OpenFlow-enabled dynamic DMZ for local networks

by

Haotian Wu

B.S., Beihang University, China, 2012

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2017

Abstract

Cybersecurity is playing a vital role in today's network. We can use security devices, such as a deep packet inspection (DPI) device, to enhance cybersecurity. However, a DPI has a limited amount of inspection capability, which cannot catch up with the ever-increasing volume of network traffic, and that gap is getting even larger. Therefore, inspecting every single packet using DPI is impractical.

Our objective is to find a tradeoff between network security and network performance. More explicitly, we aim at maximizing the utilization of security devices, while not decreasing network throughput. We propose two prototypes to address this issue in a demilitarized zone (DMZ) architecture.

Our first prototype involves a flow-size based DMZ criterion. In a campus network elephant flows, flows with large data rate, are usually science data and they are mostly safe. Moreover, the majority of the network bandwidth is consumed by elephant flows. Therefore, we propose a DMZ prototype that we inspect elephant flows for a few seconds, and then we allow them to bypass DPI inspection, as long as they are identified as safe flows; and they can be periodically inspected to ensure they remain safe.

Our second prototype is a congestion-aware DMZ scheme. Instead of determining whether a flow is safe or not by its size, we treat all flows identically. We measure the data rates of all flows, and use a global optimization algorithm to determine which flows are allowed to safely bypass a DPI. The objective is to maximize DPI utilization.

Both prototypes are implemented using OpenFlow in this work, and extensive experiments are performed to test both prototypes' feasibility. The results attest that the two prototypes are effective in ensuring network security while not compromising network performance. A number of tools for SDN network configuring and testing are also developed.

OpenFlow-enabled dynamic DMZ for local networks

by

Haotian Wu

B.S., Beihang University, China, 2012

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2017

Approved by:

Co-Major Professor
Don Gruenbacher

Approved by:

Co-Major Professor
Caterina Scoglio

Copyright

Abstract

Cybersecurity is playing a vital role in today's network. We can use security devices, such as a deep packet inspection (DPI) device, to enhance cybersecurity. However, a DPI has a limited amount of inspection capability, which cannot catch up with the ever-increasing volume of network traffic, and that gap is getting even larger. Therefore, inspecting every single packet using DPI is impractical.

Our objective is to find a tradeoff between network security and network performance. More explicitly, we aim at maximizing the utilization of security devices, while not decreasing network throughput. We propose two prototypes to address this issue in a demilitarized zone (DMZ) architecture.

Our first prototype involves a flow-size based DMZ criterion. In a campus network elephant flows, flows with large data rate, are usually science data and they are mostly safe. Moreover, the majority of the network bandwidth is consumed by elephant flows. Therefore, we propose a DMZ prototype that we inspect elephant flows for a few seconds, and then we allow them to bypass DPI inspection, as long as they are identified as safe flows; and they can be periodically inspected to ensure they remain safe.

Our second prototype is a congestion-aware DMZ scheme. Instead of determining whether a flow is safe or not by its size, we treat all flows identically. We measure the data rates of all flows, and use a global optimization algorithm to determine which flows are allowed to safely bypass a DPI. The objective is to maximize DPI utilization.

Both prototypes are implemented using OpenFlow in this work, and extensive experiments are performed to test both prototypes' feasibility. The results attest that the two prototypes are effective in ensuring network security while not compromising network performance. A number of tools for SDN network configuring and testing are also developed.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to express my deepest gratitude to my advisors, Dr. Gruenbacher and Dr. Scoglio. Thank you for your excellent guidance, encouragement, and patience over the years. Your advice has been helpful and influential, both to my research and my future career.

I would also like to thank my outside chariperson, Dr. Lei, and my committee members, Dr. Wu and Dr. Andresen. Thank you for the time spent being on my committee, and for all your brilliant comments on my work and dissertation.

Also, I want to express my gratitude to my friends in our NetSE group: Xin Li, Futing Fan, Qihui Yang, Heman Shakeri, Aram Vajdi, and all the others. You were helpful and patient when I turned to you for assistance. I would like to thank my other friends in our department as well: Tianyu Lin, Wenji Zhang, Bo Liu, and all others. Your inspiration and support are most appreciated.

Special thanks to my parents, Yue Wu and Zheng Qiu. Thank you for always being supportive. And thanks to my daughter Hannah — your shining eyes are my greatest motivation.

Finally, I want to express my special thanks to my wife, Xin. You have always been helpful, both on my research and in my life — always standing by me, through thick and thin. There are never enough words to express how grateful I am to you.

Dedication

For my beloved wife, Xin, and daughter Hannah.

Preface

This dissertation, "OpenFlow-enabled dynamic DMZ for local networks", is submitted for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering at Kansas State University. The research was performed under the supervision of Professors Don Gruenbacher and Caterina Scoglio.

To my best knowledge, this work is original except where acknowledgements and references are indicated. Part of the work has been presented, either published or submitted, in the following publications:

1. **Haotain Wu**, Xin Li, Caterina Scoglio, Don Gruenbacher, and Daniel Andresen. "Size-based flow management prototype for dynamic dmz." [1] In Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the, pp. 191-196. IEEE, 2015.

2. **Haotian Wu**, Xin Li, Caterina Scoglio, and Don Gruenbacher. "Middlebox resources management using OpenFlow." [2] In Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on, pp. 976-977. IEEE, 2016.

3. **Haotian Wu**, Xin Li, Caterina Scoglio, and Don Gruenbacher. "Security enhancement in real time using SDN." [3] Submitted to the Journal of Computer Networks, special issue: Security and Performance in SDN and NFV, 2017.

This research is supported by the Kansas State Electrical Power Affiliates Program (EPAP) and the Department of Electrical and Computer Engineering at Kansas State University.

Chapter 1

Introduction

1.1 Motivation

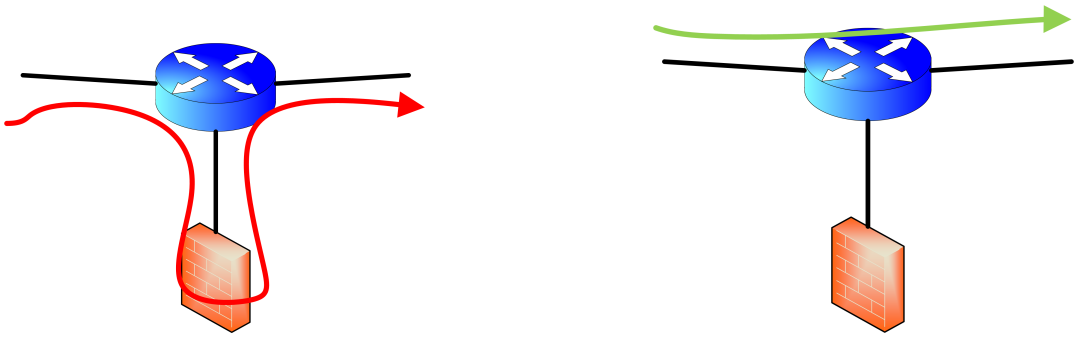
Security plays a vital role in today's network. Without proper attention to security issues, information may be leaked, altered, intercepted, or become inaccessible. On May 12, 2017, a ransomware cyberattack called WannaCry[4] was launched. The attacker encrypted the victim's files, and asked for a $300 ransom to decrypt them. This attack infected more than 200,000 computers in 150 countries. This would have been even more destructive if the network had been a cyber-physical network used in a smart city or on the Internet of Things[5]. For example, Koscher et al.[6] hacked into a smart car, which gave them full control of the instrument panel cluster. They were able to display arbitrary messages on car's LED, display false speedometer and fuel-level readings, adjust the illumination of instruments, etc. These can be very dangerous events when driving a car. Also, on Oct. 21, 2016, a successful DDoS attack[7] was launched toward Dyn's DNS servers by exploiting the vulnerability of IoT devices, which led to large-scale websites becoming unreachable including Twitter, Netflix, Reddit, etc. This was the largest DDoS attack in history; however, the majority of devices the attacker exploited were network printers, IP cameras, and baby monitors.

We can act upon many aspects to enhance network security. Examples are encrypting messages, keeping systems up to date, etc. We can also use security devices (e.g., deep

packet inspection, DPI) to analyze packet contents, filter unsecured flows, and prevent networks from being attacked. Currently, DPIs are widely deployed in most campus networks for security inspection. However, these DPIs don't solve our problem entirely. Ideally, security inspection should be applied on all traffic. However, user demand on networks is ever increasing[8], and it is unrealistic to expect that all traffic can be properly inspected. Many universities such as Yale, University of Michigan, UAB, and others have updated their core network to 100 Gbps. However, on the other hand, security devices in the network only provide a fixed amount of inspection capability, which cannot catch up with the increasing user demand, and the gap between them is getting even larger. Since DPI represents the bottleneck of the network, if we perform a full inspection on all traffic, network throughput will be brought down to the DPI's capacity, which is destructive for the network. But if all traffic is not inspected, network security is compromised. Therefore, we need to find a way to allocate the traffic these security resources. This is the problem we are trying to solve. Specifically, *in a local area network with network security enhanced by security devices, how best can the limited security resources be allocated to all network traffic, in order to achieve a tradeoff between network security and network performance?*

The demilitarized zone (DMZ) model is a good fit for such purposes. DMZ refers to a subnetwork or subset of network traffic where no security inspection is performed. This is justified as it is assumed that all traffic within the DMZ is trusted. The science DMZ model is already widely deployed in universities for performance consideration. For example, Yale University has a 100G network that deploys the DMZ[9]. The University of Alabama at Birmingham (UAB) also upgraded its campus network in 2016, installing a 100G edge with a science DMZ[10]. Inspired by the science DMZ prototype, we can allow a subset of flows to bypass DPI inspection for a short period of time when congestion occurs. This process is shown in Fig. 1.1. In normal cases, flows are inspected by DPI, as shown in Fig. 1.1a. When DPI resources are scarce, we allow some flows to bypass DPI inspection, as shown in Fig. 1.1b.

For now, DMZ rules are static. However, we need to configure them dynamically, i.e., implement a dynamic DMZ. This requires us to find a way to obtain traffic information in



(a) A flow inspected by the DPI.

(b) A flow bypassing DPI inspection.

Figure 1.1: In usual cases, flows are inspected by DPI, shown as Fig. 1.1a. When DPI resource is scarce, we allow some flows to bypass DPI inspection, shown as Fig. 1.1b.

real time, and reroute flows automatically and dynamically. Fortunately, software-defined networking (SDN) allows us to acquire flow statistics information in near real-time resolution and configure the network on the fly. SDN brings new features to a network: dynamic flow control, network-wide visibility with centralized control, network programmability, and simplified data plane — features proven to be beneficial in enhancing network security[11;12]. Moreover, SDN is capable of enabling dynamic service chains[13;14]. In our case, the service chains are either routing a flow through a DPI, or bypassing DPI inspections. OpenFlow[15], as an enabler of SDN, serves as the communication protocol between the SDN control plane and data plane.

1.2 Contributions

Our contributions toward this work are listed below.

1. Develop an OpenFlow application to balance network security and performance for networks with insufficient security resources, using a size-based criterion.

   We proposed and implemented a flow-management prototype with size-based criterion. In this prototype, we proposed to differentiate elephant flows and mice flows by a threshold. Since elephant flows cause congestion and are usually science data, we can allow the flows to bypass security inspection. We also performed theoretical calculations on the ingress data rate of the security device, which can guide in the selection

of the threshold value. This will be explained in depth in Chapter 4.

2. Develop an OpenFlow application to realize a dynamic DMZ in a general-topology network with multiple security devices, avoiding congestion on the security devices.

   We proposed and implemented a congestion-aware dynamic DMZ prototype. The scheme can be applied on a general local area network with distributed DPIs. In this scheme, limited security resources are allocated to the flows. We aimed at 1) maximizing utilization of security devices, therefore enhancing maximum network security; and 2) avoiding overwhelming the security devices in order to minimize packet loss at the network bottleneck. We proposed a capacity reservation scheme to accomplish this. In order to maximize the devices' utilization, we formulated the problem into an integer linear programming problem, and obtained a near-optimal solution for this problem. We will explain this scheme in detail in Chapter 5.

3. Implement a real-time flow-rate-obtaining module.

   For both prototypes mentioned above, the controller needs accurate flow rate information in order to optimize. However, obtaining flow rates in OpenFlow is nontrivial. In this dissertation, we elaborate on our work on the flow-rate-obtaining process in OpenFlow in Section 4.1 and 5.4.4.

Lastly, though we were focusing on the resource allocation on security devices, the work can be extended to any resource-limited middlebox.

1.3 Organization

The structure of this dissertation is outlined below. The fundamental concepts used in our network, including SDN, OpenFlow and middleboxes, are illustrated in Chapter 2. State-of-the-art research reviews are presented in Chapter 2 as well. Chapter 3 introduces the networking software tools used in this work: we explain how Open vSwitch, POX controller, and Mininet are used. A size-based flow-management prototype is discussed in Chapter 4, and the congestion-aware flow-management prototype is explained in 5. In fact, the work

shown in Chapter 5 can be seen as an extension of the work in Chapter 4. Finally, Chapter 6 concludes this dissertation and discusses possible future work and challenges.

Chapter 2

Background

In the previous chapter, we defined the problem we want to solve and explained SDN is the best solution for this problem. In this chapter, we will introduce the basic concepts used in our network prototype. We will give a brief introduction to SDN and middleboxes; explain what OpenFlow is, how it works, and why we use it; and present our DMZ model. At the end of this chapter, we will present the current state-of-the-art work that is closely related to this topic.

2.1 Software-defined networking

Conventional networks are still constrained after decades of development. Limitations of conventional network are as follows:

1. Flexibility. Too many complex functions are added to the network devices, e.g. OSPF, NAT, Multicast, BGP, etc. Moreover, new protocols keep emerging at an ever-increasing speed. The complexity is already a huge burden on conventional networks, which have difficulty adapting.

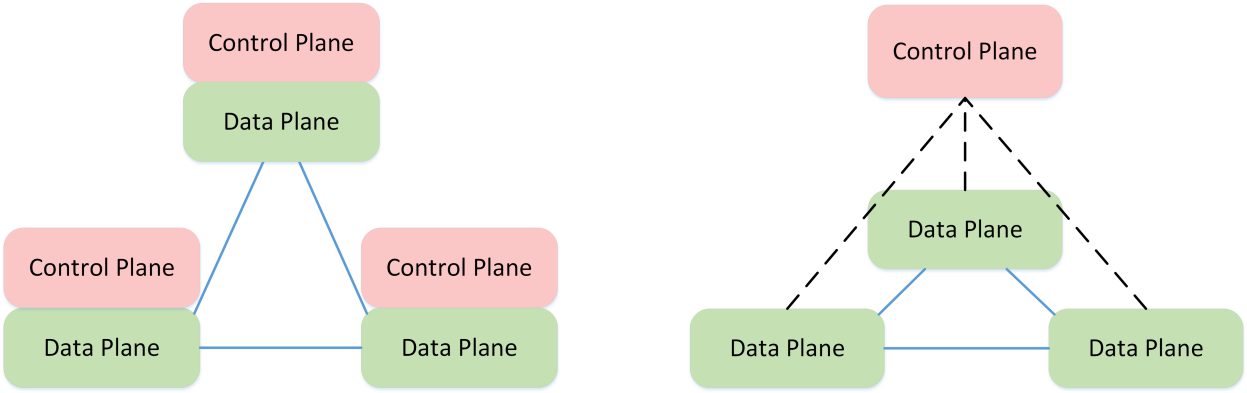
2. Global optimization. Devices in conventional networks either don't have a global view of the entire network, or it takes too long for the devices to obtain a global view. To make things even harder, the devices have to start updating again when a topology

change occurs, and the converging process can take a long time as well. Therefore, devices in a conventional network are not able to obtain the real-time global view of the network, let alone perform global optimization to improve network performance. However, global optimization is critical, since high bandwidth utilization is vital in data centers and enterprise networks.

3. Customization. We have the needs to configure network devices dynamically in today's environment. Conventional network devices mostly use vendor-specific configuration commands; moreover, many configuration changes are done manually on each device. This is neither portable nor scalable.

Software-Defined Networking (SDN) is introduced to address these limitations. The main contribution of SDN is its ability to decouple the network data plane and control plane, as shown in Fig. 2.1b. On the contrary, the control plane and data plane are combined in conventional networks as depicted in Fig. 2.1a. The control plane is where all controlling decisions are made. In an SDN network, the centralized SDN controller is located in the control plane. On the other hand, the data plane does the work of transmitting data from one place to another. The data plane doesn't make decisions; when unsure what to do about a packet, the data plane will ask the control plane instead. The most well-known protocol for controller-switch communication is OpenFlow (introduced in Section 2.2), but there are other protocols/standards in SDN as well. For example, OpenConfig is trying to use a more accurate, descriptive way to configure SDN devices in networks, including both switches and wireless APs. OpenConfig is aiming at allowing convenient and dynamic network device configuration. A number of vendors are already starting to support OpenConfig.

By moving all control logic into the controller, SDN allows the feasibility of configuring the switches when required. SDN brings the following new features to a network: dynamic flow control, network-wide visibility with centralized control, network programmability, and a simplified data plane — features proven beneficial for enhancing network security[11;12]. Moreover, SDN is capable of enabling dynamic service chains[13;14]. In our network prototypes, the service chains are either routing a flow through a DPI, or bypassing the DPI.



(a) Conventional network architecture.

(b) SDN architecture.

Figure 2.1: Architecture comparison between conventional network and SDN.

The SDN architecture does have its own limitations. First, the centralized controller becomes the single point of failure. If the controller is unreachable, the entire network is down. For the same reason, the controller is very likely to become the target of attackers. Second, SDN network is difficult to scale up further when it reaches a ceiling. This is limited by the processing capability of the controller. Third, SDN network is error-prone. It's likely the controller code contains bugs, which will result in the network behaving unexpectedly. However, SDN still brings more benefits than drawbacks, because of its high flexibility and programmability.

2.2 OpenFlow

OpenFlow[15] is an open standard communication protocol in the SDN environment. It realizes all concepts brought up by SDN. Therefore, OpenFlow is the enabler of SDN. OpenFlow protocol is used for communication between the SDN controller and OpenFlow-enabled switches.

2.2.1 Basic architecture of OpenFlow

Fig. 2.2 shows the basic architecture of an OpenFlow enabled network. The infrastructure layer (or data layer) is where all switches reside. It talks to the control layer with OpenFlow

protocol, so the control layer can store network status. The application layer communicates with the control layer with northbound API. In this way, the applications can have access to the current state of the network, and furthermore, perform complex tasks. For clarification, though this OpenFlow architecture looks like the OSI 5-layer model, they are not related. The application layer in OSI is the place where applications can write in their data; however, the application layer in OpenFlow is where the control logic resides.

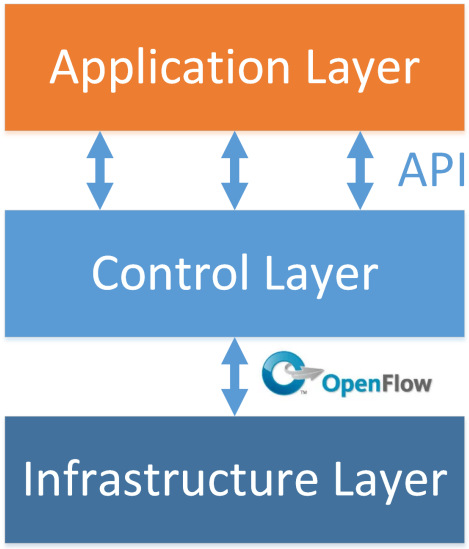


Figure 2.2: Basic architecture of an OpenFlow-enabled network.

2.2.2 Flow entry

In OpenFlow networks, a switch keeps track of a flow table, which contains multiple flow entries. The structure of a flow entry is shown in Fig. 2.3, consisting of three parts: headers, action, and statistics. In short, any packet whose header matches the header field of this flow entry should perform all actions in the action field, and then the switch will update the counters in the statistics field.

Specifically, the header contains 12 matching fields. The fields are retrieved from the packet. It includes source and destination MAC address, IP address, port number, and protocol type, physical interface, etc. They range from layer 2 (data link layer) to layer 4

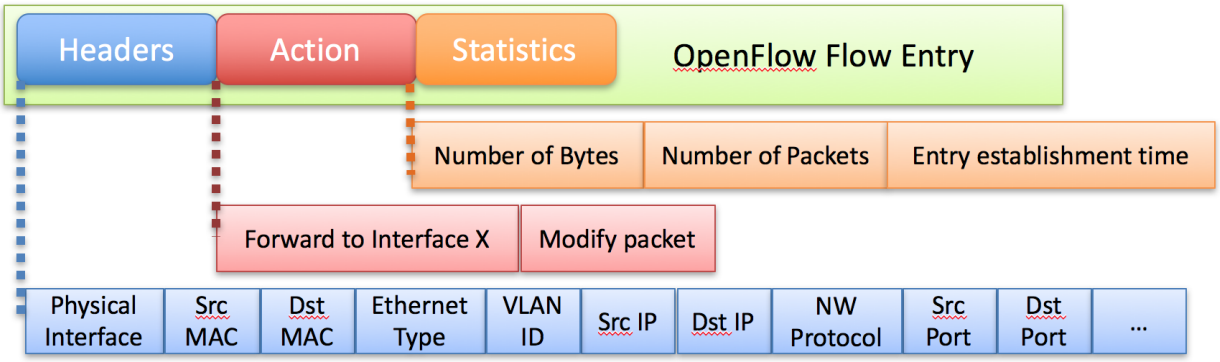


Figure 2.3: Structure of OpenFlow flow entry.

