# 修 士 学 位 論 文

# Master's Thesis

論文題目

Thesis Title

A Study on Flow Table Overflow Attack Mitigation

in SDN using Network Functions Virtualization

SDN におけるネットワーク機能仮想化を用いた

フローテーブルオーバーフロー攻撃低減手法

提 出 者

東 北 大 学 大 学 院 情 報 科 学 研 究 科 応用情報科学専攻

Department of Applied Information Sciences

Graduate School of Information Sciences, Tohoku University

氏名（Name）Mustafa SOYLU　（　ムスタファ・ソユル　）

| | |
|---|---|
| Advising Professor at Tohoku Univ. | Professor Takuo Suganuma |
| Research Advisor at Tohoku Univ. | |
| Dissertation Committee Members Name marked with "○" is the Chief Examiner | ○ Professor Takuo Suganuma<br>1 Professor Go Hasegawa    2 Professor Hiroki Nishiyama<br>3 Assoc. Prof. Toru Abe    4 Assoc. Prof. Satoru Izumi<br>5    6 |

令和 3 年度　修士学位論文

# A Study on Flow Table Overflow Attack Mitigation in SDN using Network Functions Virtualization
# (SDNにおけるネットワーク機能仮想化を用いたフローテーブルオーバーフロー攻撃低減手法)

東北大学大学院情報科学研究科 応用情報科学専攻

博士課程前期2年の課程

情報ネットワーク論講座(菅沼・水木研究室)

B9IM4503　Mustafa SOYLU

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Background

### 1.1.1 SDN and OpenFlow

Software Defined Networking (SDN) is an emerging networking paradigm that separates data and control layers and gives an overall view of the network domain [1]. Instant access to the networking device and centralized control of the domain enables flexible and dynamic approaches to manage the network. SDN brings novelty to the networking field by using a standard protocol to manage all networking devices. An SDN controller communicates with the SDN-enabled networking devices using a Southbound Protocol while traditional networking devices are managed through a CLI. However, the CLI syntax of traditional networking devices greatly vary according to their brand and model. The flexibility granted by SDN also stems from being able to control different devices with the same controller and protocol.

The centralized view and control of SDN makes the operation simpler within its domain. Moreover, SDN allows networks to be more programmable. Cloud giants such as Google and Amazon started implementing SDN in their cloud domain in 2017 [2]. Also, there is research about implementing SDN in a low-cost home network environment, just using a Raspberry Pi as both an SDN switch and controller [3]. Therefore, networks with different sizes and purposes implement SDN in their domain nowadays.

The communication from the controller to the devices is conducted through South-

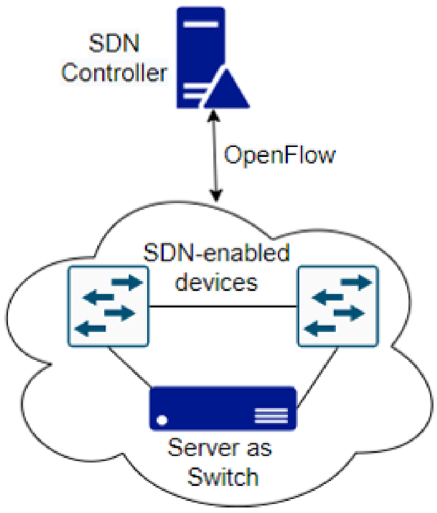

Figure 1.1: Representative SDN Topology

bound Interfaces (SBI), which can be driven by different protocols such as such OpenFlow (OF) and NETCONF. Due to its widespread use, this research will only consider OF as the representative SBI [4]. An OpenFlow switch embodies OpenFlow channels that connect to the external controller(s) and the flow tables that handle the network traffic.

Table 1.1: Components of a Flow Table

| Match Fields | Priority | Counters | Instructions | Timeouts | Cookie | Flags |
|---|---|---|---|---|---|---|

Table 1.1 shows the main components of a Flow Table, namely: *Matching Fields* to specify the values of packet headers; and *Instructions* to define how to handle the packets if a packet matches flow rule. Additionally, it also contains other features, such as *Timeouts* which show the maximum amount of time or idle time before the flow is expired; *Cookies* which makes grouping easier in the controller. Finally, *Flags* can be set to implement some functionality or to change the way flow entries are processed. For instance, the OFPFF_SEND_FLOW_REM flag enables an asynchronous message to be sent to the controller if the flow is deleted.

SDN switches handle the data plane based on the flow entries of the table, if a packet matching happens then the device applies the necessary action. Figure 1.2 shows the flowchart of a packet processing procedure from OpenFlow Switch Specification

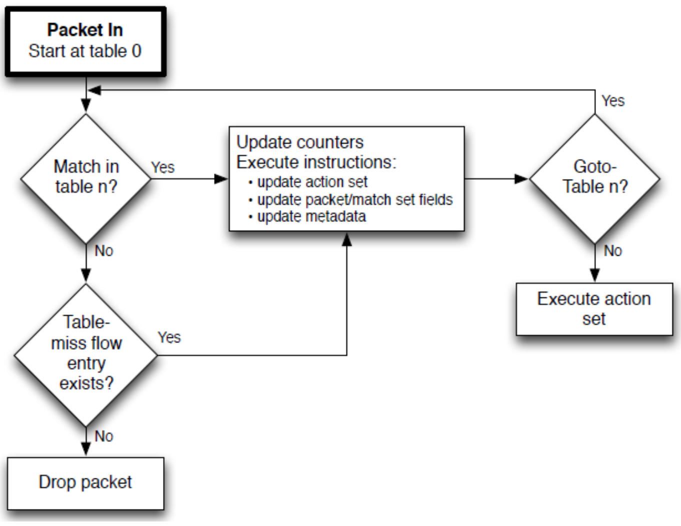

Figure 1.2: Packet Processing Flowchart [5]

v1.3.5 [5]. As observed, the switch conducts a table lookup when it receives a new packet, which is conducted on the first table, and if the packet matches the header fields then the necessary actions are implemented. If no match is found in the first table, then it goes through the next table in the pipeline. However, if there is no match in any of the tables, the *table-miss* action is applied; this action might be drop the packet (i.e., default) or any other action based on the network policy.

A table-miss flow entry is a default flow entry with all of its matching fields are “do not care” values. It is a must-have for the OpenFlow switches according to the OpenFlow Switch Specification. Traditional networks have the same concept as a default rule that drops packets with unknown origins. However, in some cases, the device would want to request the controller for further instructions. In which case, the packet is encapsulated in a *packet-in* message and send to the controller.

Packet matching is one of the most critical parts of SDN. The packet header is dissected into fields for matching, along with the ingress port, metadata field, and other pipeline fields. There are 40 matching fields in total on OpenFlow 1.3.5. Some of these matching fields, such as Ethernet address and IP address, are open to masking. However, the transport layer port addresses are not open to masking in any version of OpenFlow yet. Traditional networking devices use subnetting Classless Inter-Domain Routing (CIDR) masks to manage the traffic. However, OpenFlow allows the usage of arbitrary bitmasking as well as CIDR masks. This feature of OpenFlow increases the possibility to create new traffic management and security measures.

Table 1.2: Matching with the Value and Mask

| | | **Value** | |
|---|---|---|---|
| | | **0** | **1** |
| **Mask** | **0** | no constraint | error |
| | **1** | must be 0 | must be 1 |

Table 1.2 illustrates the relationship between a mask bit and the corresponding value for the matching. For instance, in case we want to mask a bit (i.e, the bit can be either 0 or 1) the corresponding bit of the mask is set to 0. However, the value to match must be 0 in this case in order to prevent the OFPBMC_BAD_WILDCARDS error. Mask bit is set to 1 when the value bit for matching is a specific value.

### 1.1.2 Network Function Virtualization

The significance of the internet in our daily life increases day by day; thus, the demand for better and faster network environments increases too. Networks are getting more complex and hard to manage as traditional networks combine diverse network functions statically. Traditional networks consist of vendor-specific software and hardware.

Network Function Virtualization (NFV) [6] allows network functions to be implemented by general computing resources such as commodity servers. The separation of software and hardware allows each of them to be improved independently. Also, reassigning and sharing of the hardware resources give flexibility to network management. With NFV, new functions or resources can be added to the network in a short period of time, which grants a dynamic operational ability as well as the capability to implement function automation. However, one of the biggest issues with NFV is its performance. Since general-purpose servers are used in the implementation of the NFV, there are latency and throughput concerns.

The European Telecommunications Standards Institute (ETSI) created an NFV Architecture Framework document that summarizes a possible NFV system structure [7]. ETSI envisions that NFV Infrastructures to replace physical devices providing network functions, replacing them with software-based solutions, ultimately preventing vendor dependency. As shown in Figure 1.3, NFV consists of three main components:

- Virtualized Network Function (VNF): a network function that is implemented as software on an NFVI.
- NFV Infrastructure (NFVI): physical devices that can support running VNFs.
- NFV Management and Orchestration (NFV MANO): administration of both physical and software resources for virtualization.

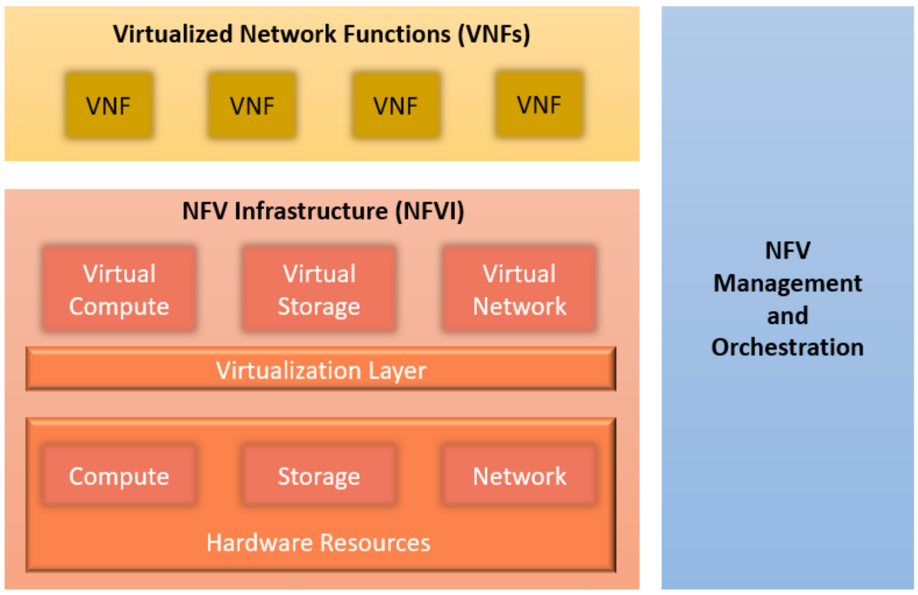

Figure 1.3: Architectural Framework of NFV

## 1.2 Purpose of this Research

Nowadays, there is a massive number of devices connected to the networks in the whole world. Internet is the web of all of these connected devices, and it is still growing exponentially. The number of internet-connected devices also increases every day, with the diffusion of smart devices such as home appliances. Devices with limited processing power are an easy target for hacking activities since they cannot implement elaborate malware protection. M. Antonakakis et al. [8] indicate that the Mirai botnet, a group of hacked devices and servers controlled from a remote control center, has reached a population of 200,000-300,000 infections. Members of the Mirai botnet are used for countless attacks, such as Distributed Denial of Service (DDoS) attacks. DDoS attacks aim to overwhelm network resources by sending enormous network traffic. Legitimate users cannot access the service/network in a DDoS attack; thus, it is called Denial of Service.

In an SDN domain, OpenFlow switches handle the traffic by the flow entries in their flow tables. The flow table structure is implemented in hardware using Ternary Content Addressable Memory (TCAM). TCAM accesses its contents parallelly, which grants quick flow matching to OpenFlow switches. However, TCAMs have nearly 100 times greater power consumption and they cost 100 times more, when compared to the conventional Random Access Memory (RAM) [9]. Therefore, the size of a flow table is limited to several thousand flow entries [10]; which is precisely the vulnerability that a flow table overflow attack exploits.

