 14.7 %), due to the overhead in fetching packets from the nic through the QPI.

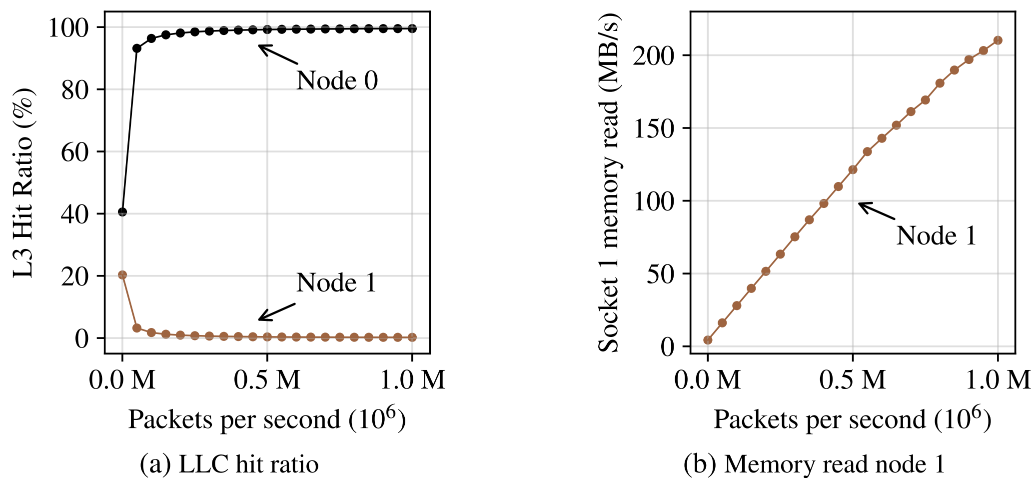


Figure 5.6: LLC cache hit ratio and memory read rate on node 1 for two placements of the minimal VNF for increasing packet rates up to 1 million packets per second. The figure confirms that a placement on the remote NUMA node without DDIO support results in cache misses and high main memory access rates.

The figure shows that the core utilization increases by 41 % (14.7 % utilization compared to 20.8 %) when the NIC is attached to a different NUMA node than the RX process. This is due to overhead which is required for transferring the packets first to the remote memory and afterwards to the processor cache. In case the NIC is attached to the same node, DDIO allows a direct transfer to the processor cache as shown in Figure 5.1.

Subsequently we take a look at the LLC hit ratio and memory read throughput for the two placements to confirm the source of the bottleneck described above. Figure 5.6 shows the hit ratio (5.6a) and the memory throughput (5.6b) for packet rates up to 1 million packets per

second. For packet rates close to zero (200 packets per second) we observe a LLC hit ratio of 40 % for the node 0 placement and 20 % for the node 1 placement. For higher packet rates the hit ratio increases rapidly to about 100 % for the node 0 placement and 0 % for the node 1 placement. The unexpected hit ratios for low packet rates are due to the influence of the underlying operating system and measurement scripts. That influence diminishes with higher packet rates. Figure 5.6b gives the memory read throughput for the node 1 placement. The figure shows that the throughput increases linearly with the packet rate. The results confirm the previous statement that DDIO allows a direct transfer of the packets from the NIC to the LLC cache of the processor, which results in a 100 % cache hit ratio. In case of the placement on node 1, the packets are transferred via DMA to the main memory of the node first. The access by the receiver results in a cache miss for every packet and therefore the memory read throughput increases.

5.3.2.2 NUMA Impact - Function Chain

The influence of the NUMA placement on the performance of the function chain is described in the following. We place a chain of 3 elements on two possible locations, which results in 9 possible placements. The placements are denoted with RX-ACL-TX on the two NUMA nodes 0 and 1. E.g. 0-1-0 indicates a placement where the RX function is put on NUMA node 0, the ACL function on NUMA node 1 and the TX function is placed on NUMA node 0. The NIC is connected again to node 0. With placement 0-0-0 no packet copying between the NUMA nodes is necessary, therefore this placement is expected to be the best case. On the other hand, placement 0-1-0 is expected to be the worst case regarding the ACL utilization, as the packets first have to be copied to NUMA node 1 for ACL classification and then back to node 0 for transmitting.

Figure 5.7 illustrates the utilization of the ACL core for the four placements (0-0-0, 0-1-0, 1-0-1 and 1-1-1) of the function chain. The figure shows a linear increase of the core's utilization for increasing packet rates up to 2 million packets per second. For 2 Mpps, the measurements show an utilization of about 29 % for the worst case 0-1-0 and 17 % for the best case 0-0-0. Hence, there is a penalty of roughly 73 % regarding the CPU load between best case and worst case placement This means that NUMA-level copying caused by a non-optimized placement has severe performance impacts.

5.3.2.3 Impact of Cache Exhaustion

Next, we discuss the performance impact of cache exhaustion on the ability of a core to process packets. For this we keep the packet rate constant at 2 Mpps and increase the size of the ACL.

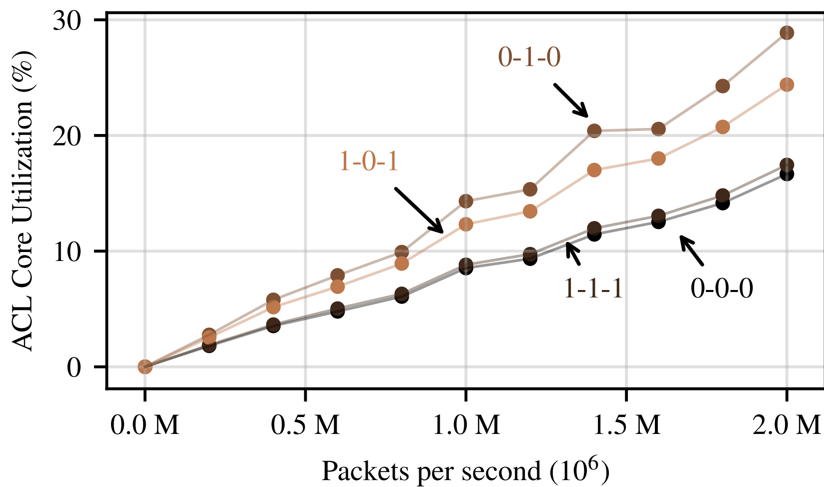


Figure 5.7: ACL core utilization for four different placements of the firewall function chain (RX-ACL-TX) depending on packet rates. Placement 0-0-0 represents the best case placement without remote NUMA access and 0-1-0 the worst case where remote NUMA memory has to be accessed. The results shows an increase in utilization for the worst case placement of roughly 73 % compared to the best case.

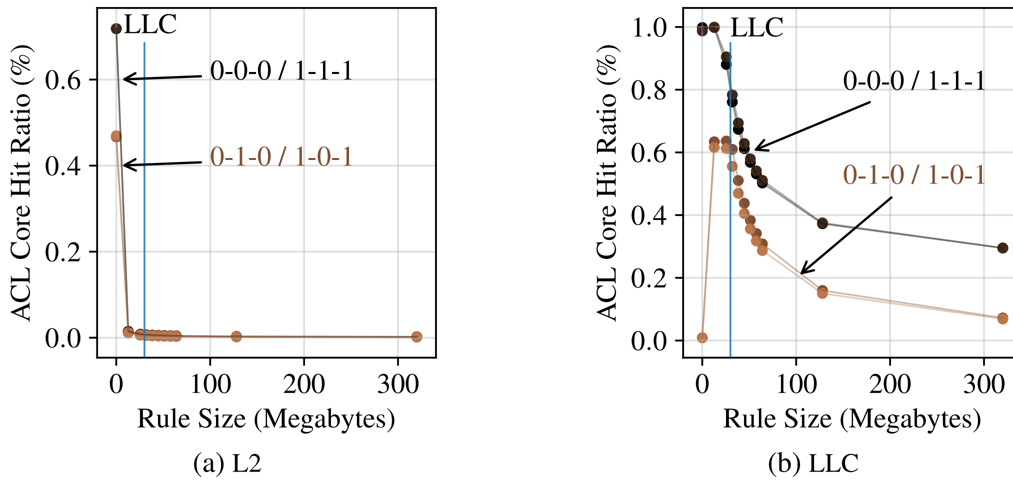
We evaluate ACL sizes from 64 KB to 320 MB. For an LLC size of 30 MB, an ACL size of 320 MB results in a 10.6 times over-subscribed LLC.

Core utilization of the ACL function is caused by packet processing (copying packets, accessing headers, etc.) and additionally by accesses to the memory for the ACL. The performance penalty for accessing data in the memory, largely depends on the locality of the data, i.e. if it is in a L1, L2 or LLC or if it is in RAM, as the processing core has to wait for the data [100].

Figure 5.8 illustrates the ACL core cache hit ratios of the L2 and LLC depending on the ACL size for the four different placements 0-0-0, 0-1-0, 1-0-1 and 1-1-1. As expected, the hit ratio of the small L2 cache falls fast to about 0 % for all four placements. As the core does not access the packet before the classification, the packet can not be available in the L2 cache. Furthermore, the chance that a specific ACL rule was accessed before decreases fast as the quotient between L2 size and ACL size gets very small.

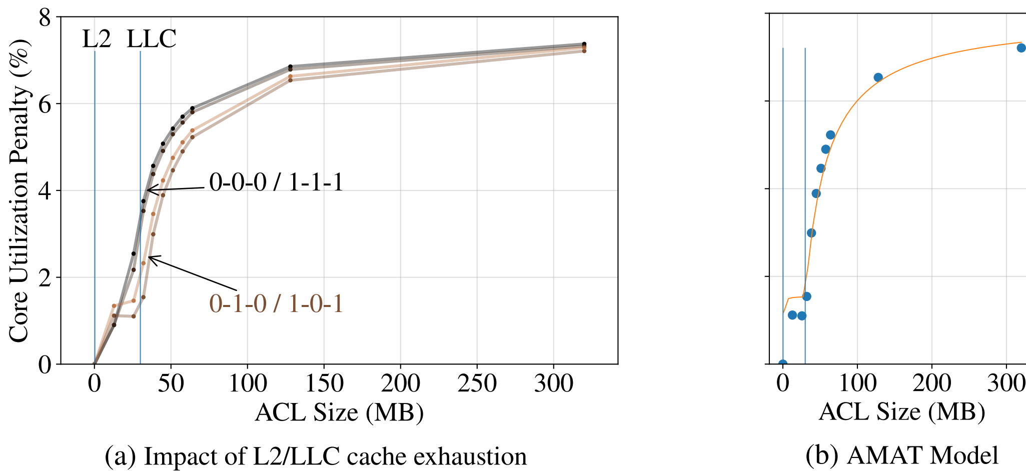
For the LLC, we measure a hit ratio of 7 % for the worst case placements 0-1-0 and 0-1-0 where the packet is not yet in the LLC. For the best case placements 0-0-0 and 1-1-1, where the packet is already available in the LLC due to the TX process being placed on the same NUMA node, we measure a hit ratio of 30 %.

Figure 5.9a illustrates the ACL core utilization penalty for increasing ACL size for the four placements of the RX, TX and ACL cores. The two horizontal lines mark the capacities of the L2 and LLC. The Figure shows that for the 0-0-0 placement, the utilization increases about linearly for an ACL size between the L2 and LLC capacity. For the 0-1-0 and 1-0-1



(a) L2 (b) LLC

Figure 5.8: ACL core cache hit ratios of the L2 and LLC depending on the ACL size for the four different placements. As expected, the hit ratio of the small L2 cache falls fast to about 0 % for all four placements. For the LLC, we measure a hit ratio of 7 % for the worst case placements 0-1-0 and 0-1-0, where the packet is not yet in the LLC. For the best case placements 0-0-0 and 1-1-1, where the packet is already available in the LLC, we measure a hit ratio of of 30 %.



(a) Impact of L2/LLC cache exhaustion (b) AMAT Model

Figure 5.9: Impact of memory accesses on the core utilization of the ACL core. Increasing ACL size for a constant packet rate. L2 cache capacity is 256 KB and LLC cache capacity is 30 MB. The Figure shows the additional utilization caused by memory access. The additional utilization follows for the placements 1-0-1 and 0-1-0 roughly the trend of a simple AMAT model (right figure). The dots in the right figure are the measurement points of the 0-1-0 placement.

placement, the penalty first jumps from 0 % to 1.2 % and subsequently stagnates until the LLC is exhausted. The stagnation is due to the fact that the 0-1-0 and 1-0-1 placements allow the ACL core to use the NUMA 1 node's LLC exclusively. Therefor, the impact of the LLC exhaustion is only visible for larger ACL sizes compared to the 0-0-0 and 1-1-1 placements where the LLC is shared between the RX, TX and ACL cores. After the LLC cache is exhausted, the utilization increase depends, in addition to the ACL size, on the cost in terms of time for accessing the memory hierarchy and on the cache hit ratios.

The behavior follows roughly the recursive AMAT model from [101]. Equation (5.1) and Equation (5.2) define the AMAT model. H_x denotes the hit time, the time the processor needs to access the data on cache level x, if it is available in the cache. MR_x denotes the miss rate. We assume in our model that one ACL rule can be fetched by one cache access and that every packet triggers the same amount of cache accesses uniformly. Hence, we define MR_x as the chance of missing the cache based on the size of the cache S_x and size of ACL rule set S_{ACL}: $MR_x = max(0, 1 - \frac{S_x}{S_{ACL}})$.

$$AMAT = H_{L1} + MR_{L1} \cdot AMP_{L1} \tag{5.1}$$

with AMP_x as average miss penalty of cache level x:

$$AMP_x = H_{x+1} + MR_{x+1} \cdot AMP_{x+1} \tag{5.2}$$

The CPU load L_{CPU} for accessing data depending on the packet rate R_P can then be computed with Equation (5.3), where f_{CPU} is the CPU frequency.

$$L_{CPU} = AMAT \cdot \frac{R_P}{f_{CPU}} \tag{5.3}$$

Due to performance optimizations and pipelining in modern CPUs, the timings can not be named easily, we fitted the the cache timings to the measuring results using least squares method. Figure 5.9b illustrates the memory access penalty as a function of the ACL size S_{ACL} according to the AMAT model. The dots show the measurements with placement 0-1-0, the line denotes the model. From the figure follows, that the core utilization penalty in general follows the AMAT model, a small offset is visible for smaller ACL sizes $< 30\ MB$.

To summarize the findings, we can see that the additional CPU utilization caused by cache exhaustion is clearly visible but smaller than the NUMA penalty (roughly 15 % vs 7 %).

5.4 NFV Last Level Cache Scheduler

In the last section we saw that VNF is sensitive to the performance of the CPU caches. Within the NFV concept resources are virtualized and multiple VNFs are sharing one server. This pattern causes interference between the consolidated VNFs at different places in the shared system. In this section we concentrate on one specific interference effect caused by the co-location of VNFs on one single CPU chip: the LLC interference. In modern multi-core processors, some of the on-chip resources such as the LLC are shared between all cores, which causes interference. To resolve this issue some chip manufacturers like Intel [102] or Qualcomm [103] are providing means to explicitly allocate shares of the LLC to specific cores and processes.

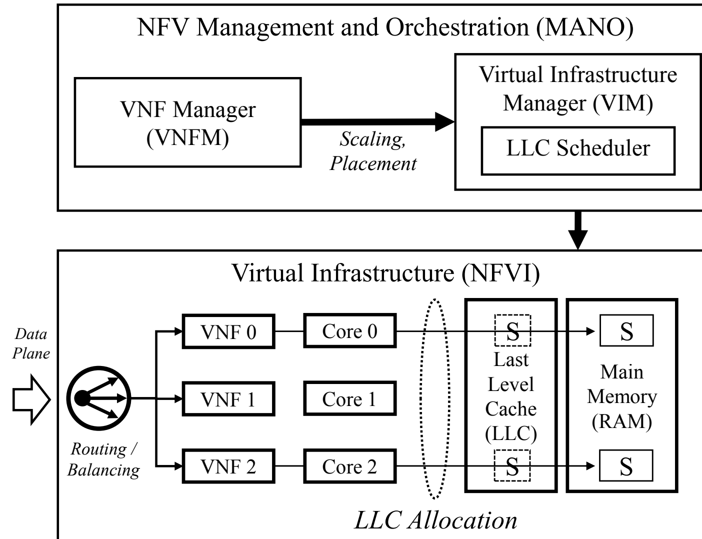


Figure 5.10: LLC scheduling in the bigger picture of NFV Management and Orchestration (MANO). Horizontal scaling of VNF instances for a specific service, e.g., a firewall, is performed by the VNF manager through the VIM. Through the routing/load-balancing functionality of the network, the VNFM dictates which fraction of the traffic is assigned to a specific VNF. Network elements then forward and distribute traffic to the running VNFs. The VNFs have to share the available LLC and only part of the working set S of each VNF can be kept in the LLC.

We first describes the role of LLC scheduling in the bigger picture of NFV. Afterwards, a brief description of the Intel Cache Allocation Technology and the cache monitoring mechanisms provided by Intel is given. Then we give the memory model that we used to derive our optimal scheduler. Finally we show the gains that can be achieved with the introduced scheduler.

5.4.1 NFV MANO

Figure 5.10 depicts the LLC scheduler in the bigger picture of NFV Management and Orchestration (MANO). The architecture has been proposed by the ETSI NFV working group [104]. The architecture describes the components required in all stages of the life-cycle of a VNF, from the definition in terms of deployment and operational requirements, to the allocation and the release of the required resources. The proposed LLC scheduler can be implemented as part of the Virtualized Infrastructure Manager (VIM), the MANO component which manages the available resources of the physical infrastructure. With the presented scheduler, the VIM can optimize the distribution of the LLC to the active VNFs on the physical server. The amount of memory that is required from the VNF to fulfill its functionality is called working set. The working set consists of the binary and as well the state of the VNF such as tables, rules etc.

Commonly, a CPU core is assigned exclusively to a VNF to benefit most from processor register, L1 and L2 caching [105]. A load balancer on the data-plane, either in the network or in software on the host, distributes the network packets to the available instances. The goal of such packet load balancers is to keep the utilization homogeneous between the instances. Down and up-scaling of VNFs is done by the Virtualized Network Function Manager (VNFM) by stopping or starting additional VNF instances [106] in order to improve resource utilization. For example the VNFM may add a new instance at an average utilization of 90 % for all instances and remove an instance if the average utilization of all cores drops below 60 %.

5.4.2 Cache Allocation Technology

The Intel CAT [102] enables the allocation of the LLC to specific CPU cores. The allocation can be done shared, i.e., multiple cores share specific parts of the cache, or exclusively, i.e., parts of the cache are allocated to specific cores. In detail, CAT introduces 16 Classes of Service (COS) for CPU cores. Each core has to be assigned exactly to one class, but multiple cores can be assigned to the same class. The LLC is organized in cache ways, each cache way has a size of 1536 KByte. A bit-mask per class configures which of the available 20 cache ways can be used by which COS. In a nutshell, CAT enables the allocation of 20 LLC chunks to specific CPU cores. Due to limitations of the technology CAT only restricts write accesses to the LLC. This means that a core that had access to a larger share of the cache before a reallocation can still access cached data stored in ways that are allocated to a different COS. This restriction causes an transient behavior of the cache after CAT changes. A detailed evaluation of the transient phase of the LLC is done in Section 5.4.5.3.

5.4.3 Memory Access Model

The sensitivity of VNFs to LLC contention, and also the interference caused by a VNF, depends on a number of factors, such as the size of the accessed memory range and also how often each memory location is accessed. In this section, we introduce a memory model that serves us for the derivation of the scheduler algorithm and which led us in the design of the memory access emulator.

A VNF, or any program, accesses different data with different frequencies, e.g., parts of the binary are accessed often while some other data might only be accessed for startup. Theoretically, we split the complete allocated memory of a VNF in chunks, e.g., each chunk is 64 Bytes large as in the cache lines. With this abstraction we can assign every chunk of memory a distinct access frequency and we can sort the chunks in decreasing order by the access frequency. In this work we denote this as *access pattern*.

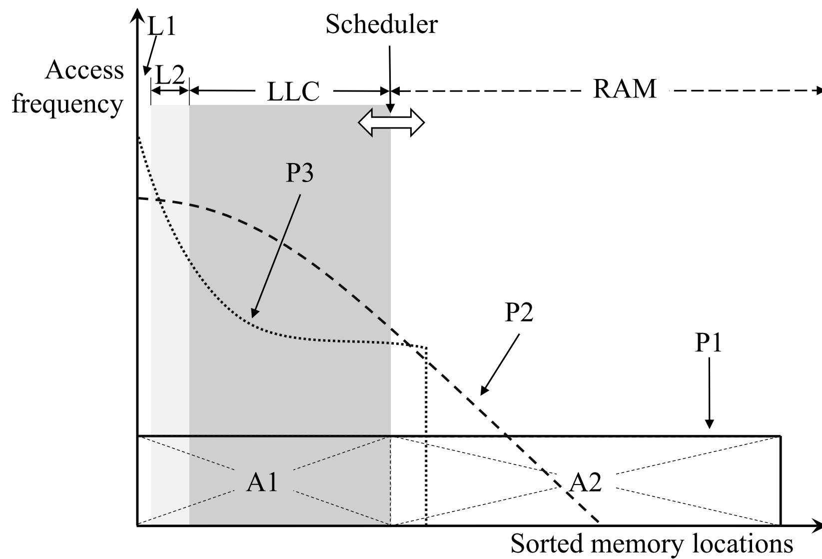


Figure 5.11: Possible memory access patterns.

Figure 5.11 abstractly visualizes three different possible access patterns (P1, P2, P3). The x-axis represents the different memory chunks and the y-axis of the plot represent the corresponding access frequency. Pattern P1 represents a program that accesses all its memory chunks with the same frequency. This is not realistic, but resembles the behavior of a memory stressing benchmark. Pattern P2 is more fitting to VNFs: some chunks are accessed frequently while others are only accessed seldom. It is known that network traffic commonly has elephant flows and mice flows. If we imagine a router with a routing table in memory, the entries that are matched by elephant flows are accessed more often than mice flow packets. This could cause an access pattern like P2. We do not want to restrict ourselves to specific patterns, as there might also be patterns like P3 that do not have smooth transitions, but rather a step at some point.

In the figure we also sketched how this relates to the caches. As the caches mainly work in a Least-recently-used (LRU) manner, this pattern translates to hit rates of the caches. The hit rate of a cache measures what ratio of the accesses to the cache were served from the cache. Thus the hit rate of the LLC is proportional to the following formula:

$$LLC\ Hitrate \sim \frac{Area\ within\ Cache}{Area\ not\ within\ Cache}$$

In the figure we marked the area within cache as A1 and the area not within cache as A2 for pattern P1. With CAT we can influence how much LLC each VNF can use. In the depicted case the cache is more useful to the VNFs with P2 and P3 than the VNF with P1 as they have a higher access frequency in this region. Further, as the LLC is limited, not all of the data accessed by the VNF might fit even in the complete LLC. As a consequence, a hit

rate of 100% is only possible if and only if the size of the memory that is accessed by the VNF is smaller than the complete LLC. This memory is often referred as *working set*, meaning the set of data that is used by the VNF. The working set can be equal to the allocated memory, but it can also be smaller, e.g. if some data is only used for the initialization of the VNF. Note that the optimal allocation also depends on the optimization objective of the scheduler and thus cannot only be derived from the access pattern. Further, the access pattern depends also on the network condition, e.g., an Intrusion Detection System (IDS) serving highly diverse traffic has a different access pattern than an IDS that only filters a single connection [88].

The CPU utilization is the share of the CPU cycles where the CPU is active, i.e., not in a sleep state. Besides actual processing cycles the CPU is also active while waiting for data. Consequently, the CPU utilization increases for the same number of executed instructions if the $LLC\ Hitrate$ decreases.

In order to emulate different possible behaviors of VNFs, we propose simplified memory access patterns which can be described with the allocated memory M, the maximum access frequency R and a distribution parameter α. The access pattern can be described with the function $r(m)$, which is the access rate at memory location m:

$$r(m) = R \cdot (\alpha + 1) \cdot (1 - m/M)^{\alpha}\ with\ m \in [0, M] \tag{5.4}$$

The parameter R models the packet rate, as the memory access rate of VNFs is proportional to the packet rate. M is the working set in both the emulation and for real VNFs. Finally parameter α describes a probability distribution that allows us to emulate a range of different access patterns. By setting $R = 1$ and $M = 1$ in $r(m)$, we get the underlying probability distribution, with the Propability Density Function (PDF) $f(x)$ and the Cumulative Distribution Function (CDF) $F(x)$:

$$f(x) = (\alpha + 1) \cdot (1 - x)^{\alpha} \tag{5.5}$$

$$F(x) = 1 - (1 - x)^{\alpha+1} \tag{5.6}$$

Figure 5.12 shows the PDF and CDF for different parameters α. If we chose for example $\alpha = 0$, the distribution becomes uniform and we get an uniform access pattern with constant rate R for all memory positions $m \in [0, M]$. This kind of access pattern is also sketched in Figure 5.11 as P1 (solid line). For other values of the parameters R, M and α, we obtain different access patterns.

We only considered a quasi-static set of VNFs such that the access pattern $r(m)$ is not time dependent. This means that the VNFs and the traffic pattern of the VNFs does not change over time. This enables us to study the potential gains of LLC scheduling in deep.