(transport layer), and OpenFlow is capable of matching all the fields simultaneously. Therefore, an OpenFlow switch can be seen as a layer 2/3/4 switch.

A flow is defined as the set of all packets which share the same packet-matching fields. However, this does not necessarily mean all packets in a flow share the same 12-field header. We can set a field as wildcard in a flow entry, so that the field is ignored in header matching, whatever its value is. We can either use all 12 fields, whereby one single difference in the packet header will be recognized as a different flow, or set some fields as wildcards.

Setting the wildcard rule is important when we need to combine or distinguish traffic between applications. For example, consider two packets both sourced from 10.0.0.1 and destined to 10.0.0.2. Both packets have the same destination port — 80, while their source ports are 5100 and 5200, respectively. If no field is set as a wildcard, then these two packets belong to different flows, since their source port numbers are different. However, if we set the source port number field to be a wildcard, these two packets are now in the same flow. We care which flow a packet belongs to because the minimal unit in OpenFlow is a flow, not a packet. A flow entry performs the same actions on every packet that belongs to this flow.

The next field in OpenFlow flow entry is actions. Typically, an action is simply forwarding the packet to a specific interface. However, the action can be more complex, as in examples of broadcasting, dropping, modifying a packet, or even several actions mentioned above combined. For security purposes, a flow is dropped by default, unless we explicitly specify its outgoing interface.

Lastly, the statistics field keeps track of several counters of the flow, including number of bytes and of packets in the flow, duration of the flow, etc. Though this information does not directly interfere with the packet's forwarding process, it is still important because the controller can learn the speed of all traffic in the network from these data, and further reroute some flows for optimizing the usage of network resources.

The switch doesn't proactively send the stats to the controller. Instead, the controller will first send a flow-statistic request to the switch, then the switch will answer with a flow-statistic response packet containing the stats of all active flow entries installed on the switch.

Apart from the three fields in a flow entry, two other properties are important to a flow entry: priority and timeout. The priority is an integer ranging from 1 to 65535. Because of the existence of wildcards, it's possible that a packet is able to match multiple flow entries. In this case, the flow entry with the highest priority is matched. A flow entry with no wildcard rule (exact match) always has the highest priority (priority 65535).

The timeout field consists of a hard timeout and an idle timeout. When the flow entry is installed, we must set the values of hard timeout and idle timeout. These values specify when the flow entry should expire. For example, we can set the hard timeout as 30 seconds and the idle timeout as 10 seconds. This means if the switch hasn't received a packet matching this flow entry for 10 seconds, the flow entry reaches its idle timeout time and therefore, is removed from the switch. Also, as soon as the flow entry is 30 seconds old, it reaches its hard timeout and is removed from the switch. The timeout mechanism helps keep the switch in a clear state; otherwise, switch flow tables will overflow with obsolete flow entries. For flow entries we want to keep on the switch indefinitely, we can set both hard timeout and idle timeout as permanent.

2.2.3 Packet forwarding process in OpenFlow

Fig. 2.4 shows the basic process of a packet being forwarded with OpenFlow. When a packet arrives at an OpenFlow switch, the switch will first look up its flow table, trying to find a

matching flow entry, which will indicate the action that should be applied on this packet. But if the switch failed to find a matching flow entry, it will pack this packet together with the OpenFlow header (forms an OpenFlow packet-in type packet, OFPT_PACKET_IN), and then send them to the controller, in order to query where to send this packet. The controller will make this decision, and send the action that should be applied on this packet back to the switch. Finally, the switch applies the action to this packet, and saves the packet header and its corresponding action in its flow table (i.e. install flow entry) for serving upcoming packets in this flow. Though this entire process may take long (tens or a hundred milliseconds), it only applies to the first packet of a flow.

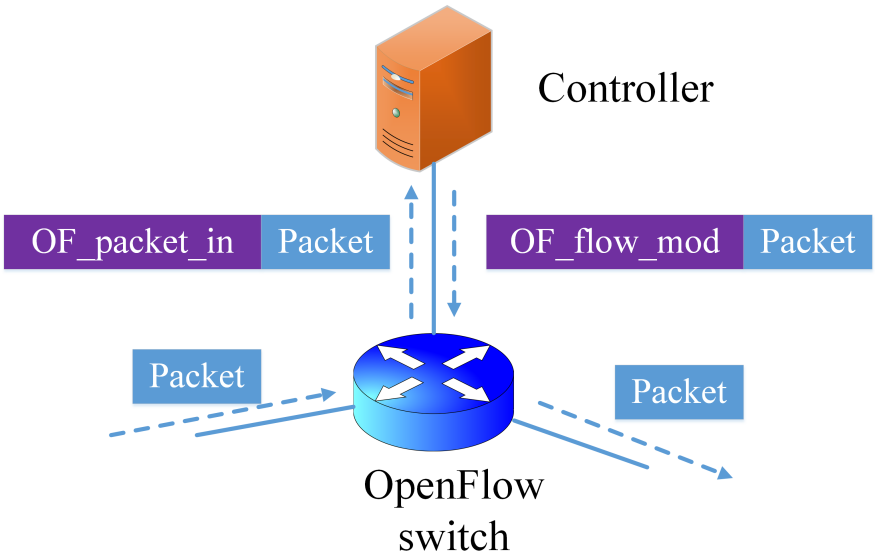


Figure 2.4: Basic architecture of OpenFlow.

In the procedure above, we use a flow-modification type packet (OFPT_FLOW_MOD) to install a new flow entry on the switch. The installed flow entry can help speed up the forwarding process when new packets of this flow arrive. Alternatively, in some cases we just want to forward this packet, but not install any new flow entries. For example, when the controller receives a packet without registered forwarding information, we may want to broadcast this packet. In this case, a packet-out type packet (OFPT_PACKET_OUT) can be used. When the switch receives a packet-out type packet, it applies the actions on the

packet just as a flow-modification packet. The only difference is that it doesn't install the flow entry on the switch.

2.2.4 Advantages of using OpenFlow

There are numerous benefits in using OpenFlow on the network.

- OpenFlow makes centralized control easier. In a conventional network, since the control layer is distributed across every single networking device, no device can obtain a global view of the entire network, let alone perform global optimization. However, in an OpenFlow network, a centralized controller is deployed. The controller is capable of obtaining information on all switches and all flows, and issuing commands to the switches. Therefore, centralized control is made much easier. Centralized control will make discover-based protocols run much faster, e.g., spanning tree protocol and OSPF protocol. It can also help keep the control logic consistent on all switches.

- OpenFlow is highly programmable and flexible. In a conventional network, there's no way we can program packet-forwarding logic. For example, in a network, we desire all flows from H_1 to H_2 to be routed through route A for only the first two seconds, and then reroute them to route B. We may find it very hard to implement these regulations in a conventional network. However, in an OpenFlow network, since the controller is a software running on a server, we can easily program the control logic.

- OpenFlow can help modify the packet according to our rules. One possible application, with the help of this feature, is that an OpenFlow switch can serve a NAT device, because we can easily modify the IP address and port numbers of a packet. This feature is also highly critical from a security perspective. For example, as shown in Fig. 2.5, the controller realized H_1 is an attacker that is trying to send malicious traffic to the server S. Instead of simply dropping H_1's traffic, we can make H_1 talk to a "fake" server, which is in fact the network monitor device M, in order to gather more information about the attacker. The monitor should talk to H_1 as if it was the attacked

server. This can be easily achieved by installing the flow entries listed in Table. 2.1 to the switch.

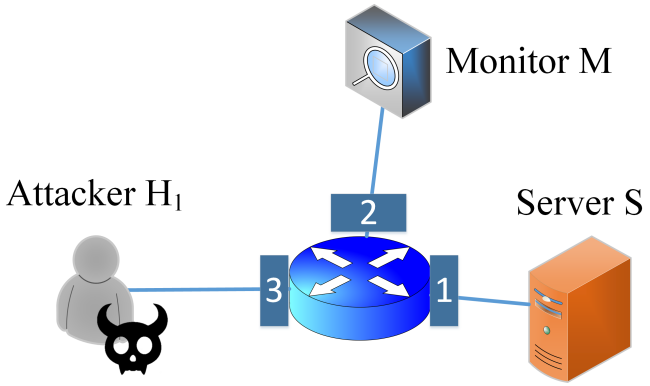


Figure 2.5: An example topology — H_1 is launching an attack to server S.

Table 2.1: Flow entries to be installed on the switch.

Match field		Action field
Source	Destination	
H_1	S	Forward to interface 2
M	H_1	Change source IP to S, forward to interface 3

- OpenFlow standardizes the switch configuration API. This is very important when the devices in a network are from multiple vendors. To configure the switches in conventional networks, usually means doing so manually one by one. Moreover, the fact that each vendor has its own set of CLI (command line interface) commands makes the configuration process much more complicated. OpenFlow solves this problem by establishing and using a vendor-neutral standard, so the configuration commands will be identical no matter what vendor the device is from. This also helps automate the configuration process, which makes the configuration of multiple switches much easier.

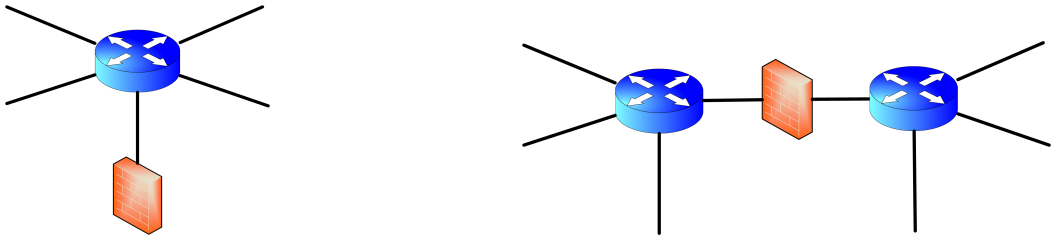
- OpenFlow is completely transparent to end-users and even legacy switches. OpenFlow is a protocol only used between the controller and switch, so end-users can talk normally without awareness of OpenFlow protocol in use. For the same reason, we can also run

OpenFlow in a hybrid mode, i.e., place OpenFlow switches in the same network with legacy switches, as long as the OpenFlow switches are connected to a controller.

2.3 Middlebox and deep packet inspection

A middlebox (MB), as suggested by the name, is a device that lies in the middle of network traffic and performs actions on the traffic that goes through it. Formally, a middlebox is a networking device which inspects and manipulates traffic[16;17]. Traditional packet-forwarding devices, e.g., switches and routers, are not considered middleboxes.

Middleboxes can be deployed either off-path or on-path. An off-path middlebox deployment means the middlebox is connected, and only connected to one switch, as shown in Fig. 2.6a. On the contrary, on-path deployment means the middlebox is connected to two switches, thus lying in the middle of a path, as shown in Fig. 2.6b.



(a) Off-path Middlebox Deployment. (b) On-path Middlebox Deployment.

Figure 2.6: Comparison between off-path middlebox and on-path middlebox.

An on-path middlebox has the advantage of easy deployment; however, because the middlebox lies in the middle of a path, this will result in every flow that needs to go through this link being processed by this middlebox. However, this flow might already have been processed by another middlebox. On-path deployment also lowers the link bandwidth to the middlebox's capacity. Therefore, on-path deployment is often a waste of resources. Conversely, off-path middlebox placement enables efficient use of middleboxes[18].

Among security devices deployed in the campus network, deep packet inspection (DPI) device is the most commonly used network packet-filtering middlebox. Unlike a firewall, which only inspects the header fields of a packet, DPI inspects the traffic content to determine

whether a packet is malicious. DPI can be used by companies to block content from a certain websites or applications as well, for example, Twitter. Currently, DPI is widely used in networks for security inspection, spam filtering, etc. In our network prototype, we use off-path deployed DPI to identify malicious packets.

DPI has the following drawbacks as well: using DPIs brings down network performance. Security inspection by DPI is expensive. Security devices such as firewalls only try to match the packet header with some pre-defined rules; however, DPI analyzes the content of a packet and evaluates it for anomaly classification. This is the reason why the DPI processing rate is slow when compared with network bandwidth. If everything is forwarded to DPI for security inspection, the throughput of the entire network will be limited to the DPI's processing capability, therefore bringing down the network performance. This is the problem we are trying to solve.

2.4 DMZ model

As introduced in Section 1.1, DMZ is a conceptual network design where no security inspection is performed. DMZ was first introduced to isolate servers exposed to the public from internal workstations, as shown in Fig. 2.7.

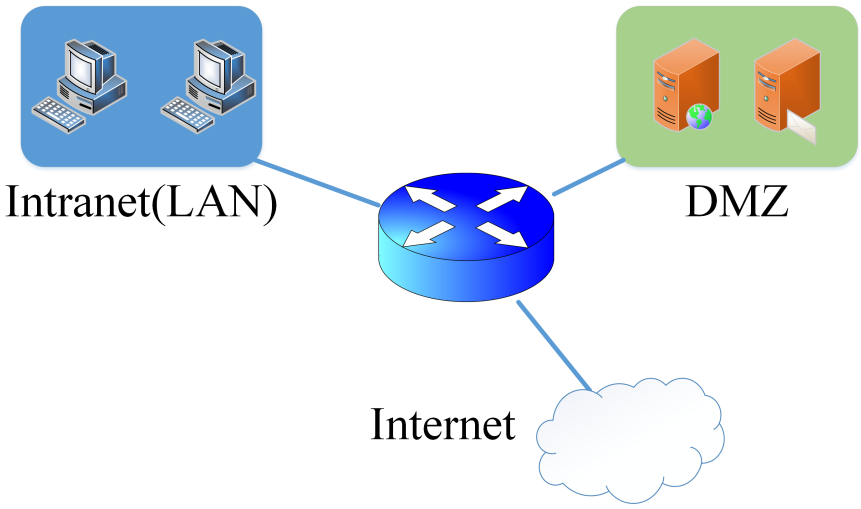


Figure 2.7: DMZ isolates internal workstations from exposed servers.

In a campus network, science DMZ model has been introduced to differentiate science-data networks from general-purpose networks. We call it science DMZ because we don't perform any security inspection on science data, just as with the traffic to the servers exposed to the public in the DMZ model. Science DMZ can allow trusted flows to bypass security inspection and improve network performance[19].

We can set pre-defined rules in the network to realize science DMZ; however, it will only apply to the network where we have had a whitelist beforehand. In case we have no pre-knowledge about the network, we will have to propose our own DMZ rules. We will propose two different DMZ rules and illustrate how we can enhance network security with these rules in Chapter 4 and Chapter 5, respectively.

2.5 Literature review

We are trying to balance network performance and network security. Therefore, we will present the literature review on both the network performance aspect as well as the network security aspect.

2.5.1 Performance

On the performance side, this problem can be seen as a traffic engineering problem. Four aspects are most critical to a traffic engineering problem: resource allocation (flow management), fault tolerance, topology update, and traffic analysis[20]. In our work, we focus on resource allocation and traffic analysis.

Resource allocation

Resource competing may occur in many places — for example, the link[21], switch[22;23], controller[24;25], and middleboxes[26]. Since in our network prototype, middleboxes are the bottleneck, we will focus on the resource allocation problem on middleboxes.

When allocating middlebox resources, conventional works have focused on admission

control, optimizing routes for given traffic demands, or traffic sampling to avoid overwhelming the DPI device. Admission control[27] avoids congestion on links or middleboxes by checking if current resources are sufficient before connection establishment. Therefore, when the "resources" are link capacities in bps, it requires us to know the data rate of a flow as soon as its first packet arrives. The optimization approach[28;29] performs an offline optimization before the network starts; therefore, it requires pre-knowledge of exact traffic demands of the entire network. In a general campus network, we can neither obtain a flow's data rate as soon as it arrives, nor have a pre-knowledge of all traffic demands. Traffic sampling[30;31] is a good approach in general; however, it is not suitable in a network secured by DPI. At the application level or content level of DPI inspections, the inspection of contiguous packets is important because this process can help a DPI "understand" the content of the flow, and further increase accuracy of the inspection; however, packet sampling breaks down the flow.

Traffic analysis

In order to perform optimization, we need accurate and real-time traffic statistics. In a conventional network, we can use MIB/SNMP[32] or traffic sampling[33;34] to obtain real-time traffic statistics. SNMP is the most widely used protocol for monitoring network status. However, SNMP is complex, unscalable, and hard to implement. Traffic-sampling approaches such as sFlow have proven to work well together with OpenFlow[31]; however, they only offer an estimation of traffic rate instead of an accurate one. Though there are other excellent works on flow statistics monitoring, e.g. OpenNetMon[35], which provides flow statistics in a more accurate way, they will make our network prototype unnecessarily complex. We found the best way to obtain traffic statistics in an OpenFlow network is by using OpenFlow's built-in counter fields in flow entries.

Traffic Engineering with SDN

SDN makes it easier to implement traffic engineering strategies such as load balancing and algorithms for flows routing[20;21;36;37]. Paper[21] studies a traffic engineering solution for a

network where SDN is only partially implemented, i.e. SDN switches were incrementally introduced into an existing network. This means only a few nodes are controllable by the controller. They formulated this scenario as a linear programming problem and then solved it, which in turn improved link bandwidth utilization, and reduced delay and packet loss. Zhang et al. presented a hybrid routing approach, which combined destination-based routing with explicit routing, in order to achieve load balancing with low complexity and good scalability[36].

No matter what TE technique we are using, the controller's ability of rapidly retrieving real-time monitoring information is a must, using the less overheads the better[22;38]. Curtis et al. developed DevoFlow[22], an improved OpenFlow model, to reduce the cost of unnecessary overheads and achieve high-performance networks. Rasley et al. presented a network measurement architecture "Planck"[38], which is able to extract network information in hundreds of microseconds to milliseconds. Fioreze et al.[39] analyzed the reliability of network information collected from NetFlow.

2.5.2 Security

On the security side, many works focused on the science DMZ/firewall bypassing prototype. Calyam et al. presented a campus science DMZ reference architecture in[19] and demonstrated how the DMZ model could simultaneously achieve network security and high throughput by allowing specific users to bypass security devices. However, their architecture requires pre-knowledge of trusted users.

Balas et al. proposed a firewall bypassing prototype called Scipass[40]. Scipass uses both firewall and load-balanced IDSes as security devices, because a single IDS can no longer support the entire network. This is a common approach that uses distributed devices for load balancing. Scipass performs bypassing according to a whitelist. An entry on the whitelist can be, for example, a trusted sending pattern. However, the main drawback of this approach is that it requires pre-knowledge of characteristics of secure flows. Also, a whitelisted flow will never be sent back for inspection again, even if there are sufficient resources.

Miteff et al. introduced a new science DMZ design called NFShunt[41]. NFShunt proposes a hybrid design using a science DMZ design and a Netfilter as a stateful firewall to enhance security of a science DMZ. To shunt a flow, it is routed from the slow path (security inspection path) to the fast path (bypassing path), and an idle/hard timeout timer is set on the fast path route. When the timer expires, the flow is routed back to the slow path.

Therefore, though these studies did good work on addressing the issue of security resource shortage, the solutions still have the following limitations:

1. The solutions relied on pre-knowledge of the flow/user whitelist.
   First, obtaining the entire whitelist can be difficult, and the list also changes frequently. Second, considering the possibility that a whitelisted host can also be compromised, no host or flow can be considered 100 percent safe. In our prototype, we assume we have no pre-knowledge about the network.

2. The solutions didn't take flow size and security device capacity into consideration.
   Security device capacity is, instead, critical in our DMZ decision making. First, a large flow is more likely to be a science data flow, so it is more likely to be rerouted, bypassing DPI inspection. Second, we should be aware of not overwhelming the security devices.

3. The solutions assumed a single-switch, single-security device topology.
   However, we need to provide a security inspection for a general-topology network as well. Distributed security devices are also widely deployed in today's networks.

Therefore, we proposed two DMZ prototypes in Chapter 4 and Chapter 5, respectively, aiming at solving the three issues listed above. Limitations 1 and 2 are resolved in the first prototype. In addition, our second prototype solved all three limitations.

Chapter 3

Networking tools

Before we introduce our work in the following two chapters, we will first briefly introduce the software tools used or altered in this work: Open vSwitch, POX controller, and Mininet. We used Open vSwitch to emulate a software switch for experiments in the work in Chapter 4, and used Mininet to emulate a network for performance testing in the work in Chapter 5.