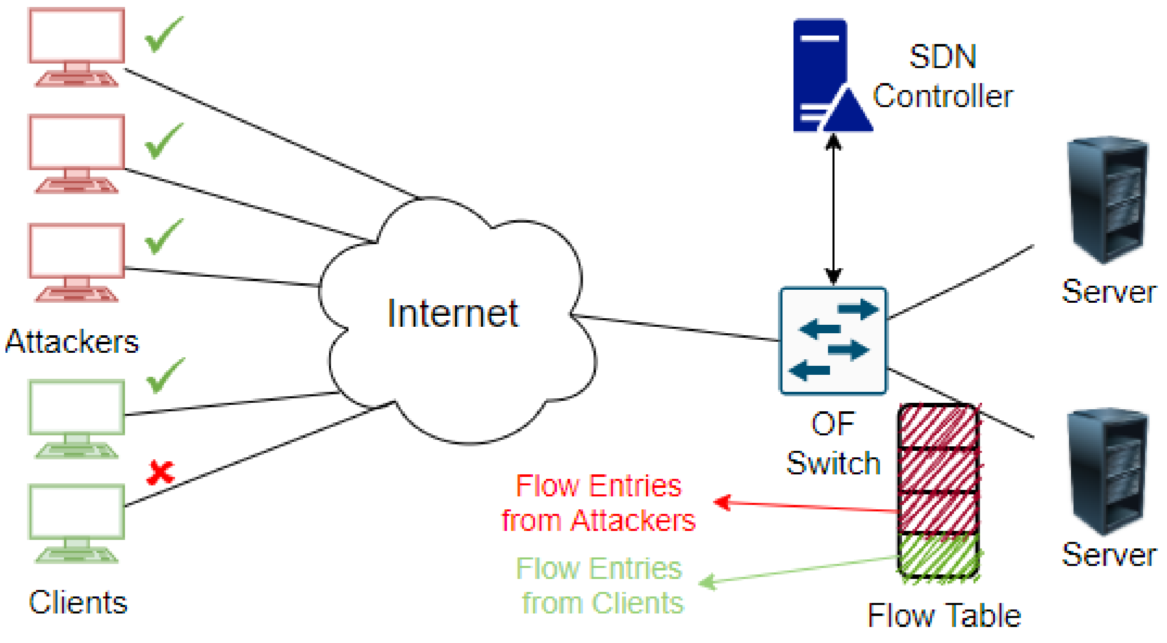

Figure 1.4: Flow Table Overflow Attack

Figure 1.4 shows how a flow table overflow occurs in a network domain with an OpenFlow switch. In normal traffic without any attack, the OpenFlow switch can handle the client traffic with its flow table capacity. An enormous number of attackers also sends requests to the servers in the target domain, in case of a flow table overflow attack. The number of flow entries generated from attacker packets rises so high that the entire flow table is filled with attackers' flow entries. Benign packets (from legitimate clients) are dropped since their corresponding flow entries are not installed. Therefore, the client cannot access the server, and there is also a possibility of an overwhelmed and unresponsive SDN switch and controller. Even without an unresponsive controller, the overall performance of the network decreases [11]. If the switch or controller becomes unresponsive, the whole network is down.

Flow table overflow may occur in a DDoS attack depending on the packet variety. In addition, there are also some other situations in which a flow table overflow might occur. For instance, T. A. Pascoal et al. [12] show an attack called Slow-TCAM (Slow TCAM Exhaustion Attack) that is influenced by slow rate DDoS attacks. In Slow-TCAM attack, attackers keep their connection with the server (on the target domain) by sending packets at a slow rate which can be detected as benign traffic. However, attackers send the traffic from bots that are slightly greater than the flow table capacity in number. Also, the packets are sent at a rate within the idle timeout limits, which can be inferred. This way, attackers ensure that the flow table will be kept full during the attack.

Finally, we can say the flow table overflow attacks cause Denial of Service (DoS) effect for the clients because of the service outages, as well as high response time [9]. Flow table overflow attacks can be prevented simply by having a big enough flow table. However, the cost of an OpenFlow switch increases with the size of the TCAM. Besides, the number of bots in a botnet increases exponentially; thus, we cannot be sure that it is possible to have big enough flow table in the future.

In this work, we aim to mitigate the flow table overflow attacks, regardless the size of the physical networking infrastructure devices in the network domain; which will consequently fix the issues of communication interruption and high delay that stem from high flow table usage. We summarize the following problems from the current flow table overflow attack mechanisms:

**(P1) High dependency on the size of the physical infrastructure**: Number and the capacity of physical network devices that reside in the domain greatly affects mitigation method.

**(P2) Coarse-grained flow entries block legitimate users**: Traditional Access Control List (ACL) compression methods use subnet masking based mechanisms, which limits compression rate. However, more abstract masks are implemented in order to decrease the rule size which results in blocking of legitimate packets.

**(P3) Harsh eviction policies with high probability of dropping legitimate traffic**: Proactive firewall policies are not sufficient to implement fine-grained blocking rules that only filter attackers.

To achieve the main goal of this research and solve the above-mentioned problems, we propose an NFV-based mitigation mechanism for SDN-based environments. More concretely, we propose:

**(S1) Dynamic NFV infrastructure**: A dynamic network topology created with virtual OpenFlow switches takes the burden of physical switches.

**(S2) Fine-grained flow entries distributed among the virtual infrastructure**: Increased overall flow table capacity allows a fine-grained filtering that only blocks attackers.

## 1.3 Organization of this Thesis

The structure of this thesis is as follows. Chapter 1 (this chapter) gives a brief introduction to the scope of this research. Then, Chapter 2 presents related work, and Chapter 3 shows a detailed overview of the proposal, which is then evaluated in Chapter 4. Finally, Chapter 5 concludes this thesis with some final thoughts and future work.

# Chapter 2

# Related Work

## 2.1 Traditional Mechanisms for Table Overflow Mitigation

Flow table overflow is a problem addressed since the early stages of the implementation of SDN and OpenFlow. For instance, one of the early scenarios of table overflow (without an attack) is a Local Area Network (LAN) environment with a significant number of end devices. Flow entries that are originated from the communication between all of the devices may trigger table overflow. Table overflow mitigation techniques can be classified into three main categories [13].

*Compression* methods merge similar flow entries into a single flow entry. However, to be compressed, flow entries need to have at least the same action, which may be challenging in OpenFlow since flow entries might have a variety of actions. Matching fields such as IP addresses can be masked using wildcard rules in the flow entries. For example, two ternary strings, "00*10" and "01010" can be merged as "0**10". Some traditional methods [14] uses the IP field to create CIDR masks on Access Control Lists (ACL). While others [15] uses binary tree to calculate optimized CIDR mask for forwarding flows in LAN environments. However, both methods are time-consuming because of their complexity and do not have enough compression rate for a server that is open to the internet.

*Eviction* methods replace an existing flow entry with a new one when the flow table is already full. Cache replacement algorithms such as First In First Out (FIFO) and

Least Recently Used (LRU) can be directly implemented to some OpenFlow switches. However, implementing these procedures without a controller-based policy, would render the centralized management underused. In SDN, a controller can manage the eviction too, by using the OFPFMFC_ALL_TABLES_FULL message. The controller sends a delete message to the switch, which points out which flow entry to delete. This method is also impractical due to the significant signaling overhead resulting from time-consuming processes such as statistic collection. Another option would be to use the timeouts in the flow entries. Every flow entry has an idle and hard timeouts in its structure. There are some related works that find the best-suited timeout values for the current network state. Nevertheless, all eviction methods suffer from the same drawback, namely they occur when the flow table is already overflowing.

*Distribution* methods split the workload of an SDN switch with an overloaded flow table to the SDN switches with idle space in their flow tables. For instance, B. Yuan et al. [16] propose a QoS aware mitigation mechanism called peer support strategy. Switches help their peers in case of flow table resource outage by using their idle space. The peer support strategy takes the following features into account when choosing which peer to guide the traffic: table fullness, the distance between switches, traffic rate, and connection count with other switches. However, there is no blocking mechanism for attackers in this approach.

K. Bhushan et al. [2] also present a distribution-based mechanism. Their mechanism distributes the flow entries by checking the distance between switches, flow table fullness, and whether the corresponding switch resides on the alternate path. This approach also contains a *Blacklist* collected by previous attacks. Flow entries with wildcard rules are installed to the switches to block attacker packets. In addition, underutilized flow entries in a flow table are removed in case of an attack.

The above-mentioned distribution methods request flow table status by polling in a set period. The polling of the flow tables of every switch on the domain creates a significant CPU load on the SDN Controller, and it takes time to process all those information, which creates a signaling overload. Besides, they also assume that we have enough SDN switches to distribute incoming flow entries generated from incoming packets. Therefore, these mechanisms are not suited to mitigate an attack with great numbers that overwhelm the whole domain resources or the domains with limited resources such as small or medium-sized private networks. Regardless, distribution methods have a great potential for mitigating flow table overflow attacks.

## 2.2 SDN-based Flow Table Overflow Mitigation

This section presents related work of mitigation mechanisms that uses SDN. For instance, P. T. Duy et al. [17] create a mechanism that replaces forwarding flows originated from the attackers with dropping flows. Also, they decrease the timeouts with the increase of the flow entries. Finally, if heavy network traffic occurs in a short period, table-miss flow entry is removed which results in no addition of new flow entries. This work increases the controller load by decreasing the timeouts. However, it does not ensure legitimate traffic will be handled under an attack.

STAR [9] uses eviction and alternate paths in order to maximize flow table utilization and prevent flow table overflow. Firstly, a path-set database is pre-generated between source-destination pairs. Also, an LRU eviction algorithm is implemented into the switches. When a new flow entry enters the system, the flow entry is rejected if there is a fully utilized switch on every possible path. The cost of the possible routes is calculated according to flow table utilization and path length, and the flow entries are installed to the path with the lowest cost. STAR aims to maximize the flow table utilization by creating possible routes on the network. However, not blocking or forwarding attacker packets may result in constant packet-in messages that consume controller resources. In addition, during an attack, the flow tables would be filled with attacker's flow entries, which means incoming new legitimate packets are dropped and resulting in DoS.

## 2.3 NFV-based Attack Mitigation Mechanisms

NFV realizes network functions on commodity hardware which allows network devices to be virtualized too. In that regard, there are some interesting related work on using this approach. For instance, VGuard [18] uses NFV to create virtual switches on an

NFV Infrastructure to create two virtual tunnels. The Source IP of every incoming packet is assigned a priority value, which shows the likelihood of being a legitimate packet given by an external source. A relatively small number of packets with higher priority use the High Priority Tunnel, which has ensured QoS. Packets with a higher probability of originating from an attacker have to compete for the Low Priority Tunnel resources.

Furthermore, VFence [19] proposes NFV based defense mechanism against SYN attacks. VFence uses NFV for creating every network device on the domain and they assign a commodity server per network device. They first create a dispatcher, which is connected to the outside of the domain; thus, the Internet. This dispatcher also performs load balancing on agents placed in parallel. These agents check whether the incoming packets are spoofed, if not, then add the source IP of non-spoofed packets to a whitelist. Whitelisted packets pass directly to the domain; hence, the SYN attack that aims to cripple the servers fails. Nonetheless, the structure of VFence is not open to apply dynamic features of NFV, such as adding more agents or change the placement of agents.

# Chapter 3

# NFVGuard: An NFV-based Table Overflow Mitigation for Software Defined Networks

## 3.1 Overview of the Proposal

This chapter describes our proposed mechanism to mitigate flow table overflow attacks in SDN, called NFVGuard. The overall structure is shown in Figure 3.1. As observed, legitimate users and attacker can access the servers through an *Edge Switch* located on the access gateway to the target domain.

