博士論文

Exploring Benefits of Deep Dataplane Programmability Through In-Network Processing Use Cases

(ネットワーク処理ユースケースに基づくディープデータプレーンプログラマビリティの利点の探求)

ハミド　ファルハディイ

Dedicated to the soul of wisdom.

Acknowledgements

I would like to express my deepest appreciation and gratitude to my supervisor, Professor Akihiro Nakao for his outstanding and invaluable direction. He continuously supports my research with attractive ideas and timely assistance throughout the period of my Ph.D. study. He teaches me how to be a good researcher and always encourages me to improve my skills and catch chances for a better academic career. This thesis would not have been possible to be completed without his exemplary supervision, guidance and advice.

Finally, I thank to my family and to all my friends, who are always there when I need them.

Abstract

Software-Defined Networking (SDN) is a new networking paradigm that is getting more and more popular because it is considered as a prominent method to facilitate networking practices. SDN defines two core features, i) decomposition of control and data planes and ii) a centralized control plane (i.e., controller) based on a network-wide view. Those SDN characteristics provide control plane programmability, support networking applications, and enable research and innovation in networking. While SDN has a strong proposal in control plane, it may have some shortcomings in data plane.

In this thesis, we recognize and address two major problems in the current SDN data plane:

1. *Inefficient packet classification in the switch:* SDN data plane uses a limited set of predefined packet headers (e.g., source and destination IP addresses) to classify packets for switching and repeats checking all header fields for classification on every switch in the network. Moreover, ther are potentially many header fields than can be matched together to classify every packet that can make complex combinations. So, there are three subproblems in this problem: i) SDN classification is redundant ii) SDN classification is predefined and iii) SDN classification is complex.
2. *Switch actions are hardwired in the switch hardware:* After classifying packets SDN data plane applies a set of predefined actions on every packet (e.g., forward and drop). All these actions are hardwired directly on the switch hardware and network operators can not make any change or adapt new requirements. So there are two subprobles in this problem: i) SDN switch actions are predefind and ii) there are limited number of actions.

We posit that solving aforementioned problems in the SDN data plane is a key to the successful migration of current hardware-centric networking practices to SDN. The general benefits of such a migration lie in two aspects:

- *For academia:* We propose technologies that foster Data plane programmability along with an open data plane architecture to unlock innovation in the networking.
- *For industry:* We relax the constraints of traditional predefined, hardwired data planes which lets new and small firms enter the networking software development market and conduct the industry to a more competitive and innovative atmosphere.

Building on top of Deeply Programmable Networking (DPN) concept, which is already proposed in the community, we developed the following technologies as mitigation solutions to SDN problems. To address *Problem 1* we propose TagFlow. TagFlow is a classification and forwarding architecture for SDN. The main contributions of TagFlow are two folds: first, using lightweight classification at the core it offloads the main classification load to the network edge. Second, we show TagFlow releases the classification load on core devices and using that opportunity it is possible to leverage from heavier classification mechanisms (e.g., application layer classification) than common methods (i.e., header classification).

As for *Problem 2* we propose User-Defined Actions (UDA), a flexible architecture to accommodate user generated packet processing mechanisms within the switch data plane. UDAs let SDN programmers to freely build arbitrary use cases specific to their own needs without the limitation of traditional predefined actions in SDN. The main contributions of UDAs are two folds: first, through extensive experiments we indicate that our UDAs can elevate millisecond-scale running time of current proposals to nanosecond-scale (including proposals from northbound applications of SDN community and virtual appliances of Network Function Virtualization or NFV community). Second, to raise the importance of the ease of programmability when dealing with network programmability, we show that our proposal decrease

the lines of code compared to implementing the same functionality as a northbound application and as a standalone middlebox.

The technical benefits of our proposals lie in two aspects:

- Primary benefit: We can adapt new architectures or protocols and basic packet processing without hardware replacement
- Secondary benefit: Reduced cost for networking devices and reduced latency in packet transmission over the network.

Aforementioned proposals are examples of application programming interfaces of the DPN. These interfaces are the mean to expose programmability deeply and comprehensively in data in addition to control plane (in the form of APIs) to realize deeply programmable networks.

Contents

Contents **vi**

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

1.1 Background 1

1.2 Problem Statement 3

1.2.1 Packet classification is inefficient 3

1.2.2 Switch actions are hardwired in the switch hardware 4

1.3 Thesis Statement 5

1.4 Strategy and Philosophy 5

1.5 Scope 6

1.6 Summary 6

2 Software-Defined Networking (SDN) **8**

2.1 Introduction 8

2.2 Software-Defined Networking 9

2.3 Current Research on SDN 11

2.3.1 Controller and Switch 11

2.3.1.1 Availability and Resilience 12

2.3.1.2 Scalability 14

2.3.1.3 Security . . . 15
2.3.1.4 State Management and Policy Enforcement . . . 16
2.3.1.5 Flexibility . . . 17
2.3.2 SDN-enabled Networks . . . 18
2.3.2.1 Virtualized Network . . . 19
2.3.2.2 Large-scale Network . . . 20
2.3.2.3 Data Center Network . . . 20
2.3.3 Debugging . . . 21
2.3.3.1 Network . . . 21
2.3.3.2 Controller . . . 22
2.3.3.3 Available Tools . . . 23
2.3.4 Abstractions . . . 23
2.3.4.1 Declarative Language . . . 24
2.3.4.2 Achieving Specific Goals . . . 24
2.3.4.3 Misc. . . . 25
2.4 Summary . . . 25

3 Deeply Programmable Networks (DPN) 26

3.1 Necessity of Data Plane Programmability . . . 26
3.2 OpenFlow Data Plane . . . 27
3.3 Potential Data Plane-related Proposals for SDN . . . 28
3.3.1 Packet Classification and Forwarding . . . 28
3.3.2 Easy Programmability . . . 29
3.3.3 Resource Allocation . . . 30
3.3.4 Security . . . 30
3.3.5 Network Measurement . . . 30
3.3.6 Statefull Packet Processing . . . 31
3.3.7 Wireless Networking . . . 31
3.3.8 Misc. . . . 32
3.4 Extending SDN to Deeply Programmable Networks . . . 32

3.5 Summary 33

4 Tag-Based Flow Classification in DPN 34

4.1 Introduction 34

4.1.1 Flexibility of Flow Classification 36

4.1.2 Performance of Flow Classification 36

4.2 Related Work 37

4.2.1 Hashing Approach 39

4.2.2 Tagging Approach 40

4.3 System Architecture 42

4.3.1 Source Edge 43

4.3.2 From Network Core to Destination Edge 44

4.4 Evaluation 45

4.4.1 TagFlow Control Plane 45

4.4.2 TagFlow Backward Compatibility 45

4.4.3 Single Node Experiments 46

4.4.4 End-to-end Experients 55

4.4.5 Northbound versus Southbound Applications 64

4.5 Discussion 66

4.6 Summary 69

5 User-Defined Switch Actions in DPN Data Plane 70

5.1 Introduction 70

5.2 Related Work 71

5.3 User-Defined Action Usecases 72

5.3.1 Northbound (or Control Plane) Applications 74

5.3.2 NFV Appliances 75

5.3.3 User-Defined Switch Actions 76

5.4 System Architecture 77

5.4.1 Ease of Programmability 79

5.5 Throughput Evaluations 82

5.5.1 OpenFlow Plus UDAs . . . 82
5.5.2 More Complex UDAs . . . 83
5.5.3 Portscan Detector UDA Experiment on Attack Trace . . . 84
5.6 Summary . . . 84

6 Conclusion and Future Directions 86

6.1 Summary . . . 86
6.2 Future Directions on DPN . . . 87
6.2.1 Combining SDN and NFV . . . 87
6.2.2 Data Plane Abstractions . . . 88
6.2.3 Data Plane Verification . . . 88
6.2.4 Netwrok Programming languages . . . 88

References 90

List of Figures

1.1 Components of SDN. 2

2.1 High-level illustration of networking paradigms. 10

2.2 SDN overview 11

3.1 SDN vs. DPN. 32

4.1 TagFlow System Architecture 44

4.2 High-level illustration of FLARE switch 46

4.3 Tagging overhead (64-byte packets) 47

4.4 TagFlow versus OpenFlow versus vanilla forwarding using one CPU core 48

4.5 TagFlow versus OpenFlow versus vanilla forwarding using 2 CPU cores 48

4.6 TagFlow versus OpenFlow versus vanilla forwarding using 3 CPU cores 49

4.7 TagFlow versus OpenFlow versus vanilla forwarding using 4 CPU cores 49

4.8 TagFlow versus OpenFlow versus vanilla forwarding using 5 CPU cores 50

4.9 TagFlow versus OpenFlow versus vanilla forwarding using 6 CPU cores 50

4.10 TagFlow versus OpenFlow versus vanilla forwarding using 64-byte packets 51

4.11 TagFlow versus OpenFlow versus vanilla forwarding using 128-byte packets . . . 51

4.12 TagFlow versus OpenFlow versus vanilla forwarding using 256-byte packets . . . 52

4.13 TagFlow versus OpenFlow versus vanilla forwarding using 512-byte packets . . . 52

4.14 TagFlow versus OpenFlow versus vanilla forwarding using 1024-byte packets . . . 53

4.15 TagFlow versus OpenFlow versus vanilla forwarding using 1514-byte packets . . . 53

4.16 TagFlow throughput in Gbps using different number of CPU cores 54

4.17 TagFlow throughput in Mpps using different packet sizes 54
4.18 End-to-end experiment architecture 55
4.19 End-to-end TagFlow versus OpenFlow roundtrip latency for a single packet . . . 56
4.20 End-to-end TagFlow versus OpenFlow cumulative roundtrp latency 56
4.21 Number of packets processed in 1036 millisec using five different applications . . 57
4.22 Bothunter architecture 58
4.23 Architecture of HTTP header parser, modified IPSec Anti-replay checker and portscan detector 59
4.24 P2P Plotter architecture 59
4.25 Bothunter alternative scenario 1: Single intrusion prevention system 60
4.26 Bothunter alternative scenario 2: multiple intrusion prevention systems 61
4.27 Bothunter IDS style 62
4.28 Bothunter IPS style 62
4.29 Bothunter IDS style detection latency 64
4.30 Bothunter IPS style detection and reconfiguration latency 64
4.31 Bothunter API methods and parameters 65

5.1 Northbound application overhead evaluation architecture 74
5.2 NFV application overhead evaluation architecture 75
5.3 UDA overhead evaluation architecture 76
5.4 UDA/API control and data plane architecture 77
5.5 OpenFlow extended to support UDAs 82
5.6 Per attack detection time of Portscan Detector UDA on the attack trace from [1] 84

List of Tables

2.1 Current Research on Controller Management 15
2.2 Current Research on Management of SDN-enabled Networks 18
2.3 Current Research on SDN Debugging 21
2.4 Current Research on SDN Abstractions 23

4.1 Summary of Tagging Approaches 38
4.2 Northbound versus Southbound (i.e., TagFlow) Applications 66

5.1 Comparison of overhead in northbound (i.e., FRESCO) applications versus southbound user-defined actions versus virtual appliance middleboxes 73
5.2 Comparison of ease of programmability in Northbound applications versus southbound UDAs versus middleboxes 79
5.3 Comparison of ease of programmability in northbound versus southbound (i.e., UDAs) 80
5.4 Overhead of heavy User-Defined Actions using extended OpenFlow 83

Chapter 1

Introduction

1.1 Background

Current Internet infrastructure is a set of networking devices with purpose-built ASICs and chips that are used to achieve high throughput, so realizing *hardware-centric networking*. However, current hardware-centric Internet suffers from several shortcomings such as manageability, flexibly, and extensibility. Networking devices usually support a handful of commands and configurations based on a specific embedded OS or firmware. As a result, network administrators are limited to a set of pre-defined commands, even though it would be easier to support more protocols and applications in a simpler and more efficient way if they can program network controls in the way that they want. In addition, researchers usually have to make their own testbeds or take advantage of simulation rather than real world implementation scenarios to realize their ideas. In other words, innovation and research is costly in the current hardware-centric networking.

To overcome such limitations, Software-Defined Networking (SDN) concept has been proposed. SDN can be defined as "*an emerging network architecture where the network control is decoupled from the forwarding and is directly programmable*" [2]. In SDN, there is a logically centralized controller that has a network-wide view, and controls multiple packet forwarding devices (e.g., switches) that can be configured via an interface (e.g., ForCES [3] and OpenFlow [4]). For example, an OpenFlow switch has one or more forwarding tables that are controlled by

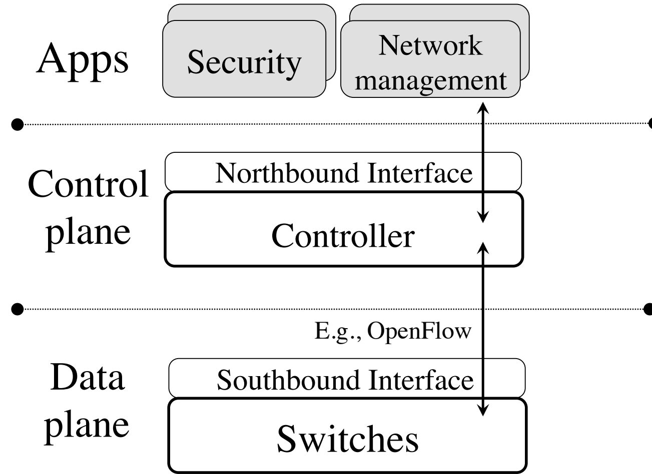

Figure 1.1: Components of SDN.

a centralized controller, so realizing the control plane programmability. Forwarding tables are used to control packets (e.g., forwarding or dropping). Therefore, according to the controller's policy that controls the forwarding tables, an OpenFlow switch can act as a router, switch, NAT, firewall, or similar functions that depends on packet-handling rules. Because of the decoupling, SDN is believed to be a new networking technology that simplifies today's network operation and management and also enables network innovations and new network designs. Due to the potential benefits of SDN in the current Internet and future Internet architectures such as information-centric networking [5], it has gained considerable attention from the community.

An SDN instance consists of three major parts: application, control plane, and data plane (Fig. 1.1). *Application* indicates the part that exploits the decoupled control and data plane to achieve specific goals such as a security mechanism [6] or a network measurement solution [7]. Application communicates with a controller at the control plane via *northbound* interface of the control plane. *Control plane* is the part that manipulates forwarding devices through a controller to achieve the specific goal of the target application. The controller uses *southbound* interface of the SDN-enabled switch to connect to data plane. *Data plane* is the part that supports a shared protocol (e.g., OpenFlow) with the controller and handles the actual packets based on the configurations that are manipulated by the controller. Therefore, we believe that deep understanding of each part and balanced research attention on each of the three parts is

important to maximize the potential benefits of SDN.

1.2 Problem Statement

The existing studies on SDN points out numerous solutions and optimizations to redesign different aspects of the SDN control plane. However, they leave a significant problem which is taking care of the SDN data plane. We posit that there are two major problems regarding SDN data plane to be solved in order to achieve an efficient and flexible SDN:

1.2.1 Packet classification is inefficient

Current proposals of SDN data plane consistently insist on the necessity of a hardware-centric data plane meaning that all the functionality should be implemented on the hardware. This point of view includes classification of packets as well. Almost any data plane device including switches and middleboxes have to classify packets before taking any other action or decision. The main consequence of hardware-centric packet classification is inflexibility in term of customizability and non-extensibility during the time. Hence, upgrading the protocols and functionality of the data plane which implemented on hardware solutions such as Ternary Content-Addressable Memory (TCAM) results in replacing the hardware module. While this guarantees a profit margine for chip producers and hardware makers, it is not a viable solution from scientific point of view. There are three specific problems in SDN packet classification:

- *Predefined Classification:* The packet classification mechanism is implemented on hardware and it is costly to adapt new protocols and architectures. So, all the supported protocols and header fields are predefined. There is a need for a more flexible and evolvable classification mechanism.

- *Complex Classification:* SDN classifies packets based on the packet header fields. The number of the fields used for classification can potentially be many and cause a complex and heavy classification.

- *Redundant Classification:* Finally, exactly the same form of classification happens on every switch a packet visits throughout its route to get to the destination. This classification is

repeated unnecessarily many times in order to let the switch decide where to forward the packet.

1.2.2 Switch actions are hardwired in the switch hardware

Current SDN data plane supports a set of few actions. Actions are atomic operations that are applied to every single packet by switch after classification. Therefore, once the packet enters to the switch, first the switch classified the packet based on the header fields and then applies a set of actions to the packet. Currently there are only a few actions supported by the dominant SDN data plane definition such as: *Forward*, which means forward the packet to some specific output port, *Drop* which means to discard the packet and finally *Meter*, which means to measure different aspects of the traffic and collect statistics. However, such a predefined and one-size-fits-all way of treating with networking may not be sufficient for fast paced and changing environments of todays' networks. Similar to the first problem, any attempt to change the action set of SDN data plane results in hardware module replacement.

Specifically there are two problems here:

- *Predefined actions:*all actions to be taken by the switch are predefined and implemented on hardware. So, deploying any additional and even small mechanisms such as minor packet modifications may lead to adding a middlebox (of course with some overhead) to the network. While tolerating the overhead of a new box on the network may be acceptable for very heavy proceedings such as sophisticated firewalling or heavy deep packet inspection, it may not be reasonable for minor tasks.

- *Shortage of actions:* The number of actions available in the current SDN solutions is limited. There reason is that they are defined on the hardware and thus it is costly to define many actions. Moreover, it is not easily possible to introduce programmability to actions and let the user define arbitrary actions.

1.3 Thesis Statement

This thesis posits that solving aforementioned problems in the SDN data plane is a key to the successful migration of current hardware-centric networking architectures to SDN. The benefits of such a migration lie in two aspects. First, for academia, using our proposals, researchers can test their proposed methods freely without any concern about dealing with predefined architectures and protocols of current hardware-centric networking solutions yet with a reliable performance. Second, for industry, as we relax the constraints of traditional predefined, hardwired data planes which lets new and small firms enter the networking software development market and conduct the industry to a more competitive and innovative atmosphere.

To be more specific we reveal the potential benefits of extending SDN packet switching architecture using two examples:

- First, a packet classification mechanism which is simple and low overhead (called TagFlow)
- Second, a packet processing mechanism within the switch which enables adding arbitrary processing of packets (called User Defined Actions)

Benefits of the above solutions are two folds:

- Primary benefit: We can adapt new architectures or protocols and basic packet processing without hardware replacement
- Secondary benefit: Reduced cost and latency

1.4 Strategy and Philosophy

We take a data plane oriented strategy towards solving the two problems mentioned earlier even though it is possible to attack to the problems from control plane centric approach as well. In particular, among available alternatives in the research community, as our fundamental basis, we focus on the idea of Deeply Programmable Networks (DPN) [8] to solve problems. We discuss the definition and details of DPN in Chapter 3 where we try to elevate the SDN definition to cover a broader range of concepts. Hence, from Chapter 3 and after explaining DPN we may use the term SDN instead of DPN as it is more familiar to the community. Since the SDN concept

territories is roughly defined, using DPN and SDN interchangeably could be safe. There is a philosophy behind our strategy which is out of the scope of this thesis. Therefore, we only review it briefly here. Our strategy is based on proposing open architecture and democratizing the ability to program networking devices. Currently there is a monopoly of hardware producers which are a few vendors that have the power to program networking devices and advertise that their (almost) one-size products fit all needs for every environment. We believe democratizing the access to networking controls and providing users with commodity programable networking devices is a key to unlock innovation on networking. That is, getting advantage of the crowed wisdom usually ends in a better solution.

1.5 Scope

OpenFlow in early days is initially proposed for campus networks. Since campus networks have some similarity with datacenter networks, the industry quickly adapted the same idea to datacenter networks. The main similarities between campus and datacenter networks is their limitation in size and management. They are both in single administrative domain that have the authority over all aspects of the network. Since main competitor of this research is traditional SDN and the dominant implementation of SDN is OpenFlow we define our scope to datacenters. Even thought, there are some activities about using OpenFlow in optical and wide area networks they are not the initial purpose of OpenFlow and thus we do not cover them. For more information on OpenFlow application in wide area and optical networks please refer to Chapter 2. In short, the scope of all networks mentioned in this research is datacenter networks unless otherwise is stated.

1.6 Summary

Chapter 1 provides a general introduction of SDN and the problems it is facing.

Chapter 2 introduced Software-Defined Networking and reviews current significant research efforts in this area and classifies the literature in different classes. It also illustrates the gap in the current research focus which is the data plane related research on SDN.

Chapter 3 starts with the gap introduced in the previous chapter and defines the superset of SDN as Deeply Programmable Network (DPN). Then it covers possible chances to catalyze the speed of DPN research specifically on the data plane using already existing technologies.

Chapter 4 moves the discussion to the solution domain and proposes TagFlow, and efficient tag-based flow classification for DPN.

Chapter 5 proposes User-Defined Actions (UDA), a flexible architecture to accommodate user generated packet processing mechanisms within the switch data plane.

Chapter 6 concludes the thesis and proposes future work.

Chapter 2

Software-Defined Networking (SDN)

2.1 Introduction

The current Internet consists of networking devices with purpose-built ASICs and chips that are not easily programmable, so realizing hardware-centric networking. The hardware-centric networking causes a barrier towards its growth because any modification that goes deep into architecture can not be deployed and tested easily. For example, networking devices usually support a handful of commands based on a specific embedded OS or firmware. As a result, network administrators are limited to a set of pre-defined commands, even though it would be easier to support more protocols and applications in a simpler and more efficient way if they can program network controls in the way that they want. In other words, the current hardware-centric Internet suffers from several shortcomings such as manageability, flexibiliy, and extensibility.

Software-Defined Networking (SDN) is now becoming more and more popular because it is considered as a prominent method to facilitate networking practices due to its two features, i) decomposition of control and data planes and ii) a centralized controller based on a network-wide view. Those SDN characteristics provide control plane programmability, support networking

applications, and enable research and innovation in networking.

Even though SDN is considered as one promising way to solve limitations of the current hardware-centric networking, its potential benefits come at several challenging issues (e.g., availability and resilience of the centralized controller). In this paper, we first introduce a survey of current research that tries to solve the identified challenges of SDN, particularly on management aspects. Even though most current research on SDN focuses on the programmability of control plane that manipulates the data plane handling packets (i.e., switches), we argue that research on the data plane programmability is also required.

2.2 Software-Defined Networking

Back in the 1980s, a router was simply a server forwarding packets to/from a few NICs. As the complexity and capacity of networks improved, general-purpose hardware was not capable of handling packets fast enough and thus software routers on servers were hardwired into single-purpose devices that we recognize as *network controls* today. That is the status of current *hardware-centric networking.* The image of a network router for example, in the mind of an average networking engineer is a hardware appliance with purpose-built ASICs and chips as well as firmware which is not flexible enough to be customized dynamically. With this paradigm shift from software to purpose-built hardware, networking devices achieve higher throughput and yet few shortcomings as well. Because of hardware-centric networks, network management tasks are not flexible, modifiable, and generic enough. For example, network management tasks highly depend on measurement of network traffic. But, unfortunately traffic measurement typically depends on a set of customized hardware and pre-defined protocols designed for specific purposes.

Networking paradigms can be divided into three types according to the deployment of control and data planes (Fig. 2.1). In the traditional hardware-centric networking, switches are usually *closed* systems that have their own control and data planes and support manufacturer-specific control interfaces. Therefore, in the traditional hardware-centric networking, deploying new protocols and services (and even new versions of existing protocols) is a challenging issue because all the switches need to be updated or replaced. In contrast, in SDN, switches become

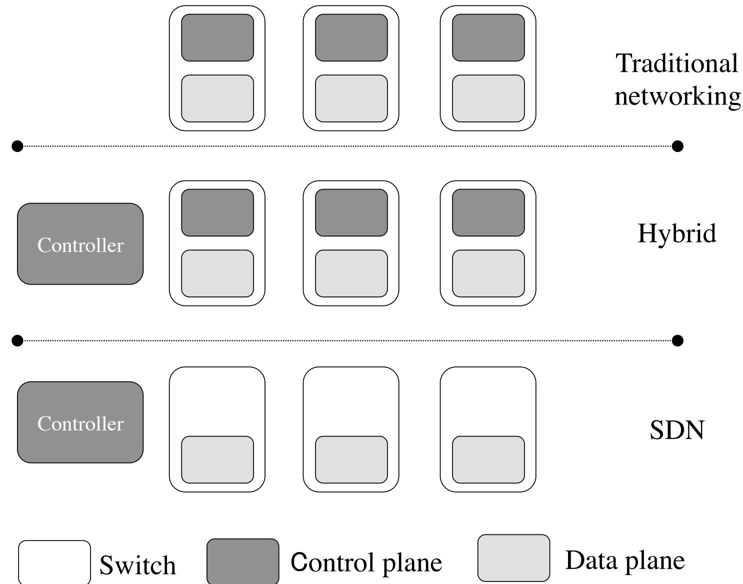

Figure 2.1: High-level illustration of networking paradigms.

more simpler (in terms of removing the control plane from the device) forwarding devices while a centralized controller derives the control mechanism of the network. This decomposition of control and data planes allows easier deployment of new protocols and services because the decomposition enables us to *program* switches via the controller. Finally, hybrid approach supports both distributed and centralized control planes. For example, common commercial OpenFlow switches are hybrid switches that support OpenFlow in addition to traditional operation procedures and protocols.

SDN is considered as one promising way to overcome such limitations. Recently, SDN is now becoming more and more popular based on the full support of industry and academia even though the concept itself is not new. Fig. 2.2 illustrates the high level components of an SDN network. At the lowest level, we have the data plane that consists of switches. The data plane is responsible for dealing with actual packets and transmitting them among nodes according to the forwarding rules that are manipulated by controllers. On top of data plane, there are some controllers that manage data plane controls based on a network-wide view. Network applications work on top of controller and use one or more controllers to gather network information and possibly impact the controllers behavior to achieve specific goals (e.g., network manage-

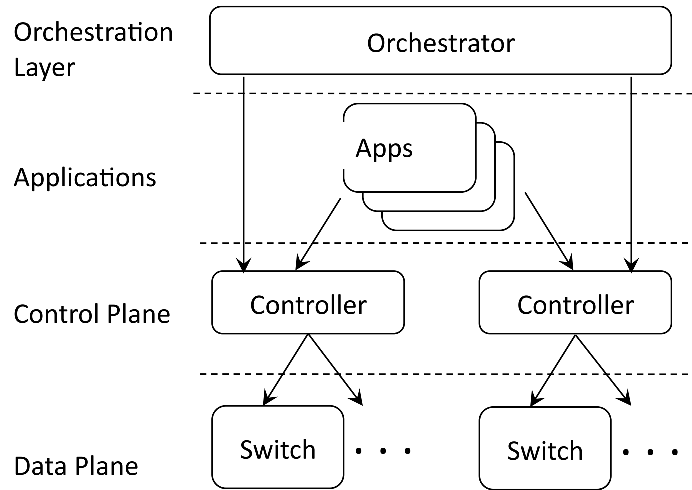

Figure 2.2: SDN overview

ment or security). At the highest level, there is an orchestrator that is in charge of keeping the network-wide state consistent among multiple controllers. The orchestrator realizes the logically centralized control over the network. Orchestrator may communicate with applications depending on their functionality.