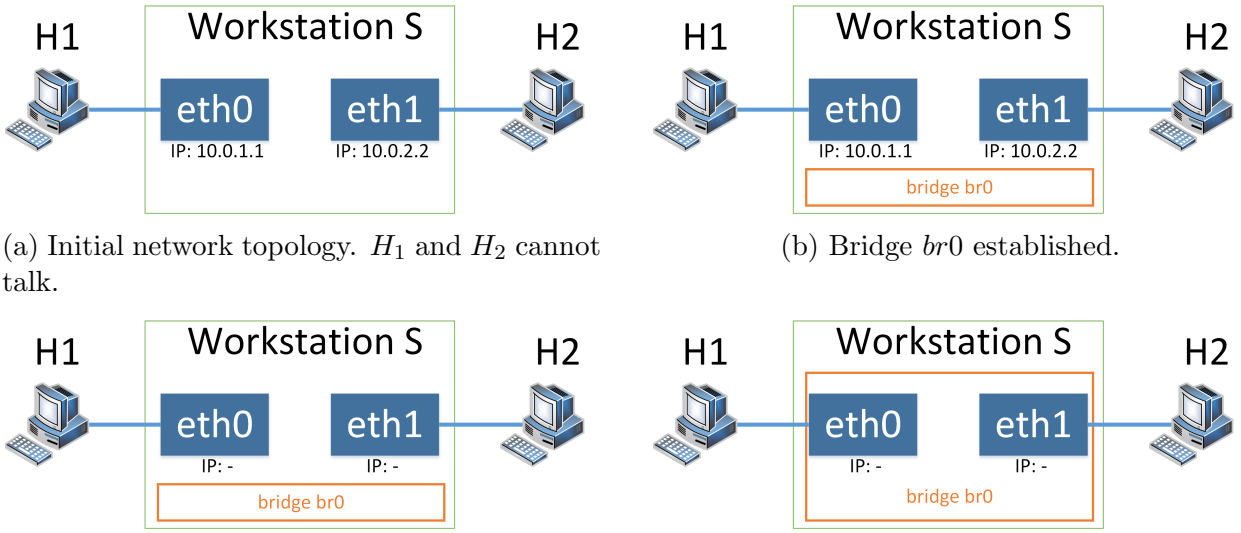
3.1 Open vSwitch

Open vSwitch (OVS)[42;43] is an open-source virtual switch software, that can run on any Linux machine, turning the machine into a virtual switch. OVS gives very good OpenFlow protocol support: for now, OpenFlow 1.0 - 1.2 is fully supported by OVS, and OpenFlow 1.3 - 1.5 is partially supported.

The process whereby OVS turns a workstation into a virtual switch is shown below. Consider a topology where two hosts (H_1, H_2) are both connected to workstation S, as shown in Fig. 3.1a, and our objective is to make the workstation a virtual switch so the two hosts can communicate.

1. First, install OVS on the Linux workstation. Type this one-line command in the terminal:

   ```
   apt-get install openvswitch-switch
   ```



(a) Initial network topology. H_1 and H_2 cannot talk.

(b) Bridge $br0$ established.

(c) IP addresses of the interfaces are removed.

(d) Interfaces are added into bridge $br0$. Now workstation S is acting as a switch.

Figure 3.1: The process of a bridge is established.

OVS should be installed automatically. If the workstation already has Mininet installed, this step can be skipped, since OVS is contained in the Mininet installation package.

2. Then create a bridge on the workstation.

   ```
   ovs-vsctl add-br br0
   ```

   This command creates a bridge named $br0$ on workstation S, as shown in Fig. 3.1b. A bridge works on the data link layer (layer 2) and is capable of connecting separated networks. We will use this bridge to allow H_1 and H_2 to talk to each other.

3. Next clear the IP address settings on both interfaces, as shown in Fig. 3.1c:

   ```
   ifconfig eth0 0
   ifconfig eth1 0
   ```

   Since a switch's interfaces shouldn't have IP addresses, we clear the IP address settings on both interfaces.

4. Finally, place the interfaces into the bridge.

   ```
   ovs-vsctl add-port br0 eth0
   ```

`ovs-vsctl add-port br0 eth1`

Adding interfaces into the bridge is similar to plugging Ethernet cables into a switch. Now the two interfaces, $eth0$ and $eth1$, are no longer isolated, but are connected by the bridge. For now, the workstation is already acting as a switch, where the two interfaces are $eth0$ and $eth1$. Fig. 3.1d shows the current state of this step.

The workstation has already been configured as a virtual switch; however, the hosts may still not talk to each other. This is because OVS emulates a virtual OpenFlow switch, which can only work together with an OpenFlow controller. To connect the virtual switch to a controller, we should first start a controller either locally or remotely, then:

```
ovs-vsctl set-controller br0 tcp:10.1.2.3:6633
```

where 10.1.2.3 is the IP address of the machine the controller is running on. The controller is running on port number 6633 by default.

Finally, we set the controller failure mode to "secure."

```
ovs-vsctl set-fail-mode br0 secure
```

The failure mode can be either "secure" or "standalone." This configuration takes effect when the switch is not able to talk to the controller. In standalone mode, the switch itself will take over the responsibility and act as a layer 2 learning switch. Instead, in secure mode, the switch will drop all received packets. In our network prototype, we are more concerned about network security than network availability. Therefore, we set the failure mode to "secure."

3.2 POX controller

As introduced in Section 2.2, an OpenFlow-enabled switch has to be connected to an OpenFlow controller to start working. There are many choices for OpenFlow controllers: OpenDaylight[44], Beacon[45], POX[46], etc. Among these, we use the POX controller to control our OVS switches. POX is a python-based, light-weight OpenFlow controller. Compared with other "powerful" controllers, POX provides just enough functionalities to program a con-

troller. The simplicity of POX's architecture and the flexibility of Python language allows us to only focus on the real control logic of the network.

The POX controller provides a layer 2 learning switch code as a starting point (file `l2_learning.py`). The basic diagram of `l2_learning.py` is shown in Fig. 3.2. We can develop our own control logic by simply modifying this code. For example, in our network prototype, we need to query the switches every two seconds. We can either start a query thread on every single switch-connection-handling instance, and send statistic requests every two seconds; or start only one query thread on the main process, to send statistic requests to all switches every two seconds. It is often the case a task can be accomplished in two or even more ways, thanks to the programmability of the controller and the flexibility in SDN architecture.

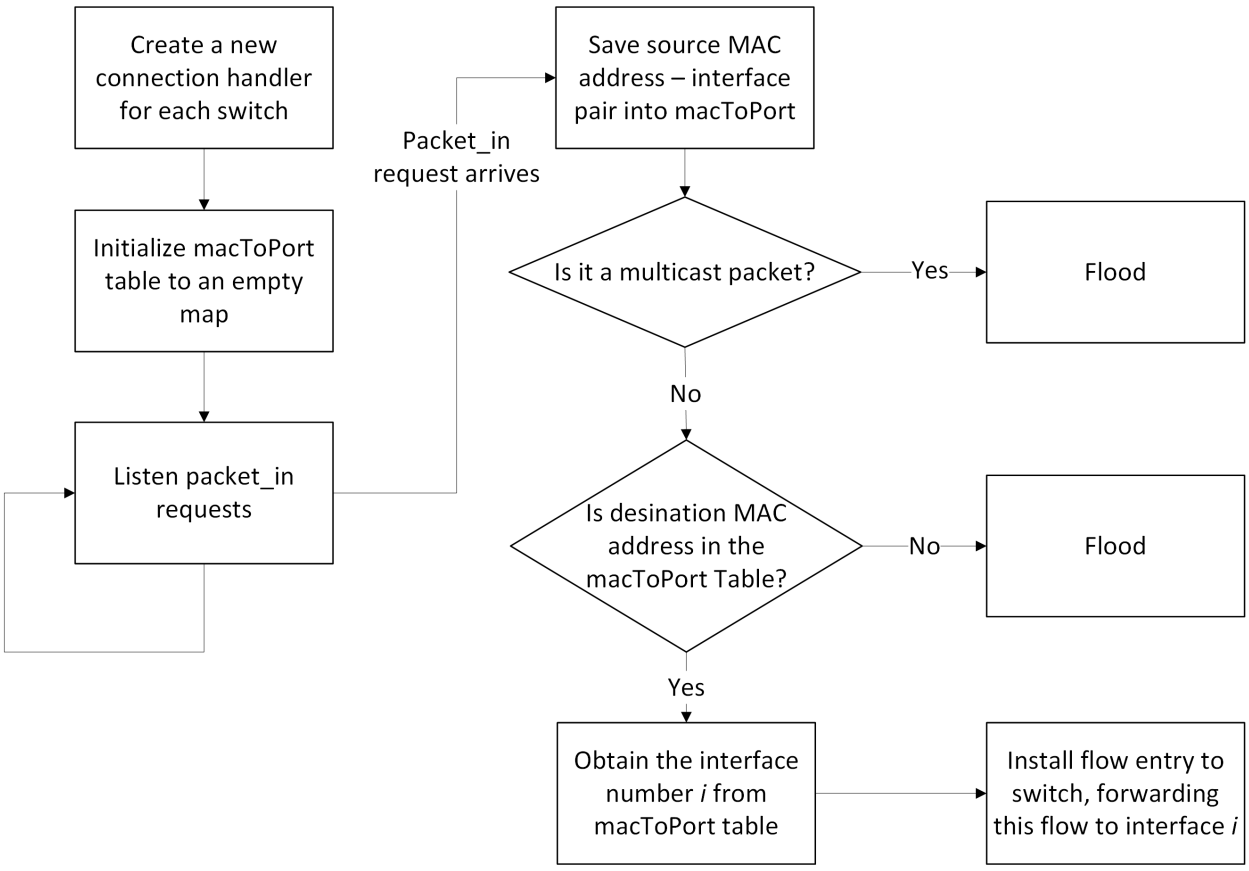
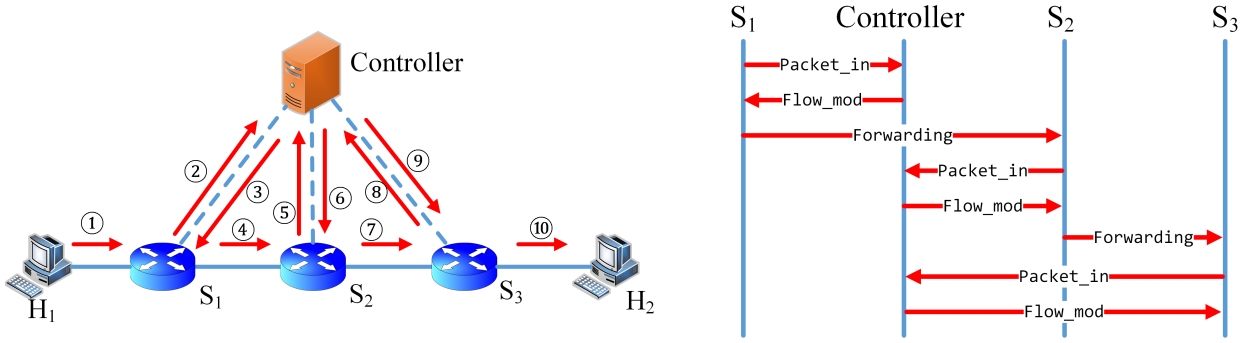


Figure 3.2: POX controller's layer 2 learning switch code diagram.

Though the control logic in file `l2_learning.py` looks simple and straightforward, it can

cause potential problems when multiple switches are in the network. Let's take the network topology shown in Fig. 3.3a as an example. The topology is a linear topology with three switches. Now assume Host H_1 establishes a flow with Host H_2. The red arrows show the entire route of the first packet in this flow, and the circled number next to the arrow is the sequence number. The detailed packet forwarding process, including how the switches talk to the controller, is depicted in Fig. 3.3b.



(a) The sequence that the first packet of a flow travels from source to destination.

(b) The detailed packet forwarding process.

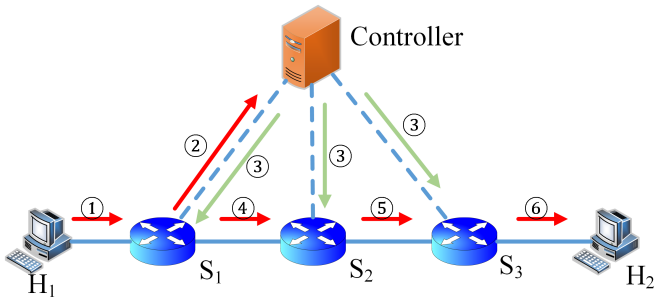
Figure 3.3: The process that the first packet of a flow is forwarded in a linear topology network using simple l2-learning POX controller.

This forwarding process, simple as it is, is not recommended for at least two reasons:

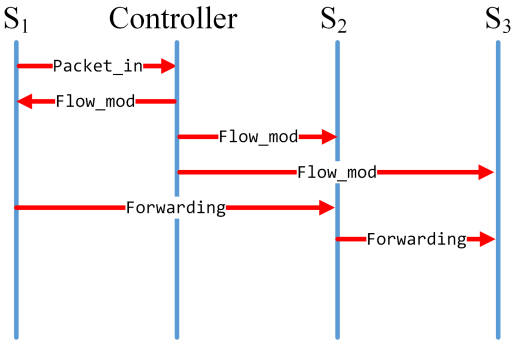
- The packet delay is too long for the first few packets. This can be easily seen in Fig. 3.3. The round trip between switch and controller is the most time-consuming step in the entire forwarding process; however, we need to multiply this time by n if we have n switches in a row. As a result, the delay of the first packet will be huge, and this can even cause problems if the packet is delay-sensitive.

- Resources are wasted in the controller. When the packet reaches the controller for the first time, i.e., when switch S_1 sends the packet to the controller, the controller is already aware of this flow and has calculated the entire path from source to destination. However, we made the controller recomputed the paths for a second and third time because S_2 and S_3 would send the exact same packet to the controller.

To solve this problem, we can slightly modify the routing process: when the controller notices the flow for the first time, it computes the entire source-destination path, and installs flow entries on all switches on the way. The packet's forwarding process details are shown in Fig. 3.4.



(a) The sequence that the first packet of a flow travels from source to destination. Note the flow entry installation are performed simultaneously in step ③.



(b) The detailed packet forwarding process.

Figure 3.4: The process that the first packet of a flow is forwarded in a linear topology network using a revised forwarding approach.

However, this approach brings a new problem while solving the long-delay issue. The problem is, though we expect all switches to install the flow entry simultaneously, a slight difference exists between their flow entry installation times, and this can cause a big problem. Assume switch S_1 installed the flow entry first, so the packet is forwarded to switch S_2. However, by the time the packet reaches S_2, S_2 has not yet finished installing the flow entry, so it will query the controller again about this packet. However, the controller's logic is that it will receive packet_in requests from the first switch on the flow's path only, thus this will cause unexpected behavior from the controller. Sending the flow entry installation request in sequence $S_3 \rightarrow S_2 \rightarrow S_1$ doesn't help either. Since the flow_mod requests can be sent out instantly, but processing the request has a large variance in time, the switches may still install the flow entries in any order and this is totally unpredictable. We have to find a way to send the flow_mod request to S_1 only when all other switches are done installing their flow entries. Fortunately, in OpenFlow, a feature called barrier is designated for this.

There are barrier_request and barrier_response in the barrier module. When the switch

processes a barrier_request packet, it will reply to the controller with a barrier_response packet. If we first send the switch a flow_mod packet and then a barrier_request, then we know the flow_mod request is processed for sure when the controller receives the barrier_response back from the switch. Therefore, we can let the controller send a barrier_request to all switches on the flow's path except S_1; and only when we get all barrier_responses, do we then install the flow entry on S_1. The entire process is shown in Fig. 3.5.

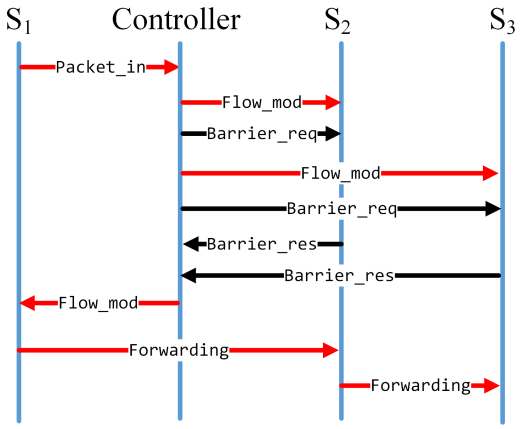


Figure 3.5: The process that the first packet of a flow is forwarded in a linear topology network using barrier.

Another issue occurs in this forwarding process when we send a UDP flow in the network. When sending a UDP flow, the host sends packets out at a very fast rate. When these packets reach the first switch, the first packet of this flow is sent to the controller to query for the actions. However, querying for actions takes time (around 100 milliseconds), and in the meantime, new packets of this flow arrive at the switch as well. Ideally a table should be maintained in the switch, recording which flow has already been sent to the switch for action querying, so that we can just send the first packet of a flow to the controller and buffer the rest in the switch. This is not the case in reality. In fact, all subsequent packets are forwarded to the controller as well until the flow_mod packet arrives. If we don't do anything about this, the controller will process the same request again and again. This will be a great waste of precious controller resources.

We can solve this problem in the controller. We can keep track of a table, with the

packet's signature (tuple: source_IP, destination_IP, source_port, destination_port) as the key, and the content of this packet as the value. When a new packet_in request comes in, we first check to see if the signature exists in the table. If so, we don't process this packet_in, but we still buffer the packet content to avoid losing this packet; if not, we insert this signature into the table and process the packet_in. Algorithm 1 presents this process in detail as follows:

Algorithm 1 Avoid duplicate packet_in

1: $signature \leftarrow$ tuple $(source\_IP, destination\_IP, source\_port, destination\_port)$
2: **if** $signature$ exists in $table$ **then**
3: Push current packet into $buffer[signature]$
4: **else**
5: Insert $signature$ into $table$
6: Process this packet_in
7: Send all packets in $buffer[signature]$ back to the switch

3.3 Mininet

Mininet[47] is a network emulator and testbed that creates and emulates a network of virtual hosts, switches, controllers, and links. By using Mininet, we can easily emulate a network with complex topology and send control commands to the hosts. In Mininet, all hosts are running in a VM-like environment: each host has its own IP address. All switches are emulated using OVS.

Advantages of using Mininet for networking experiments are numerous. First, it eliminates the restriction on the amount of hardware switches; we can experiment on a 10-node network, even if we don't have a single switch. All experiments can be carried out with one workstation. Second, the experiment topology is highly scalable. We are able to test on a topology with different numbers of switches by only changing a parameter. Last but not least, using Mininet makes the testing process highly automated. In an experiment with a conventional network, where we need the hosts to send traffic or execute a command at the same time, it can be hard to synchronize them. However, in a Mininet testbed, it can

be easily achieved since the Mininet main process is able to send commands to the hosts it created.

This feature is very helpful in the experiments. When we need to generate random traffic, our requirement may be, for example, "send a flow every 0.1 second from a random host to another host," or "send traffic to any random host at random speed, but limit total ongoing traffic in the network to no more than 1 Gbps." These tasks cannot be done if we don't have a coordinator. Fortunately, Mininet can act as the coordinator. Using Mininet, the random traffic generating process can be very simple, as shown in Algorithm 2.

Algorithm 2 Generate random traffic

```
time ← 0
while time < targettime do
    Generate random host src and dst where src ≠ dst
    Send host src the following command: "use iPerf to send traffic to host dst"
    time ← time + 0.1
    Sleep 0.1 second
end while
```

Moreover, we can modify the sending interval to a random number in order to make the sending rate follow a particular random distribution. For example, we can generate an exponential distribution sending interval to make the sending rate follow a Poisson distribution.

When generating the desired traffic, we use iPerf to generate a constant bit rate UDP flow, and combine multiple iPerf instances to obtain the variable bit rate traffic. Ideally, we should use iPerf to measure total throughput as well, since iPerf is able to output the statistics for each flow it generated, including transferred bytes, packet loss, etc., as shown in Fig. 3.6a. However, iPerf sometimes gives anomalous output, shown in Fig. 3.6b, and the occurrence pattern is unpredictable. There's only one thing we know about the bug: it only occurs when the network is congested. Moreover, the bug is inside iPerf, so there's nothing we can do about it. Fortunately, we can use Mininet to circumvent this problem.

Since Mininet provides a VM-like environment for each emulated host, each host has its own emulated interface, and there are statistics on the interface. The statistics are maintained per-interface. Though this is not as good as the per-flow statistics provided by

```
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 0.20 MByte (default)
------------------------------------------------------------
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  5] local 10.0.0.10 port 5001 connected with 10.0.0.2 port 43753
[  5]  0.0-22.8 sec  16.5 MBytes  6.08 Mbits/sec   0.022 ms 1835/13631 (13%)
[  5]  0.0-22.8 sec  96 datagrams received out-of-order
```

(a) Normal output of iPerf.

```
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 0.20 MByte (default)
------------------------------------------------------------
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  9] local 10.0.0.4 port 5001 connected with 10.0.0.11 port 55378
[  9]  0.0- 0.3 sec  17592186044416 MBytes  440649483249308 Mbits/sec   0.000 ms 402390891/402390892 (1e+02%)
```

(b) Anomalous output of iPerf.

Figure 3.6: Normal and anomalous outputs of iPerf.

iPerf, the per-interface statistics are sufficient to calculate total throughput. Moreover, it is an error-proof way of throughput measurement. In the Mininet console, we type the following command to obtain the number of bytes sent (TX) and received (RX) at this interface:

```
mininet> h2 ifconfig h2-eth0
```

Irrelevant output.....

```
RX bytes:12845466 (12.8 MB) TX bytes:11036 (11.0 KB)
```

So fetching the number itself can be easily done with a single-line shell script.