NFVGuard is composed by two elements, a Controller (NFV Controller) and a Virtual Honeypot, which are responsible for protecting SDN switches in the target domain against flow table overflow attacks using the following process. In case a packed coming from the outside of the target domain does not match any of the flow entries of the Edge Switch, the packet is sent to the NFVGuard Controller through a packet-in message. The controller processes these messages and installs corresponding flow entries to the switches. In order to filter the attacker traffic, all of the incoming traffic from outside of the target domain is directed to a *Virtual Honeypot*. These main components are discussed in detail in the following subsections.

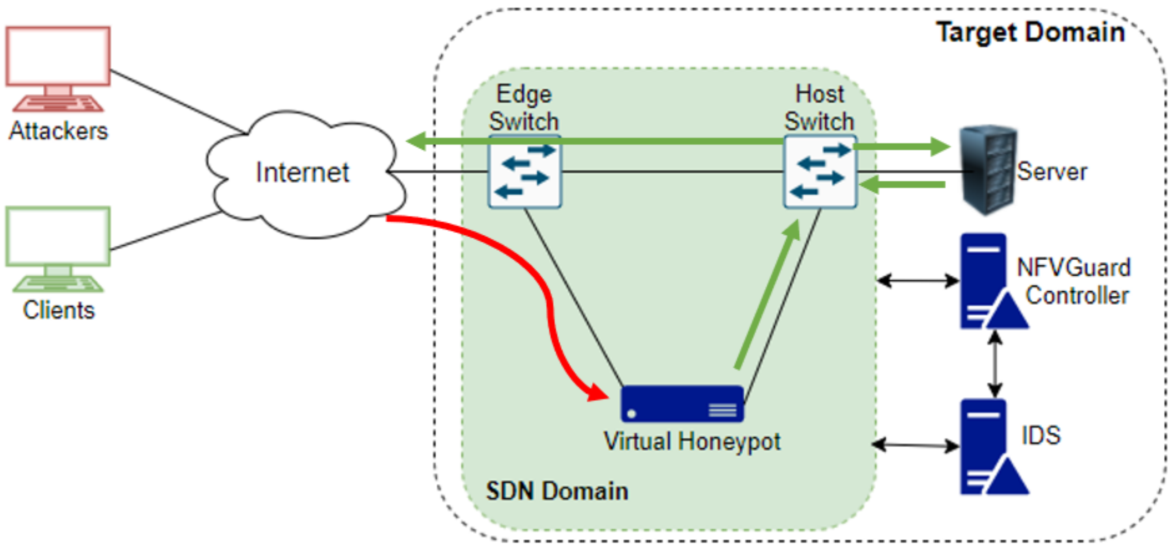

Figure 3.1: Proposed NFVGuard Deployment

## 3.2 Virtual Honeypot

Current flow table overflow mitigation mechanisms cannot cope with the attacks because of their reliance on the limited physical infrastructure. Thus, we use the Virtual Honeypot to mitigate the attack by adjusting the Virtual topology. Using these virtual devices arranged in a specific manner increases the capacity of the Virtual Honeypot; allowing fine-grained flow entries, which are installed to block attackers as precisely as possible. The name Virtual Honeypot comes from the fact that this infrastructure lures attackers and blocking them accordingly. It can be implemented on a Network Function Virtualization Infrastructure (NFVI). For instance, in the current implementation, we use OpenVSwitch (OVS) as it can be easily implemented on any commodity hardware. However, the Virtual Honeypot could be implemented in any other form of NFV such as the container-based Docker. As mentioned in Chapter 2, distribution

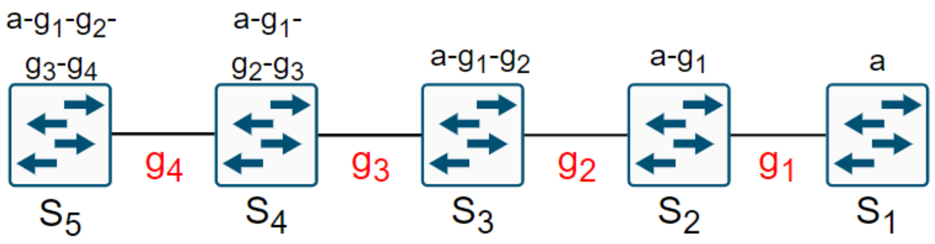
$a\text{-}g_1\text{-}g_2\text{-}g_3\text{-}g_4$
$a\text{-}g_1\text{-}g_2\text{-}g_3$
$a\text{-}g_1\text{-}g_2$
$a\text{-}g_1$
$a$
$g_4$
$g_3$
$g_2$
$g_1$
$S_5$
$S_4$
$S_3$
$S_2$
$S_1$

Figure 3.2: Worst-case Scenario for Mitigation

approaches use existing physical switches in the domain to install fine-grained flow entries. However, these devices initiate slower than virtual switches; even if the devices start booting as soon as the attack is detected, the rule installation and other processes will take a long time. Duration is the essence of a flow table overflow attack; if we are too late, the attack might be successful.

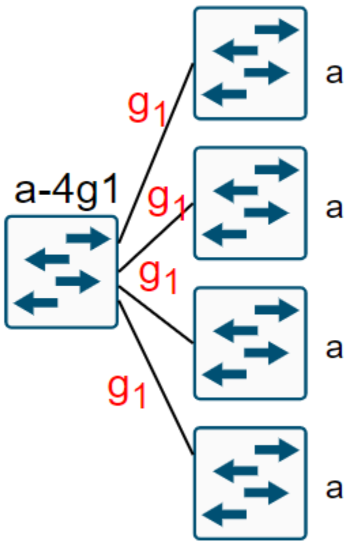

Figure 3.3: An Advantageous Virtual Honeypot Topology for Mitigation

Another handicap of distribution approaches is the static topology of the domain. We cannot change the switch locations, and adding additional switches into the domain is costly. However, there are some topologies that will handle the table overflow attacks more easily. For example, consider a physical topology as in Figure 3.2, in this serially-connected topology, there is not many possibilities to mitigate a table overflow attack, as all the traffic need to traverse the switches. We could even say that this is worst-case scenario for flow table overflow mitigation topology. Each switch has to carry guiding flow entries ($g$) of the next switch in line. For example, S3 has to carry both $g_1$ and $g_2$. This repetition decreases the available flow entry capacity ($tc$).

In contrast, as observed in Figure 3.3, there are some other topologies that might offer a more favorable environment to mitigate the attack. In this topology, a switch distributes the flow entries to parallelly connected switches, therefore avoiding flow entry repetition.

To estimate the total flow table capacity, we use the holding time ($h$), which refers to the duration until the flow table is full in case of an attack, with a given attack rate

Table 3.1: Abbreviations and Terms

| | |
|---|---|
| $g$ | Guiding flow entries |
| $tc$ | Available total flow entry capacity |
| $h$ | Holding time |
| $r_a$ | Attack rate |
| $d$ | Attack duration |
| $C_n$ | Total legitimate flow entry count |
| $r$ | New flow entry rate |
| $v$ | Flow entry removal rate |
| $t_s$ | Time spent in the system |
| $n_h$ | Total host count |
| $C_s$ | Legitimate flow entry count per switch |
| $N$ | Switch count in domain |
| $a$ | Available flow entry count |
| $C$ | Flow entry capacity of a switch |
| $h_n$ | Holding time of naive approach |
| $c$ | Flow entry compression rate |
| $h_d$ | Holding time of distribution approaches |
| $n_{rs}$ | Required switches to mitigate |
| $n_s$ | Switch count to start in dynamic approach |
| $s$ | Safety value |
| $s_{threshold}$ | Percentage to add a new switch |
| $p_t$ | Packet-in rate threshold |
| $T_{emrg}$ | Emergency state duration |
| $fe_{limit}$ | Flow entry limit per switch |
| $mw$ | Mask width for compression |
| $mw_t$ | Mask width threshold for merging |
| $d_t$ | Ternary Hamming Distance threshold |

($r_a$) and duration ($d$).

Table 3.1 summarizes the abbreviations and terms used in this section.

Initially, the total flow entry count on all switches ($C_n$) is calculated as shown in Equation 3.1. As observed, the total flow entry installation ratio is determined by the average flow entry installation per host ($r$) and average flow entry removal per host ($v$). Then, we multiply this ratio with the time spent in the system ($t_s$) to get average number of flow entries for a host, at any time. Finally, we can get average total flow entry count ($C_n$) originating from legitimate users by adding all.

$$C_n = \sum_{i=1}^{n_h} \frac{r}{v}.t_s \tag{3.1}$$

Next, we can calculate the legitimate flow entry count per switch ($C_s$) as shown in Equation 3.2.

$$C_s = \frac{C_n}{N} \tag{3.2}$$

For instance, in the case where there is no defense mechanism to mitigate the attack (i.e., naive approach), the holding time ($h_n$) can be calculated as shown in Equation 3.3. Note that, since there is no defense mechanism, first switch in the line is filled with all flow entries.

$$tc = C - C_s$$

$$h_n = \frac{tc}{r_a} = \frac{C - C_s}{r_a} \tag{3.3}$$

Then, to calculate the guiding flow entry counts ($g$), we first need to calculate the available space in a specific switch ($a$), as shown in Equation 3.4.

$$a = C - C_s \tag{3.4}$$

Since, we compress the flow entries with a compression rate ($c$) that fills all available space in the switch, as shown in Equation 3.5.

$$g = ac \tag{3.5}$$

For instance, if we apply these formulas in Figure 3.2, the resulting available flow entry count ($a$) and the total available flow entry count ($tc$) would be as follows:

$$
\begin{aligned}
&a_1 = a_1 && g_1 = a_1.c \\
&a_2 = a_1 - g_1 = a_1(1-c) && g_2 = a_2.c \\
&a_3 = a_1 - g_1 - g_2 = a_1(1-c)^2 && g_3 = a_3.c \\
&a_2 = a_1 - g_1 - g_2 - g_3 = a_5(1-c)^3 && g_2 = a_2.c \\
&a_1 = a_1 - g_1 - g_2 - g_3 - g_2 = a_1(1-c)^4 \\
&tc = a_1 + a_2 + a_3 + a_4 + a_5 \\
&tc = a_1 + a_1(1-c) + a_1(1-c)^2 + a_1(1-c)^3 + a_1(1-c)^4
\end{aligned}
$$

Then, we can use Equation 3.6 to calculate total available flow entry count.

$$\sum_{i=1}^{N-1} r^n = \frac{1-r^N}{1-r} \tag{3.6}$$

As observed, the number of switches in domain ($N$) is used to formulate total available capacity.

$$
\begin{aligned}
tc &= a_1 \frac{1-(1-c)^N}{1-(1-c)} \\
tc &= (C - C_s)\frac{1-(1-c)^N}{c}
\end{aligned}
$$

Finally, we can calculate the holding time for distribution methods ($h_d$), as follows.

$$h_d = \frac{tc}{r_a} = (C - C_s)\frac{1-(1-c)^N}{c.r_a} \tag{3.7}$$

Also, we can now calculate the Virtual Honeypot holding time ($h_{vh}$). Since we have $N-1$ parallel switches and the guiding flow entries ($g$) values are the same, we can directly use Equation 3.4 and 3.5, as follows.

$$
\begin{aligned}
tc &= N.a - (N-1)g \\
tc &= (C - C_s)(N - N.c + c) \\
h_{vh} &= \frac{(C - C_s)(N - N.c + c)}{r_a}
\end{aligned} \tag{3.8}
$$

In order to mitigate a table overflow attack, our flow tables should not be full regardless of the scale of the attack. Therefore, the holding time ($h$) have to be bigger than the attack duration ($d$).

$$h > d \tag{3.9}$$

We can obtain the number of required switches to mitigate ($n_{rs}$) by replacing $N$ in Equation 3.8.

$$\frac{(C - C_s)(n_{rs} - n_{rs}.c + c)}{r_a} > d \tag{3.10}$$

To summarize, we can calculate the necessary number of switches to mitigate based on a given attack rate and duration as follows.

$$n_{rs} > \frac{d.r_a - c(C - C_s)}{(1 - c)(C - C_s)} \tag{3.11}$$

It is worth noting that the Virtual Honeypot might be of two types: a static, and dynamic.