SDN facilitates networking practices due to its two features, i) decomposition of control and data planes and defining an open interface between them and ii) a centralized controller based on a network-wide view. With SDN concept, it is possible to realize the centralized control of the network based on a network-wide view, which leads to a simpler and more efficient network control and management. The definition of an open interface along with the separation of the control and data planes allows customized control. However, the potential benefits of SDN come at the cost of several challenges and thus there have been much research on solving those challenges to fully exploit the potential benefits of SDN.

2.3 Current Research on SDN

2.3.1 Controller and Switch

In SDN, the controller is a key component because physically or logically centralized controller(s) manages the whole network by manipulating switches based on a network-wide view.

As a result, a way to manage, install, and implement the controller heavily affects the performance of network. Because of this reason, much research focuses on improving the controllers in various aspects. We group existing work into subgroups according to the goal that they want to achieve as follows. The following survey is summarized in Table 2.1.

2.3.1.1 Availability and Resilience

Unlike the current networks where control plane and data plane are tightly coupled and thus switches can act independently, in the centralized control of SDN, controllers and switches need to communicate with each other. Therefore, in SDN, the placement of controllers may affect the availability of SDN because it influences the communication between controllers and switches.

In [9], the authors try to understand how the number of controllers and their placement influence the reliability of SDN control networks. By introducing a metric, called expected percentage of control path loss, they formulate the reliability-aware control placement problem. In addition, they show that a strategic controller placement improves the reliability of SDN control networks noticeably without introducing unacceptable switch-to-controller latencies. In [10], the authors analyze the effect of controller placement on network resilience based on interdependence graph and cascading failure analysis. Based on their analysis, they propose a partition and selection approach for improving the resilience. In SDN with multiple controllers, placement of controllers may affect the overloads of controllers and the propagation delays among controllers [11]. To optimize the management of the network, the authors define principles for designing a scalable control layer for SDN. Then, they introduce one approach that finds the minimum number of controllers and corresponding locations so as to deal with failures robustly and balance the load among the selected controllers. In [12], the authors introduce the fault tolerant controller placement problem that determines how many controllers need to be used and which network nodes need to be managed by each of them. Through simulations with publicly available network topologies, they show that realizing the fault tolerant SDN is possible through careful placement of controllers (e.g., they show that each switch needs to connect to 2 or 3 controllers to achieve more than five nine reliability).

How to implement the controller and switches or how to setup SDN may also affect the availability and resilience. In [13], the authors aim to study a replication technique that is

a proven method for achieving the resilience. With two types of replication techniques (i.e., passive and active replication), they try to understand the difference between two types, how each technique can be used to improve the resilience, and which technique is better for specific cases. In [14], the authors introduce an approach to spawn a new controller instance to recover the failure. In addition, they dicuss a way to replay inputs observed by the old controller. A similar approach is used in [15] to support the upgrade of SDN controller to a new version in a disruption-free manner. Using a history of network events managed by the old controller, the new controller is bootstrapped.

In [16], the authors investigate the trade-offs among consistency, availability and partition tolerance in the context of SDN. In particular, they prove that it is impossible to enforce consistency and partition tolerance without sacrificing availability. They also discuss some ways to avoid these impossibility results. In [17], the authors introduce CPRecovery component that provides resilience against several types of failures of the centralized controlled network baed on the primary-backup mechanism. RuleBricks [18] introduces three key primitives (i.e., drop, insert, and reduce) to embed high availability support in existing OpenFlow policies. The authors describe how these primitives can express various flow assignment and backup policies.

Even though frequent issue of monitoring messages is needed for fast recovery, frequent report may pose a significant load on the controller. In [19], the authors discuss a way to implement a monitoring function on the switches so as to enable switches to send monitoring messages without posing a noticeable load on the controller. In [20], the authors propose a way to install static rules on the switches in order to verify topology connectivity and locate link failures. CORONET [21] is an efficient fault tolerant system to solve the data plane faults (i.e., a switch or link fails) with multi-path support. By computing link-disjoint shortest routing paths based on the up-to-date information of network topology status, CORONET recovers from multiple link failures. In [22], the authors aim to realize a fast failover mechanism for OpenFlow networks. Based on graph search-related approach, they discuss three algorithms to computer failover tables that achieve tradeoffs among the number of required failover rules, the number of tags, and the resulting path lengths.

2.3.1.2 Scalability

Even though the centralized control of SDN has various advantages, it naturally faces scalability issue (e.g., because of excessive control traffic overhead). Some work aims to discuss the necessity of the research on the scalability of SDN. In [23], the authors argue that more attention should be paid to improve the scalability of SDN by showing the impact of the flow insertion rate on the controller's CPU utilization and the packet loss rate. In [24], the authors discuss several scalability trade-offs in SDN design space.

One promising way to solve the scalability issue is to reduce the overhead of the centralized controller. In DIFANE [25], a centralized controller partitions the forwarding rules and distributes the partitioned forwarding rules to switches. With the distributed forwarding rules, DIFANE allows the switches to handle all traffic in the data plane by selectively directing packets through other switches that store the necessary forwarding rules. A similar approach is found in DevoFlow [26]. In DevoFlow, switches handle short flows and direct only large flows to a centralized controller. In [27], the authors combine centralized control and distributed control to provide a more scalable control. Defining new features (e.g., the proactive flows to be activated under a certain condition), they aim to reduce the overhead of the centralized controller. SUMA [28] is a switch-side inline middlebox that provides management abstraction and a filtering layer among SDN controllers and switches. In particular, the filter module of SUMA facilitates control message/traffic aggregation to reduce the overhead of the controller.

Using a hierarchical architecture of controllers is another promising way to tackle the scalability issue. In [29], the authors model the behavior of a scalable SDN deployment where local controllers handle frequent events and a centralized controller handles rare events. Using the network calculus and queuing theory, they capture various aspects of the scalable SDN deployment (e.g., the closed form of the event delay and buffer length inside the local SDN controller). In Kandoo [30], controllers form a two layers of controller so as to limit the overhead of frequent events on the control plane. Controllers at the bottom layer try to handle most events using the state of a single switch. A logically centralized controller of the top layer is only used to handle the events that cannot be handled by the controllers at the bottom layer.

Table 2.1: Current Research on Controller Management

	Detailed Area	Summary
Controller	Availability & Resilience	Placement of controllers ([9], [10], [11], [12]) How to implement/setup controllers and switches ([13], [14], [15], [18], [20], [19], [17], [21], [16], [22])
	Scalability	Reducing the overheads of a centralized controller ([25], [26], [27], [28]) Using a hierarchical architecture of controllers ([29], [30])
	Security	Constraining SDN applications ([31], [32], [33]) Handling controllers under attack ([34], [35], [36])
	State Management & Policy Enforcement	Rewriting forwarding rules ([37], [38], [39], [40]) Using transactional semantics ([41], [42]) Updating in parallel ([43]) Using a database ([44], [45])
	Flexibility	Allowing additional network information ([46], [47, 48]) Implementing the required functions in a flexible way ([49], [50], [51], [52])

2.3.1.3 Security

SDN may suffer from trust issues on SDN applications because it allows third-party development. The abuse of such trust may result in various attacks in SDN network. Some research aims to understand the potential vulnerabilities of SDN. In [53], the authors first discuss threat vectors that may enable the exploit of SDN vulnerabilities. Then, they introduce the design of a secure and dependable SDN control platform. In [54], the authors discuss a new attack to fingerprint SDN networks. They also introduce a possibility of resource consumption attacks (e.g., DDOS-like attack by issuing many requests to centralized controller supporting reactive-mode).

One natural way to deal with the potential trust issue is to constraint SDN applications. In [31], the authors first discuss several vulnerabilities of OpenFlow protocol. Then, they propose PermOF that applies minimum privilege on the applications. The authors introduce a set of permissions to be enforced at the API entry of the controller. For the security constraint enforcement in NOX OpenFlow controller, the authors in [32] propose the role-based authorization. FRESCO [33] provides a programming framework to execute and link together security-related applications.

Another security concern may arise when a controller is compromised or under attack. In [34], the authors introduce a secure SDN structure where each switch is managed by multiple controllers that are dynamically managed by the cloud. By deploying Byzantine fault tolerance

replicas in different instances in the cloud, they aim to guarantee that each switch updates the flow table correctly even when some compromised controllers issue false instructions. In [35], the authors propose an on-line disaster management framework for SDNs, called NEOD. NEOD event detector embeddded in switches monitors the disaster events according to the configured policies. Then, NEOD manager that resides in the controller performs a network-wide disater event corrrelation to find the root of potential attacks. Fleet [36] is proposed to solve the malicious administrator problem where network administrators attempt to attack the network by misconfiguring controllers. Fleet allows administrators to upload their configurations to the shared storage and only some selected configurations to be translated into flow rules and pushed into the switches. In this context, Fleet applies two approaches (i.e., single-configuration approach and n routing configurations) to address the malicious administrator problem.

2.3.1.4 State Management and Policy Enforcement

For consistent packet handling, a centralized controller needs to enable switches to have the same forwarding rules. Some research aims to understand the state consistency in various aspects. In [55], the authors show that the control state inconsistency significantly degrades the performance of logically centralized control applications. In [56], the authors study the trade-off between update time and rule-space overhead. In [57], the authors discuss the trade-off between maintaining consistency during configuration updates and the update performance.

One common approach for the state consistency is to rewrite forwarding rules for switches while preserving the overall forwarding policy. FlowAdapter [37] converts the forwarding rules of the controller to switch hardware flow table pipeline so as to allow the same forwarding policy to be fitted into different types of hardware. In [38], the authors propose a set of axioms for policy transformation to allow changes of rules across multiple switches while keeping the same forwarding policy. In Palette distribution framework [39], a large forwarding table is divided into small ones to be distributed across the network while preserving the overall policy semantics. In [40], the authors propose an efficient rule-placement algorithms to distribute forwarding policies across the network while managing rule-space constraints.

Another way for managing states over the network is to use transactional semantics. The reactive establishment of paths can cause inconsistent packet processing. To handle this issue,

in [41], the authors adopt the transactional semantics at the controller. In a similar spirit, in [42], the authors propose a policy composition abstraction to support concurrent and consistent policy updates when SDN policy specification and control is distributed. They aim to support all-or-nothing semantics using a transactional interface. For example, one policy update is done successfully so that switches of the entire network support the newly updated policy or not.

Some approaches exploit a database for the consistency. ONOS [44] supports a network topology database to the controller so as to preserve the consistency across distributed state. In [45], the authors also use the idea of a network information base to satisfy the state consistency and durability requirements.

To facilitate installing or modifying a large number of rules in a time efficient way, ESPRES [43] divides a large network-state update into sub-updates that are independent of each other and thus can be installed in parallel. ESPRES calculates an order of install of sup-updates so as to fully utilize processing capacities of switches without affecting the performance of switches. In SDN, the dynamic and traffic-dependent modifications by middleboxes may make it difficult to ensure network-wide policy enforcement. To overcome this problem, FlowTag [58] allows middleboxes to add tags to outgoing packets so that packets carry the necessary causal context and thus switches and other middleboxes examine the tag to examine the systematic policy enforcement.

2.3.1.5 Flexibility

Additional information may be able to make the network management easy. To enrich network management services, FleXam [46] allows OpenFlow controller to have the packet-level information. In [47, 48], the authors aim to incorporate application-awareness into SDN that is currently agnostic to applications.

The flexibility in implementing the required functions may also be able to make the network management easy. To realize far more flexible processing of counter-related information, the software-defined counters that utilize general-purpose CPU rather than ASIC-based inflexible counters is introduced in [49]. In a similar spirit, CPU in the switches is used to handle not only control plane but also data plane traffic to overcome ASIC-based approach's limitations [50]. In [51], the authors present research directions that can significantly reduce TCAM and

Table 2.2: Current Research on Management of SDN-enabled Networks

	Detailed Area	**Summary** (Reference list)
Networks	Virtualized Network	Managing a large number of SDN instances ([61], [62], [63], [64], [65]) Managing SDN in the virtualized carrier network ([66], [67])
	Large-scale Network	Supporting inter-domain interoperability between SDN-based ASes ([68]) A state distribution system ([45]) Applying SDN into access network ([69])
	Data Center Network	Reducing the complexity of management ([70]) Managing traffic aggregation ([71])

control plane requirements via classifier sharing and reuse of existing infrastructure elements. In [59], the authors propose the reconfigurable match tables that allows the forwarding plane to be changed without modifying hardware. The reconfigurable match tables also allow the programmer to modify all header fields much more more comprehensively than in OpenFlow. In [60], the extensible session protocol is proposed to install flow entries proactively based on the requirements of the application. The authors aim to provide a general and extensible protocol for managing the interaction between applications and network-based services and between the devices. To simplify the network management by allowing applications developed for different controller platforms, the authors in [52] propose a new kind of hypervisor that allows different applications to process the same traffic. Using a extensible configuration language, they aim to combine updates to each prioritized list of forwarding rules according to the hypervisor policy.

2.3.2 SDN-enabled Networks

Even though SDN facilitates networking practices based on the decomposition of control and data planes and the centralized control with a network-wide view, SDN features may introduce new challenging issues to the management. In this subsection, we introduce existing research that tries to improve the management of various SDN-enabled networks. The following survey is summarized in Table 2.2.

2.3.2.1 Virtualized Network

Using the network virtualization technique [72], several SDN instances can be created over the same physical network. However, the scalability may become an issue in supporting a large number of SDN instances (e.g., because of significant configuration overheads). In [61], the authors introduce SDN hypervisor. SDN hypervisor generates the required flow entries to transparently setup arbitrary virtual SDN and manipulates control messages so as to allow each virtual SDN operator to configure its own virtual SDN. In addition, SDN hypervisor supports the automated node and link migration through flow table updates. In a similar spirit, for scalable SDN slicing, AutoSlice [62] introduces a distributed hypervisor architecture to handle large number of flow table control messages from many SDN instances. In FlowN [63], a database technology is used to efficiently provide each SDN instance with the information of its own topology and controller instead of running a separate controller for each SDN instance. In [64], the authors aim to provide an integrated way for managing several SDN instances using distinct network operation systems. They introduce SDN mashup concept that lets administrators create SDN management solutions to meet their own requirements so as to cope the heterogeneity of virtual resources on SDN. With the SDN mashups, administrators can customize and combine their SDN management tools in a high-level abstraction. In [65], the authors introduce a novel mixed integer programming formulation for coordinated link and node mapping for SDN. Allowing administrators to embed virtual network requests according to predefined policies, the algorithm maximizes the number of virtual network requests that can be optimally embedded into a given substrate.

In the context of a virtualized carrier network shared by multiple customers, the authors in [66] analyze the applicability of the current SDN model specified by ONF by discussing the required procedures for configuring and managing the virtualized network. Then, they reveal some shortcomings of the current SDN model and discuss possible extensions including updates to the SDN and NOS model and extensions of the management data model. In the same context, the authors in [67] propose an autonomic management architecture for SDN-based multi-services network. The autonomic management architecture is designed to satisfy different requirements of network management and to deploy different type of services quickly

by utilizing autonomic management and network virtualization technologies.

2.3.2.2 Large-scale Network

In the context of wide area SDN where multiple controller solutions are natural, the authors in [73] discuss various design alternatives for multi controllers (e.g., horizontal vs. vertical setup for multi controller architecture). They also present a possible management architecture for a single administrative wide area SDN. In addition, they elaborate to explain three main functions (i.e., distributing topology information, flow setup, and monitoring) in the presented management architecture. In [68], the authors first argue that the nice features of SDN architecture are lost in the constraints that BGP imposes on inter-domain routing. Then, to support inter-domain interoperability for SDN-based ASes with both traditional ASes and other SDN ASes, they introduce a concept of horizontal slicing witihin an SDN. The horizontal slicing enables two SDN-based networks to be connected without revealing their exact network information such as topology while allowing independent routing and policy control approaches are used for each SDN-based network. In [45], using standard distributed system design practices, the authors introduce a control platform, called Onix that allows the control plane to be run on a network-wide view and to use basic state distribution primitives. With this approach, they aim to satisfy several important requirements for a production-quality control platform such as scalability, reliability, and control plane performance in the large-scale production networks. In [69], the authors try to extend the benefits of SDN and NFV into broadband access network by introducing software-defined access network (SDAN). Using the virtualized broadband access and the streamline operations, SDAN aims to handle the issues caused in broadband access network (e.g., the customer broadband behavior that are becoming increasingly sophisticated).

2.3.2.3 Data Center Network

In the SDN-based data center networks, the authors in [70] try to reduce the complexity of management caused by information overwhelming. For this, they introduce a concept of network aggregation (i.e., regional networks on lower layers are treated as single switches to upper layers) and information division mechanism (i.e., management information is divided into three parts and each part is only visible to network managers, regional controllers, or tenants). For scalable

Table 2.3: Current Research on SDN Debugging

	Detailed Area	**Summary** (Reference list)
Debugging	Debugging Network	Checking network-wide problems ([75], [76], [77], [78], [79], [80])
	Debugging Controller	Finding events that lead to network errors ([81], [82], [83], [84], [85])
	Available Tools	Network-level debugging (OFRewind [86, 87]) Debugging OpenFlow controllers (Cbench [88], NICE [89, 90], STS [91]) Debugging OpenFlow switches (Oflops [92, 93], OFTest [94])

and flexible traffic management in OpenFlow-enabled data center network, in [71], the authors argue that the controller needs to exploit the wildcard rules to direct large aggregates of client traffic to server replicas in a more scalable way. They discuss several approaches to compute wildcard rules in order to achieve a target distribution of the traffic and to adjust to changes in load-balancing policies. In [74], the authors study the storage aspect of the SDN-enabled data network and investigate the potential challenges.

2.3.3 Debugging

It is important to understand weak points and potential errors of SDN implementations to make SDN robust and stable. The following survey is summarized in Table 2.3.

2.3.3.1 Network

VeriFlow [75] is proposed to check network-wide invariants in real time even when the network state evolves. As a layer between the controller and the switches, VeriFlow examines the network-wide invariant violations whenever a new forwarding rule is inserted. In particular, VeriFlow aims to achieve extremely low latency for the checks not to affect the network performance. ATPG [76] is proposed to test and debug networks including SDNs. Given the router configurations, ATPG generates a device-independent model that is used to generate a set of test packets to exercise every link or rule in the network. Using the test packets, ATPG detects functional (e.g., incorrect forwarding rule) and performance problems (e.g., congested queue). HSA [77] aims to identify an important failures (e.g., forwarding loops and traffic isolation) by checking network specifications and configurations regardless of the protocols running. To

achieve this, in HSA, the entire packet header is treated as a bit string while ignoring the associated meaning. In [78], to verify and debug SDN applications with dynamically changing verification conditions (e.g., dynamic access control), the authors introduce an assertion language. The assertion language enables programmers to use regular expressions to describe various properties about the data plane. Whenever the controller adds or removes elements such as switches, it generates new verification conditions that the data plane must meet. In [79], the authors propose a framework to modify the controller program transparently using the graph transformation rather than using the manual code modification. In addition, for the network debugging, they introduce a storage system to log flow entries and corresponding parameters. The archived flow records enable a administrator to detect network anomalies and to perform forensic analysis.

2.3.3.2 Controller

To debug the controller, in [81], the authors propose the cross-layer correspondence checking (to find what problems exist and where in the control software the problem first developed) and the simulation-based causal inference (to identify the minimal set of events that triggered the problem). *ndb* [82] (similar to gdb used for debugging software programs) pinpoints the sequence of events that lead to a network error by using familiar debugger actions such as breakpoint and backtrace. In [83], the authors introduce a simulation-based tool, called fs-sdn, to facilitate prototyping and evaluating new SDN applications. In particular, fs-sdn aims to support that job at large scale and tries to enable easy translation to real controller platforms. In [84], the authors aim to detect and resolve control conflicts by proposing a new programming model for SDN controllers. In the new model, a uniform presentation of the operational objectives is used to enable controllers to execute the controller code only when that code is expressed independently of one another. VeriCon [85] is a system for verifying SDN controller. VeriCon examines the correctness of the controller on all admissible network topologies or the correctness of execution of any single network event with given specified network-wide invariant. To achieve this in a scalable way, VeriCon uses first-order logic to specify network topologies and network-wide invariants and then implements verification using existing tools.

Table 2.4: Current Research on SDN Abstractions

	Detailed Area	Summary (Reference list)
Abstractions	Declarative Language	Using declarative language ([95], [96], [97])
	Achieving Specific Goals	Fault-tolerant network program ([98]) OpenFlow security application development ([33]) Traffic isolation ([99]) Reducing data-collection overhead for debugging ([100])
	Misc.	Overcoming the challenge of translating a high-level policy into sets of rules ([101]) Abstractions of five main versions of OpenFlow ([102]) Protocol-oblivious forwarding ([103])

2.3.3.3 Available Tools

There exist available debugging tools. OFRewind [86, 87] is a tool for network-level troubleshooting and debugging at multiple levels. OFRewind acts as a transparent proxy between the controller and the switches. With OFRewind, a administrator can record the control and data traffic and replay it at an adaptive rate for the purpose of debugging. Some tools focus on debugging OpenFlow controllers. Cbench [88] emulates a bunch of OpenFlow switches and generates packet-in events for new flows to test the controllers. NICE [89, 90] aims to test the controller to identify potential programming errors that make communications less reliable by using the combination of the model checking and symbolic execution. SDN troubleshooting system (STS) [91] is proposed to find the problematic codes that lead the controller software to break in an automatic way. For this, STS simulates the devices of a given network while allowing a administrator to programmatically generate tricky test cases. Some tools can be used for debugging OpenFlow switches. Oflops [92, 93] can be used to quantify an OpenFlow switch performance in various aspects. Oflops supports a modular framework for adding and running implementation-agnostic tests. OFTest [94] is a Python-based framework to test basic functionality of OpenFlow switches.

2.3.4 Abstractions

Proper abstraction is useful to make development, management, and debugging easy. The summary of this section is summarized in Table 2.4.

2.3.4.1 Declarative Language

The current configuration languages for OpenFlow systems are not so expressive enough to capture dynamic and stateful policies. To overcome this limitation, some research exploits a declarative language. Procera [95] supports a declarative policy language. Procera provides the expression of high-level network policies in various network settings and of temporal queries over events that occur in the network policies. FML [96] also supports the declarative policy language for managing the configuration of enterprise networks. FML aims to balance the desires to express policies naturally and enforce policies efficiently. Frenetic [97] is a high-level language that supports to program distributed collections of network switches. Using a declarative query language, Frenetic allows to classify and aggregate network traffic and to describe high-level packet-forwarding policies. In particular, Frenetic aims to facilitate modular reasoning and code reuse by providing compositional constructs.

2.3.4.2 Achieving Specific Goals

To enable fault-tolerant network programs, FatTire [98] supports a programming construct based on regular expressions. FatTire enables a programmer to specify the set of paths that packets may take through the network as well as the degree of fault tolerance required. FRESCO [33] is an OpenFlow security application development framework. FRESCO aims to facilitate the modular composition of OpenFlow-enabled detection and mitigation modules. In [99], for achieving the traffic isolation, the authors propose an abstraction to support programming isolated network slices by processing packets of one slice independent from all other slices. They discuss the slice abstraction for this and develop the algorithms for compiling slices. Some work tries to achieve specific goals based on the abstraction. For better SDN debugging and troubleshooting without causing large data-collection overhead, in [100], the authors propose a query language. Using regular-expression-based path language, SDN applications can specify queries of the forwarding state. Then, based on the regular expressions, the packet trajectory is tracked on the data plane by tagging the required information in each packet as it goes through the network.

2.3.4.3 Misc.

In [102], the authors first argue that the implementations of OpenFlow is far from simple (e.g., various active versions and unstable version negotiation) even though the main idea of OpenFlow is simple. To mitigate this issue, the authors summarize the core abstractions of five main versions of OpenFlow and make simple API supporting the summarized core abstractions. Maple [101] supports an abstraction that runs on every packet entering the network to overcome the challenge of translating a high-level policy into sets of rules on distributed switches. To provide higher level abstractions for the south-bound interface while allowing switches to handle all the detailed work, in [103], the author propose the protocol-oblivious forwarding that utilizes the generic flow instruction set.

2.4 Summary

In this chapter we first define SDN and then extensively review the current literature to show the focus of the research. As the conclusion of our findings throughout our survey, the is a sever lack of research on data plane compared to control plane. While control plane is well investigated by the community, there is a little interest on the data plane. In the next chapter as the fulfillment of this gap we define the superset of SDN and discuss possible topics of research in the area.

Chapter 3

Deeply Programmable Networks (DPN)

In this chapter, we review data plane-related research in SDN. The main objective of this chapter is to show two facts: i) data plane programmability has potential benefits, and ii) there are a series of data plane technologies that can be applied to SDN data plane so as to extend it in different ways.

3.1 Necessity of Data Plane Programmability

We believe that SDN requires both control and data plane programmability. As we witness in many existing work, the control plane programmability realizes flexible control forwarding logic of network with less complexity. But, we also argue that data plane programmability is required for a more flexible and comprehensive coverage of applications in SDN. For example, a software-defined network measurement solution running on commodity hardware and supporting generic measurement tasks needs to touch the data plane (e.g., OpenSketch [7]). To add more examples, supporting a new protocol and architectures such as Named Data Networking (NDN) [104], eXpressive Internet Architecture (XIA) [105] and new layer 2 [8], the network may need a change in the data plane. In this regard, current hardware-centric data plane suffers from inflexibility.

On the other hand, we believe that data plane programmability may be able to satisfy those requirements relatively easily and flexibly. In addition, data plane programmability enables various functions such as network appliances (e.g., for deep packet inspection), in-network processing (e.g., cache and transcoding). We believe commodity hardware are now capable to afford such a task efficiently and flexibly. Given current advances in commodity hardware industry, we believe that the idea of a programmable data plane is possible to realize. In fact, advent of general-purpose hardware industry that has achieved significant improvement from general purpose CPUs to multi-CPU machines and then to multi-core on a chip (e.g., 100 cores of general purpose CPUs on a single board [106]) can unlock innovation in data plane from hardware-centric approach barriers.