- Get number of bytes received:

  ```
  ifconfig h2-eth0 | grep "RX bytes" | cut -d: -f2 | awk '{ print $1 }'
  ```

- Get number of bytes sent:

  ```
  ifconfig h2-eth0 | grep "TX bytes" | cut -d: -f3 | awk '{ print $1 }'
  ```

To obtain total throughput of the entire network, we can calculate the ratio between total bytes received and total bytes sent.

Using Mininet for network experiments also has its drawbacks. First, the testing topology cannot scale up arbitrarily. In Mininet, all switches and hosts are emulated by the workstation, therefore consuming the workstation's resources. Given the limited CPU and memory resources on the workstation machine, we cannot scale the experiment network up without

limits. In fact, a desktop PC with i7 CPU and 8GB memory can handle up to around 20 switches and 20 hosts, and there should be no more than 200 flows in the network simultaneously as well. Secondly, since all elements are sharing the resources of the host machine, the experiment in Mininet is easily influenced by other applications running in the host machine, especially when the host machine is a virtual machine running on another machine. Even if we use a dedicated desktop machine for the Mininet experiment, and are careful not start any other software during the experiment, the experiment results sometimes still have large variability. In this case, we have to run the experiment many times and take the average to eliminate the influence on this variability. Last but not least, even though Mininet provided APIs to set parameters such as packet loss rate and delay on links, it still cannot emulate the real-world network environment accurately. We cannot guarantee a network prototype which works in Mininet will also work in hardware switches. A gap still exists between Mininet emulations and hardware experiments; therefore, though Mininet can help test a network prototype, it can never substitute hardware experiments.

Chapter 4

Size-based flow management

Current networking solutions for cybersecurity adopt static policies for traffic routing through security devices. Packets to and from supercomputers go through a DPI device for security inspection before being routed toward their destinations. Since the DPI almost always represents the system bottleneck, it is typically the main factor that limits the performance of the entire network. In this chapter, we introduce a size-based flow management prototype to solve this problem.

4.1 Network prototype

Consider the current configuration of the K-State campus, shown in Fig. 4.1, as an example. Packets from and to Beocat (the K-State supercomputer) go through a Procera DPI device for security inspection before being routed toward their destinations. The bandwidth of the links is equal to 10 Gbps, but the speed of the DPI equals only 3 Gbps. If every packet is sent to the DPI for security purposes, the DPI will limit the bandwidth of the path down to 3 Gbps, representing the system bottleneck.

Most current designs either have no DPI at all, providing less security but higher performance, or every packet is required to route through a DPI, which is prohibitively expensive to license for large-scale dataflows and reduces performance. We aim at finding a solution

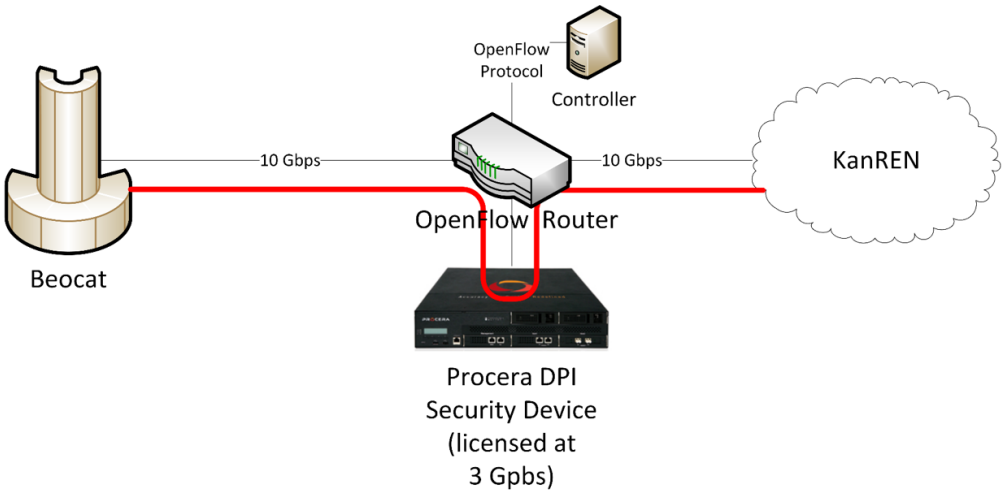


Figure 4.1: Basic DMZ configuration of the KSU network.

which takes care of both network performance and security.

Many publications[39;48–50] are interested in differentiating flows according to flow data rate (flow size), and applying different actions on them. The majority of these works focus on mice flows because mice flows often dominate more than 90 percent of the number of flows. However, in fact, elephant flows should receive a larger degree of security inspection, because they can take up more than 90 percent of the traffic, and therefore, are the flows which typically create network congestion. We observed that in a campus network, the traffic that takes the majority of the bandwidth (elephant flows) are science data. Moreover, science data is almost always secure. Therefore, we bring up the idea of allowing elephant flows to bypass DPI inspections.

Therefore, we can divide all traffic in the network into two groups based on flow sizes: one group consists of mice flows, while the other group contains elephant flows. Mice flows have a flow size smaller than a given threshold, while all other flows are elephant flows. When a flow arrives, we measure its size, and if it belongs to a mice flow, we route it through the DPI; but if it's an elephant flow, we allow it to bypass DPI. This is the general rule of our science DMZ.

When selecting threshold value, we must take both DPI processing rate and flow size distribution into consideration. We take a simple case as an example: we have two flows

with 20 Mbps and 30 Mbps data rates, respectively, and DPI processing rate is 40 Mbps. If we set the threshold to be more than 30 Mbps, it will yield network congestion; but if it is less than 20 Mbps, no packet will be inspected. Overall, 25 Mbps is a good value for the threshold. Thus, the threshold should not be picked arbitrarily. Even though the threshold is a fixed value after the network starts, we still need to pick the value carefully, considering the flow size distribution and DPI processing rate.

We need the size of a given flow to identify whether it's an elephant flow or a mice flow. We may want to obtain its flow size as soon as the new flow arrives, so we can route it to the right path at the earliest possible time, but this is neither feasible nor secure. First, when the controller realizes the existence of this new flow, it only has received a single packet on this flow. It's impossible for the controller to get any information on the size of this flow with only a single packet. Second, even though elephant flows are science data the majority of the time, there is still a chance that an attacker is trying to trick the system by sending huge amounts of traffic. Therefore, the best strategy is to route all flows into DPI for security inspection at the beginning. Later on, after we have gained enough information on this flow — both on flow-size and security sides, we make the decision if a rerouting is necessary. In this case, all flows are inspected by the DPI for at least a few seconds, so network security is retained.

With regard to how to obtain the flow size in realtime with enough measurement resolution, OpenFlow switches use statistics (counters), which are components in flow entry, to maintain flow statistics[51]. The counters collect statistics based on the number of bytes and packets received, and are maintained per table, per flow, per port, and per queue. In our prototype, we need the number of byte counters maintained per flow. The controller can query the switch for flow statistics through a secure channel[15], and the switch responds as soon as it receives the query.

Using the statistics field, we can easily get the size of any flow with the following procedure:

1. Program the controller such that it sends a statistic query every two seconds.

2. Let the controller obtain the number of bytes counter flow statistics from the switch.

   (a) If this is the first time we receive the counter of this flow, we save it in the controller.

   (b) If this is the second time we receive the counter of this flow, we find the difference with the previous counter we saved, and divide that by two seconds to get the flow rate.

It is worth mentioning that flow statistics responded by the switch are not in full real time. As shown in Fig. 4.2, flow statistics are updated at discrete time points: the red line shows the actual number of bytes received by the switch, and the blue line shows the number of bytes the switch will respond when it was queried. In OVS, it is updated every 0.5 second (0.25 second in some versions); in a hardware switch, this may take longer. This 0.5-second update interval means that flow statistics information that the controller receives can be delayed for at most 0.5 of a second. In other words, no matter how many times the controller queries the switch within this 0.5 second, the switch will always respond with the same result — the statistic that was most recently updated. Fortunately, this does not impact our prototype, because in order to obtain the size of a flow, we only care about the difference in number of bytes between two measurements. Though our first query is a little delayed, when we query for the second time, it will be delayed for the same amount of time, as two seconds is a multiple of 0.5 of a second, so the difference of byte counters will stay unchanged, as shown in Fig. 4.2.

Finally, according to the threshold mechanism on flow statistics, the controller may send a flow entry modification command to the switch when a flow is identified as an elephant flow. This finalizes the entire dynamic DMZ process. Details of how flow entry installation and modification works will be introduced in Section 4.2.

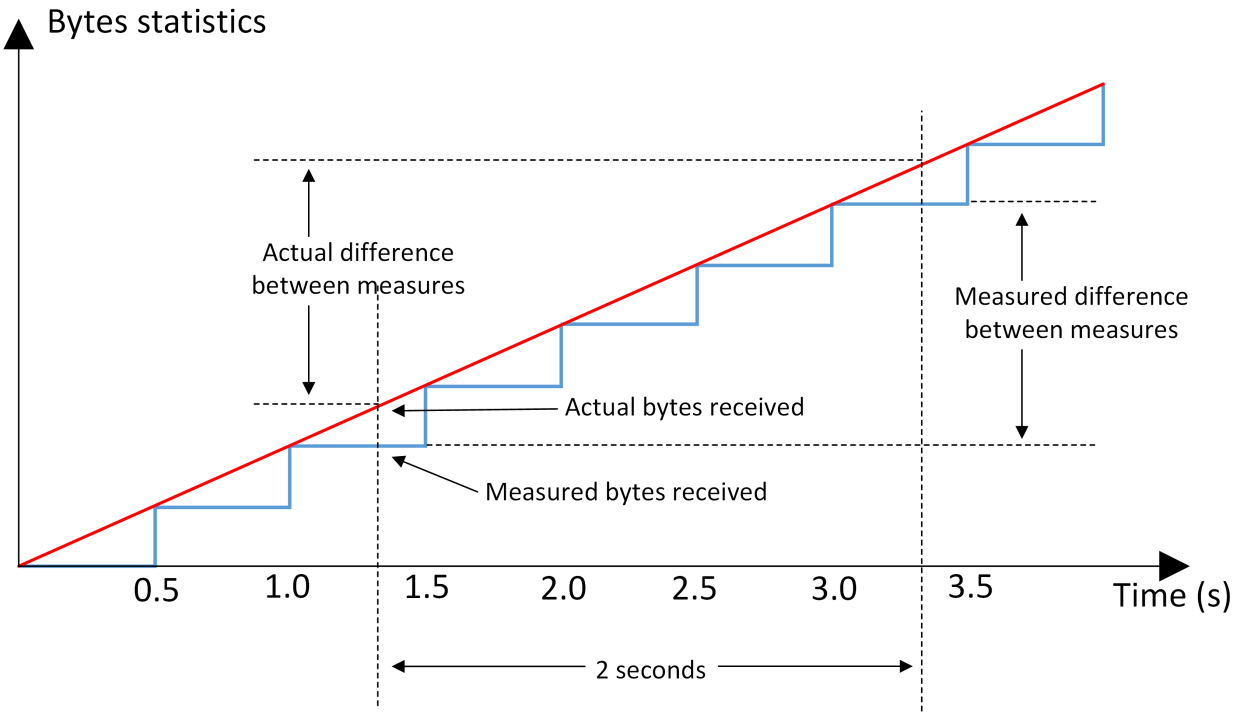


Figure 4.2: Timing approach for obtaining flow statistics.

4.2 Flow entry installation

In order to conduct a realistic experiment, the network topology is simplified, as shown in Fig. 4.3. The two hosts will represent the KanREN network and Beocat server, respectively, in Fig. 4.1.

In order to enforce network security, all flows are routed to the DPI when it arrives. This is accomplished by installing two flow entries: flow entry ① and ②, as shown in Fig. 4.4a. The flow will reach the switch twice, and is never modified; however, we need to apply different actions to the flow. The first time it arrives, it is sent to the DPI; and the second time, it's sent to the destination. Fortunately, there is an "incoming port" field in the matching tuple in the OpenFlow header that can be used to differentiate these two flows. In our case, flow entry ① tells the switch: for this single flow, if the incoming port is from H_1, then we forward it to the DPI; and flow entry ② means if the incoming port is from the DPI, we forward it to the destination. These two flow entries are listed in Table. 4.1.

When the flow is identified as an elephant flow, we can allow it to bypass DPI inspection.

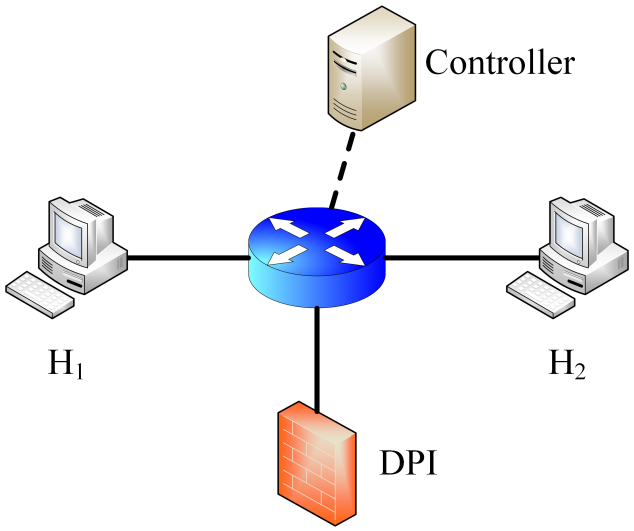
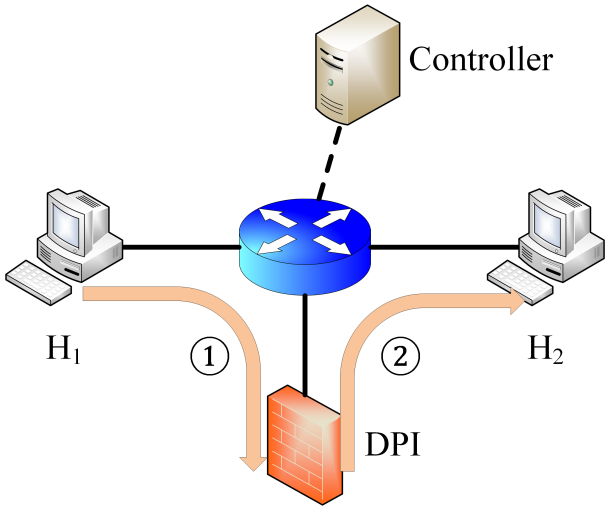
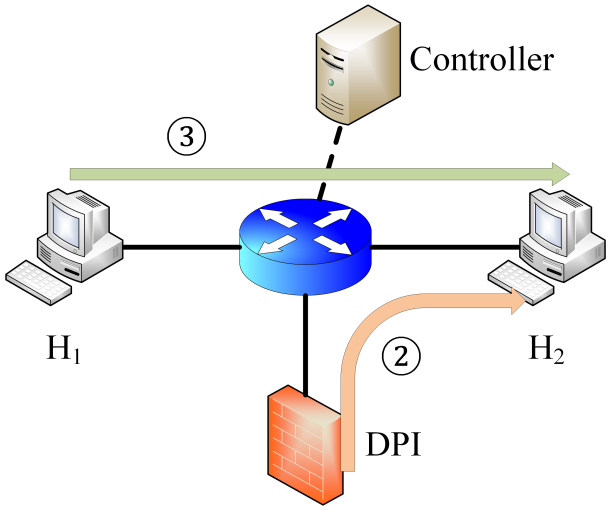


Figure 4.3: Simplified network topology.



(a) For any new flow, we enforce it to go through the DPI.



(b) Elephant flow is bypassing DPI.

Figure 4.4: Flow entry installation process.

This is done by modifying flow entry ① to flow entry ③, as shown in Fig. 4.4b. The flow previously destined for DPI will be sent directly to H_2. We don't delete the flow entry ②, because some packets may have already been sent to the DPI but not yet sent back. If we delete this entry, those packets will be lost.

Table 4.1: Flow entries to be installed in Fig. 4.4a for security enforcement.

in port	source	destination	src port	dst port	action
H_1	H_1	H_2	5001	5002	Forward to DPI
DPI	H_1	H_2	5001	5002	Forward to H_2

4.3 Rerouting example

Fig. 4.5 is an example of rerouting an elephant flow to bypass the DPI after its data rate it known to be above a threshold. This figure shows the data rate in Mbps on three interfaces of the software switch. Red and black curves are data rates on the DPI interface of two mice flows, respectively. The blue curve is the data rate of an elephant flow from the client, the pink dots represent the data rate of the elephant flow to the server, and the green curve is the data rate of the elephant flow on the DPI interface. Because the controller obtains aggregate statistics every two seconds, a peak in the DPI interface is observed. The peak represents the first several packets of the elephant flow initially routed to the DPI. The green curve drops back to 0 because this flow has already been rerouted, consequently bypassing the DPI; the blue curve and pink dots show no packet loss during the rerouting process. The two mice flows continue to be inspected normally throughout the measurement. Because the DPI interface records bidirectional packets, its curve is twice as high as those in other interfaces.

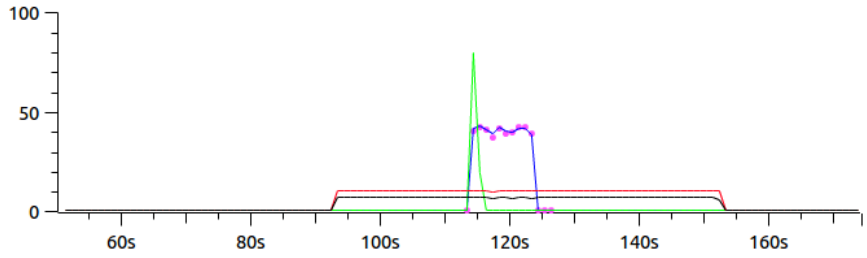


Figure 4.5: Routing example data rate curve.

4.4 Experiment

4.4.1 Experiment environment

We used three desktop computers for this experiment. The topology is shown in Fig. 4.6: two NIC cards are installed on a computer workstation configured as a virtual switch, connecting to hosts H_1 and H_2. We start a controller inside the workstation, which is also serving as the switch, and also create an emulated DPI, connecting it to the OVS switch. It doesn't perform any actual security inspection, but instead sends back whatever traffic it receives.

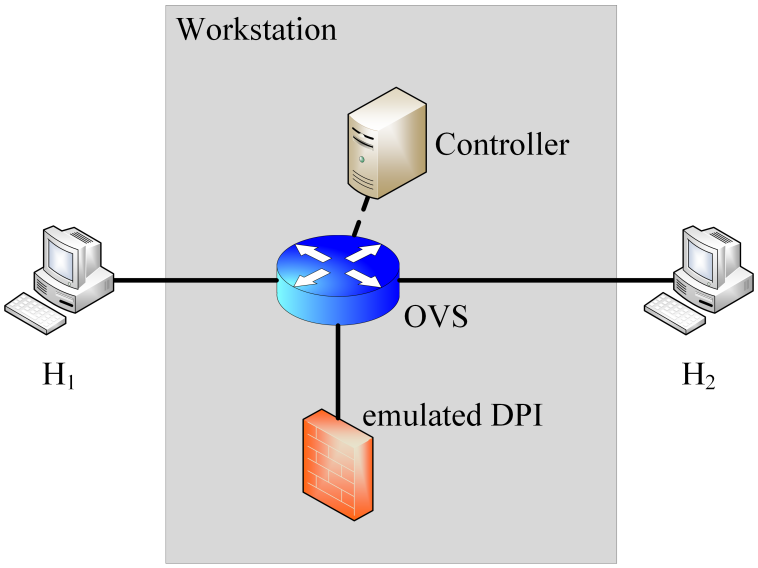


Figure 4.6: Topology used in the experiment.

All NIC cards have a licensed sending rate of 100 Mbps. We use the Linux command `tc` to limit the bandwidth between the switch and the DPI to 30 Mbps so it becomes the bottleneck. Consequently, when more than 30 Mbps data are coming into the DPI, the exceeded part will be dropped after the buffer is full. The same behavior will occur on a real DPI device.

We also use tcpdump to obtain network traffic measurements. Tcpdump is an open-source software useful for counting total size of multiple flows matching a specific filter. In order to obtain the current data rate and total number of packets, we run the command "`sudo tcpdump -i eth0 -l -e -n udp dst port 5001`" to capture all packets the switch

receives and sends. Then we pipe the results to a small script that counts the statistics and outputs them into a file every 0.1 of a second. We should add a parameter "`-Q in`" or "`-Q out`" in the command in order to identify ingress and egress packets on the DPI.