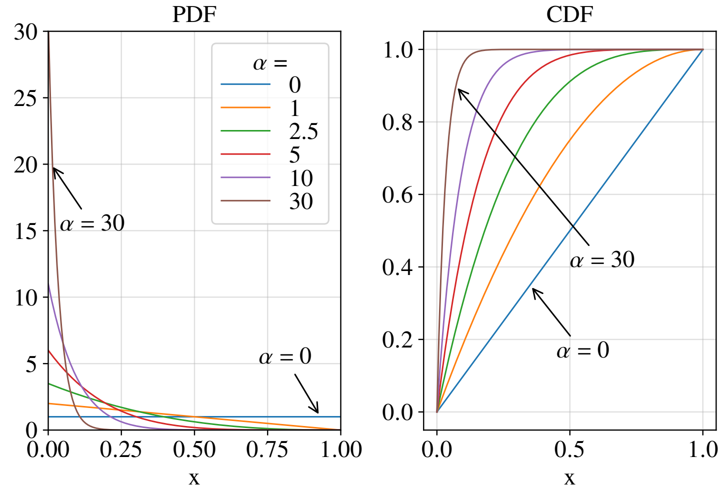


Figure 5.12: CDF and PDF of the developed memory access distribution.

5.4.4 Optimal Scheduler Design

In this section we first describe the developed algorithm which determines the optimal allocation for the static case.

5.4.4.1 Optimization objective

There are different optimization objectives possible. We imagine to use the scheduler in an NFV environment enabling scaling as described in Section 5.4.1. In this environment the NFV orchestration would scale up, i.e., launch new instances, if some threshold is exceeded. As a result it makes sense to reduce the CPU utilization and thus make the scaling unnecessary, consequently saving resources. Hence, we choose to minimize the maximum CPU utilization of all cores. But the objective could also be, e.g., to minimize the sum of utilizations of all cores, or the reduction of memory access delays for selected VNFs. Furthermore, with a CPU utilization of 100%, packet loss is caused that should be avoided. Summarizing, an optimization towards the reduction of the maximum CPU utilization of all cores is desirable.

5.4.4.2 Algorithm

Algorithm 4 defines the algorithm that finds the LLC allocation which minimizes the maximum CPU utilization of all cores ($Allocation^*$). The system is initialized with the default allocation, i.e. all cores compete for the LLC. W_c denotes the number of exclusively assigned cache ways of core c. No core has exclusive cache ways in the beginning (line 2). Before starting the algorithm loop, the upper bound for the minimal maximum CPU utilization of all cores Z is 1, as this is the maximum value of utilization. The algorithm also remembers the current allocation to cover cases where the initial allocation is already the optimal allocation (line

Algorithm 4: Min-max scheduler determining the LLC allocation that minimizes the maximum CPU utilization

```
1  initialize: all cores share the LLC;
2  W_c = 0 ∀c;
3  Z = 1;
4  Allocation* ← current allocation;
5  while ∑_c W_c < LLC_tot do
6      c* = arg max_c(U_c);
7      if W_c* != 0 then
8          W_c* ← W_c* + 1;
9      end
10     else
11         W_c* ← ⌊LLC_c* / 1.5MByte⌋
12     end
13     if max(U_c) < Z then
14         Z = max(U_c);
15         Allocation* ← current allocation;
16     end
17 end
```

4). The following steps are repeated until all possible cache ways are distributed. First the core that has currently the highest CPU utilization is determined, this core is called c^*. If this core was already scheduled before, i.e., there are already cache ways exclusively assigned, the number of exclusive cache ways is increased by 1. Otherwise it is measured how much LLC the core currently uses and this value is used to determine an initial number of ways. Each cache way corresponds to 1.5 MB of LLC, as the number of ways has to be integer the algorithm applies the floor operation. In most cases the LLC for the initialized core is increased in a subsequent step. The floor operation is a conservative choice in this case, as it guarantees that not too many cache ways are allocated.

After each schedule update, the algorithm checks if the step resulted in a new upper bound Z. In this case, the new bound and the new allocation are saved (lines 14 & 15).

The algorithm assigns at least one cache way in each step until all cache ways are assigned. In our set-up the LLC is 20-way associative, at minimum 2 ways must be left for cores without exclusive ways. This results in a maximum of 18 steps for the algorithm to find the optimal allocation. In general the algorithm is of linear complexity with the number of cache ways.

5.4.4.3 Example Run

Figure 5.13 shows one example run of the scheduler as presented in Algorithm 4. The upper graph shows the CPU utilization of the used cores, the lower one shows the LLC occupation

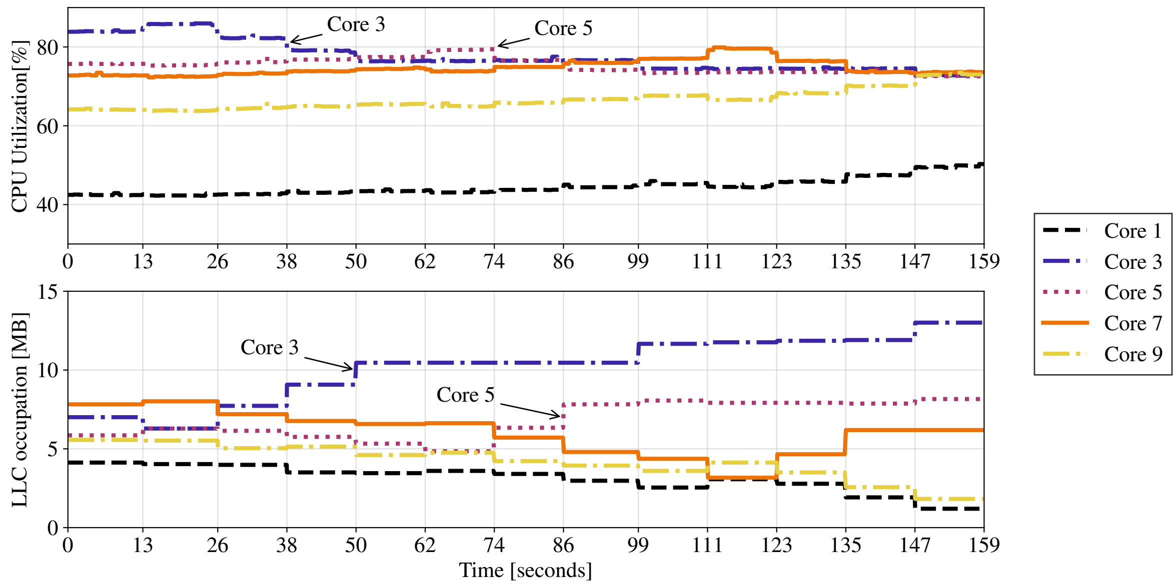


Figure 5.13: CPU utilization and LLC occupation of an exemplary scheduler run with 5 cores. The duration of each step is 12 seconds.

of each core. The LLC Occupation metric measures how much LLC each core is currently using.

At time $t = 0\,s$ the system is initialized with the LLC shared by all cores.Before doing anything the scheduler determines the CPU utilization of all cores using PCM as described in section 5.4.5.2.

In the first LLC allocation update at $t = 13s$, the highest core, in this case Core 3, is assigned exclusive ways. The number of ways for this first allocation is computed as in line 11 of Algorithm 4. As Core 3 can use less LLC than before, the CPU utilization of Core 3 is increased after this step. After this update, the scheduler waits until the CPU utilization has stabilized and measures again the CPU utilization of the Cores.

The next schedule update is done at $t = 26$ and increases the LLC of Core 3 by one way. After further updates at $t = 38$ and $t = 59$, the CPU utilization of Core 3 is below the utilization of Core 5, thus Core 5 is now scheduled in the updates at $t = 62$, $t = 74$ and $t = 86$.

This scheme continues until the optimum allocation is reached at $t = 147$ after 12 steps of the algorithm. It can be seen that we reached an absolute gain of 10.5% with respect to the maximum CPU utilization of all cores in this run. This is the difference between the utilization of core 3 in the beginning (83%) and the utilization of core 7 with the final allocation (72.5%).

Table 5.1: Notation

Symbol	Description
$c \in \{0, ...C-1\}$	core number
C	number of cores of the CPU
LLC_c	LLC allocation to core c
$\underline{LLC} = \begin{bmatrix} LLC_0 \\ \vdots \\ LLC_{C-1} \end{bmatrix}$	LLC allocation of the CPU as a vector
$U_c(LLC_c) \in [0,1]$	CPU utilization of core c
$f(\underline{LLC}) = \underset{c}{max}(U_c(LLC_c)$	Maximum CPU utilization of all cores
LLC_{tot}	Total Last Level Cache available

5.4.4.4 Optimality Discussion

The scheduler minimizes the maximum CPU utilization of all cores for the static case. The CPU utilization of a core depends only on the allocated LLC if the memory access pattern is static. Thus the CPU utilization of core c is given with the function $U_c(LLC_c)$ where LLC_c denotes the share of the LLC usable by core c. The maximum CPU utilization of all cores is then:

$$f(\underline{LLC}) = \underset{c}{max}(U_c(LLC_c)) \tag{5.7}$$

$\underline{LLC}$ is a vector with length C, that denotes the current allocation of the LLC to the cores. Therefore, our optimization problem is:

$$\begin{aligned} min \quad & f(\underline{LLC}) & & \\ \text{s.t.} \quad & \sum_c LLC_c & \leq & \; LLC_{tot} \,, \forall c \\ & LLC_c & \geq & \; 0 \,, \forall c \end{aligned} \tag{5.8}$$

The constraints are due to the limitation of the total LLC of the chip LLC_{tot} and that the LLC allocated to one core must be non-negative.

The CPU utilization $U_c(LLC_c)$ is monotonically decreasing with LLC_c. More cache can only decrease the CPU utilization, as the CPU has to wait less for data. This means that the minimum f^* must be on the edge of the feasibility space, as we could otherwise increase LLC_c for any c and at least reach the same or a lower value of f.

The gradient of f, ∇f is a vector with $\nabla f_c = \frac{df}{dLLC_c}$. f in some point $\underline{LLC}^\circ$ is the maximum of the functions $U(LLC_c)$ in this point. It only depends on one dimension $c^\circ = \underset{c}{argmax}(U(LLC_c))$.

Consequently it holds:

Table 5.2: Experimental parameters used for evaluation.

Parameter	Range	Description
α	$\{0.3, 1.01, 2.5, 5\}$	Distribution parameter
M	$[1, 30]$ [MB]	Working set size
R	$[2000, 7000]$ $[s^{-1}]$	Access rate
C	5	Number of VNFs
U_c	$40\% < U_c < 90\%$	CPU utilization constraints

$$\nabla f_c = \begin{cases} g, & \text{if } c = \underset{c}{argmax}(U_c(LLC_c)) \\ 0, & \text{otherwise} \end{cases} \tag{5.9}$$

As all functions U_c are monotonically decreasing, we can state that $g \leq 0$. Note that we do not consider edge cases where the utilization of two cores is exactly equal, as they are obviously very rare in reality.

In each step the scheduler increases LLC_c for the core with the highest value and decreases it for the not scheduled cores. This means that the scheduler is moving along a line on the edge of the feasible space in every step. It always increases the LLC in dimension $c = argmax_c(U_c(LLC_c))$ and reduces it for the not scheduled dimensions $c^- \in NS \subset \{0, ...C-1\}$ and thus the scheduler is doing a gradient descent on the surface of the feasible space.

The step size is set to the size of one LLC way, which means we could overshoot in the case the step before was closer to the minimum. The scheduler takes this into account by saving the last valid Z (Z can never increase due to the conditions). Additionally, we argue that due to the characteristics of f and ∇f, in every point the final Z is close (within one step) to the global minimum f^*. We can not guarantee to reach f^* as the scheduler uses integer number of ways for scheduled cores.

5.4.5 Evaluation

This section first discusses the experiment design used for evaluation. First, we present the results that indicate a transient phase of the LLC after an update. Secondly the convergence of the scheduler is presented and finally, we show the scheduler gain depending on different parameters of the co-located VNFs.

5.4.5.1 Experiment Design

In order to evaluate the developed LLC scheduler, we evaluate 4000 scheduler runs. Depending on how many steps of the algorithm are needed each run has a duration of 150-200 s with one measurement point per second yielding an extensive data set.

One example scheduler run is shown in Figure 5.13. In every run, 5 VNFs are active and each one running pinned to one core. As explained before we consider a static scenario, i.e. within one run, the CPU utilization is constant, if the LLC allocation is not changed. Obviously this assumption is not realistic in real deployments, as e.g. the packet rate changes continuously, but it enables us to analyze the scheduler gain and the inertial behavior of the CPU utilization after an LLC allocation update. We emulate VNFs using a C++ program running inside a VM virtualized with KVM. The program is allocating a table of size M and accessing the memory with rate R using the distribution given in Equation 5.6. The code for the emulation is published for reference [1].

The scenarios are generated such that they are in accordance with the NFV MANO architecture that is described in Section 5.4.1. Hence, no VNF should be underutilized or overloaded. As a result, all VNFs in one scenario have a CPU utilization in the interval $[40\%, 90\%]$. The runs are generated as follows. The settings of each emulated VNF namely R and M are chosen randomly within the interval shown in Table 5.2, α is chosen randomly from the set shown. Note that $\alpha = 1.01$ is used, as otherwise with $\alpha = 1$ Equation 5.6 is much less complex to compute and thus the emulated VNF behaves different. Afterwards the 5 chosen VNFs are executed on our measurement server and the CPU utilization is measured. If the CPU utilization of an VNF is not within the defined interval R is increased or decreased. Then the CPU utilization is measured again. This pattern is repeated until the CPU utilization of all VNFs fall into the defined interval.

5.4.5.2 Monitoring

Intel processors provide low-level cache statistics via performance counters. These low-level counters can be read and interpreted using the Processor Counter Monitor (PCM) tool [107]. The developed algorithm uses the following metrics provided per core: current CPU utilization and LLC occupation.

5.4.5.3 Transient phase of the LLC

As we are dealing with a real system, the CPU utilization is always not fully constant over time. Reasons for this can be, e.g., periodic tasks the operating system or the hypervisor is

[1] VNF emulation source code: https://github.com/tum-lkn/vnf-emu

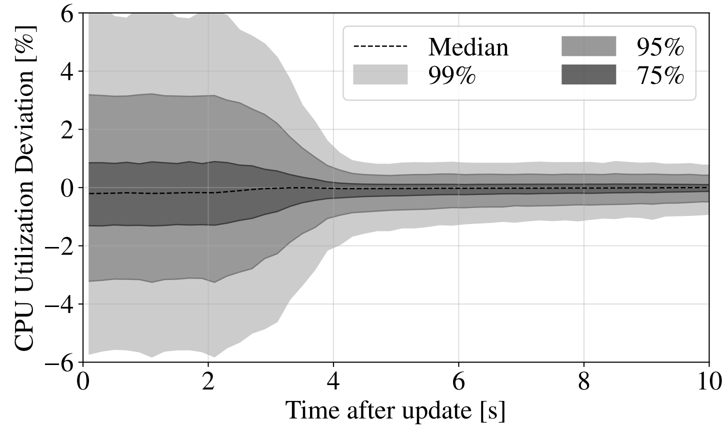


Figure 5.14: Transient phase after an LLC allocation update: Difference between the CPU utilization at a certain time after the update with respect to the median utilization of the CPU utilization in the interval [7,10].

performing. More importantly, the CPU utilization shows a transient behavior after an update of the LLC allocation: Cache lines are only evicted if other data not cached yet, is accessed by the CPU. Furthermore, the CPU gain from LLC cache only shows if the cache line is accessed after that a second time, as only then the accessing delay is reduced.

Figure 5.14 visualizes this transient phase. Results were gathered from 500 scheduler runs with one measurement point per second, each scheduler run needs multiple steps and thus yielding multiple transient phases. We define the median of the CPU utilization in the interval [7,10] s after a reallocation of the LLC as baseline or true utilization after the update. Next we compute the difference of each measurement value with the baseline and show the distribution as a contour plot showing the percentiles of the outcomes. It can be seen that the CPU utilization can differ significantly from the baseline for the first four seconds, after this the CPU utilization clearly stabilizes.

In line 6 of Algorithm 4, the scheduler measures the current CPU utilization of all cores. As this measurement must not be influenced by the last iteration of the scheduler, the algorithm has to wait until the CPU utilization is in a steady state. As we are aiming for the optimal allocation, we used a conservative time of 12 seconds for each step. Though in real systems one might trade off the step time for a faster scheduler convergence.

5.4.5.4 Scheduler Convergence

The presented scheduler is not optimized to converge quickly, rather than that it is designed to find the optimum. Nevertheless, we evaluate how long it takes until the optimum is found. As the scheduler gives exclusive ways until all cache ways are distributed, the maximum is 18 steps (as 2 ways have to be kept back for all other cores without exclusive ways). In our measurements this does not happen, as the scheduler assigns already more than one way to a

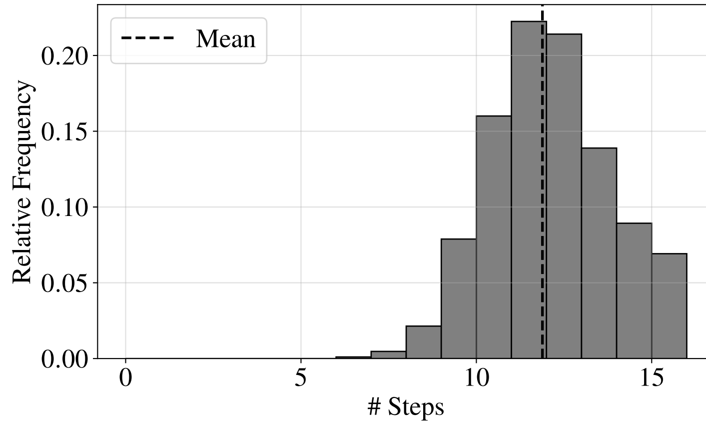


Figure 5.15: Number of steps the scheduler needs until the maximum is found. The scheduler needs on average 12 steps until all ways are distributed.

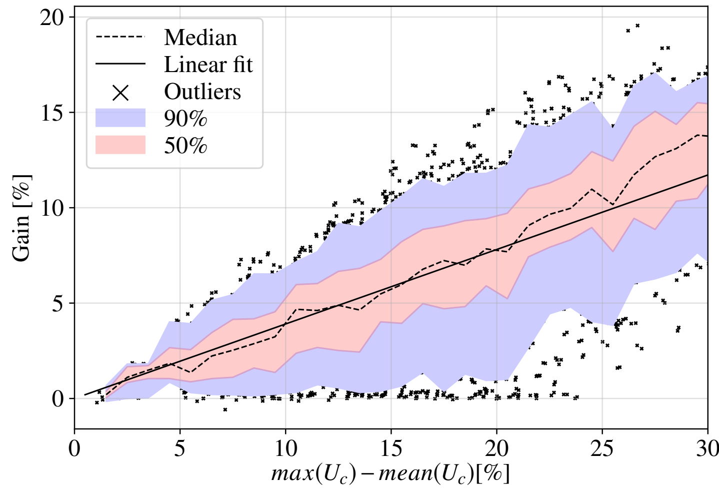


Figure 5.16: Relation between gain and difference between the CPU utilization of the highest core and the mean of the CPU utilization of all cores

core if the core gets exclusive ways for the first time (line 11) based on the current LLC usage in the shared case. Figure 5.15 shows the histogram of the frequency for a certain number of steps. It can be seen that for most of the runs 11 to 14 steps where necessary. On average the scheduler needs about 12 steps to find the optimum allocation.

5.4.5.5 Scheduler Gain

Next we evaluate how much gain can be expected from such an approach in a real system. In order to analyze this we evaluate 4000 different sets of 5 VNFs and determine the optimal allocation with our scheduler. From this sets we compute the gain as:

$$Gain = max(U_c)^{Shared} - max(U_c)^{Scheduled}$$

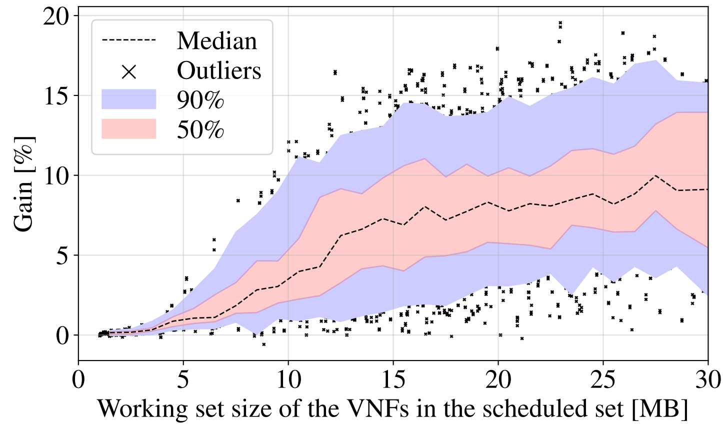


Figure 5.17: Relation between gain and the (mean) allocated memory of the scheduled VNF(s)

where $max(U_c)^{Shared}$ is the maximum CPU utilization with no LLC allocation (LLC is shared) and $max(U_c)^{Scheduled}$ is the maximum CPU utilization in the optimized case.

Figure 5.16 shows the gain with respect to the difference between the maximum CPU utilization and the mean utilization for the shared allocation. The linear fit line shows that for every percent of difference between max and mean one can expect 0.4 percent more gain. In the extreme case, where the maximum is equal to the mean, the scheduler cannot achieve anything as a reduction of the utilization of one core increases the utilization of other cores. This means that a real system needs a certain degree of variability between the VNFs running on one server, otherwise no scheduling is possible. Consequently setups that deploy only equal VNFs on one server and additionally load-balance between the instances, such that the utilization is as well equal for all VNFs, cannot reduce CPU utilization with LLC scheduling.

On the other hand, the outliers without gain show that this metric can not solely explain the achievable gain, but also depends on other metrics. One of these metrics is the allocated memory of the VNFs. The scheduler increases the LLC share for the core that has the maximum CPU utilization. If the complete working set already fits into the share of the LLC, a further increase of the allocated LLC does not yield any further gain. As a consequence it can be expected that LLC scheduling works worse for VNFs with a small working set. Thus we analyze the influence of the mean allocated memory of the scheduled set to the gain. We define the scheduled set as the VNF(s) that have exclusive ways in the final state of the scheduler. This means that each of the VNFs in the set is pinned to a CPU core which had the highest utilization in at least one scheduler interval.

Figure 5.17 shows how the allocated memory of the VNFs influence the gain. It can be seen that some minimum memory of around 6 MB per VNF is necessary to achieve gains. As we are using 5 VNFs in our experiments, the total allocated memory sums up to 30 MB which is the capacity of the LLC. This makes sense as a lower amount of memory doesn't

cause contention. As the scheduler is designed to reduce contention effects, it is not useful without contention.

Between 6 and 12 MB the median gain quickly increases and flattens out for higher amounts of memory. Consequently VNFs that do not store much data, like for example a stateless firewall that only needs to access its ACL regularly cannot reduce their CPU utilization. On the other hand, due to the limited size of the LLC (the LLC is 30 MB in total), the achievable gain is also limited even for a large working set size. A VNF requiring 30 MB of data can never cache everything in the LLC as this would leave the other VNFs without cache, which is technically impossible due to restrictions in the CPU chip architecture.

5.5 Summary and Discussion

With the move from dedicated hardware to multi purpose hardware for packet processing applications, performance aspects of this hardware are getting increasingly important. In this chapter we evaluate the effects of the memory architecture to the packet processing performance. The results show that copying packets between the NUMA nodes increases the CPU load drastically and should be avoided if possible. Overall performance penalty of copying between NUMA nodes can be up to 73%. The measurements using an increasing ACL size in memory show that the effects of CPU cache exhaustion should be considered when designing security VNFs. Additionally, we show that the CPU cycles needed for memory access using caches follow the AMAT model. This means that cache exhaustion increases the load on the CPU.

As multiple CPUs share the same LLC, the performance penalty is also existing for VNFs that do not exhaust the LLC alone. To mitigate this, we introduce an optimal LLC scheduler for NFV environments. We show that LLC scheduling can reduce the maximum CPU utilization of one NFV server by up to 20%. How much gain can be achieved depends on the employed VNFs and their traffic. As the CPU utilization of one VNF is decreased by increasing the allocated LLC of this VNF, the other VNFs that are co-located on the same server can use less LLC. Consequently this increases the CPU utilization of the co-located VNFs. Thus in order to enable the scheduler to reduce the overall maximum CPU utilization the difference between mean and maximum CPU utilization of all cores must not be too small. We show that the working set size of the scheduled VNFs must be large enough to lead to a LLC contention. Otherwise, with no contention present, LLC scheduling cannot bring any gains as every VNF can use sufficient LLC.

We have seen that the transient phase of the LLC is around 4 s. Thus the duration of one step must be about 4 s, as otherwise the measurement for the next step is faulty. In addition

on average we need 12 steps to find the optimum, thus the time the scheduler needs to find the optimum is about 48 s. Thus, the developed scheduler algorithm can only be used in static cases, i.e. traffic and VNFs must not change for some time. This assumption does not old in many real applications, as network traffic is known to be dynamic over time. Therefore, based on the static scheduler an adaption is needed, to be able to cope with dynamics in network traffic. Internet traffic changes in small scales over seconds but also in larger scales over minutes and hours. The shown approach can not work on a second scale as already the transient phase, i.e. the time until effects of the LLC allocation are effective is around 4 s. However, the shown algorithm can cope with more long-term changes.