### 3.2.1 Static Approach

The static approach on the Virtual Honeypot aims to increase the overall flow table capacity of the domain a single time (i.e., when the attack is detected). Figure 3.4 illustrates the structure of the proposed Virtual Honeypot using a static approach. The number of created virtual switches is constant, and they are instantiated directly with the installation of the Virtual Honeypot. We can decide the number of virtual switches using Equation 3.11 for an anticipated attack. The Edge Switch guides all of the incoming traffic to the Virtual switches in the Virtual Honeypot. Once the virtual switches are created, the NFVGuard Controller installs fine-grained flow entries into these switches. Note that, the flow entries forward the legitimate traffic to the physical switch (host switch) and block attackers on the virtual switches.

### 3.2.2 Dynamic Approach

In the Dynamic version, the Virtual Honeypot has a three-level structure, as illustrated in Figure 3.5. The top and bottom levels connect the Virtual Honeypot to the physical switches. In order to relieve the burden from the Edge Switch, the top-level switch distributes the traffic to the middle-level switches. Although there is only one virtual

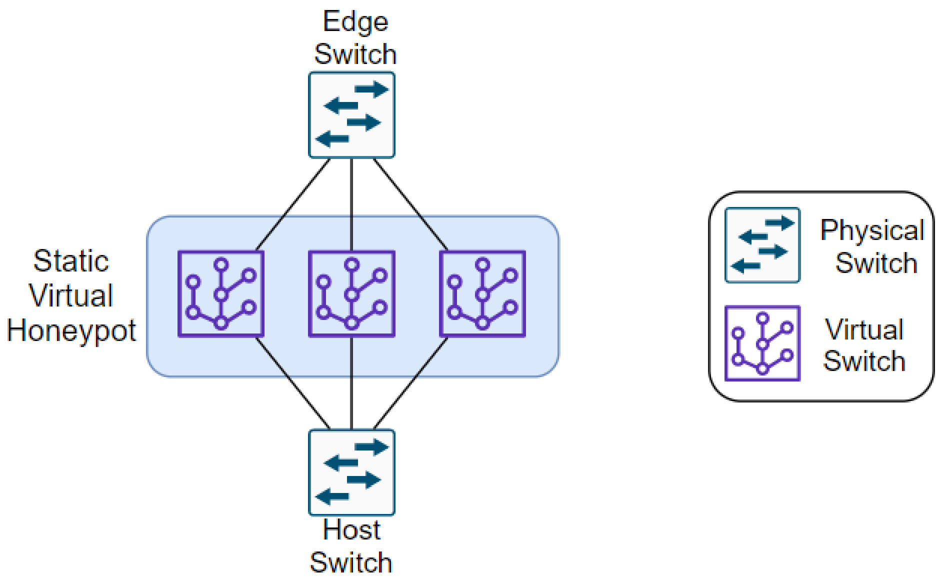

Figure 3.4: Static Virtual Honeypot Structure

switch on the top level in diagram, this can be adjusted according to traffic volume and diversity of the attack.

The middle level is in charge of applying fine-grained filtering policies. The dynamic features of NFV and SDN are applied by adding/removing virtual switches at run-time. The number of virtual switches is increased according to attack density and new virtual switches are added as soon as a threshold ($s_{threshold}$) is passed to prevent flow table overflow on the middle level. Consequently, only legitimate traffic passes to the target domain's physical network devices.

The number of virtual switches on middle level ($n_s$) can be calculated easily by using total flow entry count of legitimate traffic ($C_n$) by Equation 3.12, in only legitimate traffic.

$$n_s > \frac{C_n}{C} \tag{3.12}$$

The threshold ($s_{threshold}$) is determined by the ratio of existing middle level switches ($n_s$) and how many switches we need to mitigate the attack ($n_{rs}$), which is calculated in the previous section by Equation 3.11. Also, we multiply this value by a safety value ($s$) since we do not want to fill the tables.

$$s_{threshold} = \frac{n_s}{n_{rs} - n_s}.s.100 \tag{3.13}$$

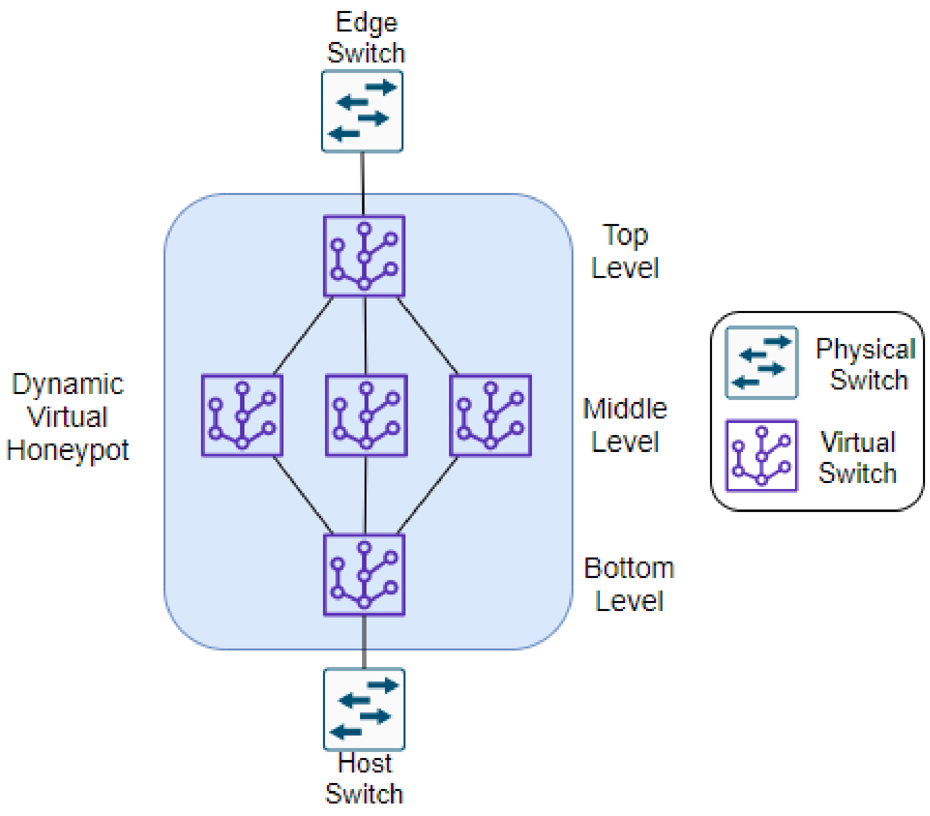

Figure 3.5: Dynamic Virtual Honeypot Structure

## 3.3 NFVGuard Controller

The other important component of the proposal is the SDN controller called the *NFV-Guard Controller*. NFVGuard Controller is responsible for standard networking operations such as traffic handling and state monitoring of the domain. However, it also has the role of NFV Orchestrator for handling the Virtual Honeypot topology and resources.

NFVGuard Controller consists of four main components and three databases. The *Policies database* includes routines that enable traffic handling in the domain, such as routing and firewalling. Then, the *Flow Database* includes information of flow entries that guides the traffic to Virtual Honeypot's middle layer. Finally, the *Switch Database* entries are switch information such as datapath, ID, the number of flow entries in a switch, and the out port from the guiding switch. All of the databases are implemented as a simple list that resides in the RAM. We do not use any sophisticated database system in the implementation since a prompt response is expected from the controller. Figure 3.6 shows the relationship between the NFVGuard Controller modules and the interactions between NFVGuard Controller and other physical structures whose functionality are explained in the following subsections.

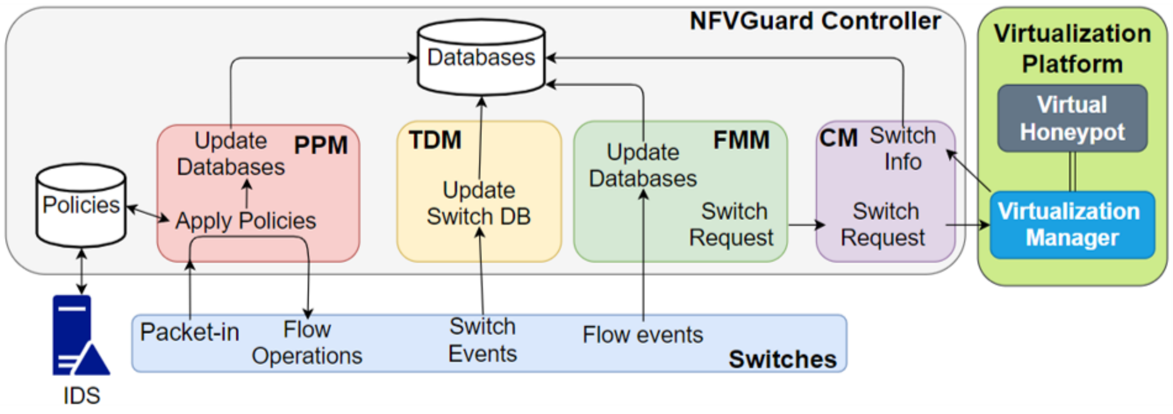

Figure 3.6: NFVGuard Controller Modules

### 3.3.1 Communication Module (CM)

The Communication Module (CM) provides a channel to NFVGuard Controller for interactions with Virtualization Platform that creates virtual switches (i.e., Virtual

Honeypot). The module forwards requests to the Virtualization Platform and processes the response, which is the new information. The CM updates the Switch Database with this new information. In the current implementation, the communication is done via a simple TCP client that sends/receives compressed (pickled) binary data.

### 3.3.2 Flow Monitoring Module (FMM)

This module oversees updates of Flow and Switch Databases. In addition, it also creates requests for switch addition/removal in the dynamic approach. This module can be implemented using two versions, polling type in the static approach and asynchronous type in the dynamic approach.

In the case of the polling type, it collects flow table information from the OpenFlow switches. Then, this module also deletes the flow entries that are removed from the flow table. This process is done by comparing all entries in the Flow Database and the response. Finally, it also calculates the flow entry counts in each switch to update the Switch Database. Note that this method has $O(n^2)$ of time complexity.

Flow entries with the OFPFF_SEND_FLOW_REM flag trigger the asynchronous OFPT_FLOW_REMOVED message when it is removed from the switch. The switches with the flow table overflow possibility have all of their flow entries set the OFPFF_SEND_FLOW_REM flag in the dynamic approach. Databases are updated event-based, which decreases the time complexity to $O(n)$ which is a significant improvement compared to polling type.

Based on the information collected by this module, NFVController can add/remove switches, as explained in the following sub sections.

#### Switch Addition

To add a switch, FMM checks the occupancy rates of all switches every $T$ seconds. When the occupancy rates of all switches pass a threshold ($s_{threshold}$), a request for a new switch is sent to Communication Module (CM). Details on how to calculate $s_{threshold}$ is given at Section 3.2.2.

**Switch Removal**

Since the removal process depends on the duration of the attack, to simplify the model we can assume the network traffic with a Poisson Distribution [20]. Events, which are client accesses, are independent, and two events from the same source cannot occur simultaneously. We use the average Packet-in per second as the Poisson Distribution parameter ($\lambda$). An abnormal number of Packet-in messages may arrive in case of an attack or a high load of legitimate traffic in specific cases. In both cases, abnormal traffic load might reoccur in a short period again. Therefore, removing underutilized virtual switches right away is not a wise solution.

$$\sum_{i=0}^{k} e^{-}\lambda \frac{\lambda^i}{i!} \tag{3.14}$$

A Packet-in rate threshold ($p_t$) is calculated using the Cumulative Distribution Function (CDF) 3.14, with a 99% probability. When the Packet-in rate passes the threshold, "Emergency State" is declared on the controller. This emergency state is not lifted until the normal packet-in rate continues for a time ($T_{emrg}$); thus, no flow removal happens. Switches with lower occupancy than $s_{threshold}$ are removed if the controller is not in the emergency state by sending a switch removal request to the Virtualization Platform.