Specifications on the various components used for this evaluation are shown below:

Workstation: Ubuntu 12.04 LTS with Intel i7 Processor, 8GB RAM

H_1/H_2: 10/100Mb Ethernet card, iPerf 2.0.5-2[52] on Windows 7 Enterprise

Software switch: Open vSwitch 2.3.0

OpenFlow controller: POX carp branch

DPI: Open vSwitch on Ubuntu

4.4.2 Traffic generator

From the client, we send multiple UDP flows to the server. To generate flows with desired random characteristics, three parameters are required: flow size, flow start time, and flow duration. Flow start time follows a Poisson distribution, i.e., let N_i denote the number of flows starting in the i^{th} second, $P(N_i = k) = \frac{\lambda^k e^{-\lambda}}{k!}$, where λ is the average number of flows starting per second. For each flow, we assume the duration follows a uniform distribution, i.e., $t_j \sim \text{unif}(a, b)$ with an average of $\frac{a+b}{2}$. For flow size, the probability that a flow is an elephant flow follows a Bernoulli distribution with parameter p; i.e., a flow is an elephant flow for probability p, otherwise it is a mice flow. Flow sizes of both elephant flows and mice flows follow Gaussian distributions: $r_e \sim \Phi(\mu_e, \sigma_e^2)$ and $r_m \sim \Phi(\mu_m, \sigma_m^2)$, respectively. Therefore, the average data rate of the flow is

$$d = \frac{a+b}{2}\lambda[\mu_e p + \mu_m(1-p)] \tag{4.1}$$

When $\mu_e = 20$ Mbps, $\sigma_e^2 = 16$, $\mu_m = 3$ Mbps, $\sigma_m^2 = 3$, $p = 0.2$, the probability density is shown in Fig. 4.7. In general, the probability density is

$$f_R(r) = p\mathcal{N}(\mu_e, \sigma_e^2) + (1-p)\mathcal{N}(\mu_m, \sigma_m^2) \tag{4.2}$$

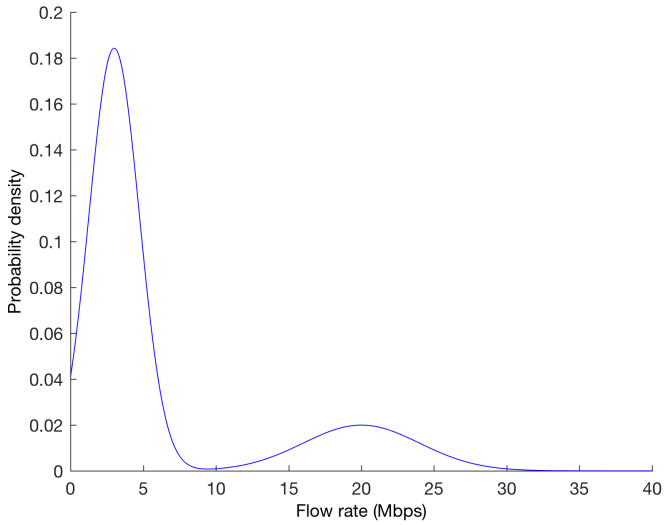


Figure 4.7: Sample flow rate probability density.

In addition, if $\lambda = 1.25$, $a = 5$ seconds, and $b = 15$ seconds, average data rate is $d = 80$ Mbps. Mice flows account for 80 percent of the number of flows, with a total data rate of 30 Mbps on average. The remaining 20 percent are elephant flows and total data rate is 50 Mbps.

Flows are generated with iPerf, an open-source network performance evaluator that can conveniently generate constant bit rate flows. We write a python script that generates flows by running iPerf multiple times with various flow rates and durations for each flow. Fig. 4.8 illustrates randomly generated traffic in which the Y-axis indicates total Mbps and the X-axis indicates time in seconds.

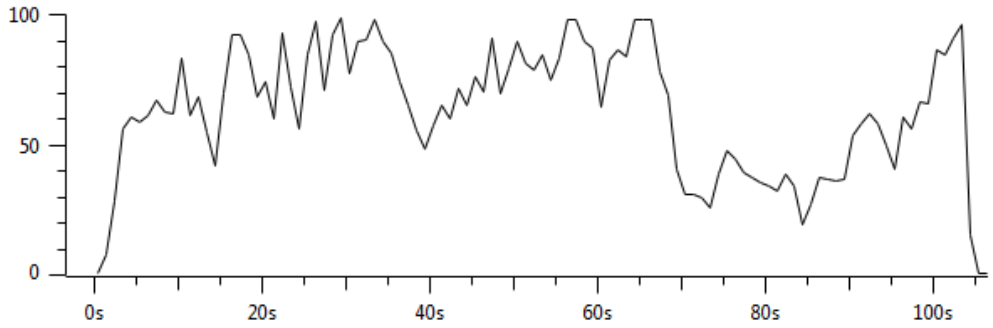


Figure 4.8: Sample generated traffic.

4.4.3 Experiment plan

In order to attest our prototype can help improve network performance, and learn how mice-elephant threshold influences performance of the entire network, we propose the experiments and corresponding definitions as follows:

1. Throughput vs threshold: throughput is defined as
   (Data Received by Server)/(Data Sent by Client)[53].

2. Packet loss vs threshold: packet loss is defined as
   1 − (Packets Received by Server)/(Packets Sent by Client).

3. DPI utilization vs threshold: utilization is defined as
   (Egress DPI Data Rate)/(Maximum DPI Data Rate).

Maximum data rates from H_1 to switch and from switch to H_2 are both 100 Mbps, and the DPI maximum processing rate is 30 Mbps. In all three experiments, we vary the threshold value from 2 Mbps to 28 Mbps, in 2 Mbps steps. For each step, we generate flows for 100 seconds and evaluate data for the corresponding experiment. We run each experiment five times and calculate the average of the results in order to minimize the influence of random traffic.

4.5 Packet loss theoretical analysis

Although total data rate of mice flows is 30 Mbps and the DPI processing rate is also 30 Mbps, it does not indicate that all mice flows can be transmitted successfully through the DPI. The two-second statistic pull interval results in a fraction of the DPI processing rate occupied by elephant flows that have not yet been rerouted. When the threshold is equal to 2 Mbps, elephant flows use an average of 11.62 Mbps DPI processing rate. When the threshold is equal to 10 Mbps, elephant flows take up 9.96 Mbps. That is because elephant flows have very high data rates and consume huge amounts of resources in a very short time (less than 2 seconds).

We can use theoretical calculations to analyze the amount of traffic transmitted to the DPI on average. Let D denote traffic sent to the DPI in one second. We use $f_R(r)$ to denote flow size probability density, h to denote the threshold, and t_r to indicate average duration of a flow, but we cannot simply write $D = \lambda t_r \int_{-\infty}^{h} r f_R(r)\mathrm{d}r$ because we are not rerouting elephant flows whenever one reaches the switch. Instead, we have to consider flow arrival time. Without loss of generality, we assume statistics are pulled from the switch every n seconds, and for a certain flow, we can always find a n-second interval that covers its arrival time $t(0 \leq t < n)$. Furthermore, t follows a uniform distribution in $[0, n]$. Therefore, we have[54]

$$f_T(t) = \frac{1}{n} \tag{4.3}$$

We also derive their joint probability density function as $f(t; r)$. Because f_T and f_R are independent from each other, we have[54]

$$f(t; r) = f_T(t) f_R(r) \tag{4.4}$$

Consider the arrival time t of an elephant flow. If t is very close to n, it is very likely we identify this flow as mice flow in this n-second time interval, and we need another n seconds to learn this is actually an elephant flow. So in order to identify an elephant flow in the n-second interval, we must have $\frac{(n-t)r}{n} \geq h$, i.e. $t \leq n - \frac{nh}{r}$. If an elephant flow can be identified in the current interval, we know the total data sent to the DPI will be equal to $g = (n-t)r$; otherwise, we have to wait for another n seconds. Therefore, $g = (2n-t)r$. So D can be formulated as

$$D(h) = \lambda t_r \int_0^h r f_R(r)\mathrm{d}r + \lambda \int_h^\infty \int_0^n g(t; r) f(t; r)\mathrm{d}t\mathrm{d}r \tag{4.5}$$

$$g(t; r) = \begin{cases} (n-t)r, & \text{if } t \leq n - \dfrac{nh}{r} \\ (2n-t)r, & \text{if } t > n - \dfrac{nh}{r} \end{cases} \tag{4.6}$$

In the experiment, we have $n = 2$, $t_r = 10$, $\lambda = 1.25$, and $f_R(r) = 0.8\mathcal{N}(3, 3) + 0.2\mathcal{N}(20, 16)$.

Therefore,

$$
\begin{aligned}
& D(h) \\
&= \int_0^h 12.5 r f_R(r) \mathrm{d}r + 1.25 \int_h^\infty \int_0^n g(t;r) f(t;r) \mathrm{d}t \mathrm{d}r && (4.7) \\
&= \int_0^h 12.5 r f_R(r) \mathrm{d}r + 1.25 \times \\
&\quad \int_h^\infty \left[\int_0^{2-\frac{2h}{r}} \frac{1}{2}(2-t) r \mathrm{d}t + \int_{2-\frac{2h}{r}}^2 \frac{1}{2}(4-t) r \mathrm{d}t \right] f_R(r) \mathrm{d}r && (4.8) \\
&= \int_0^h 12.5 r f_R(r) \mathrm{d}r + \int_h^\infty 1.25(r+2h) f_R(r) \mathrm{d}r && (4.9)
\end{aligned}
$$

The theoretical graph of D is the blue curve shown in Fig. 4.9, while the red curve shows results from the experiments. They agree very closely.

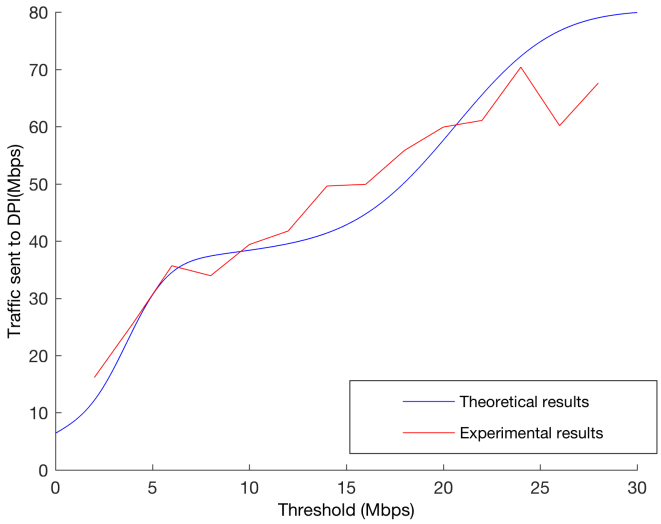
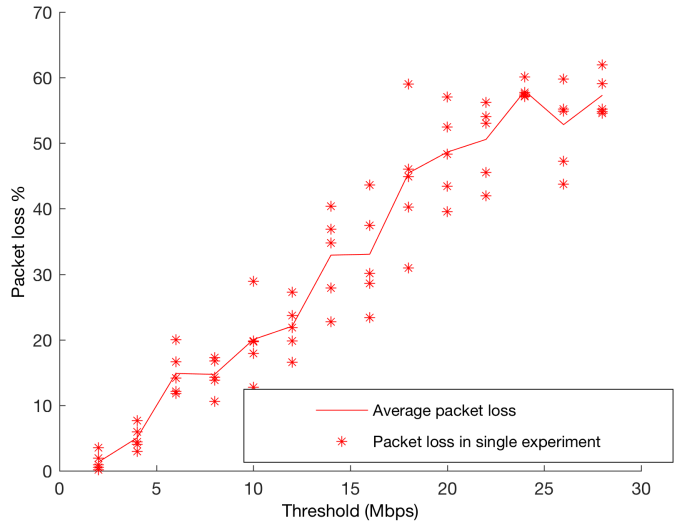


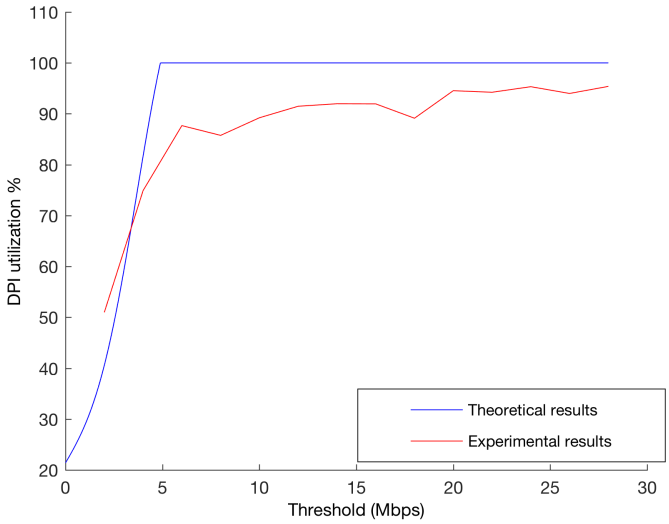
Figure 4.9: DPI ingress data rate D.

Therefore, according to theoretical analysis, when given a specific flow rate distribution and a DPI processing rate, the best threshold value can be found using this graph, thereby maximizing DPI utilization while guaranteeing high throughput. D as a function of h is a monotonously increasing function, and the function value of the best threshold value should equal (or slightly less than) the DPI processing rate.

4.6 Analysis of experimental results



(a) Packet loss.



(b) DPI utilization.

Figure 4.10: Dynamic DMZ experimental results.

We compared network performance with and without the dynamic DMZ approach. When the dynamic DMZ model is not implemented, the DPI is the bottleneck of the network and the egress data rate is limited to 30 Mbps because every packet is sent to the DPI. In the experiment, the egress data rate is 28.4 Mbps and corresponding throughput is 37.8 percent. However, when the dynamic DMZ model is implemented on the switch and the threshold

is set to equal 8 Mbps, the egress data rate becomes 61.7 Mbps and the corresponding throughput is 81.1 percent.

When the threshold increases, we expect the percentage of lost packets to increase because more packets will go to the DPI. Experimental results are shown in Fig. 4.10a. The throughput demonstrated the opposite trend with respect to packet loss. Packet loss and throughput will sum to 1, because all packets generated by iPerf have an identical size. For DPI utilization, as the threshold increases, DPI utilization is expected to become closer to 100 percent and ultimately remain there. The red curve in Fig. 4.10b shows results from experiments and the blue curve is the theoretical result following the model illustrated in Section 4.5.

The graphs meet our expectations very well, with experimental results nearly reaching the theoretical peak of 30 Mbps or 100 percent, while remaining subject to normal system overhead.

4.7 Summary

We proposed a dynamic DMZ model and developed a controller that simultaneously achieved high network performance for research data flows and higher network security than traditional DMZ configurations. The controller periodically pulled flow statistics from the switch, and sent a flow modification command to the switch to reroute elephant flows directly to the destination, thereby bypassing the DPI. Our experiments verified the efficiency of our approach and tested the influence of the threshold value on several important network performance indicators. Implementation of our approach allowed more traffic to be sent than the DPI processing rate, which is the bottleneck of the entire network. Setting a smaller threshold reduced packet loss and achieved higher throughput; however, fewer packets would be sent to the DPI for inspection because only initial packets of elephant flows would be inspected. Finally, we performed a theoretical analysis of the DPI ingress data rate, which is a guide of threshold selection with given flow rate distribution and DPI processing rate.

The work in this chapter is one of our main contributions, listed as contribution 1 in

Section 1.2.

Chapter 5

Congestion-aware flow management

A few drawbacks in the prototype are discussed in the previous chapter. First, we assumed all elephant flows are secure, and we will not send a flow back to a DPI for security inspection after it is permitted to bypass the DPI. However, a potential loophole exists which the attacker could exploit: the attacker can first send huge amounts of benign traffic to get permission to bypass the DPI, then start launching the attack. Second, the network is limited to a single-switch single-DPI topology, which cannot scale up. In this chapter, we will present a work which solves all these issues in the work of the previous chapter. This will be based on a general topology network with distributed DPIs. We proposed a new bypassing rule, which is no longer simply based on flow size. Instead, this bypassing rule is congestion-aware, because it can detect and eliminate congestions on DPI automatically.

5.1 Network prototype

In this network prototype, security is enforced by routing a flow through a DPI, and the DMZ is realized by allowing a flow to bypass DPI inspection. Consider a network topology with multiple switches and multiple DPIs, as shown in Fig. 5.1. Traffic demand can exist between every pair of hosts in the network. Each traffic demand is a flow, since it has a unique combination of source and destination IPs and port numbers. For each flow, we can

either force it to go through one of the DPIs, or allow it to bypass all DPIs. For example, consider a flow from H_3 to H_2. If this flow is assigned to a DPI, say DPI_1, then the path will be $H_3 \rightarrow S_5 \rightarrow DPI_1 \rightarrow S_5 \rightarrow S_2 \rightarrow H_2$, which is the shortest path from H_3 to DPI_1, followed by the shortest path from DPI_1 to H_2. If this flow bypasses the DPIs, the path will be $H_3 \rightarrow S_5 \rightarrow S_2 \rightarrow H_2$, which is the shortest source-destination path.

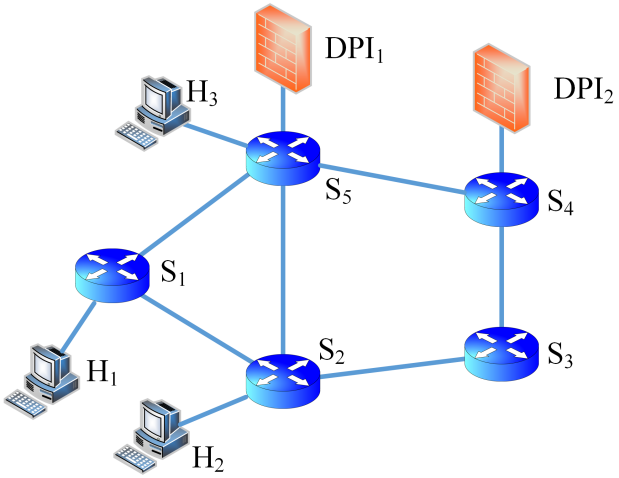


Figure 5.1: Sample multiple DPI topology.

Our goal is to maximize the security of the entire network, i.e., maximize the total data rate of inspected flows. However, in the meantime, we don't want to overly sacrifice network performance. Therefore, our objective is to achieve a balance between network throughput and network security.

To quantify network throughput and network security, we define network throughput as total bytes transmitted to a destination (in percentage of total bytes sent), and network security as the proportion of traffic inspected. In the following section of this work, we refer to these two objectives as "throughput" and "security."

5.1.1 Assumptions

No pre-knowledge about the network

We do not assume to know anything about the network before it starts. This includes whitelisted or blacklisted users, patterns of secure flows, demand matrix between users, etc.

In a general network, knowing the traffic-demand matrix in advance is impossible. Also, considering the possibility that a whitelisted host can also be compromised, these hosts are not 100 percent safe. Therefore, our approach is more general, secure, and realistic.

Links have sufficient bandwidth

In a network with security inspection devices, the devices always represent the bottleneck of the network, as the processing rate is always much lower compared to link bandwidth. Therefore, we can safely assume links always have sufficient bandwidth.

Flows don't split

First, flow splitting is not natively supported in OpenFlow. Also, though a DPI can work with individual packets, a DPI needs contiguous packets to "understand" the traffic content better when working at application or content level.

5.1.2 Wildcard rule for flow entries

As illustrated in Section 2.2.2, flow is defined as the set of packets which share the same packet-matching fields. In OpenFlow, 12 matching fields can be used as flow recognition[51], which includes source and destination MAC addresses, IP addresses, port numbers, etc. We can either use all 12 fields or set some fields as wildcards, and setting the wildcard rule is important to combine or distinguish traffics.

In our prototype, we will use a full 12-field matching, i.e., no field is set as a wildcard. This is because in the perspective of security, even if one application from a specific user is secure, it does not imply the other applications from this user are also secure. The decision of whether traffic from one application should bypass or not should not be interfered with other applications. A 12-field matching includes source and destination port numbers; therefore, it is capable of isolating each application from one another.

5.1.3 Accommodating new flows and optimizing current flows

When the switch receives a packet that doesn't belong to any existing flow entries, it knows a new flow has arrived and this packet is its first one. That packet is then sent to the controller so the controller is also aware of this new flow.

Ideally, the controller should find out the data rate of this flow, then execute a global optimization algorithm[55] to decide which DPI this flow should be sent to and how to reroute the other flows to accommodate it, finally sending rerouting commands to the switches. However this procedure is not feasible. First, the controller is not able to know the data rate of this flow at this point in time. Usually, we can determine the data rate of a flow by dividing the number of bytes received by flow duration. However, since we just received the first packet, flow duration is still 0. We need more time to find out its data rate. Second, even if we know its data rate, such frequent global optimization will overwhelm the controller. Lastly, frequent rerouting of the flows will cause packet loss in the network, further bringing down network throughput.

Therefore, we cannot find a perfect-fit DPI for a new flow when it first arrives; however, this flow has to be assigned to a DPI instantly to enhance network security. We propose the following two-step approach to achieve both security and throughput:

Step 1: Settlement

When the first packet of a flow arrives, since we don't yet know its data rate, we settle this flow in the DPI with the greatest remaining capacity, in order to maximize the probability the DPI will be able to accommodate this flow. This settlement step applies only to new flows; additionally, the arrival of this flow will not impact any other flows.

Step 2: Optimization

Every few seconds, a global optimization is executed. Unlike the settlement step, this step is global: all flows currently residing in the network will have this optimization performed jointly. The purpose of the optimization is to maximize utilization of the DPIs. It is performed with the following steps: the controller will first obtain flow statistics from all switches, which includes number of bytes in each flow and flow establishment time; thus

the switch knows the data rate and duration of each flow. Then the controller will forward this data to the optimization module, and results will be obtained shortly afterwards. Results will show which DPI is assigned to each flow, or this flow may bypass DPI inspection. Finally, the controller sends commands back to the switches to reroute flows according to optimization results. This step is performed periodically, so a flow may go through this step several times, thus being rerouted several times.