The shown algorithm assigns more cache to the highest loaded VNF in each step until all cache is assigned. If the load changes it must be restarted with a completely shared allocation. This is a good starting point if the load and the VNFs have changed completely. In real cases the load will change slightly over time, so a better approach is needed. An approach that is extended with a mechanism that is also able to free up already assigned cache by removing it from lesser loaded cores would improve this. Otherwise the algorithm can stay the same. Nevertheless, the presented algorithm shows the necessary preconditions for effective LLC scheduling in terms of working size set and diversity of the VNFs.

Chapter 6

Connection Offloading using Software Defined Networking

The security architecture proposed in this work supports an omni-present fine grained access control throughout the network. Isolation between the virtual networks is provided using Software-Defined Networking (SDN). On the other hand Network Function Virtualization (NFV) is used to provide stateful and application layer filtering. Thus the complete traffic shall be filtered which causes high load on the Virtualized Network Functions (VNFs). As a relief, we propose in this chapter an offloading method that combines SDN and NFV. It provides a firewall that secures all connections between any hosts in the network via offloading specific flows to fast hardware processing on SDN-based network nodes.

With the move to NFV, network functions are realized in software, based on commodity hardware. On the one hand, costs are reduced and flexibility is increased, as virtualization techniques can be used and hardware resources are shared. On the other hand, VNFs may provide lower throughput when compared to hardware based functions, as they are not benefiting of specific hardware features. In the chapter at hand and the following chapter, we explore combinations of hardware and software network functions.

The Network Function Virtualization Infrastructure (NFVI) is a data center like centralized infrastructure, hosting the virtualized Firewall (vFW) instances. If every connection is filtered using the vFWs the capacity of the links to the NFVI must be high. Otherwise a few rate intense flows can lead to capacity bottlenecks and poor performance. One solution is to offload the connection filtering to the SDN switches and using the direct path. Offloaded flows are filtered in SDN hardware, reducing the resource consumption in the NFVI and in the network by avoiding detours to the NFVI.

In [5] we showcased this solution with a demonstrator using Linux netfilter as a vFW.First the vFW acts like a normal stateful firewall. With the first packet an entry is added to the

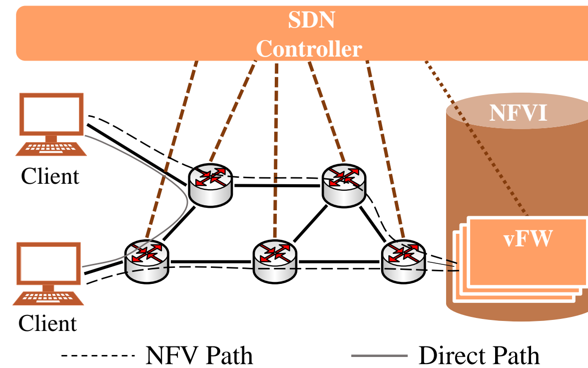


Figure 6.1: Network Function Offloading in an SDN/NFV Environment. The virtual firewall (vFW) filters the packets. As the infrastructure is centralized, a detour is caused for internal connections. After connection initiation the direct path can be used. The packets are still filtered by hardware, but only connections with high data rate should be offloaded to keep the rule count in the hardware tables small.

connection table. Subsequent packets are then checked if the protocol's state machine is followed. In the demonstrator the offloading decision was realized by detecting the current run time of a connection. If one connection has been active a certain duration the vFW a flow is offloaded and a message containing the flow five-tuple is sent via the northbound API to the SDN controller. The SDN controller then installs the necessary rules in the SDN switches and the load on the vFWs is reduced.

This simple approach is clearly in need of improvement as it does neither consider the table size nor it is able to offload connections from the first packet. But it shows that already a simple approach is of benefit with respect to the achievable data rate.

Due to the high number of connections in the network and limited size hardware tables, not every connection can be offloaded. The presented offloading algorithms can be used to decide which connections should be offloaded to the SDN switches.

We evaluate the offloading decision based on the number of rules that are needed and how much of the overall data rate can be offloaded. The better the offloading decision is, the higher the rate that is handled in hardware while keeping the table size small enough. Furthermore elephant flow detection is well known to introduce large monitoring overheads, that must be taken into account when designing a solution.

Nevertheless, a vFW has more capabilities in terms of filtering the packets, if compared to SDN network nodes. Software functions can be used for stateful packet filtering or even application layer filtering of the traffic. SDN devices can provide stateless filtering, but can not track the state of a protocol nor inspect the packets itself. That is SDN provides stateless firewalling only. Consequently only flows that do not need higher layer filtering can be offloaded.

Further, we belief that the results can be applied in a more general way than our motivated SDN/NFV firewall use case. Firstly offloading is not limited to firewall VNFs but can also be applied to other network functions. Depending on the hardware support, offloading can be used for a wide range of network functions. Secondly the offloading can not be done only with SDN devices, the same decision mechanism can also be applied for hardware that is using e.g. P4 [108] or even Network Interface Cards (NICs) that provide additional matching capabilities as studied in the next chapter of this thesis.

The offloading approach using a simple offloading algorithm that offloaded flows with a longer runtime than some threshold was demonstrated in [5]. One focus of the author of this thesis was, among other topics, on the offloading approach. The machine learning based offloading algorithm and the comparison to a sampling based approach was presented first in [8].

[5] B. Pfaff, J. Scherer, D. Hock, N. Gray, T. Zinner, P. Tran-Gia, R. Durner, W. Kellerer, and C. Lorenz. "SDN/NFV-enabled Security Architecture for Fine-grained Policy Enforcement and Threat Mitigation for Enterprise Networks." In: *Proceedings of the ACM SIGCOMM Posters and Demos*. 2017, pp. 15–16. DOI: 10.1145/3123878.3131970.

[8] R. Durner and W. Kellerer. "Network Function Offloading through Classification of Elephant Flows." In: *Under Submission in Transactions on Network and Service Management (TNSM)* (2019).

The rest of this chapter is structured as follows:

In Section 6.1 related work in the fields of network function offloading, elephant flow detection and traffic classification is discussed. Section 6.2 introduces the machine learning approach, the evaluated algorithms and shows a way how the data can be gathered. Afterwards the sampling based approach is introduced in Section 6.3. In Section 6.4 and 6.5 both approaches are evaluated in terms of possible share of the offloaded rate and necessary table size. Furthermore different parameters are studied in order to fine tune the approaches. We discuss the findings and compare both approaches directly in Section 6.6.

6.1 Related Work

In this section we give an overview of the related work in the literature. This can be categorized in three fields:

Some works introduce SDN offloading techniques for network functions, but are not using machine learning for the decision. Further there is related work in the field of elephant flow

detection. Finally other works examine the usage of machine learning for classification of network traffic. These works aim to classify the application (HTTP, SMTP, etc.) of the flows, while we specifically try to decide if a flow should be offloaded, without caring about the application.

6.1.1 SDN Offloading

SciPass [109] use an OpenFlow switch for the offloading of an institutional firewall in a science network. The system consists of a 10G Firewall, a 100G OpenFlow switch and the Bro Intrusion Detection System (IDS) that is used to identify the flows that can be offloaded. The evaluation shows that a firewall bypass can significantly improve performance of the network as the OpenFlow switch has a much higher data rate. In contrast to our approach, SciPass is using a specialized IDS system for identification. The work also focuses on the very specific use case of a science Demilitarized Zone (DMZ).

NFShunt [110] is a prototype implementation for firewall offloading, realized with a Linux Netfilter based software firewall and a OpenFlow switch, that is used as hardware accelerator. The paper details the use-case of a science DMZ and is showing implementation details of the prototype. In contrast to our analysis, the authors are using static rules for the offloading decision.

Recently Heimgaertner et. al. [111] published their work on firewall offloading that is specifically used to avoid congestion at the firewall. The authors are using two different algorithms for the offloading decision: A random decision choses random flows for offloading and a so called intelligent algorithm that decides for offloading based on the byte count of the flow. The results show that the bypassing of the firewall can significantly improve performance. Furthermore it can be seen that the decision algorithm is important, as more load is bypassed using fewer rules in the OpenFlow switch with the intelligent algorithm than with the random algorithm.

Sarrer et. al.[112] propose a heavy hitter detection for offloading traffic from software routers to hardware. They exploit that the amount of traffic per flow follows Zipf's law. In contrast to our work they are focusing on an offloading design for routers, while we focus on stateless firewalls.

All of the works before are using the history of the flows for a decision and do not consider prediction and machine learning techniques that try to decide with the first packet, as we do.

6.1.2 Elephant flow detection

In general there is a lot of work on elephant flow detection, mostly focusing on detecting elephant flows, while they are already elephants. This means, many works concentrate on detecting the flows that at a certain time compromise the most to the overall network traffic. The overall problem is that the number of network packets is too large in modern networks to be analyzed packet by packet. There are a number of different approaches to tackle this issue:

Widely used approaches such as NetFlow [113] and FlowRadar [114] are using hash based approaches in the data plane. Other works are using tables of hardware switches to detect active elephant flows [115, 116]. Both require specific hardware in the data-plane, while our approach does not rely upon such hardware.

Some works are using sampling based approaches [111, 117–119]. The underlying idea of these approaches is that the number of packets of an elephant flow is much higher than for a mice flow. Therefore, the probability to miss an elephant flow is low, while the complexity of the detection can be greatly reduced.

A number of works consider elephant flow detection for routing improvements in data center networks: Sarrer et. al. [112] propose a heavy hitter or elephant flow detection for offloading traffic from software routers to hardware. They exploit that the amount of traffic per flow follows Zipf's law. DevoFlow [120] and Mahout [121] improve routing in data-centers by rerouting elephant flows while mice flows are routed using ECMP. This improves the scalability of the solutions as fewer flows have to be considered. Mahout is using end-host based elephant flow detection, which requires modification at the end-hosts. DevoFlow and Sarrer et al. are using flow statistics, thus they detect elephant flows much later when already substantial number of packets was routed. Instead, we aim to detect elephant flows with the first packet. Additionally due to the use cases they do not consider table occupation.

Further there are also a number of works studying machine learning approaches for elephant flow detection. Pouper et al. [122] propose the use of neural networks, Gaussian mixture models and Gaussian process regression for the prediction of the flow size. Xiao et al. [123] use C4.5 decision trees and Viljoen et al. [124] are using a neural network to classify flows into mice and elephant. Chao et al. [125] uses a 2-stage detection scheme with C4.5 trees as a first stage and stream mining with Hoeffding trees in the second stage. All works above show the feasibility of machine learning for elephant flow detection, however none aims at a classification with the first packet. Further their evaluation is based on traffic engineering use cases as well, i.e. they do not consider table size restrictions.

6.1.3 Traffic Classification

Some years ago, Nguyen and Armitage have published a survey on using machine learning for traffic classification [126]. They identify different categories: Port-based classification uses IANA-Standard ports and suffer from non-standard ports and the use of http for essentially different applications and not web only. Payload based classification methods use deep packet inspection and are not able to classify encrypted traffic. Statistical classification uses statistical features like mean-packet size and is the category which our approach belongs to.

Zhang et. al. [127] propose a traffic classification method, that tries to find the applications, even if some applications are unknown. They also compare their results with the algorithms C4.5, NaiveBayes and IBk.

CAIDA presented a traffic identification framework in [128], which includes different traffic classification approaches. One highlighted use case is the classification of applications using algorithms like C4.5, K-Nearest Neighbor and NaiveBayes.

More recently [129] and [130] published works on classifying applications from statistical traffic features. They study RandomForest, C4.5 and NaiveBayes as classification algorithms and show that the algorithms can yield high accuracy in this field.

All of the works above are tailored to classify the application in order to employ Quality of Service (QoS), none tries to find heavy hitters or is tailored for SDN offloading.

6.2 Offloading with the first packet

In order to overcome the limitations in the state of the art with respect to flow classification, we introduce a classification system that decides with the first packet if a flow is worth to be offloaded to hardware or not. The envisioned VNFs provide stateful and application layer firewalling; stateful and application layer firewalls have to handle the first packet of a flow separately anyway. E.g., a stateful firewall tracks the state of each connection and adds a new entry in its state table with the first packet. In our approach we can make use of this specific processing path to forward the features of the first packet from the VNF to an SDN controller can then classify the flow and take the offloading decision.

In this work we introduce a classification system that decides if a flow is worth to be offloaded to hardware or not with the first packet. The classification system is implemented using machine learning and includes data gathering and pre-processing.

Figure 6.2 illustrates the steps of the classification system:

1. In a first step the traffic is recorded by a monitoring system. This step is described in more detail in the next section.

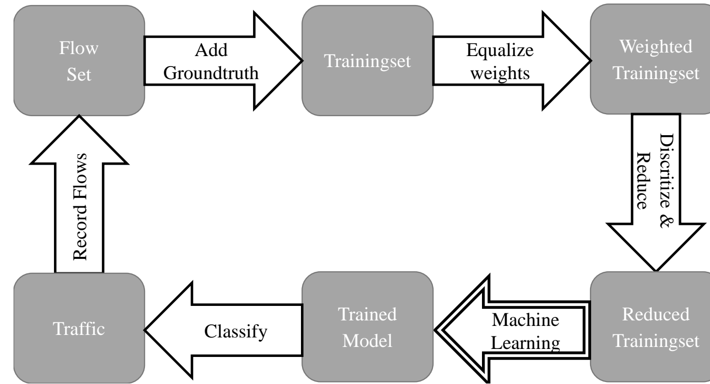


Figure 6.2: The classification system: The packets from the trace resp. on the line are recorded as flows. The ground truth is added depending on the gathered statistics. After that some filters are applied aiming to build a basis for the machine learning algorithm. The algorithm uses the set to train a model that can be used to classify new connections. These connections are recorded again for the next iteration.

2. From this raw data, a ground truth is derived: All flows that are bigger than a threshold are labeled as 1 all others are labeled with 0. This means the class is binary where 0 represents no offloading and 1 offloading.

3. The weights of the classes are equalized by weighting some flows higher than others. This is necessary as much more flows are of class 0 and the resulting models would consequently also be biased towards class 0.

4. The feature first packet size is discretized. Infrequent nominal values of all features are merged.

5. The model is trained by the respective machine learning algorithm.

6. The trained model is used by the SDN Controller to decide if new flows are offloaded or not.

For the evaluation in this work Step 1 is replaced by a packet-trace parsing script. Step 2-6 are implemented using Weka [131]. Steps 1-5 have to be repeated regularly to retain an up to date model for the classification.

6.2.1 Gathering the training-data

Gathering of the training-data is not the focus of this work, nevertheless we want to outline an architecture that could perform this necessary task. In particular, it is about the realization of the measurement points. At the measurement points the packets have to be grouped to flows and the features described in 6.2.2 are extracted from the first packet of a flow. Therefore the

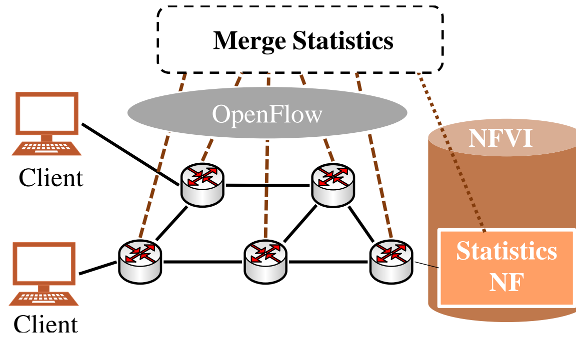


Figure 6.3: Gathering the necessary statistics using multiple points in the network. In a normal case packets are processed using a function chain located at the NFVI, in this case statistics can be gathered using a statistics network function. If a flow is offloaded the statistics have to be gathered by the hardware. All statistics have to be consolidated in the end, e.g. using an SDN controller or a network management system.

measurement points have to be able to group the flows using their five tuple, gather statistics and to extract the size of the first packet. After the first packet the accumulated size of the transmitted data of the flow has to be gathered, this information is needed to get the ground truth. In order to provide valid data all the network traffic has to be analyzed and not only a subset. This is especially important for a system with deployed NFV hardware offloading. As the offloading bypasses the VNFs, in turn these VNFs cannot be used alone for the data gathering.

A centralized network monitoring system that merges the measurements from hardware and software systems can solve this problem. Figure 6.3 depicts such an architecture. By default, offloading is not used, then all statistics, i.e. the size of the first packet and the total size of the flow can be easily extracted in a statistics network function (Statistics NF) realized in software. Additionally the hardware has to be capable to count the transmitted data of the offloaded flows. This is possible with current networking hardware that provides statistics via OpenFlow or sFlow. Finally statistics from hardware and software have to be merged together by a central entity, e.g. an SDN controller or a network management system. The resulting data set can then be used for labeling the ground truth. All flows that have a total size of more than a threshold Θ_f are labeled with class 1, all others with class 0. How the threshold influences the offloading decision is shown in Section 6.4.2.

In the presented work we used a Python script to parse the network traces that are shown in Section 6.4.1. Dumping the network traffic to a file could also be used in a real system, but might be problematic in practice, due to high overhead for storing and recording such a trace and additionally causes privacy issues.

Feature	Type
IP source	Nominal
IP destination	Nominal
IP protocol number	Nominal
L4 source port	Nominal
L4 destination port	Nominal
Size of first packet	Nominal (Discretized)

Table 6.1: Features used for machine learning. All features are available with the first packet of a flow. This enables early classification and a larger gain compared to other approaches.

6.2.2 Features

Our classification approach essentially tries to separate mice from elephant flows. As the classification is used as input for the hardware offloading of network function, the decision should be available from the beginning. This is why only features available with the first packet are used. Secondly with more and more traffic encrypted, only features that can be directly deduced from the packet header should be used. Especially upcoming standards like QUIC and Transpor Layer Security (TLS) 1.3 reduce the clear-text parts of the packets even further when compared to current standards. The chosen features that are shown in Table 6.1 are available with the first packet even when TLS encryption is used.

IP source and destination combine both IPv6 and IPv4 addresses. L4 source and destination port are both UDP and TCP ports, for other protocols this feature equals 0. Our notion of flows is bidirectional, this means the first combination of a five tuple is saved as a flow according to this packet's headers. If a packet of the same connection is seen in the opposite direction it is counted for this flow. The five-tuple features IP source, IP destination, IP protocol number, source port and destination port are nominal or categorical features, as the information depends on the specific number and not on the range. In order to avoid over-fitting of the model and

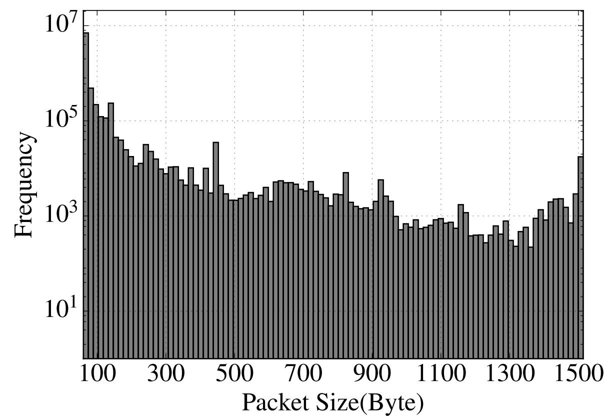


Figure 6.4: Histogram of the size of the first packet from each flow for the Wide A trace. The distribution shows high frequency for small packets.

reduce the complexity of both training and classification we merge all nominal feature values with a frequency of less than f into one value. An evaluation of parameter f is give in Section 6.4.4

For each flow the size of the first packet in Bytes is stored during flow recording. Figure 6.4 shows a histogram of the size of the first packet from each flow. As can be seen from the figure the frequency of small packets with a size close to the minimum Maximum Transmission Unit (MTU) of Ethernet is very high. These small packets can be either e.g. ICMP packets that constitute a mice flow, or possibly a TCP-Syn resulting in an elephant flow. On the other hand, bigger first packets can be part of a small or large flow as well (e.g. a Syn with TCP Fast Open [132] or a DNS request). Therefore using the packet size as a numeric feature will not give the best classification results. Consequently the size of the first packet is discretized. We are using equal frequency binning that creates bins with an equal number of instances, instead of regular discretization where the bins cover the continuous numeric space equally. This takes into account for the differences regarding the first packet frequencies and results in more bins for small packet sizes and fewer bins for larger packet sizes. We performed studies that show a significantly increased performance with equal frequency binning compared to the numeric feature.

6.2.3 Machine Learning Algorithms

In this section we briefly introduce the employed machine learning algorithms.

NaiveBayes is a simple algorithm for creating a classification model. It is based on Bayes' theorem. To compute the probabilities NaiveBayes uses the assumption that the value of one feature given the class is independent of the values of the other feature. For our real world problem this is an assumption that will not hold as for example the port and IP of a server are coupled and are clearly not independent. Nevertheless it has been shown in [127] that NaiveBayes still works well for many real world cases even if the assumptions of Bayes' theorem do not hold.

On the one hand the machine learning algorithm should be able to build a model that is accurate, i.e. has a high classification performance. On the other hand it should also build a model that is fast in classifying new instances and also understandable to support debugging. Decision tables can support these concerns. For classifying the instances, the table is searched for an exact match if no match is found, the majority class is returned. For building the table we used the IDTM algorithm presented in [133]. Note that not all features are included in the table, the feature subset is chosen using BestFirst heuristic according to [133]. On the other hand decision tables are known to have a tendency to overfit. This means that the learned table fits very well to the training set but is bad for predicting new unseen instances.

This can be solved by using tree algorithms, that can be tuned in a way that avoids overfitting. Additionally the complexity for the prediction in the offloading logic is low, when trees are used. J48 is the Java implementation of the C4.5 algorithm presented in [134]. The algorithms generates a decision tree from the training data, that is then used for the classification. Every decision node in the tree takes a decision based on one attribute. J48 uses the entropy to decide the distance to the root of the different attributes in the decision tree. Each leaf of the tree identifies a class value, in our case as the class is binary, 0 or 1. Therefore the maximum number of decisions that have to be made for a classification is p+1, where p is the number of attributes. For our experiments we set the algorithm such that each leaf has to contain at least 2 instances. Additionally we used pruning to reduce the tree size and avoid over fitting with a pruning confidence of 0.5, following best practice.

One way to improve classification performance is to use ensemble methods. Ensemble methods combine the prediction of multiple other learning algorithms. The drawback is that complexity is increased, as multiple base models have to be trained or evaluated. We used two popular ensemble methods namely Random Forest [135] and Adaptive Boost Algorithm or shortly AdaBoost [136].

The Random Forest algorithm generates an ensemble of trees, the decision is then taken by voting for the most popular class. Random Forest is designed to improve classification accuracy while being robust against outliers and noise. The used algorithm uses bagging together with random feature selection for creating the trees. Bagging is a method to combine different models created from samples of the training set. We used a bag size of 5 % and a minimum number of instances of 100 per leaf. Each tree uses $F = log_2 p + 1$ randomly selected features from a total of p features.

Random Forest is specifically created to be used with tree algorithms, AdaBoost on the other hand can be used in conjunction with many other algorithms. It combines team of multiple inner models in a weighted sum, which is then used for predicting the class. The AdaBoost algorithm chooses the inner algorithms systematically to find a good classifier for all instances in the learning set. For each iteration one inner model is chosen such that the team with the new inner model performs better than the old team. In this work we are using the M1 variant of the algorithm with 10 iterations. As inner classifiers we are using DecisionStump. DecisionStump builds a decision tree based on one attribute with only one level, i.e. it consists of the tree root and the leaves. Nevertheless in conjunction with an ensemble learner like AdaBoost, the DecisionStump algorithm can be used for building well performing classifiers [136].

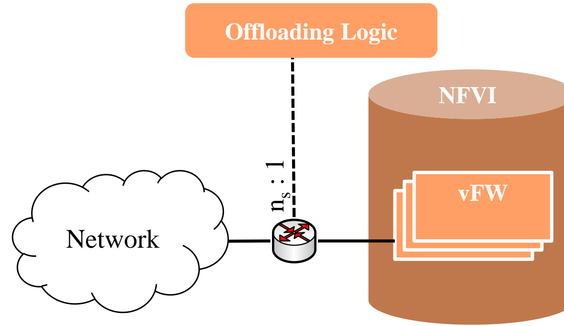


Figure 6.5: Sampling based Offloading approach: Every n_s packet is forwarded to the Offloading logic. If the decision is for offloading the flow of the corresponding packet is offloaded by the SDN Controller.

6.3 Sampling based approach

An alternative approach for SDN Offloading is a sampling based approach: Instead of being based on the first packet of a flow the offloading decision is based on randomly sampled packets. Sampling is a common method for monitoring networks and is realized e.g. by sFlow [137]. We want to compare our machine learning algorithm with algorithms based on sampling. The employed algorithms are modified versions of Heimgaertner's [111] algorithm. Heimgaertner's approaches are designed for imminent congestion and therefore not directly comparable.

In general the sampling based offloading approach shown in Figure 6.5 work as follows: The sampled packets are forwarded to the decision logic. The logic decides if the flow corresponding to this packet should be offloaded or not, based on the packet and the internal state of the logic. If the decision is for offloading a new flow is installed in the OpenFlow switch by the SDN Controller and the load on the firewall is reduced.

The employed decision algorithms are modified versions of Heimgaertner's [111] algorithm. Heimgaertner's approaches are designed for imminent congestion and therefore not directly comparable. The results show that the offloading algorithm itself is, somehow surprisingly, of minor importance for the sampling based approach if real network traces are used. Instead, we show in Section 6.5 that the sampling rate is the most important parameter. Therefore, we only show two rather simple algorithms for the offloading decision in the following.