Even though Open Networking Foundation (ONF) [107] is still in development stages of concepts such as Protocol independent/oblivious/agnostic Forwarding (PiF, PoF, PaF), they are the direct product of data plane programmability. In contrast, OpenFlow is now based on a match/action mechanism in which the packet header is matched against a pre-defined set of header fields (e.g., source and destination IP addresses) and then based on the matching result, some actions (e.g., drop or forward) may be executed on the packet. Thus, we may need a departure from current <pattern-match, action> architecture of SDN to a more flexible schemes based on data plane programmability in addition to control plane programmability. There is a limited contribution from SDN community in this area (e.g., FLARE [108]). We believe that data plane programmability should come to the forefront in addition to control plane in SDN research.

3.2 OpenFlow Data Plane

There are a few researches that address OpenFlow data plane shortcomings from different points of view. In [59] authors design a chip as a TCAM alternative. They propose a new RISC-inspired architecture for switching chips that allows the SDN data plane programmers to modify all header fields to be matched much more comprehensively than in OpenFlow. In [109], the authors apply network processor-based acceleration cards to perform OpenFlow switching. They show a 20% reduction on packet delay and a comparable packet forwarding throughput compared to

conventional designs. In [110], the authors propose an architectural design to improve lookup performance of OpenFlow switching in Linux using a commodity NIC. In [111], the authors compare OpenFlow switching, layer-2 Ethernet switching and layer-3 IP routing performance in terms of forwarding throughput and packet latency in underloaded and overloaded conditions with different traffic patterns. Some researchers suggest adding a commodity/programmable hardware such as FPGA to the data plane to enable SDN programmability extended to data plane [112]. There are also some radical approaches inspired by active networks to inject small applications into the packets to be executed by the SDN data plane [113]. In [51], the authors present research directions that can significantly reduce TCAM and control plane requirements via classifier sharing and reuse of existing infrastructure elements.

3.3 Potential Data Plane-related Proposals for SDN

There is a group of proposals that are considered out of the main stream focus in the SDN community or they are not proposed for SDN at all. These technologies that focus on different data plane functionalities can be applied to the current SDN and extend the scope and definition of existing hardware-centric SDN proposal. In this subsection, we try to elevate them and express their important role in the development of a software-centric SDN data plane.

3.3.1 Packet Classification and Forwarding

Software based packet switching and forwarding is the most basic and fundamental requirement for a software-centric data plane. There are many proposals for a software forwarding plane that mainly focus on the performance aspects of the research using different underlying commodity hardware such as CPU (e.g., RouteBricks [114]), GPU (e.g., PacketShader [115]), NPU (e.g., FLARE [108]) or FPGA (e.g., HyperSplit [116]). RouteBricks is a software router using a full mesh of interconnected PCs. It uses some form of randomized order to pass the packet from one router box to another in the mesh. PacketShader use GPU to accelerate the core packet forwarding function. It considers CPU as a performance bottleneck and thus offloads computation and memory-intensive tasks of packet forwarding to GPU. FLARE switch is a programmable switch using Click environment and multicore CPUs. It has a couple of

SFP+ ports and provides a Linux and Click environment for network research. FLARE uses many core NPUs to run the packet forwarding and processing routines in concurrent manner. HyperSplit claims 100 G/s of packet classification which is component of every forwarding fabric. It designs a pipelined classifier architecture on the FPGA that make an overall classification tree. L7Classifier [117] is a packet classification method based on the packet payload (i..e, application layer) compared with traditional methods that use L2-L4 header information. L7Classifier stores TCP flow information and performs regular expression matching to packet payload. Such as technology helps developing a classification method for SDN data plane that can include unforeseen future requirement throughout the flexibility it provides.

3.3.2 Easy Programmability

Every software network control (e.g., a switch or router) can be considered as programmable. But, the design of the control can vary if the control is made to be *easily* programmable and extensible while in other cases some other priorities (e.g., throughput) come to the top of the list. Click modular router [118] is a well-known highly programmable router including many elements written in C++. Click owns an active development community and serves as the base of many research projects. The ideas behind Click that provides many abstractions over typical networking task such as packet manipulation borrowed in many other projects either directly (e.g., ClickOS [119]) or indirectly (e.g., in FRESCO [33] that uses the overall principals of Click modularity). In [119], the authors shift focus towards making the data plane more programmable by introducing ClickOS, a tiny, Xen-based virtual machine that can run a wide range of middleboxes. So, such ideas can be used to implement an SDN data plane when there is a need to design an environment in which we can add and remove features very fast and dynamic. vNode [120] is a programmable network node that mainly focuses on data plane programmability and uses network virtualization technology. There is network of vNodes that form a virtual testbed laid all over JGN-X network in Japan using 40G links. The objective of vNode is to run multiple programmable infrastructures on a single physical infrastructure. vNode give a Click-like programming environment to the user in order to enable implementing new features and requirements easier and faster.

3.3.3 Resource Allocation

SDN software-centric data plane specially when running the forwarding mechanism together with some extra services (e.g., transcoding) needs to maintain a predictable performance. There are some proposals for resource allocation techniques in the literature that can be utilized to fulfill this need. For example, a prediction method is proposed in [121] that predicts how the performance may degrade if some resource contention happens among applications. This sort of technologies are enablers to replace hardware-centric devices with software based services in the SDN data plane.

3.3.4 Security

While several security threats are introduced to OpenFlow itself in the literature [122], some data plane and control plane mechanisms can provide security for end-points and users. An example of the latter case can be FRESCO that proposes a framework to develop security mechanisms on top of control plane. While FRESCO opens up a lot of opportunities to develop new security applications, since it is deployed at control plane, inherently it is bounded by SDN coverage limitation. Consequently, data plane is more interesting plane to host security applications than control plane even though in some cases it is the only choice that looks feasible (e.g., flow encryption). There are some security mechanisms in the literature that can be exploited in SDN data plane. For example, SSLShader [123] is a software SSL accelerator using GPU massive parallelism capabilities to realize a fast security service for software data planes that can processes up to 13 G/s of traffic.

3.3.5 Network Measurement

A first step to manage an SDN instance is to measure and then based on the data gathered in the measurement we can tune the network in different ways. The measurement usually happens in data plane. OpenFlow uses ASIC-based counters to measure a few aspects of the traffic such as size of the flow and the number of packets. There are a couple of works that foster ASIC-free counters and propose commodity CPU based counters as an alternative which are more flexible (e.g., [49]) or suggest adding a different commodity hardware to measure the traffic (e.g., [7]).

Such technologies can be used in a software-centric SDN data plane for measurement tasks.

3.3.6 Statefull Packet Processing

Current SDN data plane does not allow statefull processing of packets within the switch box. OpenState [124] argues the inability of OpenFlow to enable a statefull programming environment within the data plane and proposes a viable abstraction based on extended finite state machines as a generalized OpenFlow match/action abstraction. In the similar way, proposals such as user-defined switch actions [125] based on commodity hardware helps developing in-network services. As an example, in [126] authors propose a statefull software control that injects related advertisement based on the session content. Moreover, there are some proposals that investigate how we can deploy multiple services on a single box in an efficient manner. Thus, some frameworks such as NetOpen [127] are proposed to help developing customized in-network services (also [128] and [129]).

3.3.7 Wireless Networking

Besides wired networking, SDN data plane programmability can help wireless networking data plane. WiVi [130] is a Wi-Fi network virtualization infrastructure that not only enables multiple coexisting access points to work concurrently, but also enables data plane programmability for potential application developers. One example application over WiVi is an advertisement targeting mechanism that inserts custom ads to arbitrary flows [126]. OpenRadio [131] is a programmable wireless data plane that separates provides modular and declarative programming interfaces across the entire wireless stack. OpenRadio [131] supports a programmable wireless data plane and decouples processing and decision plane in the AP. The decoupling hides the underlying complexity of execution from the programmer of the AP. Odin [132] is another interesting a programmable AP for WLAN to foster making new service applications. It uses OpenFlow as the control plane to pass incoming packets to appropriate services on the AP. It is an example that fuse the potential of the traditional SDN in control plane and the data plane flexibility as an added value.

3.3.8 Misc.

There some other projects such as eXtensible Open Router Platform (XORP) [133], XIA [105] and MobilityFirst Future Internet Architecture Project [134] that provide a comprehensive data and control plane solutions as a mean to enhance extending and adapting the current solution to accommodate new, as yet unforeseen, features. Such ideas can help the current SDN data plane that is changing very quickly in time due to new requirements and developments to be able to adapt them easier and cheaper.

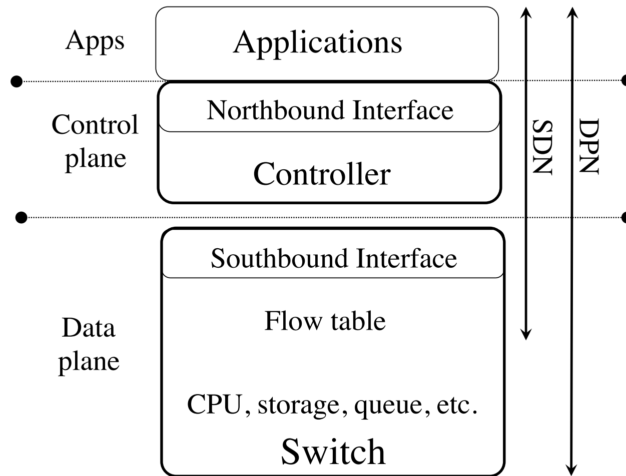

Figure 3.1: SDN vs. DPN.

3.4 Extending SDN to Deeply Programmable Networks

Fig. 3.1 shows the *programmability* characteristics of planes. At the top, we have a layer that corresponds to SDN applications. They communicate with the control plane via the northbound interface and thus they are called northbound applications. Next, in the middle, we have the control plane that manages data plane at the bottom via the southbound interface on the data plane. Even though we have many OpenFlow-enabled switches as the data plane of SDN, Fig. 3.1 shows partial coverage of SDN in the data plane programmability. That is because, ONF-defined SDN has a little attention to data plane programmability yet. Therefore, Fig. 3.1 indicates that SDN does not propose any solution for data plane programmability while it

does propose a solution for data plane functionality. In conclusion, there is a gap in data plane programmability which we believe should be fulfilled. So, we may be able to define SDN as a network architecture that decomposes control and data plane to provide and penetrate programmability, deeply and comprehensively into the networking stack by extending ONF definition to data plane. We call the total solution that covers programmability of all three parts as *Deeply Programmable Network (DPN)*.

3.5 Summary

In this chapter we discuss one possible and promising extension to the current SDN (i.e., DPN) by discussing the potential benefits of data plane programmability and reviewing existing proposals that can be used to realize data plane programmability.

Chapter 4

Tag-Based Flow Classification in DPN

4.1 Introduction

In recent years, the Internet has had to cope with changes in our communication patterns and usage practices. The growing popularity of smartphones, the interest in small wearable gadgets such as smart-glasses and smart-watches, the existence of billions of sensors, and the move towards cloud data centers and cloud computing have put unprecedented strains on the network architecture. In Japan, communication failures resulting from rapid increases in smartphone traffic have even led the ministry concerned to issue administrative guidance to major mobile carriers [135]. Also, a plethora of cybersecurity problems are reported every day, such as denial of service, spamming, phishing, spoofing, network security breaches, and invasion of privacy often caused by botnets (a large number of networked computers and smartphones compromised and controlled by adversaries). Meanwhile, cloud-based distribution of large content is increasing network traffic. The potential benefits of the Internet (which has been described as the infrastructure for anyone to transmit and receive any data freely anywhere to anyone) may be lost because of changing use and growing misuse. This concern highlights the fragile nature of a fixed and inflexible infrastructure.

Software-Defined Networking (SDN) defines publicly available open interfaces between the control plane and the data plane. It enables software programs to monitor and manage resources, operate and manage networks, control access, and so on. The primary benefit of SDN is to reduce operational expenses (by automating operations, administration and management) and capital expenses (by bringing openness to network equipment). Work in the area of SDN is currently attempting to enable the programmability of network applications and control plane elements, but there seems to be little interest in the programmability of data plane elements, thus limiting programmability and the use of computing within the network. Enabling deep programmability lifts the limitations imposed by current SDN practices, and makes it possible to realize a deeply programmable network that fully supports the whole range of programmability mentioned above. We believe that SDN should be extended to support the programmability of data plane elements, so that new data plane functionalities can be plugged in and unplugged flexibly. Extending SDN to enable simple programmability for data plane functionalities and to support the capability of defining or redefining interfaces for data plane functionalities, along with publishing those interfaces to control plane elements and network applications, also further reduces operational and capital expenses, because we can add or remove or modify data plane functionality by simple programming. Thus, we can reduce the complexity of maintenance, and decrease the life-cycle costs often observed in hardware-based inflexible data plane elements. Simple modification to interfaces to access new data plane functionalities enhances the capabilities and improves the efficiency of network applications and control plane elements.

While there are many efforts towards developing control plane architecture and applications such as OpenFlow [136], ForCES [3] and rule based forwarding [137], there is a limited attention to design effective APIs and methods for data plane. OpenFlow is a wide-spread API for SDN. This API installs a set of <*match , action*> rules on network switches. Any flow matching to the rule, receives the corresponding action. The matching process classifies particular flows for the action. Similar to many other protocols (e.g., SNMP [138]) to classify incoming flows, OpenFlow parses the packet against predefined protocol headers (e.g., Ethernet, IP and TCP). Then, it matches extracted fields against a set of rules. If enough fields match to a rule, it applies the corresponding action (e.g., drop or forward to port X) to the packet. One of the problems in this process is that the protocol headers should be *predefined*. Updating such patterns in

a timely fashion as well as keeping the performance at a reasonable level is not a trivial task. Because of performance considerations, matching process is implemented on hardware (e.g., Ternary Content Addressable Memory).

4.1.1 Flexibility of Flow Classification

An obvious solution is to support more protocols via adding more fields. Probably this is the solution OpenFlow is following. It is also possible to add some general fields (i.e., smarter wildcard rules) so that users can freely add arbitrary bits to the packet bit stream. The process of adding fields may (practically) converge after a decade. Before convergence, hardware upgrades may cause some overhead costs for users. That is, the same problem current hardware-centric networks suffer from. If we neglect all these issues, flexibility of flow classification can be a major shortcoming. Low level classification based on some static or wildcard matching rules suffer from inflexibility. Furthermore, traditional header matching is already overloaded in different ways. For example, port 80 is now used for many applications that can have different processing/forwarding requirements. We can not classify these applications easily using field-matching classification of OpenFlow. This is the main drawback of field-matching classification. Hence, any extra support for emerging protocols is costly.

4.1.2 Performance of Flow Classification

Eventhough OpenFlow is advertising the idea of SDN, the classification part is now implemented (from several vendors e.g., NEC and HP) on Ternary Content Addressable Memory (TCAM) which is a hardware. That is because researches suggest offloading classification and matching part of OpenFlow from software to hardware results at least in a 40% throughput gain [139]. It is the case when using commodity hardware. Obviously, using purpose-built hardware (e.g., ASIC) we can gain a much faster classification. Increasing requests to support new protocols/fields pushed 15 fields defined in OpenFlow 1.1 to about 40 fields in version 1.3 [136]. Accordingly, if a network owner buy a switch supporting OpenFlow 1.3, after next protocol specification upgrade, the hardware needs to be replaced resulting in a high CAPital EXpenditure (CAPEX). This is the problem almost any emerging technology faces while growing; to support already existing

technologies as well as new ones.

Aforementioned problems are of the type “flexibility versus performance tradeoff”. In order to have the TCAM performance and at the same time open up TCAM programmability limitations, community suggests *Network Processors* [140]. Network processor Units (NPUs) are a family of ICs using system-on-a-chip technology. They provide more efficient communication specific functions than general purpose CPUs. Network processors usually include a set of APIs to program the chip in high level languages. Therefore, we may gain more flexibility than TCAM based solutions. However, while using NPUs are an step forward to use commodity hardware, they are still inefficient for core switching heavy loads. In conclusion, still we need to lighten the flow classification overhead in addition to adding faster hardware.

4.2 Related Work

We can divide current state of the art to two main classes of methods based on the way they filter the target traffic for packet forwarding. Different approaches use variant solutions to initiate, store and maintain flow information. In the following we mention two classes of methods that follow different goals but share their core functionality:

Table 4.1: Summary of Tagging Approaches

	Main Objective	Tagging	Limitation	Classification Programmability	Flow Classification	Action based Forwarding	Software Defined?
MPLS	L2 Switching, Traffic Engineering	by Network	Purpose-specific HW	No	Balanced tag classification	No	No
OpenTag	Network Slicing	by User	Not transparent from user	by Sliver	Balanced tag classification	No	Yes
LIPSIN	Multicast Forwarding	by Network	Publish/subscribe Networks	No	Balanced tag classification	No	N/A
SourceFlow	Source Routing	by User	Not transparent from user	N/A	N/A	Yes (OpenFlow actions)	Yes
Scissor[141]	DC Power consumption reduction	by Network	Modified OpenFlow HW	No	Unbalanced tag classification	Yes (OpenFlow actions)	Yes
FlowTags[58]	Middlebox Policy Enforcement	by Network	Needs new protocols	No	N/A	N/A	Yes
Comp.TCAM[?]	Power consumption reduction	by Network	New TCAM design	No	Unbalanced tag classification	Yes (OpenFlow actions)	Yes
TagFlow	L7 Processing	by Network	Commodity HW	Yes (L7)	Unbalanced tag classification	Yes	Yes

4.2.1 Hashing Approach

Most of todays measurement and forwarding solutions apply a <*match,action*> rule to every packet passing through the system. The *match* section means a pattern or attribute that the packet includes. After classifying and matching packets with specific filters, an *action* is applied to the packet. Example of the action can be *count* for a measurement solution which means the counter for that specific matching criterion should be incremented based on the counter type. A counter could be a packet counter, a flow size counter or a timer etc.. In the hashing approach, basically the packet is parsed to find some special fields targeted by the *match* rule. In many cases such as OpenFlow, NetFlow [142] and sFlow [143], we may have 10 to 15 distinct fields to be parsed and involved in the classification. These fields can be located anywhere in the packet. The main bottleneck of many-field parsing is the number of memory accesses which grows as the load increases. Finally, a sort of *hashing function* is applied to the parsed fields to generate a *key*. The key is used to search a list of records containing flow information. In case of packet forwarding solutions (e.g., OpenFlow) the table is the forwarding table and usually implemented using a *hashtable* data structure with lookup complexity of $O(1)$.

PSAMP [144], IPFIX [145], NetFlow, sFlow and SNMP are a group of protocols that can be used for network management and measurement. They share in a couple of characteristics. They are dedicated to IP networks and if they are applied to a network measurement scenario, every node is responsible for both classify and measurement of traffic. None of these protocol support triggers to launch an event in case of a flow match except the SNMP which gets advantage of some triggering feature. The way they meter is *sampling* in time to reduce the overhead caused by the continuos sensing.

OpenFlow, DevoFlow [146] and Hedera [147] are a group of SDN APIs primarily designed to control the forwarding logic of network efficiently. They place as a logically centralized control plane in the network and install <*match,action*> rules on forwarding devices to manage the network. They are all designed to work on commodity hardware and they have some measurement features such as raw packet counters or approximated counters. DevoFlow supports limited data plane programmability and all of them support control plane programmability.

4.2.2 Tagging Approach

This approach received a limited attention from the community in comparison to hashing method. The major work in this area is the series of work starting from the 1990s by the Ipsilon Flow Management Protocol (IFMP), which is a protocol for allowing a node to instruct an adjacent node to attach a layer-2 label to a specified IP flow [148]. The work followed by tag switching and label switching proposals from Cisco and finally leaded to MPLS [149]. MPLS assigns a label to every packet and use it to forward packets in contrast to IP switching in which a sophisticated prefix matching decides the destination port of each packet. While the main objective of MPLS is to implement the theory of differentiated services as well as better traffic engineering (and it provides a level of abstraction over Ethernet layer), it is still dependent on a fixed layer-2 protocol. MPLS uses the edge network device to assign a label to the packet.

Our work is different from MPLS design and objective from several perspectives. First, MPLS provides a cut-through forwarding solution, however we propose a store and forward switching mechanism. Second, MPLS provides a hardware-based solution as it is not feasible to implement cut-through forwarding on software. Rather, our proposal is fully based on software. Third, as the MPLS solution is implemented on the hardware, any updates in the MPLS working mechanism and also any modification to the supported protocols leads to switch replacement which can be very costly. In contrast, we propose a software-based solution in order to enable timely, low cost upgrade and support for new architectures and protocols. MPLS proposes hardware-based solution to guarantee the switching performance. However, in our experiments we show it is feasible to use general purpose commodity hardware solution to achieve reasonable performances. Forth, MPLS embeds its tag right in the middle of the packet which causes low backward compatibility and can confuse switches which do not support MPLS protocol while in our proposal we use trailer tags at the end of the packet. That guarantees any switch on the way, which is not aware of our proposed system packet format, can forward packets without any problem since the packet headers are all untouched and understandable using traditional networking protocols. Fifth, because of header tagging in MPLS, in order to deploy MPLS on a network, every single switch needs to be replaced which can be a considerable cost barrier. In contrast, in our solution, any traditional switch can stay in the network as long as the

network operation does not require tag-based forwarding on those specific switches. Sixth, as our proposal is a software-based solution it enables two capabilities which are both absent in MPLS networks: a) a packet classification mechanism at the edge of the network that can be programmed without hardware replacement. In our experiments we show multiple examples of such classifications (see Section 4.4); and b) our solution provides user-defined packet processing mechanism within the switch which is explained in Chapter 5.

In addition to old contributions, recently some new requirements motivate researchers to revisit the same idea. LIPSIN [150] is a forwarding platform using source routing method for publish/subscribe networks. The whole path a packet should pass through (i.e., virtual link), is encoded using some hash functions in a fixed-sized label within the packet. Every virtual link that represents a complete source to destination path, has an identifier. The label carries the virtual link identifier and every forwarding node check the label against a forwarding table to find out which physical port is the right output port.

IETF is now following discussions on Network Service Header (NSH) [151] to create network service paths. That is a fixed size, transport independent per-packet service metadata (i.e. tag) added to the packet. Current network service deployment models are usually tied to underlying topology and are not adapted to elastic model of cloud environments powered by network virtualization. Virtual environments require more agile service insertion as well as flexible network service deployment models. Tagging and metadata on the packet gives the ability to the network to assign service policy to granular information such as per flow state.

Beside network oriented tagging where the network edge is responsible for putting the tag on the packet there are several works having different approach. User oriented tagging which implicitly proposes source routing, lets the user put the tag on the packet and in this way controls the route of packet to the destination.

OpenTag [152] is a network slicing mechanism for network virtualization that supports both performance and security isolation. In OpenTag, the user injects a slice ID tag per packet that denotes which slice the packet belongs to. In addition, a redirector is installed on conventional routers so that they can parse the tag and classify them for later slice-specific packet processing. OpenTag inherits some characteristics from source routing solution since user has some sort of control over the network behavior regarding his packet. SourceFlow [153] is another source

routing mechanism mainly designed to reduce the number of flow entries in TCAM using tags. The idea is to put forwarding actions (e.g. forward to port X) within the packets instead of classifying packet and then retrieving the corresponding action in the switch. When a server sends a packet to the network it calculates the route and then inserts the series of actions to be executed by switch nodes in the network to transmit the packet to the destination. Once the packet reaches the network edge device, it encodes the series of actions to allocate less space in the packet.

However, source routing suffers from security problems as it hands most of control decision making off to the user. This function can be used in attempts to route traffic around security controls in the network and security guidelines suggest to disable such a mechanism [154]. Moreover, source routing needs some modifications on the user side to enable calculation and injection of the tag which is a disadvantage compared to network oriented tagging where the system is totally transparent from OSs and applications. Table 4.1 summarize the comparison of tagging solutions from various aspects. As SourceFlow does not perform any kind of classification we put *N/A* mark in the related cells. By the *Flow Classification* column we refer to the way the network devices look into packets to determine which flow the packet belongs to. We emphasis on the similarity of the classification at edge versus core. *Balanced tag classification* mean edge devices and core devices perform exactly the same operation which is classifying packets based on tags. Similarly, *Unbalanced tag classification* means different types of classification happens at the core and edge due to different characteristics they have. *L7 Processing* basically refers to any type of switching, routing, Deep Packet Inspection and caching and other application layer proceedings. We discuss about our solution in details in later sections.

4.3 System Architecture

In this section we discuss the TagFlow architecture in details. Figure 4.1 illustrates the top level TagFlow architecture. In the following first we discuss the system briefly and then we explain details.

A datacenter network usually includes two network edges: source edge as well as destination edge. Source edge refers to switches that connect application servers to the internal datacenter

network. The source edge switch is the first device in the network which receives the packet can be a Top of Rack (ToR) switch. Destination edge means where the packet leaves the network. The network edge devices usually have less loads in comparison with core devices. When a packet arrives at the source edge, the edge switch classifies the packet and puts a tag on it and forwards it to the next hop. While packet is transmitted to the destination edge the only field that indicates where to forward the packet is the tag. Therefore, all core switches use a tag based forwarding table that maps a tag to a destination port. The destination edge switch removes the tag and sends out the packet. Using this method of classification and forwarding, we have different classification loads among edge and core devices (e.g., application layer classification at the edge and 1-field matching at the core). Obviously, application layer classification can reduce to many-field matching classification (i.e., similar to OpenFlow). Moreover, to keep backward compatibility with current networking practices we use *trailer tagging* that adds the tag to the very end of the packet. That is, most conventional devices that are not aware of tags can be placed in the middle of the network without any change. Needless to say, a logically centralized controller can manage the tags and assign them to slices.

A demonstration of different loads compared to OpenFlow (denoted as O.F.) is shown in the left side of the figure. OpenFlow has the same load of classification (i.e. many-field matching) at core and edge. However, TagFlow offloads the majority of classification load to the source edge. Please note that the bars are not in the exact scale. The point is to indicate the difference between two approaches.

4.3.1 Source Edge

When an end-user accesses some content on a server, packets flow in the network from the source edge. The source edge switch classifies packets and puts a *Tag* on every packet. The classification could be any application layer classification that distinguishes among applications with different characteristics. Each class could own a tag and treated as a separate flow. When the network administrator wants to apply different policies to different flows s/he can assign different tags to related flows. Therefore, every Tag type represents a class of packets with some shared criteria. For example, using a specific tag, HTTP packets including a special

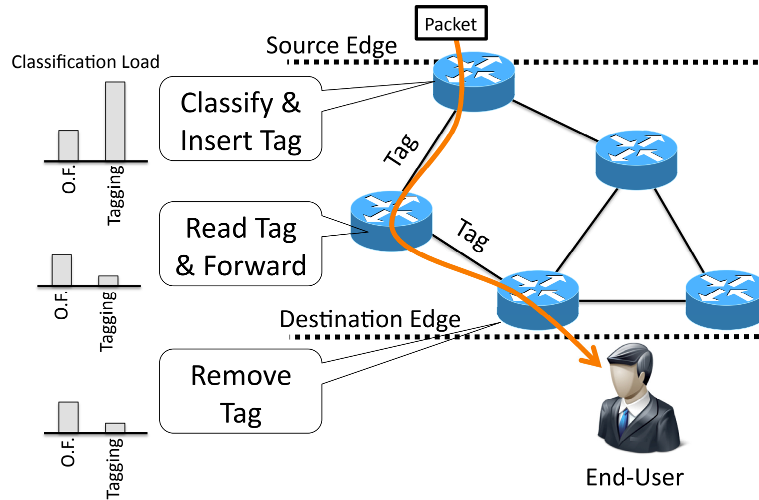

Figure 4.1: TagFlow System Architecture

domain name in the referrer section of the http header may receive different services compared to others(e.g. a special caching service or forwarding service).