### 3.3.3 Topology Discovery Module (TDM)

This module checks the switches that enter/leave to this domain and update the Switch Database, including the virtual switches in the Virtual Honeypot.

### 3.3.4 Packet-in Processing Module (PPM)

PPM is the core of our NFVGuard mechanism. This module creates wildcard rules to guide and distribute the flow entries between the Virtual Honeypot switches. PPM is explained in the next section thoroughly.

## 3.4 Filtering and Flow Distribution

The traffic is filtered by applying fine-grained policies on the Virtual Honeypot. As explained in the preceding section, the Static approach has a single level, therefore, the packets are filtered in the whole Virtual Honeypot. However, in the dynamic approach, the Virtual Honeypot has a multilevel structure and the filtering is conducted on the middle level. The controller creates wildcard rules to guide the traffic to the filtering layer of Virtual Honeypot. However, a flow distribution method is necessary to prevent flow table overflow on the middle-level switches too.

### 3.4.1 Switch Selection

Wildcards rules are created by changing the mask of an OpenFlow match field. We use the source IP of incoming packets to compress the flow entry rules. The Hamming distance between two strings is the total number of different elements. We create a term we call Ternary Hamming Distance (THD) inspired by [21]. IP is a ternary string with do-not-care bits (i.e., *) as well as 0 and 1 bits. If the bit pair has either a do-not-care bit or they are the same, Ternary Hamming Distance is 0; otherwise, it is 1. For example, THD is 2 for given strings "10*001" and "00*101".

THD is shown in Algorithm 1 of Appendix A-1. An IP (N) and its mask (K) that resides in the Flow Database is compared to a newcomer packet IP (M). The merged version of both IPs is also calculated together with the THD in order to remove redundancy.

### 3.4.2 Flow Distribution

PPM picks a switch to forward the traffic with created wildcard rules. The switch with the lowest occupancy is selected as the next hop in the static approach. On the other hand, in the dynamic approach, more elaborate policies can be applied. For instance, we used round-robin for rule distribution in the current implementation. Flow entry admission is stopped to the switches that pass the flow entry limit ($fe_{limit}$). Equation 3.15 shows that $fe_{limit}$ is the safe flow table capacity which depends on the before-mentioned capacity ($C$) of the flow table and the aforementioned safety value ($s$).

$$fe_{limit} = C \times s \tag{3.15}$$

The reason we are using the safety value twice, both in Equation 3.13 and Equation 3.15 , is because we do not forward new flow entries to switches which pass $fe_{limit}$ because compressed guiding flows might carry too much flow entries than intended. We use $fe_{limit}$ as a fail-safe for flow table overflow that might stem from the compression.

### 3.4.3 Packet-in Processing

Packet-in processing is the most critical part of this work. We explain the Packet-in processing on the guiding switch first. The guiding switch is the Edge Switch in the static approach and the top-level switch in the dynamic approach.

Algorithm 2 of Appendix A-2 shows the proposed Packet-in processing mechanism for the guiding switch ($GS$). First, the switch to forward is selected as explained in the previous section, then action for forwarding to the chosen switch is created. Packet-in message ($M$) is compared to existing flow entries in the Flow Database ($flowDB$). THD between an existing $flowDB$ entry and the source IP of the packet ($M.IP$) is calculated. We introduce mask width ($mw$) and mask width threshold ($mw_t$) here. Mask width is the number of zeros in the bitmask, do-no-care bits in another word. The mask width is limited because making wide wildcard rules makes flow distribution ineffective.

THD will be zero if the flow entry is deleted from the switch but still exist in the Flow Database. We update the Flow Database and install the flow entry with a match field that only has the source IP of the packet. If THD is lower than the threshold ($d_t$) and mask width is lower than its threshold, we remove the existing flow entry from the switch and database. Then, the new flow entry with the wildcard rule is installed into the switch, and the database is updated. At last, if none of the flow entries match the source IP of the new incoming packet, a new flow entry is installed without any wildcards.

The filtering process is implemented by applying fine-grained policies to the filtering level switch packet-in messages i.e, all virtual switches in the static approach, and middle level in the dynamic approach). The incoming packets can be forwarded, dropped, or altered on this level. Fine-grained policies can be supported by an Intrusion Detection System (IDS) or machine learning techniques [22] to decide which packets to block. Since this work focuses on the pragmatic implementation of a flow table overflow mitigation mechanism rather attack detection, we assume that we already have a blacklist provided by an external source.

# Chapter 4

# Experiment

## 4.1 Experiment Outline

In our first two experiments, we emulate both the physical network infrastructure and the Virtual Honeypot using Mininet [23]. In these experiments, it is assumed all switches except the ones in Virtual Honeypot are physical switches. Also, RYU [24] is used to implement the NFV Controller modules we mention in Chapter 3.

We use a simple approach that does not do mitigation as the benchmark approach (i.e., naive approach). The naive approach has the same "physical" topology as the proposed method in the experiments, except the Virtual Honeypot. Moreover, its flow entry install policy is the default one (i.e., once an unknown packet arrives to the edge switch, it is sent to the controller and the corresponding flow entry is installed).

We use metrics such as Round Trip Time (RTT) and CPU usage of the controller to assess the performance. In addition, we use flow table occupancy as it is stated in Equation 4.1.

$$Occupancy = \frac{Flow\ entry\ count}{Capacity} \tag{4.1}$$

We also conducted a third experiment, wherein we made a theoretical comparison with existing distribution approaches.

## 4.2 Experiment 1: Performance Test Using the Static Approach

### 4.2.1 Outline

The purpose of Experiment 1 is to test the feasibility of the **static approach** and overall confirmation of our proposal. Figure 4.1 shows the experiment's topology, and it is implemented on an Ubuntu virtual machine with four cores of CPU and 4 GB RAM.

Table 4.1: Experiment 1 Metrics

| Attackers | Clients | $d_t$ | $mw_t$ | $C$ |
|---|---|---|---|---|
| 60 | 15 | 2 | 3 | 300 |

There are two hosts set as Web Servers in the network domain on the right-hand side of the topology, which are the targets of the attack. The Edge Switch connects outside domain to the target domain, and it is the main target of the table overflow attacks. The attacker domain on the left-hand side of the topology has two hosts. Both of the hosts create HTTP Get requests in order to simulate the scenario. The attacker simulates 60 attackers, and the client host simulates 15 clients with unique IP addresses. The attack duration is 120 seconds for this experiment. Finally, the Virtual Honeypot has three virtual switches.

### 4.2.2 Results

Figure 4.2 illustrates the results obtained using the benchmark approaches. As observed, the flow table overflow succeeds in the naive approach, while in the case of NFVGuard the attack was successfully mitigated. Note that, although the wildcard creation metrics $d_t$ and $mw_t$ have small values, there is only 50% of flow table usage. The occupancy is lower on non-attack times too, which is an expected outcome due to the compression applied in our method.

However, regardless of the successful mitigation, the controller CPU usage was high even in NFVGuard. After carefully inspecting the modules, we found out that

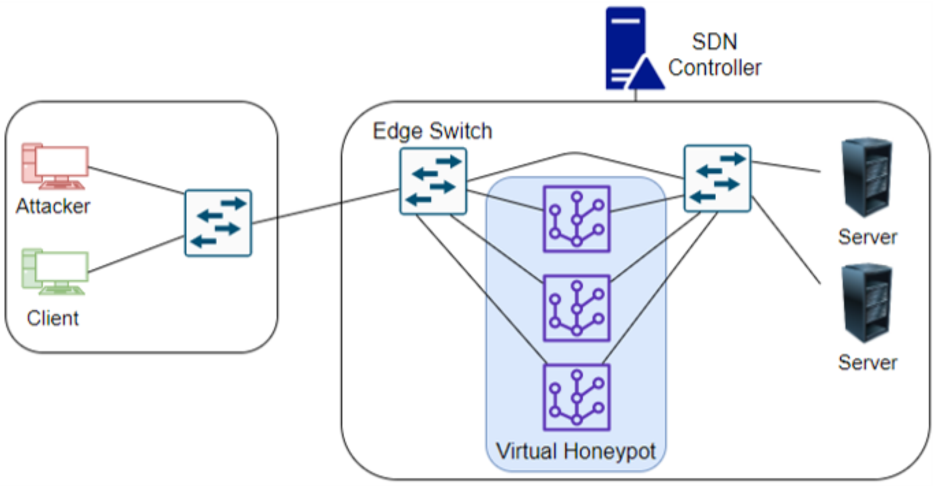

Figure 4.1: Topology of Experiment 1

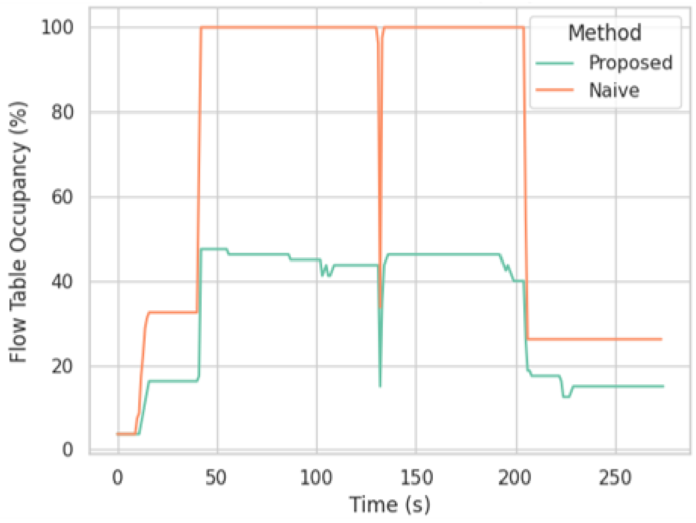

Figure 4.2: Flow Table Occupancy on the Edge Switch

polling of the switch statistics in a short period was the culprit. Therefore, we set the asynchronous method on the Flow Monitoring Module of the controller for the second experiment.

## 4.3 Experiment 2: Performance Test Using the Dynamic Approach

### 4.3.1 Outline

Experiment 2 aims to test the feasibility of the **dynamic approach** of our proposal. As mentioned before, the dynamic approach adds virtual switches to the Virtual Honeypot on-the-fly during the attack. Figure 4.3 shows the topology used for this experiment. As observed, we used two different domains with their corresponding controller. The network simulator and the SDN controller of each domain run on a Ubuntu virtual machine separately. Moreover, the network domains are connected via a GRE Tunnel.

Table 4.2: Experiment 2 Metrics

| Attackers | Clients | $d_t$ | $mw_t$ | $C$ | $s$ | $s_{threshold}$ |
|---|---|---|---|---|---|---|
| 450 | 100 | 3 | 4 | 1000 | 0.7 | 10 |

There are 550 hosts in the client domain in this experiment, where 450 are attackers, and 100 are legitimate clients. We use iperf [25] for creating the data traffic. Note that, the data traffic of Experiment 1 is more realistic, but iperf allows us to easily measure RTT and other metrics. Moreover, we configured two hosts on the target domain as the iperf servers. Every client and attacker sends data through iperf client. The attack starts approximately 60 seconds from the beginning of the experiment and lasts 240 seconds.

Virtual Honeypot is placed right after the Edge Switch and starts with two virtual switches as default. We also increase $d_t$ and $mw_t$ by one in this experiment since the network scale is more extensive. In addition, we set $s_{threshold}$ as 10%, which is calculated with Equation 3.13.