The settlement step inspects the initial part of each flow, guaranteeing the flow is initially secure; and the optimization step is able to inspect a flow which is already bypassing inspections, guaranteeing the flow didn't change its nature from benign to malicious. Thus this two-step approach is able to secure the network.

In the settlement step, we need the information of remaining capacity for each DPI in order to pick the largest one. However, there is no such API to query a DPI for its remaining capacity. The only time we can exactly know their remaining capacities is immediately following rerouting in the optimization step, because this is when we know the exact data rate of each flow and which flow is assigned to which DPI. After a new flow has been assigned to a DPI, we can no longer know the exact remaining capacity of it until the next optimization step. The workaround is assuming the new flow's data rate is equal to the average data rate of all flows from this host as seen so far.

5.1.4 Capacity reservation

As illustrated in Section 5.1.3, in this prototype, every flow is sent to a DPI when it first arrives in the settlement step, regardless of current DPI utilization. However, a DPI may become far too congested in this process. Consider the case where DPIs are already unable to handle all flows. Then after an optimization step, all DPIs are running at near 100 percent utilization, since the optimization step attempts to maximize utilization of the DPIs. Consequently, when a DPI accepts new flows in the settlement step, it is overwhelmed and packet loss will occur.

We propose a capacity reservation approach to solve this problem. Each time we reallo-

cate the flows, we reserve some processing capability on each DPI, i.e., we only use a fraction θ of the DPI's capacity. Formally, for a given DPI with capacity c, we will only use θc of the DPI capacity and reserve the rest for new-coming flows.

With regard to the value of θ, we can either pick a constant value, or dynamically compute a best fit θ according to previous traffic. The dynamic θ approach aims at estimating a suitable θ value for the current network. We can keep track of a θ_i value for DPI_i, which indicates what capacity the DPI_i can use, at most, for allocating current flows. If this DPI gets overwhelmed, i.e., we find this DPI was allocated more traffic than its capacity at the optimization step, we can then decrease the θ value to reserve more capacity for new-coming flows; otherwise, we would need to increase θ value in consideration of higher DPI utilization.

In this work, since the flow-arrival rate and data rate distribution will not change after the network starts, we applied the constant θ approach.

5.2 Formulation and solver

Since our objective is to maximize utilization of DPIs, we can formulate this into a linear programming problem and then solve it.

5.2.1 Basic formulation

Notations:

- R: objective, total data rate of inspected flows (i.e., network security).
- c_i: capacity of DPI_i.
- r_f: data rate of flow f.
- u_{fi}: binary variable: indicating flow f is inspected by DPI_i or not. If flow f is inspected by the DPI_i, $u_{fi} = 1$; otherwise $u_{fi} = 0$.
- θ: the proportion of DPI's capacity in use for existing flows, as explained in Section 5.1.4.

Basic formulation:

maximize

$$R = \sum_{f,i} u_{fi} r_f \tag{5.1}$$

subject to

$$\sum_{f} u_{fi} r_f \leq \theta c_i \quad \forall i \tag{5.2}$$

$$\sum_{i} u_{fi} \leq 1 \quad \forall f \tag{5.3}$$

$$u_{fi} \text{ is a binary variable} \quad \forall f, i \tag{5.4}$$

The u_{fi} binary variables are the only variables in the formulation. We can define $R_f = \sum_i u_{fi} r_f$, which denotes the data rate that flow f occupies in all DPIs. In fact, a flow can only be inspected by, at most, one DPI; therefore, $R_f = r_f$ if flow f is inspected by a DPI, 0 otherwise. In the objective (Eqn. 5.1), we want to maximize network security (i.e., total DPI-inspected traffic), which is $R = \sum_f R_f = \sum_{f,i} u_{fi} r_f$.

Eqn. 5.2 indicates total data rate of flows being inspected by a specific DPI_i will never exceed its capacity θc_i, as explained in Section 5.1.4. Eqn. 5.3 guarantees a flow may be inspected by at most one DPI, that is, a flow is either going through one DPI ($\sum_i u_{fi} = 1$) or bypassing all DPIs ($\sum_i u_{fi} = 0$).

Integer variables u_{fi} make this problem NP-Hard. No efficient algorithm exists to solve it. Fortunately, sub-optimal solutions are also acceptable in this situation. We can solve this problem using a heuristic algorithm, e.g. simulated annealing algorithm (SAN), or an ILP problem solver with a relative tolerance (EpGap in CPLEX, for example).

5.2.2 The impact of θ

From the analysis above, it is clear that picking the value for θ is important, as it has great impact on network performance. In extreme cases, if we want to maximize security, we should try to make the best use of DPI resources, with no resource reserved for future flows, i.e. $\theta = 1$. We can easily see this will cause massive packet loss. If we want to maximize

throughput, we should reserve as much capacity as possible at the DPIs, i.e. $\theta = 0$. However, $\theta = 0$ means no security resources are provided to existing flows. That is to say, all flows are inspected for only the first few seconds until they reach the first optimization step, then bypass inspection from that point on. Needless to say, either case is unacceptable.

In fact, we can balance network throughput and network security by merely picking a proper θ. As θ increases from 0 to 1, more DPI capacity will be utilized for current flows, while the capacity reserved for new flows will decrease. As a consequence, DPIs are more likely to be overwhelmed and the packets are more likely to be dropped. Therefore, we can predict that network throughput will drop while network security will rise. We will prove this with simulation results in Section 5.5.

5.2.3 Revised formulation

The objective of basic formulation is maximizing inspected traffic. In theory, this should work fine. However, a flow may have to be rerouted back and forth between DPIs frequently in order to satisfy the maximization of the security inspection, and the rerouting process may cause packet loss. We have to study further whether frequent rerouting has an impact on network performance. Therefore, we propose the following problem formulation, in which we aim at minimizing flow rerouting, while maximizing inspected traffic. However, since these two objectives are conflicting, we will merge them into one by computing a weighted sum.

Notations in addition to the basic formulation:

W: weighted sum of two conflicting objectives: total inspected traffic and flows stayed in place.

F: total number of flows.

u^*_{fi}: this constant is equal to 1, if in the previous optimization step, flow f is assigned to DPI_i; 0 otherwise.

u'_{fi}: a transformation variable for absolute value.

α: constant for weight balancing between total inspected traffic and flows staying in place.

Revised formulation:

maximize

$$W = \alpha \frac{\sum_{f,i} u_{fi} r_f}{\sum_i \theta c_i} + (1-\alpha)\left(1 - \frac{\sum_{f,i} |u_{fi} - u^*_{fi}|}{2F}\right) \tag{5.5}$$

subject to

Eqns. 5.2, 5.3 and 5.4

The only variables in the formulation are still u_{fi}. The revised formulation has the same constraints as the basic formulation; the only difference is the objective. As discussed previously, the objective W is a weighed sum of network security and in-place DPI assignment. Therefore, the objective is in the form of $W = \alpha W_1 + (1-\alpha)W_2$, where α is the weight used for balancing the two parts, ranging from 0 to 1. To make W_1 and W_2 comparable, we will normalize them to the range $[0, 1]$.

W_1 represents network security, i.e., DPI utilization; and thus it is the same with the objective R in the basic formulation, except it is normalized by the total capacity of all DPIs $\sum_i \theta c_i$.

W_2 shows to what extent flows are staying in place at this time point. Note that u^*_{fi} are known constants showing whether flow f was assigned to DPI_i in the previous optimization step, so $\sum_i |u_{fi} - u^*_{fi}|$ shows whether this flow f has moved (either moving to another DPI or bypassing). Both moving out from a DPI and moving into a DPI will contribute a value of one to the sum. Therefore, this sum equals two if this flow f is rerouted from one DPI to

another; the sum equals one if this flow was in a DPI and is now bypassing, or was bypassing and is now assigned to a DPI; and the sum is zero if this flow does not move at this point in time. This is coincident with how much packet loss will occur in each case: not moving at all being the lowest, and moving from one DPI to another being the highest. This is because starting flow-rerouting commands cannot be synchronized, thus in the case where a flow is moved from DPI_A to DPI_B, it's likely the flow is already moved in DPI_B, while the flows in DPI_B that are planning to move out have not yet done so. Returning to the formulation, because one flow can contribute at most a value of two to the sum and there are F flows in total, we can divide the sum by $2F$ to normalize it into range $[0, 1]$. Lastly, the normalized expression shows the extent that flows are moved, thus it is subtracted by one to show the extent that flows are staying in place.

This formulation contains the absolute value of a variable, which is non-linear. To adjust it to a linear programming problem, we introduce dummy variables u'_{fi} to replace the absolute value part, and constraints Eqns. 5.7, 5.8 to enforce absolute value rules. The final form of the ILP formulation is shown below:

maximize

$$W = \alpha \frac{\sum_{f,i} u_{fi} r_f}{\sum_i \theta c_i} + (1 - \alpha) \left(1 - \frac{\sum_{f,i} u'_{fi}}{2F}\right) \tag{5.6}$$

subject to

Eqns. 5.2, 5.3 and 5.4

$$u_{fi} - u^*_{fi} \leq u'_{fi} \quad \forall f, i \tag{5.7}$$

$$u_{fi} - u^*_{fi} \geq -u'_{fi} \quad \forall f, i \tag{5.8}$$

5.2.4 The impact of α

Variable α is the weight constant for balancing the two contradictory objectives. When $\alpha = 1$, the objective becomes "maximize $W = \frac{\sum_{f,i} u_{fi} r_f}{\sum_i \theta c_i}$." Because the denominator is a constant, the entire linear programming problem is identical to the basic formulation. As

alpha reduces, we emphasize more on the in-place flow assignment side. Our objective is to find a proper value for α, where the network has the largest throughput.

5.3 Optimization problem solver

Since both basic and revised formulation are ILP problems in nature, their solving steps are similar. Therefore, we will only show how to solve the basic formulation. Because of the binary variables in the formulation, this problem is NP-hard and cannot be solved in polynomial time. Even the branch and bound algorithm (BBA)[56], a very efficient algorithm for ILP, takes exponential time in terms of number of binary variables.

Many algorithms can provide a sub-optimal solution for this ILP problem. We will compare three of them: simulated annealing algorithm (SAN)[56], feasible ILP solver (FEA), and relative tolerance ILP solver (GAP)[57].

5.3.1 SAN algorithm

The SAN algorithm is a general heuristic algorithm for these NP-hard problems. SAN simulates the physical process of cooling down a material in order to approximate the optimal solution. SAN is a probabilistic algorithm in nature, so it's only able to provide a sub-optimal solution. Despite this, SAN is still an excellent algorithm in finding sub-optimal solutions in a discrete search space optimization problem because of its fast running speed, high optimality, and ability to jump out of local optimum.

5.3.2 Feasible ILP solver algorithm

We can also transform the basic formulation into a feasibility problem and run an ILP solver on it, for example, CPLEX. We call this feasible ILP solver algorithm (FEA). It provides a feasible, sub-optimal solution instead of the global optimal one. The formulation is shown below: original constraints remain unchanged, but we altered the original objective (Eqn. 5.1) into a new constraint (Eqn. 5.9).

maximize 0

subject to

Eqns. 5.2, 5.3 and 5.4

$$\sum_{f,i} u_{fi} r_f \geq \tau \sum_{i} \theta c_i \tag{5.9}$$

In the formulation, maximizing 0 means this is a feasibility problem; any solution which satisfies the constraints terminates the algorithm, thus shrinking running time greatly. However, we still require the original objective to be large enough, otherwise a trivial solution is $u_{fi} = 0 \; \forall f, i$, which is obviously feasible but not acceptable, since no flow at all is inspected.

Therefore, we introduced Eqn. 5.9 to enforce that the original objective must be greater than a threshold. In fact, the original objective is to maximize total inspected traffic, which is bounded by total DPI capacity. In other words, we want the total inspected traffic to be as close to total DPI capacity as possible, so the threshold should be less than, but close enough to, total DPI capacity. We can achieve this goal by using Eqn. 5.9 and setting the threshold parameter, τ, to be slightly less than 1, say 0.95. Though it's not guaranteed a feasible solution exists with this τ value, we can reduce the value of τ and rerun the algorithm until we find a feasible solution for the problem.

5.3.3 Relative tolerance ILP solver method

An ILP solver is able to solve ILP problems and find the optimal solution, but it usually takes a long time to do so. Running time can be greatly reduced by setting a relative tolerance, a.k.a. gap, in the ILP solver. In this case, the solver terminates as soon as the current integer feasible solution is proved to be within a certain percentage of the optimal solution.

The relative gap is a parameter. The larger the gap, the worse the solution, and the faster the algorithm. We can adjust the gap parameter according to our needs, so this is a very flexible algorithm.

5.3.4 CPLEX ILP solver

We formulated the optimization problem into an integer linear programming problem in FEA, GAP, and BBA algorithms. However, solving an integer linear programming problem is NP-hard. In order quickly to solve the optimization problems, we used a linear programming problem solver called CPLEX, developed by IBM.

When CPLEX solves an integer linear programming problem, it uses the branch-and-bound algorithm. It gives the optimal solution but the running time is long. By setting the "EpGap" parameter, we can set a relative tolerance between the current feasible solution and the optimal solution so the algorithm will run faster, but the solution we get is no longer optimal. We can also set the objective to 0, then the ILP problem becomes a feasibility problem.

The controller needs to call the CPLEX optimization module frequently. The best way to do this is calling the CPLEX API. Since the controller is written in Python, we desire to call CPLEX API in Python as well. However, the CPLEX Python wrapper PyCPX[58] has a memory-leaking bug, which will consume all memory gradually and eventually freeze the controller. Moreover, running speed of Python is a big concern. Therefore, the C++ version of CPLEX API was eventually used. We compiled the C++ code into an executable file, and communicated with the controller in Python with file I/O.

5.3.5 Comparison of algorithms

Fig. 5.2 shows the comparison of performance between these algorithms when optimizing a problem with four DPIs and 100 flows. In this chart, performance of each algorithm is evaluated in two aspects: objective value (blue bar on the left, normalized by optimal objective value) and running time (yellow bar on the right). Our aim is to find an algorithm with a large objective value and low running time.

All algorithms perform well on objective value. The GAP algorithm has the best performance, as it provides high objective value (99.9 percent of optimal solution) with low running time (30.7 ms). The SAN algorithm provides a lower objective value and takes

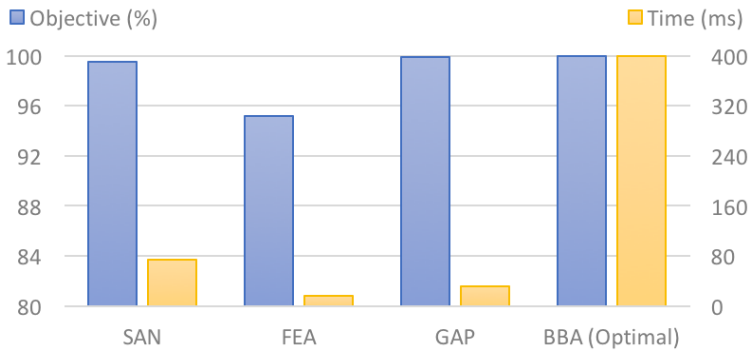


Figure 5.2: Algorithms efficiency and effectiveness comparison.

more time, so it's eliminated. The FEA algorithm is faster, but the objective value is not as good. In fact, we can adjust the gap parameter in the GAP algorithm to make it run faster, so FEA and GAP actually have a comparable performance. However, we eventually adopted the GAP algorithm instead of FEA, mainly because the FEA algorithm may be infeasible with the τ value we picked. Because we don't know the exact optimal value, we set the threshold proportional to total DPI capacity. Instead, the GAP algorithm is guaranteed to be feasible, since the termination condition is proportional to the optimal value. In case FEA is infeasible, we will have to run it again with a lower τ. The GAP is more stable in running time than FEA.

5.4 Experiment implementation

5.4.1 Network topology

To test the efficiency of our network prototype, we use a 14-node network topology as shown in Fig. 5.3. Every node shown in the figure is a switch; in addition, a host is connected to each switch, which is not drawn on the figure for simplicity. Four DPIs are deployed in the network. Note that a flow does not necessarily pick the nearest DPI. For example, there is a chance a flow from switch 1 to switch 2 is assigned DPI_4 for security inspection. In the experiment, each DPI has a processing capability of 140Mbps.

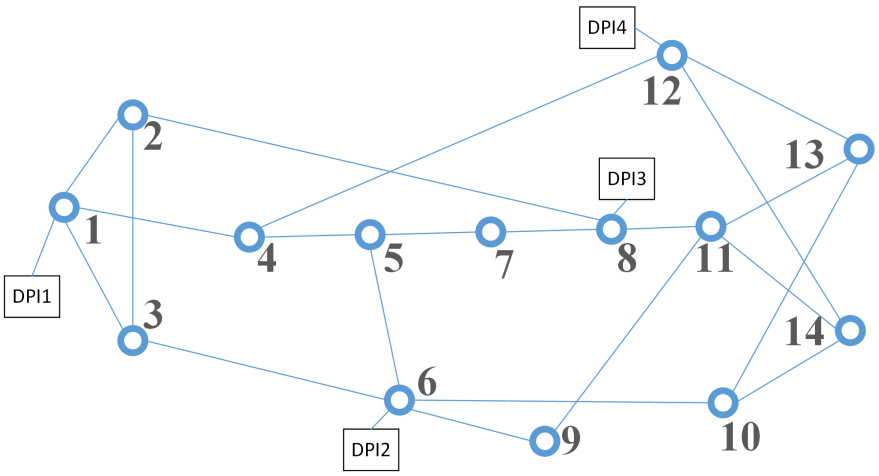


Figure 5.3: Experiment topology.

5.4.2 Flow generator

All flows are randomly generated. They are "random" in terms of sender, receiver, flow starting time, flow duration, and flow data rate.

Every host is potentially both a sender and a receiver. We can pick a random sender/receiver pair easily: first pick a sender among all 14 hosts in uniform distribution, then pick a receiver among the remaining (13 hosts) in uniform distribution. Flow arrivals follow a Poisson distribution, i.e., the time interval t between two flows' arrival follows the exponential distribution: $f_e(t) = \lambda e^{-\lambda t}$. Flow duration d follows the Weibull distribution[59] $f_w(d;\omega,k) = \frac{k}{\omega}(\frac{d}{\omega})^{k-1}e^{-(x/\omega)^k}$. Since network traffic consists of mice flows, which dominate in amount, and elephant flows, which dominate in size, we model the flow rate distribution as a Gaussian mixture model. Specifically, let r_e be the data rate of elephant flows, r_m be the data rate of mice flows, and proportion p be the probability a flow is an elephant flow, and we have $r_e \sim \mathcal{N}(\mu_e, \sigma_e^2)$, $r_m \sim \mathcal{N}(\mu_m, \sigma_m^2)$; and for a flow f, $r_f \sim p\mathcal{N}(\mu_e, \sigma_e^2) + (1-p)\mathcal{N}(\mu_m, \sigma_m^2)$. Therefore, the average data rate of each flow is $\mu_e p + \mu_m(1-p)$, and the total data rate inside the network in one second is $\lambda^{-1}\omega\Gamma(1+k^{-1})(\mu_e p + \mu_m(1-p))$.

For the experiment, we pick the following parameters: $\lambda = 0.2, k = 10$, and $\omega = 21$, so five new flows will arrive in one second on average, and the mean flow duration is 20 seconds. We also have $\mu_e = 16\text{Mbps}, \sigma_e = 4\text{Mbps}, \mu_m = 3\text{Mbps}, \sigma_m = 1\text{Mbps}$, and $p = 0.2$, which

gives us the average data rate of a flow as 5.6Mbps, and total data rate in the network as 560Mbps. We call this data rate "standard data rate," because at this point, total traffic is equal to the total processing capacities of all DPIs (140Mbps $\times$ 4). In the experiment, network performance will be measured when total traffic varies from 0.5$\times$ standard data rate to 4$\times$ standard data rate, in order to evaluate network performance in cases when DPI resources are abundant, just enough, or limited.

5.4.3 Experiment environment

We tested our flow-management prototype with Mininet[60], a network emulator which creates and emulates a network of virtual hosts, switches, controllers, and links. Mininet virtual hosts run standard Linux network software, and its switches run Open vSwitch software, which supports OpenFlow. By using Mininet, we can easily start a network with complex topology and send control commands to the hosts. All 14 switches started by Mininet are connected to a POX OpenFlow controller[61].

Traffic generating is done by iPerf, a useful tool for measuring TCP and UDP bandwidth. In order to measure bandwidth, iPerf will generate testing traffic so it can be used as a traffic generator as well. In iPerf, we can select our desired sending data rate in UDP. Though iPerf can only generate constant bit-rate traffic, we can write a script to start multiple iPerf processes, in order to get a time-variant total traffic.

All software mentioned above runs on an Ubuntu 16.04 Linux workstation, CPU: i7-860, memory: 8GB.

5.4.4 Flow data rate determination

We need an accurate flow data rate for each flow in the optimization step. We introduced how to obtain flow sizes for each flow in Section 4.1, and this method can be used directly in this prototype. At the beginning of this step, the controller will send a flow statistics request to each switch and wait for the reply. In the replied statistics, we are primarily interested in the following data: flow duration and byte count (i.e., number of bytes received). We can

obtain the data rate by simply dividing bytes received by flow duration.

