PATH VERIFICATION IN SOFTWARE-DEFINED NETWORKS USING PROGRAMMABLE DATA PLANES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HAKAN BOSTAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2022

Approval of the thesis:

PATH VERIFICATION IN SOFTWARE-DEFINED NETWORKS USING PROGRAMMABLE DATA PLANES

submitted by **HAKAN BOSTAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Ertan Onur
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. İbrahim Körpeoğlu
Computer Engineering, Bilkent University _____

Prof. Dr. Ertan Onur
Computer Engineering, METU _____

Assist. Prof. Dr. Pelin Angın
Computer Engineering, METU _____

Date: 24.01.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Hakan Bostan

Signature :

ABSTRACT

PATH VERIFICATION IN SOFTWARE-DEFINED NETWORKS USING PROGRAMMABLE DATA PLANES

Bostan, Hakan
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Ertan Onur

January 2022, 63 pages

Software-Defined Networks (SDNs) revamp the traditional network architectures by segregating the data plane and control plane and introducing a programmable and logically centralized control plane. Although SDNs bring along extensive improvements, as well as solutions to some of the network security problems, the security of SDN itself is often overlooked. Especially the security of the data plane is often overlooked because of the widespread assumption that the data plane devices are trusted. However, an adversary can compromise data plane devices and change their behavior. Due to a lack of verification mechanisms, the controller cannot verify that the forwarding behavior has not been altered and packets follow their intended paths. Solutions for traditional IP networks cannot be readily applied to SDN deployments. Moreover, solutions in SDN domain are held back by the limitations of the broadly used OpenFlow devices on the data plane. In this thesis, we present the path verification problem and propose a controller design, P4thV, that provides path verification and packet integrity verification by leveraging the benefits of SDNs and the programmable data planes using P4. P4thV works by analyzing the packets entering and exiting the network to detect packet integrity violations as well as abnormal forwarding behaviors.

Additionally, P4thV collects flow statistics from switches to verify the forwarding behavior of the switches further and detect attacks against the data plane. We prototype P4thV using Python and P4 enabled open source BMv2 software switch. We then evaluate its performance using Mininet emulations and present our results. Further, we compare P4thV against recent studies FOCES and SPHINX. Our experiments show that P4thV outperforms FOCES by achieving over 97% verification accuracy and almost two times faster anomaly detection while requiring 50% less control channel messages than SPHINX and causing negligible additional forwarding delays and 10% throughput degradation.

Keywords: Software defined networks, path verification, programmable data plane, p4

ÖZ

YAZILIM TANIMLI AĞLARDA PROGRAMLANABİLİR VERİ DÜZLEMİ İLE PATİKA DOĞRULAMA

Bostan, Hakan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ertan Onur

Ocak 2022 , 63 sayfa

Yazılım tanımlı ağlar, geneksel ağ mimarilerini kontrol ve veri düzlemlerini ayırarak ve programlanabilir mantıksal olarak merkezileştirilmiş kontrolcü tanımlayarak yenileştirmiştir. Yazılım tamımlı ağlar kapsamlı geliştirmelerin yanında bazı ağ güvenliği problemlerine çözümler sunmasına rağmen, yazılım tanımlı ağların kendisinin güvenliği sıklıkla göz ardı edilmiştir. Özellikle veri düzleminin güvenliği, veri düzlemindeki cihazların güvenilir olduğu varsayımının yaygın olarak kabul edilmesi yüzünden sıklıkla göz ardı edilmiştir. Fakat, veri düzlemindeki cihazlar ele geçirilerek davranışları değiştirilebilir. Kontrolcü, doğrulama mekanızmalarının eksikliğinden dolayı ağ cihazlarının veri iletim biçimlerinin değişip değişmediğini ve paketlerin ağda istenilen patikaları takip edip etmediğini doğrulayamamaktadır. Geleneksel IP ağları için üretilmiş çözümler ise yazılım tanımlı ağ yerleştirmelerine kolayca uygulanamamaktadır. Buna ek olarak, yazılım tanımlı ağlar için üretilmiş çözümler de veri düzleminde yaygın olarak kullanılan OpenFlow cihazlarının limitasyonları ile kısıtlanmaktadır. Bu çalışmada, patika doğrulama problemi tanımlanıp, yazılım tanımlı ağ ve programlanabilir veri düzleminin avantajlarını kullanarak patika doğrulaması

ve paket bütünlüğü sağlayan bir kontrolcü dizaynı olan P4thV sunulmuştur. P4thV, paket bütünlüğünü ihlal eden durumları ve anormal iletim davranışlarını, ağa giren ve ağdan çıkan paketleri analiz ederek saptamaktadır. Bununla birlikte, P4thV, ağ cihazlarinin paket iletim davaranışlarını doğrulamaktadır ve veri düzlemine karşı gerçekleştirilen saldırıları tespit etmektedir. Python ve P4 özellikle açık kaynak kodlu BMv2 yazılımsal ağ anahtarı kullanarak P4thV'nin bir prototipi yapılmıştir. P4thV'nin performansı Mininet emülasyonları ile değerlendirilmiş ve sonuçlar sunulmuştur. Ayrıca, P4thV, yakın zamanda yapılmış FOCES ve SPHINX çalışmaları ile karşılaştırılmıştır. Deneylerimiz sonucunda, P4thV, 97% doğrulama doğruluğu ve yaklaşık iki kat hızlı anomali tespiti ile FOCES'ten daha iyi sonuç vermiştir. Bu performans, SPHINX'e oranla 50% daha az kontrol mesajı, ihmal edilebilir iletim gecikmesi ve sadece 10% veri hacmi kaybıyla elde edilmiştir.

Anahtar Kelimeler: Yazılım tanımlı ağlar, patika doğrulama, programlanabilir veri düzlemi, p4

To my family, friends, and people who supported me throughout this long journey.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Prof. Dr. Ertan Onur for providing me with the opportunity to work with him. I am deeply grateful for his guidance, support, and endless patience during my studies. I would also like to extend my sincere thanks to all the members of WINSLab for providing a friendly and productive environment. Their immense knowledge led me in the right direction in the moments I was lost.

I offer my sincere gratitude to my dearest parents and sister for their endless love and care. They provided me with constant support and encouragement throughout my life. Without their support in the past few years, it would be impossible for me to complete my study.

I also would like to thank my friends Kadir Çetinkaya, Ahmet Alperen Akçay, Ülfet Çetin, and Bilal Yaylak. Their friendship means a lot to me. Also, their counsel was invaluable, especially in technical areas. I would like to offer my special thanks to Damla San for studying endless hours with me. Without them producing this thesis would be more excruciating.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xvii

CHAPTERS

1 INTRODUCTION 1

1.1 Problem Definition 4

1.1.1 Network Model 5

1.1.2 Adversary Model 6

1.2 Contributions 9

1.3 The Outline of the Thesis 9

2 BACKGROUND AND RELATED WORK 11

2.1 Background 11

2.1.1 P4 Workflow 11

2.1.2 Path Verification 12

2.2 Related Work 13

3 CONTROLLER BASED PACKET INTEGRITY AND PATH VERIFICATION IN SDN WITH P4 SWITCHES 19

3.1 P4thV Overview 20

3.1.1 Core Module 20

3.1.2 Flow Module 22

3.2 P4 Switch Processing Logic 27

3.2.1 Packet Parsing Pipeline 27

3.2.2 Ingress Pipeline 28

3.2.3 Egress Pipeline 30

3.2.4 Example 31

3.2.5 Complexity Analysis 33

4 SECURITY ANALYSIS 35

4.1 Packet Delaying Attacks 35

4.2 Packet Dropping Attacks 35

4.3 Path Deviation Attacks 36

4.4 Packet Injection Attacks 37

4.5 Packet Tampering Attacks 38

4.6 Modifying Statistics Packets 38

4.7 Coward Attacks 38

4.8 Collusion Attacks 39

5 RESULTS AND DISCUSSIONS 41

5.1 Experiment Setup 41

5.2 Selection of δ 43
5.3 Detection Accuracy 45
5.4 Detection Delay 47
5.5 Overhead 48
6 CONCLUSION AND FUTURE WORK 53
6.1 Conclusion 53
6.2 Future Work 54
REFERENCES 55

LIST OF TABLES

TABLES

Table 2.1 Comparison of P4thV with the related work. 14

LIST OF FIGURES

FIGURES

Figure 1.1 A typical SDN. The controller converts network policies into forwarding rules. 6

Figure 1.2 Types of path deviation attacks. 8

Figure 2.1 P4 packet processing pipeline stages. 12

Figure 2.2 Figure depicting the flow conservation principle. 17

Figure 3.1 One Core Module creates and manages multiple Flow Modules. Red arrow denotes a packet sent to the controller from the data plane. . 21

Figure 3.2 The flowchart of the Core Module. 22

Figure 3.3 Overview of the operation of a Flow Module. (1) p_1 is received from ingress switch and a timer is started. (2) Some timers expire, increasing the E value. (3) p is received from egress switch, corresponding timer is removed. (4) T_s expires triggering a statistics collection. . . 26

Figure 3.4 The flowchart of packet parsing pipeline. 27

Figure 3.5 The flowchart of the ingress pipeline. Dotted arrows and boxes show the modifications of P4thV-Hold. 29

Figure 3.6 The flowchart of the egress pipeline. 30

Figure 3.7 Overview of example. Numbers correspond to the steps in the example. Green arrows depict rule installation. Blue arrows depict statistics collection. 31

Figure 4.1 An example scenario of a packet dropping attack. Packets of a flow (orange arrow), are dropped by the compromised S_1 switch. 36

Figure 4.2 An example scenario of a path deviation attack. S_1 forwards the packets of a flow (orange arrow), out of the authorized path. 37

Figure 4.3 An example scenario of colluding attackers. S_1, S_2 and S_4 are colluding to elude detection. Since S_1 and S_2 are adjacent they can be considered as a single malicious entity. 39

Figure 5.1 India35 problem instance of SNDLib [1] used in Mininet emulations. 42

Figure 5.2 Impact of δ on detection accuracy on a flow where each packet has (a) a 1% chance of experiencing excessive delays that expires the packet timer at the controller, (b) a 1% chance of being dropped. 45

Figure 5.3 Effect of δ on network with both delay and loss chance under different attacks. 46

Figure 5.4 Comparison of true-positive and false-positive percentages of P4thV and FOCES under different attacks. 47

Figure 5.5 Detection delays with different $\mathcal{T}$ values under path shortcut attacks. 48

Figure 5.6 Detection delays of (a) P4thV and (b) FOCES under different attacks. (c) depicts the detection delays of FOCES and P4thV with increasing network size. 49

Figure 5.7 Round trip times of different controller designs throughout the lifetime of a flow. 51

Figure 5.8 (a) Effect on throughput. (b) Number of control channel messages with increasing number of flows (same pps). (c) CPU consumption of controller on different algorithms. (d) RTT values of different designs compared with P4thV-Hold. 52

LIST OF ABBREVIATIONS

ABBREVIATIONS	
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application-specific Integrated Circuit
AS	Autonomous System
ATPG	Automatic Test Packet Generation
BGP	Border Gateway Protocol
BMv2	Behavioral Model version 2
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAI	Dynamic ARP Inspection
DDoS	Distributed Denial of Service
DHCP	Dynamic Host Configuration Protocol
FCM	Flow Counter Matrix
FOCES	Flow Counter Equation System
HSA	Header Space Analysis
IP	Internet Protocol
IPv4	Internet Protocol version 4
ISP	Internet Service Provider
IoT	Internet of Things
MAC	Media Access Control
MIC	Message Integrity Code
OPT	Origin and Path Trace
P4	Programming Protocol-Independent Packet Processors

REV	Rule Enforcement Verification
RTT	Round Trip Time
SDN	Software Defined Network
TCP	Transmission Control Protocol
TTL	Time to Live
UDP	User Datagram Protocol

CHAPTER 1

INTRODUCTION

A network stack consists of data, control, and management planes [2]. While the user governing the general operation of the network resides on the management plane, control functions for carrying out the main tasks at each layer of the network live in the control plane. The control plane carries out the routing protocols that can informally be defined as finding some good paths between sources and destinations. Operations executed on data packets, such as switching user messages hop by hop to their destinations, are the responsibility of the data plane.

The hardware layer, called the data plane, is responsible for switching incoming packets through an outgoing interface over the path determined by the routing protocol. Take Internet Protocol version 4 (IPv4) as an example; the data plane of an IPv4 router performs several pipeline stages: verify header checksum, decrement time to live (TTL) field, update header checksum, making a forwarding table lookup to determine the outgoing interface and put the packet in the queue of the designated interface.

The control plane is responsible for populating the forwarding tables used by the data plane. Network administrators can configure the routing tables manually or configure the control plane to modify routing tables dynamically. While manually managing the network is feasible for modest and static networks, it becomes increasingly laborious and error-prone as the network becomes dynamic and grows complex. [3]

Software-defined networking (SDN) is a modern and highly regarded option for dealing with such complexity. SDN is a novel networking design that aims to solve some of the difficulties that have plagued traditional networking. SDN architecture segre-

gates the control logic (control plane) from the forwarding logic (data plane); therefore, traditional closed network systems are partitioned into the data plane, and the control plane [4, 5]. SDN promises a centralized control of networks that is versatile and programmable, thanks to this separation. Meanwhile, SDN moves the control logic to the logically centralized controller. Without the burden of the control logic, the network switch transforms into an unintelligent forwarding device, and policy implementation and network configuration become simplified [6].

A centralized controller in a traditional SDN network is in charge of converting network policies into forwarding rules and installing them at forwarding devices over a standard control channel (e.g., OpenFlow [7]).

SDN, compared to traditional networks, brings along advantages such as having a global view of the network, multi-granularity in network control, traffic optimization [8, 9]. Emerging fields like cloud services and the internet of things (IoT) started adopting SDN [2, 10, 11].

SDN's ability to view the network from a global perspective enables the control plane to observe the network flows and extract information for security applications' use. Additionally, the programmable structure of SDN architecture grants security mechanisms the ability to swiftly change the configuration of the data plane, and thus the forwarding behavior, to annul threats in real-time [12, 13, 14].

While SDN comes with numerous improvements, the unintelligent data plane of SDN causes performance and scalability concerns [15, 16]. Unintelligent data plane devices lack stateful packet processing and rely heavily on the controller for traffic forwarding and monitoring, which in turn introduces a communication overhead between the control and data planes. For example, widely used OpenFlow switches are bounded by the fixed design of the application-specific integrated circuit (ASIC) they contain. Even though OpenFlow switches partially support stateful packet processing, they are restricted by the fixed number of predefined headers and actions provided by the OpenFlow specification. While the number of recognized headers of OpenFlow has increased from 12 to 44 over multiple versions, it is still beyond the capabilities of the network operators to extend this predefined set of headers. As the new header fields are being introduced regularly, this lack of flexibility can impact the

services that the switches can provide. For example, network administrators may not be able to satisfy the needs of various applications. Tackling these issues of OpenFlow, a high-level, domain-specific language of Programming Protocol-Independent Packet Processors (P4) [17] has emerged. The flexibility provided by P4 enables a programmable data plane without compromising the line rate performance of the switch. With P4, programmers are able to customize the packet processing logic of the switch as well as declare and implement new custom protocols that may require custom header fields. In addition to rapid implementation and deployment of complex and custom protocols, a programmable data plane facilitates the tools for innovative uses of the network, such as in-band telemetry [18].

SDNs may also suffer from attacks that burden traditional networks. Such attacks are still effective in SDNs due to the complexity of SDNs or underlying protocols (e.g., Address Resolution Protocol (ARP) [19]). However, switches lacking intelligence in the SDN context make integrating safeguards in traditional networks against such attacks into SDN non-trivial. Although this lack of intelligence of switches can be compensated by patching the SDN controller, it is not extensive enough to cover all security concerns. For example, in traditional networks, switches use Dynamic ARP Inspection (DAI) [20] as protection against ARP poisoning attacks [21]. Switches process all Dynamic Host Configuration Protocol (DHCP) messages that pass through in order to keep track of authentic media access control (MAC) address to IP address mapping and ward off a malicious DHCP server from serving clients. Even though DAI is a well-known defense, unintelligent switches in SDN make the extension of DAI into SDNs complicated.