### 4.3.2 Results

We compare Edge Switch in the naive approach and the Top Level Switch in the NFVGuard since they are the most vulnerable switch for table overflow attacks. Figure

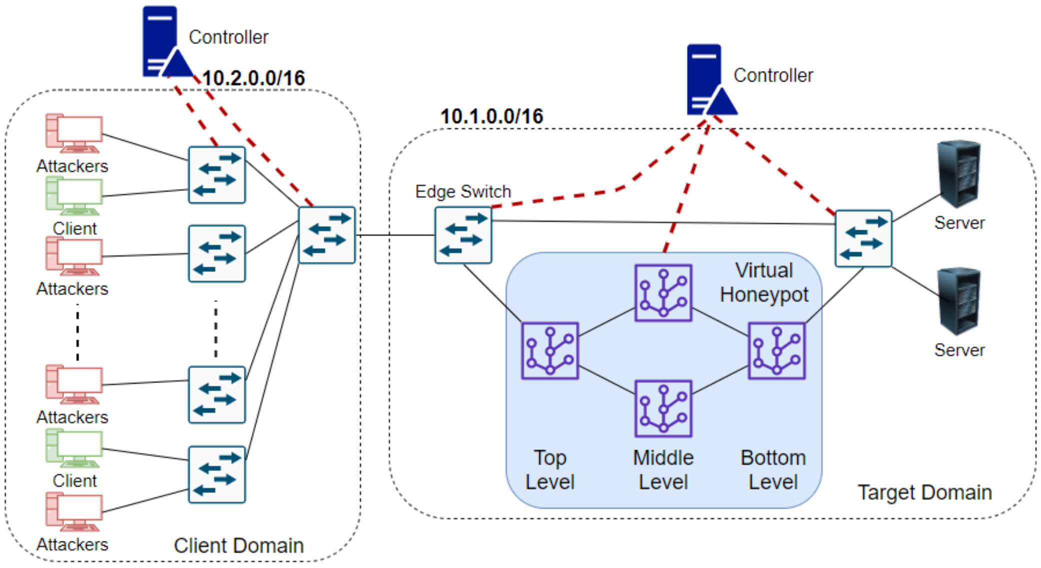

Figure 4.3: Topology of Experiment 2

4.4 shows the flow table occupancy of these two vulnerable switches. As observed, a Flow table overflow happens on the Edge Switch of the naive approach within 30 seconds after the attack. On the other hand, top level switch of our proposed dynamic approach reaches only 30% flow table occupancy. Therefore, we can conclude that flow table overflow is successfully mitigated in our proposed approach. Furthermore, the flow table usage is mostly occupied by legitimate traffic is nearly 50% lower as it is in Experiment 1.

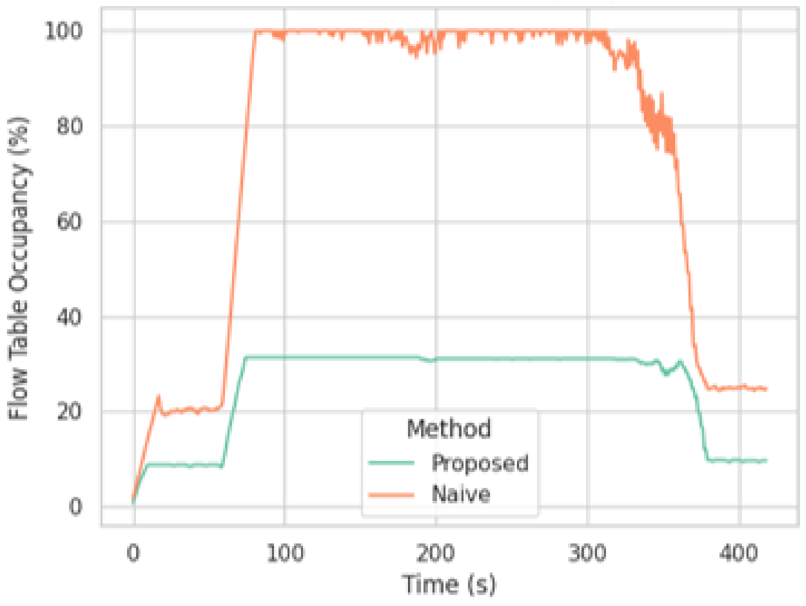

Figure 4.4: Flow Table Occupancy on Vulnerable Switches

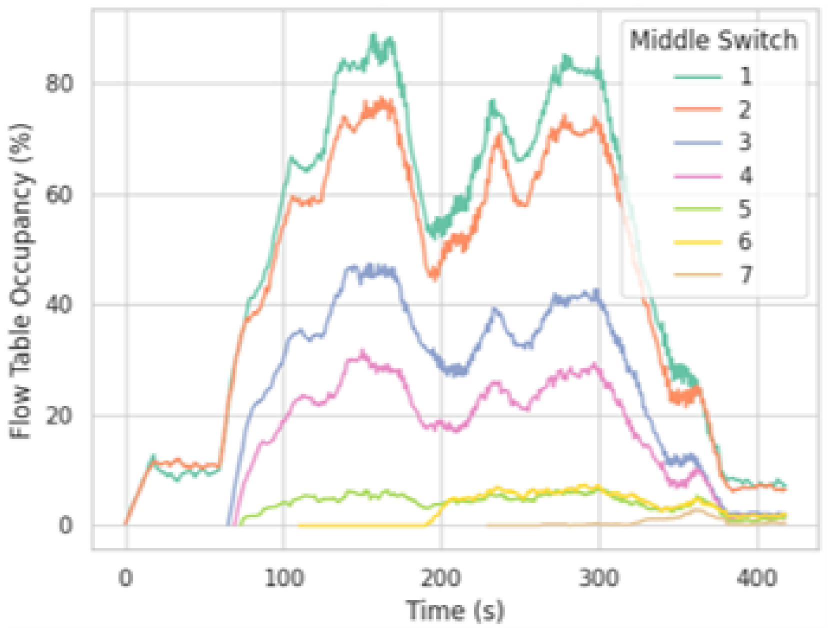

Figure 4.5: Middle Level Flow Table Occupancy

Since the middle level switches of Virtual Honeypot contains fine-grained flow entries; there is a higher chance of flow table overflow at this level. We prevent this by setting the threshold as mentioned before. Note that, within 20 seconds after the beginning of the attack, three virtual switches are added to the middle level as it can be seen from Figure 4.5. These switches take the burden of switches 1 and 2; later on, other two switches added during the attack, just before the peaks, that helps newly added switches too.

In order to evaluate service delay, we get the RTT between a client and a server. As illustrated in Figure 4.6, the RTT value of the proposed method increases at various parts, such as the beginning and peak points of the attack. However, the proposed method has no service outages, unlike the naive approach. The naive approach has an irregular service coverage which decreases almost half the attack without connection.

Concerning the CPU usage of the controller, we can say that we improved the CPU load of polling by using asynchronous updating on Flow Monitoring Module. Figure 4.7 shows the CPU usage of the controller. There is a peak in CPU usage at the beginning of the attack since complex processing. However, this value is within the expected bounds. Also, we observe CPU usage is similar to the naive approach when there is no attack.

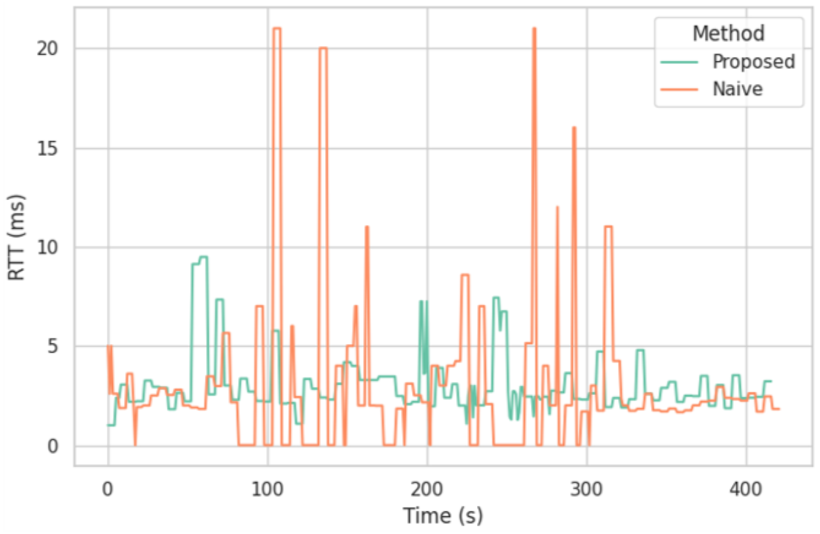

Figure 4.6: RTT of Clients

## 4.4 Experiment 3: Theoretical Holding Time

The aim of Experiment 3 is to compare NFVGuard mechanism with Virtual Honeypot and distribution approaches, [2] and [16], on physical topologies as well as the naive approach. We use the holding time as our result metric, as done in [2,16]. The topology of naive approach and the distribution methods is shown in Figure 4.8.

### 4.4.1 Outline

We consider a topology with serially connected switches as the worst-case scenario for distribution methods since we cannot change the physical topology. In this experiment, we started the number of switches with three and progressively increased up to eight hundred. Again, they are serially connected.

The formula for holding time and how they are derived were explained in Section 3.2. For instance, the holding time of the naive approach ($h_n$) and holding time of distribution methods ($h_d$) are as follows:

$$h_n = \frac{tc}{r_a} = \frac{C - C_s}{r_a}$$

$$h_d = \frac{tc}{r_a} = (C - C_s)\frac{1 - (1 - c)^N}{c.r_a}$$

NFVGuard uses the Virtual Honeypot to create an advantageous topology using virtual switches. Figure 4.9 show the Virtual Honeypot Topology. $S_1$ is used to guide

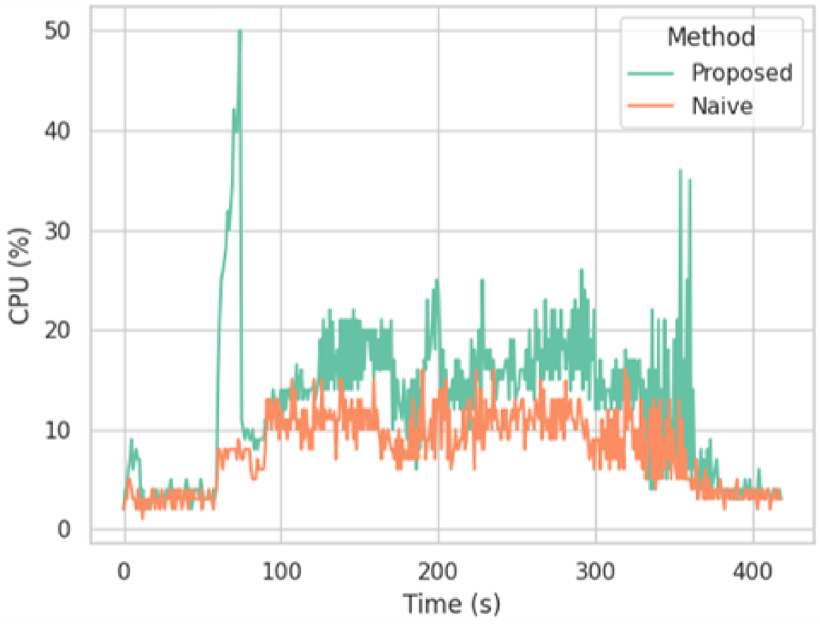

Figure 4.7: Controller CPU

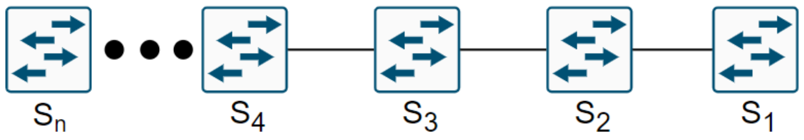

Figure 4.8: Naive and Distribution Topology

flow entries into the switches. We set switches after $S_1$ in parallel. For example, if we have 25 virtual switches in the Virtual Honeypot, there are 24 parallel switches.