6.3.1 Baseline Algorithm

One very simple yet effective offloading algorithm is to always decide on offloading for every sampled packet. We call this algorithm Baseline as it achieves the maximum reachable

Algorithm 5: Sample+ offloading algorithm

```
Input: Sampled Packet, t
Output: Offload Decision
1  5t = get5tuple(packet);
2  if 5t ∈ T then
3  |  T_c[5t] = T_c[5t] + 1;
4  else
5  |  r_F = updateFlowRate(r_F, t);
6  |  T_c[5t] = 0;
7  end
8  p_off = min( n_by / (t_out · r_F), n_by^rem / (t_out · r_F · th) );
9  if random[0, 1] < p_off · T_c[5t] then
10 |  return True;
11 else
12 |  return False;
13 end
```

offloaded rate for a certain sampling rate. Due to the sampling, not every packet and thus not every flow is offloaded. Thus, the maximum offloaded rate is limited. The only parameter of this approach is the sample rate n_s, i.e. only one out of n_s packets is forwarded to the offloading algorithm.

6.3.2 Table restricted approach Sample+

In order to restrict the table size necessary for the offloading, the algorithm can be improved. The following algorithm is based on Heimgaertner's algorithm. In contrast to our approach, Heimgaertner's algorithm is designed for Offloading during immanent congestion and assumes that this is only there for a fraction of the time. Furthermore it does not adapt the offloading probability to the number of appearances of a flow.

We denote the algorithm shortly as *Sample+*. The target offload r_{off} rate should be as large as the inverse of the usage time of one entry t_{reuse} multiplied by the target number of entries in the table n_{by}.

$$r_{off} = \frac{n_{by}}{t_{reuse}} \tag{6.1}$$

If Equation 6.1 holds, then the number of rules in the table is exactly as large as targeted. The reuse time is not known beforehand, it consists of the remaining active time of the flow after offloading it t_{rct} and the soft timeout t_{out} of the OpenFlow switch table.

$$t_{reuse} = t_{rct} + t_{out} \tag{6.2}$$

Therefore it holds $t_{out} < t_{reuse}$ and we can upper-bound $r_{off} < \frac{n_{by}}{t_{out}}$. We denote the rate of new flows seen by the algorithm as r_F. r_F is estimated by the offloading logic using UTEMA method [138]. Further we can then upper bound the offloading probability p_{off} as follows:

$$p_{off} = \frac{r_{off}}{r_F} \leq \frac{n_{by}}{t_{out} \cdot r_F} \tag{6.3}$$

In order to compensate that Equation 6.3 gives only an upper bound we use a decreasing probability if the table is nearly full. The offloading probability p_{off} is then defined as:

$$p_{off} = min\left(\frac{n_{by}}{t_{out} \cdot r_F}, \frac{n_{by}^{rem}}{t_{out} \cdot r_F \cdot th}\right) \tag{6.4}$$

n_{by}^{rem} denotes the number of remaining entries, i.e. the number of used entries subtracted from n_{by}. The threshold th specifies the border between the first and the second term in the minimum. In our experiment we have set $th = 0.1$, i.e. for a table occupation of 90% both terms in Equation 6.4 are equally large.

6.4 Evaluation of the Machine Learning approach

For the evaluation of the machine learning approach we initially present the results using stratified cross-validation. This approach delivers multiple outcomes for one data set and allows a better grading of the algorithms than a single result for each data set.

6.4.1 Data Sets

As it was not possible to employ the presented system in a real network, we used publicly available data sets for the evaluation:

	Wide A	**Wide B**	**WIDE IX-24h**	**CAIDA 1**	**CAIDA 2**
Duration	15 min	15 min	24 h	15 min	10 min
Flows	8M	8.6M	344M	45.6M	40.6M
Packets	99M	113M	15965M	977M	874M
Total Traffic	70 GiB	91 GiB	19 TiB	716 GiB	631 GiB
TCP flows share	85.8 %	77.6 %	46.0 %	83.6 %	80.6 %
UDP flows share	12.4 %	22.1 %	48.33 %	16.2 %	19.2 %

Table 6.2: Data sets used in this work

The machine learning approach tries to detect patterns in the network traffic. One such pattern could be an IP address of a popular video service, connections to this IP would be

elephant flows regularly. These patterns might be very diverse in nature and therefore hard to model. As a consequence simulated or emulated traffic can no be used for the evaluation and we must use real traces.

We have chosen multiple traces among the publicly available traces to evaluate our results on diverse data sets in terms of link capacity, average flow size and composition. We use five publicly available data sets from three different measurement points. Main parameters of the traces are shown in Table 6.2.

The Wide data sets where retrieved from the MAWI Working Group Traffic Archive [139]. The trace is collected from the 1 Gbps transit link of WIDE to their Internet Service Provider (ISP). The traces from two consecutive days, Thursday and Friday, out of the daily traces from the MAWI Working group. The daily traces are collected every day at 14:00 to 14:15. Furthermore we use an 24 hours trace (WIDE IX 24h) to study temporal effects of the machine learning solution. This trace was gathered from the main link of wide to the internet exchange point DIX-IE on a Tuesday. The WIDE IX 24h trace has a significant larger share of UDP flows mostly due to a high number of DNS flows.

The CAIDA data sets were retrieved from the Center for Applied Internet Data Analysis [140]. The trace was collected at an data center's link to the backbone link of a Tier 1 provider. The link has a maximum data rate of 10 Gbps, this yields a much higher data rate and a larger number of parallel flows.

All traces where anonymized from the respective organizations. As statistical features were retained by the anonymization, our results are not influenced.

6.4.2 Data Labeling

In this work we are using supervised learning algorithms. This class of algorithms needs labeled data. Our classification problem is only binary with the class being the offloading decision that can be True or False. We labeled the data using a threshold with the following rule:

$$Offload = \begin{cases} True & L_F > \Theta_F \\ False & L_F \leq \Theta_F \end{cases}$$

Where L_F is the total size of a flow, meaning the sum of the lengths of the packets belonging to one flow. Figure 6.6 shows the offloaded share and the table occupation for different thresholds Θ_F in the Wide A and the CAIDA 1 data set. The results in the figure show the table occupation and offloaded share if the labels are used directly for the decision. As this requires an all knowing system this is not possible in reality. Nevertheless the results

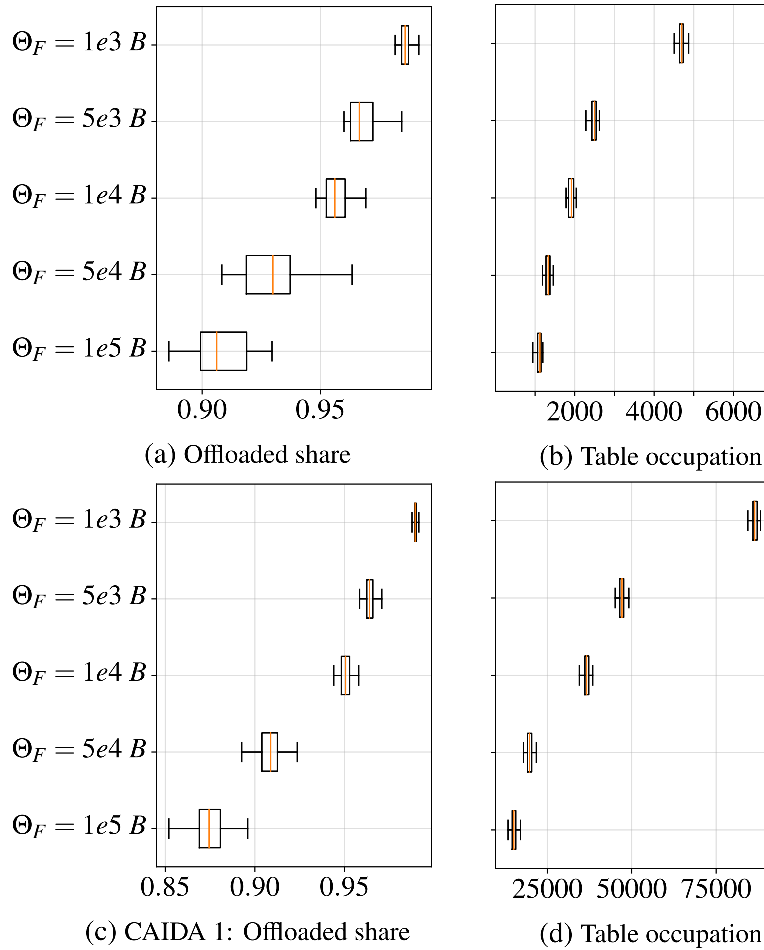


Figure 6.6: Data Labeling: Offloaded share and Table occupation if the labels are used for the decision directly. For a small threshold Θ_F the table occupation is quite high even in this all-knowing case. But it quickly decreases. We chose $\Theta_F = 1e4\,B$ as a compromise between table occupation and offloaded rate.

show the trade off between required table size and offloaded share. Small thresholds yield a high table occupation obviously, on the other hand the results show that rate share and table occupation are not proportional. E.g. in the WIDE A changing the threshold Θ_F from $1e4$ to $5e4$, reduces the offloaded rate share by only 1.1%, while reducing the necessary table size by 11%. The threshold should be chosen such that the hardware table is fully utilized but not over-utilized. In our test-bed we have an OpenFlow enabled NEC PF5240 switch with a table capacity of 64K entries suitable for the CAIDA data set. Following these guidelines, we decided to use $\Theta_F = 1e4$ as a threshold in the following reaching an ideal offloading rate of 95% for both data sets.

6.4.3 Feature selection

Next we evaluate the information gain of the features, in order to understand the main drivers of a good classification scheme. Additionally the results can be used to reduce the complexity of training and classification, by omitting features with low gain.

Table 6.3 shows the information gain of each attribute p:

$$InfoGain(Class, p) = H(Class) - H(Class|p)$$

where $H(X)$ is the entropy of random variable X.

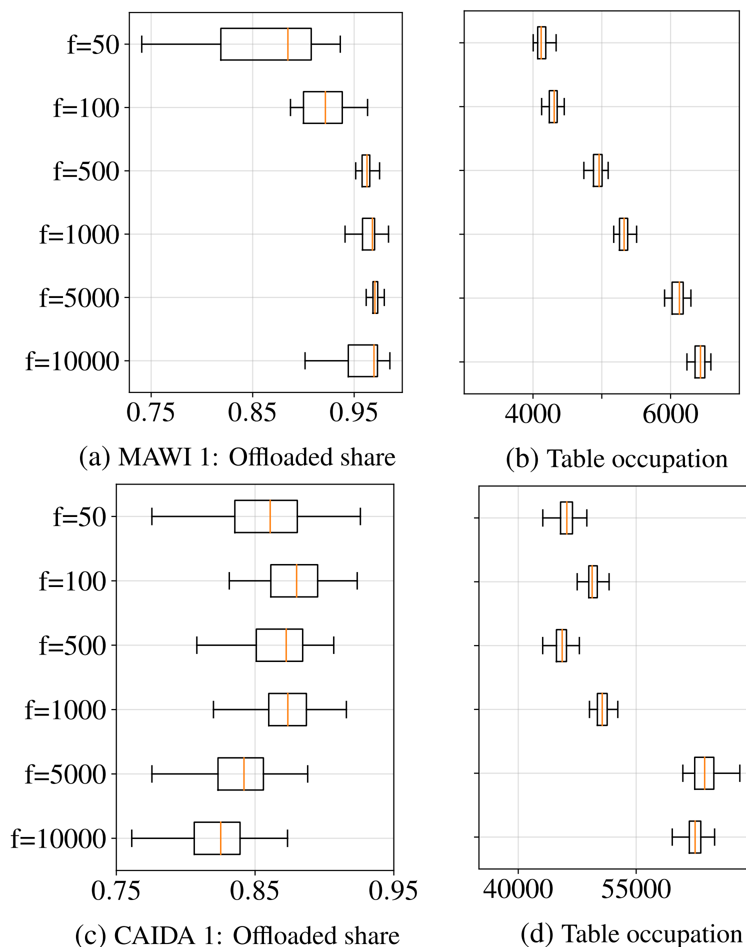
	Wide A	**Wide B**	**WIDE IX-24h**	**CAIDA 1**	**CAIDA 2**
IP source	0.009	0.009	0.042	0.030	0.029
IP destination	0.007	0.007	0.029	0.021	0.019
IP protocol number	0.001	0.001	0.017	0.002	0.002
L4 source port	0.001	0.001	0.001	0.006	0.006
L4 destination port	0.009	0.010	0.037	0.008	0.008
Size of first packet	0.009	0.009	0.027	0.023	0.022

Table 6.3: Information gain of the attributes in bit

From the table can be seen that the information gain of each feature is small (maximum 0.042 bit for IP source and WIDE IX-24h) and no single attribute can be used for a decision. Additionally the gain of attributes depends on the network conditions, as the size of the first packet has a much higher gain for the CAIDA data sets than for the WIDE data sets. Only the IP protocol number shows little information gain and could be omitted for performance reasons. As the notion of 5-tuples is very common in packet processing systems we keep the feature for the rest of the evaluation nevertheless. On the other hand using only the 5-tuple features would cause a loss in information for the classification and will therefore reduce the precision. This complicates data gathering a little as for example sFlow cannot be used to record the size of the first packet.

6.4.4 Merging infrequent nominal values

The employed features are mainly nominal, e.g. IP addresses which are numerically close to each other do not necessarily have similar effects on the classification. Consequently the models have to use single nominal values to give a prediction. As the learning sets can be build of a large number of flows the value sets for nominal features can be quickly very large. This is especially a problem for the IP addresses, even if only one packet is sent from one IP, the models have to keep track of that.

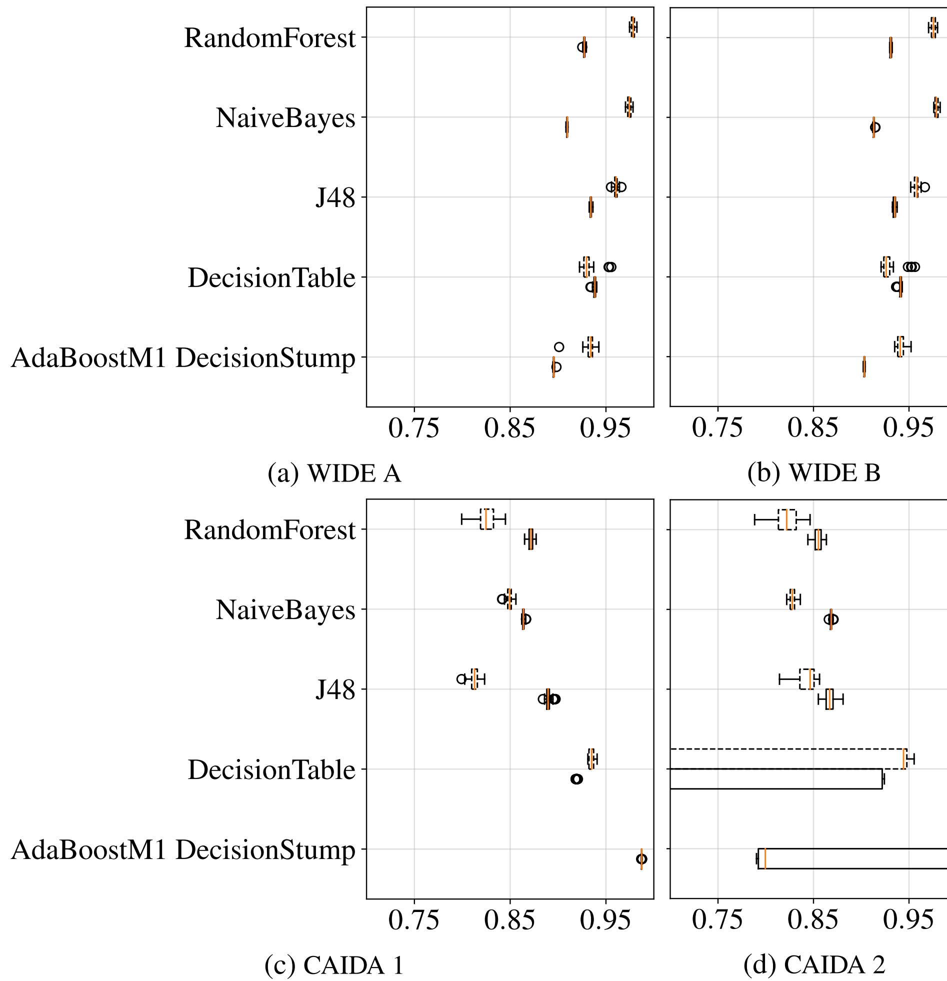


(a) MAWI 1: Offloaded share (b) Table occupation

(c) CAIDA 1: Offloaded share (d) Table occupation

Figure 6.7: Merging of infrequent values: The Figure shows the performance of the offloading algorithm with J48 as machine learning algorithm. Nominal features with a lower frequency than the minimum frequency f are merged. Table Occupation and offloaded bit rate increases with increasing f. f=500 provides a good compromise.

In order to overcome this issue infrequent nominal feature values are merged to one value. This is done by merging all values that did occur in the training set less often than the frequency f into one single nominal value. We apply merging to all nominal features with the same f. Figure 6.7 shows the trade off between table occupation and offloaded bitrate that occurs with this parameter. A low f causes over-fitting and elephant flows are not recognized well. On the other hand a high f under-fits and the table occupation necessary in turn increases. Additionally the complexity of training and classification largely depends on f. A frequency of f=500 provides a good compromise between over- and under-fitting for both data sets, even though they are quite different. Thus f=500 is used for all other results.



(a) WIDE A (b) WIDE B

(c) CAIDA 1 (d) CAIDA 2

Figure 6.8: Precision of the selected algorithms with the different data sets. The mean precision for class 0 (offload=No) ranges between 0.84 and 0.96 depending on data set and algorithm. The precision for class 1 is worse in general. J48 and RandomForest show the best classification precision in general. The Decision Table classification precision is differing largely for the CAIDA data sets. Boosted DecisionStump is showing decent performance for the WIDE data sets but has only low accuracy in the CAIDA data sets.

6.4.5 Algorithm evaluation

The precision of the classification is defined as usual:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

As we want to select those flows that are worth offloading in hardware, we want a high true positive rate for class 1. On the other hand if we select too many flows for the offloading, we need a large hardware forwarding table and might exceed its capacity. Figure 6.8 shows the results for the different data sets and both classes. The boxes are ranging from the lower to the upper quartile of the outcomes. The whiskers mark the full range of the outcomes. The line in the box represents the mean.

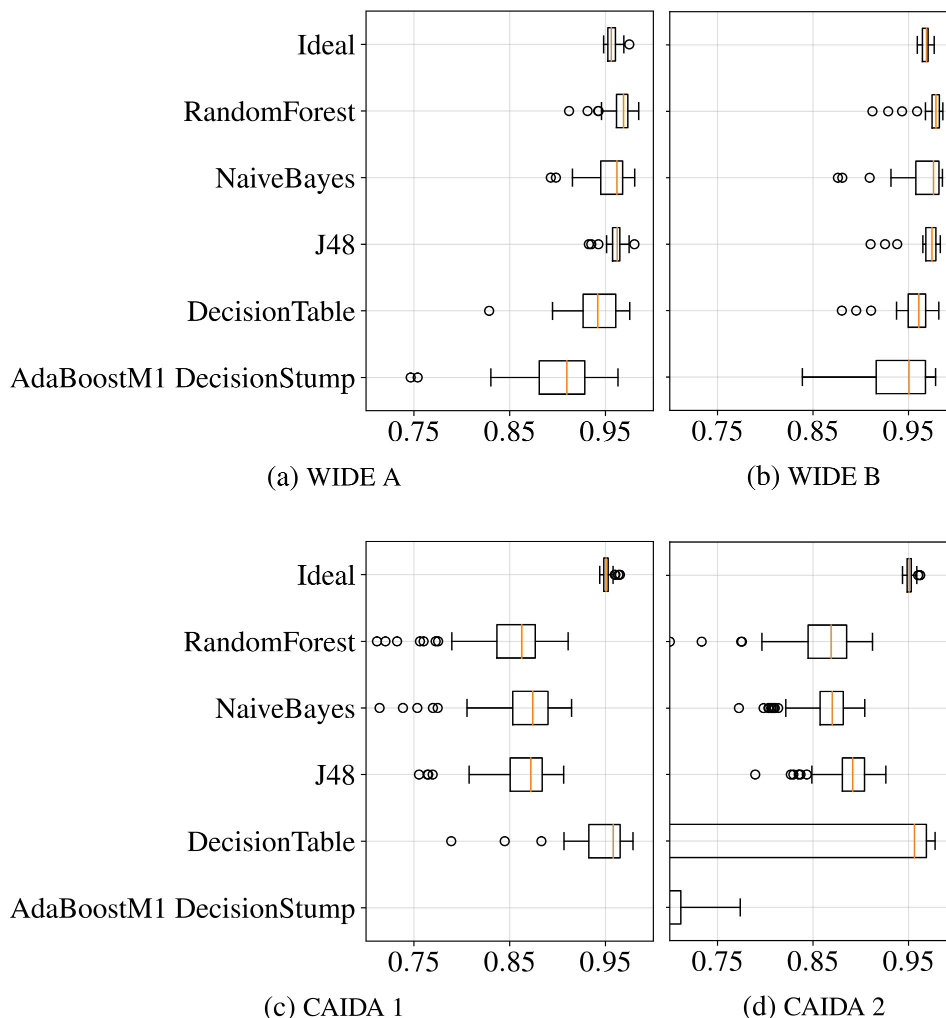


Figure 6.9: Offloaded data rate using the different learning schemes. The bit rate that is processed by hardware is approximately as good as with the all-knowing ideal scheme. This is due to the much higher number of processed flows compared to the ideal case. In general the lion's share of the rate can be processed in hardware.

The mean precision ranges between 0.85 and 0.95 depending on data set and algorithm. J48 and RandomForest show the best classification precision in general. Especially J48 shows a high precision for both classes. The precision of the DecisionTable classifier is below 85 % for some folds in the CAIDA data sets, on the other hand the results for the Wide data sets are unremarkable. Another outstanding result is the precision of the boosted DecisionStump algorithm in the CAIDA 1 data set. The model almost always decides for class 0, this results in a precision of almost 100 % for class 0 and close to 50 % (out of the scale of the figure) for class 1. For the other data sets the performance is clearly worse than that of other algorithms. Consequently we did not consider boosted DecisionStump further in the following.

One main drawback of networking hardware compared to software solutions, is the limited hardware resources especially the tables. In the ideal case we are only using hardware

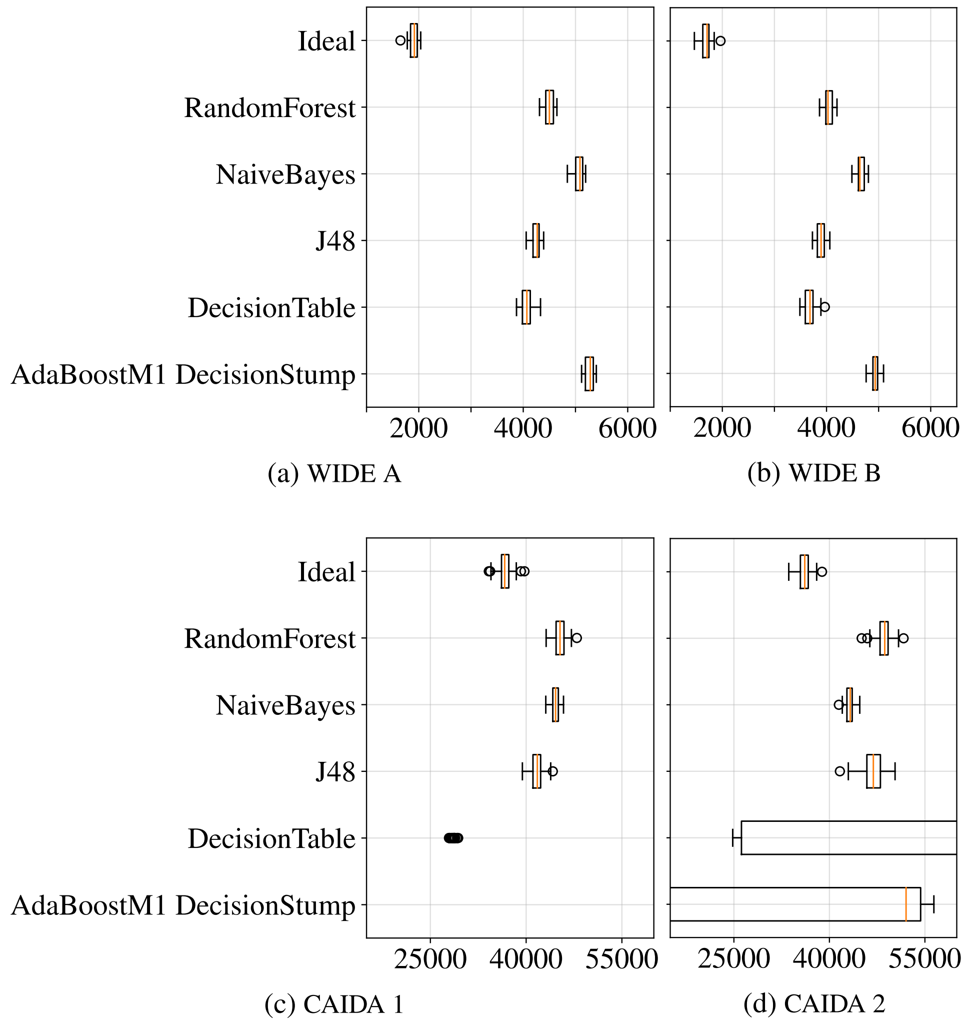


Figure 6.10: Mean table occupation. One major limitation is the number of rules in the hardware table. J48 requires the lowest number of table entries. On the other hand the overhead compared to the ideal case is quite large for all learning schemes.

offloading for large flows and consequently can treat the lion's share of the total data rate in hardware using few matching rules. As the machine learning algorithms are imperfect, we have to either sacrifice data rate or use a larger hardware table.