While SDN is a building block to mitigate some of the network security problems, the SDN data plane contains many vulnerabilities. The data plane's fragile nature seems to stem from the assumption that all network devices on the data plane are inherently trusted and assumed to follow the network policies put in place by the controller.

Data plane behavior can change due to various reasons. For example, switches can experience software and hardware bugs resulting in abnormal behavior [22, 23, 24, 25, 26, 27, 28]. Moreover, recent studies show that SDN switches can be exploited reliably to take control of the switch and modify its behavior [29, 30, 31]. These

situations violate the assumption that the data plane adheres to the determined policies and the controller has a global view of the network. Detecting these behavioral changes has paramount importance in the security and stability of SDN.

However, because of SDN's open-looped control paradigm, controllers are unable to verify that policies are being obeyed, and the SDN data plane does not provide any proof that the policies set by the controller are followed and not violated.

Misbehaving switches can drop, clone, or tamper with packets that pass through, and throttle, forge, deviate or inject network traffic in order to execute attacks against network operators or users. As discussed in [32, 33, 34], attackers can leverage the compromised switches to usurp control of the entire network. Due to a lack of behavior and integrity verification in the data plane, the controller may utterly be oblivious in these types of situations.

To this end, many *data plane debugging* tools have been proposed. Using these debugging tools, network administrators can test whether the forwarding rules at switches correspond to the controller's view of the network [35, 36, 37] or monitor the compliance of the forwarding behavior of switches with control plane policies [38, 39]. However, these tools work by checking the correctness of the network policies on the controller's side [40, 41, 42] or assume that the switches are trusted [43, 35, 38]. As a result, they fail to work correctly under adversarial settings.

1.1 Problem Definition

In SDN, the controller has no means to verify the actual path taken by a packet traversing the network. In other words, the controller only knows about the path it authorized for the packet, not the path actually taken by the packet. Additionally, the controller cannot know whether the packet has been altered while it traverses the network.

Let p be a packet of a flow f. When p enters the network through an ingress switch S_0, S_0 tries to match p against a set of forwarding rules. These forwarding rules collectively define the next-hop switch for p, S_1. When p arrives at S_1, it is again matched against a set of rules and forwarded to the next-hop along its path, S_2. This

match-action process continues until p reaches the egress switch S_n, which forwards p out of the network.

Let $P(f)$ be the path authorized by the controller for packets of flow f to follow and it is represented by $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n$. Let $P'(f)$ be the actual path followed by the packets of flow f. Successful path verification ensures that the packets traverse all switches on the authorized path in the correct order. For path verification, the controller must ensure that $P(f) = P'(f)$ for every flow in the network.

On top of this, let the contents of p when entering and exiting the network be c and c', respectively. Contents of p may be altered while p traverses the network, causing $c \neq c'$. Successful integrity verification ensures that the contents of p when it enters and exits the network are exactly the same, so the controller must ensure the $c = c'$ for every packet of every flow in the network.

In this thesis, we tackle the path verification and packet integrity problem. Combining the advantages of SDN and P4, we propose an extension to the SDN controller, which ensures packets follow authorized paths and their contents do not change in transit. Therefore our design proposes solutions to both the path verification and the packet integrity problems.

1.1.1 Network Model

We consider a typical internet service provider (ISP) backbone SDN consisting of a centralized controller and a set of switches managed by the controller. The controller provides the means for specifying high-level policies to the network operators. The controller processes these high-level policies down into a set of forwarding rules. The controller then modifies the switches with these forwarding rules via a standard control channel such as P4Runtime, or OpenFlow [44, 45] as shown in Figure 1.1.

Each forwarding rule is a pair of a *matching field* and an *action*. When a packet arrives at a switch, its contents are compared against the *matching field*s of existing rules, and in the case of a match, the corresponding *action* is taken by the switch. In addition, new rules can be added to and removed from the switches in a reactive manner (e.g., when a new flow enters the network). We assume that the controller

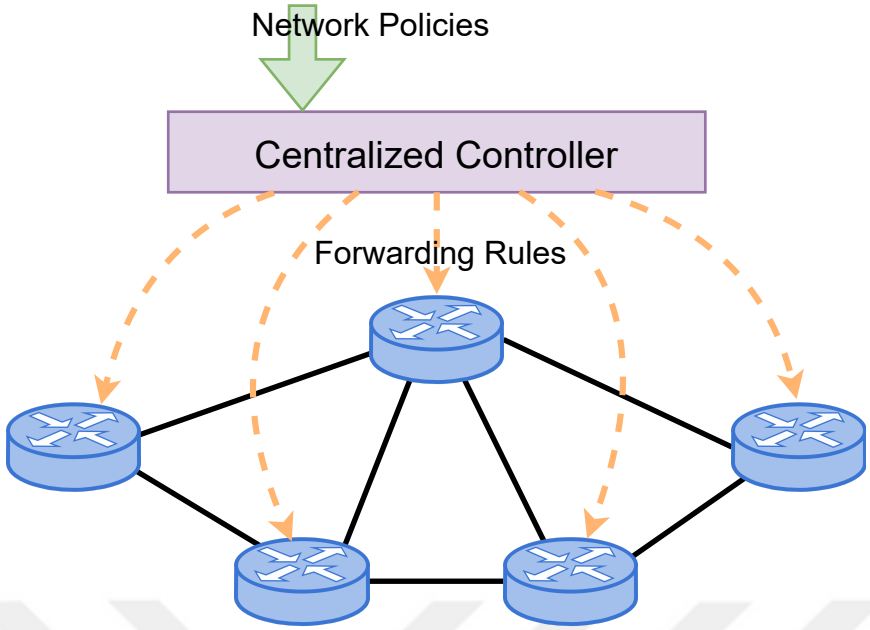


Figure 1.1: A typical SDN. The controller converts network policies into forwarding rules.

has a global view of the entire network and can request statistics from switches at any time.

Since a trusted controller is essential for a correctly functioning network, we assume that the controller is trusted in our network model. However, we trust neither the switches nor the end hosts. Because the switches communicate with the controller over different TCP connections, untrusted switches can lie about everything except their identity. We further assume that the edge switches in the network are trusted and honest. Finally, we assume that the applications operating at the controller are also trusted as long as the controller is trusted.

1.1.2 Adversary Model

In order to leverage any internal vantage points in the network, attackers often break into the network in order to conduct attacks on other internal network targets. Additionally, detection of attacks on the data plane is our primary purpose. Therefore, we model our adversary model to launch attacks from the internal network. As a result,

we consider SDN as a closed system. Removing unknown external communication allows us to concentrate our study on messages internal to the SDN.

The adversary tries to thwart the network operator's network policies, interrupt communication between end hosts, or send unsolicited packets to end hosts. For example, an attacker may change the specific flow rules assigned by the controller to forward packets along a different path from the one appointed by the controller to bypass various middleboxes (e.g., a firewall service). As another example, an attacker may also aim to deny a service to some or all clients by dropping packets of the service under attack.

We consider the adversary is capable of performing packet delaying, injection, tampering, dropping, and path deviation attacks [46, 47, 48, 49, 50, 51]:

- **Packet Delaying Attacks:** Adversary configures a switch to impose extra intentional delays that go through the malicious switch in order to break the underlying protocol.
- **Packet Injection Attacks:** Malicious switch injects packets into an existing flow. Injected packets can either be a replay of the previous traffic seen over the flow or can be crafted from scratch by the malicious entity.
- **Packet Tampering Attacks:** Adversary alters the packets' contents (headers or payload) as the packets visit the compromised switch to hijack or disrupt the communication without being detected.
- **Packet Dropping Attacks:** Adversary modifies a switch to discard packets of specific flows, causing packets to disappear at the malicious entity.
- **Path Deviation Attacks:** The adversary alters the switch to forward packets of a flow out of the path assigned by the controller. Packets may or may not return to the original path. In the case the packets do not return to the original path, this attack is synonymous to *packet dropping attacks*. Path deviation attacks can manifest in various ways (Figure 1.2):
  - *Path Detour:* A switch forwards packets to a switch outside of the assigned path. After one or more hops, the packet returns to one of the

downstream switches after the compromised switch on the original path.

- *Path Forgery:* A switch forwards packets to a switch outside of the assigned path. Later the packet does not return to the assigned path.

- *Path Shortcut:* A switch forwards packets such that the traffic goes over a subset of the switches on the assigned path.

- **Modifying Statistics Packets:** Malicious switch can generate altered statistics packets as responses to the controller's statistics collection requests to hide from the detection mechanisms employed by the controller.

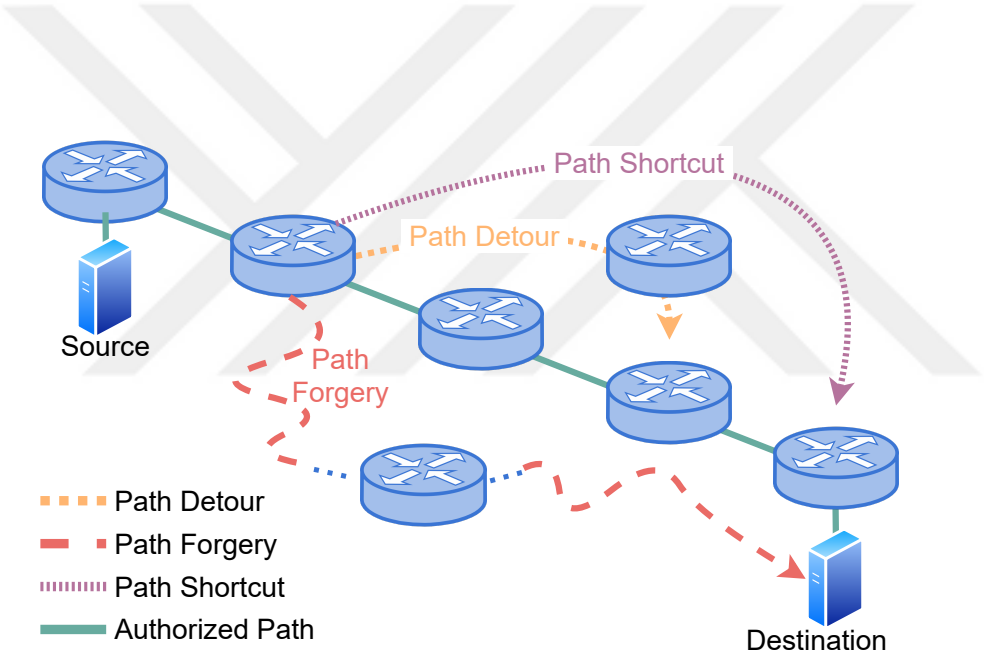


Figure 1.2: Types of path deviation attacks.

Further, a malicious entity can record and analyze the control messages between switches and the controller to infer the detection mechanism and perform attacks only when it believes that the attack will not be detected [52].

An adversary may also compromise more than one switch. He can use the compromised switches in tandem to perform a combination of the above attacks or to elude detection.

1.2 Contributions

In this study we provide a new mechanism to ensure packet integrity and provide path verification. Our contributions are as follows:

1. We propose an extension to the SDN controller, P4thV, that provides path verification and packet integrity verification. Combining the advantages of SDN with programmable data plane using P4, P4thV gathers statistics from the network switches and keeps track of packets entering and exiting the network to detect forwarding behavior changes and packet modifications.
2. We perform a security analysis on P4thV to justify its effectiveness in detection of various data plane attacks. We explain how P4htV detects each attack separately. We also consider collusion attacks where multiple switches work together, and switches reporting false statistics to evade P4thV.
3. We implement our design and perform extensive experiments with Mininet emulations. Our experiments show that P4thV can detect attacks effectively with 97.2% accuracy. We also perform comparisons against FOCES [53] and SPHINX [54]. Further, we inspect the overhead introduced by P4thV and compare it to related work.

1.3 The Outline of the Thesis

The outline of the thesis is as follows. In Chapter 2, we present the required background information and inspect the related work in the literature. Our design is explained in detail in Chapter 3. Chapter 4 covers the analysis of the proposed design. Experiment results are presented and discussed in Chapter 5. Finally, the conclusion, final remarks, and future work are presented in Chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Background

This chapter presents the background information necessary to make this study more accessible and discusses other research in path verification.

2.1.1 P4 Workflow

One drawback of the widely used OpenFlow interface is that it is constrained by a fixed-function switch design. OpenFlow switches only support a pre-defined set of headers [55], and the controller must work within this limitation. Addressing these issues, a domain-specific, high-level language for Programming Protocol-independent Packet Processors (P4) [17] emerged. Utilizing P4, operators are able to describe the whole packet processing behavior of the data plane devices.

P4 packet processing pipeline consists of several stages, as shown in Figure 2.1. When the packet first arrives at the switch, it first enters the *parser*. The parser works as a state machine to transform the bit representation of the packet into *header types* defined in the P4 configuration. For example, a UDP header type may consist of four-bit vectors for the source and destination port numbers, length, and checksum. Then, the packet passes through the ingress and egress match-action tables. At match-action tables, parsed headers are used as keys to the rules installed by the controller, and the corresponding action is taken at each table. Although both actions at ingress and egress may modify the packet headers, only the actions at ingress may determine the egress queue the packet will be put into. After passing through match-action tables,

deparser reconstructs the bit representation of the packet from the parsed headers.

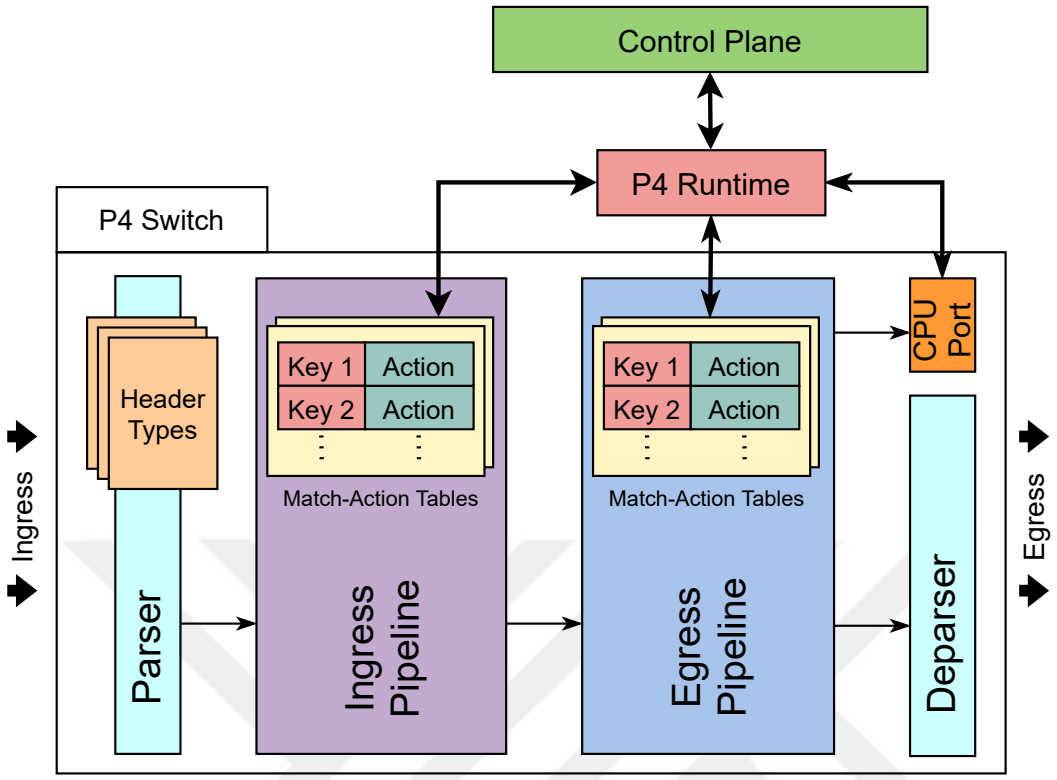


Figure 2.1: P4 packet processing pipeline stages.