We use the formula we derived in 3.2 for calculating holding time for Virtual Honeypot ($h_{vh}$). We can use this formula for both static and dynamic approaches of NFVGuard Mechanism. In case of the static approach, the number of virtual switches would be $N$. Dynamic approach starts with several virtual switches. However, the number of maximum virtual switches can be used in Virtual Honeypot is limited with the capabilities of the NFV Infrastructure. In this experiment, we assume NFVI can only support $N$ virtual switches.

$$h_{vh} = \frac{(C - C_s)(N - N.c + c)}{r_a}$$

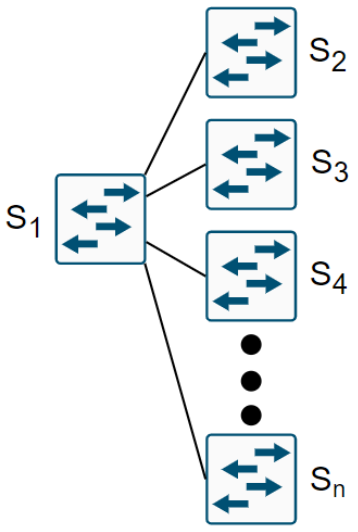

Figure 4.9: Virtual Honeypot Topology

### 4.4.2 Results

Firstly, we measured the effect of attack rate on holding time while keeping the same number of physical and virtual switches. In this part, both physical topology and Virtual Honeypot have 16 and 755 switches. As observed in Figure 4.10 and Figure 4.11, NFVGuard has better holding times than the distribution and naive approaches. Distribution approaches lose their effectiveness with increasing the number of switches. Moreover, the naive approach has an evident difference since it does not take any defensive measures, therefore, the edge switch gets overflowed easily. Regardless of the attack rate, no significant updates can be seen in naive approach holding time.

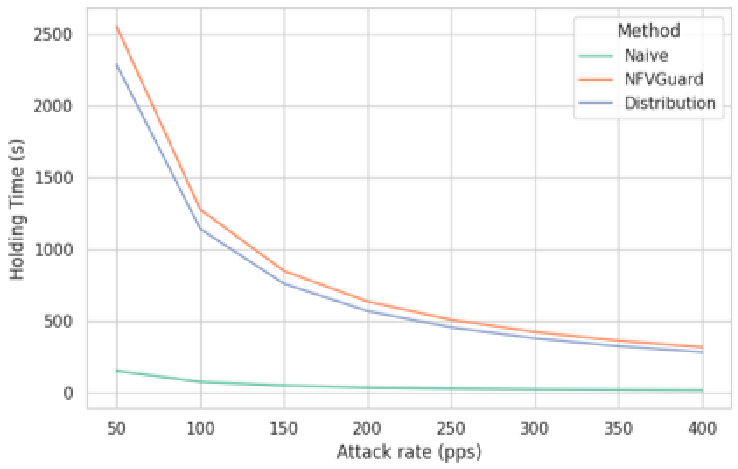

Figure 4.10: Holding Time for the Topology with 16 Switches

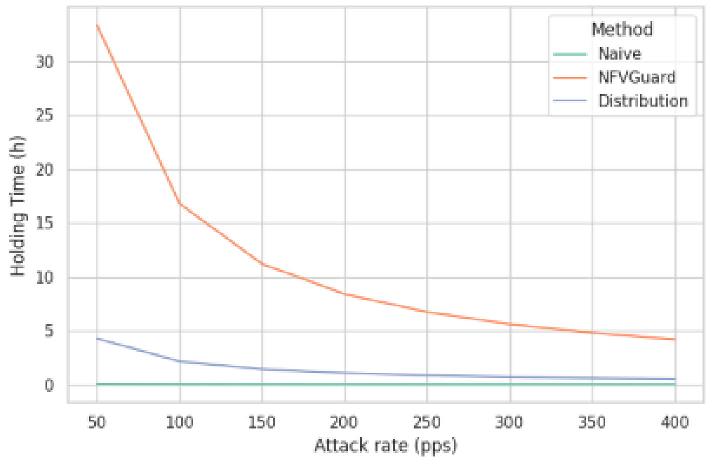

Figure 4.11: Holding Time for the Topology with 755 Switches

Next, we evaluated how switch count effects holding time difference while having same attack rate, 100 packet-ins per second (pps). Since switch count does not have any effect in the naive approach, the holding time does not change. Moreover, NFVGuard has a linear increase while distribution methods have non-linear increase. This non-linearity stems from cumulative guiding flow entries. The guiding flow entry count increases with the switch count. Therefore, the available space decreases while the holding time becomes significantly different.

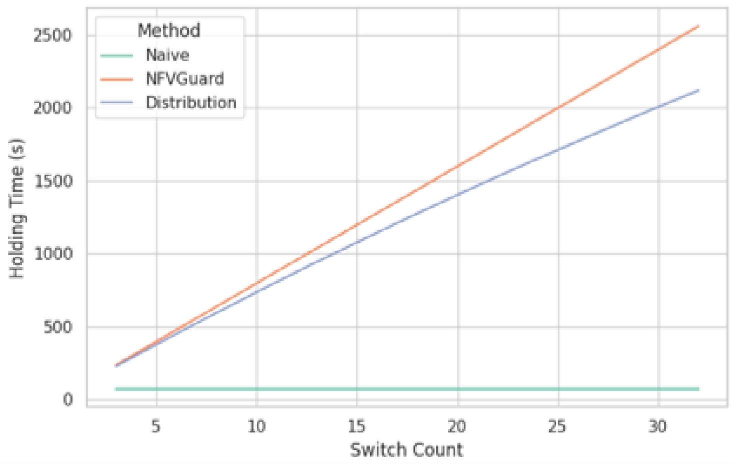

Figure 4.12: Holding time for Maximum 32 Switches

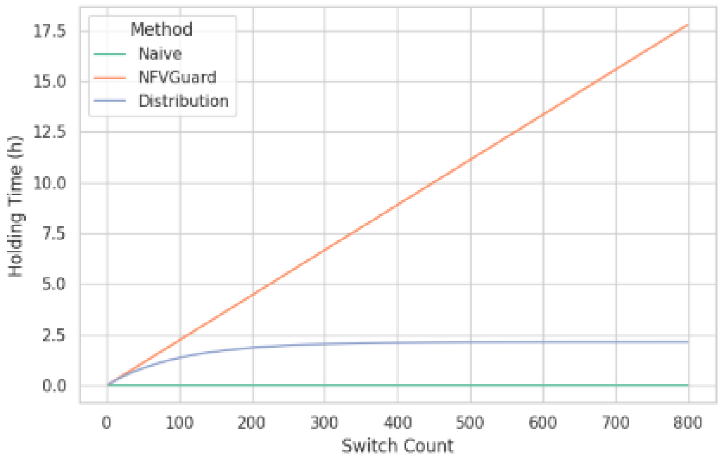

Figure 4.13: Holding time for Maximum 800 Switches

## 4.5 Discussion

In this chapter, we tested the feasibility of both static and dynamic approaches in the experiments and confirmed that NFVGuard is feasible to implement and effective on mitigating the flow table overflow attack. However, this section presents a brief discussion on the results obtained.

Firstly, concerning the static approach. All virtual switches need to run continuously. Therefore, if we have a dedicated hardware for Virtual Honeypot, it would work without the concerns of resource management or cost. However, it would be costly for both on cloud environments and on an in-house shared server. For instance, Cloud environments charges for every extra minute of resource usage. An in-house shared server would have other programs or services running in there so a dynamic management of resources would be better. In addition, we realized there was a high CPU usage in the NFVGuard Controller. The cause of it is the polling of the flow tables in order update databases on Flow Monitoring Modules (FMM).

Next, concerning the dynamic approach. We believe the dynamic approach, which adapts to attacks, would be cost-effective for cloud environments and resource-effective for an in-house server implementation. However, we saw an imbalance on middle level switches. In current flow entry distribution implementation, we are simply distributing new flow entries in a round-robin manner. We are not re-routing existing flow entries; thus, switches that initialized earlier have more flow entries. This does not lead to table overflow because of the safety value we set but it might create a bandwidth competition in highly utilized switches. In addition, NFVGuard updates databases using an asynchronous OpenFlow message, triggered when a flow entry is removed from the flow table. Therefore, the high CPU usage we detected on static approach is avoided in the dynamic approach.

Finally, concerning the last experiment. We conducted this experiment in order to assess how the topology might affect the holding time. From the results, we observed how the advantageous mitigation topology we created in virtual honeypot results in better holding time. In addition, while NFVGuard has a linear efficiency on the switch count, distribution approaches lose their efficiency near three hundred switches. However, we can still improve NFVGuard based on the flexible capabilities of the NFVI, regardless of the duration and density of the attacks.

# Chapter 5

# Conclusion

## 5.1 Summary

As the diffusion of SDN increases in different fronts, more vulnerabilities will be discovered and exploited. Flow table overflow is one of those vulnerabilities that needs to be addressed. We state the issues regarding the flow table overflow and its current mitigation methods as follows:

**(P1) High dependency on the size of the physical infrastructure**: Number and the capacity of physical network devices that reside in the domain greatly affects mitigation method.

**(P2) Coarse-grained flow entries block legitimate users**: Traditional Access Control List (ACL) compression methods use subnet masking based mechanisms, which limits compression rate. However, more abstract masks are implemented in order to decrease the rule size which results in blocking of legitimate packets.

**(P3) Harsh eviction policies with high probability of dropping legitimate traffic**: Proactive firewall policies are not sufficient to implement fine-grained blocking rules that only filter attackers.

To achieve the main goal of this research and solve the above-mentioned problems, we propose an NFV-based mitigation mechanism for SDN-based environments. More concretely, we proposed following.

**(S1) Dynamic NFV infrastructure**: A dynamic network topology created with virtual OpenFlow switches takes the burden of physical switches.

**(S2) Fine-grained flow entries distributed among the virtual infrastructure**: Increased overall flow table capacity allows a fine-grained filtering that only blocks attackers.

Based on the experimental results shown in Chapter 4, overall, we can say that the proposed mechanism can successfully mitigate the above mentioned issues. Moreover, we can conclude that the static approach is suitable for the cases when the target domain uses commodity servers; where the maximum number of virtual switches does not change the overall cost. By contrast, the dynamic approach is more suited to cloud environments where every resource used has a cost and in-house shared servers where resource usage and management might influence other services. The dynamic approach is a cost efficient method since it only creates extra resources in case of an attack, and according to the attack rate. We also compared our method with some of the existing distribution methods and find out that the proposed approach has better holding time as it creates better conditions to mitigate the table overflow attack (i.e, in terms of topology, policy management).

## 5.2 Future Work

A future work plan to include some points that were over simplified in the current implementation, for instance:

**Scaling of Virtual Honeypot topology**: Currently the topologies used were an initial step, and the switch removal mechanism was not implemented. This function is necessary to improve the dynamic scaling. In addition, the Virtual Honeypot topology can be changed into different structure and find a better structure to mitigate the attack.

**Imbalance on filtering virtual switches**: In our current implementation, we are distributing flow entries in middle level using a round robin method. However, we do not re-route existing flow entries in the switches. Therefore, the switches that exist more, have more flow entries. This imbalance does not result in flow table overflow due to the safety value we set on distribution. Regardless, it might create bandwidth competition on highly utilized switches. Implementing a better flow distribution policy that does efficient load balancing would solve this problem.

**Evaluation in real environments**: In all of our experiments, we did not use actual physical switches or a dedicated NFVI. Therefore, it would be helpful to evaluate the effectiveness of NFVGuard in these environments as well. For instance, we would like to test our methods in a cloud environment and in an in-house shared server to see cost and resource efficiency better.