For this evaluation we made the simplifying assumption that the flow can be offloaded from the first packet. As we can not always sense the end of a flow, we always assumed that one rule has to be kept in the hardware table for additional three seconds after the last packet. This matches the soft-timeout mechanism used in OpenFlow devices, but could also be used for other acceleration technologies e.g. using P4. In Figure 6.10 we present the mean table occupation for the offloading solution, Figure 6.9 shows the offloaded data rate. Specifically for the CAIDA data sets more than 80% and for the MAWI data sets even more than 90% of the data rate can be handled in hardware. On the other hand a fairly large hardware table is

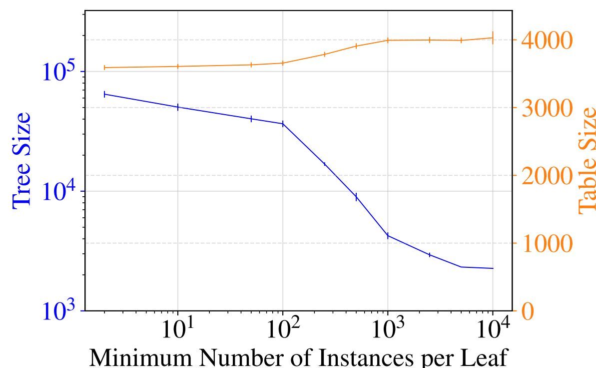


Figure 6.11: Decision tree size and table size when increasing the minimum number of instances per leaf. The Figure shows the results using WIDE B data set. A reduction of the tree by one magnitude only increases the table size necessary by 12.5 %.

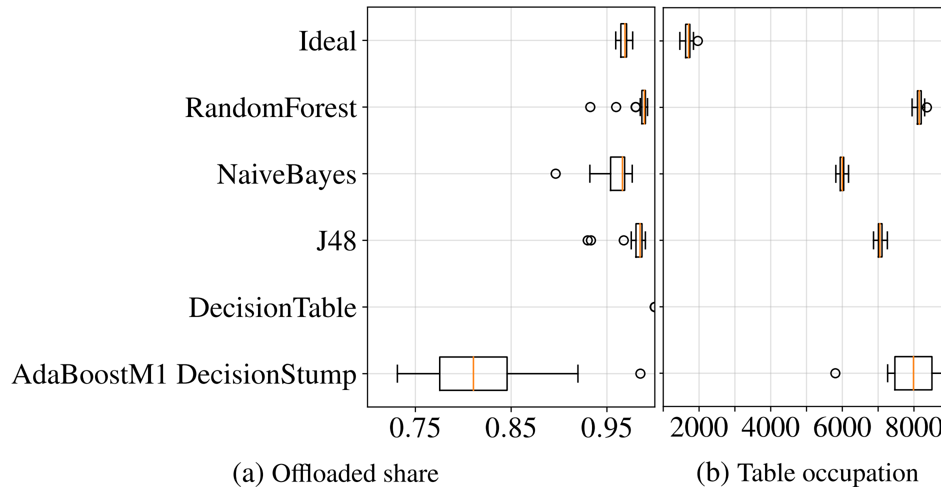
necessary, 8000 entries in the MAWI case and 45000 in the CAIDA case. This corresponds to roughly 20 % of all flows handled in hardware. This is surely a significant number, but as regular Binary Content Addressable Memory (BCAM) can be used instead of expensive Ternary Content Addressable Memory (TCAM) it is still feasible with the hardware available today.

6.4.6 Classification Complexity

From the results shown before, we can see that especially the tree algorithms, J48 and RandomForest, are always in the group of the best performing algorithms. This is why we want discuss the aspect of the classification complexity. J48 builds a decision tree which makes a fast classification possible. RandomForest is more complex in general as multiple trees are evaluated and weighted. As RandomForest does not show a superior performance, J48 seems to be a better choice. The other algorithms require the evaluation of tables (NaiveBayes and DecisionTables). This means their classification process is more complex in general.

In case of J48, the complexity of the decision depends on the depth of the leaves and the degree of the decision nodes. As this metric is not easy to tackle we evaluated the decision tree of the J48 learned model with the tree size. The tree size is the number of all nodes in the tree.

We reduced the tree size (and consequently the decision complexity) by allowing only leafs in the tree that are matching a minimum number of weighted training instances. Figure 6.11 shows the tree size on the left axis and the necessary table size on the right axis for a growing number of minimum instances per leaf. As the main effect of a modification in the learning algorithm is visible in the necessary table size only this metric is shown here. The results show that a reduction of the tree by one magnitude increases the table size necessary by 12.5 %. We



(a) Offloaded share (b) Table occupation

Figure 6.12: Temporal Stability: Offloading performance when using the Model trained with WIDE A data set for classifying WIDE B data set. DecisionTable, NaiveBayes and boosted NaiveBayes perform worse in this scenario. J48 and RandomForest algorithm show less degradation. The offloaded rate is still in the same range as for the ideal decision, the table occupation is ~20 % higher when compared to the model learned by cross validation.

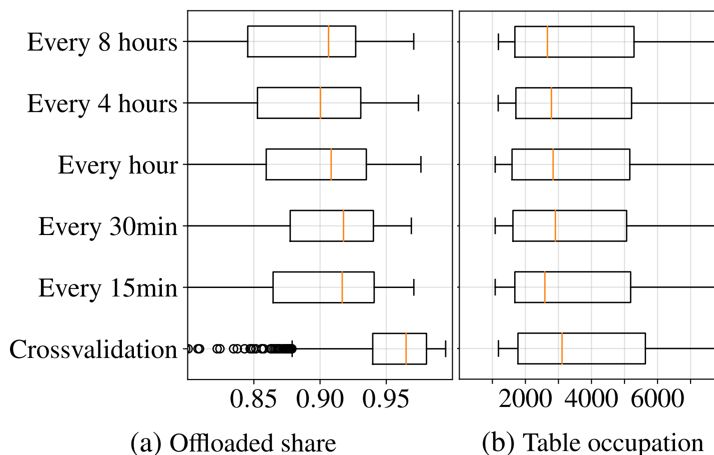
can therefore reduce the complexity significantly while only increasing the necessary table size gradually. Nevertheless in a deployment scenario it could be more pressing to reduce the table size while scarifying the decision complexity.

6.4.7 Temporal stability

In this work we are using an offline learning approach. One question that is arising is how stable the model is regarding a longer period of time between the time the training set was recorded and the time the model is used. Figure 6.12 shows the results in this case: We have used the WIDE A data set for training the model and applied it then to the WIDE B data set. We again stratified the test set to end up with results that can be compared. This way the test sets are the same as in Figure 6.10b, only the training sets differ.

The results show that the offloaded bit rate does not change much with this delay between training and classification. Only the offloaded bit rate using DecisionTable classification is clearly lower, this can be explained by over fitting the training data. Both tree algorithms, J48 and RandomForest, show a slightly higher offloaded rate when compared to NaiveBayes.

A more in deep view can be gained if we apply the approach to a longer trace. Thus we also applied it to the WIDE IX-24h data set and used different training intervals. Figure 6.13 shows the results for different training intervals, e.g., every 15min means that the model is retrained every 15 minutes. We use always data from 15 minutes for training. The results show that the performance is slightly reduced due to the time shift. A more frequent training



(a) Offloaded share (b) Table occupation

Figure 6.13: Temporal Stability: Offloading performance for the WIDE 24h data set and J48. Performance is not impacted largely when different learning intervals are used.

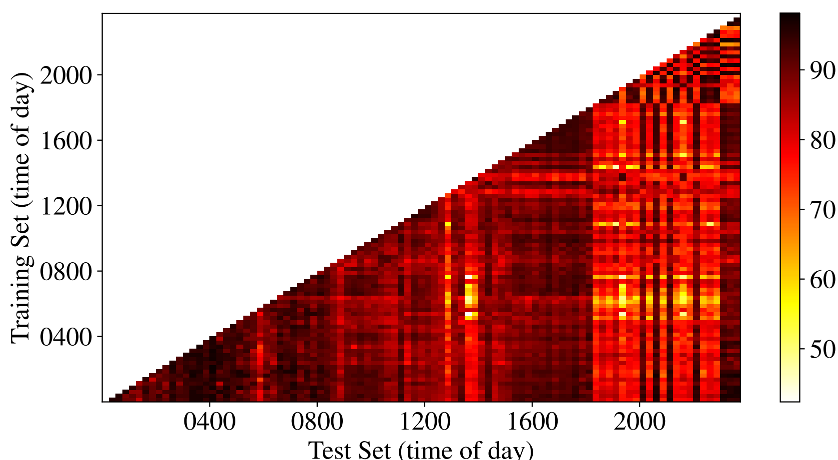


Figure 6.14: Temporal Stability: Offloading share for the WIDE 24h data set in a heat map. This view on the results shows that some times of the day have a different traffic pattern than others.

does not yield significantly better results. Further, the spread of both offloading share and table occupation is quite large.

We see the reason for this result in Figure 6.14. For this figure we split the data into 15 minutes chunks. The figure shows the mean offloading share of all combinations of training and test sets. From the figure it can be seen that there are more and less challenging times of the day. While the performance in the morning (until 12:00 pm) and in the night (after 23:00 pm) is above 90% for most of the training sets, it is different for a short interval around 13:00 pm and especially in the evening after 18:00 pm. This indicates that the traffic patterns change at this time of the day after work time. It should be noted that the pattern reverts to its normal behavior after 23:00 as can be seen by the high offloading shares even with training sets in the early morning. On the other hand, training more frequently does not significantly

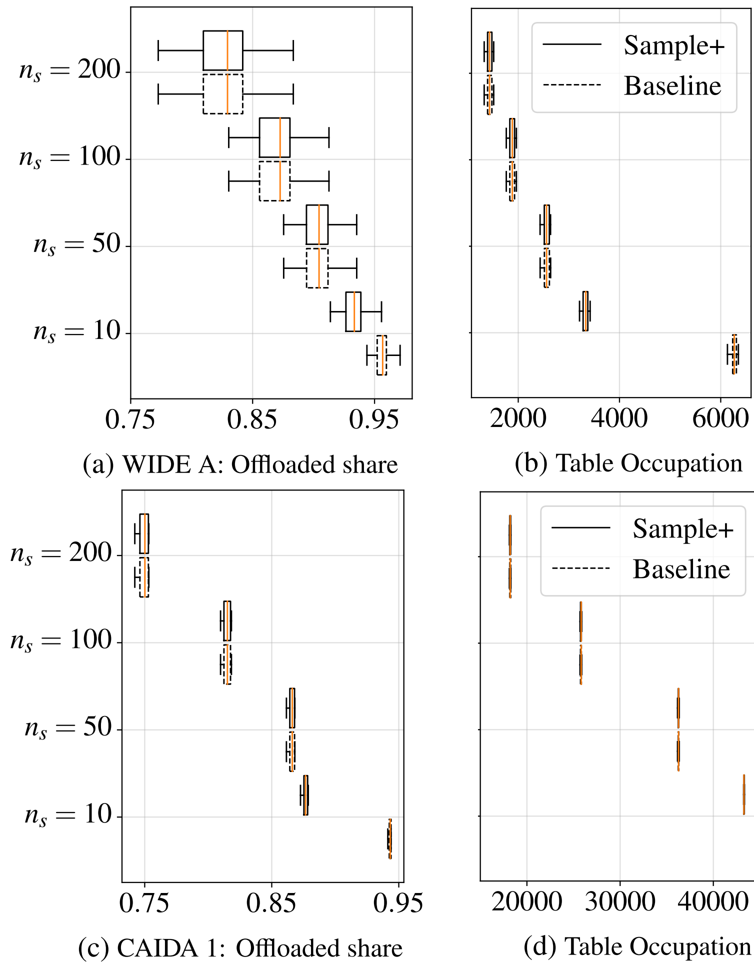


Figure 6.15: Offloading using Sampling: For lower sampling rates all flows that belong to sampled packets are offloaded, thus Sample+ and Baseline algorithm perform equally. Only for higher rates we can see that sample+ is much more economically regarding table occupation.

improve the performance, as the pattern is changing quite often at this time of the day (visible by the stripes in the upper right corner of Figure 6.14).

6.5 Evaluation of the sampling approach

In order to evaluate the sampled approach we applied both algorithms described in Section 6.3 for the data sets WIDE A and B and both CAIDA data sets. One main parameter of both algorithm is the sampling rate n_s, we used different settings for n_s, starting with moderate sampling of $n_s = 200$ up to a high sampling rate with $n_s = 10$. This is the only parameter of the baseline algorithm, from runs with this algorithm we can derive how many table entries are necessary in the restricted case. We have chosen a table size of 4000 for the data set WIDE A and 45000 for CAIDA A, this allows for a high offloading rate in both cases.

Figure 6.15 show the results for both algorithms. It can be seen that for lower rates up to $n_s = 50$ both algorithms have the exact same performance and the table is not filled to the targeted value. This is due to the fact that many flows are not even seen by the offloading algorithms as no packet of these flows is sampled. As the heuristic tries to use the table up to the allowed level it behaves equal to the baseline algorithm. On the other hand it can be seen that the offloaded rate is quite high if compared to the table occupation. This is due to the fact that sampling itself already filters the flows as large flows have a higher probability to be sampled, this directly resembles the fact that elephant flows makeup the major share of the complete traffic.

The downside of the sampling based approach is that the overhead is quite high. While the sampling itself can be done in hardware that is deployed quite often nowadays, the rate that has to be processed by the algorithm is still high. E.g. if the share of the offloaded traffic should reach 90% it is necessary to set the sampling rate to $n_s = 50$, this results in a packet rate of $110 \cdot 10^3$ packets per second (pps) for the WIDE A data set. Even though it is feasible to process such a number, it is demanding and can be infeasible for scenarios with higher rates.

6.6 Summary and Discussion

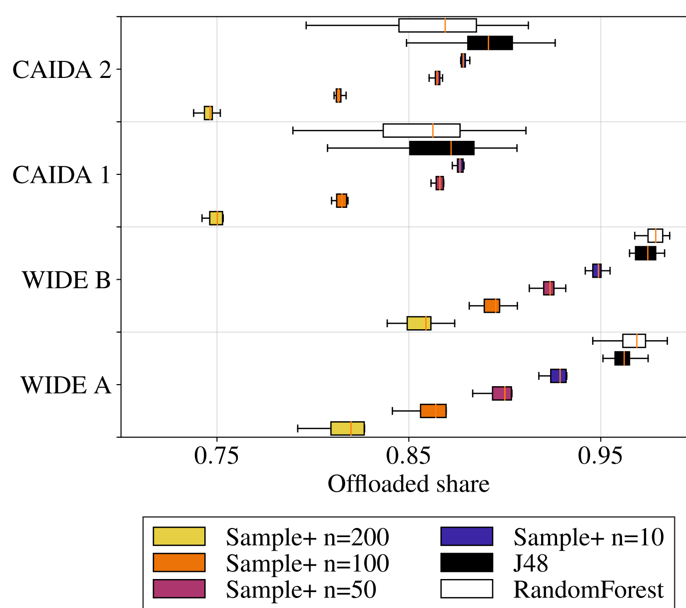


Figure 6.16: Overview of the offloaded share for all algorithms and data sets. In order to reach the performance of the Machine Learning algorithms a high sampling rate is necessary.

In this section we are summarizing the findings and assess them in context of the NFV acceleration approach.

In Section 6.2 we are introducing a classification model, that is used to build a model for the classification of the flows in small flows and large flows used for offloading. We present a monitoring approach for gathering the necessary training data. We argue that a combined approach that uses statistics from a NFV based tailored statistics network function for the first packet size and a conventional monitoring system using OpenFlow statistics or sFlow can be used. Furthermore, we introduce the features that are used for training and how they are preprocessed. The preprocessing is based on domain knowledge and is an important step for a good classification performance. It uses the parameters offloading threshold Θ_F and the minimum frequency f of the nominal features. We tuned the parameters independent of each other using the data sets CAIDA A and WIDE A. The results are similar for both data sets even though the data sets are quite different, as CAIDA A has much more data rate and concurrent flows. Further, we use the same parameters for the data sets CAIDA B, WIDE B and WIDE IX-24 with good results. This implies that the parameters hold for typical internet traffic. However, for different network conditions such as, e.g. an industrial network, the optimal parameters might be different. Nevertheless we argue that the pipeline itself can be applied to other conditions with minor or no changes.

In Section 6.4 we apply different machine learning algorithms using the presented classification system. We rely on publicly available data sets. Figures 6.8, 6.9 and 6.10 show the results. It can be seen that DecisionTable and boosted DecisionStump algorithm do not provide a reliable performance with the investigated data sets. Therefore we argue that they are not suitable for the presented application. Overall RandomForest and J48 achieve high offloaded rates with slightly lower table occupation. We argue that J48 has a lower complexity than RandomForest and analyze this property more deeply for J48 in Section 6.4.6.

The model is employed in an SDN Controller for classifying new flows with the first packet and flows can be offloaded using an SDN controller. As the model is network specific and not static it has to be retrained regularly. In Section 6.4.7 we have shown that a daily retraining is sufficient for the evaluated network. As the old model can still be used while the new one is trained we do not have any interruptions of the acceleration.

Section 6.3 introduces two sampling based approaches that can be used alternatively for the offloading decision. The baseline algorithm always decides for offloading. Due to the sampling not all flows are offloaded, as it can happen that all packets of a flow are not part of the sampled subset. The table restricted approach, *Sample+* takes a maximum allowed table size as input and keeps the number of rules below this threshold.

Figure 6.16 shows an overview of the results for both approaches. We have chosen the best performing machine learning algorithms namely J48 and RandomForest here. It can be seen that the offloaded share for the MAWI data sets is significantly higher compared to the CAIDA data sets. This indicates that large flows have a smaller share in this network. In general both presented learning algorithms perform better and especially more stable for the MAWI sets. We can follow that the algorithms are more suitable for smaller scale networks like campus networks. All results support the finding that the sampling based algorithms require a large sampling rate to reach the performance of the Machine learning algorithms with sampling based algorithms: For the MAWI data sets we do not even meet the performance with a sampling rate of 1:10, for the CAIDA data sets still a sampling rate of 1:50 is necessary. A high sampling rate requires significant compute resources as all the sampled packets have to be processed. That is with the presented sampling rate and the MAWI A data set a packet rate of $110 \cdot 10^3$ pps. Even though the sampling rate for the CAIDA traces can be lower, the resulting packet rate that must be processed by the sampling approach is higher: $1 \cdot 10^6$ pps.

The traces differ a lot in terms of table occupation. The traces that were retrieved from the WIDE traffic archive require table sizes in the range of a few thousand entries. In contrast to that, the traces retrieved from CAIDA require much bigger tables of several ten thousand entries. This observation holds for both approaches as can be seen from Figure 6.10 and Figure 6.15. This difference in the order of magnitude stems from the fact, that the WIDE traces are gathered from a 1G uplink, while the CAIDA traces are gathered at a 10G backbone link. This indicates that different hardware, in terms of size of the hardware table, is necessary, depending on how aggregated a link is. On the one hand, highly aggregated links require acceleration hardware providing large tables. On the other hand less aggregated like those in the WIDE case links suffice with smaller tables.

In this chapter, we explore the capabilities of different algorithms for deciding if a flow should be offloaded to hardware or not. We consider two fundamentally different approaches: First offloading with the first packet of a flow using machine learning. The first packet is presented by the connected VNF to the algorithm. Secondly offloading with sampling, where all packets are sampled and the sampled packets are used for the decision.

The results show that the lion's share of the data rate can be handled in hardware using our approaches. The drawback is that the necessary hardware table is comparably big, as a fairly big number of flows are classified for offloading falsely. As regular BCAM can be used for matching, the approach is feasible nevertheless.

The results show that the J48 algorithm has the best properties of all the investigated machine learning algorithms. It combines low table occupation with a high offloaded data

rate and additionally has a low complexity of the classification. On the other hand other algorithms like RandomForest and NaiveBayes are only slightly worse and could also be used.

The sampling based approach can also reach a quite high offloaded share, though it needs a high sampling rate to meet the performance of the machine learning approach.

Chapter 7

Offloading Virtual Network Functions to Smart-NICs

In the previous chapter, we introduced a method that increases the performance of the architecture using Software-Defined Networking (SDN) offloading. This reduces overheads as fewer traffic has to be served by the packet filters in the Network Function Virtualization Infrastructure (NFVI). On the other hand, SDN offloading is only possible if stateless filtering is sufficiently secure.

Other network function such as stateful network functions can not be implemented in SDN, due to lack of support for stateful operations. To overcome this we propose in our architecture the use of Network Function Virtualization (NFV).

Figure 7.1 shows how higher layer filtering is provided in the security architecture. The network traffic is routed via the NFVI where the software packet filters reside. An important feature of NFV is the support of seamless scaling of the Virtualized Network Functions (VNFs) depending on the traffic demand. Thus the packets have to be distributed between the different instances using a load balancer. In addition, it is necessary that all packets of one connection are forwarded to the same instance, in case of stateful or application layer VNFs. E.g., an application layer firewall needs all packets, in order to reassemble application layer content, that is split over multiple packets. A stateful load balancer has to keep track of connection such that all packets of one connection are forwarded to the same VNF.

The use of software packet processing is envisioned to provide higher layer filters. In contrast to that, packet processing hardware can be more optimized for a certain task. Therefore performance boosts might be missed. In this chapter we show Network Interface Card (NIC)-offloading, an implementation which aims to flexibly combine hardware with software packet processing. For our approach, we do not require specialized hardware, but only utilize NICs already available on many servers today. We evaluated the stateful NIC offloading approach

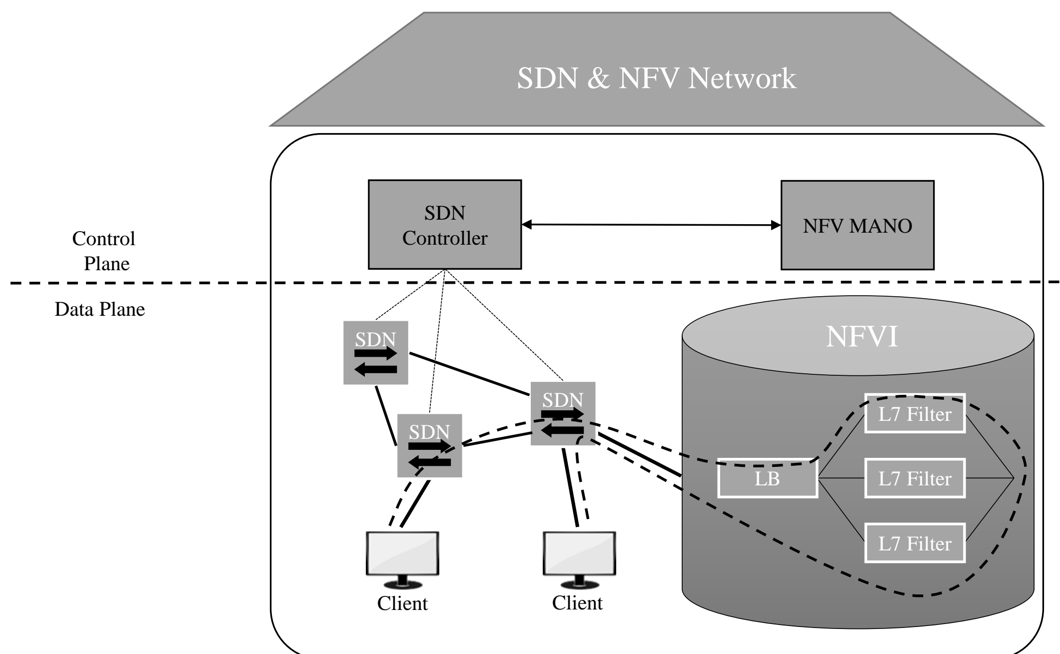


Figure 7.1: Application layer filtering in the SDN/NFV architecture. The traffic is routed through the NFVI where application layer filters (L7 Filter) check the traffic. In order to provide scaling and performance the traffic is distributed between multiple instances using a Load Balancer (LB).

using the use case of a load balancer. Nevertheless, we argue that the performance boosts exists for many network functions that store connection dependent state.

The core idea is as follows: Every stateful network function has to keep a table storing the state of connections. Using software the network function matches incoming packets against its software table and forwards the packet to the endpoint responsible for this connection. We utilize the hardware table of NICs by offloading the software table to this hardware table. This aims to reduce load on the CPU, since the packet matching is offloaded to fast hardware. Note that the NIC cannot fully takeover the processing of packets, since it is not able to perform packet forwarding. Our evaluation results indicate that a reduction of the load on the CPU can increase the maximum achievable throughput significantly.

This chapter is based on [9] that presented the NIC offloading approach first.

[9] R. Durner, A. Varasteh, M. Stephan, C. Mas Machuca, and W. Kellerer. "HNLB: Utilizing Hardware Matching Capabilities of NICs for Offloading Stateful Load Balancers." In: *2019 IEEE International Conference on Communications (ICC'19)*. 2019.