During runtime, the control plane controls P4 devices via the P4Runtime [44]. P4Runtime facilitates the manipulation of match-action tables. The control plane can also collect statistics from P4 devices, such as the number of packets processed via this control channel. Figure 2.1 also depicts the P4 Runtime control channel.

2.1.2 Path Verification

The goal of path verification is to verify that a packet has traversed the path authorized by the controller. If the actual path taken by a packet while traversing the network does not exactly match the authorized path and deviates in some way, path verification should fail.

Secure routing [56, 57], source routing [58], and traceroute [59, 60] are commonly thought to provide path verification. Secure routing solutions' primary goal is to se-

cure the path-finding process against attackers and to find the best option provided by the routing protocol in use [56]. This may give the impression that secure routing provides secure forwarding on the path. However, it is beyond the scope of secure routing if the packets are transported along selected paths. To have more control over the forwarding paths, source routing [61] is proposed. In source routing, packet headers are extended to relay forwarding information to each hop on a path. Each device forwards the packet depending on the routing information extracted from this header. In source routing solutions, forwarding devices are considered to obey these instructions when forwarding packets. Therefore, there is no effort to secure the path directives, and they are stored as plaintext in packet headers. However, compromised switches may tamper with plaintext directives to alter the path of the packet without detection. Traceroute initially emerged as a solution to IP spoofing and DDoS attacks [60, 62]. Traceroute solutions either store packet metadata or embed marks into packet headers to keep track of traffic history. Using these packet markings or stored metadata, one can work out the path a packet has taken. Unfortunately, traceroute cannot be used directly as a path verification tool since plaintext logs or packet markings are vulnerable to forging or tampering by a compromised switch.

2.2 Related Work

In this section, we introduce previous related works and briefly explain their design. Table 2.1 presents a general comparison of detection capabilities of prominent works.

Multiple path verification tools have been proposed for traditional networks. These tools mainly work by embedding cryptographic information to packet headers. ICING [63] is one of the most prominent works on this topic. The source node in ICING computes a verifier message authentication code (MAC) for each router along the path of the packet. As the packet travels through routers on the path, each router computes a MAC for the packet and verifies it against the MAC values stored in the packet header. Finally, the destination can check if the packet followed the expected path inspecting these MAC values. A drawback of ICING is that it uses Diffie-Helmann for the key distribution between source nodes and the routers. Additionally, ICING requires routers to perform $n + 1$ MAC calculation operations, where n is the number

Table 2.1: Comparison of P4thV with the related work.

Scheme	**Detection Capability**				
	Delaying	Tampering	Dropping	Injection	Path Dev.
ICING, OPT, PPV	×	✓	×	✓	✓
SPHINX	×	×	✓	×	✓
FOCES	×	×	✓	✓	✓[1]
SDNSec, REV	×	✓	×	✓	✓
P4Label	×	✓	×	✓	✓
WedgeTail	✓	×	✓	✓	✓
P4thV	✓	✓	✓	✓	✓

[1] Fails under specific conditions.

of hops on a path, causing significant per-packet overhead. Even though using aggregate MACs to reduce the size, information embedded in the packet header imposes considerable communication overhead.

Similar to ICING, Origin and Path Trace (OPT) [64] relaxes the constraints and assumes that all nodes on a path trust the source node and share a symmetric key with it. Security of both ICING and OPT are similar, but OPT can also provide protection against coward attacks.

Different from ICING, observing that not all services require strict per-packet verification, PPV [65] proposes a probabilistic method to verify flows. Nodes are configured to mark packets with certain probabilities. Since not all packets are marked, a single packet may not verify the forwarding path. However, packets of a flow can collectively verify the forwarding path. Working on autonomous system (AS) level, Passport [66] employs additional header space in border gateway protocol (BGP) route advertisements to provide AS authentication.

Works have also been proposed to increase the security of SDN. FRESCO [67] proposes a modular security application framework enabling operators to combine and deploy security applications to the controller. FloodGuard [68] and Avant-Guard [69] alters flow management at switches to protect the controller, making security applications more resilient and responsive.

Policy verification tools that verify the high-level policy against the control or data plane have also been proposed. FortNOX [70] works on the controller and extends controller behavior to detect rule conflicts in real-time. Anteater [40] and Header Space Analysis (HSA) [41] perform static analysis on the controller in order to verify the correctness of the data plane and discover bugs or conflicts in the configuration of the data plane. VeriFlow [42] uses a trie to classify the entire network according to forwarding behavior and perform verification in real-time. NetPlumber [71] uses HSA incrementally to maintain a dependency graph of update rules to enforce runtime policy checking.

However, even if the data plane configuration is correct, switches may experience faults. ATPG [43] addresses this issue by probing the network to check pairwise reachability. VeriDP [72] removes this constraint by checking forwarding behavior against data plane configuration at the controller. Veriflow, NetPlumber, Anteater, HSA, ATGP, and VeriDP all assume that the switches are trusted and do not work in adversarial conditions.

In the area of P4 related verification tools, plenty of tools have emerged [22, 23, 26]. However, most only debug P4 programs using static analysis techniques. Runtime bugs and inconsistency that may happen under traffic cannot be detected by these tools.

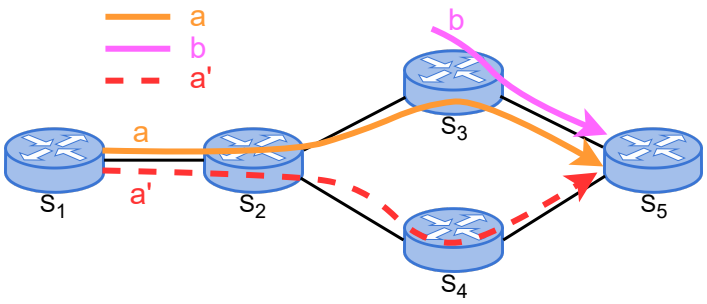
Extending the capabilities of ICING to the SDN context, SDNSec [73] provides detection mechanisms against both path deviation and packet tampering. In SDNSec, the controller generates a path for each flow and configures the ingress switch to add related forwarding information to the packet header, and each switch on the path performs forwarding according to these instructions. Additionally, switches create and embed a MAC into each packet's custom header as proof of processing the packet. Later, the egress switch reports the packet to the controller. The controller can check whether the packet followed the right path. Computation requirements for SDNSec are quite significant because of the cryptographic operations performed on each switch for each packet. SDNSec also introduces a serious bandwidth overhead as the custom headers can grow quite large.

REV [74] reduces this overhead by utilizing compressive MACs [74] based on homo-

morphic MACs [75], which allows switches to reduce the extra header space required by MACs. Egress switches in REV only forward a single flow packet to the controller for the verification of the whole flow. If the packet passes the verification at the controller, it shows that every packet has followed the authorized path with high probability. In the realm of P4, P4Label [76] protects the network against packet tampering, forgery and path deviation attacks by utilizing an identity-based cryptosystem and leveraging the flexibility of P4 in defining custom headers and protocols.

Based on traceroute, SDNTraceroute [77] employs a probing approach. To query the path of a packet, SDNTraceroute sends probe packets to travel the suspicious path. On every hop, these probe packets cause switches to send notification packets to the controller. Generating notifications on every hop incurs significant overhead. Moving the focus from following the path to inspecting specific rules, [31] proposes a detection mechanism to identify compromised switches. However, the proposed design is quite inefficient as it needs to check every rule at every switch individually. Improving on this idea, Chao et al. [78] employs a similar strategy to monitor rules at a switch. Assume that the rule r at switch S is monitored. The controller inserts additional rules to each neighboring switch of S with the same matching rules as r to count the matches at the neighboring switches. Then the controller retrieves the counts from neighbors of S, and if the difference is greater than some threshold value, the verification fails. To reduce the number of flows and rules to inspect, FADE [79] introduces the concept of *rule path*s and selects a subset of all flows and rules to cover the whole network. However, these last three methods need to install dedicated rules for the verification process and may exhaust the memory of the switch.

Similar to the workflow of Veriflow and NetPlumber, SPHINX [54] collects metadata of the network from control messages and constructs *flow graphs*. Flow graphs model the flows in the network in a graph theoretical way, where nodes and edges in the graph represent the switches and the metadata of the flows, respectively. These flow graphs provide an abstraction of the network and its operation, allowing SPHINX to monitor the network continuously and detect any irregular behavior. From a high-level point of view, SPHINX works in three phases. First, SPHINX intercepts OpenFlow messages that provide information on the topology changes, flow paths, and flow metadata (e.g., flow counters). Second, SPHINX uses these intercepted mes-



Rule	Expected Counter		Observed Counter
r1	a		a
r2	a		a
r3	$a+b$	$\rightarrow$	b
r4	0	$\rightarrow$	a
r5	$a+b$		$a+b$

Figure 2.2: Figure depicting the flow conservation principle.

sages to construct a complete representation of the network graph. As more relevant messages are intercepted, SPHINX updates this representation. Finally, SPHINX performs verification on the flows' current state to check whether they (1) are within the acceptable values and (2) conform to the administrative policies. As the result of this verification process, SPHINX can detect unexpected deviant behavior as well as administrator-defined known attacks. However, SPHINX is unable to identify ingress and egress switches that inject or drop packets into a flow without the help of additional mechanisms. Moreover, the frequency of updates may cause SPHINX to miss short-lived attacks as attacks may begin and finish between the updates SPHINX receives.

Another method that uses flow statistics to detect forwarding anomalies and provide path verification is *Flow Counter Equation System (FOCES)* [53]. FOCES uses the flow conservation principle, which states that the counter value for a rule on a switch should be equal to the sum of the number of packets of all flows that have the switch on their path and use that rule. For example, consider a network as shown in Figure 2.2. There are two flows a and b following $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_5$ and $S_3 \rightarrow S_5$, respectively. Each flow matches exactly one rule at each switch, furthermore, assume that both a and b matches the same rules on S_3 and S_5. Consider the case S_2's behavior changes and it forwards a via S_4 to S_5 (flow a' shown with a dashed red line). In this case, the counter value of S_3 would become b instead of $a + b$, violating the flow conservation principle.

Different from previous work that also utilize the flow conservation principle, FOCES does not consider each flow in the network separately. Instead, it checks the

consistency of rule counters of the whole network together. First, FOCES creates a *Flow Counter Matrix (FCM)*. FCM represents whether a flow matches a rule in the network. In a network with n flows $f_{1..n}$ and m rules $r_{1..m}$, FCM, H, is defined as:

$$H_{i,j} = \begin{cases} 1 & \text{if } f_j \text{ matches } r_i \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

Then, two vectors *counter vector* and *volume vector* are defined. Counter vector, Y, contains the counter values of each rule, and volume vector, X, represents the traffic volume of each flow. FOCES then constructs flow-counter equation system:

$$HX = Y \tag{2.2}$$

FOCES detects attacks by using the observed Y values. In case of an attack, H would be modified and observed Y' values would not satisfy $HX = Y'$. Therefore, FOCES would detect this attack. To solve the equation 2.2 with observed values, FOCES relies heavily on expensive matrix operations on large matrices, which impose heavy computation overhead.

Neither FOCES nor SPHINX can recognize packet tampering attacks. Moreover, FOCES also falls short in the case where the forwarding behavior is altered in such a way that the counters seem to satisfy the flow conservation principle while forwarding wrong flows.

CHAPTER 3

CONTROLLER BASED PACKET INTEGRITY AND PATH VERIFICATION IN SDN WITH P4 SWITCHES

In this chapter, we explain our proposed design, P4thV, to satisfy path verification and to provide packet integrity in SDN. We consider an ISP backbone network where network traffic flows between various source and destination nodes through intermediate nodes. Our design enables the controller to detect anomalies on any flow. We satisfy these design goals with our proposed design:

- *Path verification*: Packets of a given flow should traverse the network along the path authorized by the controller. Due to malicious entities, compromised switches, or misconfiguration, switches might not act as expected and forward packets on an unexpected route. Even if these changes in forwarding behavior do not prevent the packet from reaching its destination, they may create security risks. Thus the controller should be able to verify that the path taken by packets of a flow match exactly the legitimate path assigned by the controller.

- *Packet integrity*: Packets entering the network should leave the network without having their contents altered. Malicious switches along the legitimate path of a packet can alter the packet's contents in order to hijack the communication or disrupt the communication between two hosts. For example, a malicious switch can convert every TCP-SYN packet into a TCP-RST by changing a few bits in the packet to prevent two hosts from establishing a TCP communication. When the contents of the packets of a flow are changed during transit, the controller should be able to detect that some packets are altered.

- *Efficiency*: Achieving the above two goals should not cripple the performance

of the network by introducing either too much computation overhead or bandwidth overhead.

These goals shape our architecture, from the controller design to processing logic in the switches. First, we start with our controller design; then, we explain our design on P4 switches and modifications we make to the data plane packet processing logic.

3.1 P4thV Overview

In our design, the controller tracks the packets entering and exiting the network with the help of information provided by ingress and egress switches. Tracking these packets enables the controller to verify that the contents of a packet do not change while it traverses the network, and it enters and exits the network following the legitimate path assigned by the controller. Additionally, the controller collects statistics from all switches currently taking part in the forwarding process with random intervals. These statistics provide the number of packets processed by the switches for a given flow. Using these statistics, the controller then can verify the forwarding behavior of the data plane is as expected.

In other words, our design detects packet modification and delaying attacks by pairing up the packets that enter and exit the network and detect packet dropping and path deviation attacks by comparing statistics collected from the data plane.

The controller is composed of two modules: *Core Module* and *Flow Module*. The Core Module is responsible for identifying packets' flows, keeping track of active flows in the network, and creating and maintaining a Flow Module for each active flow. Each Flow Module provides path verification and packet integrity verification for a single flow by keeping track of packets and statistics of that flow.

3.1.1 Core Module

Core Module's three main tasks: (1) identifying flows of packets received from the data plane, (2) creating new Flow Modules for new or unknown flows in the network

and (3) forwarding the received packets to the related Flow Module.

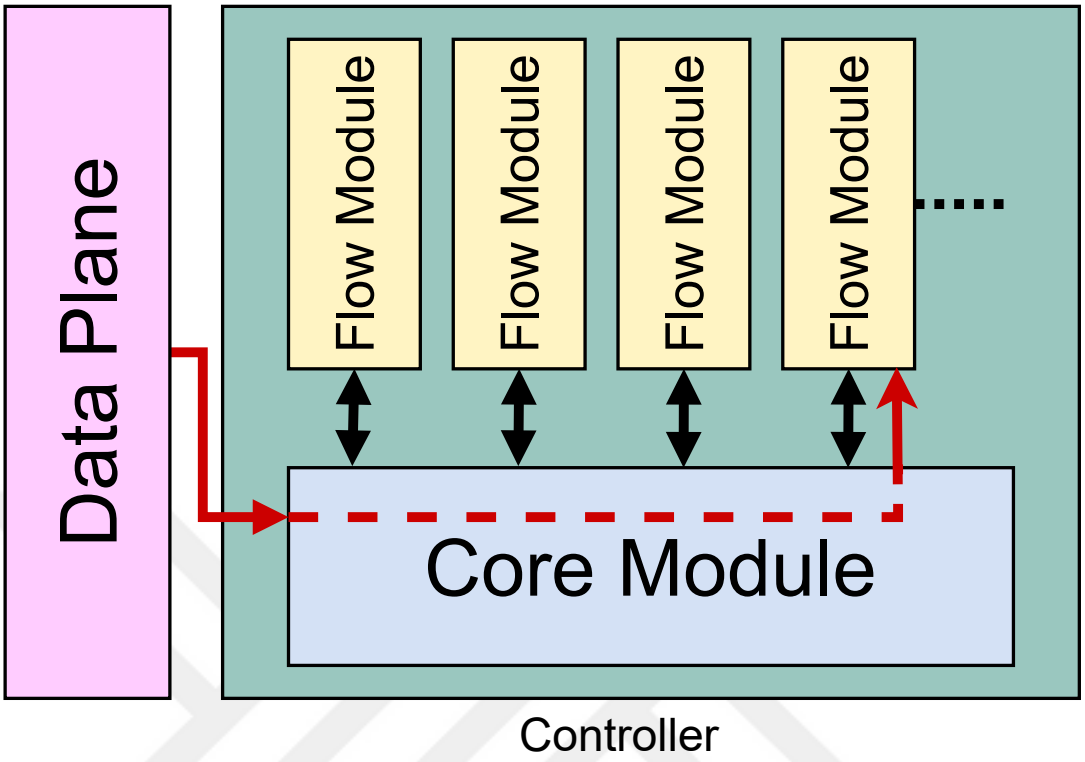


Figure 3.1: One Core Module creates and manages multiple Flow Modules. Red arrow denotes a packet sent to the controller from the data plane.