# Appendix A

# Algorithms

## A-1

**Algorithm 1** Calculate THD & Merge IP

```
1: procedure THD($M, N, K$)
2:     $distance = 0$                                  ▷ Ternary Hamming Distance
3:     $IP =$ “ ”
4:     $mask =$ “ ”
5:     for i in 0..31 do
6:         $m_i, n_i, k_i \leftarrow M[i], N[i], K[i]$
7:         if $k_i = 1 \land m_i \neq n_i$ then
8:             $distance \leftarrow distance + 1$
9:         $IP = IP + k_i \land m_i \land n_i$
10:        $mask = mask + k_i \lor (m_i \oplus n_i)$
    return $distance, IP, mask$
```

## A-2

**Algorithm 2** Packet-in processing

```
1: procedure PACKET_IN($M$)
2:     $switch = pick\_switch()$
3:     $action = output(switch)$
4:     for each $flow$ in $flowDB$ do
5:         $d, ip, mask =$ THD($M.IP, flow.ip, flow.mask$)
6:         $mw = mask.count('0')$
7:         if $d = 0$ then                                ▷ Flow Timeout
8:             $match = M.IP$
9:             Update $flowDB$
10:            $addFlow(GS, M.IP, action)$
11:            return
12:        else if $d < d_t$ and $mw < mw_t$ then
13:            $old\_match = flow.IP, flow.mask$
14:            $match = ip, mask$
15:            $addFlow(GS, match, action)$
16:            Update $flowDB$
17:            $removeFlow(GS, old\_match)$
18:            return
19:    Create a new flow
20:    Update $flowDB$
21:    $match = M.IP$
22:    $addFlow(GS, match, action)$
```

# Acknowledgments

First of all, I would like to thank my advisor, Professor Takuo Suganuma, for accepting me into his laboratory both times. For this master course, and in 2016, I participated in an exchange program JYPE in Tohoku University. The freedom he gave me on choosing the topic and his guidance allowed me to conduct research I felt motivated to pursue. Furthermore, his wise and finely chosen words while interacting with me made me respect him academically and personally.

I would also like to thank Professor Go Hasegawa and Professor Hiroki Nishiyama for their constructive comments during the preliminary examination. Their words helped me improve this thesis and made me realize the strong points of my research better. It was an honor to have you on my thesis committee.

Thanks to Associate Professor Toru Abe, who provided remarkable comments with his refined understanding of academic writing and presentation storytelling. I believe his remarks improved not only my thesis but also myself, which I am grateful for.

Thanks to my long-time tutor, Associate Professor Satoru Izumi, for his support from the time of my exchange program until now. It was a long journey, and I could depend on him even I was not in Japan.

I want to thank Assistant Professor Luis Alberto GUILLEN BARJA, who excels in this field. He gave me sound advice and guidance on the process of creating this work. Also, I cannot thank him enough for reminding me what is essential in life and supporting me through this thesis and the conferences I participated in.

Thanks to students at Suganuma-Abe Laboratory, who made this academic pursue and daily life fun. In particular, to Yuki Ito and Masaki Mita for their moral support in harsh corona times. Also, I would like to thank Misumi Hata for all her help during my exchange time in this laboratory.

Thanks to my friends in Japan with whom I shared joys and misfortunes of life here. In particular, to Yiğit Çay and Roland Schimanski, who supported me in these difficult times.

I would also like to acknowledge the Ministry of Education, Culture, Sports, Science, and Technology of Japan (MEXT) for their scholarship. Their financial support allowed me to accomplish academic achievements. Furthermore, it nurtured me as a person thanks to all the people I met in Japan. Graduating with a Master's Degree in Japan was a long-time dream of mine; thus, I cannot express how grateful I am.

In addition, I would like to thank my parents for their wise counsel and sympathy; and to my little brother, whose smile always gives me immense joy in my life. You were always there for me. Finally, I could not have completed this thesis without the support of my friends C. Emre Ardıç, Doruk B. Kuzucu, and Ozan Nesipoğulları. They provided me with stimulating discussions and entertainment to balance my non-academic life.

# Publications

## Journal Papers

(1) S. Izumi, M. Hata, H. Takahira, M. Soylu, A. Edo, T. Abe, T. Suganuma, "A proposal of SDN based disaster-aware smart routing for highly-available information storage systems and its evaluation," Int. j. softw. sci. comput. intell., vol. 9, no. 1, pp. 69–83, 2017.

(2) M. Hata, M. Soylu, S. Izumi, T. Abe, and T. Suganuma, "A design of SDN based IP mobility management considering inter-domain handovers and its evaluation," Adv. Sci. Technol. Eng. Syst. J., vol. 2, no. 3, pp. 922–931, 2017.

(3) M. Hata, M. Soylu, S. Izumi, T. Abe, and T. Suganuma, "SDN based end-to-end inter-domain routing mechanism for mobility management and its evaluation," Sensors (Basel), vol. 18, no. 12, p. 4228, 2018.

## International Conferences

(4) M. Hata, M. Soylu, S. Izumi, T. Abe and T. Suganuma, "Data Flow Control With SDN Based Mobility Management," Proc. of the AEARU 11th Web Technology and Computer Science Workshop 2016 (WTCS2016) - Computer Science and Data Science, pp.31-32, 2016.

(5) M. Hata, M. Soylu, S. Izumi, T. Abe, and T. Suganuma, "Design of SDN based end-to-end routing over multiple domains for mobility management," in 2017 13th International Conference on Network and Service Management (CNSM), pp. 1–4, 2017.

(6) M. Soylu, L. Guillen, S. Izumi, T. Abe, and T. Suganuma, “ NFV-GUARD: Mitigating Flow Table-Overflow Attacks in SDN Using NFV”, IEEE International Conference on Network Softwarization

## Domestic Workshops

(7) M. Hata, M. Soylu, S. Izumi, T. Abe and T. Suganuma, “Basic Design of SDN Based Mobility Management for Multiple Domain Networks,” 平成 28 年度 電気関係学会東北支部連合大会, 1A19, 2016.

(8) 畑美純，M. Soylu，和泉諭，阿部亨，菅沼拓夫，“SDN 型モビリティマネジメント手法の設計と実装，” 電子情報通信学会技術研究報告, Vol.116, No.231, IN2016-45, pp.53-58, 2016.

(9) M. Hata, M. Soylu, S. Izumi, T. Abe, T. Suganuma, “A Study on SDN Based End-to-end Routing Mechanism for Mobility Management in Multiple Domain Networks,” 平成 29 年度 電気関係学会東北支部連合大会, 2B04, 2017.

(10) M. Hata, M. Soylu, S. Izumi, T. Abe and T. Suganuma, “Design and Implementation of SDN Based Mobility Management,” IEICE Tech. Rep., vol. 116, no. 231, IN2016-45, pp. 53-58, 2016.

(11) M. Hata, M. Soylu, S. Izumi, T. Abe and T. Suganuma, “SDN Based End-to-end Inter-domain Routing Mechanism for Mobility Management and Its Implementation,” IEICE Tech. Rep., vol. 117, no. 299, IA2017-46, pp. 71-76, 2017.

(12) ムスタファ ソユル，ルイス ギリエ，和泉　諭，阿部　亨，菅沼拓夫, SDN 環境における NFV を用いたテーブルオーバーフローの緩和の一検討, 情報処理学会　第 83 回全国大会講演論文集，4T-04，2021. 【学生奨励賞】

# References

[1] Open Networking Foundation, “Software defined networking (SDN) definition,” [Online] https://www.opennetworking.org/sdn-definition/. (Accessed: May 30, 2021)

[2] K. Bhushan and B. B. Gupta, “Distributed denial of service (DDoS) attack mitigation in software defined network (SDN)-based cloud computing environment,” J. Ambient Intell. Humaniz. Comput., vol. 10, no. 5, pp. 1985–1997, 2019.

[3] D. Gonzalez, C. Mellado, K. Waltam, and A. Lara, “Low-cost SDN switch comparison: Zodiac FX and raspberry pi,” in 2019 IV Jornadas Costarricenses de Investigación en Computación e Informática (JoCICI), pp. 1–5, 2019.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” SIGCOMM Comput. Commun., Vol. 38, no. 2, pp. 69-74, 2008.

[5] Open Networking Foundation, “OpenFlow Switch Specification ,” [Online] https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf. (Accessed: May 30, 2021)

[6] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” IEEE Commun. Mag., vol. 53, no. 2, pp. 90–97, 2015.

[7] European Telecommunications Standards Institute (ETSI), “Network Functions Virtualisation (NFV); Architectural Framework,” [Online] https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf. (Accessed: May 05 2021)

[8] M. Antonakakis et al., “Understanding the Mirai Botnet,” in 26th USENIX Security Symposium, pp. 1093–1110, 2017.

[9] Z. Guo, R. Liu, Y. Xu, A. Gushchin, A. Walid, and H. J. Chao, “STAR: Preventing flow-table overflow in software-defined networks,” Comput. netw., vol. 125, pp. 15–25, 2017.

[10] R. Sanger, B. Cowie, M. Luckie, and R. Nelson, “Characterising the limits of the OpenFlow slow-path,” in 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pp. 1–7, 2018.

[11] M. Yu, T. He, P. McDaniel, and Q. K. Burke, “Flow table security in SDN: Adversarial reconnaissance and intelligent attacks,” in IEEE Conference on Computer Communications, pp. 1519–1528, 2020.

[12] T. A. Pascoal, Y. G. Dantas, I. E. Fonseca, and V. Nigam, “Slow TCAM Exhaustion DDoS Attack,” in ICT Systems Security and Privacy Protection, Cham: Springer International Publishing, pp. 17–31, 2017.

[13] X.N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, “Rules placement problem in OpenFlow networks: A survey,” IEEE Commun. Surv. Tutor., vol. 18, no. 2, pp. 1273–1286, 2016.

[14] C. R. Meiners, A. X. Liu, and E. Torng, “Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs,” IEEE ACM Trans. Netw., vol. 20, no. 2, pp. 488–500, 2012.

[15] B. Leng, L. Huang, C. Qiao, H. Xu, and X. Wang, “FTRS: A mechanism for reducing flow table entries in software defined networks,” Comput. netw., vol. 122, pp. 1–15, 2017.

[16] B. Yuan, D. Zou, S. Yu, H. Jin, W. Qiang, and J. Shen. “Defending against flow table overloading attack in software-defined networks,” IEEE Transactions on Services Computing, vol. 12, no. 2, pp. 231–246, 2016.

[17] P. T. Duy, L. D. An, and V.H. Pham, "Mitigating flow table overloading attack with controller-based flow filtering strategy in SDN," in Proc. of the 2019 the 9th International Conference on Communication and Network Security, 2019.

[18] C. J. Fung and B. McCormick, "VGuard: A distributed denial of service attack mitigation method using network function virtualization," in 2015 11th International Conference on Network and Service Management (CNSM), pp. 64–70, 2015.

[19] A. H. M. Jakaria, W. Yang, B. Rashidi, C. Fung, and M. A. Rahman, "VFence: A defense against distributed denial of service attacks using network function virtualization," in 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 431–436, 2016.

[20] G. Somani, M. S. Gaur, D. Sanghi, and M. Conti, "DDoS attacks in cloud computing: Collateral damage to non-targets," Comput. netw., vol. 109, pp. 157–171, 2016.

[21] S. Luo, H. Yu, and L. Li, "Practical flow table aggregation in SDN," Comput. netw., vol. 92, pp. 72–88, 2015.

[22] N. Sultana, N. Chilamkurti, W. Peng, and R. Alhadad, "Survey on SDN based network intrusion detection system using machine learning approaches," Peer Peer Netw. Appl., vol. 12, no. 2, pp. 493–501, 2019.

[23] Mininet Team: Mininet: an instant virtual network on your laptop, [Online]. Available: http://mininet.org/. (Accessed: May 30, 2021)

[24] RYU project team: RYU SDN Framework, [Online]. Available: https://osrg.github.io/ryu-book/ja/Ryubook.pdf. (Accessed: May 30, 2021)

[25] V. Gueant, "iPerf - The TCP, UDP and SCTP network bandwidth measurement tool," Iperf.fr. [Online]. Available: https://iperf.fr/. (Accessed: May 30, 2021)