The data rate obtained in this way is not 100 percent accurate. Though flow duration is accurate to the millisecond, byte count in the switch is not updated in full real time. Instead, it's updated every half second. This means, within this half second, no matter how many times we send flow a statistics request to the switch, we will always get the same result.

However, this method is causing new issues in this prototype. The 0.5-second flow statistic updating period is maintained per flow; therefore, rerouting a flow will reset the flow's statistic updating timer. Two seconds later, when we obtain a new flow statistics response from the switch, we cannot tell whether the data we collected is the byte count of 1.5 seconds (updating happens right after the query) or the byte count of two seconds (updating happens right before the query), as shown in Fig. 5.4. A possible way of solving this problem is, instead of sending flow statistics request every two seconds, we can send every 2.01 seconds. The extra 0.01 second gives the switch the time to update its flow statistics, so we can guarantee the response is the byte count of two seconds.

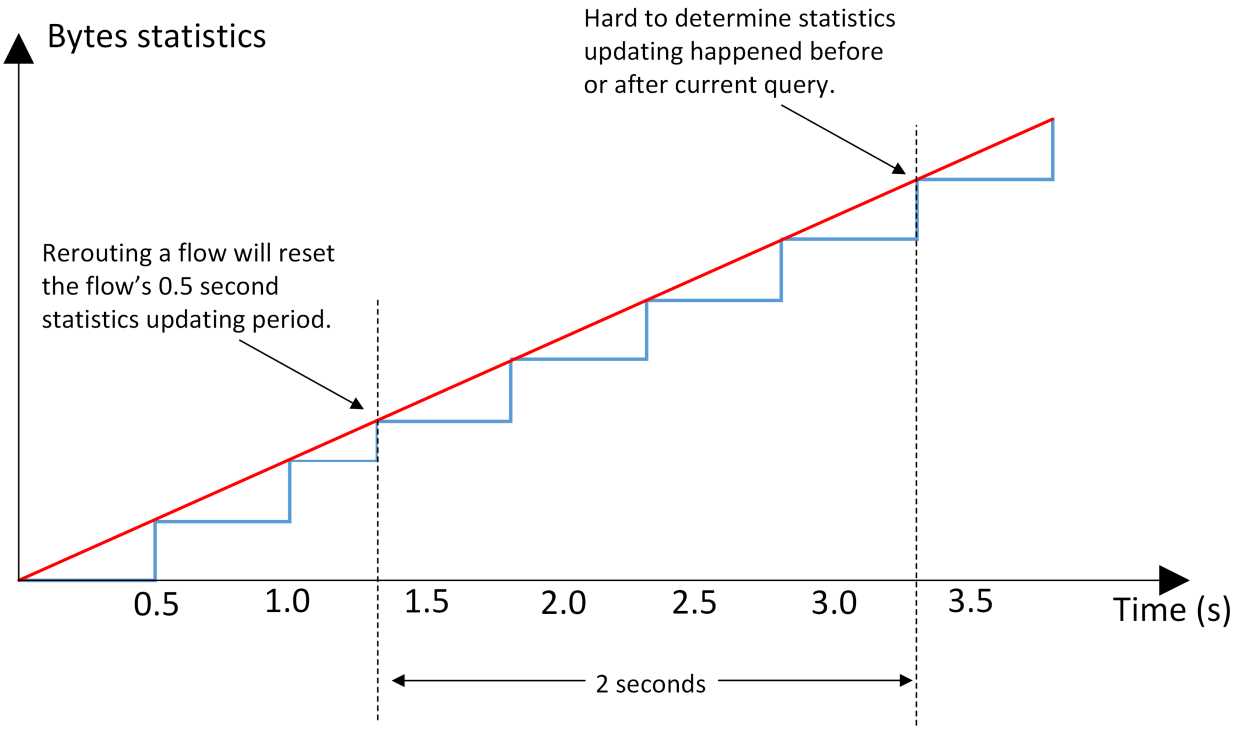


Figure 5.4: Flow statistics timing.

The multi-switch topology brings in a new issue as well. Because a switch will report flow statistics for all flows traversing through it, the controller will receive multiple reports of a given flow — equal to the number of switches on the path of this flow. Flow durations are identical, while byte counts sometimes may vary by a small amount. Maximal byte count among all switches always occurs on the first switch on its path because of packet loss when traffic travels in the network. Therefore, we used the maximum byte count in calculating the data rate as it's closest to the real amount of data transmitted.

5.4.5 Flow establishment and flow migration

In order to route a flow along a given path in the network, we need to install flow entries on all switches on that path. Flow entries will be modified in two scenarios: flow establishment in the settlement step, in which we need to install flow entries to accommodate the new flow; and flow migration in the optimization step, in which we need to modify the path for each flow that changed its path. We must pay attention to these flow-entry modification processes, because during them, packet losses are likely to happen, in addition to the scenario of packet loss on the DPI because of an overwhelmed DPI.

Packet loss will occur during flow establishment. This is because when a UDP flow arrives, the switches will not be ready to handle all packets. The only way to avoid packet loss is to pre-install the flow entries before the flow arrives[62], but this is not possible given we have no pre-knowledge about the upcoming traffic. However, we can still do our best to mitigate packet loss during flow establishment.

In OpenFlow when a new flow arrives at its first switch, in order to query what action should be applied to it, the first packet is sent to the controller. The controller determines the action and sends the flow-installation command to the switch. However, while the controller is making decisions, the switch is still receiving packets from this flow. Ideally, these packets should be buffered at the switch, waiting for the command from the controller. However, this is not the case. First, switches are usually implemented in a way that all packets are sent to the controller, as long as there is no corresponding flow entry, regardless of whether

another packet belonging to the same flow has already been sent to the controller. Second, even if the switch chooses to buffer all these packets on itself, it's very likely that its buffer will overflow. We used a controller packet buffer method to reduce packet loss. We first created a packet buffer at the controller to save a packet if the flow it belongs to is being processed by the controller. We made the switches send all packets to the controller as long as there's no matching flow entry installed. So before the switch receives a flow-entry installation command from the controller, it will send many packets to the controller. The controller will process the first packet and buffer the others, then send control commands to the switches, and finally send the buffered packets back.

Packet loss also occurs at flow migration. Rerouting a flow will more or less drop some packets, and thus we want to minimize the number of reroutings in the revised formulation. In order to reroute a flow at the optimization step, flow entries must be inserted, modified, or removed from switches. To avoid flow-entry modification and removal commands, which are likely to bring packet-loss issues, we performed the following flow rerouting procedure, which only consists of flow installation:

1. In the settlement step, we installed flow entries to switches with a priority equal to 1, which is the lowest priority.

2. In the following optimization steps of this flow, we installed flow entries with a higher priority value than before; for example, priority 2 in the first optimization step and priority 3 in the subsequent optimization step. We can do this safely because if a flow can be matched with multiple flow entries, the flow entry with the highest priority is adopted. We don't need to remove the low-priority flow entry manually, as it will no longer be matched to a packet. Its idle timeout will trigger and it will be removed by the switch automatically.

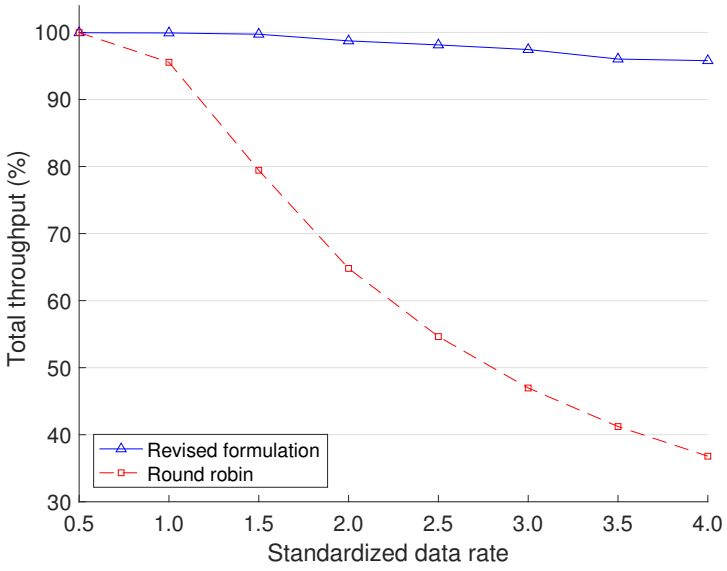
5.5 Results analysis

5.5.1 Effectiveness of our approach

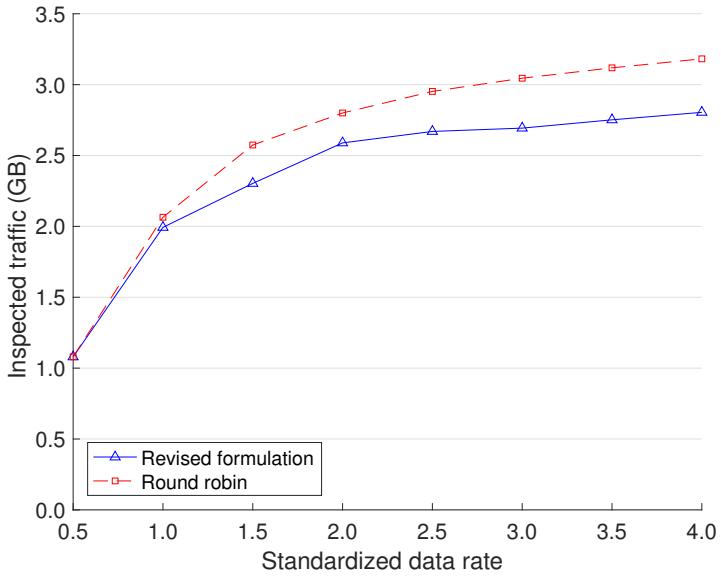
We first evaluated the effectiveness of our approach. Fig. 5.5 shows our experiment results on a performance comparison between our prototype (revised formulation with $\theta = 0.8$ and $\alpha = 1$) and a round robin approach:

In both graphs in Fig. 5.5, the X-axis represents the standardized traffic rate, i.e., the total data rate at this point normalized by the standard data rate (standardized traffic rate = $\frac{\text{total data rate (Mbps)}}{\text{standard data rate (Mbps)}}$). The standard data rate is a constant value as introduced in Section 5.4.2, which is 560 Mbps in this experiment, so the X-axis is essentially ranging from 280 Mbps to 2240 Mbps. The Y-axis in Fig. 5.5a shows the throughput of the entire network, i.e., what percentage of data sent are actually received. In Fig. 5.5b, the Y-axis denotes total traffic inspected by all DPIs in gigabytes.

In the round robin approach, the first-arriving flow will be assigned to DPI_1, the second will be assigned to DPI_2, then DPI_3 and DPI_4, and then loop back to DPI_1. No rerouting is performed in the round robin approach, so flows will compete for DPI resources, and therefore, DPIs become the bottleneck of the network. In this case, DPIs have near 100 percent utilization at all times, so the round robin approach will inspect a little more traffic than our approach. In fact, as shown in Fig. 5.5b, our approach inspects around 11.8 percent less traffic than the round robin approach. However, we can predict the total throughput of our approach is much higher than that of the round robin, because in a round robin, the network throughput is restricted by total DPI capacity. The throughput in our approach is close to 100 percent, since our rerouting mechanism allows flows to bypass DPI inspection, which can avoid DPIs being the bottleneck. Results are shown in Fig. 5.5a: when the standardized traffic rate is greater than 1, the round robin begins to show a huge decrease in throughput, while our approach consistently keeps the throughput greater than 95 percent. Overall, our approach greatly outperforms the round robin approach.



(a) Total throughput comparison.



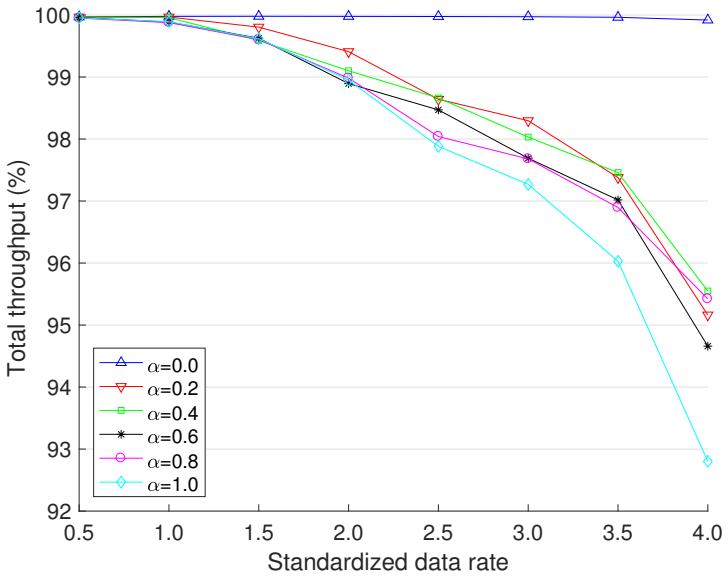
(b) Inspected traffic comparison.

Figure 5.5: Performance comparison between our prototype and round robin approach.

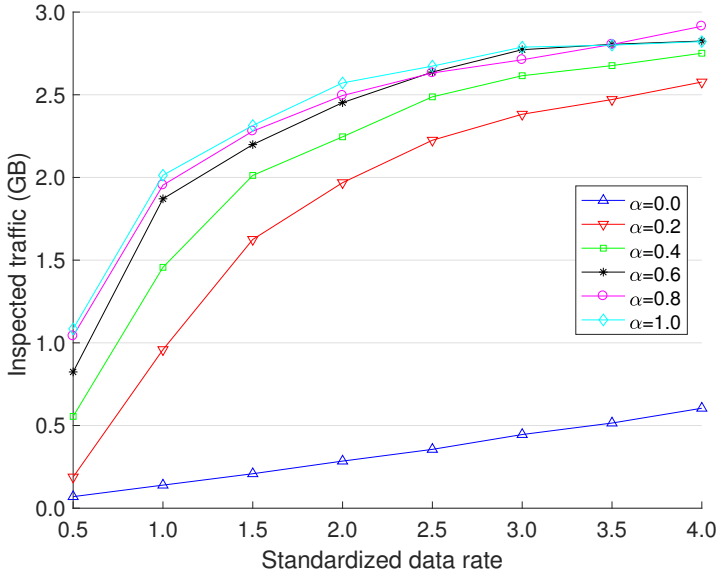
5.5.2 Impact of α in the revised formulation

Next, we studied the impact of α on network throughput and security in the revised formulation. We performed thorough experiments to measure network performance with different values of α when $\theta = 0.8$, and the results are shown below in Fig. 5.6. The axes have the

same meaning as Fig. 5.5. Each line in the figure shows network performance as a function of standard data rate for a given α.



(a) Total throughput at different α.



(b) Inspected traffic at different α.

Figure 5.6: Performance comparison with different α values when $\theta = 0.8$.

Results in Fig. 5.6b show inspected traffic elevated as α increases from 0 to 1. In fact, as α increases, we place more and more emphasis on total DPI utilization; therefore, more traffic will be inspected. Specifically, when $\alpha = 0$, DPI utilization is no longer a part of the

objective: the objective becomes "minimizing flow rerouting" when $\alpha = 0$. In this case, once a flow bypasses security inspection, it will never return to a DPI, even if the DPI has enough capacity to allocate it, because we want to minimize flow rerouting, and total inspected traffic is not even a consideration. That's why the total inspected traffic amount is so low at $\alpha = 0$.

On the other hand, as shown in Fig. 5.6a, network total throughput is very high when $\alpha = 0$ — near 100 percent. This is because almost all flows are bypassing DPIs; therefore, the DPIs never get overwhelmed. The throughput decreases as α shifts from 0 to 1.

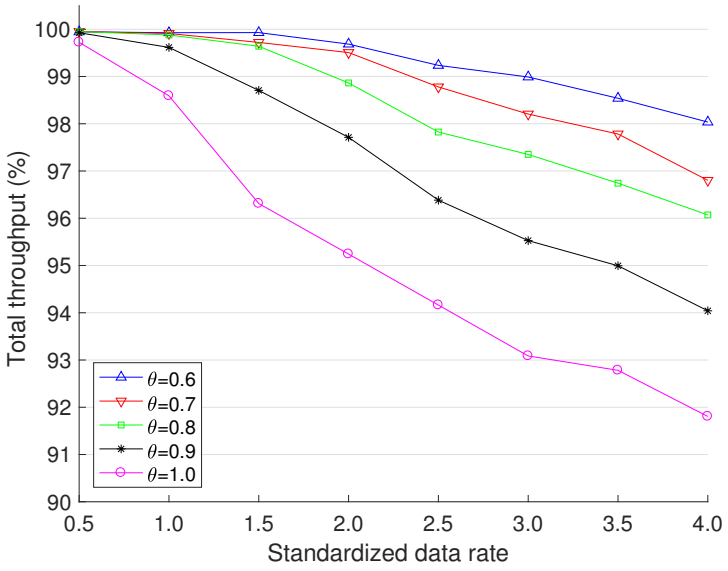
Considering both network throughput and security, the network performs best when $\alpha = 0.8$, since it has comparable security and more throughput than $\alpha = 1$. This means our revised formulation is effective: appropriately reducing the number of reroutings does have a positive impact on network throughput. The difference between their performances is not huge, so both $\alpha = 0.8$ and $\alpha = 1$ are acceptable α values. In the following experiment, $\alpha = 1$ is applied, though we can use $\alpha = 0.8$ as well. As analyzed in Section 5.2.4, a revised formulation with $\alpha = 1$ is identical to the basic formulation.

5.5.3 Impact of θ parameter

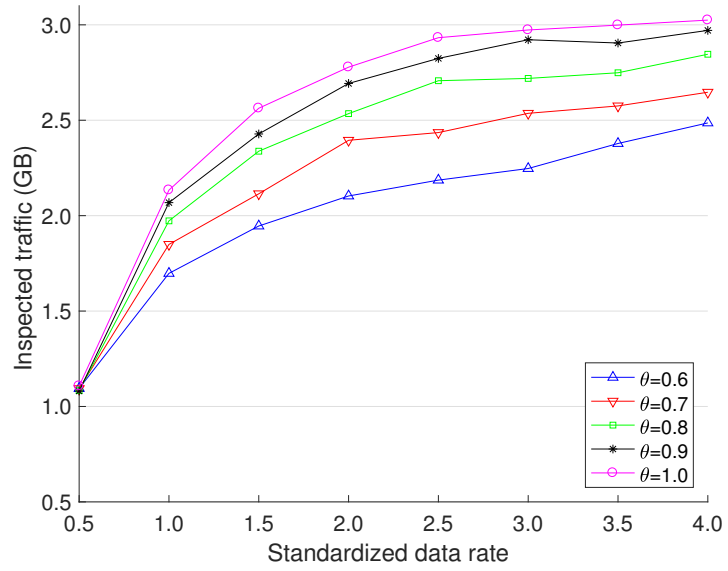
We studied the impact of θ on network performance. We iterated the value of θ from 0.6 to 1.0, and measured network throughput and inspected traffic when $\alpha = 1$. See Fig. 5.7 for the results.

In Section 5.2.2, we explained that as θ value increases, more DPI capacity is used for serving current flows, and less capacity is reserved for future flows, which leads to a higher chance of DPI becoming overwhelmed. Therefore, the total throughput will drop, while network security will rise. These predictions are well verified by experiment results shown in Fig. 5.7.

Picking the most proper value of θ depends greatly on network flow distribution. In our case, taking both throughput and security into consideration, we can select $\theta = 0.8$, which balances the two objectives nicely.



(a) Total throughput at different θ.



(b) Inspected traffic at different θ.

Figure 5.7: Performance comparison with different θ values when $\alpha = 1$.

5.5.4 Theoretical analysis on θ value selecting

Finally, we provide a theoretical analysis on selecting a proper θ value.

The ideal amount of capacity to reserve is equal to the total data rate of newly arriving traffic before the next optimization step. However it's impossible to predict upcoming traffic

amounts in the future. It's also not possible to guarantee a certain amount of capacity is 100 percent sufficient, because the total amount of traffic can be arbitrarily large in theory. Fortunately, we can predict reserving how much capacity is sufficient within a certain level of confidence by collecting some data from the network. For example, we can tell how much capacity to reserve such that the reserved capacity is enough 90 percent of the time.

We first run the network for a while to collect the following information: flow data rate cumulative distribution function (CDF) $S_R(r)$, number of flow arrivals in one second PDF $P_U(u)$, and average flow duration T. Note we don't have to obtain the exact formulation (and are not able to most of the time); an approximate statistical result will suffice.

Next, since the optimization step is performed every τ seconds, we just need to ensure our reserved capacity is greater than the total data rate arriving in this τ seconds. Total data rate can be denoted as $\sum_f r_f$, which is the total data rates of all flows arriving in τ seconds. Since flow data rate variables r_f are independent and identically distributed (i.i.d), we have $\sum_f r_f = F_\tau R$, where F_τ is a random variable representing the number of flows arriving in τ seconds, and R is a random variable denoting the data rate of a flow. Moreover, we have $F_\tau = \tau F$, where F is also a random variable denoting the number of flow arrivals in one second. The equality obviously holds by definition.