For every packet received from the data plane, the Core Module needs to calculate the **FLOWID** of the packet in order to identify the flow to which the packet belongs. **FLOWID** is calculated as the **CRC32** value generated from the source IP address, destination IP address, IP protocol type, source port, and destination port fields of the incoming packet. We choose the **CRC32** algorithm to compute the **FLOWID** to satisfy compatibility with the capabilities of the P4 switches. Then, the Core Module looks up **FLOWID** on a hash table used to store the Flow Modules corresponding to each known **FLOWID**. If the **FLOWID** does not exist in the hash table, it means that the received packet is a packet of a new flow. Then, the controller creates a new Flow Module for the new flow and adds its **FLOWID** and newly created Flow Module to the hash table. If the **FLOWID** already exists in the hash table, the Core Module simply passes the packet along to the corresponding Flow Module.

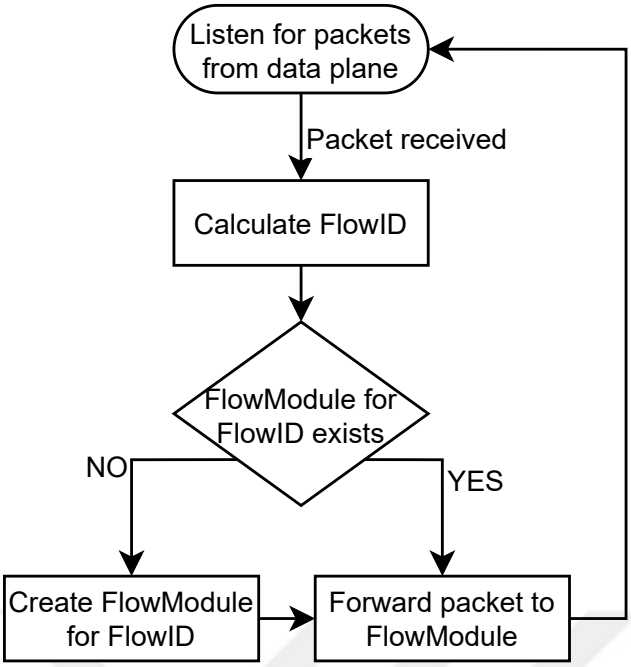


Figure 3.2: The flowchart of the Core Module.

3.1.2 Flow Module

Flow Module is also responsible for three tasks: (1) finding and setting up a suitable path for the flow, (2) keeping track of when a packet enters and exits the network, and (3) collecting and analyzing statistics collected from the switches on a flow's path. The algorithm of the Flow Module is shown in Algorithm 1

When a Flow Module is first created, it initially searches for a suitable path connecting the source and the destination of the flow. After a path is found, forwarding rules matching the flow are installed at the switches along this path. Additionally, ingress and egress switches (i.e., the first and last switches) of the flow are configured to clone any incoming packets of the flow to the controller. Flow Module also starts a timer, T_s, for statistics collection operation. The expiration time of T_s is chosen from a uniform random distribution between 0 and $\mathcal{T}$ seconds when the timer first starts and each time it resets. Then, Flow Module starts to wait for packets from the Core Module (lines 1-5). This concludes the initialization phase of a Flow Module.

When a Flow Module receives a packet p from the Core Module, first a **PktID** for the packet is calculated. **PktID** values are message integrity codes (MIC) calculated

Algorithm 1 Flow Module Algoritm

Input: A packet received from Core Process p , Statistics timer T_s, a hash table mapping $PktID$s to T_{PktID}s H, set of statistics S, counter for correctly forwarded packets F, counter for expired timers E

Find a suitable $path$ for the flow.
for all s in $path$ **do**
    Add forwarding rules to s
end for
Start T_s with $UniformRandom(0,\mathcal{T})$
while true **do**
    if $p \leftarrow$ a packet received from Core Process **then**
        $PktID \leftarrow$ MIC(p)
        if p is from INGRESS **then**
            $H[PktID] \leftarrow T_{PktID}$
        else if p is from EGRESS **and** $PktID$ is in H **then**
            Remove $PktID$ from H
            Cancel timer for T_{PktID} and Update *AvgTripTime*
            $F \leftarrow F + 1$
        end if
    else if A timer T_{PktID} is expired **then**
        $E \leftarrow E + 1$
        if $(\frac{E}{|H| + F}) > \delta$ **then**
            return Verification Fails
        end if
    else if T_s expires **then**
        $S \leftarrow$ statistics from switches on flow's path
        for $i = 1; i < |S|; i++$ **do**
            if $(\left|\frac{S[i] - (|H| + F)}{|H| + F}\right| > \delta)$ **then**
                return Verification Fails
            end if
        end for
        for all s in $path$ **do**
            Reset counter of s for current flow
        end for
        $E \leftarrow 0, F \leftarrow 0$
        Start T_s with $UniformRandom(0,\mathcal{T})$
    end if
end while

over the data of the whole packet and are used to identify the packets. If p is from an ingress switch, Flow Module stores the **PKTID** in a hash table H along with a timer T_{PktID} which serves as the lifetime of the **PKTID** (lines 8-10). If p is from an egress switch, and **PKTID** exists in H, it means that p has left the network without any changes to its content and in the expected time frame without T_{PktID} expiring. In this case, Flow Module removes **PKTID** of p from H, cancels T_{PktID}, and increments its internal counter for correctly forwarded packets, F, by one (lines 11-15).

To keep track of the packets, Flow Module uses **PKTID**, which is a message integrity code (MIC) calculated over the data of the whole packet with a key only the controller knows. These **PKTID**s are used to identify individual packets and verify the integrity of a packet when it exits the network. When a Flow Module receives a packet, p, from the Core Module, first the **PKTID** for that packet is calculated. If p is from an ingress switch, it means that p is just entering the network and its **PKTID** is stored in a hash table, H, with **PKTID** as the key and a timer, T_{PktID}, as the corresponding value (lines 7-10). The timer is started as soon as it is inserted into H, and it is used as a tracking mechanism to see if p leaves the network in a timely manner. The value of T_{Pkt_ID} is determined by the *AvgTripTime* variable. *AvgTripTime* is the average time it takes for a packet to arrive at the egress switch after passing the ingress switch. Flow Module updates this value with each correctly forwarded packet. If p is from an egress switch, Flow Module searches H for the packet's **PKTID**. If H contains the **PKTID**, that means p exited the network in the expected time frame, and the T_{PktID} did not expire. Next, Flow Module removes the **PKTID** from H and cancels T_{PktID}. It also updates the *AvgTripTime* according to the elapsed time of the T_{PktID}. Flow Module also considers p as correctly forwarded and increments an internal counter for correctly forwarded packets, F (lines 11-15). In the case of p coming from an egress switch and H not containing its **PKTID**, it means that the ingress switch did not encounter p. There are two possible situations where this could be the case. First, if a switch starts injecting packets to the flow, new packets would not go through the ingress switch, and H would never contain their **PKTID**s. Second, some switch along the path is changing the contents of the packets. Assume that a packet p' enters the network through the ingress switch, and as the packet travels through the network, it transforms into p. In that case, after p' goes through the ingress switch H would

contain the **PKTID** of p'. Somewhere along its route, a switch would change p' into p and go through the egress switch as p. Later, when p leaves the network, Flow Module would not find p's **PKTID** in H. Moreover, the timer for the p' in H would expire and reveal that p' has entered the network but did not exit in the expected time frame.

If one of the T_{PktID}s in H expires before its **PKTID** can be matched, Flow Module marks that entry as expired and increases an internal counter for the expired packets, E. Then the Flow Module compares the ratio of number of expired packets to the total number of packets received from the ingress switch to a threshold value of δ. δ is configured by the network operator and denotes the amount of error that should be considered normal. If the inequality,

$$\frac{E}{F + |H|} > \delta \tag{3.1}$$

holds it means that too many counters in H expired. Therefore the verification fails, and the Flow Module alerts that this flow is experiencing some anomalous behavior (lines 16-20).

Additionally, to detect oddities in the forwarding mechanism, the Flow Module utilizes statistics collected from the switches. However, Flow Module does not collect statistics continuously or periodically. When the statistics are collected from the switches are governed by the timer T_s, which is started with a random value when the Flow Module is created. When T_s expires, Flow Module starts to collect statistics. Note that statistics are only collected from the switches that take part in the delivery of packets of the flow that the Flow Module is responsible for. Collected statistics are stored in an array called S, where S_i is the number of processed packets reported by the i^{th} switch along the flow's path. Under ideal conditions where no packets are dropped or delayed and every switch on the path exhibit normal forwarding behavior, all S_i should be equal for a flow. Moreover, the number of processed packets reported by each switch should be equal to the number of packets the controller receives from the ingress switch. In other words, under ideal conditions with no adversary, each switch reports the number of packets that enter the network.

Once the data collection process finishes, the controller compares the statistics reported by each switch to the total number of packets the ingress switch sent to the controller.

If the inequality,

$$\left| \frac{S_i - (|H| + F)}{|H| + F} \right| > \delta \tag{3.2}$$

holds, this means that i^{th} switch forwarded much less or much more packets than the original number of packets in the flow. Therefore, verification fails, indicating an anomaly in the forwarding behavior of the i^{th} switch (lines 21-27). After this verification, Flow Module resets the packet counters at switches, as well as its internal counters E and F. Then the T_s is restarted with a uniform random value between 0 and $\mathcal{T}$ (lines 28-32).

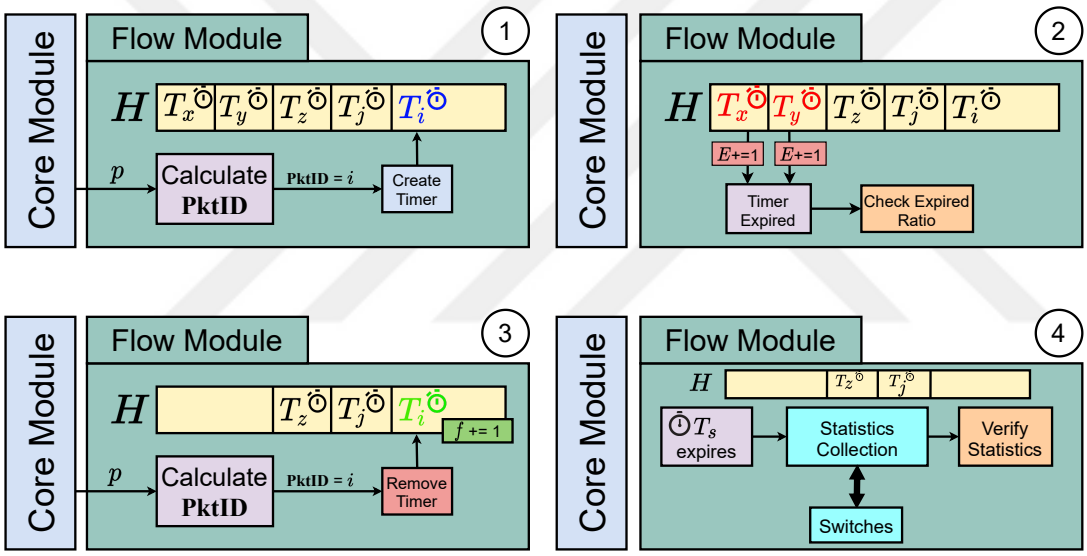


Figure 3.3: Overview of the operation of a Flow Module. (1) p_1 is received from ingress switch and a timer is started. (2) Some timers expire, increasing the E value. (3) p is received from egress switch, corresponding timer is removed. (4) T_s expires triggering a statistics collection.

When the controller starts to collect statistics from the switches, some packets may be in the process of traversing the path, or new packets may enter the network while the statistics collection continues. This can cause a difference in the number of processed packets reported by the switches. Consider a three-hop path, with switches sw_0, sw_1, and sw_2. Also, assume that when the controller starts to collect statistics, sw_1 and sw_2

are in the middle of forwarding a packet, such that the retrieved counter values are: $S_0 = x$, $S_1 = x-1$, $S_2 = x-2$. If the δ value is small enough, this kind of discrepancy in the S_i may trigger the Flow Module to detect an anomaly. To address this issue, we also create a modification to P4thV, called P4thV-Hold. In P4thV-Hold, before retrieving statistics from switches, the controller instructs the ingress switch to hold any incoming packet from entering the network and then waits a small amount of time based on the *AvgTripTime* to allow packets that are already in the network to leave the network. After the collection process ends, the controller instructs the ingress switch to work normally. Therefore, the controller prevents any new packets just entering the network or packets currently in the network from interfering with the collected statistics. How the ingress switch holds the incoming packets are explained in Section 3.2.2. However, this approach introduces a significant delay to the flow during the statistics collection period. Compared to P4thV-Hold, P4thV needs a greater δ value to account for this kind of interference. We inspect the performance impact of P4thV-Hold and compare the δ values of both approaches in Chapter 5.

3.2 P4 Switch Processing Logic

In order to enable P4 switches to work in tandem with our controller design, we need to add new packet processing logic to the data plane. In the following sections, we detail the changes we make to the parsing, ingress, and egress pipelines of P4 switches.

3.2.1 Packet Parsing Pipeline

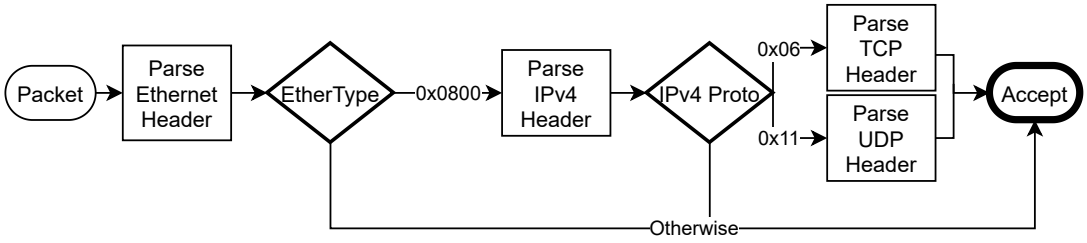


Figure 3.4: The flowchart of packet parsing pipeline.

Packet parsing pipeline of our P4 program parses headers starting from link layer up to transport layer depending on the protocol identifiers of each layer's header. In Figure 3.4, we show the overall process. When a packet arrives at a switch, first, the ethernet header is parsed, and the underlying protocol number is checked. Since we are only interested in packets of IPv4, we directly *accept* any other format, i.e., any packet that is not an IPv4 packet is passed to the ingress pipeline and parsed no further. If the packet is an IPv4 packet, we parse the IPv4 header and *accept* any packet that is not UDP or TCP packet. If the packet is either UDP or TCP packet, finally, we parse its transport layer header and pass the packet to the ingress pipeline.