This chapter is structured as follows. Section 7.1 introduces related work on NIC offloading. Section 7.2 describes background on NIC offloading and stateful load balancing. Afterwards Section 7.3 introduces the implementation of the hybrid solution. In Section 7.4

a method for measuring the utilization of the approach is shown. Section 7.5 presents the results and finally Section 7.6 discusses the findings.

7.1 Related Work

In this section we give an overview on related work in the fields of software network functions, stateful hardware network functions and also NIC-Offloading.

7.1.1 Software network functions

With the introduction of network softwarization and NFV, many network functions have started to being deployed as software instances on commodity servers [141]. Thus large efforts have been undertaken in order to meet the requirements of high data rate networks.

An early concept of software network functions and network softwarization in general is the click modular router [142]. The authors present an C++ based framework that facilitates the development of network functions. In order to enhance performance, the original click router was using a Linux kernel extension. More recently an DPDK back-end was added to the framework in order to provide up to date performance.

An important performance issue of software packet processing is that network stacks of current operating systems can not provide high throughput. Thus frameworks have been implemented to increase network performance:

Netmap [143] provides higher performance while being tightly integrated into the Linux kernel in contrast to other frameworks that are implemented fully in user space. This can provide safety by providing checks on the data that was provided by the user space program.

The Data Plane Development Kit (DPDK) [99] is a framework originally developed by Intel. Besides a basic implementation for sending and receiving packets it also provides other features such as a hash table or an API for the Intel Flow Director. DPDK applications are running in user space only and the NIC can not be used for applications with the normal OS networking stack.

Ntop develops a high packet rate framework called PF_RING [144]. The main focus lies on network filtering and analysis such as packet captures. Furthermore user space NIC drivers are provided called PF_RING ZC. While being conceptually similar to DPDK, PF_RING ZC has worse support for NIC hardware.

PacketShader [145] utilizes GPUs for packet processing. The authors show that the performance can be significantly increased if a large number of packets can be processed in

parallel. This is due to the fact that GPUs have a high number of compute units optimized for parallel processing.

Summarizing this, several frameworks that improve performance of software packet processing exist. Nevertheless, packet processing hardware is known to provide superior performance for some use cases. This lead to novel hardware and especially hybrid approaches.

7.1.2 Stateful Hardware and Hybrid Network Functions

Despite low cost and high availability/flexibility of software network functions, they suffer from high latency and high resource consumption. These issues can be tackled by developing network functions utilizing hardware. A candidate hardware to enhance the performance, is ASICs in network switches. A popular approach is presented by Bianchi et. al. [146]. The authors extend OpenFlow by adding finite state machines to the API. Like this the SDN switches are also able to implement stateful network functions such as a stateful firewall. One drawback is that the hardware table size is limited and thus also the number of parallel flows that can be supported.

Authors in [147] propose Duet, a hybrid load balancer that proposes to move the load balancing function to existing hardware network switches (at no extra cost). In fact, network switches perform traffic splitting (using ECMP) and packet encapsulation tasks of load balancing. Notably, the state table is stored in switching ASICs, which can contribute to reduce the latency to several microseconds. In addition to hardware load balancing, they developed a small software load balancer to act as a backstop, and to provide high availability and flexibility.

Similar to Duet, SilkRoad [148] also leverages features of programmable ASICs to build a load balancer for data center networks. In contrast to Duet [147] that stores the (not stateful) table in switching ASICs, SilkRoad utilizes SRAM in ASICs and like this stores per-connection state at ASICs. In this way, in addition to providing high throughput and low latency, SilkRoad ensures per-connection consistency during DIP pool changes. However SilkRoad requires specialized switching hardware usually not available in DataCenters.

All solutions above claim to provide high throughput and sufficient flexibility. One disadvantage of the approaches presented above is that they require specialized hardware, like e.g. stateful OpenFlow devices for [146] or P4 switches for [148]. NICs can be an alternative that is easier to deploy.

7.1.3 NIC Offloading

As an example for using NIC offloading, authors in [149] offloaded the firewall logic to a NIC. They utilized 5-tuple filtering in the NIC and showed that a offloading firewall to NIC can improve both CPU utilization as well as packet throughput.

Further, authors in [150] proposed NDN-NIC, a network interface card for performing name-based filtering on the NIC. In this approach, names are maintained in a bloom filter, which is used as a reference for filtering the incoming packets on NIC. Authors showed that this approach is able to reduce CPU overhead and energy consumption.

In a recent work, Microsoft researchers [151] presented a NIC Offloading mechanism which uses custom NICs with a built-in Field Programmable Gate Array (FPGA). Their solution is used in Microsoft's Azure cloud to reduce the CPU load caused by networking. Moreover, their approach uses an offloading method that handles the first packet in software. In contrast to our work, a more feature-rich FPGA is utilized, while we are only using existing capabilities of the NIC.

However, to the best of our knowledge, we are the first work that present a hybrid hardware-software load balancer in which NIC offloading (Intel Flow Director [152]) is utilized for offloading matching to hardware.

7.2 Background

In this section, we provide background on stateful load balancers and introduce briefly Intel's FlowDirector technology.

7.2.1 Stateful Load Balancers

As we evaluate the performance of the NIC offloading approach for a stateful load balancer we present details about this concept in the following.

The main goal of a load balancer is to distribute the load (i.e., the packets) between several back-end servers (service instances) that deliver the actual service. The packets can be either distributed stateless (e.g., using round robin) or stateful (packets belonging to the same connection are always delivered to the same back-end server). Further, existing load balancers work on different layers, e.g. layer 4 (L4) and layer 7.

In this work, we utilize connection table offloading on NICs that support header matching. Thus, we focus on the design and evaluation of a stateful L4 load balancer.

Two different sets of IPs exist in the load balancer concept: *i)* Virtual IP (VIP), and *ii)* Direct IP (DIP). VIPs are the IPs that the users are addressing. They can be seen as the service

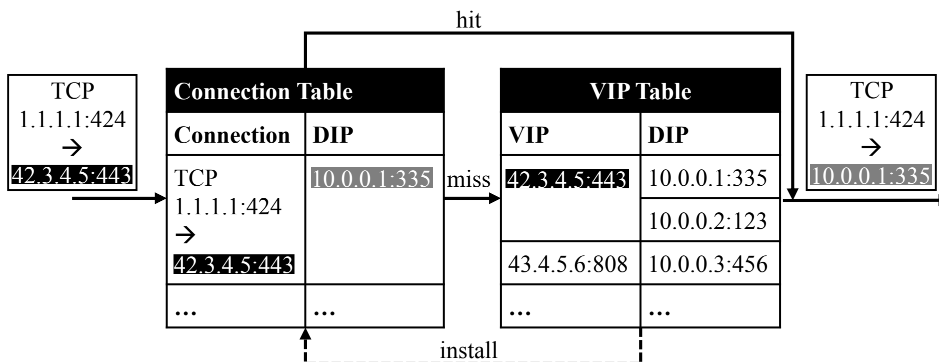


Figure 7.2: Basic working principle of a stateful L4 load balancer. The VIP of the served packet is highlighted in black, the DIP is highlighted in gray.

addresses. On the other hand, DIPs are the addresses that belong to the actual instances delivering the service. Consequently, the load balancer distributes the packets destined to one VIP between several DIPs.

Accordingly, in order to guarantee connection consistency, a stateful L4 load balancer has to maintain two tables:

I) The VIP table: which contains the VIPs of all services and the respective active DIPs belonging to this service.

II) The connection table: which holds the mapping of active connections to their designated DIPs.

The basic working principle of a stateful L4 load balancer is shown in Figure 7.2: Incoming packets are first checked against the Connection Table. If an entry already exists (*hit*), the packet header is rewritten according to the entry, and the packet is forwarded. If the 5-tuple of the packet does not match any entry (*miss*) in the Connection Table, a new DIP for the corresponding VIP is selected from the VIP table. This selection can be based on round robin algorithm or according to the current load of the DIPs. Afterwards, the packet is rewritten using the chosen mapping and forwarded. Finally, the new connection is installed in the Connection Table (*install*).

The figure shows an incoming packet with the 5-tuple: TCP, 1.1.1.1, 424, 42.3.4.5, 443, that directly matches an entry in the Connection Table (hit). Thus the packet's destination is rewritten to the DIP 10.0.0.1 with destination port 335.

7.2.2 Intel Flow Director

In order to support multi-core packet processing, packets have to be distributed among the cores. One technique that can achieve this is called Receive Side Scaling (RSS). RSS firstly

computes the hash of the packet header. It then forwards the packets according to the hash value to one of the NIC queues. Each core is then processing packets of one queue. This procedure constitutes a stateless load balancer. The Flow Director technology is originally designed to expand the RSS functionality, by not only performing load balancing, but also forwarding the incoming packets to the core where the related application is running. Additionally, the match table can be programmed using an API. In the hybrid approach, we utilize this functionality to offload the connection table of the load balancer to the NIC.

Depending on the configuration of the NIC, Flow Director can support between around 2000 and 8000 table entries [153]. As a drawback the NIC's memory is shared between the Flow Director and the receive buffer. Therefore, as the number rules is increased, the receive buffer is shrinked.

We measured the latency of a Flow Director enabled echo software for different number of filters. The absolute maximum latency that occurred in our tests increased from 95 μs for no rules to 105 μs for 8000 rules. These results clearly show that the induced latency by Flow Director is marginal, even for 8000 rules.

7.3 Implementation

In this section we present how the Hybrid NIC Offloading Load Balancer (HNLB) and the Software Load Balancer (SLB) is implemented in detail.

As mentioned before, we use the matching capabilities of the NIC to increase the throughput of HNLB. Figure 7.3 shows the implementation of HNLB, supporting hardware table offloading. Accordingly, the load balancing steps can be presented as follows:

(1) Packets from new or unseen connections do not match any rule in the Flow Director table. Therefore, they are forwarded to the default queue of the NIC, i.e., Queue 0.

(2) The developed software in HNLB polls all queues in a round robin manner for packets. As packets are processed in bursts, it cannot be guaranteed that the second packet of one connection is already matched by the hardware table. Therefore, packets' headers from the default queue are hashed and checked against the software connection table.

(3) If the packets do not match any entry in the connection table, a DIP is chosen from the VIP table as usual.

(4) The resulting mapping is installed in the software connection table (4a) and the hardware table (4b).

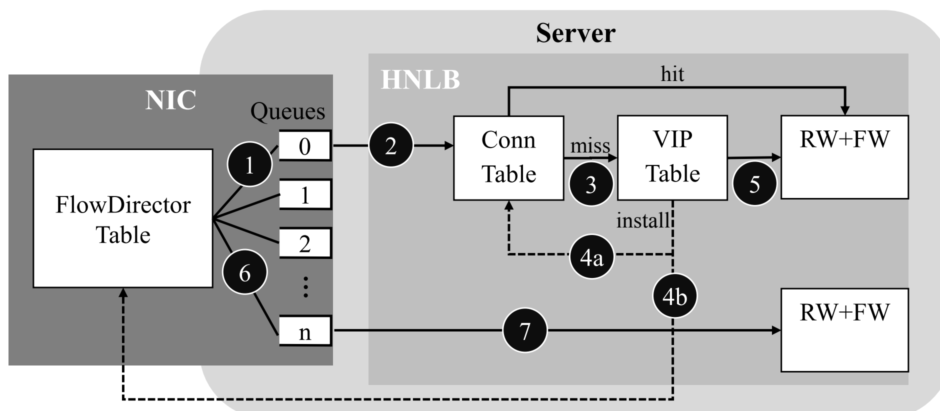


Figure 7.3: Load balancer with hardware table offloading: Packets from unseen connections are put into queue 0 (1) and polled from the HNLB Software (2). These packets do not match any connection in the software connection table (3), therefore a DIP is chosen from the VIP table and the mapping is installed in software (4a) and hardware (4b). The packet header is rewritten and forwarded (5). Subsequent packets of this connection are put in the queue of the corresponding DIP (6). When they are polled from the queue the matching for the rewriting is encoded in the queue number already (7).

(5) The packet is rewritten according to the selected VIP-to-DIP mapping and put into the output buffer of the NIC, where it is forwarded.

(6) Subsequent packets are matching the new rule in the hardware table and put into the queue that encodes the DIP.

(7) The header of these packets are then rewritten in software accordingly and are put to the output buffer of the NIC to forward them.

The implementation is based on DPDK [99].

For comparison, we also developed a SLB. To perform this, we use the implementation from Figure 7.3 without step 4b. This implementation does not use any hardware offloading, since it is realized only in software. As a result, no hardware table is used, and only the default queue (Queue 0) is polled.

7.4 Assessing utilization

In this section, we introduce a utilization metric and a novel algorithm to determine it.

One of the techniques that DPDK uses to increase the packet throughput is the change from an interrupt-based packet retrieval to a polling-based packet retrieval. For instance, if a packet arrives on the NIC, the CPU is interrupted and the packet is then copied and processed by the OS. However, DPDK does not use interrupts, instead, it checks for packets at the NIC, processes these packets and then checks again for packets in an infinite loop. As a

consequence, the conventional CPU utilization metric does not reveal the load of the system as the CPU is always fully utilized by the loop. In fact, our proposed metric can be useful to monitor the system and also to scale up the load-balancer in high-load conditions.

To overcome the challenges the metric shall fulfill the following conditions for a constant number of concurrent connections and packet sizes:

(i) The utilization shall reach 100 %, when the maximum possible packet rate is being processed and there is no packet loss.

(ii) The utilization with no traffic (packet rate of 0 pps) shall be 0.

(iii) Otherwise (when conditions (i) and (ii) are not met), it should be linear with the packet rate.

To form our metric, we adapt the algorithm presented in Section 5.3.1.4. The gathering of the metrics works as follows:

The cycles counter is read before every iteration of the loop, the cycles spent in the last iteration are computed in line 3 and added up to the REF counter. If packets are processed, these CPU cycles that are used for processing packets are summed up in line 13 and are called OPS. If we output REF and OPS, we can compute the utilization in some time interval as:

$$util = \frac{OPS}{REF}$$

Therefore, if packets are processed in every iteration, we have $util = 100\%$. Otherwise, with busy waiting iterations present, $util$ would be smaller than 100%. However, this simple definition does not take into account that packets are processed in bursts: If every iteration of the loop would process exactly one packet, the resulting utilization is 100%, although the system can cope with higher rates. In fact,we observe these effects in both cases. For instance, for the offloading case, with more used queues the queues are empty less often when polled. Therefore $util$ overestimates the utilization. As a result, $util$ can fulfill conditions (i) and (ii), but fails for condition (iii).

In order to resolve this, we also count the number of processed bursts n_b and the number of processed packets n_p in the lines 7 and 8. This leads to an improved utilization metric presented as below:

$$util_{+} = util \cdot (1 + \frac{n_p}{n_b \cdot B})/2$$

where B is the maximum feasible burst size. Essentially, $util_{+}$ weights $util$ with the mean burst utilization.

From our experiments, we can see that $B = 32$ in the SLB case, which is exactly the maximum number of packets we read in one loop iteration from a queue. For more than one

Algorithm 6: Receiving-Loop

```
1  while true do
2      cpu_cycles_before = get_cycles();
3      REF+= (cpu_cycles_before - cpu_cycles_last);
4      cpu_cycles_last = cpu_cycles_before;
5      number_rx_packets = recieve_function();
6      if number_rx_packets > 0 then
7          n_p += number_rx_packets;
8          n_b += 1;
9          ...
10         Packet processing
11         ...
12         cpu_cycles_proc = get_cycles();
13         OPS += (cpu_cycles_proc - cpu_cycles_before);
14     end
15 end
```

queue, i.e. HNLB, this value can never be reached without losing packets. Therefore, we set $B = 16$ and adjust the $util_+$ equation slightly:

$$util_+ = util \cdot min(1, (1 + \frac{n_p}{n_b \cdot B})/2)$$

In Section 7.5.2, we show how the utilization evolves over the packet rate and how $util_-$ differs from $util_+$.

7.5 Performance Evaluation

In this section, we compare HNLB with the SLB approach as described in Section 7.3. We focus on evaluating performance by measuring two metrics: *i)* maximum throughput (without loss), and *ii)* our novel utilization metric defined in Section 7.4. Maximum throughput directly relates to the resource consumption of the load balancer, as more servers are needed, if less throughput per core is achievable. The utilization on the other hand, gives more insights into the reasons of the throughput gain. Additionally, it is necessary for scaling as packet loss should be avoided in real setups: The utilization can be used as input metric for a scaling solution. Service instances can be scaled up/down to improve resource efficiency depending on the utilization.

All results only refer to a single physical CPU core without exploiting parallelisms. All measurements were repeated 30 times and confidence intervals were derived. Since all results are very stable, the confidence intervals are not visible and, consequently, not shown. The

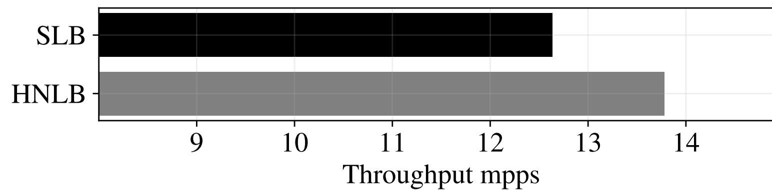


Figure 7.4: Best Case Packet Throughput for a paket size of 64 Byte, a single flow and a single DIP.

results are shown with respect to the sending rate of the traffic generator in million packets per second (mpps).

Our used testbed includes a Desktop PC which uses DPDK-Pktgen [154] for the load generation and a server equipped with 2x AMD EPYC 7301 x86 processor. Each of the processors has 16 Cores and 64MB L3 Cache. The turbo feature was disabled as it mainly increases the performance of single-threaded applications, which is not a realistic use case on such a server. Simultanous Multithreading (SMT) support was disabled as well to avoid possible contention and interference effects, which are inherent of this technology. Moreover, both devices are equipped with Intel X550 Dual-Port 10G Ethernet interface cards. We note that only one port of both NICs is used for the measurements.

7.5.1 Maximum Throughput

Let us first explore the maximum achievable throughput in terms of packet rate. To perform this experiment, we use a single flow and a single DIP.

As it is depicted in Figure 7.4, both solutions can achieve more than 12 mpps of packet rate. In detail HNLB can achieve packet rates up to more than 13 mpps until it shows some loss (not shown in the figure). However, in SLB case, packet loss starts when the packet rate reaches around 12 mpps. This resembles the overhead of reading and hashing the header in the SLB case, that is not necessary in the HNLB implementation.

The number of queues in HNLB is equal to the number of DIPs per core, i.e.only one queue is used to explore the best case. For the following experiments, we use 10 queues.

In our next experiment, we investigate the throughput for different number of concurrent connections (nb_conn). In this case, we use only a packet size equal to 64 Bytes. This experiment is depicted in Figure 7.5. With an increased number of open connections, the application has to access a bigger table. This decreases the locality of reference of the application, such that the processor cannot store the table in the fast on-chip caches anymore; hence, it has to access the main memory more often. As a result, as is expected, higher number of connections reduces the maximum rate for both approaches. For HNLB the throughput decreases from 1 to 100 connections, is constant for 100 to 1000 concurrent connections and

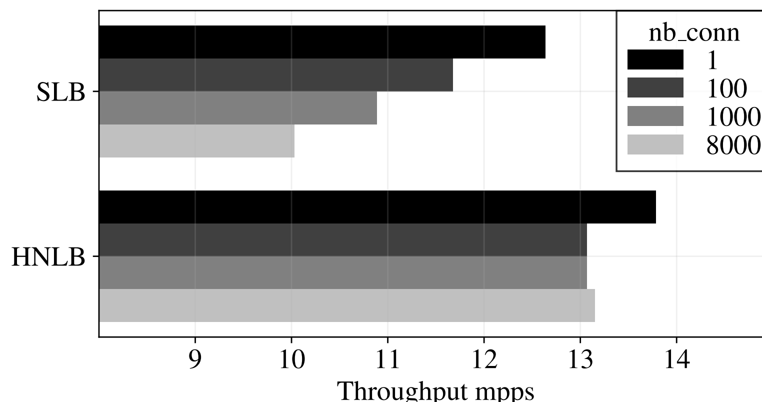


Figure 7.5: Maximum Throughput for a packet size of 64 Byte and different number of concurrent connections.

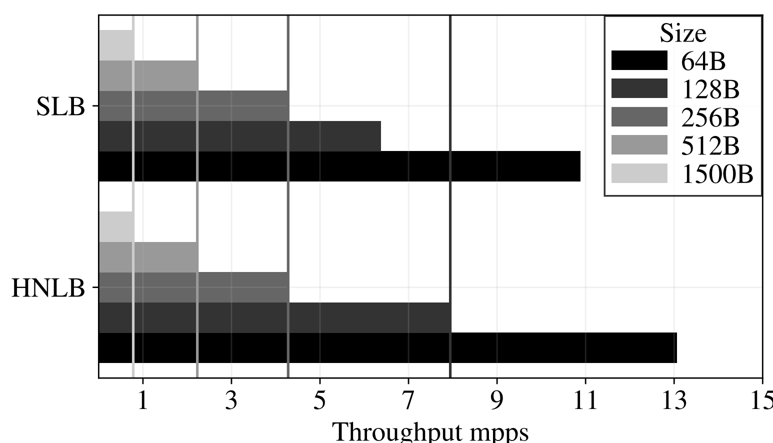


Figure 7.6: Maximum throughput for different packet sizes. Vertical lines represent the maximum offered rate of the traffic generator.

even increases very little for 8000 connections. The SLB approach throughput constantly decreases significantly with increasing number of connections. As a result, the performance gap between both approaches increases with more connections. For example, for 8000 parallel connections, HNLB can serve a 50% higher packet rate than the SLB on one core.

We also evaluate the impact of packet sizes on the throughput in both cases. Figure 7.6 shows the maximum throughput of both approaches, where the vertical lines indicate the maximum rate of the traffic generator. This maximum is not shown for 64 Byte packets as it is out of the scale. In order to get stable results the maximum rate of the generator is a little below the NIC limit.

The results show that both solutions can reach the maximum rate of the traffic generator for packet sizes larger than 128 Bytes without any loss. For a packet size of 128 Bytes the gain of HNLB is still at least 43%, as HNLB can serve the maximum offered rate while SLB is already limited beforehand. The packet rate for larger sizes is limited by the rate of the traffic

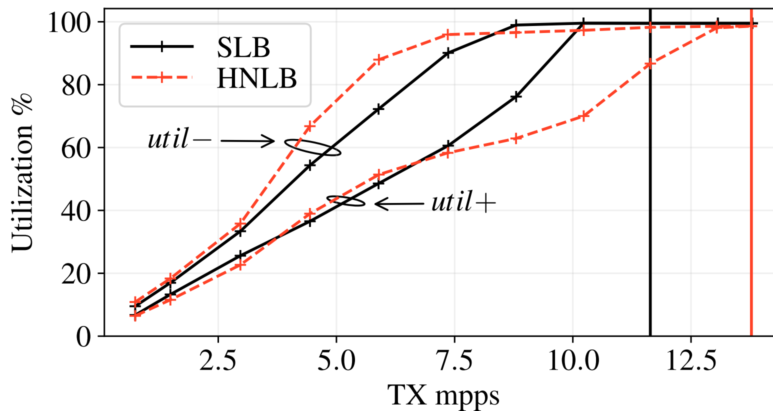


Figure 7.7: Comparison of Utilization metrics for different rates, 1000 concurrent connections and 10 Queues. Vertical lines show the measurement values where the first loss occurs.

generator. We expect that the gain of HNLB will be similar for larger packet sizes. This is supported by the utilization results in the next section.

7.5.2 Utilization

Figure 7.7 shows a comparison of the utilization metrics defined in Section 7.4. The vertical lines denote the rates where loss occurs first. It can be seen that $util_-$ reaches almost 100% already for quite low packet rates, which violates condition *(iii)*. This effect is due to the burst packet processing that is used by DPDK: the overhead of processing one burst is independent of the number of packets in each burst.

Therefore, the HNLB becomes more efficient if the throughput is close to the limit (vertical line). Notably, in this context, being efficient means that we do not waste CPU cycles by processing small bursts. In general, both solutions process bursts. Although the software-only approach polls a single queue, the HNLB approach polls a number of queues. This explains why $util_-$ for HNLB is already very high for approximately 50% of the maximum feasible packet rate.

Especially, we can argue that $util_+$ is a much better metric for HNLB to measure the the utilization. This metric also gives a good indication of load and how much more traffic can possibly be served. Further, it fulfills conditions (i) and (ii) and shows a close to linear behavior (condition(iii)) as well.