4.3.2 From Network Core to Destination Edge

After inserting appropriate tags to related classes of packets, the regular network routing and switching algorithms transmits the packet to the destination edge. The switch at the destination edge removes the tag from the packet and lets it leave the network for destination. The destination edge devices may execute some actions before removing the tag. Our system is able to assign an action to each forwarding table entry. The forwarding table has two columns: flow identifier and action. We use tags as the flow identifier. Similar to OpenFlow, we apply the same $<match, action>$ to the packet. However, there are two differences; First, we match only the tag at core network so that we get much less overhead on core devices and considerably less overhead in a network-wide view. Second, *actions* are user-defined in our system. We explain actions in more details in the following section.

4.4 Evaluation

In this section we review our evaluation that shows the validity of our approach. Particularly, we study TagFlow using two approaches. First, we use single node measurements to show the difference of tagging and hashing approaches. As the complementary evaluation to single node experiments we show some end-to-end client server emasurements. In our experiments, we use a 32-bit tag which covers enough range of possibilities and at the same time does not put a considerable overhead on the system. However, we loosely define the tag size since it is highly dependent on the environment characteristics. For example, in environments with less diversity of flows we may define smaller tags.

4.4.1 TagFlow Control Plane

We implemented a centralized controller that is responsible for manipulating API parameters. Similar to OpenFlow API our solution can be named an API. However, there are some differences between OpenFlow API and our proposal; first, OpenFlow focuses on control plane while our TagFlow focuses on data plane functionality. Second, the objective of OpenFlow is packet forwarding while TagFlow objective is application layer processing. Our controller is a standalone program that connects to the switch using sockets. The user can write program and call the API. TagFlow controller currently supports only C++ programmability.

4.4.2 TagFlow Backward Compatibility

Most of current solutions either need a special hardware or special protocol to operate. Since we add a label to the packet and it may cause some compatibility issues, we should provide some mitigations to overcome the problem. For example, MPLS puts the label in the middle of the packet and causes incompatibility with traditional switches. To address such a concern, we put the tag at the end of the packet (i.e., *trailer tagging*). Since networking devices usually look at the beginning of the packet, they can adapt to trailer tagging without any change. Therefor, only tag switching devices that need to look at the tag can parse the trailer. In case of variable length packets which are common in the current networking practices, when we need to access the tag located at the end of the packet we do no need to parse whole the packet to access the

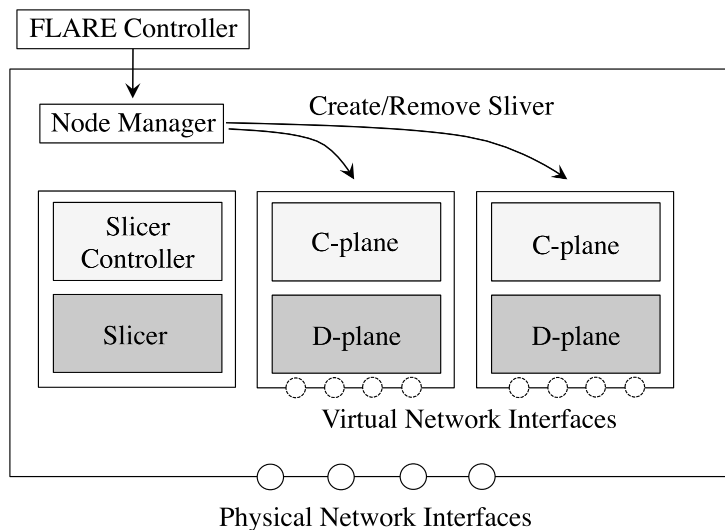

Figure 4.2: High-level illustration of FLARE switch

tag, rather, as we already have the packet length we just subtract the fixed length of the tag from the packet length and read the tag. In conclusion, while trailer tagging provides backward compatibility it does have negligible access overhead.

4.4.3 Single Node Experiments

We conducted two series of experiments to demonstrate effectiveness of TagFlow using FLARE switch as a fully programmable and high performance software switch. We use Xena packet generator to feed packets to the FLARE. For experiments that are done using a PC we use Click as the packet generator. The objective of all experiments is to measure the throughput of a single TagFlow switch against a single OpenFlow switch.

FLARE [108], is a node architecture that enables open deep programmability within a network. FLARE introduces an isolated programming environment called a *sliver* (i.e. a set of computation, storage and linked bandwidth resources). Each FLARE node has a control module called *node manager*, which dynamically installs or removes slivers, and a programmable classification engine called *packet slicer*, which quickly scans packets and multiplexes or demultiplexes them from or to slivers. Figure 4.2 illustrates FLARE architecture. A central control node called *FLARE central* remotely manages multiple FLARE nodes and creates or removes

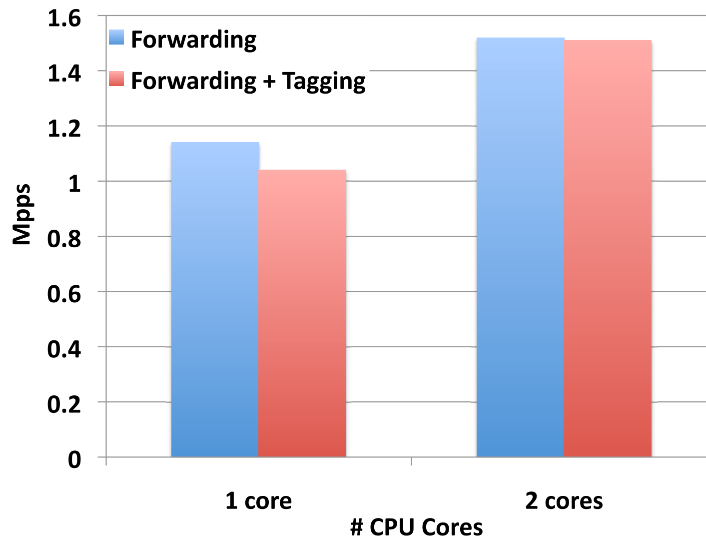

Figure 4.3: Tagging overhead (64-byte packets)

or assigns slivers for sliver programmers, on demand, communicating with each node manager to allow programmers to access their own sliver and inject their programs. FLARE supports deeply programmable SDN solutions. This feature makes it possible to apply traffic engineering to a specific device or application or piece of content. That is, FLARE provides control and packet processing according to application context. As an example of deep programmability, FLARE can define switching using layer-2 protocols. For example, FLARE makes it possible to extend MAC addresses from 48 bits to 128 bits, not only staving off MAC address exhaustion, but also supporting a large number of tenants in data centre networks while also maintaining transparency for Internet protocol applications.

As the OpenFlow installs the same forwarding logic engine on all switches, in order to measure forwarding performance we need to measure forwarding throughput of a single switch. Whereas, TagFlow uses two different logics at the edge and core. Thus, we perform two tests on TagFlow to show the feasibility of the method. First, as the edge measurement, we consider the same classification among OpenFlow and TagFlow. In the forwarding process at the edge TagFlow has a tagging overhead after classification to put the tag on the packet. So, we measured the tagging overhead to see how much extra work TagFlow should do at the edge compared to OpenFlow. In order to measure the overhead itself we compared vanilla forwarding

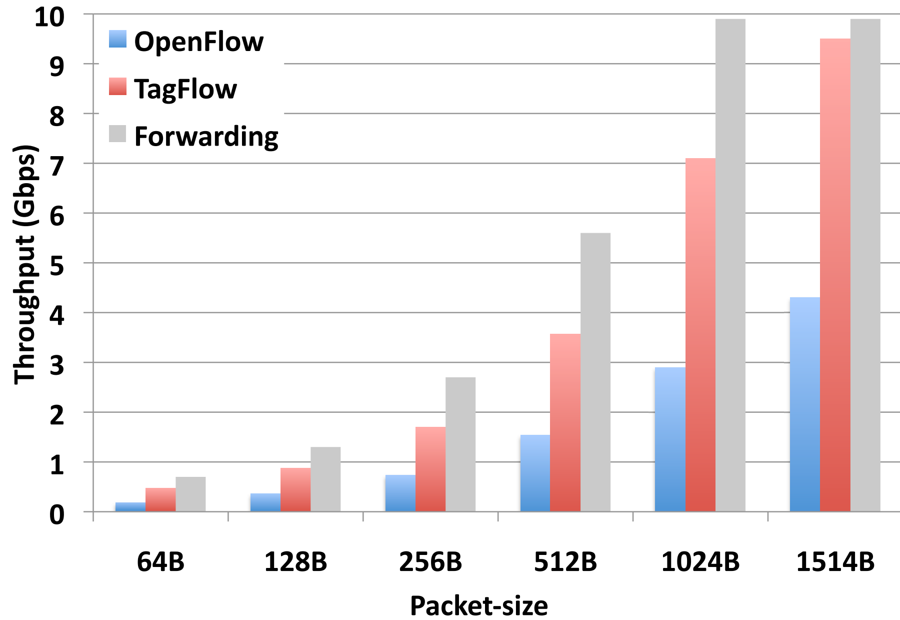

Figure 4.4: TagFlow versus OpenFlow versus vanilla forwarding using one CPU core

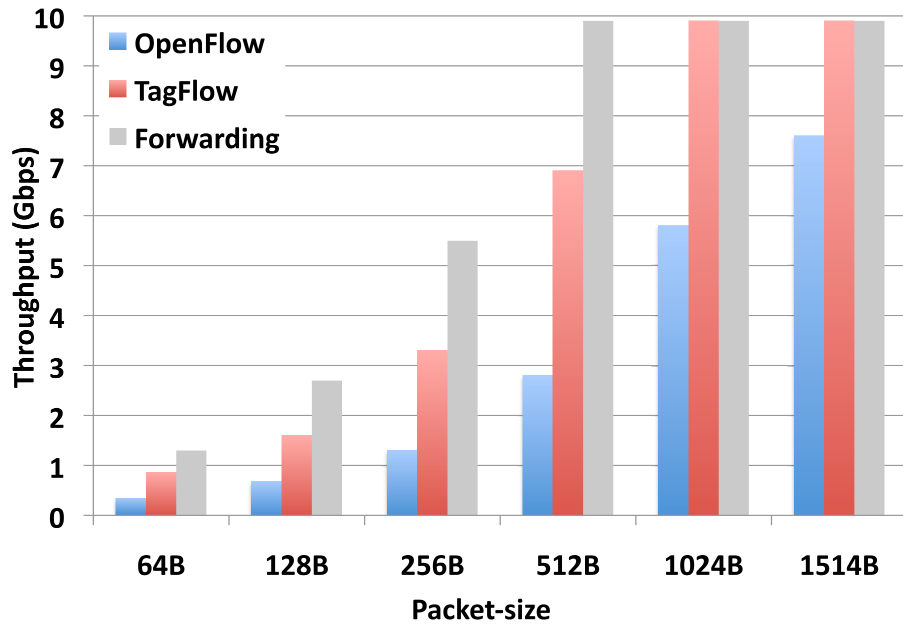

Figure 4.5: TagFlow versus OpenFlow versus vanilla forwarding using 2 CPU cores

with another forwarding logic in which we embedded the tagging mechanism. We use a 4-byte tag on 64-byte packets. Figure 4.3 indicates the result. We repeated experiment twice, once

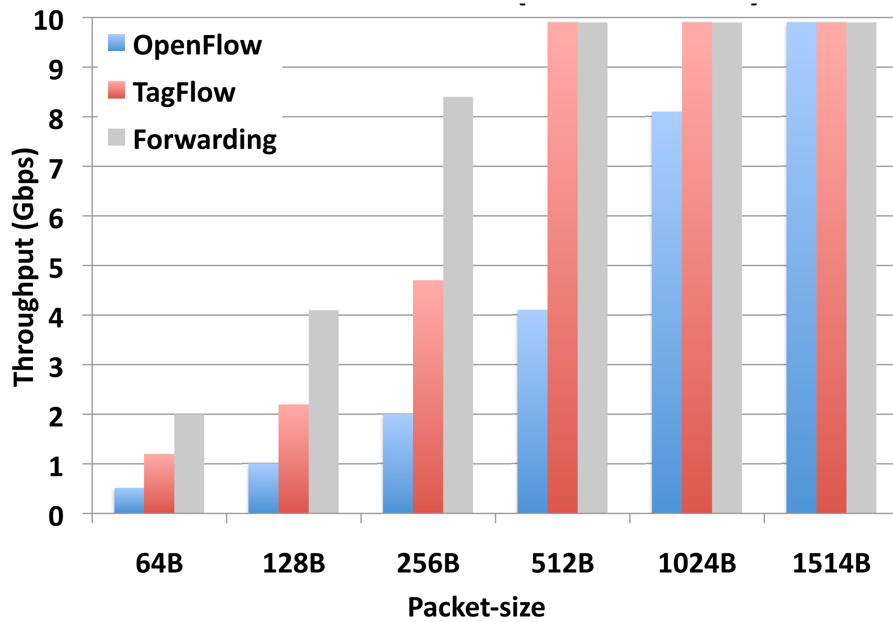

Figure 4.6: TagFlow versus OpenFlow versus vanilla forwarding using 3 CPU cores

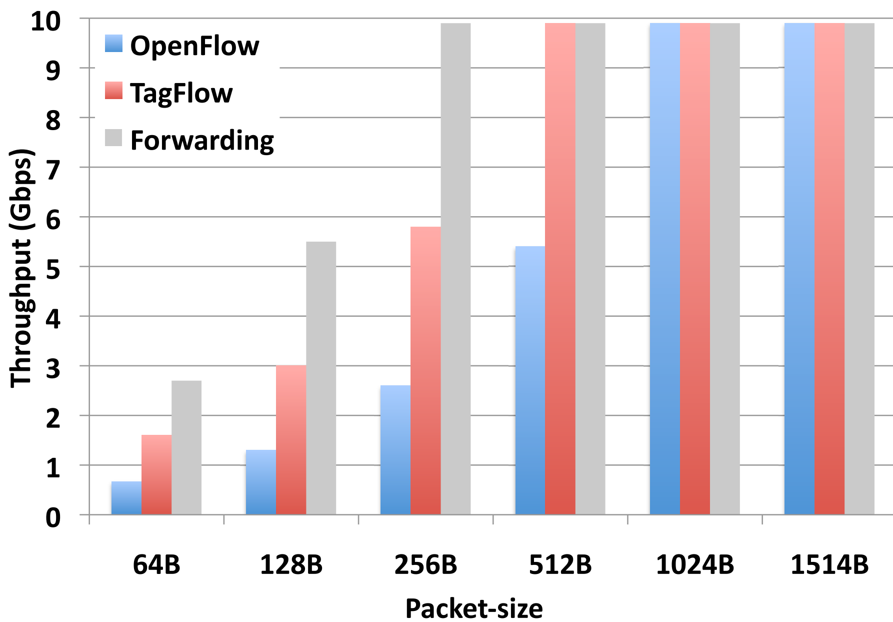

Figure 4.7: TagFlow versus OpenFlow versus vanilla forwarding using 4 CPU cores

using one CPU core and another time using two CPU cores. As the former case, tagging process shows some difference (i.e., 8% overhead) however in the latter case the performance is almost

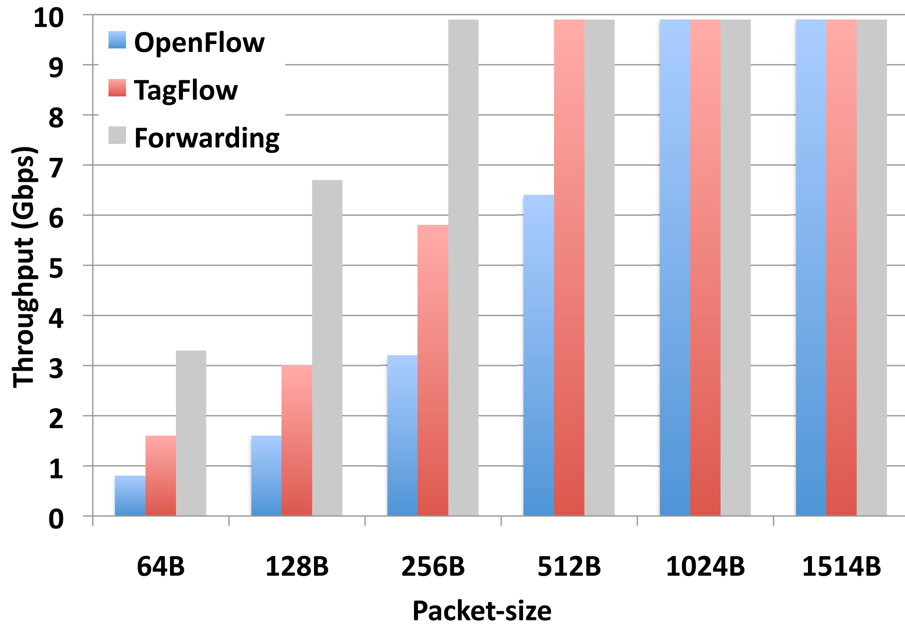

Figure 4.8: TagFlow versus OpenFlow versus vanilla forwarding using 5 CPU cores

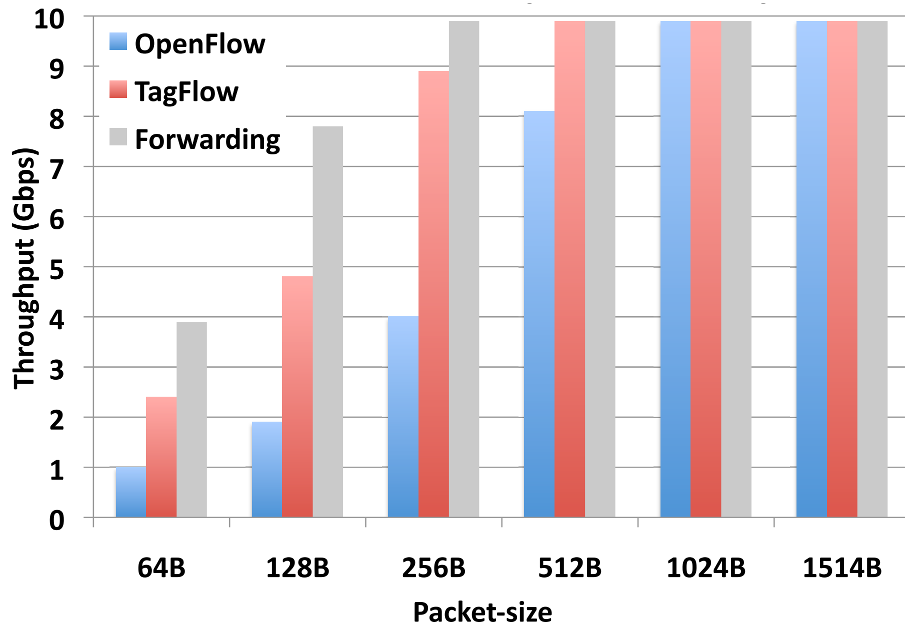

Figure 4.9: TagFlow versus OpenFlow versus vanilla forwarding using 6 CPU cores

the same (i.e., 0.6% overhead) while processing 1.51 Mpps of the size 64-byte. Since using more number of cores gives us the same level of difference, we excluded the results from the graph to

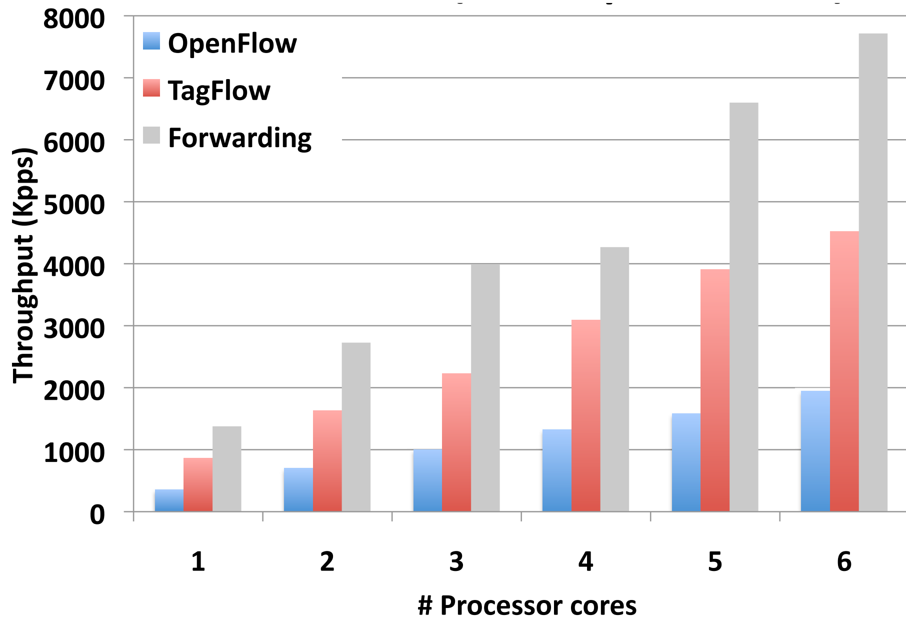

Figure 4.10: TagFlow versus OpenFlow versus vanilla forwarding using 64-byte packets

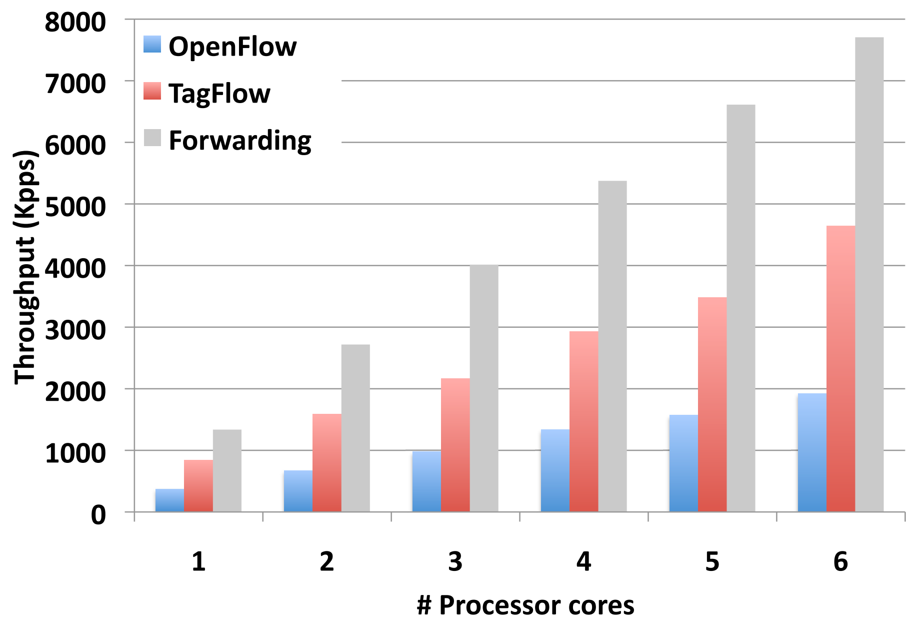

Figure 4.11: TagFlow versus OpenFlow versus vanilla forwarding using 128-byte packets

keep the simplicity and readability of the figure. In conclusion, tagging has a negligible overhead on edge devices. The process of removing the tag from packets also has similar overhead. Since

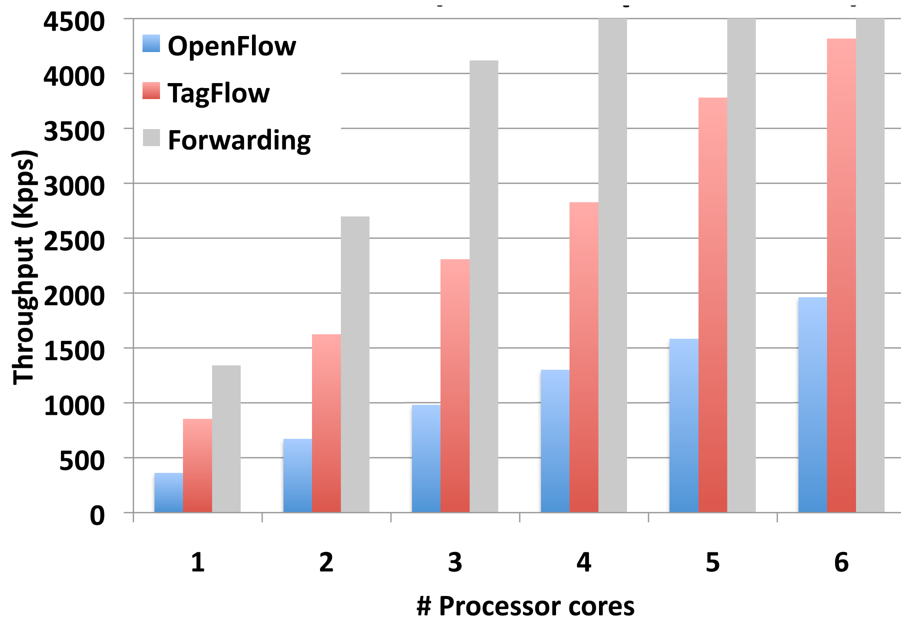

Figure 4.12: TagFlow versus OpenFlow versus vanilla forwarding using 256-byte packets

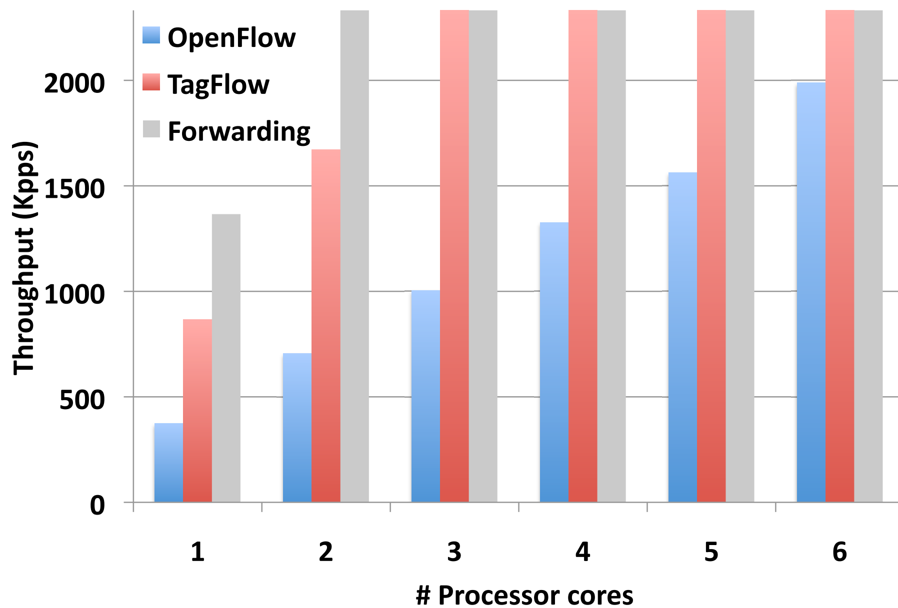

Figure 4.13: TagFlow versus OpenFlow versus vanilla forwarding using 512-byte packets

memory manipulation happens in injection of the tag is not very different from the memory manipulation and access of the elimination of the tag from packet.