3.2.2 Ingress Pipeline

After passing the parsing pipeline, the packet enters the ingress pipeline. The ingress stage has three match-action tables. One of the tables is used exclusively to determine the egress port to reach the next-hop, called *ForwardingTable*. The other two are used to count the packets of different flows and determine if this switch is the ingress switch for the flow. First, regardless of the type of the received packet, *ForwardingTable* is applied. If the packet is not a UDP or a TCP packet, it is directly passed to the egress pipeline. If the packet is either a UDP or a TCP packet, then the **CRC32** hash of the packet is calculated in the same way it is calculated at the controller to generate the **FLOWID** of the packet. The switch tries to match this **FLOWID** against the entries in the *KnownFlows* table. If a match is found, a copy of the packet is sent to the controller, and the original packet is passed on to the egress pipeline. If no match for **FLOWID** exists in the *KnownFlows* table, switch uses the *IsIngressSwitch* table to determine if it is the ingress switch for the packet's flow. If indeed the switch is the ingress switch, the packet is sent to the controller, overriding the forwarding information calculated using the *ForwardingTable*, for the creation of this new flow at the controller side. In the case the switch is not the ingress switch, no additional actions are performed, and the packet is passed to the egress pipeline.

To send packets to the controller, we use the packet cloning mechanism of the P4 Language. Using the cloning mechanism, we can create copies of packets and put them before various pipeline stages. Also, during these pipeline stages, we can check

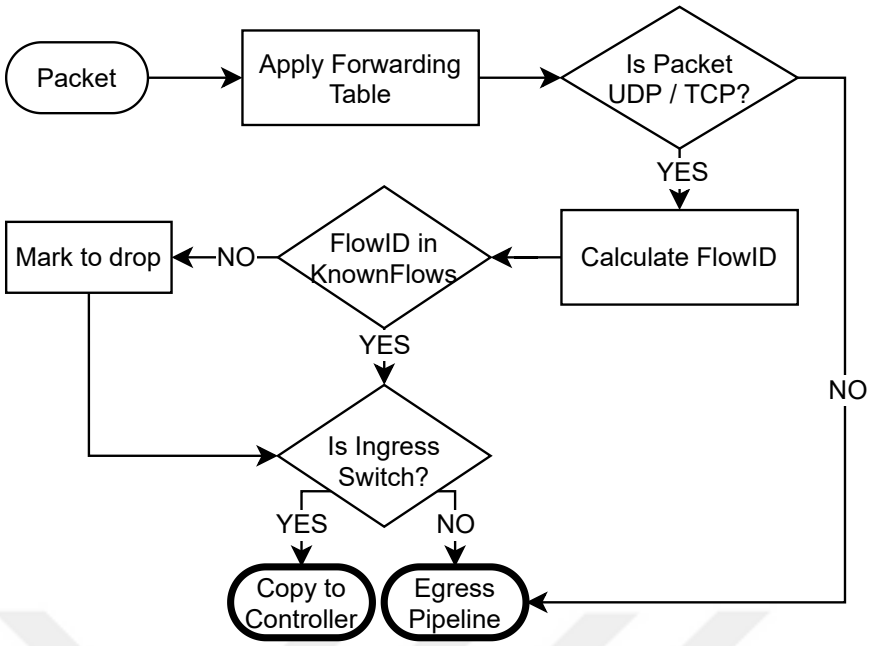


Figure 3.5: The flowchart of the ingress pipeline. Dotted arrows and boxes show the modifications of P4thV-Hold.

whether the packet that is currently being processed is a copy of the original packet. In our P4 program, when we send a packet to the controller, we always create and send a clone of the original packet. The fate of the original packet is decided depending on the situation. For example, when the switch receives a packet of a new flow, that packet needs to be sent to the controller so that the controller can allocate required resources and install forwarding rules, and the original packet is dropped at the switch. As another example, consider an already established flow. When a packet of this flow is received by the egress switch of the flow, the switch should report to the controller that the packet is about to leave the network. As a result, it creates a copy of the packet and sends it to the controller. Meanwhile, the original packet is also sent to its destination.

We also implement a packet holding mechanism that is used by P4thV-Hold. To implement this mechanism, we introduce a register to our P4 program called *Should-Hold* which holds a single bit flag. This register's value is only set and cleared by the controller before and after the statistics collection period. When a packet arrives at a switch, the switch first reads *ShouldHold* and either continues with the normal opera-

tion or resubmits the packet to the ingress pipeline without performing any operations. *Resubmit* mechanism of P4 works by putting the packet back on to the beginning of the ingress pipeline after the packet passes through the ingress pipeline, creating a rudimentary loop. The switch keeps resubmitting packet while the *ShouldHold* is set. When the controller clears the *ShouldHold* register, switch continues to process packets normally, as explained above. Note that only ingress switches perform this packet holding behavior.

3.2.3 Egress Pipeline

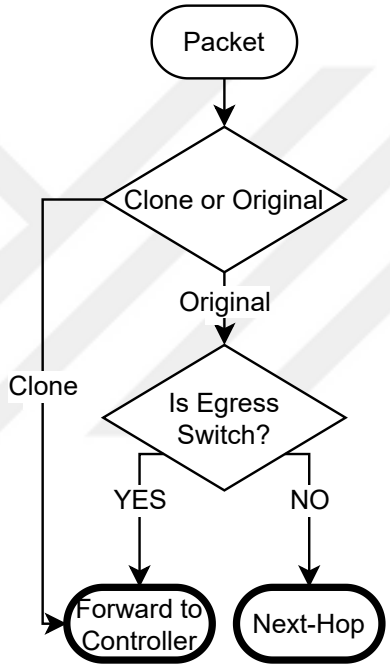


Figure 3.6: The flowchart of the egress pipeline.

After a packet passes the ingress pipeline, it arrives at the egress pipeline. The egress pipeline of our design is responsible for two things: (1) sending a copy of the packet to the controller if this switch is the egress switch of a flow, (2) detecting packets cloned in the ingress pipeline and taking action to send them to the controller. The switch separates cloned packets and original packets by inspecting a metadata field provided by the P4 Language. This metadata field indicates whether the packet is the original, a clone, resubmitted, etc. If the packet is not a cloned packet, the switch first checks whether it is the egress switch for the flow.

Similar to the ingress pipeline, we store an *IsEgressSwitch* table in the egress pipeline. This table is analogous to the *IsIngressSwitch* table at the ingress pipeline and works in the same way. If a match is encountered in *IsEgressSwitch*, a copy of the packet is created for the controller. The original packet then exits the egress pipeline. For the cloned packets, however, the egress pipeline modifies the ethernet header of the packets to have the EtherType of 0x969 and changes their output port to be the port connected to the controller. EtheType of 0x969 is used by the controller to identify the packets that are coming from the switches implementing P4thV.

3.2.4 Example

Now we explain how P4thV works on a small example. Consider a flow that will travel through three switches in the network, as shown in Figure 3.7. Flow enters the network through S_0, forwarded through S_1 and leaves the network via S_2.

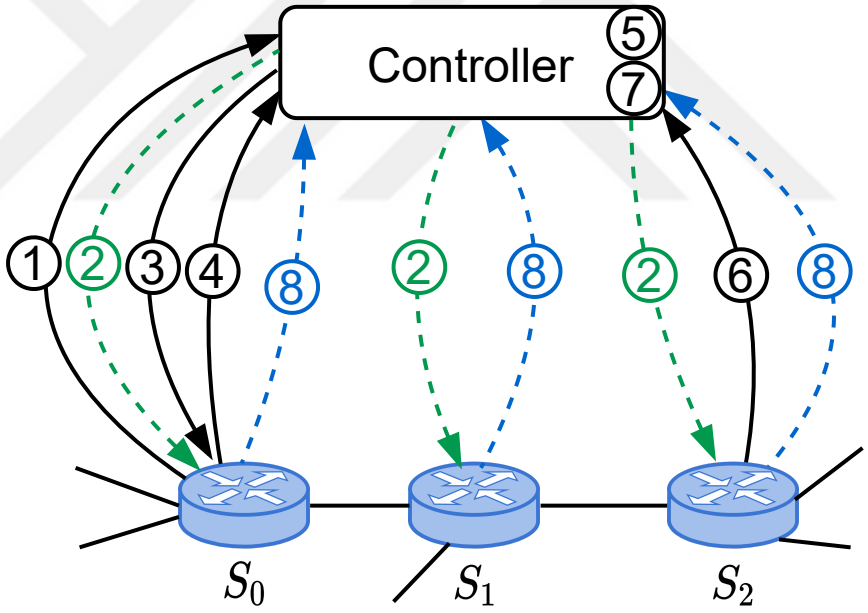


Figure 3.7: Overview of example. Numbers correspond to the steps in the example. Green arrows depict rule installation. Blue arrows depict statistics collection.

1. Packet p of flow f arrives at S_0. The switch computes the **FLOWID** of p. S_0 fails to match **FLOWID** of p to any entry in *KnownFlows* because f is a new flow in the network. As an ingress switch, it sends p to the controller.

2. Core Module at the controller receives p and computes its **FLOWID** to find the Flow Module responsible for flow f. However, since it is a new flow, no such Flow Module exists. The Core Module creates a new Flow Module for f. Newly created Flow Module, installs appropriate table entries to S_0, S_1, and S_2. It also starts a timer for statistics collection.

3. The controller sends p back to S_0.

4. Upon receiving p a second time, S_0 is now able to forward the packet to S_1 with the help of rules installed by the Flow Module. S_0 also generates a copy of p and sends it to the controller.

5. Controller forwards a copy of p to the Flow Module for f. Flow Module calculates the MIC of p and stores it, and starts a corresponding timer.

6. When p arrives at S_2, being the egress switch, S_2 creates a copy of p, p', and sends it to the controller, while forwarding p to its destination.

7. The Core Module, using the **FLOWID** of p', delivers p' to the correct Flow Module. Flow Module calculates the MIC of p' and compares it to the MIC value generated by p. If two MIC values are equal, then $p = p'$, meaning that p left the network exactly as it entered the network. If two MIC values differ, that means that the contents of p have been changed. Also, if the timer started at step 4 expires, it shows that p experienced unforeseen delays in the network, indicating a possible malicious behavior.

8. After some time, timer for statistics collection (started at step 2) expires. This starts the statistics collection from the switches, where the controller retrieves the number of packets of f processed from each switch. Flow Module responsible for f compares these numbers and tries to detect forwarding anomalies. If all retrieved counter values are the same, it means that the forwarding behavior of switches are correct. If one switch has a significantly different counter value, this means that there is a malicious or misbehaving switch.

3.2.5 Complexity Analysis

The time complexity of the Core Module of P4thV is $O(1)$, because **FLOWID** calculation and searching the existing Flow Modules in a hash table using **FLOWID** are both performed in constant time. Flow Module algorithms time complexity changes depending on the method for path calculation when the flow module is first created. This path calculation can be pre-computed and cached so that on Flow Module creation a simple lookup is performed. Complexity apart from the path generation of the Flow Module is $O(n)$ where n is the number of switches on a path. Most of the operations performed, except the statistics collection, are performed in constant time. Therefore, the time complexity is bounded by the statistics collection. Overall, P4thV's time complexity is $O(n)$, where n is the length of the paths in the network.

CHAPTER 4

SECURITY ANALYSIS

In this chapter, we discuss the security of P4thV. We consider various attacks on the data plane performed by misbehaving or malicious switches. For each attack, we examine how P4thV detects the attack. In particular, we consider packet delaying, dropping, tampering, and path deviation attacks. Additionally, we consider the case where two or more switches collude to perform an attack. Further, we examine the situations where the malicious switches report false information to the controller. We also present how our design is able to counter coward attacks.

4.1 Packet Delaying Attacks

A malicious switch performing a packet delaying attack will impose extra delays to the packets that go through it in order to disrupt and break the underlying communication. These additional delays experienced by packets will cause the T_{PktID}s stored in H of the corresponding Flow Module to expire. P4thV detects packet delaying attacks by keeping track of the expired timers. As a result, P4thV will identify packet delaying attacks as long as the delays experienced by packets are greater than the expiry periods of T_{PktID}s. Note that if the extra delays are not sufficient to expire T_{PktID} timers, the attacks will not be effective.

4.2 Packet Dropping Attacks

A malicious switch discards the packets of a specific flow. This will cause the number of processed packets reported by the predecessors and successors of the malicious

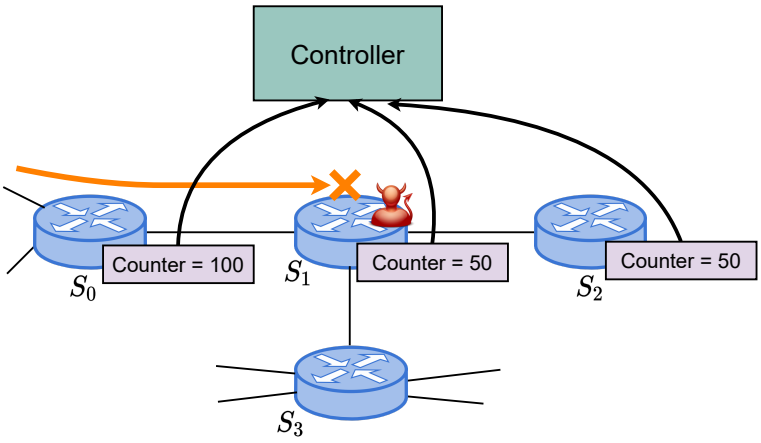


Figure 4.1: An example scenario of a packet dropping attack. Packets of a flow (orange arrow), are dropped by the compromised S_1 switch.

switch will be different. Since the successors never receive the dropped packets, the packet count will be smaller than the predecessors of the malicious switch. Leveraging this discrepancy, P4thV detects packet dropping attacks. Additionally, T_{PktID} timers of dropped packets will expire because the egress switch never reports their departure from the network to the controller. This allows P4thV to detect packet dropping attacks even if one or more downstream switches report fake statistics.

In Figure 4.1, an example scenario is shown. In the figure, S_1 is compromised by an attacker and it starts to drop packets of the flow depicted with the orange arrow after the first 50 packets. As seen in the figure, after collecting statistics from S_0, S_1, and S_2 controller can deduce that S_1's behavior is not as expected because it reports a significantly smaller counter value than its previous hop, S_0.

4.3 Path Deviation Attacks

Path deviation attacks are a special type of packet dropping attacks. Malicious switch alters the paths of packets such that packets leave the authorized path at some point and may or may not return to the authorized downstream switches. In the case they do not return to the authorized path, a detection mechanism for packet dropping attacks can be applied directly because from the view of the authorized path, packets leave the path and never return. In this case, the packets return to the original path, and since

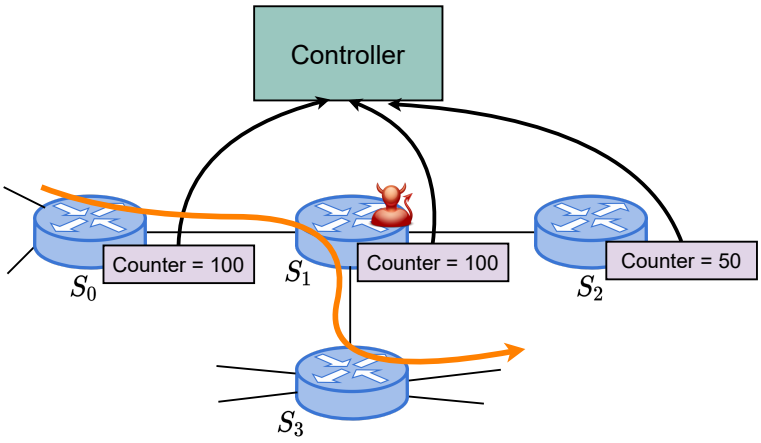


Figure 4.2: An example scenario of a path deviation attack. S_1 forwards the packets of a flow (orange arrow), out of the authorized path.

the packets skip over some switches, counter values reported by these will trigger the detection mechanism.