Figure 7.8 shows $util_+$ for both approaches and different number of concurrent connections. The vertical lines show the packet rate where the first loss occurs. From the Figure, it can be derived that HNLB outperforms the SLB approach, especially if a higher number of concurrent connections are served. For instance, for 1000 connections and a packet rate of 10 mpps, the SLB has already a utilization of 100%, while HNLB has only a utilization of

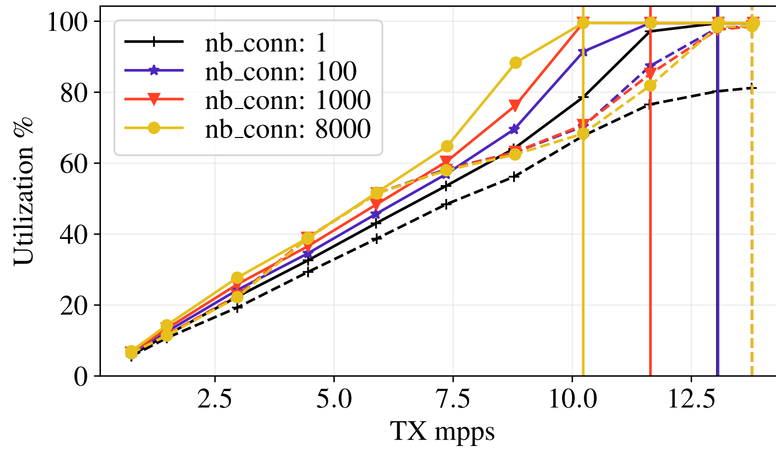


Figure 7.8: $util_+$ for different rates, different number of concurrent connections, a packet size of 64 Byte and 10 Queues. Vertical lines show the measurement values where the first loss occurs. HNLB is marked with dashed lines. SLB is shown with solid lines.

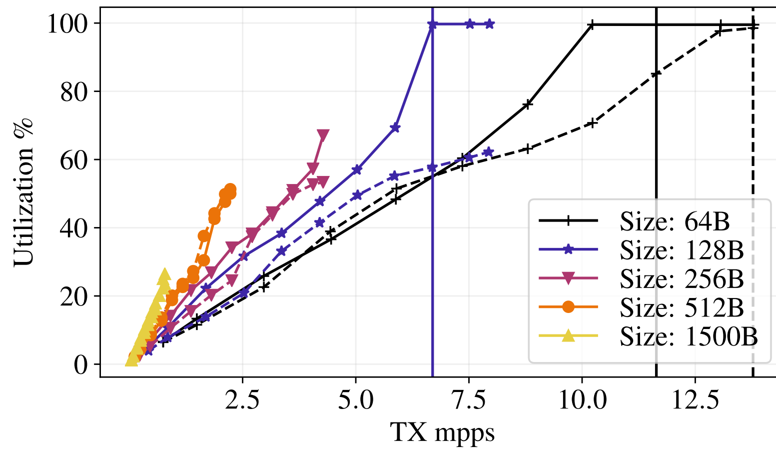


Figure 7.9: $util_+$ for different rates, different packet sizes, 1000 concurrent connections and 10 Queues. Vertical lines show the measurement values where the first loss occurs. HNLB is marked with dashed lines. SLB is shown with solid lines.

70%. Further, we can argue that $util_+$ is a good metric to measure the utilization. It gives a good indication of the load and how much more traffic can possibly be served: For 1000 connections $util_+$ reaches 100% with 10 mpps in the SLB case we and 13 mpps with HNLB. For the SLB case we have already loss with 10 mpps, for HNLB loss starts with the next measurement value. This shows that $util_+$ fulfills conditions (ii) and shows a close to linear behavior (condition(iii)) as well. Further, $util_+$ fulfills condition (i) per definition. $util_+$ with one connection on HNLB is always lower than 100%, in reality this is not an issue as load balancing is not possible for one connection.

As our final experiment, Figure 7.9 shows the utilization for larger packet sizes. Since there is no loss in both cases for packets larger than 128 Bytes, we can observe the scalability with respect to larger packets using $util_+$ only. We know that packets have to be copied to

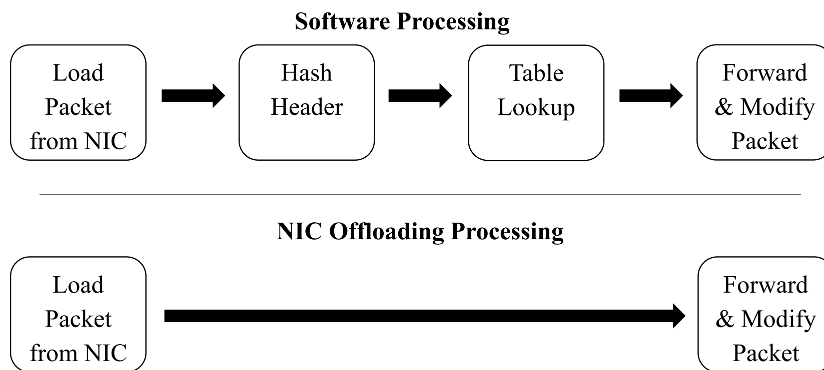


Figure 7.10: Software packet processing without Offloading and the matching capabilities. The flow assumes that a DIP table entry exists which holds for all packets of a connection except the first one.

the main memory for processing and forwarding. Thus, larger packet sizes cause a higher utilization. Nevertheless, we can clearly observe that HNLB does scale better for high packet rates. However, for low rates, both approaches have a comparable utilization or sometimes the SLB has even a little lower utilization. This can be explained by the large number of small bursts that have to be processed by HNLB in these cases. Even though we do not reach the limit of both approaches for larger packets, we still can observe that $util_+$ behaves similarly in regions we can cover with our setup. Consequently a gain that is similar to the gain using small packets can be expected as well for high packet rates with large packets.

7.6 Discussion

In this chapter we showed that by utilizing hardware matching capabilities the throughput of stateful network functions can be increased. We used Flow Director technology, available in modern high-speed server NICs, to perform NIC offloading. More specifically we used the matching capabilities of the NIC to increase the throughput of the VNF. Especially, if a higher number of concurrent connections have to be processed, the throughput can be increased quite significantly. For example, for 8000 concurrent connections, throughput can be increased by 50% compared to the software-only case. This is due to the fact that the software only solution has to maintain a state table in software. This directly reflects the effects of the CPU cache overload. For every packet a lookup has to be performed. The table is stored in memory and more connections lead to a table that cannot be stored in the cache anymore. Thus the lookup becomes more expensive in terms of CPU time and the throughput is reduced.

One important aspect of the approach is that the first packet of each connection has no entry in the stateful table. Thus, in our implementation, the first packet must always be processed in software; therefore, no gains are possible with NIC offloading in this case. Figure 7.10 shows

the potential gains of the NIC Offloading approach, compared to the software approach with this restriction. The goal of our approach is to utilize Flow Director matching capabilities in order to increase the throughput of the stateful load balancer. Since Flow Director is not able to forward or modify packets, these two steps have to be done in the offloading and the software processing case. Nevertheless, we can utilize Flow Director for matching and lookup. In software, this is implemented using a hash over the 5-tuple and a table lookup. Both operations are considered to be $O(1)$. However, the table lookup can be comparatively costly in reality, as we showed in Section 5.3. These architectures are designed for heavy computation tasks mainly. One main point is the CPU cache architecture: As long as all the data that is needed for computation is fitting in the caches, the performance is high. On the other hand, if the cache size is exceeded e.g. for slightly larger lookup tables in our case, the performance decreases drastically. Consequently, we can observe a performance drop of the software implementation if we use larger tables. Additionally, the gain of NIC offloading with small tables is smaller, as we can only save the computation cost of the hashing, as the cost of the table lookup in the software case gets smaller.

Chapter 8

Conclusion and Outlook

Traditional network security concepts filter the traffic on the edge of the network. Consequently, traffic between devices in the same network is not filtered. If one client in the network is exploited, an attack can be started between the clients within the networks without filtering capabilities of the security administrator. In recent years several attacks used weaknesses in the protocols used within the network and revealed this lack of filtering options. As a relief, in this thesis a fine-grained SDN/NFV security architecture is presented. SDN is used to provide isolation on a connection level. Though splitting the network into many fine grained virtual networks is not enough to detect all kinds of attacks. Therefore NFV paradigm is used to provide stateful and application layer filtering. Nevertheless, all traffic must be routed through the NFVI, even though some connections might not be filtered. Thus SDN offloading is used to avoid such detours.

8.1 Summary

In Section 1.2 several research challenges were introduces. In the following we will summarize the contributions and findings of this thesis to resolve these challenges.

- **C1 Secure Operation of SDN** With the centralization introduced by SDN, several attack vectors are introduced. In this thesis main attack vectors are summarized, the latency overhead of encryption is evaluated and a Denial of Service (DoS) attack detection method is introduced.
- **C2 Isolation of SDN** Providing access isolation with SDN is relatively straight forward. In contrast to that, performance isolation is more difficult to achieve and only some devices fulfill the requirements. Especially dynamic behaviors of Quality of Service (QoS) mechanisms cause unfairness. We conducted measurements using several hardware

and software SDN switches. One device even violated the configured bandwidth guarantees. We show that Stochastic Fairness Queuing (SFQ) can be a solution to resolve this issue, unfortunately it is not supported by most hardware devices.

C3 Performance of security VNFs NFV is envisioned to use commodity hardware to reduce costs. However, this hardware was not designed initially for this purpose. It is shown that due to the high I/O load of VNFs even architectural details like the placement of the VNF on the Non Uniform Memory Access (NUMA) nodes matter. Further, as multiple VNFs are deployed on the same CPU chip, some on chip resources, such as the Last-Level-Cache (LLC), are shared. This can cause contention and performance degradation. We show an LLC scheduler that reduces the overall load by giving more LLC to the VNF that has the highest load.

C4 Hardware Offloading Even though NFV focuses on software packet processing, the capabilities of hardware packet processing devices shouldn't be forgotten. In this thesis two mechanisms that combine hardware and software packet processing are shown. First we propose to use offloading of traffic to SDN by identifying elephant flows with a machine learning approach. Secondly we show how NIC offloading can reduce the load on the VNFs.

8.2 Outlook

With the findings of this thesis different directions for future work can be followed.

SDN provides visibility and central control in the network. As a drawback the complexity of the SDN controller is high. Furthermore a centralized controller is a single point of failure. Distributed controllers can be a relief, but increase the complexity of the system further. Upcoming white label switches and operating systems could be an opportunity to provide fine grained filtering with simpler means.

In the field of NFV novel and changing technologies can cause new problems and opportunities. The trend towards chiplet designs with smaller CPU chips that are composed to one larger chip for example is such a new technology. The effects on performance of such a design should be evaluated. Further, the LLC scheduler can be extended to be able to cope with more dynamic behavior of the VNFs.

Finally opportunities also exist in the field of NIC offloading. The presented approach offloads the state table to the NIC, the underlying performance gains largely depend on the improved data locality. Thus it is reasonable to assume that the approach also can improve performance for other use cases such as Deep Packet Inspection (DPI).

Bibliography

Publications by the author

[1] R. Durner and W. Kellerer. "The cost of security in the SDN control plane." In: *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT) - Student Workshop*. 2015.

[2] R. Durner, C. Lorenz, M. Wiedemann, and W. Kellerer. "Detecting and mitigating denial of service attacks against the data plane in software defined networks." In: *IEEE Conference on Network Softwarization (NetSoft)*. 2017.

[3] R. Durner, A. Blenk, and W. Kellerer. "Performance study of dynamic QoS management for OpenFlow-enabled SDN switches." In: *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*. 2015, pp. 177–182. DOI: 10.1109/IWQoS.2015.7404730.

[4] S. Gebert, T. Zinner, N. Gray, R. Durner, C. Lorenz, and S. Lange. "Demonstrating a Personalized Secure-by-Default Bring Your Own Device Solution Based on Software Defined Networking." In: *28th International Teletraffic Congress (ITC 28)*. 2016, pp. 197–200. DOI: 10.1109/ITC-28.2016.133.

[5] B. Pfaff, J. Scherer, D. Hock, N. Gray, T. Zinner, P. Tran-Gia, R. Durner, W. Kellerer, and C. Lorenz. "SDN/NFV-enabled Security Architecture for Fine-grained Policy Enforcement and Threat Mitigation for Enterprise Networks." In: *Proceedings of the ACM SIGCOMM Posters and Demos*. 2017, pp. 15–16. DOI: 10.1145/3123878.3131970.

[6] C. Sieber, R. Durner, M. Ehm, W. Kellerer, and P. Sharma. "Towards Optimal Adaptation of NFV Packet Processing to Modern CPU Memory Architectures." In: *Proceedings of the 2Nd Workshop on Cloud-Assisted Networking (CAN)*. 2017, pp. 7–12. DOI: 10.1145/3155921.3158429.

[7] R. Durner, C. Sieber, and W. Kellerer. “Towards Reducing Last-Level-Cache Interference of Co-located Virtual Network Functions.” In: *28th International Conference on Computer Communication and Networks (ICCCN)*. 2019.

[8] R. Durner and W. Kellerer. “Network Function Offloading through Classification of Elephant Flows.” In: *Under Submission in Transactions on Network and Service Management (TNSM)* (2019).

[9] R. Durner, A. Varasteh, M. Stephan, C. Mas Machuca, and W. Kellerer. “HNLB: Utilizing Hardware Matching Capabilities of NICs for Offloading Stateful Load Balancers.” In: *2019 IEEE International Conference on Communications (ICC'19)*. 2019.

General publications

[10] SecureList. *WannaCry ransomware used in widespread attacks all over the world*. https://securelist.com/blog/incidents/78351/wannacry-ransomware-used-in-widespread-attacks-all-over-the-world/.

[11] O. N. F. (ONF). *Software-Defined Networking: The New Norm for Networks [white paper]*. 2012.

[12] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. “Ethane: Taking Control of the Enterprise.” In: *ACM SIGCOMM Computer Communication Review* 37.4 (2007), pp. 1–12.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. “OpenFlow: enabling innovation in campus networks.” In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[14] European Telecommunications Standards Institute (ETSI). *Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action [white paper]*. 2012.

[15] F. Bannour, S. Souihi, and A. Mellouk. “Distributed SDN control: Survey, taxonomy, and challenges.” In: *IEEE Communications Surveys & Tutorials* 20.1 (2018), pp. 333–354.

[16] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern. *RFC 5810 - Forwarding and control element separation (ForCES) protocol specification*. Tech. rep. 2010.

[17] Open Networking Foundation (ONF). *OpenFlow Switch Specification - Version 1.5.0*. 2014.

[18] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits." In: *ACM SIGCOMM Computer Communication Review* 34.4 (2004), pp. 193–204.

[19] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. "Implementing a Distributed Firewall." In: *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS)*. 2000.

[20] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. "Resonance: Dynamic Access Control for Enterprise Networks." In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. 2009.

[21] H. Zhao, C.-K. Chau, and S. M. Bellovin. "ROFL: Routing As the Firewall Layer." In: *Proceedings of the 2008 New Security Paradigms Workshop*. 2008.

[22] J. Deng, H. Hu, H. Li, Z. Pan, K.-C. Wang, G.-J. Ahn, J. Bi, and Y. Park. "VNGuard: An NFV/SDN combination framework for provisioning and managing virtual firewalls." In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 2015.

[23] *Cisco Identity Services Engine*. URL: https://www.cisco.com/c/en/us/products/security/identity-services-engine/index.html.

[24] *Aruba ClearPass*. URL: https://www.arubanetworks.com/products/security/network-access-control/secure-access/.

[25] *Forescout CounterACT*. URL: https://forescout.de/produkte/counteract/.

[26] G. P. et. al. *IEEE 802.1Q-1998 - IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks*. 1998. URL: https://standards.ieee.org/standard/802_1Q-1998.html.

[27] "IEEE Standard for Local and metropolitan area networks–Port-Based Network Access Control." In: *IEEE Std 802.1X-2010 (Revision of IEEE Std 802.1X-2004)* (2010).

[28] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, et al. "ONOS: towards an open, distributed SDN OS." In: *Proceedings of the third ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*. 2014.

[29] D. Kreutz, F. M. Ramos, and P. Verissimo. "Towards secure and dependable software-defined networks." In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*. 2013.

[30] R. Klöti, V. Kotronis, and P. Smith. In: *Proceedings - International Conference on Network Protocols (ICNP)*. 2013.

[31] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, et al. “The Design and Implementation of Open vSwitch.” In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015.

[32] R. Kloti, V. Kotronis, and P. Smith. “OpenFlow: A security analysis.” In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE. 2013.

[33] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack. “Threat modeling-uncover security design flaws using the stride approach.” In: *MSDN Magazine-Louisville* (2006), pp. 68–75.

[34] K. Benton, L. J. Camp, and C. Small. “OpenFlow Vulnerability Assessment Categories and Subject Descriptors.” In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*. 2013.

[35] L. Schehlmann, S. Abt, and H. Baier. “Blessing or curse? Revisiting security aspects of Software-Defined Networking.” In: *10th International Conference on Network and Service Management (CNSM) and Workshop*. 2014.

[36] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid. “Taking Control of SDN-based Cloud Systems via the Data Plane.” In: *Proceedings of the Symposium on SDN Research (SOSR)*. 2018.

[37] S. W. Shin and G. Gu. “Attacking software-defined networks: A first feasibility study.” In: *ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*. 2013.

[38] S. M. Mousavi and M. St-Hilaire. “Early detection of DDoS attacks against SDN controllers.” In: *International Conference on Computing, Networking and Communications (ICNC)*. 2015.

[39] L. Wei and C. J. Fung. “FlowRanger: A request prioritizing algorithm for controller DoS attacks in Software Defined Networks.” In: *IEEE International Conference on Communications (ICC)*. 2015.

[40] R. Kandoi and M. Antikainen. “Denial-of-service attacks in OpenFlow SDN networks.” In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2015.

[41] H. Wang, L. Xu, and G. Gu. *OF-Guard: A DoS Attack Prevention Extension in Software-Defined Networks*. Tech. rep. Texas A&M University, 2014.

[42] H. Wang, L. Xu, and G. Gu. “FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks.” In: *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2015.

[43] Y. Qian, W. You, and K. Qian. “OpenFlow flow table overflow attacks and countermeasures.” In: *European Conference on Networks and Communications (EuCNC)*. 2016.

[44] M. Brooks and B. Yang. “A Man-in-the-Middle attack against OpenDayLight SDN controller.” In: *Proceedings of the 4th Annual ACM Conference on Research in Information Technology (RIIT)*.

[45] 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 ACM SIGCOMM workshop on Hot topics in Software Defined Networks (HotSDN)*. 2012.

[46] E. Sakic, N. Đerić, and W. Kellerer. “MORPH: An Adaptive Framework for Efficient and Byzantine Fault-Tolerant SDN Control Plane.” In: *IEEE Journal on Selected Areas in Communications* 36.10 (Oct. 2018), pp. 2158–2174.

[47] S. Hong, L. Xu, H. Wang, and G. Gu. “Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures.” In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2015.

[48] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. “Infinite CacheFlow in Software-defined Networks.” In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*. 2014.

[49] R. Bifulco and A. Matsiuk. “Towards Scalable SDN Switches: Enabling Faster Flow Table Entries Installation.” In: *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. 2015.

[50] G. Fowler, L. C. N. and Kiem-Phong Vo, D. Eastlake, and T. Hansen. *The FNV Non-Cryptographic Hash Algorithm*. https://tools.ietf.org/html/draft-eastlake-fnv-17. 2019.

[51] A. Varga et al. “The OMNeT++ discrete event simulation system.” In: *Proceedings of the European simulation multiconference (ESM)*. 2001.

[52] *Google Play Protect*. URL: https://www.android.com/play-protect/.

[53] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. *RFC 7348 - Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. 2014.

[54] E. Al-Shaer and S. Al-Haj. “FlowChecker.” In: *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration (SafeConfig)*. 2010.

[55] P. Kazemian, G. Varghese, and N. McKeown. "Header Space Analysis: Static Checking for Networks." In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[56] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. "VeriFlow: Verifying Network-Wide Invariants in Real Time Ahmed." In: *Proceedings of the ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*. 2012.

[57] P. Kazemian, M. Change, and H. Zheng. "Real Time Network Policy Checking Using Header Space Analysis." In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2013.

[58] C. Prakash, Y. Zhang, J. Lee, Y. Turner, J.-M. Kang, et al. "PGA: Using Graphs to Express and Automatically Reconcile Network Policies." In: *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. New York, New York, USA, 2015.

[59] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, and M. Tyson. "Fresco: Modular composable security services for software-defined networks." In: *20th Annual Network & Distributed System Security Symposium*. 2013.

[60] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. *Flowvisor: A network virtualization layer*. Tech. rep. OpenFlow Switch Consortium, 2009.

[61] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer. "Survey on Network Virtualization Hypervisors for Software Defined Networking." In: *IEEE COMMUNICATIONS SURVEYS & TUTORIALS* 18.1 (2016).

[62] P. Georgopoulos, Y. Elkhatib, M. Broadbent, M. Mu, and N. Race. "Towards network-wide QoE fairness using OpenFlow-assisted adaptive video streaming." In: *Proceedings of the 2013 ACM SIGCOMM Workshop on Future Human-Centric Multimedia Networking (FhMN)*. Hong Kong, China, 2013.

[63] T. Zinner, M. Jarschel, A. Blenk, F. Wamser, and W. Kellerer. "Dynamic application-aware resource management using Software Defined Networking: Implementation Prospects and Challenges." In: *Proceedings of the 2014 IEEE Network Operations and Management Symposium (NOMS)*. Krakow, Poland, 2014.

[64] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula. "Automated and scalable QoS control for network convergence." In: *Proceedings of the Internet Network Management Conference on Research on Enterprise Networking (INM/WREN)*. 2010.

[65] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. "Tango: simplifying SDN control with automatic switch property inference, abstraction, and optimization." In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 2014.

[66] M. Kuzniar, P. Peresini, and D. Kostic. *What you need to know about SDN control and data planes*. Tech. rep. EPFL-REPORT-199497. École polytechnique fédérale de Lausanne - EPFL, 2014.

[67] *Open vSwitch - an open virtual switch*. 2014. URL: http://openvswitch.org/.

[68] V. Mann, A. Vishnoi, A. Iyer, and P. Bhattacharya. "VMPatrol: dynamic and automated QoS for virtual machine migrations." In: *Proceedings of the 8th International Conference on Network and Service Management (CNSM)*. 2012.

[69] Z. Bozakov and A. Rizk. "Taming SDN controllers in heterogeneous hardware environments." In: *econd European Workshop on Software Defined Networks (EWSDN)*. 2013.

[70] M. Kuzniar, P. Peresini, and D. Kostic. "What You Need to Know About SDN Flow Tables." In: *Passive and Active Measurement*. Ed. by J. Mirkovic and Y. Liu. Vol. 8995. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 347–359.

[71] P. M. Mohan, D. M. Divakaran, and M. Gurusamy. "Performance study of TCP flows with QoS-supported OpenFlow in data center networks." In: *Proceedings of the 19th IEEE International Conference on Networks (ICON)*. 2013.

[72] A. Nguyen-Ngoc, S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, and M. Jarschel. "Investigating isolation between virtual networks in case of congestion for a Pronto 3290 switch." In: *Proc. of the Workshop on Software-Defined Networking and Network Function Virtualization for Flexible Network Management (SDNFlex 2015)*. 2015.

[73] P. McKenney. "Stochastic fairness queueing." In: *Proceedings of the Ninth Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*. 1990.

[74] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. e. a. Gibbs. *Iperf: the TCP/UDP bandwidth measurement tool*. Version 2.0.5. 2008. URL: http://sourceforge.net/projects/iperf/.

[75] V. Jacobsen, C. Leres, and S. McCanne. *Tcpdump/libpcap*. Version 4.5.1. 2012. URL: http://www.tcpdump.org.

[76] *Floodlight - open SDN controller*. URL: http://www.projectfloodlight.org/floodlight/.

[77] *tc - show / manipulate traffic control settings*. URL: http://www.lartc.org/manpages/tc.txt.

[78] V. Paxson, M. Allman, J. Chu, and M. Sargent. *RFC 6298 - Computing TCP's retransmission timer*. 2011.

[79] M. Majkowski. *Cloudflare Blog: How to receive a million packets per second*. 2015. URL: https://blog.cloudflare.com/how-to-receive-a-million-packets/.

[80] *x86 Server CPUs Remain Market Mainstream*. 2018. URL: https://www.dramexchange.com/WeeklyResearch/Post/2/5156.html.

[81] C. Wang, O. Spatscheck, and V. e. a. Gopalakrishnan. "Toward High-Performance and Scalable Network Functions Virtualization." In: *IEEE Internet Computing* 20.6 (Nov. 2016), pp. 10–20.

[82] P. Li, X. Wu, Y. Ran, and Y. Luo. "Designing Virtual Network Functions for 100 GbE Network Using Multicore Processors." In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2017.

[83] I. Cerrato, M. Annarumma, and F. Risso. "Supporting fine-grained network functions through Intel DPDK." In: *Third European Workshop on Software Defined Networks (EWSDN)*. 2014.

[84] Y. Hu and T. Li. "Towards efficient server architecture for virtualized network function deployment: Implications and implementations." In: *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016.

[85] A. Banerjee, R. Mehta, and Z. Shen. "NUMA Aware I/O in Virtualized Systems." In: *IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI)*. 2015.

[86] M. Dobrescu, K. Argyraki, and S. Ratnasamy. "Toward Predictable Performance in Software Packet-Processing Platforms." In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[87] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu. "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 2017.

[88] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li. "Demystifying the Performance Interference of Co-Located Virtual Network Functions." In: *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 2018.

[89] Y. Hu, M. Song, and T. Li. "Towards "Full Containerization" in Containerized Network Function Virtualization." In: *ACM SIGARCH Computer Architecture News* 45.1 (2017), pp. 467–481.

[90] P. Veitch, E. Curley, and T. Kantecki. "Performance evaluation of cache allocation technology for NFV noisy neighbor mitigation." In: *IEEE Conference on Network Softwarization (NetSoft)*. 2017.

[91] M. Dobrescu, K. Argyraki, and S. Ratnasamy. "Toward Predictable Performance in Software Packet-processing Platforms." In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2012.