Considering flows are terminating in this τ seconds as well, let r_q be the total data rate of flows terminated; the net data rate arriving in this τ seconds can be denoted as $(\sum_f r_f) - r_q$. Now we can start to formulate the CDF of total data rate arriving in the next τ seconds. We use a random variable X to denote the total data rate arriving in the next τ seconds, and with $S_X(x)$ as the CDF of X, we have $S_X(x) = p((\sum_f r_f) - r_q \leq x)$ by definition. If we wish the reserved capacity to be sufficient 90 percent of the time, for instance, we simply let $S_X(x) = 0.9$ and solve for x.

r_q can be approximated as $r_q \approx \frac{B-x}{T/\tau}$, where B is total DPI capacity. This is because the flows being inspected by the DPI are taking up a bandwidth of $B - x$; we assume their starting time ranges uniformly, so $\frac{\tau}{T}$ of them will terminate in the upcoming τ seconds. Let $N = \frac{T}{\tau}$, and substitute the approximation of r_q in the formula, we get:

$$S_X(x) = p(\tau FR - r_q \leq x) \tag{5.10}$$

$$= p(\tau FR - \frac{B-x}{N} \leq x) \tag{5.11}$$

$$= p(\tau FR \leq \frac{B+(N-1)x}{N}) \tag{5.12}$$

by applying the law of total probability,

$$= \sum_{u=0}^{\infty} p(\tau FR \leq \frac{B+(N-1)x}{N} | F = u) P_U(u) \tag{5.13}$$

$$= \sum_{u=1}^{\infty} p(R \leq \frac{B+(N-1)x}{N\tau u}) P_U(u) + P_U(0) \tag{5.14}$$

$$= \sum_{u=1}^{\infty} S_R(\frac{B+(N-1)x}{N\tau u}) P_U(u) + P_U(0) \tag{5.15}$$

Now we can substitute $S_R(r)$ and $P_U(u)$ into Eqn. 5.15 to get the final expression of $S_X(x)$, and then plot it. So if we desire the reserved capacity to be sufficient for 90 percent of the time, with the help of the plot, we can find the value of x^* such that $S_X(x^*) = 0.9$. x^* is the best amount of total DPI capacity to reserve, and therefore $1 - \frac{x^*}{B}$ is the best θ value to pick.

We have validated that we can obtain an approximation of the original CDF by using the approach above. We first plot an actual CDF curve of the flow generator used in our experiment. Next, we generate a few flows using the flow generator, and reversely approximate the original CDF in the following cases: using 10 flows, using 100 flows, and using 1000 flows. Finally, we compare the estimated CDF with the actual CDF. Fig. 5.8 shows the results.

The figure proves the effectiveness of this approach. The CDF estimation from 1000 flows accurately matched the actual CDF. Even the estimation result from 100 flows is already very close. Note, while the figure may look like we are approaching the actual CDF from above as the number of flows increases, that's not always the case. It also happens that we are underestimating the CDF and approaching it from below. If we overestimate the

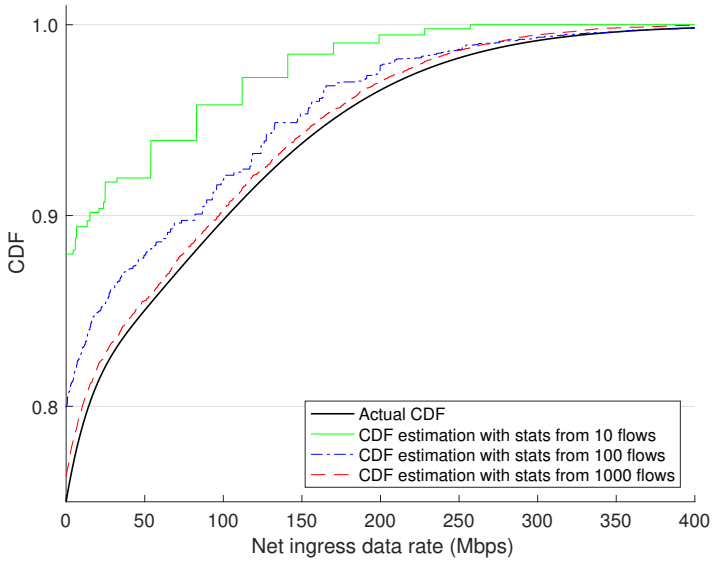


Figure 5.8: Estimating original CDF using flow stats.

CDF, we will underestimate the arriving traffic; therefore, the DPIs will more likely become overwhelmed. Conversely, if we underestimate the CDF, we waste DPI resources. We desire the CDF estimation to be as close to actual CDF as possible.

In practice, we can safely use the approximate CDF from 100 flows, and furthermore, find the best-fit θ. We can even integrate this approach into the controller in future work, so the controller can pick a proper θ parameter automatically after collecting information from 100 flows.

5.6 Enhancements over previous work

In general, this work presented our flow-management prototype for a network with limited security inspection resources. It aimed to solve the problem of how to allocate limited security inspection resources, or allow a specific flow to temporarily bypass security inspection, in order to enhance network security, while maintaining high-level network performance.

Compared to previous work on this topic, our prototype is superior in the following aspects:

1. General network topology: Previous work[1;40;41] has focused on a single-switch star topology or a star-like topology, while the network in our prototype is a general network. In a general network, flow routing becomes more complicated, and we must be careful when installing flow entries to minimize packet loss when establishing or rerouting a flow.

2. Distributed security devices: Previous work assumed resource shortages occur on a single, centralized security device. Scipass[40] integrated a load-balancing multiple IDS model; however, its 1Gbps firewall is the actual resource-limited security device, so essentially it is still a single-point resource-limited architecture.

3. No pre-knowledge: We had no knowledge of any network characteristics beforehand. This included whitelist, traffic demands, etc. This means our prototype is not an ad hoc design but a general approach to a common network issue.

4. Global optimization in real time: We designed a real-time optimization architecture for maximizing inspected network traffic. Optimization theory has been used successfully in many areas related to communication networks, such as optimal routing, flow control, and power control[63]. In this work, we formulate and solve an integer linear programming problem to maximize network security. Moreover, this architecture is general and can be applied on any prototype with real-time optimization, regardless of the objective or what solver/algorithm is used.

5. Capacity reservation: Instead of using up all processing capability in the DPI, we reserved a portion of the DPI capacity to accommodate new-coming flows. This mechanism enhances network security since we inspect all flows from the beginning and avoid overwhelming the DPIs, consequently reducing network packet loss.

6. No bypassing is permanent: In some previous work, once a flow was determined to be secure, it bypassed security inspection indefinitely from that point on. However, we can't guarantee the nature of a flow will never change. If bypassing is permanent, an attacker may trick the security system by sending benign packets first, then launch the

attack after the flow bypasses the security inspection. In our prototype, no bypassing is permanent. A currently bypassing flow may be rerouted back to a DPI at some time; moreover, the rerouting time is unpredictable. Therefore, we can minimize the possibility that an attacker can trick us by exploiting the loophole in the design.

5.7 Summary

Security devices are becoming the network's bottleneck, since security devices' capabilities are not able to catch up with increasing demands from users. To solve this problem, we introduced an OpenFlow-based flow-management prototype. The prototype allows flows to temporarily bypass DPI inspection and further increases network throughput. We also maximized utilization of DPIs by formulating an integer linear programming problem and solving it. Experiment results showed our approach gains 150 percent more throughput than the round robin approach. Results also proved that by picking proper parameters for θ and α in the formulation, we could achieve a balance between network throughput and security. At the end, we presented a theoretical analysis on how to pick a proper network parameter θ by collecting flow characteristics. This chapter is one of our main contributions, listed as contribution 2 in Section 1.2.

We have to admit, there are still some limitations in this work. We assumed the links always have sufficient bandwidth, and this may cause a flow to be assigned to a far-away DPI. Even in a network with unlimited link bandwidth, statistically, longer paths will lead to more packet loss. In future work, we can merge path length as part of the objective in the formulation. Li et al. has done a similar work, latency-aware middlebox scheduling, in[64]. Also, this work assumed flow arrival rate and data rate distribution will not change after the network starts; therefore, we used a constant θ value throughout each experiment run. However, considering peaks and valleys in real networks, a dynamic θ mechanism is more appropriate. The θ parameter should be adjusted to the traffic dynamically and automatically. We hope to accomplish this goal as well in future work. In the end, we would like to mention network function virtualization (NFV). It helps elastically allocate middleboxes' re-

sources, which is a cost friendly technology for increasing or decreasing middlebox resources as needed[65–67].

Chapter 6

Conclusions and future work

6.1 Conclusion

This dissertation is providing a solution for the following issue: in a local area network with limited security resources, how to allocate these resources to the flows, in order to trade-off network performance and network security. We proposed a size-based flow management prototype in Chapter 4, and a congestion-aware flow management prototype in Chapter 5. Networking tools utilized in the network emulation and experiment are also introduced, see Chapter 3.

Networking tools used in all the author's previous work are listed in Chapter 3. This includes Open vSwitch, POX controller, and Mininet. Open vSwitch is a virtual switch software, which can turn a Linux workstation into a switch. POX controller is an OpenFlow controller in charge of sending control commands to the switches. Mininet is a network emulation testbed. In Mininet, we can easily emulate and perform experiments on a network. We used hardware workstations and Open vSwitch for network performance evaluation in the first work, and Mininet to emulate a 14-switch network for testing in the second.

Chapter 4 introduces how we enhance network security in a single-switch network (star topology). We realize this dynamic DMZ model based on an OpenFlow-enabled switch and controller. In our approach, the controller detects flows with bit rate greater than a given

threshold (elephant flows) and controls the switch in order to reroute elephant flows bypassing the security device. Extensive experiments are performed to verify the feasibility of this approach and test how the threshold value influences network performance. Results indicate our approach effectively increases network performance but does not significantly influence flow security. Finally, we perform theoretical calculation on the deep packet inspection (DPI) input data rate in order to guide selection of the threshold value with a given traffic flow distribution and maximum DPI processing rate.

Chapter 5 explains how we extend the work in Chapter 4 into a prototype, which has general LAN network topology and distributed DPIs. In this work, the security inspection resources are still the bottleneck of the network, bringing down network throughput. We propose a flow management prototype, which can properly allocate limited security resources in order to achieve the objective of making the best use of security resources without compromising network throughput. We introduce a capacity reservation scheme to enforce network security and avoid security devices becoming congested. In order to optimize utilization of security devices, we formulate the resource-constrained problem as an integer linear programming problem and solve it. Extensive experiments are performed to attest to the effectiveness of our prototype. Finally, we analyze results of the experiment, including the impact on network performance of two parameters in the optimization formulations. Compared to other works, we have the following strengths: our model is implemented on a general network topology with distributed security devices; we formulate the flow allocation problem into a linear programming problem and perform the optimization in the controller in real time; and no pre-knowledge about the network, hosts, or traffic is needed.

6.2 Future work

In this work, all DPIs mentioned are hardware DPIs. However, hardware DPI has the following disadvantages: scalability issues, high expenses, hard-to-change configuration, vendor-specific configuration, etc. If we have more flows needed to be inspected, we have to buy a new device with higher capability to substitute the old one, deploy it, and reconfigure

it. Network function virtualization (NFV) is an emerging network technology that provides possibilities to solve those issues by decoupling the physical network equipment from the functions that run on them[68]. With the NFV technology, we no longer need hardware DPIs — we can just install DPI software on a virtual machine (VM) and scale it as needed. In addition, deploying NFV to provide DPI function is easier and with lower expenses — in case we need to change configurations, we just need to update the software.

Furthermore, not only can the DPI be virtualized to a function, but also any other types of middleboxes, say Firewall, intrusion detection systems (IDS), NAT, etc. NFV allows flexible network function deployment and promotes utilization of the resources. For example, if we have cloud NFV, we can obtain those resources based on our needs with more flexibility and lower expenses. We can adjust the resource we allocate to a given function dynamically, depending on usage of each virtual network function.

We propose to apply NFV in the science DMZ model in the future. Potential research work can be done on how to efficiently allocate resources in a NFV-enabled DMZ network, to maximize network security without compromising network performance.

There are also some limitations in our previous works. First, we used a workstation as a virtual switch in the work in Chapter 4. However, ideally, a hardware OpenFlow switch should be adopted. Second, in the congestion-aware network prototype, we assumed all links have sufficient bandwidth, so link bandwidth is not a part of the formulation. However, this may result in a long path when picking DPI for a flow. In the future, work should be done on adding path length as a variable in the formulation as well, to avoid link bandwidth being wasted.

Bibliography

[1] Haotian Wu, Xin Li, Caterina Scoglio, Don Gruenbacher, and Daniel Andresen. Size-based flow management prototype for dynamic dmz. In *Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the*, pages 191–196. IEEE, 2015.

[2] Haotian Wu, Xin Li, Caterina Scoglio, and Don Gruenbacher. Middlebox resources management using openflow. In *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*, pages 976–977. IEEE, 2016.

[3] Haotian Wu. Security enhancement in real time using sdn. *Computer Networks: Special issue Security and performance in SDN and NFV*, "submitted".

[4] Wikipedia. Wannacry ransomware attack. [Online]: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, 2017.

[5] Natalie Allen. Cybersecurity weaknesses threaten to make smart cities more costly and dangerous than their analog predecessors. *USApp–American Politics and Policy Blog*, 2016.

[6] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010.

[7] Wikipedia. 2016 dyn cyberattack. [Online]: https://en.wikipedia.org/wiki/2016_Dyn_cyberattack, 2017.

[8] Eli Dart, Lauren Rotman, Brian Tierney, Mary Hester, and Jason Zurawski. The science

dmz: A network design pattern for data-intensive science. *Scientific Programming*, 22 (2):173–185, 2014.

[9] Yale Center for research computing. High speed science network. [Online]: http://research.computing.yale.edu/services/high-speed-science-network, 2017.

[10] UAB-IT. Uab's $2.5 million investment will help speed campus network. [Online]: https://www.uab.edu/it/home/about-uab-it/announcements/item/739-uab-s-2-5-million-investment-will-help-speed-campus-network, 2016.

[11] Seungwon Shin, Lei Xu, Sungmin Hong, and Guofei Gu. Enhancing network security through software defined networking (sdn). In *Proceedings of The 25th International Conference on Computer Communication and Networks (ICCCN'16)*, August 2016.

[12] Changhoon Yoon, Taejune Park, Seungsoo Lee, Heedo Kang, Seungwon Shin, and Zonghua Zhang. Enabling security functions with sdn: A feasibility study. *Computer Networks*, 85:19–35, 2015.

[13] Jeremias Blendin, Julius Rückert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Position paper: Software-defined network service chaining. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 109–114. IEEE, 2014.

[14] Jeremias Blendin, Julius Rückert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Software-defined network service chaining. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 139–140. IEEE, 2014.

[15] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[16] Wikipedia. Middlebox. [Online]: https://en.wikipedia.org/wiki/Middlebox, 2017.

[17] Brian Carpenter and Scott Brim. Middleboxes: Taxonomy and issues. Technical report, 2002.

[18] Dilip A Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 51–62. ACM, 2008.

[19] Prasad Calyam, Alex Berryman, Erik Saule, Hari Subramoni, Paul Schopis, Gordon Springer, Umit Catalyurek, and DK Panda. Wide-area overlay networking to manage science dmz accelerated flows. In *Computing, Networking and Communications (ICNC), 2014 International Conference on*, pages 269–275. IEEE, 2014.

[20] Ian F Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, 71:1–30, 2014.

[21] Sugam Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2211–2219. IEEE, 2013.

[22] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[23] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 40(4): 351–362, 2010.

[24] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[25] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for

openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3, 2010.

[26] Vyas Sekar, Sylvia Ratnasamy, Michael K Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 21. ACM, 2011.

[27] Tamás Lukovszki and Stefan Schmid. Online admission control and embedding of service chains. In *International Colloquium on Structural Information and Communication Complexity*, pages 104–118. Springer, 2015.

[28] Xin Li, Haotian Wu, Don Gruenbacher, Caterina Scoglio, and Tricha Anjali. Efficient routing for middlebox policy enforcement in software-defined networking. *Computer Networks*, 110:243–252, 2016.

[29] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. *ACM SIGCOMM computer communication review*, 43(4):27–38, 2013.

[30] Taejin Ha, Sunghwan Kim, Namwon An, Jargalsaikhan Narantuya, Chiwook Jeong, JongWon Kim, and Hyuk Lim. Suspicious traffic sampling for intrusion detection in software-defined networks. *Computer Networks*, 109:172–182, 2016.

[31] Kostas Giotis, Christos Argyropoulos, Georgios Androulidakis, Dimitrios Kalogeras, and Vasilis Maglaris. Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments. *Computer Networks*, 62:122–136, 2014.

[32] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1990.

[33] Mea Wang, Baochun Li, and Zongpeng Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 628–635. IEEE, 2004.

[34] Benoit Claise. Cisco systems netflow services export version 9. 2004.

[35] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–8. IEEE, 2014.

[36] Junjie Zhang, Kang Xi, Min Luo, and H Jonathan Chao. Load balancing for multiple traffic matrices using sdn hybrid routing. In *High Performance Switching and Routing (HPSR), 2014 IEEE 15th International Conference on*, pages 44–49. IEEE, 2014.

[37] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[38] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 407–418. ACM, 2014.

[39] Tiago Fioreze, Lisandro Zambenedetti Granville, Aiko Pras, Anna Sperotto, and Ramin Sadre. Self-management of hybrid networks: Can we trust netflow data? In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, pages 577–584. IEEE, 2009.

[40] Edward Balas and A Ragusa. Scipass: a 100gbps capable secure science dmz using openflow and bro. In *Supercomputing 2014 conference (SC14)*, 2014.

[41] Simeon Miteff and Scott Hazelhurst. Nfshunt: A linux firewall with openflow-enabled hardware bypass. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 100–106. IEEE, 2015.

[42] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *NSDI*, pages 117–130, 2015.

[43] Author Unknown. Open vswitch, an open virtual switch. *Date Unknown but prior to Dec*, 30(2), 2010.

[44] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*, pages 1–6. IEEE, 2014.

[45] David Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.

[46] MurphyMc. Pox controller. [Online]: https://github.com/noxrepo/pox, 2012.

[47] Mininet Team. Mininet: An instant virtual network on your laptop (or other pc), 2012.

[48] Zhiyang Su, Ting Wang, Yu Xia, and Mounir Hamdi. Cheetahflow: Towards low latency software-defined network. In *Communications (ICC), 2014 IEEE International Conference on*, pages 3076–3081. IEEE, 2014.

[49] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3):270–313, 2003.

[50] Joe Wenjie Jiang, Tian Lan, Sangtae Ha, Minghua Chen, and Mung Chiang. Joint vm placement and routing for data center traffic engineering. In *INFOCOM, 2012 Proceedings IEEE*, pages 2876–2880. IEEE, 2012.

[51] OpenFlow Switch Specification. Version 1.0. 0 (wire protocol 0x01), 2009.

[52] Iperf. Iperf 2.0.5-2. [Online]: https://iperf.fr/, 2014.

[53] Sergio Marti, Thomas J Giuli, Kevin Lai, and Mary Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 255–265. ACM, 2000.

[54] John A Gubner. *Probability and random processes for electrical and computer engineers*. Cambridge University Press, 2006.

[55] Huu Thanh Nguyen, Anh Vu Vu, Duc Lam Nguyen, Manh Nam Tran, Quynh Thu Ngo, Thu-Huong Truong, Tai Hung Nguyen, Thomas Magedanz, et al. A generalized resource allocation framework in support of multi-layer virtual network embedding based on sdn. *Computer Networks*, 92:251–269, 2015.

[56] Michal Pióro and Deepankar Medhi. *Routing, flow, and capacity design in communication and computer networks*. Elsevier, 2004.

[57] IBM. relative mip gap tolerance. [Online]: https://www.ibm.com/support/knowledgecenter/en/SS9UKU_12.4.0/com.ibm.cplex.zos.help/Parameters/topics/EpGap.html, 2017.

[58] Hoyt Koepke. Pycpx. [Online]: http://www.stat.washington.edu/~hoytak/code/pycpx/, 2014.

[59] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[60] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[61] Hamid Farhady, HyunYong Lee, and Akihiro Nakao. Software-defined networking: A survey. *Computer Networks*, 81:79–95, 2015.

[62] Daniel Turull, Markus Hidell, and Peter Sjödin. Performance evaluation of openflow controllers for network virtualization. In *High Performance Switching and Routing (HPSR), 2014 IEEE 15th International Conference on*, pages 50–56. IEEE, 2014.

[63] George Tychogiorgos and Kin K Leung. Optimization-based resource allocation in communication networks. *Computer Networks*, 66:32–45, 2014.

[64] Qing Li, Yong Jiang, Pengfei Duan, Mingwei Xu, and Xi Xiao. Quokka: Latency-aware middlebox scheduling with dynamic resource allocation. *Journal of Network and Computer Applications*, 78:253–266, 2017.

[65] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[66] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Chunfeng Cui, Hui Deng, et al. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, pages 22–24, 2012.

[67] Leonhard Nobach and David Hausheer. Open, elastic provisioning of hardware acceleration in nfv environments. In *Networked Systems (NetSys), 2015 International Conference and Workshops on*, pages 1–5. IEEE, 2015.

[68] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2016.