In Figure 4.2, a scenario similar to 4.1 is shown. In figure 4.2 instead of dropping the flow packets, S_1 forwards them over S_3, which is not in the authorized path. In that case, the counter values for S_0 and S_1 will report the correct number of packets because S_1 still processes the packets it deviates. However, S_2 will report a smaller counter value compared to S_0 and S_1 because the packets will not go through S_2 after S_1 changes their path. After statistics collection, the controller will notice this attack because the counter value reported by S_2 will be significantly smaller than its previous hop.

4.4 Packet Injection Attacks

Injected packets have to pass through an egress switch which will notify the controller of a packet exiting the network. However, notifications caused by injected packets will fail the verification process since they do not have any corresponding entries in H. Moreover, the controller can detect this type of attack with a statistics collection since the switches before the malicious switch will report fewer processed packets than the switches after the malicious switch.

4.5 Packet Tampering Attacks

When the contents of a packet are tampered with, the **PKTID** values generated by that packet at ingress and egress will not match since **PKTID** is directly generated from the contents of the packet. This will cause entries in H to be created when a packet passes through the ingress switch to expire because the modified packet's **PKTID** will not match these entries. Therefore, P4thV will detect tampering attacks.

4.6 Modifying Statistics Packets

A switch reporting fake statistics to disguise an attack can be detected using the statistics from the next-hop and previous-hop switches. Consider the situation where i^{th} switch on a path of a flow drops some or all packets that pass through it while reporting that it processed the same number of packets as $(i-1)^{th}$ switch. However, in this case $(i+1)^{th}$ switch will report less number of packets processed than i^{th} or $(i-1)^{th}$ switches. Therefore, the controller will still detect an anomaly. Also, consider the situation where i^{th} switch injects packets instead of dropping them. In this case, the number of packets reported by $(i+1)^{th}$ switch will be greater than $(i-1)^{th}$ switch, causing the verification to fail. Moreover, the packet injection attack can still be detected, as explained above.

In Figure 4.1, assume that S_1 reports the same counter value as S_0 by modifying statistics packets. In that case, S_2's counter value will still be smaller than S_1. Even though S_1 reports the same statistics as a benign switch, the attack will still be detected because of the statistics reported by the next-hop switch.

4.7 Coward Attacks

An adversary may first try to sample the statistics collection packets to infer the periods where statistics collection is not performed. However, since the timing of the statistics collections is random, an adversary cannot determine an interval in which he can evade detection.

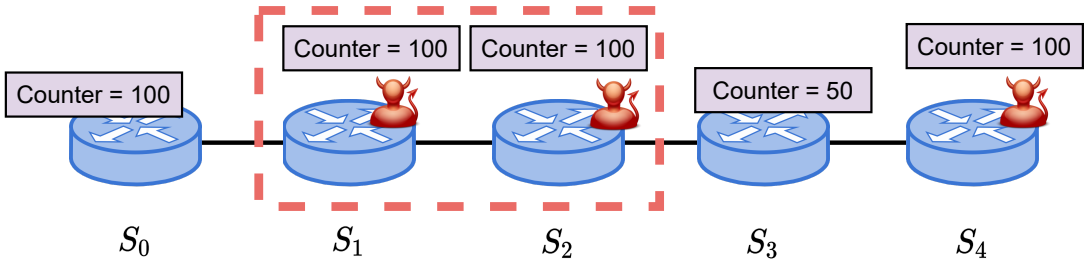


Figure 4.3: An example scenario of colluding attackers. S_1, S_2 and S_4 are colluding to elude detection. Since S_1 and S_2 are adjacent they can be considered as a single malicious entity.

4.8 Collusion Attacks

Switches may collude and perform any of the attacks discussed above. If the colluding malicious switches are adjacent, then all of the malicious switches can be considered as a single malicious entity to which the benign traffic enters. Therefore, each of the attacks discussed above can be detected as explained previously, as long as the number of colluding switches is smaller than the length of the whole path. In the case of colluding switches not being adjacent, each malicious switch can be detected by comparing their statistics to their neighbors.

In Figure 4.3 a scenario where S_1, S_2 and S_4 are colluding is shown. Assume that S_1 and S_2 collude to deviate the paths of the packets while modifying and reporting the same statistics as S_0. At the same time, S_4 injects the same number of packets S_1 and S_2 deviate in order to elude detection. However, due to S_3's statistics, the controller will detect the attack.

CHAPTER 5

RESULTS AND DISCUSSIONS

In this chapter, we create a prototype implementation of P4thV combining our controller design and data plane modifications and perform experiments to measure the performance and effectiveness of the proposed method. We first present the experiment setup, then we present our experiment results and discuss obtained results.

5.1 Experiment Setup

We build our network topology from the India35 problem instance (Figure 5.1) provided by SNDLib [1], which consists of 35 nodes connected by 80 links. In our network, each node is a P4 enabled switch, running the `simple_switch` implementation of BMv2 reference provided by the P4 Language Consortium. Additionally, we have 35 hosts in the network. Each host is connected to one of the 35 switches. Mininet emulation environment is used to setup the links between the switches. All switches and links are identical and share the same characteristics. Links have unlimited bandwidth and have no propagation delays in order to minimize the physical layer's impact on experiments. ETH Zurich's P4 utilities [80] are used to orchestrate the topology and their configurations. All of the emulations are performed on a computer with Intel Core i7-8700 CPU @3.20GHz and 8GB RAM.

In our experiments, we use different types of attacks in order to measure the performance of P4thV. For each experiment, we present the average of 50 runs. Various tools are used for implementing attacks. The construction of each attack is performed by scripts automatically. Packet delaying attacks are performed using the `tc` command. Specifically `tc` commands `netem` mode is used to add delays and

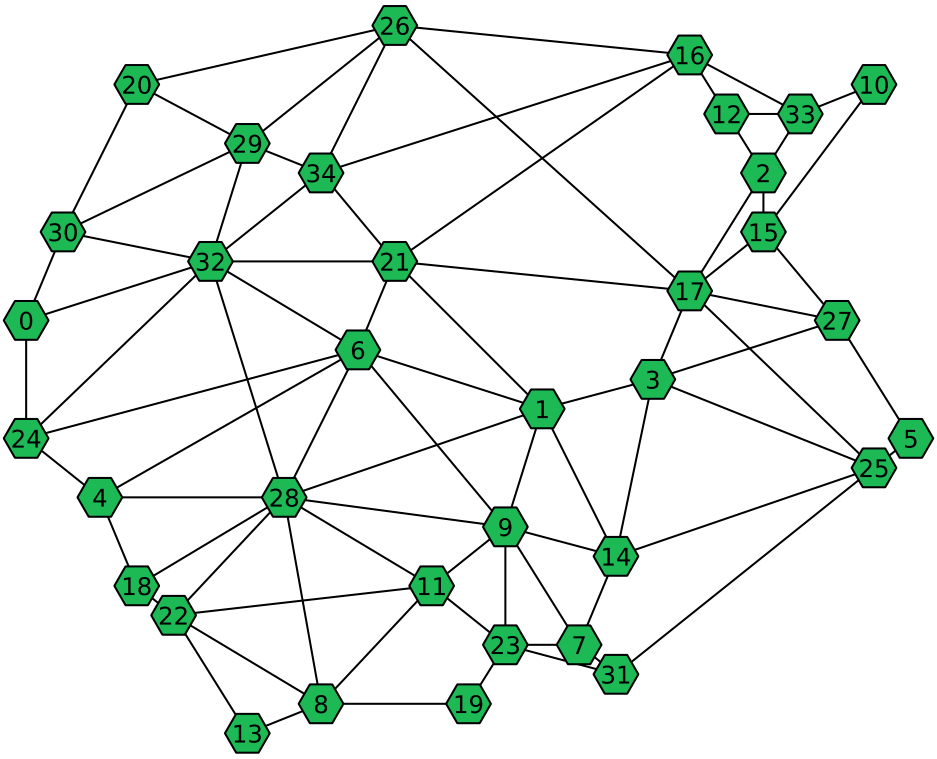


Figure 5.1: India35 problem instance of SNDLib [1] used in Mininet emulations.

reorder packets construct packet delaying attacks. For packet dropping and path deviation attacks Thrift API of the P4 switches are used to modify the actions of specific rules. For packet dropping, corresponding actions of some rules are modified to `mark_to_drop` action of the `simple_switch`. For path deviation, a series of forwarding rules are added to the switches on an existing path or a forged path. To record, replay and inject packets into the network for packet injection attacks `tcpreplay` tool is used. First, a `pcap` file is created from the recorded traffic. Then the pcap file is used to inject traffic into the network. Packet tampering attacks are performed by using the Thrift API of switches. Similar to packet dropping and path deviation attacks, new rules are added to the switches so that the packets matching these rules are modified.

We measure the performance of P4thV using three metrics:

1. *Detection Accuracy*. Due to its construction, P4thV may miss some attacks or detect attacks when there is none depending on the network performance

and configurable values in the algorithm (e.g., δ). Therefore, we evaluate our proposal under different network conditions and various attacks. We observe the effectiveness of the detection mechanism by observing the number of true-positive and false-positive detections.

2. *Detection Delay*. We define detection delay as the time it takes from the start of an attack to the detection by the detection mechanism. Shorter detection delays allow network operators to take actions quicker and reduce the impact of an attack on the network. To this end, we measure the influence of the algorithm parameters and different attacks on detection delays.

3. *Overhead*. While providing path verification, the detection mechanism should not cripple the network performance. An excessive computation or bandwidth overhead can impede the forwarding performance of the switches. As a result, high overhead may cause the design to become infeasible to deploy. Therefore, we measure the impact of P4thV on forwarding delays, network throughput, control channel messages, and CPU consumption to see the consequences of using P4thV on network performance.

5.2 Selection of δ

Network performance directly influences the performance of the detection process. P4thV may detect packets that experience processing delays or packets that are dropped due to filled queues at switches as an attack when a small δ value is selected. On the other hand, with large δ values, P4thV may fail to detect attacks because the number of anomalous packets may not be enough to trigger detection. Therefore, the chosen δ value should be able to endure detection noise generated by network performance while still being adequate to identify potential attacks efficiently.

In other words, δ works as a threshold value to weed out anomalies that normally occur on a network, such as packet delays. In an ideal network where no extra packet delays occur and no packets are dropped, $\delta = 0$ would cause P4thV to report an anomaly even a single packet gets dropped or delayed. However, in real-world networks occurrence of extra delays and packet losses are possible in normal working

conditions. δ value provides us a way to minimize these occurrences from being detected as abnormal behavior.

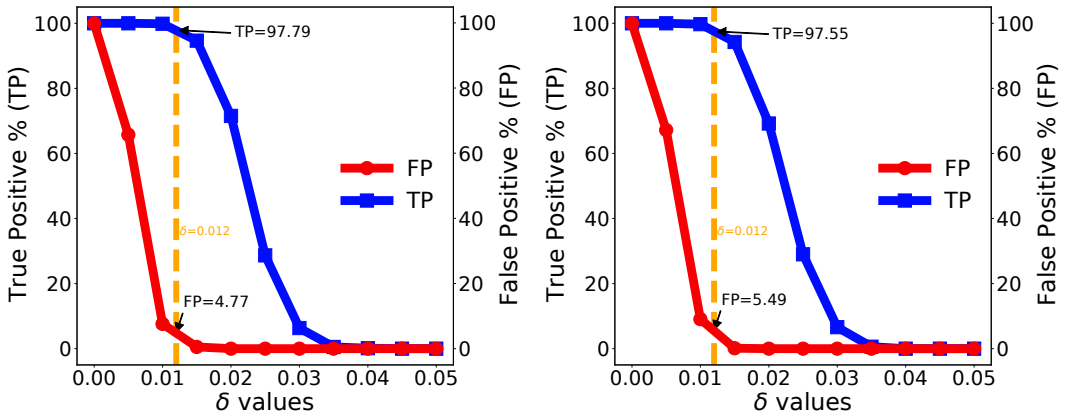
In order to select an appropriate δ value that can tolerate noise in detection while maintaining high accuracy in attack detection, we measure the effect of varying δ values on the performance of P4thV under packet delaying attacks. We choose packet delaying attacks because their detection depends only on the value of the number of expired packet timers and δ in our algorithm.

In our experiment, we choose two hosts connected by 4 switches as the source and destination of a flow. Source sends UDP packets with a payload size of 4 bytes (32 bytes total with IP and UDP headers) to the destination at a rate of 250 packets/second for 60 seconds. Moreover, we configure the switches such that every packet has a 1% chance to experience an additional 20 ms delay to simulate the delays that may occur in a real-world network. Note that the amount of delay the packet experiences does not matter as long as the delay causes the packet timer at the controller to expire. We then randomly select one of the switches to be the malicious switch. The malicious switch starts to delay every packet by 20 ms 30 seconds after the traffic starts.

Under these conditions, we observe the effect of δ on our detection accuracy as in Figure 5.2a.

As δ increases, the true-positive ratio starts at 100% and slowly decreases until $\delta = 0.01$, then it starts to decrease dramatically. On the other hand, the false-positive ratio starts at around 100% and rapidly decreases to around 5% at $\delta = 0.01$. Both true-positive and false-positive ratios keep decreasing as δ values increase. Considering these results, we choose δ value as 0.012 as it provides 97.8% true-positive rate and 4.8% false-positive rate.

We next perform a similar experiment with path detour attacks. Where a randomly selected switch on the path has a 1% chance to forward packets to a downstream switch via a different path. This causes the reports to match from ingress and egress switches. Thus the attack can only be detected by statistics collection. Again we see that the true-positive ratio starts decreasing as the δ value increases, as well as the false-positive ratio (Figure 5.2b). This confirms our previous δ selection is suitable



(a) (b)

Figure 5.2: Impact of δ on detection accuracy on a flow where each packet has (a) a 1% chance of experiencing excessive delays that expires the packet timer at the controller, (b) a 1% chance of being dropped.

for this kind of network performance.

Further, we evaluate the accuracy of different δ values under different attacks where the packets both have a chance to be delayed or dropped (Figure 5.3). This further confirms the δ value. So we fix our δ value as 0.012 for the rest of the experiments.

5.3 Detection Accuracy

For detection accuracy experiments, we generate 60 flows between 30 randomly selected pairs of hosts. Each pair generates two flows: one from the first host of the pair to the second and one from the second host to the first host. In total, 60 flows create 600 packets per second into the network. Under these conditions, we perform various attacks on 10 randomly selected flows and measure the detection accuracy of P4thV. We also perform the same experiments with FOCES to compare P4thV's performance against FOCES. We present our findings in Figure 5.4. We can see that P4thV can detect all types of attacks with an average true-positive rate of 97.2%. However, we also note that P4thV generates 4.3% false-positive detections on average. This is because of the underlying packet loss and delays caused by natural network phenomena. Par-

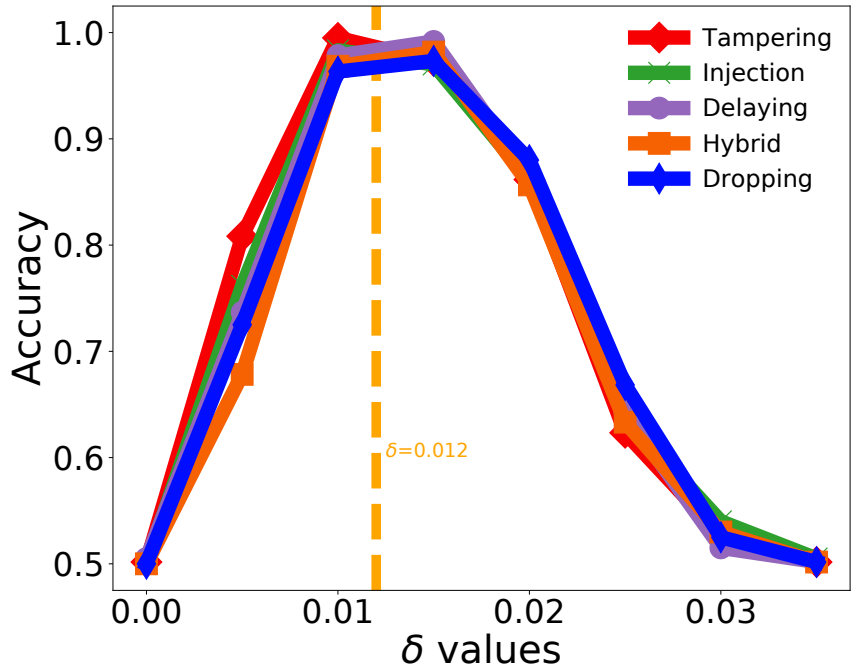


Figure 5.3: Effect of δ on network with both delay and loss chance under different attacks.

ticularly, P4thV has significantly more false-positives under packet dropping attacks. This suggests that P4thV is more sensitive against packet dropping attacks.