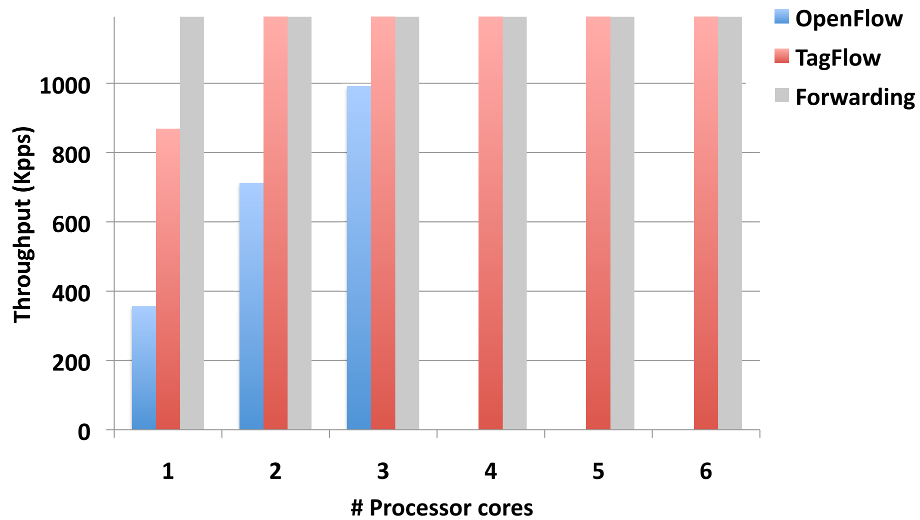

Figure 4.14: TagFlow versus OpenFlow versus vanilla forwarding using 1024-byte packets

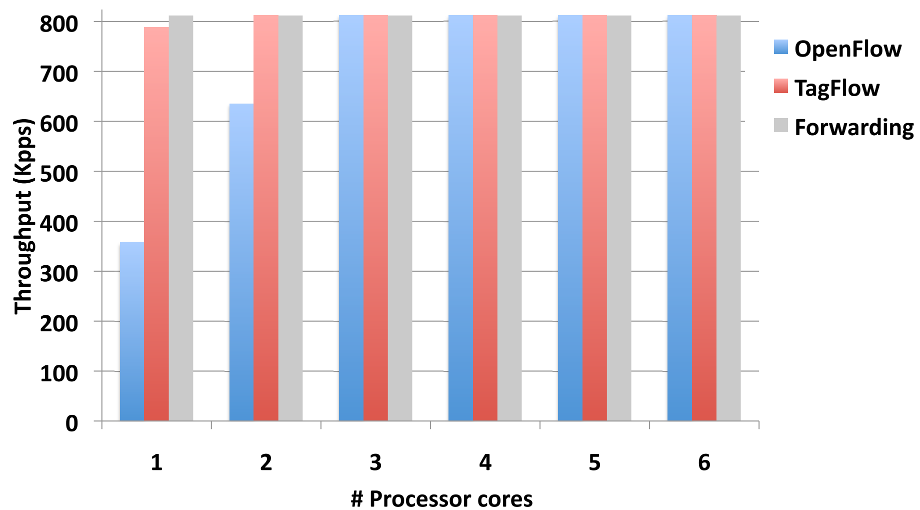

Figure 4.15: TagFlow versus OpenFlow versus vanilla forwarding using 1514-byte packets

Next, we conducted a comparison between OpenFlow and TagFlow core switches. In particular, we compared OpenFlow, TagFlow and simple forwarding using packets of different sizes. While using more than one CPU core of FLARE, vanilla forwarding and TagFlow give very close results we conducted the experiment using a single core. Figures 4.4 through 4.15 illustrate how TagFlow differs from OpenFlow and Figures 4.16 and 4.17 indicate TagFlow performance using different metrics. Current version of the FLARE switch has four 10G SFP+ ports. We

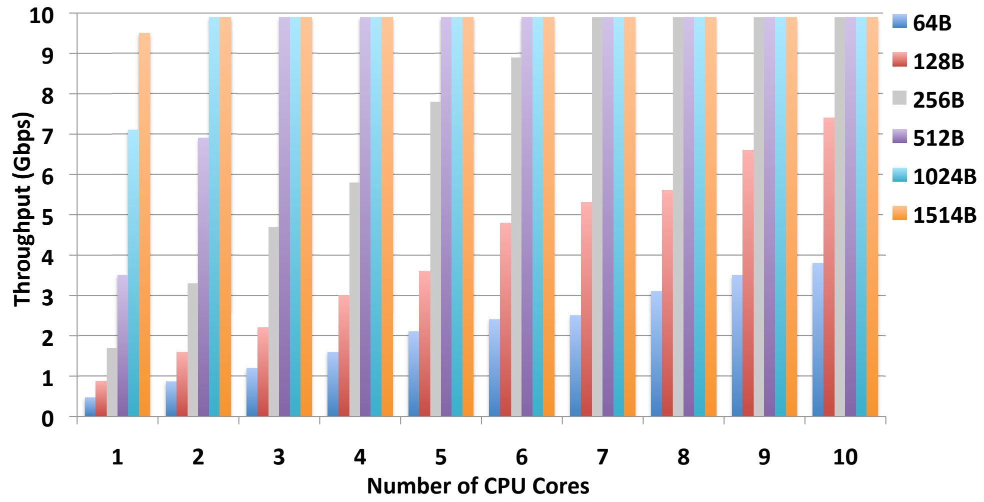

Figure 4.16: TagFlow throughput in Gbps using different number of CPU cores

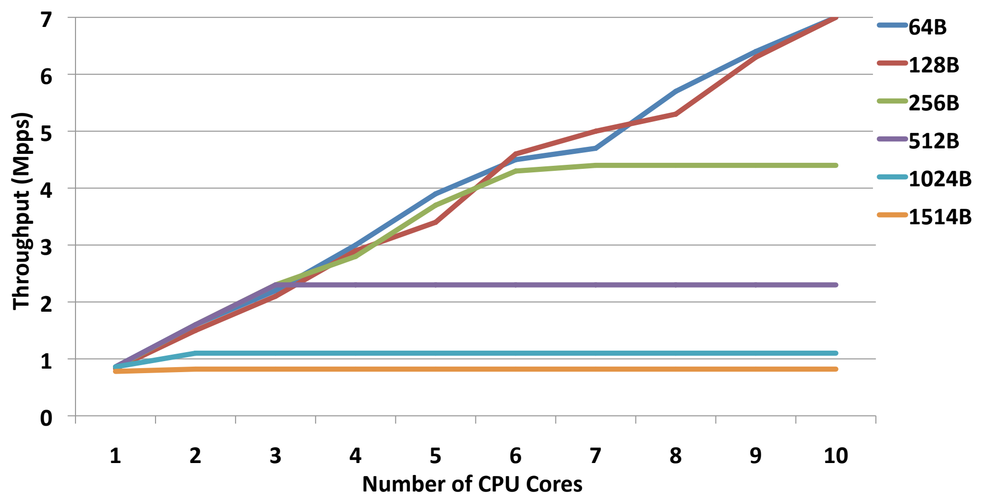

Figure 4.17: TagFlow throughput in Mpps using different packet sizes

connected FLARE to XENA packet generator and configured XENA to send packets to FLARE and measure the received traffic. In conclusion, TagFlow is less than 40% faster (in average) than OpenFlow considering different packet sizes.

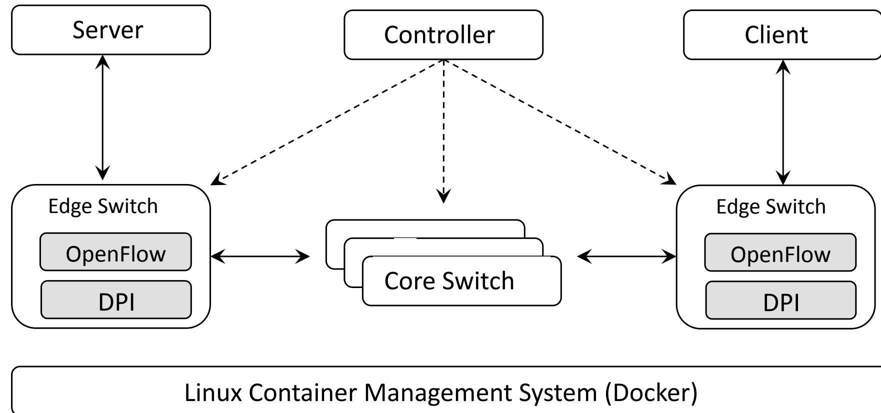

Figure 4.18: End-to-end experiment architecture

4.4.4 End-to-end Experients

In addition to single node experiments we conducted a series of end-to-end experiments. Figure 4.18 shows our experiment architecture. There is a client that sends packets to the server and receives the server response. We put different number of switches between client and server to see how it affects the packet roundtrip latency using TagFlow and OpenFlow. Implementation wise, the whole system is running on a Linux container management system.

Figure 4.19 indicates average packet roundtrip latency when we have 3,5 and 10 nodes in the route from the client to the server. For example, three nodes means a route in which there is one switch between client and server. So, the number of nodes includes the client and server nodes. This figure compares latencies of TagFlow and OpenFlow. The figure reflects the fact that the benefit of TagFlow increases as the number of hops in the route increases. That is, TagFlow works better in larger networks. The difference between OpenFlow and TagFlow shows the processing capacity that TagFlow releases compared to OpenFlow case. The freed capacity grows as the number of nodes in the network grows. We can use this capacity for useful applications.

Figure 4.19 shows the latency for a single packet while Figure 4.20 projects the same system over 1000 packets. So, the latency indicated in Figure 4.20 is cumulative roundtrip latency comparing TagFlow and OpenFlow cases. This figure projects the 10 hop route scenario. We can see a steady linear growth in the latency which guarantees more latency as the network

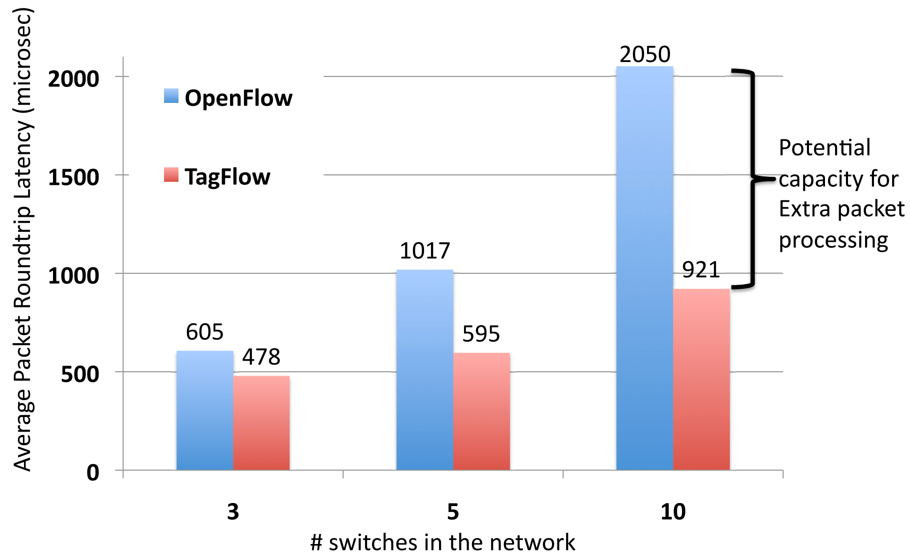

Figure 4.19: End-to-end TagFlow versus OpenFlow roundtrip latency for a single packet

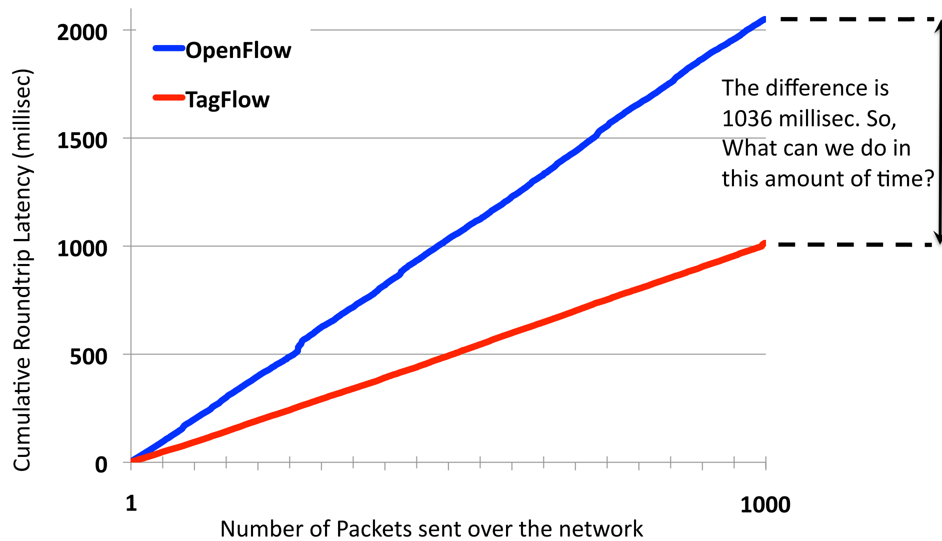

Figure 4.20: End-to-end TagFlow versus OpenFlow cumulative roundtrp latency

grows and we have routes with more number of hops. In our experiment, for 1000 packets in a 10 node end-to-end scenario we have a 1036 milliseconds of cumulative delay that can be used for useful applications.

Now the question is what do we mean by useful applications? What are the examples of

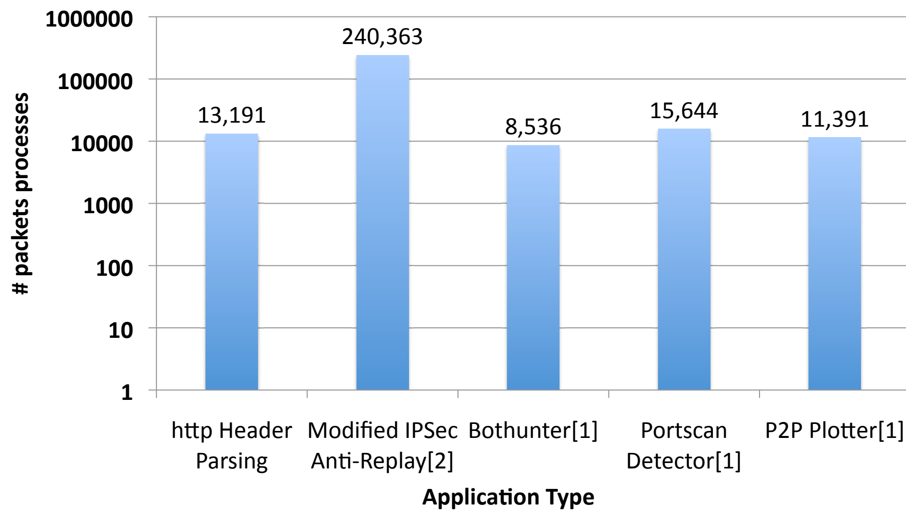

Figure 4.21: Number of packets processed in 1036 millisec using five different applications

useful applications? Figure 4.21 shows five examples of such useful applications. This figure explains what we can do in 1036 milliseconds (we got from the previous experiment) using five different applications. Bothunter, Portscan Detector and P2P Plotter are security applications. The Modified IPSec Anti-replay attack is a simple counter based replay attack detector which is similar to the one used in IPSec protocol. HTTP Header parser is a full http protocol header parser that detects different fields in the http header. The figure shows that based on the type of application we can process many packets using the freed capacity. So, till now we show TagFlow frees some capacity compared to OpenFlow and then we show we can use the freed capacity for some applications. As an example, here we review one of the applications in more details to reflect the potential value of such applications. We choose Bothunter as the example.

Bothunter is an application that clusters network nodes based on their port scanning behavior to detect bots via co-clustering of nodes producing network anomalies. Bot can be defined as a self-propagating application that infects vulnerable nodes in the network. Botnets is a group of bots managed in a systematic fashion using a central operation console. They are one of the important threats against computing assets. Bot propagation starts with port scanning of potential vulnerable nodes. So, bots are the primary source of most of the port scanning happens on the Internet. Bouthunter uses the fact that infected machines that are

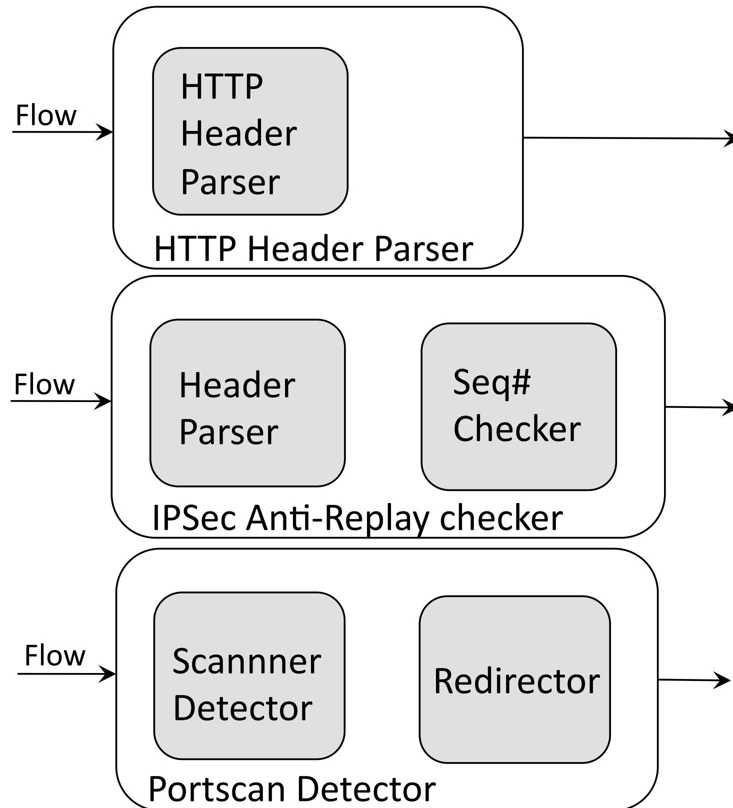

Figure 4.22: Bothunter architecture

trying to infect others usually have a low bit per second and packet per second traffic patterns. Bothunter clusters the machines with such criteria and blocks them from spreading over the network. Figure 4.22 shows the internal Bothunter architecture. The C-Plabe measures the communications of each node based on the bit per second and packet per second metrics. The A-Plabe detects if a node scans another node. The redirector component offers the blocking feature of portscanning detection which means we can block the nodes that scan others. This specific feature is not useful in Bothunter and it exists in the architecture since Bothunter reuses this component from another application in Figure 4.21 (i.e. port scan detector). Figure 4.23 and Figure 4.24 demonstrate the architecture of other applications.

There are some alternatives for the same objective as Bothunter. We review two major ones here. The objective of alternative mechanisms are also blocking bots from spreading all over a datacenter network. Figure 4.25 describes the first alternative. There is a high-end Intrusion

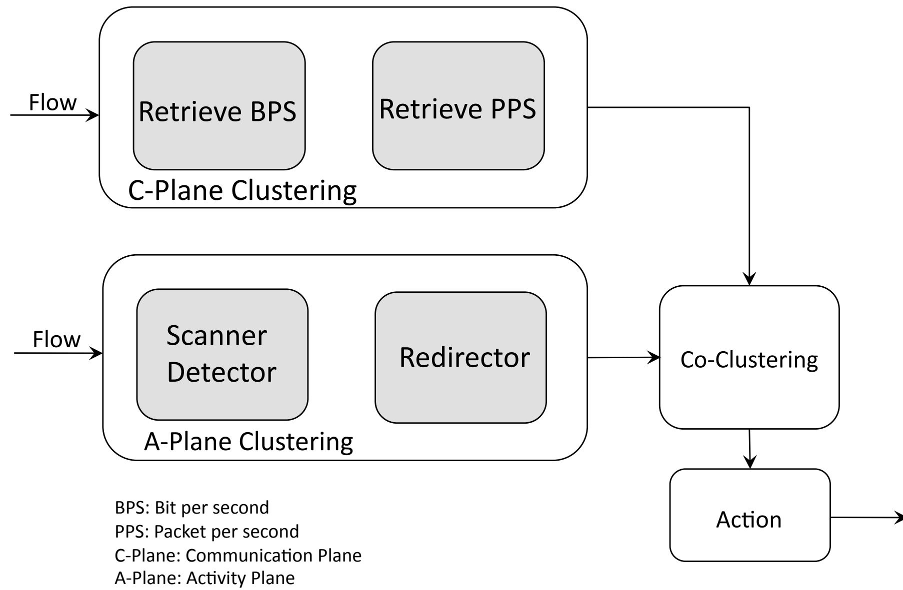

Figure 4.23: Architecture of HTTP header parser, modified IPSec Anti-replay checker and portscan detector

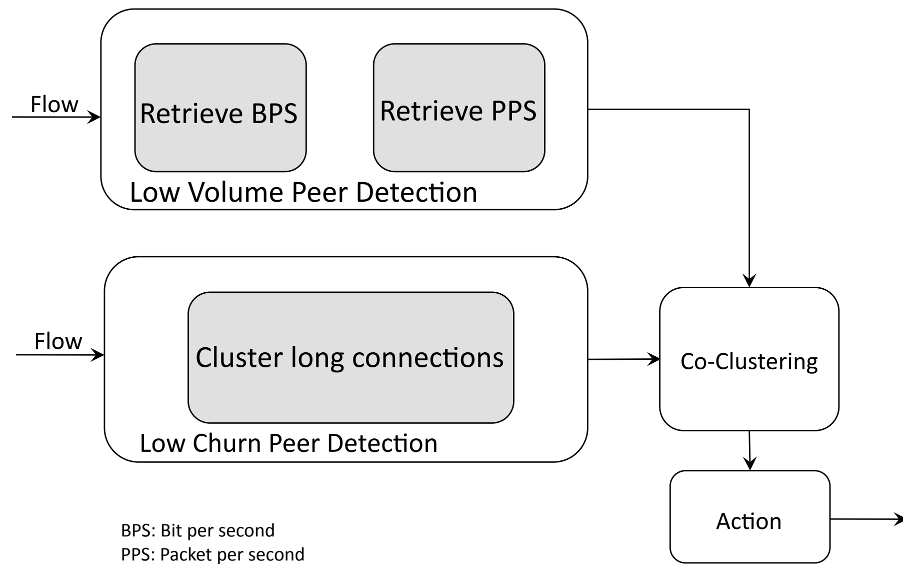

Figure 4.24: P2P Plotter architecture

Prevention System (IPS) that is connected to every switch in the network. All the traffic passing over a switch will be first sent to the IPS for processing and then comes back to the switch to be forwarded to the destination port. There are two shortcomings in this alternative. First, the

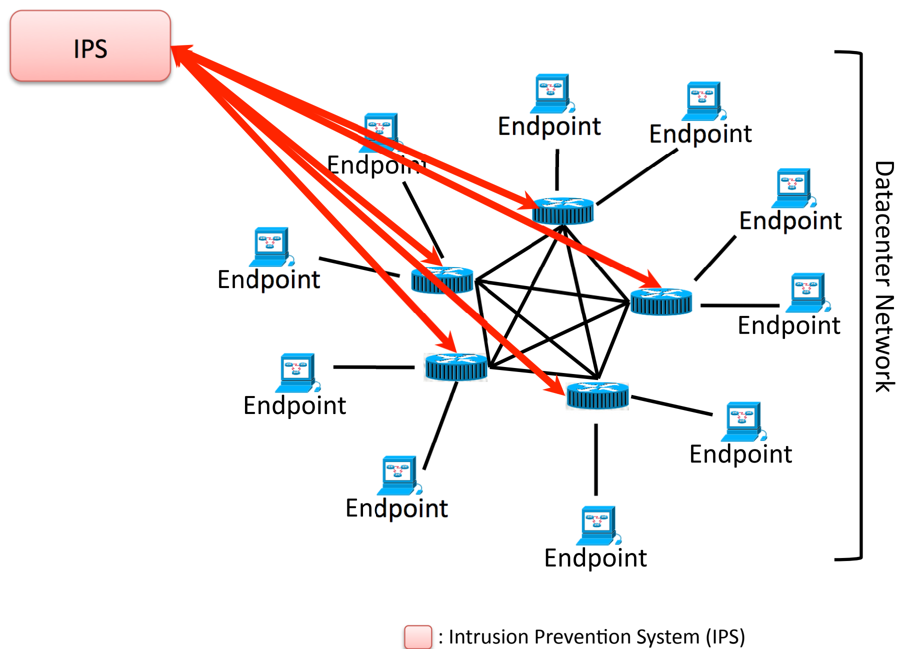

Figure 4.25: Bothunter alternative scenario 1: Single intrusion prevention system

price of a high-end IPS can go high. Second, sending every packet to an external IPS increases the packet transmission delay. As the IPS can be considered a single point of failure (SPOF), it may raise some security considerations. In case of an attack, the IPS may go out of response and this can cause a failure in the whole network.

A second alternative scenario to Bothunter can be using a low-end IPS on every switch to prevent the network delay caused by the first alternative scenario. Figure 4.26 show this scenario. There is an IPS located on every switch on the network edge. In case of servers in a datacenter the edge switches can be Top of Rack (ToR) switches plus the switches that connect the datacenter network to external networks. Because of the number of IPSs used in this scenario, the cost can easily go high while this scenario can potentially cause less delay than the first alternative.