[92] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gomez. "Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology." In: *Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2017.

[93] S. G. Kim, H. Eom, and H. Y. Yeom. "Virtual machine scheduling for multicores considering effects of shared on-chip last level cache interference." In: *International Green Computing Conference (IGCC)*. 2012.

[94] R. Nathuji and A. Kansal. "Q-Clouds : Managing Performance Interference Effects for QoS-Aware Clouds." In: *Proceedings of the 5th European conference on Computer systems*. 2010.

[95] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. "Fairness via source throttling: A configurable and high-performance fairness substrate for multicore memory systems." In: *ACM Transactions on Computer Systems (TOCS)* 30.2 (2012), 7:1–7:35.

[96] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. "Heracles: Improving resource efficiency at scale." In: *Proceedings of the 42nd Annual ACM International Symposium on Computer Architecture (ISCA)*. 2015.

[97] A. R. Nath, A. Kansal, S. Govindan, J. Liu, and Suman. "PACMan: Performance Aware Virtual Machine Consolidation." In: *10th International Conference on Autonomic Computing (ICAC)*. 2013.

[98] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. "ResQ: Enabling SLOs in Network Function Virtualization." In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018.

[99] *Dataplane Development Kit.* 2017. URL: http://dpdk.org/.

[100] D. Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. 2009.

[101] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach Fifth Edition*. Elsevier, 2012.

[102] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016.

[103] *Qualcomm Centriq™ 2400 Processor*. URL: https://www.qualcomm.com/media/documents/files/qualcomm-centriq-2400-processor.pdf.

[104] European Telecommunications Standards Institute (ETSI). *Network functions virtualisation (NFV): Architectural framework*. 2013.

[105] M. Beierl. *NFV-KVM-Tuning*. 2016. URL: https://wiki.opnfv.org/display/kvm/Nfv-kvm-tuning.

[106] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, and A. Csaszar. "Elastic network functions: opportunities and challenges." In: *IEEE Network* 29.3 (2015), pp. 15–21.

[107] P. F. Thomas Willhalm Roman Dementiev. *Intel Performance Counter Monitor - A better way to measure CPU utilization*. 2017. URL: https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[108] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, et al. "P4: Programming protocol-independent packet processors." In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[109] E. Balas and A. Ragusa. "SciPass: a 100Gbps capable secure science DMZ using OpenFlow and Bro." In: *Supercomputing Conference (SC14)*. 2014.

[110] S. Miteff and S. Hazelhurst. "NFShunt: A Linux firewall with OpenFlow-enabled hardware bypass." In: *IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 2015.

[111] F. Heimgaertner, M. Schmidt, D. Morgenstern, and M. Menth. "A Software-Defined Firewall Bypass for Congestion Offloading." In: *Proceedings of the International Conference on Network and Service Management (CNSM)*. 2017.

[112] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang. "Leveraging Zipf's law for traffic offloading." In: *ACM SIGCOMM Computer Communication Review* 42.1 (2012), pp. 16–22.

[113] B. Claise. *Cisco systems netflow services export version 9*. Tech. rep. 2004.

[114] Y. Li, R. Miao, C. Kim, and M. Yu. “FlowRadar: a better NetFlow for data centers.” In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2016.

[115] L. Che and B. Qiu. “Landmark LRU: An efficient scheme for the detection of elephant flows at internet routers.” In: *IEEE Communications Letters* 10.7 (2006), pp. 567–569.

[116] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner. “Optimal elephant flow detection.” In: *IEEE Conference on Computer Communications (INFOCOM)*. 2017.

[117] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto. “Identifying elephant flows through periodically sampled packets.” In: *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement (IMC)*. 2004.

[118] K. Psounis, A. Ghosh, B. Prabhakar, and G. Wang. “SIFT: A simple algorithm for tracking elephant flows, and taking advantage of power laws.” In: *43rd Allerton Conference on Communication, Control and Computing*. 2005.

[119] Y. Afek, A. Bremler-Barr, S. Landau Feibish, and L. Schiff. “Sampling and large flow detection in SDN.” In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 345–346.

[120] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. “DevoFlow: Scaling flow management for high-performance networks.” In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 254–265.

[121] A. R. Curtis, W. Kim, and P. Yalagandula. “Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection.” In: *IEEE Conference on Computer Communications (INFOCOM)*. 2011.

[122] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin. “Online flow size prediction for improved network routing.” In: *International Conference on Network Protocols (ICNP)*. 2016.

[123] P. Xiao, W. Qu, H. Qi, Y. Xu, and Z. Li. “An efficient elephant flow detection with cost-sensitive in SDN.” In: *1st International Conference on Industrial Networks and Intelligent Systems (INISCom)*. 2015.

[124] N. Viljoen, H. Rastegarfar, Mingwei Yang, J. Wissinger, and M. Glick. “Machine learning based adaptive flow classification for optically interconnected data centers.” In: *18th International Conference on Transparent Optical Networks (ICTON)*. 2016.

[125] S. Chao, K. C. Lin, and M. Chen. “Flow Classification for Software-Defined Data Centers Using Stream Mining.” In: *IEEE Transactions on Services Computing* 12.1 (Jan. 2019), pp. 105–116.

[126] T. T. Nguyen and G. J. Armitage. “A survey of techniques for internet traffic classification using machine learning.” In: *IEEE Communications Surveys and Tutorials* 10.4 (2008), pp. 56–76.

[127] H. Zhang. “The optimality of naive Bayes.” In: *Proceedings of the FLAIRS Conference*. American Association for Artificial Intelligence. 2004.

[128] W. De Donato, A. Pescapé, and A. Dainotti. “Traffic identification engine: an open platform for traffic classification.” In: *IEEE Network* 28.2 (2014), pp. 56–64.

[129] A. Munther, A. Alalousi, S. Nizam, R. R. Othman, and M. Anbar. “Network traffic classification—A comparative study of two common decision tree methods: C4. 5 and Random forest.” In: *2nd International Conference on Electronic Design (ICED)*. 2014.

[130] M. Shafiq, X. Yu, A. A. Laghari, L. Yao, N. K. Karn, and F. Abdessamia. “Network Traffic Classification techniques and comparative analysis using Machine Learning algorithms.” In: *Proceedings of the IEEE International Conference on Computer and Communications (ICCC)*. 2016.

[131] F. Eibe, M. Hall, and I. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. 2016.

[132] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. *RFC 7413 - TCP Fast Open*. 2014. URL: http://tools.ietf.org/html/rfc7413.

[133] R. Kohavi. “The Power of Decision Tables.” In: *8th European Conference on Machine Learning*. Springer, 1995.

[134] R. Quinlan. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

[135] L. Breiman. “Random forests.” In: *Machine learning* 45.1 (2001), pp. 5–32.

[136] Y. Freund and R. E. Schapire. “Experiments with a new boosting algorithm.” In: *Thirteenth International Conference on Machine Learning*. San Francisco: Morgan Kaufmann, 1996.

[137] P. Phaal and M. Lavine. *sFlow Specification Version 5*. 2004.

[138] M. Menth and F. Hauser. “On Moving Averages, Histograms and Time-Dependent Rates for Online Measurement.” In: *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2017.

[139] WIDE MAWI Working Group. *WIDE Traffic Archive*. http://mawi.wide.ad.jp/.

[140] Center for Applied Internet Data Analysis. *Caida Anonymized Internet Traces 2016*. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[141] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. "Network function virtualization: State-of-the-art and research challenges." In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 236–262.

[142] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. "The Click modular router." In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[143] L. Rizzo. "Netmap: a novel framework for fast packet I/O." In: *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX Association. 2012.

[144] *PF_RING*. URL: https://github.com/ntop/PF_RING.

[145] S. Han, K. Jang, K. Park, and S. Moon. "PacketShader: a GPU-accelerated software router." In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 195–206.

[146] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. "OpenState: programming platform-independent stateful openflow applications inside the switch." In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 44–51.

[147] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. "Duet: Cloud scale load balancing with hardware and software." In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 27–38.

[148] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 2017.

[149] Y. Weinsberg, E. Pavlov, Y. Amir, G. Gat, and S. Wulff. "Putting it on the NIC: A case study on application offloading to a Network Interface Card (NIC)." In: *IEEE Consumer Communications and Networking Conference (CCNC)*. 2006.

[150] J. Shi, T. Liang, H. Wu, B. Liu, and B. Zhang. "Ndn-nic: Name-based filtering on network interface card." In: *Proceedings of the 3rd ACM Conference on Information-Centric Networking (ICN)*. 2016.

[151] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud." In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018.

[152] Intel. *Intel Ethernet Flow Director*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf.

[153] *Intel X540 datasheet*. URL: https://cloud.lkn.ei.tum.de/s/Y4QfYkjgD4AeStc.

[154] *DPDK Paket Generator*. 2017. URL: http://pktgen-dpdk.readthedocs.io.

Acronyms

ACL Access Control List 1, 62, 64, 67, 68, 69, 70, 84

AMAT Average Memory Access Time 62, 70, 84

BCAM Binary Content Addressable Memory 27, 108, 114

CAM Content Addressable Memory 27

CAT Cache Allocation Technology 60, 61, 62, 72, 73

CDF Cumulative Distribution Function 32, 74

COS Classes of Service 72

DASH Dynamic Adaptive Streaming over HTTP 43

DDIO Data Direct I/O 59, 66, 67

DIP Direct IP 121, 122, 123, 124, 127

DMA Direct Memory Access 59, 67

DMZ Demilitarized Zone 90

DoS Denial of Service 6, 16, 19, 21, 22, 23, 24, 27, 28, 36, 133

DPDK Data Plane Development Kit 7, 59, 63, 64

DPI Deep Packet Inspection 11, 134

DUT Device-under-test 63

FIFO First-In-First-Out 43, 45, 46, 49, 50, 51, 54

FN False Negative 32, 33, 34

FP False Positive 32, 33, 34

FPGA Field Programmable Gate Array 121

HNLB Hybrid NIC Offloading Load Balancer 123, 126, 127, 128, 129, 130, 131

HTB Hierarchical Token Bucket 46

HTTP Hypertext Transfer Protocol 4

IDS Intrusion Detection System 3, 74, 90

IP Internet Protocol 18, 37

ISE Identity Service Engine 13

ISP Internet Service Provider 101

LLC Last-Level-Cache 6, 56, 57, 58, 59, 60, 61, 62, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 83, 84, 85, 134

LLDP Link Layer Discovery Protocol 18, 19, 24

LRU Least-recently-used 73

MANO Management and Orchestration 71

MitM Man in the Middle 15, 17, 18, 19, 23, 24

MPLS Multi Protocol Label Switching 40

MTU Maximum Transmission Unit 38, 96

NAT Network Address Translation 11

NFV Network Function Virtualization iii, iv, 3, 4, 5, 6, 7, 11, 12, 14, 55, 56, 57, 59, 60, 61, 62, 65, 70, 71, 75, 84, 87, 89, 94, 113, 117, 119, 133, 134

NFVI Network Function Virtualization Infrastructure iv, 4, 5, 87, 117, 133

NIC Network Interface Card 5, 7, 8, 58, 59, 60, 63, 64, 65, 66, 67, 89, 117, 118, 119, 120, 121, 123, 124, 127, 128, 131, 132, 134

NUMA Non Uniform Memory Access 56, 57, 58, 59, 60, 62, 63, 65, 66, 67, 68, 69, 70, 84, 134

OVS Open vSwitch 43, 46, 48, 49, 50, 51, 53

PDF Propability Density Function 74

PQ Priority Queuing 45, 48, 50, 51, 52, 53

QFF Quality of Experience Fairness Framework 43

QoS Quality of Service 6, 42, 43, 44, 45, 46, 48, 50, 54, 92, 133

QPI Quick-Path Interconnect 58

RSS Receive Side Scaling 122, 123

RTO retransmission timeout 52

RTT Round Trip Time 22, 46, 52

SDN Software-Defined Networking iii, iv, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 35, 36, 38, 39, 40, 41, 42, 43, 44, 46, 50, 54, 55, 87, 88, 89, 92, 93, 94, 98, 113, 117, 120, 133, 134

SFQ Stochastic Fairness Queuing 43, 45, 46, 49, 50, 51, 53, 54, 134

SLB Software Load Balancer 123, 124, 125, 126, 127, 128, 129, 130, 131

SMT Simultanous Multithreading 127

SR-IOV single-root input/output virtualization 59

SRTT Smoothed Round Trip Time 52

TCAM Ternary Content Addressable Memory 27, 108

TLS Transpor Layer Security 15, 16, 24, 26, 95

UDP User Datagram Protocol 38

vFW virtualized Firewall 87, 88

VIM Virtualized Infrastructure Manager 71

VIP Virtual IP 121, 122, 124

VM Virtual Machine 22, 25, 44, 59, 60, 61

VN Virtual Network 41, 42, 54

VNF Virtualized Network Function iii, iv, 3, 4, 5, 6, 8, 11, 55, 56, 57, 58, 59, 60, 61, 62, 63, 65, 70, 71, 72, 73, 74, 75, 80, 82, 83, 84, 85, 87, 89, 92, 94, 117, 131, 134

VNFM Virtualized Network Function Manager 72

VPN Virtual Private Network 11

WFQ Weighted Fair Queuing 46, 48

List of Figures

1.1 Conventional legacy security architecture of an enterprise network 2
1.2 SDN/NFV enabled security architecture with fine-grained isolation 3
1.3 Overview of the topics covered in this thesis in relation to different modules of the security architecture 8

2.1 Software defined networking architecture 10

3.1 Attack vectors in SDNs. Attack vectors can be grouped into attacks from the data, control and application plane. 17
3.2 Normal behavior of a reactive SDN. 19
3.3 DoS attack against a reactive SDN. 20
3.4 Measurement setup: The SDN Controller is used as a relay, thus no rules are installed in the hardware table.Thus all packets have to traverse the shown path. . 25
3.5 Latency added to the first packet by switches and controller using TCP and TLS . 26
3.6 Abstracted simulation of the detection system. 31
3.7 Results of the maximum table value for different inter arrival times. 34
3.8 Working region of detection algorithm, with $\lambda_a = 10 \cdot \lambda_l$ and $H = 100$ 34
3.9 Results for the slope for different inter arrival times 35

4.1 Virtual networks are provisioned for each service. By default the user has only access to the portal. 41
4.2 Stochastic Fairness Queuing Algorithm 46
4.3 OpenFlow-based testbed setup. Queue 0 (q_0), Queue 1 (q_1), and Scheduler are configured for each experiment accordingly. 47
4.4 Time sequence of the experiments 47
4.5 Measured mean bandwidth over time using different switches and priority queuing. 95% confidence intervals for the mean values are added to all plots. 48
4.6 Mean number of duplicated packets of Flow 1 (F1) and Flow 2 (F2) in an interval of 10 s around T1 and T2. 95% confidence intervals are included. Measurement at the sink of the flows. 49

4.7 Measured mean bandwidth over time using different switches and bandwidth guarantees settings. 95% confidence intervals for the mean values are added to all plots. 50
4.8 Mean number of duplicated packets of Flow 1 (F1) and Flow 2 (F2) in an interval of 10 s around T1 and T2. 95% confidence intervals are included. Measurement at the sink of the flows. 51
4.9 Smoothed round trip time of Flow 1 from two different measurements using priority queuing on the NEC Switch 52
4.10 Bandwidth of 50 runs using OVS employing inverted PQ and SFQ 53

5.1 NUMA architecture with DDIO for the NIC-local NUMA node. DDIO allows direct access to the local LLC. DMA is used for packets destined to the remote node. Each Core has a L1 and L2 cache, the LLC is shared between all cores on one node. A Quick-Path Interconnect (QPI) bus with 2x19.2 GBps connects the nodes. 57
5.2 Simplified depiction of the Intel Xeon processor's cache hierarchy. Each core is equipped with an exclusive L1 cache (64 KB) and L2 cache (256 KB). A Last Level Cache (LLC) with a capacity of 30 MByte is shared. Access times are cumulative and range from roughly 2 ns if the memory access can be satisfied from the L1 cache to 100 ns if the data has to be fetched from the main memory. 58
5.3 Test environment: A PC running a load generator is connected with the DUT, a x86 server. Both devices are connected using one 10G Ethernet link. 63
5.4 The function chain consisting of one RX, one ACL and one TX process. Each process runs exclusively on one core. The RX and TX processes move the packets between the respective buffers and rings. The ACL process checks each packets against the its rule set. 64
5.5 Simple Receiver: Utilization of the core for increasing packet rates and two different placements (node 0 and node 1). The nic is attached to node 0. A 41 % increase in utilization is observed when the receiver is executed on node 1 (20.8 % utilization compared to 14.7 %), due to the overhead in fetching packets from the nic through the QPI. 66
5.6 LLC cache hit ratio and memory read rate on node 1 for two placements of the minimal VNF for increasing packet rates up to 1 million packets per second. The figure confirms that a placement on the remote NUMA node without DDIO support results in cache misses and high main memory access rates. 66

5.7 ACL core utilization for four different placements of the firewall function chain (RX-ACL-TX) depending on packet rates. Placement 0-0-0 represents the best case placement without remote NUMA access and 0-1-0 the worst case where remote NUMA memory has to be accessed. The results shows an increase in utilization for the worst case placement of roughly 73 % compared to the best case. 68
5.8 ACL core cache hit ratios of the L2 and LLC depending on the ACL size for the four different placements. As expected, the hit ratio of the small L2 cache falls fast to about 0 % for all four placements. For the LLC, we measure a hit ratio of 7 % for the worst case placements 0-1-0 and 0-1-0, where the packet is not yet in the LLC. For the best case placements 0-0-0 and 1-1-1, where the packet is already available in the LLC, we measure a hit ratio of of 30 %. 69
5.9 Impact of memory accesses on the core utilization of the ACL core. Increasing ACL size for a constant packet rate. L2 cache capacity is 256 KB and LLC cache capacity is 30 MB. The Figure shows the additional utilization caused by memory access. The additional utilization follows for the placements 1-0-1 and 0-1-0 roughly the trend of a simple AMAT model (right figure). The dots in the right figure are the measurement points of the 0-1-0 placement. 69
5.10 LLC scheduling in the bigger picture of NFV Management and Orchestration (MANO). Horizontal scaling of VNF instances for a specific service, e.g., a firewall, is performed by the VNF manager through the VIM. Through the routing/load-balancing functionality of the network, the VNFM dictates which fraction of the traffic is assigned to a specific VNF. Network elements then forward and distribute traffic to the running VNFs. The VNFs have to share the available LLC and only part of the working set S of each VNF can be kept in the LLC. 71
5.11 Possible memory access patterns. 73
5.12 CDF and PDF of the developed memory access distribution. 75
5.13 CPU utilization and LLC occupation of an exemplary scheduler run with 5 cores. The duration of each step is 12 seconds. 77
5.14 Transient phase after an LLC allocation update: Difference between the CPU utilization at a certain time after the update with respect to the median utilization of the CPU utilization in the interval [7,10]. 81
5.15 Number of steps the scheduler needs until the maximum is found. The scheduler needs on average 12 steps until all ways are distributed. 82
5.16 Relation between gain and difference between the CPU utilization of the highest core and the mean of the CPU utilization of all cores 82
5.17 Relation between gain and the (mean) allocated memory of the scheduled VNF(s) 83

6.1 Network Function Offloading in an SDN/NFV Environment. The virtual firewall (vFW) filters the packets. As the infrastructure is centralized, a detour is caused for internal connections. After connection initiation the direct path can be used. The packets are still filtered by hardware, but only connections with high data rate should be offloaded to keep the rule count in the hardware tables small. 88
6.2 The classification system: The packets from the trace resp. on the line are recorded as flows. The ground truth is added depending on the gathered statistics. After that some filters are applied aiming to build a basis for the machine learning algorithm. The algorithm uses the set to train a model that can be used to classify new connections. These connections are recorded again for the next iteration. . . 93
6.3 Gathering the necessary statistics using multiple points in the network. In a normal case packets are processed using a function chain located at the NFVI, in this case statistics can be gathered using a statistics network function. If a flow is offloaded the statistics have to be gathered by the hardware. All statistics have to be consolidated in the end, e.g. using an SDN controller or a network management system. 94
6.4 Histogram of the size of the first packet from each flow for the Wide A trace. The distribution shows high frequency for small packets. 95
6.5 Sampling based Offloading approach: Every n_s packet is forwarded to the Offloading logic. If the decision is for offloading the flow of the corresponding packet is offloaded by the SDN Controller. 98
6.6 Data Labeling: Offloaded share and Table occupation if the labels are used for the decision directly. For a small threshold Θ_F the table occupation is quite high even in this all-knowing case. But it quickly decreases. We chose $\Theta_F = 1e4\,B$ as a compromise between table occupation and offloaded rate. 102
6.7 Merging of infrequent values: The Figure shows the performance of the offloading algorithm with J48 as machine learning algorithm. Nominal features with a lower frequency than the minimum frequency f are merged. Table Occupation and offloaded bit rate increases with increasing f. f=500 provides a good compromise. 104
6.8 Precision of the selected algorithms with the different data sets. The mean precision for class 0 (offload=No) ranges between 0.84 and 0.96 depending on data set and algorithm. The precision for class 1 is worse in general. J48 and RandomForest show the best classification precision in general. The Decision Table classification precision is differing largely for the CAIDA data sets. Boosted DecisionStump is showing decent performance for the WIDE data sets but has only low accuracy in the CAIDA data sets. 105

6.9 Offloaded data rate using the different learning schemes. The bit rate that is processed by hardware is approximately as good as with the all-knowing ideal scheme. This is due to the much higher number of processed flows compared to the ideal case. In general the lion's share of the rate can be processed in hardware. 106
6.10 Mean table occupation. One major limitation is the number of rules in the hardware table. J48 requires the lowest number of table entries. On the other hand the overhead compared to the ideal case is quite large for all learning schemes. 107
6.11 Decision tree size and table size when increasing the minimum number of instances per leaf. The Figure shows the results using WIDE B data set. A reduction of the tree by one magnitude only increases the table size necessary by 12.5 %. 108
6.12 Temporal Stability: Offloading performance when using the Model trained with WIDE A data set for classifying WIDE B data set. DecisionTable, NaiveBayes and boosted NaiveBayes perform worse in this scenario. J48 and RandomForest algorithm show less degradation. The offloaded rate is still in the same range as for the ideal decision, the table occupation is ~20 % higher when compared to the model learned by cross validation. 109
6.13 Temporal Stability: Offloading performance for the WIDE 24h data set and J48. Performance is not impacted largely when different learning intervals are used. . . 110
6.14 Temporal Stability: Offloading share for the WIDE 24h data set in a heat map. This view on the results shows that some times of the day have a different traffic pattern than others. 110
6.15 Offloading using Sampling: For lower sampling rates all flows that belong to sampled packets are offloaded, thus Sample+ and Baseline algorithm perform equally. Only for higher rates we can see that sample+ is much more economically regarding table occupation. 111
6.16 Overview of the offloaded share for all algorithms and data sets. In order to reach the performance of the Machine Learning algorithms a high sampling rate is necessary. 112

7.1 Application layer filtering in the SDN/NFV architecture. The traffic is routed through the NFVI where application layer filters (L7 Filter) check the traffic. In order to provide scaling and performance the traffic is distributed between multiple instances using a Load Balancer (LB). 118
7.2 Basic working principle of a stateful L4 load balancer. The VIP of the served packet is highlighted in black, the DIP is highlighted in gray. 122

7.3 Load balancer with hardware table offloading: Packets from unseen connections are put into queue 0 (1) and polled from the HNLB Software (2). These packets do not match any connection in the software connection table (3), therefore a DIP is chosen from the VIP table and the mapping is installed in software (4a) and hardware (4b). The packet header is rewritten and forwarded (5). Subsequent packets of this connection are put in the queue of the corresponding DIP (6). When they are polled from the queue the matching for the rewriting is encoded in the queue number already (7). 124

7.4 Best Case Packet Throughput for a paket size of 64 Byte, a single flow and a single DIP. 127

7.5 Maximum Throughput for a packet size of 64 Byte and different number of concurrent connections. 128

7.6 Maximum throughput for different packet sizes. Vertical lines represent the maximum offered rate of the traffic generator. 128

7.7 Comparison of Utilization metrics for different rates, 1000 concurrent connections and 10 Queues. Vertical lines show the measurement values where the first loss occurs. 129

7.8 $util_+$ for different rates, different number of concurrent connections, a packet size of 64 Byte and 10 Queues. Vertical lines show the measurement values where the first loss occurs. HNLB is marked with dashed lines. SLB is shown with solid lines. 130

7.9 $util_+$ for different rates, different packet sizes, 1000 concurrent connections and 10 Queues. Vertical lines show the measurement values where the first loss occurs. HNLB is marked with dashed lines. SLB is shown with solid lines. 130

7.10 Software packet processing without Offloading and the matching capabilities. The flow assumes that a DIP table entry exists which holds for all packets of a connection except the first one. 131

List of Tables

3.1 Simple example showing the growth of a counter table over time for ten consecutive packets with three header fields **a**, **b**, **c** where **c** is varied while **a** and **b** remain fix. 28
3.2 Parameters of simulation and Analysis 32
3.3 Exemplary simulation results for a network with 100 Hosts 33

5.1 Notation 78
5.2 Experimental parameters used for evaluation. 79

6.1 Features used for machine learning. All features are available with the first packet of a flow. This enables early classification and a larger gain compared to other approaches. 95
6.2 Data sets used in this work 100
6.3 Information gain of the attributes in bit 103