Compared to FOCES, P4thV achieves a greater true-positive ratio. However, unlike P4thV, FOCES cannot detect packet delays, packet tampering, and some special cases of path deviation. Especially under hybrid attacks, P4thV performs significantly better than FOCES. This is caused by FOCES not being able to detect attacks where the same number of packets are taken out from and injected into the flow. On the other hand, FOCES achieves lower rates of false-positive reports under different attacks.

We repeat the experiments with different $\mathcal{T}$ values to see how $\mathcal{T}$ affects the detection accuracy. We observe that the value of $\mathcal{T}$ does not have a significant effect on detection accuracy.

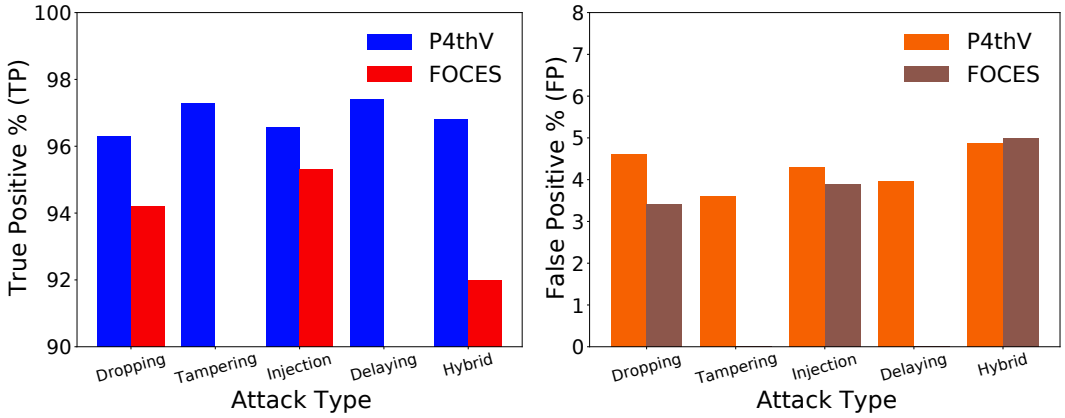


Figure 5.4: Comparison of true-positive and false-positive percentages of P4thV and FOCES under different attacks.

5.4 Detection Delay

We examine the detection delay of P4thV under different attacks. We define detection delay as the time it takes P4thV to detect an anomaly after an attack starts. Our experiment setup is similar to the experiment setup in the previous section. We have 60 flows between 30 pairs of hosts, and 60 flows introduce 600 packets into the network every second. We then randomly choose 10 flows to attack and start attacks on these flows.

Because path shortcut attacks can only be detected by the statistics collection process, we initially examine the effect of $\mathcal{T}$ on detection delays under path shortcut attacks. Since $\mathcal{T}$ only affects the interval of statistics collection, we find using path shortcut attacks is sufficient.

As seen in Figure 5.5, we observe that the detection delays increase as $\mathcal{T}$ increases. However, detection times are always shorter than the corresponding values of $\mathcal{T}$.

We also observe the detection delays of P4thV and FOCES under different attacks. In this experiment we fix $\mathcal{T} = 5$. In Figure 5.6, comparing P4thV and FOCES, we can see that P4thV is faster than FOCES while detecting attacks. Additionally, P4thV detects most of the packet tampering and packet delaying attacks quicker than other types of attacks, whereas FOCES fails to detect tampering and delaying attacks, as

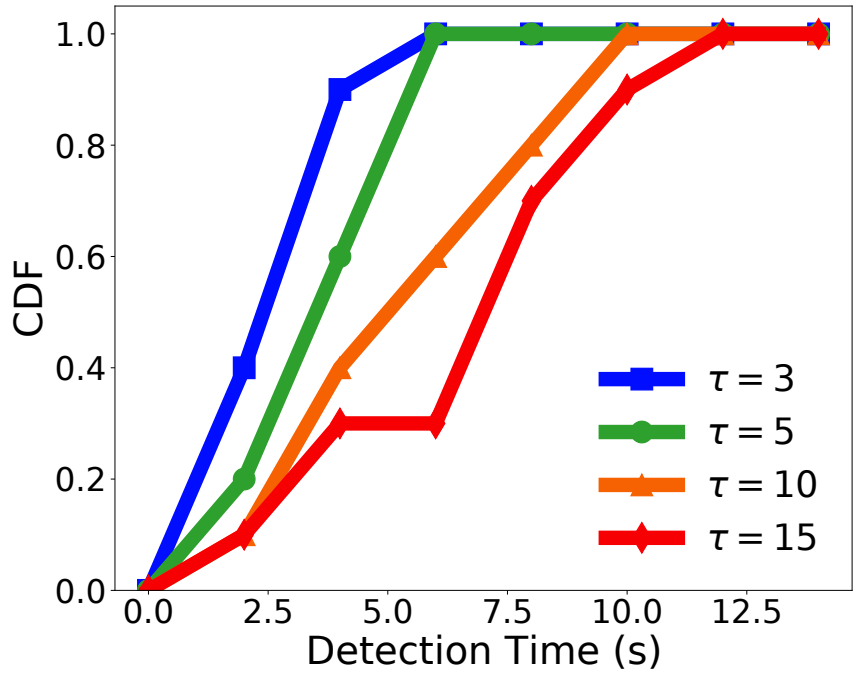


Figure 5.5: Detection delays with different $\mathcal{T}$ values under path shortcut attacks.

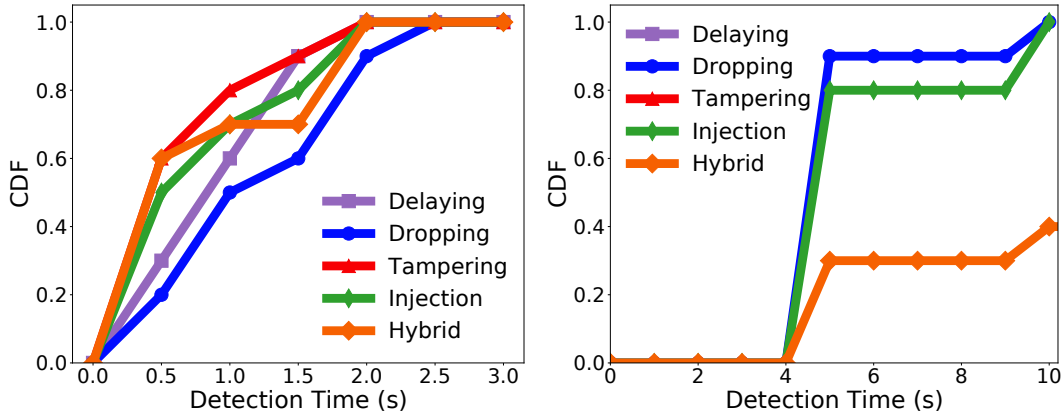
well as some of the hybrid attacks.

We show the detection delays of P4thV and FOCES as network size increases on Figure 5.6c. While FOCES is faster than P4thV for smaller networks, as the network size grows, detection delays increase dramatically. The increase in P4thV's detection delay is significantly slower than FOCES. This difference is explained by the number of switches that need to be queried for anomaly detection on a single flow. While P4thV only gathers information from switches on the path of the flow, FOCES needs to gather information from switches outside of the path as well, making P4thV more tolerant to network size compared to FOCES.

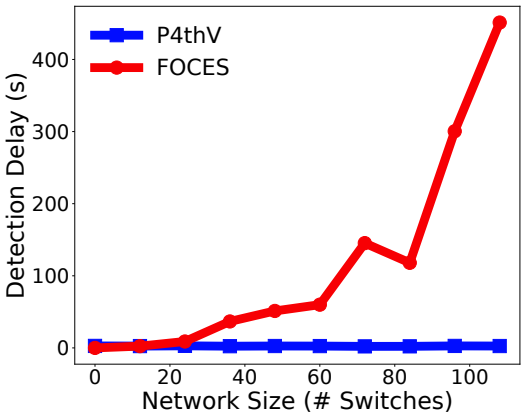
5.5 Overhead

We examine the overhead of P4thV by measuring its impact on forwarding delays, CPU consumption of the controller, network throughput, and the number of control channel messages.

In our experiments, we compare P4thV against different designs. One of these designs



(a) (b)



(c)

Figure 5.6: Detection delays of (a) P4thV and (b) FOCES under different attacks. (c) depicts the detection delays of FOCES and P4thV with increasing network size.

is what we call *Simple Forwarding*. In *Simple Forwarding*, the controller precomputes and installs forwarding rules to switches at network creation. This allows switches to forward packets without the help of the controller, similar to traditional switches. In fact, after installing the rules, the controller can be disconnected from the network.

Another design we use is what we call *RuleAdd*. In *RuleAdd* switches do not have any pre-installed rules. When a packet arrives at an ingress switch, it notifies the controller of the new flow and the controller installs flow rules to appropriate routers according to the network policies. The controller's only responsibility is to install forwarding rules when a new flow enters the network.

The above two designs do not provide path verification or packet integrity. Instead, we consider them as a baseline to see how much overhead the P4thV creates.

First, we examine the influence of P4thV on forwarding delays. In our network topology, we choose 10 pairs of hosts that generate 20 flows. We then measure the round-trip-time (RTT) on these flows. We use the `hping3` program to generate ping packets. However, we configure the arguments of `hping3` to send all ping packets over a single flow. In Figure 5.7 we show the RTT values we measure for different controller designs. On average, RTT values of P4thV are 1ms higher than the rest of the designs. Also, in P4thV and *RuleAdd*, flows experience an initial spike caused by the controller computing and installing rules. Additionally, for P4thV, the controller needs to create a Flow Module for each new flow, increasing the initial delays.

In Figure 5.8a, we also measure the P4thV-Hold's RTT values among other designs. We observe that P4thV-Hold experiences random spikes in RTT values over the lifetime of a flow. These spikes are directly caused by the statistics collection mechanism of the P4thV-Hold. Since it stops all the flow traffic during the statistics collection period, each time P4thV-Hold starts collecting statistics, flow packets experience additional delays.

We then compare the network throughput under different controller designs. Figure 5.8b shows that the throughput impact of P4thV is higher than FOCES. P4thV's effect on the throughput is caused by the switches' increased communication with the controller and the clone operations performed at switches. Compared to P4thV, the throughput degradation caused P4thV-Hold is even higher, around 40%. This huge throughput degradation is caused by P4thV-Hold's packet holding mechanism during statistics collection, in which the forwarding operations of the whole flow are stopped.

We also measure the number of control channel messages with an increasing number of flows. In this experiment, we also include SPHINX in our evaluation along with FOCES. Overall, SPHINX generates around two times the number of control channel messages generated by P4thV. Meanwhile, FOCES only requires an even smaller amount of control channel messages to work. Since FOCES do not generate any packets when packets enter and exit the network, the number of overall control channel packets generated by FOCES is smaller. Figure 5.8c shows the relationship

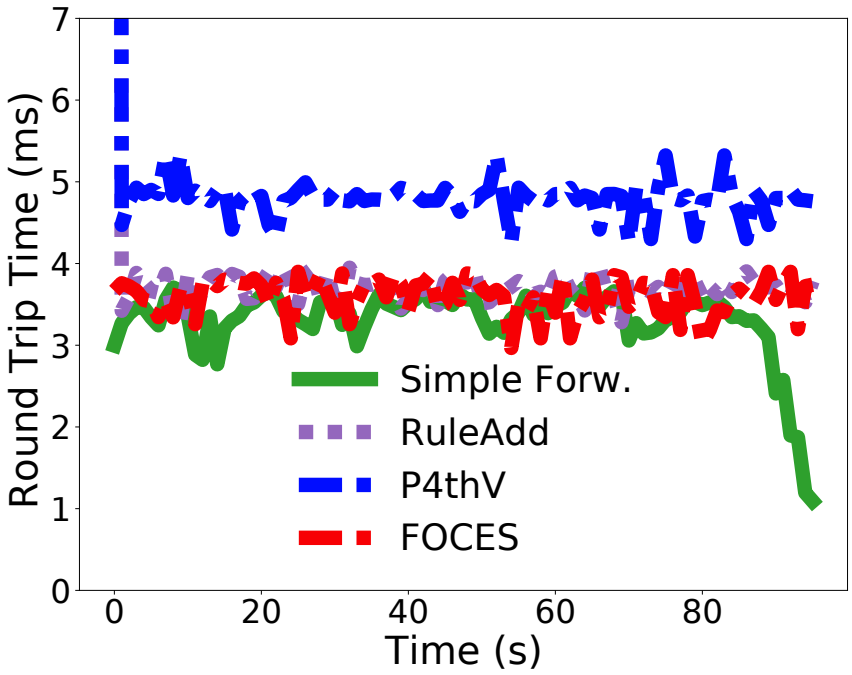


Figure 5.7: Round trip times of different controller designs throughout the lifetime of a flow.

between the number of controller messages sent under each design.

Finally, we measure the CPU consumption of the controller while different controller designs are employed. Overall, P4thV uses more CPU cycles compared to FOCES. However, FOCES has usage spikes where it performs taxing matrix operations on the CPU. Both FOCES and P4thV requires more computation power than *SimpleForwarding* or *RuleAdd*.