We explained alternative scenarios for a basic bothunter application. Now, we review our approach to put bothunter in the network. We explain two possibilities of deploying bothunter in a network. First, Figure 4.27 shows deploying bothunter in an Intrusion Detection System style (i.e., IDS style). IDS is a system that detects malicious activities and reports them to

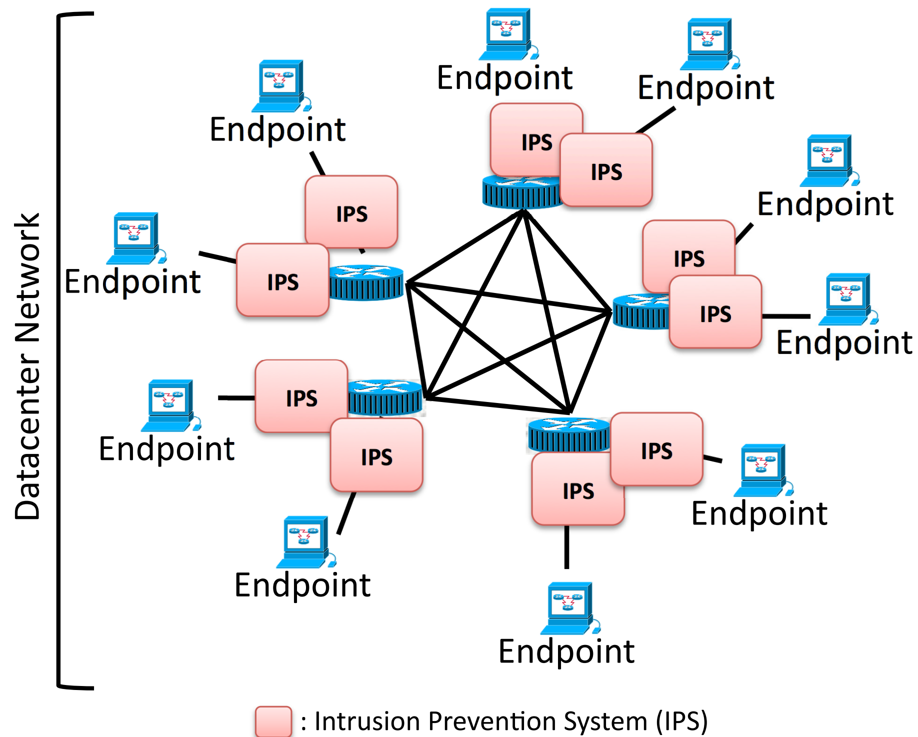

Figure 4.26: Bothunter alternative scenario 2: multiple intrusion prevention systems

the system administrator or the Security Operations Center (SOC). Boxes with numbers in the figure explain the following steps:

1. A bot infected by a virus attempts to scan other machines to find a new victim.

2. The scanning process generates a low bit per second and packet per second type of traffic. Once the scanner finds an open port it proceeds with exploitation and infection of the second computer. Now, the second computer starts scanning other computers' ports.

3. The scanning traffic of the second infected computer passes through the switch in a green box (i.e., the switch including the bothunter application installed on the switch itself). The bothunter instance on the switch detects the traffic pattern of the infected machine and fires an alarm.

4. The bothunter sends the alarm to the SOC.

Switches without the green box does not include bothunter and thus they are agnostic about

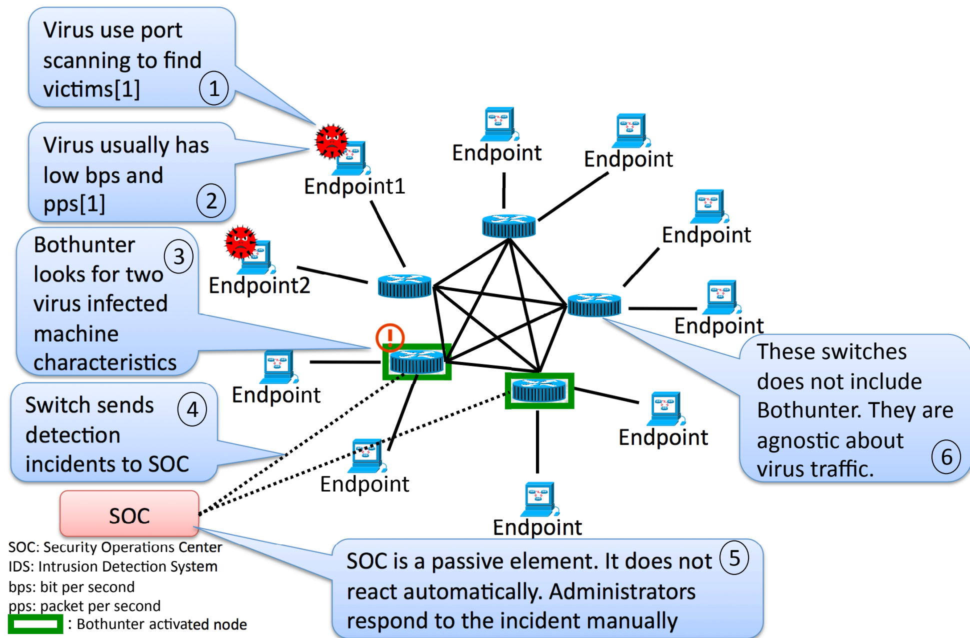

Figure 4.27: Bothunter IDS style

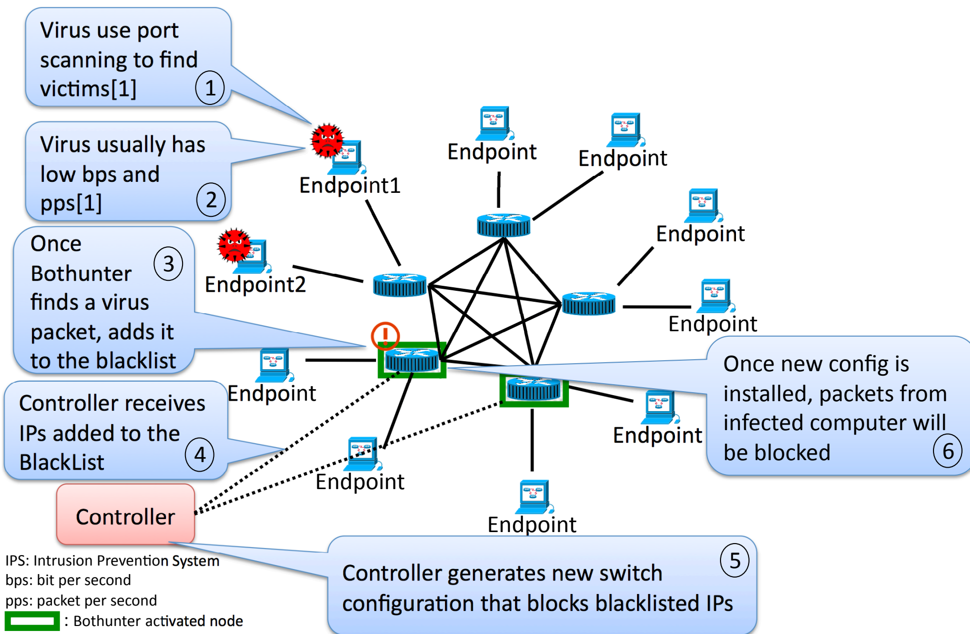

Figure 4.28: Bothunter IPS style

virus generated traffic.

Figure 4.28 shows a similar scenario to Bothunter IDS style, but instead of IDS it applies an IPS mechanism (i.e. IPS style). The difference between IDS style and IPS style is blocking the malicious traffic from the infected bot. In the IDS style scenario the bothunter only reports the detected malicious activity traffic to the SOC and administrators are responsible to react manually to the security incident. In contrast, the IPS style case blocks the traffic after firing the security incident alarm. The steps are as follows:

1. A bot machine infected by a virus is attempts to scan other machines to find a new victim.
2. The scanning process generates a low bit per second and packet per second type of traffic. Once the scanner finds an open port it proceeds with exploitation and infection of the second computer. Now, the second computer starts scanning other computers' ports.
3. The scanning traffic of the second infected computer passes through the switch in a green box (i.e., the switch including the bothunter application installed on the switch itself). The bothunter instance on the switch detects the traffic pattern of the infected machine and adds it to its own blacklist.
4. Controller receives IPs added to the blacklist.
5. Controller generates new switch configuration that blocks blacklisted IPs.
6. Controller installs the new configuration on the switch. Once it is installed, packets from infected computer will be blocked by the switch.

To compare the detection latency of IDS and IPS style scenarios we run port scan traffic through them and measured the detection latency for IDS style and detection plus reconfiguration latency for IPS style. Figure 4.29 shows Bothunter IDS style detection latency and Figure 4.30 shows the Bothunter IPS style detection and reconfiguration latency. The red line in Figure 4.30 shows the IDS style detection latency to indicate the difference. The main cause of the difference is the reconfiguration delay.

We implemented Bothunter as a simple API. Figure 4.31 shows methods and parameters for IDS style and IPS style solutions. While these APIs can be extended in many ways, here our objective is to show the potential to develop useful applications.

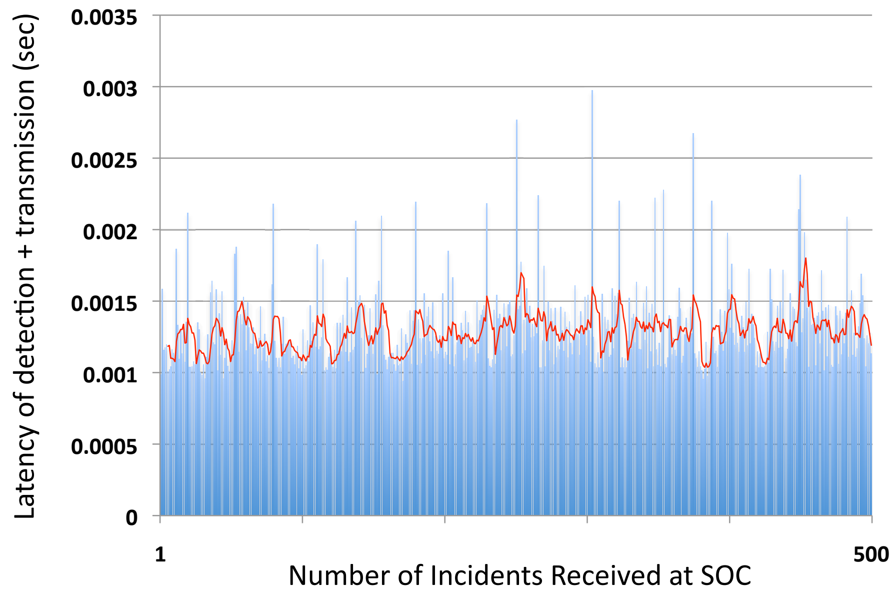

Figure 4.29: Bothunter IDS style detection latency

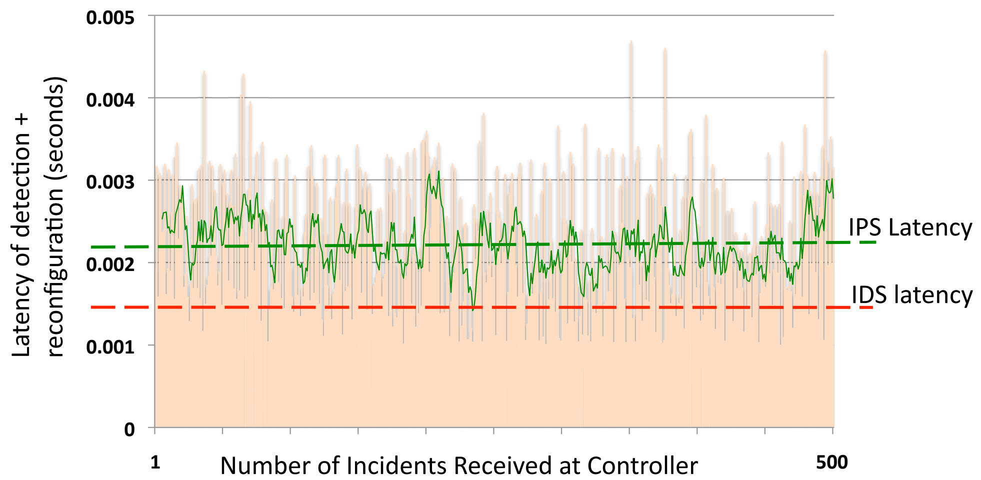

Figure 4.30: Bothunter IPS style detection and reconfiguration latency

4.4.5 Northbound versus Southbound Applications

In the OpenFlow context (and to some extent in the current SDN context) *northbound* application is implemented on top of controller and calls the controller. In contrast, *southbound* means the data plane layer consist of switches and forwarding plane. In our context, we may

IDS API

Methods	Parameters	Note
Readinc	Incident (output)	Reads the incidents from the switch once they are triggered.

IPS API

Methods	Parameters	Note
addrow	IP_address, Time (inputs)	Adds an entry to the black list
delrow	IP_address (input)	Deletes and entry from the black list
getlist	Incident_list (output)	Fetches all entries in the blacklist
config	new_config (input)	Installs the input configuration on the switch

Figure 4.31: Bothunter API methods and parameters

name our API a southbound API or southbound application as it is implemented on the data plane and provides data plane programmability compared to OpenFlow model that advertise control plane programmability. Todays, northbound applications are considered as hot topic. However, giving examples of useful applications in this area is not a trivial task as there are many limitations in the control plane. The major limitation on the control plane is the lack of access to the packet payload. That is why it is infeasible to have applications that filter more than a bitstream match (i.e. the OpenFlow classification model) on the packet. For example, an Intrusion Detection System needs to look at the whole packet and even buffer a couple of packets to detect an intrusion while sending packets to the controller for intrusion detection is infeasible because of performance considerations. To support our argument, we implemented two examples of northbound security applications from a related work [6] in the data plane (i.e. southbound) to see the difference.

Table 5.4 illustrates the difference between the overhead of running two applications on the data plane (i.e., D-plane or southbound) versus the control plane (i.e., C-plane or northbound). Port scanner detector looks for repeated attempts to connect to a closed port on a system from another system. This kind of detectors help to find worm-infected machines that are scanning

Table 4.2: Northbound versus Southbound (i.e., TagFlow) Applications

	C-plane Overhead (ms) **LoC (Python + config)[6]**	**D-plane Overhead (ms)** **LoC (C + config)**
Port Scanner Detector	7.196 / 205 + 24	0.000004 / 249 + 15
BotMiner Detector	15.421 / 312 + 40	0.00012 / 548 + 27

the network for new victims. BotMiner detector is another simple application that clusters network nodes based on the output of port scanner detector to detect bots via co-clustering of nodes producing network anomalies. The C-plane implementations show the lines of code (LoC) plus the overhead delay of running elements on FRESCO [6] framework. FRESCO is a Click like framework to develop elements for security applications at northbound. Similar to Click, FRESCO applications consist of two parts; an element code in Python and a configuration file. We implemented the same applications at data plane in C++ and compared the overhead as well as LoC. We used Click as the data plane implementation framework. Interestingly, the same application runs hundreds of times faster in data plane compared to control plane. The main reason is the delay caused by sending information from data plane to control plane in FRESCO which is the case in almost any northbound application. In short, lack of flexibility and data transmission overhead are two major limitations of northbound applications.

Such security detectors are in fact some sort of classifiers. In the context of our system we can put them at the edge as classifiers to separate flows with suspicious activity. Also, they can be considered as actions to be loaded by the switch and applied on some flows which (for example) need more security considerations. In the latter case, the user can program the controller to assign these actions in the action columns of switch forwarding table corresponding to some specific tags. All in all, tags not only can represent flows, but also can specify more information about the flow such as security level an so on.

4.5 Discussion

We believe that software-defined networking should be extended to support the programmability of data plane elements which is the main proposal of Deeply Programmable Networking, so that new data plane functionalities can be plugged in and unplugged flexibly. Deep pro-

grammability refers to the whole range of programmability, including network applications, control plane elements and data plane elements. Software-defined networking and network functions virtualization are the technologies included in this conceptually wide area of studies of programmable infrastructure.

The TagFlow idea is based on the Deeply Programmable Networking(DPN) concept as it touches the data plane. TagFlow modifies the packet classification in the network. Since packet classification happens on the data plane part of the switch, any modification to that part requires a programmable data plane. TagFlow is an example case to show the effectivity and value of the Deeply Programmable Networking idea. DPN suggests the current programmability coverage by traditional SDN, needs to be extended to fully cover the data plane and TagFlow solves a network classification problem using exactly the same extension. Expansion of programmability from coverage of control plane to the data plane lets TagFlow to instruct network switches to classify and forward packets based on a tag on the packet trailer rather than header. Providing the same functionality without using DPN is also possible. However, such an approach is equivalent to proposing a new hardware which is the same approach as traditional SDN proposal (i.e. OpenFlow). So, it suffers from the same limitations OpenFlow has. One of the main limitations of OpenFlow is flexibility problem. That is, any modification or upgrade to the protocol results in a new hardware replacement which can increase the cost considerably. In conclusion, the only feasible, cost effective approach to realize TagFlow is using the DPN approach.

Extending SDN to enable simple programmability for data plane functionalities and to support the capability of defining or redefining interfaces for data plane functionalities, along with publishing those interfaces to control plane elements and network applications, reduces operational and capital expenses, because we can add or remove or modify data plane functionality by simple programming. Thus, we can reduce the complexity of maintenance, and decrease the life-cycle costs often observed in hardware-based inflexible data plane elements. Simple modification to interfaces to access new data plane functionalities enhances the capabilities and improves the efficiency of network applications and control plane elements.

Regarding the scalability of our proposal there three types of scalability to note in our scope:

Scalability in terms of number of tags active in the network. The only effect of total

number of active tags in the network switch is on the forwarding table size. As the number of tags in the network grows, the forwarding table can grow correspondingly. The data structure for storing forwarding table in most modern networking switches is *Hash Table* (also known as hash map). Hash table is an associative array abstract data type to store key value pairs. The time complexity of all insert, delete and search operations on a hash table of size n in big O notation are *O(1)*. So, the search mechanism is independent of the number of rows and also number of tags in the forwarding table. So, TagFlow is scalable in terms of forwarding table size.

Scalability in terms of forwarding performance. We performed multiple experiments using different number of CPU cores as the processing power to measure the behavior of TagFlow on different packet sizes. Particularly, Figure 4.17 illustrates TagFlow throughput using different packet sizes. The x-axis shows the number of CPU cores used to forward packets. We used from one to ten CPU cores on six different packet sizes to see what is the effect of increasing the processing power on TagFlow throughput? The result suggests, as we increase processing power, the throughput increases almost linearly. Hence, TagFlow forwarding performance is scalable using more processing power.

Scalability in terms of network size. As the number of nodes in the network grows the packet can potentially spend more time traveling in the network and pass through larger number of hops or switches. The question is how does that affect the TagFlow operation? There are two aspects to this question: switch level and network level. First, effect on a single switch (i.e., switch level effect). TagFlow is a classification mechanism implemented and deployed on every switch and aims at simplifying the classification mechanism. In traditional SDN, packet classification on every switch is independent from another. So, higher or lower classification load on a switch does not affect any other switch. Even failure of one switch does not lead to failure of another. The same holds for TagFlow. TagFlow simplifies the classification on every single switch independent from others. So, increasing and/or decreasing the size of network has no effect on the performance of a single switches. Second, global effect on the whole network (i.e., network level effect). TagFlow has two main objectives: i) to free the core from classification load and ii) to use the freed capacity for useful applications. The switch level effect in the previous paragraph refers to the former TagFlow objective and network level

effect refers to the latter TagFlow objective. When TagFlow simplifies the classification load using tag-based approach, on every single switch there is a small amount of time saved in classification. Figure 4.19 shows that small amount of time on different network sizes. As the figure depicts, as the more number of nodes exist on the network, the larger amount of time will be saved via the TagFlow simplified classification. That is, the time saved via simplified classification is linearly increased as the number of nodes in the network increase. Figure 4.19 shows the time saved through simplified classification for only one packet. Next, in Figure 4.20 we project the same figure as 4.19 on 1000 packets to see how scalable is the time saving caused by simplified TagFlow classification. Figure 4.20 shows as the number of packets increase in the network, the cumulative latency decreased almost linearly. In conclusion, the aforementioned two figures illustrate two conclusions: First, the TagFlow latency reduction, linearly scales with the number of nodes on the network and second, the TagFlow latency reduction is linearly scaled with number of packets transmitting through the network.

4.6 Summary

In this chapter, we propose TagFlow, a simple tag based classification method as the replacement for many-field matching techniques such as OpenFlow which is reasonably fast and enables classification programmability. Our evaluation results show this method saves considerable processing power that can be used for application layer classification. Even if OpenFlow moves to ASIC our TagFlow still could be faster using ASIC.

Chapter 5

User-Defined Switch Actions in DPN Data Plane

5.1 Introduction

OpenFlow is a wide-spread SDN API. This API installs a set of <*match, action*> rules on network switches. Any flow matching to the rule, receives the corresponding action. OpenFlow proposes a programmable control plane but a configurable-only data plane. That is, the user can write a program on top of the controller to perform a task such as load balancing. By the configurable data plane we refer to the TCAMs that are only able to match flows against a predefined set of fields and the OpenFlow enabled switch (i.e., OpenFlow data plane) can be configured to execute some of the predefined actions on packets. So, OpenFlow has two major shortcomings:

- *Predefined flow field-matching*: OpenFlow defines a set of fixed predefined protocols and fields and instructs the data plane to match all the packets against them.
- *Predefined actions*: OpenFlow defines a small set of actions including forward, drop and meter to be executed on packets. The set is not extensible or programmable.

In this chapter we relax the latter limitation of OpenFlow and propose User-Defined Actions

(UDA) to increase the flexibility of current SDN definition. Our contribution in this chapter are two folds. First, we show usecases of UDAs and propose a programmable architecture that extends the traditional SDN definition to support UDA. Second, via our evaluations we show the feasibility of UDAs in terms of throughput and their effectiveness in terms of ease of programmability.

5.2 Related Work

To our best knowledge, there is a limited attention to user-defined actions in the literature. However, there are some works that use labeling and tagging approaches as well as proposing software solutions for SDN data plane. The only work (we are aware of) which focuses on user-defined actions is [59] in which authors propose a design of a chip as a replacement for TCAM. They propose the Reconfigurable Match Tables (RMT) model, a new RISC-inspired pipelined architecture for switching chips, and identify the essential minimal set of action primitives to specify how headers are processed in hardware. RMT allows the forwarding plane to be changed in the field without modifying hardware. As in OpenFlow, the programmer can specify multiple match tables of arbitrary width and depth, subject only to an overall resource limit, with each table configurable for matching on arbitrary fields. RMT allows the programmer to modify all header fields much more comprehensively than in OpenFlow. The paper describes the design of a 64x10 Gbps ports switch chip implementing the RMT model and claims that flexible OpenFlow hardware switch implementations are feasible at almost no additional cost or power [59]. However, even though the design looks promising, it is never implemented in the real world to prove the claims. Such works shows the community is skeptical about the capability of software controls. In this chapter, we argue that software controls are capable and sound for such purposes.

Another work that overlaps in problem domain with ours is [58]. Authors introduce a tagging architecture in which middleboxes export tags to provide the necessary casual context (e.g., source hosts or internal cache/miss state). SDN controllers can configure the tag generation and consumption operations using their API. These operations help bindings between packets and their origins as well as ensuring that packets follow policy mandated paths. Middleboxes

may use tags to execute an action dynamically on the packet. This research is different from ours in two folds. First, it more focuses on how actions are executed rather than ours in which we consider how they are defined. Second, it discusses actions as middleboxes in contrast to our target in which we study the feasibility of deploying actions within the same box as the switch similar to basic OpenFlow actions implemented using TCAM. For completeness of this section we can mention other related SDN technologies that can be combined with UDA. For example TagFlow which is a flow based switching system. Similar to OpenFlow case we discussed in our experiments, UDAs can be combined with TagFlow as well.

5.3 User-Defined Action Usecases

The main reason behind the need for user-defined actions is to offer a more flexible SDN data plane compared to current limited and hardware-centric SDN data plane. The main challenge in front of this is the tradeoff between flexibility and performance. In this chapter, we discuss this tradeoff and evaluate the feasibility of providing such a flexibility using real world experiments. In addition, there are other reasons that make user-defined actions interesting for the community.

In the OpenFlow context (and to some extent in the current SDN context) a *northbound* application is implemented on top of controller and calls the controller to perform a task. In contrast, *southbound* means the layer consist of switches and forwarding plane. We believe data plane programmability is as important as that of control plane programmability. However, there are less work from the community in this area compared to OpenFlow model that advertises control plane programmability. Todays, northbound applications are considered as hot topic (e.g.,[33]). However, giving examples of useful applications in this area is not a trivial task as there are several limitations in the control plane.

The major limitation in the control plane is the lack of access to the packet payload. That is why it is infeasible to have applications that filter more than a bitstream match (i.e. the OpenFlow classification model) on the packet. For example, an Intrusion Detection System needs to look at the whole packet and even buffer a couple of packets to detect an intrusion while sending packets to the controller for intrusion detection is infeasible because of perfor-

Table 5.1: Comparison of overhead in northbound (i.e., FRESCO) applications versus southbound user-defined actions versus virtual appliance middleboxes

	SDN overhead (msec)		NFV overhead (msec)
	Control Plane [33]	**Data Plane (UDA)**	**Virtual Appliance**
PortScanner Detector	7.196	0.000001	0.001280609
BotMiner Detector	15.421	0.000004	0.001630215
P2P Plotter	11.775	0.000004	0.001312178

mance considerations. With current proposals from Network Function Virtualization (NFV) community, all data plane related tasks should be executed in dedicated boxes that are possibly getting advantage of virtualization technology. While this promising approach is beneficial in many ways, it can be a limiting factor as well. Although for heavy tasks (such as running a heavy IDS) we may need an external box, in case of light small tasks that come handy in networking, making a separate box is not a reasonable approach. Current SDN proposal only offers northbound interface to realize such applications.

To support our argument, we implemented some examples of northbound security applications from a related work [33] in three deferent architectures to see the difference. In particular, once as UDAs in the data plane (i.e. southbound) and once in the form of virtual appliances as Network Function Virtualization (NFV) components and compare it with the northbound application architecture proposed in [33].

Specifically, we implemented three actions; Port scanner detector and BotMiner Detector and P2P Plotter based on the descriptions in [33]. Port scanner detector looks for repeated attempts to connect to a closed port on a system (i.e., the victim) from another system (i.e., the attacker). This kind of detectors help to find worm-infected machines that are scanning the network for new victims. BotMiner detector is another simple action that clusters network nodes based on the output of port scanner detector to detect bots via co-clustering of nodes producing network anomalies. Finally, P2P Plotter is a malware detection service that looks for two characteristics to detect peer-to-peer malware. First, P2P malware usually produce less amount of traffic compared to benign P2P client software. Second, P2P malware nodes have less churning rate. That is, P2P malware nodes commonly establish longer connections compared to benign clients. P2P Plotter co-clusters the nodes that exhibit both features.