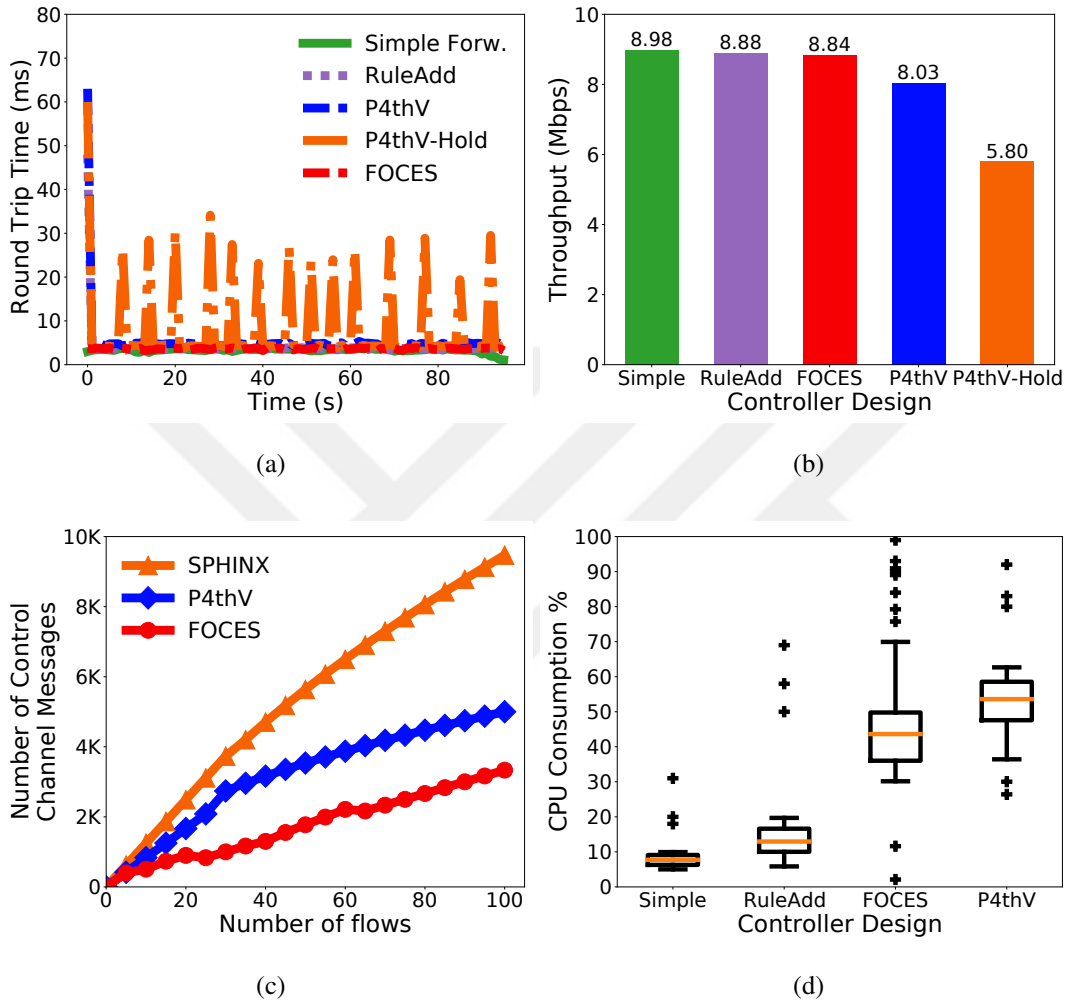


Figure 5.8: (a) Effect on throughput. (b) Number of control channel messages with increasing number of flows (same pps). (c) CPU consumption of controller on different algorithms. (d) RTT values of different designs compared with P4thV-Hold.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this chapter, we conclude the thesis and present future work that aims to improve P4thV.

6.1 Conclusion

SDN's data plane is vulnerable to attacks that change the forwarding behavior of the data plane devices. However, the controller in SDN does not have the means to verify the paths taken by the packets in the network. This issue is often neglected due to the common assumption that the data plane of SDN is trusted.

In this paper, we define the path verification problem and various data plane attacks that result in changes in data plane behavior. We describe P4thV, an extension to the SDN controller that provides path verification and packet integrity. P4thV works by gathering packet information from ingress and egress switches of flows in the network and collecting flow statistics from switches responsible for forwarding to detect attacks violating the authorized paths and changes in forwarding behavior. Moreover, we implement P4thV using Python and P4 enabled software switches. We evaluate the effectiveness of P4thV in a Mininet environment under adversarial settings. We compare our results to existing work: FOCES and SPHINX. Our experiment results show that P4thV is able to detect attacks against the data plane with 97.2% accuracy and achieves $2\times$ faster detection compared to FOCES. P4thV achieves this performance while requiring 50% less control channel messages than SPHINX. We observe that P4thV causes 10% throughput degradation and negligible additional forwarding delays.

6.2 Future Work

As future work, we plan to extend the capabilities of P4thV to localize compromised switches. Currently, our design is only able to reduce the set of possible malicious switches from the switches in the whole network to the switches taking part in the forwarding of the flow the attack is detected on. Moreover, the statistics collection process already gives a rough idea about the location of the malicious switch. Finding out the source switch of the anomalous behavior would help the network operators to reduce the impact of attacks and quickly mitigate attacks. One idea could be to allow all switches on a path to generate notifications for the controller as packets pass through. This way, the controller can keep a set of suspected switches and prune this set according to the correctness of the notifications received from the switches on the path, exposing the malicious switch in the end.

One of the shortcomings of P4thV is that it requires all packets to generate controller messages for all packets at ingress and egress switches for every flow. While this approach helps with faster detection of anomalies on the data plane, for benign flows, it just increases the overhead. Moreover, with this approach, when flows introduce a high number of packets per second, the controller may become congested and slow down the whole detection process. To address this issue, we plan to approach the problem in a probabilistic manner such that ingress and egress switches generate controller messages only for a subset of the packets they process. One idea is to make the probability of generating a controller message depend on the closeness of the values of ratios in equation 3.1 and equation 3.2 to the δ value. When the ratios are closer to the δ value, ingress and egress switches will generate controller messages with higher probabilities. This will result in the controller inspecting suspicious flows more closely. However, we expect that an approach along these lines would reduce the overhead significantly but would also negatively impact the detection accuracy. We plan to inspect this trade-off between accuracy and overhead as well.

To broaden the use-cases of our design, we plan to perform further experiments on different network topologies and compare their performance against the ever-growing number of state-of-the-art solutions. These experiments and comparisons would highlight the drawbacks and strengths of our design.

REFERENCES

[1] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski, "Sndlib 1.0—survivable network design library," *Networks: An International Journal*, vol. 55, no. 3, pp. 276–286, 2010.

[2] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27–51, 2014.

[3] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *Proc. of the 17th IEEE International Conference on Network Protocols*, pp. 123–132, IEEE, 2009.

[4] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[5] Open Networking Foundation, "Software-defined networking:the new norm for networks." Open Networking Foundation `http://opennetworking.wpengine.com/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf`, 2012.

[6] S. Din, M. M. Rathore, A. Ahmad, A. Paul, and M. Khan, "Sdiot: Software defined internet of thing to analyze big data in smart cities," in *Proc. of the 42nd IEEE Conference on Local Computer Networks Workshops (LCN Workshops)*, pp. 175–182, 2017.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, p. 69–74, March 2008.

[8] T. Guodong, Q. Xi, and C. Chaowen, "A sdn security control forwarding mechanism based on cipher identification," in *Proc. of the 9th IEEE International Conference on Communication Software and Networks (ICCSN)*, pp. 1419–1425, IEEE, 2017.

[9] H. Zhang, F. Tang, and L. Barolli, "Efficient flow detection and scheduling for sdn-based big data centers," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 5, pp. 1915–1926, 2019.

[10] K. Bhushan and B. B. Gupta, "Distributed denial of service (ddos) attack mitigation in software defined network (sdn)-based cloud computing environment," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 5, pp. 1985–1997, 2019.

[11] A. A. Diro, H. T. Reda, and N. Chilamkurti, "Differential flow space allocation scheme in sdn based fog computing for iot applications," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–11, 2018.

[12] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, "Let sdn be your eyes: Secure forensics in data center networks," in *Proc. of the NDSS workshop on security of emerging network technologies (SENT'14)*, pp. 1–7, 2014.

[13] J. R. Ballard, I. Rae, and A. Akella, "Extensible and scalable network monitoring using opensafe," *Proc. of the INM/WREN'10*, vol. 10, 2010.

[14] S. Shin and G. Gu, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," in *Proc. of the 20th IEEE international conference on network protocols (ICNP)*, pp. 1–6, IEEE, 2012.

[15] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an openflow architecture," in *Proc. of the 23rd International Teletraffic Congress (ITC)*, pp. 1–7, IEEE, 2011.

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

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[18] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *Proc. of the ACM SIGCOMM 2015*, vol. 15, 2015.

[19] D. C. Plummer, "An ethernet address resolution protocol," RFC 826, RFC Editor, November 1982.

[20] Cisco, "Cisco ios software configuration guide, release 12.2sx," 2013. Accessed on 18.10.2021.

[21] C. L. Abad and R. I. Bonilla, "An analysis on the schemes for detecting and preventing arp cache poisoning attacks," in *Proc. of the 27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07)*, pp. 60–60, IEEE, 2007.

[22] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proc. of the 2018 Conference of the ACM Special Interest Group on data communication*, pp. 490–503, 2018.

[23] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proc. of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pp. 518–532, 2018.

[24] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in p4 programs with assertion-based verification," in *Proc. of the Symposium on SDN Research*, pp. 1–7, 2018.

[25] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of p4 programs in feasible time using assertions," in *Proc. of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pp. 73–85, 2018.

[26] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for p4 programs," in *Proc. of the Symposium on SDN Research*, pp. 1–7, 2018.

[27] L. Ryzhyk, N. Bjørner, M. Canini, J.-B. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese, "Correct by construction networks using stepwise refinement," in *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 683–698, 2017.

[28] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime verification of p4 switches with reinforcement learning," in *Proc. of the 2019 Workshop on Network Meets AI & ML*, pp. 1–7, 2019.

[29] G. Pickett, "Staying persistent in software defined networks," *Black Hat Briefings*, 2015.

[30] M. Antikainen, T. Aura, and M. Särelä, "Spook in your network: Attacking an sdn with a compromised openflow switch," in *Proc. of the Nordic conference on secure IT systems*, pp. 229–244, Springer, 2014.

[31] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised sdn switch," in *Proc. of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pp. 1–6, IEEE, 2015.

[32] R. Klöti, V. Kotronis, and P. Smith, "Openflow: A security analysis," in *Proc. of the 21st IEEE International Conference on Network Protocols (ICNP)*, pp. 1–6, IEEE, 2013.

[33] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[34] D. Kreutz, F. M. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proc. of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pp. 55–60, 2013.

[35] P. Perešíni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proc. of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pp. 1–13, 2015.

[36] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect sdn forwarding with rulescope," in *Proc. of the IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, 2016.

[37] P. Zhang, C. Zhang, and C. Hu, "Fast testing network data plane with rulechecker," in *Proc. of the 25th IEEE International Conference on Network Protocols (ICNP)*, pp. 1–10, IEEE, 2017.

[38] P. Zhang, H. Li, C. Hu, L. Hu, and L. Xiong, "Stick to the script: Monitoring the policy compliance of sdn data plane," in *Proc. of the 2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 81–86, IEEE, 2016.

[39] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *Proc. of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pp. 19–33, 2016.

[40] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 290–301, 2011.

[41] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 113–126, 2012.

[42] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI) 13)*, pp. 15–27, 2013.

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

[44] T. P. A. W. Group, "P4runtime specification," tech. rep., P4 language consortium, 2021. Accessed on 09.10.2021.

[45] The Open Networking Foundation, "Openflow switch specification version 1.5.1," tech. rep., The Open Networking Foundation, 2015. Accessed on 09.10.2021.

[46] M. Ullmann and M. Vögeler, "Delay attacks—implication on ntp and ptp time synchronization," in *Proc. of the 2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pp. 1–6, IEEE, 2009.

[47] S. Deng, X. Gao, Z. Lu, and X. Gao, "Packet injection attack and its defense in software-defined networks," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 695–705, 2017.

[48] P. M. Jawandhiya, M. M. Ghonge, M. Ali, J. Deshpande, *et al.*, "A survey of mobile ad hoc network attacks," *International Journal of Engineering Science and Technology*, vol. 2, no. 9, pp. 4063–4071, 2010.

[49] B. Kannhavong, H. Nakayama, Y. Nemoto, N. Kato, and A. Jamalipour, "A survey of routing attacks in mobile ad hoc networks," *IEEE Wireless communications*, vol. 14, no. 5, pp. 85–91, 2007.

[50] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[51] U. Venkanna and R. L. Velusamy, "Black hole attack and their counter measure based on trust management in manet: A survey," in *Proc. of the 3rd International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom 2011)*, pp. 232–236, IET, 2011.

[52] B. Liu, J. T. Chiang, J. J. Haas, and Y.-C. Hu, "Coward attacks in vehicular networks," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 14, no. 3, pp. 34–36, 2010.

[53] P. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, and C. Hu, "Foces: Detecting forwarding anomalies in software defined networks," in *Proc. of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 830–840, IEEE, 2018.

[54] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: detecting security attacks in software-defined networks.," in *NDSS*, vol. 15, pp. 8–11, 2015.

[55] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, 2021.

[56] R. Lychev, M. Schapira, and S. Goldberg, "Rethinking security for internet routing," *Communications of the ACM*, vol. 59, no. 10, pp. 48–57, 2016.

[57] S. Kent, C. Lynn, and K. Seo, "Secure border gateway protocol (s-bgp)," *IEEE Journal on Selected areas in Communications*, vol. 18, no. 4, pp. 582–592, 2000.

[58] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy, "Highly secure and efficient routing," in *Proc. of the IEEE INFOCOM 2004*, vol. 1, IEEE, 2004.

[59] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical network support for ip traceback," in *Proc. of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 295–306, 2000.

[60] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer, "Hash-based ip traceback," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 3–14, 2001.

[61] C. A. Sunshine, "Source routing in computer networks," *ACM SIGCOMM Computer Communication Review*, vol. 7, no. 1, pp. 29–33, 1977.

[62] S. Yu, W. Zhou, R. Doss, and W. Jia, "Traceback of ddos attacks using entropy variations," *IEEE transactions on parallel and distributed systems*, vol. 22, no. 3, pp. 412–425, 2010.

[63] J. Naous, M. Walfish, A. Nicolosi, D. Mazieres, M. Miller, and A. Seehra, "Verifying and enforcing network paths with icing," in *Proc. of the Seventh Conference on Emerging Networking Experiments and Technologies*, pp. 1–12, 2011.

[64] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig, "Lightweight source authentication and path validation," in *Proc. of the 2014 ACM Conference on SIGCOMM*, pp. 271–282, 2014.

[65] B. Wu, K. Xu, Q. Li, Z. Liu, Y.-C. Hu, M. J. Reed, M. Shen, and F. Yang, "Enabling efficient source and path verification via probabilistic packet marking," in *Proc. of the 26th IEEE/ACM International Symposium on Quality of Service (IWQoS)*, pp. 1–10, IEEE, 2018.

[66] X. Liu, A. Li, X. Yang, and D. Wetherall, "Passport: Secure and adoptable source authentication.," in *NSDI*, vol. 8, pp. 365–378, 2008.

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

[68] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," in *Proc. of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 239–250, IEEE, 2015.

[69] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proc. of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 413–424, 2013.

[70] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proc. of the 1st ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pp. 121–126, 2012.

[71] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 99–111, 2013.

[72] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *Proc. of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pp. 19–33, 2016.

[73] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, "Sdnsec: Forwarding accountability for the sdn data plane," in *Proc. of the 25th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–10, IEEE, 2016.

[74] P. Zhang, "Towards rule enforcement verification for software defined networks," in *Proc. of the IEEE INFOCOM 2017*, pp. 1–9, IEEE, 2017.

[75] S. Agrawal and D. Boneh, "Homomorphic macs: Mac-based integrity for network coding," in *Proc. of the International Conference on Applied Cryptography and Network Security*, pp. 292–305, Springer, 2009.

[76] Z. Zuo, C. Chang, Y. Zhang, R. He, X. Qin, and K. L. Yung, "P4label: packet forwarding control mechanism based on p4 for software-defined networking," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–14, 2020.

[77] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "Sdn traceroute: Tracing sdn forwarding without changing network behavior," in *Proc. of the 3rd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pp. 145–150, 2014.

[78] T.-W. Chao, Y.-M. Ke, B.-H. Chen, J.-L. Chen, C. J. Hsieh, S.-C. Lee, and H.-C. Hsiao, "Securing data planes in software-defined networks," in *Proc. of the IEEE NetSoft Conference and Workshops (NetSoft)*, pp. 465–470, IEEE, 2016.

[79] C. Pang, Y. Jiang, and Q. Li, "Fade: Detecting forwarding anomaly in software-defined networks," in *Proc. of the 2016 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2016.

[80] E.-Z. N. S. Group, "Eth-zurich p4 learning repository." `https://github.com/nsg-ethz/p4-learning`, 2021.