Table 5.1 illustrates the difference between the overhead of running three applications as

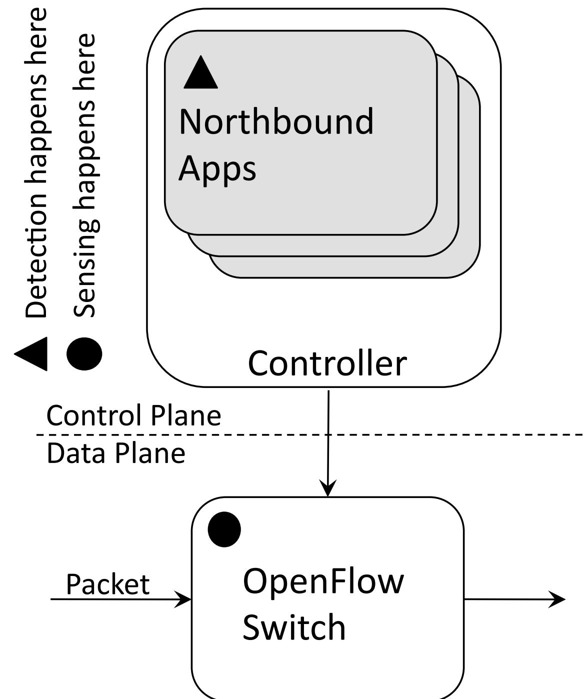

Figure 5.1: Northbound application overhead evaluation architecture

UDAs on the data plane (i.e., southbound) versus the control plane (i.e., northbound) and as a virtual appliance (i.e., NFV). To understand the difference between these three architectures we explain each of them briefly.

5.3.1 Northbound (or Control Plane) Applications

The control plane implementations show the overhead delay of running elements on FRESCO framework. FRESCO is a Click [155] like framework to develop elements for security applications at northbound. Figure 5.1 indicates the architecture of northbound application. In a traditional OpenFlow inspired SDN architecture, the data plane consists of an open flow enabled switch that depending on the specific OpenFlow version support the related features. On the control plane side, there is a programmable OpenFlow controller with a set of applications running on top of the controller. Different controllers may support different languages and features for the guest (i.e., northbound) applications. A key point to understand the underlying

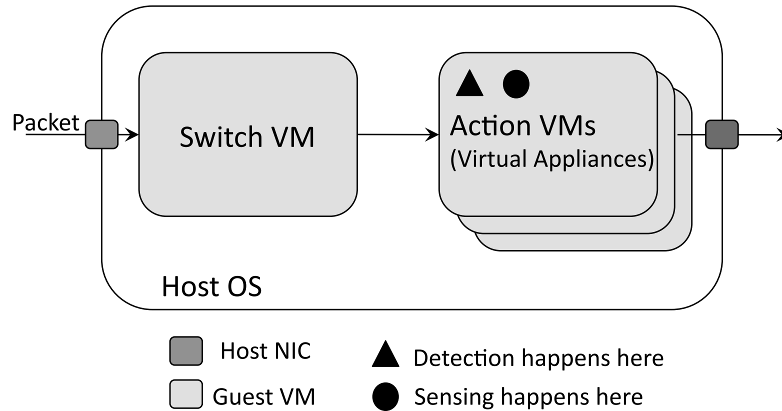

Figure 5.2: NFV application overhead evaluation architecture

reason behind the considerable difference in overheads in Table 5.1 is to note where the sensing and detection happens physically. The small black triangle shows that the detection mechanism happens on the controller based on the sensing information captured by data plane (denoted by a small black circle). Therefore, the sensing data goes via the wire from switch to the controller that produced some overhead.

5.3.2 NFV Appliances

Figure 5.2 indicates the architecture of NFV application we use for evaluation. The application is located within a guest virtual machine. The switch running the application is also running in another virtual machine and the connection among them is via the virtual network in the host operating system. Table 5.1 shows NFV applications are almost 100 times faster than control plane applications. In fact, it depends on the architecture. Since we put both the switch and the virtual appliance within a single virtual network, NFV shows a considerable performance increase. However, in a typical NFV scenario, switch is located on a separate machine from virtual appliances. In the latter case, the difference between NFV and northbound application is less as we move from virtual network to a physical network that connects machines hosting switch and appliance VMs.

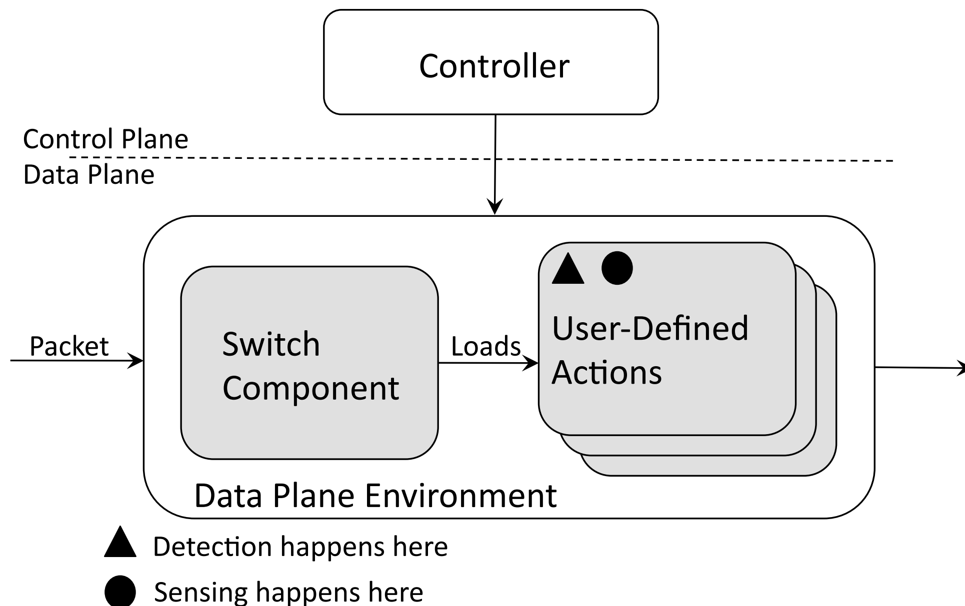

Figure 5.3: UDA overhead evaluation architecture

5.3.3 User-Defined Switch Actions

Figure 5.3 illustrates the architecture we use for UDA evaluation. The figure shows a data plane environment that settles switch and actions as different components. We use Click Modular Router to implement this architecture. Hence, the switch as well as UDAs are all Click elements. As it is shown in Figure 5.3, we do not connect actions and switches in the serial manner (as the regular use case in Click configuration files). Rather, we implemented actions as a passive-like (or plugin-like) element and include it in the switch code in a way that the switch can load the external user-defined action element to execute it. The overhead of UDA deployed on the data plane is in nanosecond scale while the others are in microsecond scale. That is because, first, sensing and detection happens physically at the same machine and no information is transferred over physical or virtual network and second, the switch and UDAs are all on a single shared platform in contrast to NFV case in which the packet should go through the whole protocol stack of virtual appliances. In conclusion, data transmission and VM overhead are two major limitations of alternative methods that make UDA an interesting candidate for applications.

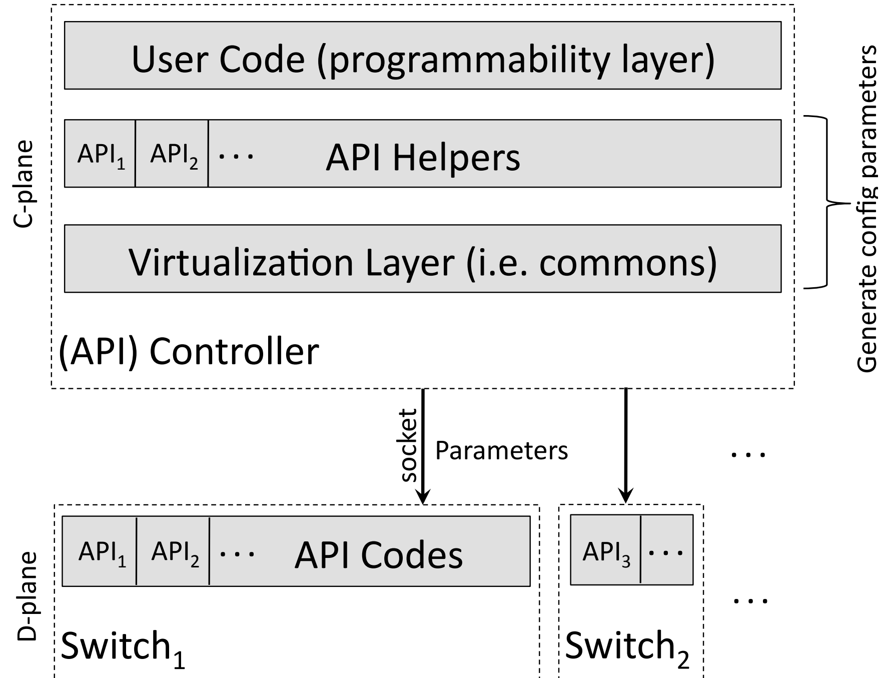

Figure 5.4: UDA/API control and data plane architecture

5.4 System Architecture

In this section we introduce the architecture that enables UDAs to be defined and programmed. Figure 5.4 depicts the overall architecture of our proposal. Following current popular SDN architecture, our system consists of two physically separated planes: control plane (at the top of the figure) and data plane (at the bottom of the figure). Before digging into our design objectives we define three main roles involved in the system. We consider an environment to publish a typical (commercial) service for end-users such as online video streaming service.

- *Network/Infrastructure provider*: Is the entity that provides the hardware and cabling infrastructure plus the programmable SDN environment. Infrastructure provider gives the user, a programmable environment including control and data plane programmability features to develop, test and run arbitrary networking software. We do not define the exact form of such an environment since depending on the application different technologies can

be used for this objective. An example technology to provide a programmable environment is network virtualization where each user has one slice and a set of resources fully isolated from other slices. There are a couple of ways to implement this kind of environment in the literature (e.g., OpenTag [152]) that are out of the scope of this thesis. Our focus in this chapter is how the infrastructure provider designs the a programmability feature of data and control planes particularly for UDAs. So, we leave other concerns such as isolation and management issues for future work.

- *User (or service provider)*: Is the entity that is going to program the data and control planes to publish a service for end-user. Needless to say, infrastructure provider and user can be the same entity.

- *End-user (or service user)*: Is an individual who consumes the service provided by the service provider (or user).

After defining the terminology we use to explain our architecture, we move on to defining our design objectives as follows:

- To keep the current architectural benefits of SDN caused by separation of data and control plane and related abstractions.

- To extend SDN data plane to support programmability. That is, using our architecture, the user is able to define arbitrary API or UDA and use it.

- To extend SDN control plane to support defining new APIs such as UDAs.

The data plane of our architecture consists of switches that can locate data plane API codes. Switches may have different types of APIs and each API may have its own set of parameters. Control plane can modify API parameters on the data plane (i.e., switches). The control plane is made up of three layers; The *Virtualization Layer* is responsible for common tasks including basic functions (e.g., connectivity) and gathering the information from the switches about available APIs and making an abstraction of network topology. The *API Helpers* layer are procedures that implement methods each API needs on the control plane. For example, if an API should read some information from the data plane and then execute some calculations on

Table 5.2: Comparison of ease of programmability in Northbound applications versus southbound UDAs versus middleboxes

	Lines of Code (LoC)		
	SDN		Middlebox [1] (C)
	Control Plane (i.e. NOX in Python) [1]	Data Plane UDA (C + config)	
TRW-CB [156]	741	196 (181+15)	1060
Rate Limit [157]	814	225 (205 + 20)	991

the data gathered from the network on the controller, then corresponding API helper component is responsible for such a task. Put it other way, each API function spans over both data and control plane. Hence, a part of the API code is physically on the data plane (indicated as API codes in the figure) and another part is located on the control plane (indicated as API helpers in the figure). Based on the user code, API helpers along with the virtualization layer compute and install appropriate parameters on the data plane. Finally, the *User Code* layer includes the user code on the controller.

The most important difference between our architecture and traditional SDN controllers is that we locate the API codes on the data plane. To clarify the issue, we use OpenFlow as an example of traditional SDN API which spans over both data plane (i.e., OpenFlow enabled switches) and control plane (i.e., OpenFlow controller). At the data plane side, OpenFlow uses hardware-centric data plane components such as TCAM that implements the data plane side of the API logic on the switch including the OpenFlow actions. In contrast, our architecture fosters a software-defined data plane that can include multiple user-defined APIs. We realize and evaluate UDAs as an example API in this architecture. So we look at UDAs as special APIs that can be loaded in a plugin-like fashion to the switch to be executed accordingly. The switch triggers execution of UDAs in a similar way to the traditional SDN. That is, once a flow is matched against a row in the switch forwarding table, a set of actions defined by the user for that specific flow are executed on every packets of that flow.

5.4.1 Ease of Programmability

In this chapter, we propose a solution for programmability of the data plane. Accordingly, an important factor to show the effectiveness of the programmability is *ease of programmability*

Table 5.3: Comparison of ease of programmability in northbound versus southbound (i.e., UDAs)

	Lines of Code (LoC)	
	Northbound implementation via FRESCO (Python + config) [33]	**Southbound implementation via UDA (C + config)**
Port Scanner Detector	229 (205+24)	264 (249 + 15)
BotMiner Detector	352 (312 + 40)	575 (548 + 27)
P2P Plotter	259 (227 + 32)	302 (281 + 21)

which refers to how easy it is to develop an arbitrary program using the proposed solution or method. To support our proposal, we implemented two anomaly detection algorithms from a related work (i.e., [?]) to compare ease of programmability in different architectures. Specifically, we implemented Rate Limit [157] and Threshold Random Walk with Credit Based Rate Limiting (TRW-CB) [156] algorithms. The TRW-CB is a method to detect infected hosts by worms that are already started scanning other nodes. It assumes the number of successful connection attempts from non-infected nodes is higher than infected nodes. The TRW-CB applies a likelihood ratio test to classify nodes using a queue of TCP SYNs for every node that is not received the SYNACK response yet. In case of time out or TCP RST message for any queued SYN, the likelihood ratio of that specific host will be incremented by the algorithm. Similarly, Rate Limit algorithm assumes infected nodes try to connect to a large number of nodes in a short span of time. It keeps track of recent attempts for connection from all the hosts in the network and matches new attempts against the recent attempts list. Connection from nodes that perform many attempts are delayed in a queue of a limited size. Once the queue reaches a threshold size, the source host will be considered as an infected node.

Table 5.2 compares the size of source code (i.e., LoC) to implement aforementioned detection methods in different architectures. The control plane implementations refer to Python language code written on top of the NOX controller. The middlebox implementations are in C language. By the term middlebox we refer to implementing the detection method on a standalone machine that sends/receives packets to/from its physical NICs.

In contrast to typical Click element usecases (as we mentioned in Section 5.3.3) we implemented UDAs as a passive-like Click elements and include them in the switch code in a way that the switch can load the external UDA element to execute it. The reason behind such an implementation is ease of programmability. Since the UDA is implemented in a separate element, the

code is cleaner and more extendable. Furthermore, including the action element in the switch code takes a few lines of code so that the extra work caused by providing programmability is reasonable. This not only keeps the action code clean, but keeps the switch code simple and clean even after adding UDAs. Since we use Click to implement our UDAs, we calculate the LoC using the summation of Click configuration file and the LoC of the element source code. Note that we do not consider the LoC of the Click Router itself similar to NOX case that we only include the LoC of the program written on top of NOX. Moreover, Our controller is a standalone program that connects to the switch using sockets. It can retrieve the list of online switches and the catalog of available APIs from every switch. Using these information user can write programs. Currently, our controller supports only C++ programmability. We build a Python wrapper over some functions of the controller so that the user can write Python script to program the controller and manipulate parameters on data plane. Table 5.3 indicates the comparison of ease of programming using a northbound programming framework and UDA implemented at southbound. FRESCO as a northbound application development framework is made for easier application development at controller. However, comparing UDA and FRESCO using the three sample applications, we can see almost similar lines of code in both solutions. The minor difference we see can be the result of difference programming languages. In fact, Python scripting language (used in FRESCO) can reduce the code size compared to C++ programming language. We believe the majority of the difference we see in Table 5.3 is caused by the difference between Python and C++. Hence, we consider similar lines of code using two different approaches eventhough FRESCO is an additional framework on top of the controller. If we count the LoC of the FRESCO framework itself (excluding the controller LoC), the result will be different and UDA lines of code will be less than FRESCO applications. In conclusion, using UDAs implemented on data plane we can reduce the LoC by 72.9% and 79.3% compared to implementing the same functionality on control plane and as a standalone middlebox, respectively.

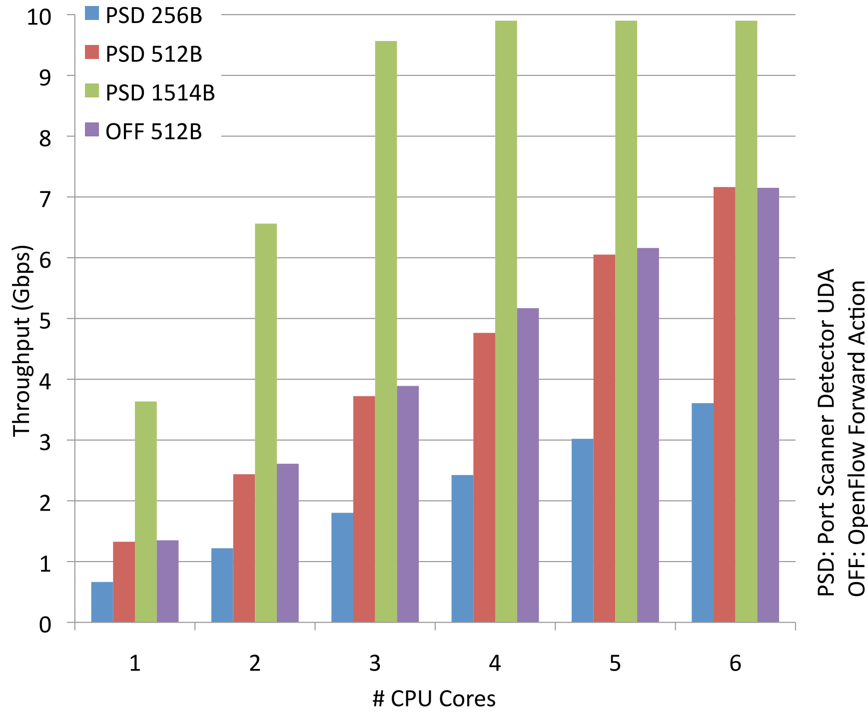

Figure 5.5: OpenFlow extended to support UDAs

5.5 Throughput Evaluations

In Section 5.3 we reviewed some PC-based experiments to show per packet overhead using different architectures. In this section, we consider testing UDAs under higher loads to see how it performs. Particularly, we extend the OpenFlow to support such a feature and measure the overhead of UDAs. We use the same UDAs as in experiments in Section 5.3 (i.e., Botminer Detector, Portscan Detector and P2P Plotter). The experiment setup we use in all evaluations is as follows. We use Xena packet generator to generate 10 Gbps traffic and send it to FLARE switch that hosts our UDAs. FLARE switch is a programmable switch using Click environment. For more details on FLARE please refer to [108].

5.5.1 OpenFlow Plus UDAs

For OpenFlow experiments we use our software OpenFlow implementation on FLARE. Since FLARE uses the Click Modular router environment we implement all actions as Click ele-

Table 5.4: Overhead of heavy User-Defined Actions using extended OpenFlow

	# Processor cores	**6**	**7**	**8**	**9**	**10**	**11**
OpenFlow + UDAs (Gbps)	**BotMiner Detector**	2.3	2.9	3.1	3.5	3.9	4.3
	P2P Plotter	2.4	2.8	3.2	3.6	4.0	4.4

ments. In case of OpenFlow, we use the original open source and publicly available OpenFlow implementation as a static library linked against the OpenFlow Click element and use library functions in the element and extend it to support UDAs. Figure 5.5 illustrates the throughput of running UDAs on OpenFlow using different packet sizes. We compared PSD with OFF. We use one to five processor cores to show the throughput is linearly increasing while we increase the number of cores.

5.5.2 More Complex UDAs

Compared with portscan detector UDA, the botminer detector and P2P Plotter UDAs are more complex since they look for specific aspects of the traffic and co-cluster the results to make the decision. For both of them we use multiple Click elements for implementation. Therefore, they have lower throughput because of the overhead of using multiple elements. In case of portscan detector we applied some optimizations to get the results presented in Figure 5.5. Particularly, before optimization, we used the CheckIPHeader Click element to set the IP address annotation on the pact header to prevent segmentation fault on our portscan element that was checking IP header on every packet. So, we embedded the annotation functionality within the portscan element. As the result, removing the overhead of using additional element, we reduced the overhead of portscan detector element from 0.000004 milliseconds to 0.000001 millisecond. In the same way, we can reduce the overhead of botminer detector and P2P Plotter UDAs since they are using the Counter and DelayUnqueue elements. Rather, we use them for another experiment. That is, how the throughput increases with the increase of the number of processor cores on heavier UDAs. Table 5.4 indicates that the throughput follows a liner increase even up to 11 cores. As FLARE provides more than 30 cores, we can keep experiencing the same linear increase of performance which we exclude from the figure for brevity. Also, we excluded the results from one to five cores for briefness. We conclude that even for heavier tasks we can consider UDAs using more number of processor cores.

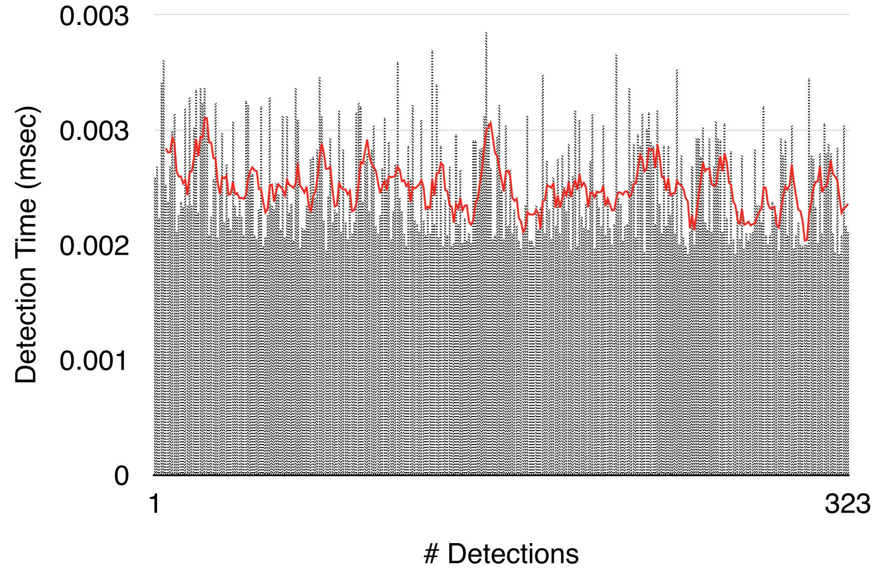

Figure 5.6: Per attack detection time of Portscan Detector UDA on the attack trace from [1]

5.5.3 Portscan Detector UDA Experiment on Attack Trace

Most of the experiments we discuss, focus on performance of UDAs. For completeness we conducted a test on an attack trace from [?] to show if the UDA we made really detects attacks or not. Obviously, as we are not proposing a detection algorithm, our measurement is not based on popular intrusion detection metrics such as false negative and false positive. Instead, we select a portion of the detection attack along with the detection time. Figure 5.6 shows the detection time of a random set of more than 300 detections the UDA fired on the attack trace. The trace consists of about 2 million packets attacking from three computers to three targets. The red line shows the trend of detection time which illustrate reasonable fluctuation we usually see in a software and we do not see an unusual increase or decrease in detection times.

5.6 Summary

In this chapter, we propose UDAs as an extension to current proposal from SDN community. We propose an architecture and an extended version of OpenFlow to support such a functionality in Click environment. We illustrated evaluations in terms of throughput and ease of

programability. We believe the community should pay more attention to the programmability of the data plane in order to have a better SDN.

Chapter 6

Conclusion and Future Directions

6.1 Summary

Software-Defined Networking (SDN) is a new networking paradigm in which we make abstractions over the network in a single point and use it to manage the network. Moreover, in contrast to current networking devices (e.g. network switches) SDN provides a programmable environment to make more flexibility in the way we manage our networks. SDN unlocks innovation on networking via providing programming interfaces for engineers compared to current command based devices (e.g. network switches). In this thesis, we recognize two major problems in the current SDN data plane: a) SDN data plane uses a limited set of predefined packet headers (e.g., source and destination IP address) to classify packets for switching and repeats checking all header fields for classification on every switch in the network and b) SDN data plane applies a set of predefined actions on every packet (e.g., forward and drop). To address these problems we surveyed current efforts in SDN and showed there is a lack of research on the SDN data plane. We also posit that there is a need for a more flexible, programmable data plane to be able to solve many networking problems. Hence, as our strategy we take the data plane-based approach to propose solutions.

Based on our proposal, to address the first problem we propose TagFlow. The main contribution of TagFlow are two folds: a) offloading the packet classification to the network edge devices and b) getting advantage of the freed capacity at the core, we provide the chance to get advantage of sophisticated application layer classification in the network. Our experiments

show TagFlow can be about 40% faster in forwarding and classification compared to the state of the art technologies. To address the second problem, we propose User Defined Action (UDA). UDAs let SDN programmers to freely build arbitrary use cases specific to their own needs without the limitation of traditional predefined actions in SDN. We demonstrate UDAs can elevate millisecond-scale action running time of current proposals to nanosecond-scale. Also, regarding ease of programmability, we show that our proposal decrease the lines of code of by more than 70% compared to the current alternatives.

All the solutions we reviewed in this thesis as different mitigation mechanism for current SDN problem using DPN approach can be considered as APIs of the DPN. Such programmable mechanisms play as interfaces to the dat plane of DPN and let programmers adjust their behavior arbitrarily.

6.2 Future Directions on DPN

Here we mention our potential future work and also possible future directions for the community towards successful DPN research.

6.2.1 Combining SDN and NFV

Recently, the community started looking at the synergy between SDN and Network Function Virtualization (NFV). NFV aims at virtualization of data plane functionalities on top of a virtualized environment. Drawing a line between SDN and NFV (i.e., considering the SDN as the part that transfers packets among nodes and the NFV as the part that process packets) may suffer from a few shortcomings. First, SDN defines southbound interface as the interface between controller and switches. At the switch side, usually an inflexible hardware-based (e.g., TCAM-based) device plays the role of data plane. If network operator can programmatically define and publish arbitrary interfaces for the controller, we may open up the opportunity for a series of enhancements in the data plane. Secondly, at the NFV side, we can see middleboxes and virtual appliances usually stick to a non-evolvable and non-extensible set of protocols. If we extend NFV architecture in the way that it can work consistently with SDN controllers and both sides can share open interfaces, we can have flexible and evolvable data plane while

we leverage the management power of SDN [158]. There are a few researches focusing on the combination of SDN and NFV such as [159].

6.2.2 Data Plane Abstractions

SDN defines almost comprehensive abstractions over the network control plane that let the architecture enable programmability. In contrast, there is a little effort towards proposing an abstraction over data plane. Among available alternatives, Finite State Machine (FSM) and Extended FSM gathers some more attentions as a promising way of model the complex functionality of data plane [160, 161]. We may also need to open up some interfaces that expose resources of switch such as storage (e.g., for in-network caching), processors (e.g., for transcoding and encryption), and packet queues (e.g., dynamic adjustment of queue sizes for specific applications) to enable programmers to develop their solution faster and more portable across hardware platforms.

6.2.3 Data Plane Verification

Besides methods of verification for forwarding state [77], the data plane software also needs verification to make sure that it satisfies the requirements. Software engineering practices has established methods for verification such as symbolic execution, input/output testing and passive testing. Recently, researches try to apply these methods to verify data plane functionality. For example, in [162], authors optimize a symbolic execution engine to go through alternative execution paths of the data plane in order to find bugs. Moreover, FlowTest [80] focuses on a trace or a interleaved sequence of packets that logically triggers a chain of specific state transitions in the data plane using state machines where each state represents a state of the data plane.

6.2.4 Netwrok Programming languages

As we showed in this thesis, the main focus of the SDN research was on programmability of the control plane and producing software that runs on top of control plane. After a couple of years passing from the initial SDN idea the community feels there is a need for a way to program the

data plane in a high-level manner, independent from underlying protocols. Following the basic Deeply Programmable Network idea, the research community started discussion around domain-specific programming languages that are independent from protocols. That is, the software producer just defines the desired behavior s/he wants from the switch in a domain-specific programming language and the rest is the compiler responsibility to translate the high level behavior definition to a switch-understandable code that realizes the desired behavior. Projects such as P4 [163] are following such an objective. P4 or Programming Protocol-independent Packet Processors is an effort to introduce a programming language which is designed for dataplane programmability and at the same time is not specific to any protocol and does not need to support a protocol to be usable. On the other hand, projects such as OpenFlow tend to expose programming interfaces that are limited to protocols that they support. That trend of abstraction we see in the community together with appearance of commodity high speed packet processing hardware can release the current networking industry from the monopoly that formed since a few decades ago by companies producing special purpose proprietary networking hardware and software and open up the innovation space to the whole community.

References

[1] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 161–180. xi, 79, 84

[2] O. N. Foundation, "Software-defined networking (sdn) definition." [Online]. Available: http://goo.gl/O2eTti 1

[3] A. Doria, R. Gopal, H. Khosravi, L. Dong, J. Salim, and W. Wang, "Forwarding and control element separation (forces) protocol specification," 2010. [Online]. Available: https://ietf.org/wg/forces/charter/ 1, 35

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

[5] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, 2012. 2

[6] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *Proceedings of Network and Distributed Security Symposium*, 2013. 2, 65, 66

[7] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *USENIX NSDI*, 2013. 2, 26, 30

[8] A. Nakao, "Deeply Programmable Network - Emerging Technologies for Network Virtualization and Software Defined Networking (SDN)," 2013. [Online]. Available: http://www.itu.int/en/ITU-T/academia/kaleidoscope/2013/Documents/Presentations/kaleidoscope-2013-kyoto-aki-nakao-short.pdf 5, 26

[9] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "On reliability-optimized controller placement for Software-Defined Networks," *China Communications*, vol. 11, no. 2, pp. 38–54, 2014. 12, 15

[10] M. Guo and P. Bhattacharya, "Controller Placement for Improving Resilience of Software-Defined Networks," in *IEEE ICNDC*, 2013. 12, 15

[11] Y. Jimenez, C. Cervello-Pastor, and A. J. Garcia, "On the controller placement for designing a distributed SDN control layer," in *IFIP Networking*, 2014. 12, 15

[12] F. J. Ros and P. M. Ruiz, "Five nines of southbound reliability in software-defined networks," in *ACM HotSDN*, 2014. 12, 15

[13] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, "Resilience of SDNs based On active and passive replication mechanisms," in *IEEE GLOBECOM*, 2013, pp. 2188–2193. 12, 15

[14] M. Kuzniar, P. Perešini, N. Vasic, M. Canini, and D. Kostic, "Automatic failure recovery for software-defined networks," in *ACM SIGCOMM HotSDN*, 2013. 13, 15

[15] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford, "Hotswap: Correct and efficient controller upgrades for software-defined networks," in *ACM SIGCOMM HotSDN*, 2013. 13, 15

[16] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "CAP for Networks," in *ACM HotSDN*, 2013. 13, 15

[17] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, "A replication component for resilient OpenFlow-based

networking," in *IEEE NOMS*, 2012, pp. 933–939. 13, 15

[18] D. Williams and H. Jamjoom, "Cementing high availability in openflow with rulebricks," in *ACM SIGCOMM HotSDN*, 2013, pp. 139–144. 13, 15

[19] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Skoldstrom, "Scalable fault management for openflow," in *IEEE ICC*, 2012, pp. 6606–6610. 13, 15

[20] U. C. Kozat, G. Liang, and K. Kokten, "Verifying forwarding plane con tivity and locating link failures using static rules in software defined networks," in *ACM SIGCOMM HotSDN*, 2013, pp. 157–158. 13, 15

[21] H. Kim, J. Santos, Y. Turner, M. Schlansker, J. Tourrilhes, and N. Feamster, "CORONET: Fault Tolerance for Software Defined Networks," in *IEEE ICNP*, 2012, pp. 1–2. 13, 15

[22] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: introducing openflow graph algorithms," in *ACM HotSDN*, 2014, pp. 121–126. 13, 15

[23] X. Guan, B.-Y. Choi, and S. Song, "Reliability and Scalability Issues in Software Defined Network Frameworks," in *IEEE GREE*, 2013. 14

[24] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 136–141, 2013. 14

[25] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM CCR*, vol. 41, no. 4, pp. 351–362, 2011. 14, 15

[26] 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 CCR*, vol. 41, no. 4, 2011, pp. 254–265. 14, 15

[27] M. Othman and K. Okamura, "Hybrid control model for flow-based networks," in *IEEE COMPSAC*, 2013, pp. 765–770. 14, 15

[28] T. Choi, S. Song, H. Park, S. Yoon, and S. Yang, "SUMA: Software-defined Unified Monitoring Agent for SDN," in *IEEE NOMS*, 2014. 14, 15

[29] S. Azodolmolky, P. Wieder, and R. Yahyapour, "Performance Evaluation of a Scalable Software-Defined Networking Deployment," in *IEEE EWSDN*, 2013. 14, 15

[30] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *ACM SIGCOMM HotSDN*, 2012, pp. 19–24. 14, 15

[31] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang, "Towards a secure controller platform for openflow applications," in *ACM SIGCOMM HotSDN*, 2013, pp. 171–172. 15

[32] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *ACM SIGCOMM HotSDN*, 2012, pp. 121–126. 15

[33] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined networks," in *Proceedings of Network and Distributed Security Symposium*, 2013. 15, 23, 24, 29, 72, 73, 80

[34] H. Li, P. Li, S. Guo, and S. Yu, "Byzantine-resilient secure software-defined networks with multiple controllers," in *IEEE ICC*, 2014. 15

[35] S. Song, S. Hong, X. Guan, B.-Y. Choi, and C. Choi, "NEOD: network embedded on-line disaster management framework for software defined networking," in *IEEE IM*, 2013. 15, 16

[36] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: defending SDNs from malicious administrators," in *ACM HotSDN*, 2014. 15, 16

[37] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, "The flowadapter: enable flexible multi-table processing on legacy hardware," in *ACM SIGCOMM HotSDN*, 2013, pp. 85–90. 15, 16

[38] N. Kang, J. Reich, J. Rexford, and D. Walker, "Policy transformation in software defined networks," in *Proceedings of the ACM SIGCOMM*, 2012, pp. 309–310. 15, 16

[39] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE INFOCOM Mini-conference*, 2013. 15, 16

[40] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *ACM CoNEXT*, 2013. 15, 16

[41] P. Peresini, M. Kuzniar, N. Vasic, M. Canini, and D. Kostic, "Of. cpp: Consistent packet processing for

openflow," EPFL, Tech. Rep., 2013. 15, 17

[42] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "Software transactional networking: Concurrent and consistent policy composition," in *ACM HotSDN*, 2013. 15, 17

[43] P. Peresini, M. Kuzniar, M. Canini, and D. Kostic, "ESPRES: transparent SDN update scheduling," in *ACM HotSDN*, 2014. 15, 17

[44] "Network OS: ONOS." [Online]. Available: http://onlab.us/tools.html#os/ 15, 17

[45] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, and T. Hama, "Onix: A distributed control platform for large-scale production networks," in *Proceedings of USENIX OSDI*, vol. 10, 2010, pp. 1–6. 15, 17, 18, 20

[46] S. Shirali-Shahreza and Y. Ganjali, "FleXam: flexible sampling extension for monitoring and security applications in openflow," in *ACM SIGCOMM HotSDN*, 2013, pp. 167–168. 15, 17

[47] K. Takagiwa, S. Ishida, and H. Nishi, "SoR-Based Programmable Network for Future Software-Defined Network," in *IEEE COMPSAC*, 2013, pp. 165–166. 15, 17

[48] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir, "Application-awareness in SDN," in *ACM SIGCOMM*, 2013, pp. 487–488. 15, 17

[49] J. C. Mogul and P. Congdon, "Hey, you darned counters!: get off my ASIC!" in *ACM SIGCOMM HotSDN*, 2012, pp. 25–30. 15, 17, 30

[50] G. Lu, R. Miao, Y. Xiong, and C. Guo, "Using cpu as a traffic co-processing unit in commodity switches," in *ACM SIGCOMM HotSDN*, 2012, pp. 31–36. 15, 17

[51] K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan, "Towards efficient implementation of packet classifiers in SDN/OpenFlow," in *ACM SIGCOMM HotSDN*, 2013, pp. 153–154. 15, 17, 28

[52] X. Jin, J. Rexford, and D. Walker, "Incremental update for a compositional sdn hypervisor," in *ACM HotSDN*, 2014, pp. 187–192. 15, 18

[53] D. Kreutz, F. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *ACM HotSDN*, 2013. 15

[54] S. Shin and G. Gu, "Attacking software-defined networks: a first feasibility study," in *ACM SIGCOMM HotSDN*, 2013, pp. 165–166. 15

[55] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *ACM SIGCOMM HotSDN*, 2012, pp. 1–6. 16

[56] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *ACM SIGCOMM HotSDN*, 2013, pp. 49–54. 16

[57] T. Mizrahi and Y. Moses, "Time-based updates in software defined networks," in *ACM SIGCOMM HotSDN*, 2013. 16

[58] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," *ACM SIGCOMM HotSDN*, 2013. 17, 38, 71

[59] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proceedings of the ACM SIGCOMM*, 2013, pp. 99–110. 18, 27, 71

[60] E. Kissel, G. Fernandes, M. Jaffee, M. Swany, and M. Zhang, "Driving Software Defined Networks with XSP," in *IEEE ICC*, 2012, pp. 6616–6621. 18

[61] Z. Bozakov and P. Papadimitriou, "Towards a Scalable Software-Defined Network Virtualization Platform," in *IEEE NOMS*, 2014. 18, 19

[62] Z. Bozakov and P. Papadim, "Autoslice: automated and scalable slicing for software-defined networks," in *ACM CoNEXT student workshop*, 2012. 18, 19

[63] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Computing*, vol. 17, no. 2, pp. 20–27, 2013. 18, 19

[64] O. M. C. Rendon, F. E. Solano, and L. Z. Granville, "A Mashup-Based Approach for Virtual SDN Management," in *IEEE COMPSAC*, 2013. 18, 19

[65] R. Guerzoni, R. Trivisonno, I. Vaishnavi, Z. Despotovic, A. Hecker, S. Beker, and D. Soldani, "A novel approach to virtual networks embedding for SDN management and orchestration," in *IEEE NOMS*, 2014.

18, 19

[66] A. Devlic, W. John, and P. Skoldstrom, “A use-case based analysis of network management functions in the ONF SDN model,” in *ACM EWSDN*, 2012. 18, 19

[67] H. Li, X. Que, Y. Hu, G. Xiangyang, and W. Wendong, “An autonomic management architecture for SDN-based multi-service network,” in *IEEE GLOBECOM Workshops*, 2013. 18, 19

[68] P. Thai and J. C. de Oliveira, “Decoupling Policy from Routing with Software Defined Interdomain Management: Interdomain Routing for SDN-Based Networks,” in *IEEE ICCCN*, 2013. 18, 20

[69] K. J. Kerpez, J. M. Cioffi, G. Ginis, M. Goldburg, S. Galli, and P. Silverman, “Software-defined access networks,” *IEEE Communications Magazine*, vol. 52, no. 9, pp. 152–159, 2014. 18, 20

[70] Y. Xu, Y. Yan, Z. Dai, and X. Wang, “A management model for SDN-based data center networks,” in *IEEE INFOCOM Workshops*, 2014, pp. 113–114. 18, 20

[71] R. Wang, D. Butnariu, and J. Rexford, “Openflow-based server load balancing gone wild,” in *USENIX Hot-ICE*, 2011, pp. 12–12. 18, 21

[72] N. Chowdhury and R. Boutaba, “A survey of network virtualization,” *Elsevier Computer Networks*, 2010. 19

[73] R. Ahmed and R. Boutaba, “Design considerations for managing wide area software defined networks,” *IEEE Communications Magazine*, vol. 52, no. 7, pp. 116–123, 2014. 20

[74] A. Alba, G. Alatorre, C. Bolik, A. Corrao, T. Clark, S. Gopisetty, R. Haas, R. Kat, B. Langston, N. Mandagere *et al.*, “Efficient and agile storage management in software defined environments,” *IBM Journal of Research and Development*, vol. 58, no. 2, pp. 1–12, 2014. 21

[75] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, “Veriflow: Verifying network-wide invariants in real time,” *ACM SIGCOMM CCR*, vol. 42, no. 4, pp. 467–472, 2012. 21

[76] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, “Automatic test packet generation,” in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012, pp. 241–252. 21

[77] P. Kazemian, G. Varghese, and N. McKeown, “Header space analysis: static checking for networks,” in *USENIX NSDI*, 2012, pp. 9–9. 21, 88

[78] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker, “An assertion language for debugging SDN applications,” in *ACM HotSDN*, 2014, pp. 91–96. 21, 22

[79] S. Hommes, F. Hermann, T. Engel *et al.*, “Automated Source Code Extension for Debugging of OpenFlow based Networks,” in *IEEE CNSM*, 2013. 21, 22

[80] S. K. Fayaz and V. Sekar, “Testing stateful and dynamic data planes with FlowTest,” in *ACM HotSDN*, 2014. 21, 88

[81] R. C. Scott, A. Wundsam, K. Zarifis, and S. Shenker, “What, Where, and When: Software Fault Localization for SDN,” UCB/EECS-2012-178, UC Berkeley, Tech. Rep., 2012. 21, 22

[82] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown, “Where is the debugger for my software-defined network?” in *ACM SIGCOMM HotSDN*, 2012, pp. 55–60. 21, 22

[83] M. Gupta, J. Sommers, and P. Barford, “Fast, accurate simulation for SDN prototyping,” in *ACM HotSDN*, 2013, pp. 31–36. 21, 22

[84] D. Volpano, X. Sun, and G. G. Xie, “Toward Systematic Detection and Resolution of Network Control Conflicts,” in *ACM HotSDN*, 2014. 21, 22

[85] T. Ball, N. Bjorner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, “VeriCon: towards verifying controller programs in software-defined networks,” in *ACM PLDI*, 2014. 21, 22

[86] “OFRewind.” [Online]. Available: http://archive.openflow.org/wk/index.php/OFRewind/ 21, 23

[87] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, “OFRewind: Enabling Record and Replay. Troubleshooting for Networks,” in *USENIX ATC*, 2011. 21, 23

[88] “Cbench.” [Online]. Available: http://www.openflowhub.org/display/floodlightcontroller/Cbench+(New)/ 21, 23

[89] “NICE.” [Online]. Available: https://code.google.com/p/nice-of/ 21, 23

[90] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A nice way to test openflow applications," in *USENIX NSDI*, 2012, pp. 10–10. 21, 23

[91] "SDN Troubleshooting Simulator (STS)." [Online]. Available: http://ucb-sts.github.io/sts/ 21, 23

[92] "Oflops." [Online]. Available: http://archive.openflow.org/wk/index.php/Oflops/ 21, 23

[93] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation," in *Passive and Active Measurement*. Springer, 2012, pp. 85–95. 21, 23

[94] "OFTest." [Online]. Available: http://archive.openflow.org/wk/index.php/OFTestTutorial/ 21, 23

[95] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *ACM SIGCOMM HotSDN*, 2012, pp. 43–48. 23, 24

[96] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *ACM SIGCOMM WREN*, 2009, pp. 1–10. 23, 24

[97] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011. 23, 24

[98] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *ACM SIGCOMM HotSDN*, 2013. 23, 24

[99] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *ACM HotSDN*, 2012. 23, 24

[100] S. Narayana, J. Rexford, and D. Walker, "Compiling path queries in software-defined networks," in *ACM HotSDN*, 2014. 23, 24

[101] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *ACM SIGCOMM*, 2013, pp. 87–98. 23, 25

[102] C. J. Casey, A. Sutton, and A. Sprintson, "tinyNBI: Distilling an API from essential OpenFlow abstractions," in *ACM HotSDN*, 2014. 23, 25

[103] H. Song, "Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane," in *ACM SIGCOMM HotSDN*, 2013, pp. 127–132. 23, 25

[104] "Named Data Networking (NDN)." [Online]. Available: http://named-data.net 26

[105] "XIA - eXpressive Internet Architecture." [Online]. Available: http://www.cs.cmu.edu/~xia 26, 32

[106] A. Shah, "Tilera targets intel, amd with 100-core processor," *PCWorld Solved, IDG News*, pp. 1–3, 2009. 27

[107] "Open Networking Foundation." [Online]. Available: https://www.opennetworking.org/index.php?lang=en/ 27

[108] Akihiro Nakao, "FLARE: Open Deeply Programmable Switch," in *The 16th GENI Engineering Conference*, 2012. [Online]. Available: http://www.pilab.jp/ipop2013/exhibition/panel/iPOP2013_uTokyo_panel.pdf 27, 28, 46, 82

[109] Y. Luo, P. Cascon, E. Murray, and J. Ortega, "Accelerating openflow switching with network processors," in *ACM/IEEE ANCS*, 2009, pp. 70–71. 27

[110] V. Tanyingyong, M. Hidell, and P. Sjödin, "Improving PC-based OpenFlow switching performance," in *ACM/IEEE ANCS*, 2010, p. 13. 28

[111] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, "Openflow switching: Data plane performance," in *IEEE ICC*, 2010, pp. 1–5. 28

[112] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan, "No silver bullet: extending sdn to the data plane," in *ACM HotNets*, 2013. 28

[113] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazieres, "Tiny packet programs for low-latency network control and monitoring," in *ACM HotNets*, 2013. 28

[114] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS SOSP*, 2009, pp. 15–28. 28

[115] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," *ACM SIGCOMM CCR*, vol. 40, no. 4, pp. 195–206, 2010. 28

[116] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li, and V. Prasanna, "Multi-dimensional packet classification on

FPGA: 100 Gbps and beyond," in *IEEE ICFPT*, 2010, pp. 241–248. 28

[117] S. Ando and A. Nakao, "L7Classier: Packet Classification applying Regular Expression to Packet Payload," IEICE, Tech. Rep., 2013. 29

[118] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000. 29

[119] M. Ahmed, F. Huici, and A. Jahanpanah, "Enabling dynamic network processing with clickos," in *Proceedings of the ACM SIGCOMM*, 2012, pp. 293–294. 29

[120] Akihiro Nakao, "VNode: A Deeply Programmable Network Testbed Through Network Virtualization," in *3rd IEICE Technical Committee on Network Virtualization*, 2012. 29

[121] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *USENIX NSDI*, 2012, pp. 11–11. 30

[122] K. Benton, L. J. Camp, and C. Small, "Openflow vulnerability assessment," in *ACM SIGCOMM HotSDN*, 2013, pp. 151–152. 30

[123] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: cheap SSL acceleration with commodity processors," in *USENIX NSDI*, 2011, pp. 1–1. 30

[124] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM CCR*, vol. 44, no. 2, pp. 44–51, 2014. 31

[125] H. Farhadi, P. Du, and A. Nakao, "User-defined actions for SDN," in *ACM CFI*, 2014. 31

[126] Y. Nishida and A. Nakao, "In-network ad-targeting through wifi ap virtualization," in *Communications and Information Technologies (ISCIT), 2012 International Symposium on*. IEEE, 2012, pp. 1092–1097. 31

[127] N. Kim, J.-Y. Yoo, N. L. Kim, and J. Kim, "A programmable networking switch node with in-network processing support," in *IEEE ICC*, 2012, pp. 6611–6615. 31

[128] J. Lee, J. Tourrilhes, P. Sharma, and S. Banerjee, "No more middlebox: integrate processing into network," *ACM SIGCOMM CCR*, vol. 41, no. 4, pp. 459–460, Oct. 2011. 31

[129] M. Shimamura, T. Ikenaga, and M. Tsuru, "A Design and Prototyping of In-Network Processing Platform to Enable Adaptive Network Services," *IEICE Transactions on Information and Systems*, vol. E96-D, no. 2, pp. 238–248, 2013. 31

[130] Akihiro Nakao, "WiVi: Wi-Fi Network Virtualization Infrastructure." [Online]. Available: http://www.ieice.org/~nv/NV-Sept_10-nakao.pdf 31

[131] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "Openradio: a programmable wireless dataplane," in *ACM SIGCOMM HotSDN*, 2012, pp. 109–114. 31

[132] L. Suresh, J. Schulz-Zander, R. Merz, A. Feldmann, and T. Vazao, "Towards programmable enterprise wlans with odin," in *ACM SIGCOMM HotSDN*, 2012, pp. 115–120. 31

[133] M. Handley, O. Hodson, and E. Kohler, "XORP: an open platform for network research," *ACM SIGCOMM CCR*, vol. 33, no. 1, pp. 53–57, Jan. 2003. 32

[134] "MobilityFirst Future Internet Architecture Project." [Online]. Available: http://mobilityfirst.winlab.rutgers.edu/ 32

[135] A. Nakao, "Deep programmability in communication infrastructure A closer look at software-defined networking and network functions virtualization, ITU-T News," 2013. [Online]. Available: http://goo.gl/Pn6Rik 34

[136] OpenNetworkingFoundation, "Openflow switch specification v1. 3.0," https://www.opennetworking.org/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, 2012. 35, 36

[137] L. Popa, N. Egi, S. Ratnasamy, and I. Stoica, "Building extensible networks with rule-based forwarding," in *USENIX OSDI*, 2010. 35

[138] W. Stallings, *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addison-Wesley Longman Publishing Co., 1998. 35

[139] V. Tanyingyong, M. Hidell, and P. Sjodin, "Using hardware classification to improve pc-based openflow switching," in *IEEE HPSR*, 2011. 36

[140] *100-Gigabit NPU with Integrated TM and CPUs, EZchip Technologies Ltd.*, 2013. [Online]. Available: http://www.ezchip.com/p_np4.htm 37

[141] K. Kannan and S. Banerjee, "Scissors: Dealing with header redundancies in data centers through SDN," in *IEEE CNSM*, 2012, pp. 295–301. 38

[142] B. Claise, "Rfc 3954: Cisco systems netflow services export version 9," *Retreived online: http://www.ietf.org/rfc/rfc3954.txt*, 2007. 39

[143] P. Phaal and M. Lavine, "sflow protocol specification version 5," http://www.sflow.org/sflow_version_5.txt, 2004. 39

[144] B. Claise, "Packet sampling (psamp) protocol specifications," *Internet draft, draft-ietf-psamp-protocol-01.txt, work in progress*, 2009. 39

[145] B. Claise, M. Fullmer, P. Calato, and R. Penno, "Ipfix protocol specifications," *IETF RFC3917*, 2004. 39

[146] A. R. Curtis, J. C. Mogul, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," *ACM CCR*, vol. 41, no. 4, p. 254, 2011. 39

[147] M. Al-Fares, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of USENIX NSDI*, 2010. 39

[148] P. Newman, W. Edwards, R. Hinden, E. Hoffman, F. C. Liaw, T. Lyon, and G. Minshall, "Ipsilon flow management protocol specification for ipv4 version 1.0," 1996. 40

[149] E. ROSEN, "Multiprotocol label switching architecture," *IETF RFC3031*, 2001. 40

[150] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "Lipsin: line speed publish/subscribe inter-networking," vol. 39, no. 4, pp. 195–206, 2009. 41

[151] P. Quinn, R. Fernando, J. Guichard, and S. Kumar, "Network service header," *Internet draft, draft-quinn-nsh-00.txt, work in progress*, 2013. 41

[152] R. Furuhashi and A. Nakao, "Opentag: Tag-based network slicing for wide-area coordinated in-network packet processing," in *IEEE ICC*, 2011. 41, 78

[153] Y. Chiba, Y. Shinohara, and H. Shimonishi, "Source flow: handling millions of flows on flow-based nodes," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 465–466, 2010. 41

[154] C. Inc., "Cisco guide to securing cisco nx-os software devices." 42

[155] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000. 74

[156] S. E. Schechter, J. Jung, and A. W. Berger, "Fast detection of scanning worm infections," in *Recent Advances in Intrusion Detection*, 2004, pp. 59–81. 79, 80

[157] J. Twycross and M. M. Williamson, "Implementing and testing a virus throttle," in *USENIX Security Symposium*, vol. 285, 2003, p. 294. 79, 80

[158] "Flare: Deeply programmable network node architecture." [Online]. Available: http://netseminar.stanford.edu/10_18_12.html 88

[159] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: enabling innovation in network function control," in *ACM SIGCOMM*, 2014, pp. 163–174. 88

[160] G. Bianchi, M. Bonola, A. Capone, C. Cascone, and S. Pontarelli, "Towards wire-speed platform-agnostic control of openflow switches," *arXiv preprint arXiv:1409.0242*, 2014. 88

[161] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in *ACM SIGCOMM*, 2014, pp. 377–378. 88

[162] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *USENIX NSDI*, 2014. 88

[163] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *ACM CCR*, vol. 44, no. 3, pp. 87–95, 2014. 89