UNIVERSIDADE Đ
COIMBRA

Rui Manuel Marques Pires

SECURITY FOR SDN ENVIRONMENTS WITH P4

Dissertation in the scope of the Master in Informatics Security, advised by Prof. Paulo Simões and Prof. Tiago Cruz and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

Department of Informatics Engineering

Rui Manuel Marques Pires

Security for SDN environments with P4

Dissertation in the scope of the Master in Informatics Security, advised by Prof. Paulo Simões and Prof. Tiago Cruz and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Rui Manuel Marques Pires

Segurança em ambientes SDN com P4

Dissertação no âmbito do Mestrado em Segurança Informática, orientada pelo Professor Doutor Paulo Simões e Professor Doutor Tiago Cruz e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Julho 2023

The work presented in this thesis was carried out within the Laboratory of Comunication and Telematics (CT) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC) in the context of the MH-SDVanet project: Multihomed Software Defined Vehicular Networks (reference PTDC/EEI-COM 5284/2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

Acknowledgements

I would like to thank the patience of the advisor teachers, Prof. Paulo Simões, Prof. Tiago Cruz, and Prof. Bruno Sousa in this long technology ramp-up.

Over the course of this work, I've engaged with other researchers from other universities who have provided access to lab environments: Jose Gomez from the University of South Carolina, and Conor Black from the Queen's University in Belfast. I am truly grateful for this as it enabled a faster ramp-up.

Also thanks to my MSC colleagues Domitília Noro and Duarte Dias for the teamwork in our P4 endeavor.

Last, to my family, whom this thesis has kept me away from.

Abstract

Software-Defined Networking (SDN) separate the data plane from the control plane, where the control is handled by the SDN controller. So, the controllers have full visibility of the network.

P4 brings a new set of possibilities, as the way the packets are processed is not defined by the vendor but rather by the P4 program. This allows moving the applications to the data plane which can now work at line speed, with great advantage for network security functions. Functions such as access control, privacy, encryption, and integrated defense can be mostly assured by the programmable data plane, thus offloading the controller. P4-INT is a P4-powered data-plane function that can monitor the network and provide telemetry per packet.

A consequence of having the data plane taking care of the security features is the difficulty to confirm if the data plane behavior corresponds to the security policies.

This work investigates how P4-INT can act as security control, how it can be attacked, and how it can be protected.

Within this thesis it was developed a Mininet and an equivalent P4Pi network with INT-MD and BMv2 switches. It was also configured an INT collector with InfluxDB and a Grafana dashboard. This solution was tested as a possible security control in which some scenarios were executed. Once the value of INT for defense purposes was assessed, it was also tested how it could also be attacked and defended. Moreover, some future improvements were proposed, in response to the vulnerabilities found in this INT network.

// Keywords

SDN, security, data plane, P4, INT.

// Resumo

As redes definidas por sofware (SDN) separam o *data plane* do *control plane*, onde o controlo é exercido pelo controlador SDN. Assim, estes controladores têm uma visão completa da rede.

A linguagem P4 traz uma série de novas possibilidades sobre a forma como os pacotes são processados no *data plane* que já não são definidas pelo fabricante, mas sim pelo programa P4. Isto permite que as aplicações funcionem no *data plane* e como tal podem funcionar à velocidade da linha, com grande vantagem para as funcionalidades de segurança. Funções como controlo de acesso, privacidade, encriptação ou defesa integrada, podem ser asseguradas pelo *data plane* programável, descongestionando assim o *control plane*.

Uma consequência de o *data plane* tomar a cargo as funcionalidades de segurança é que se torna mais difícil ao controlador verificar se o comportamento do *data plane* corresponde às políticas de segurança.

Este trabalho investiga como uma função em P4 a correr no *data plane*, o P4-INT, pode ser usada como uma controlo de segurança numa rede P4, como pode ser atacado e como pode ser protegido.

Nesta tese foi desenvolvida uma rede em Mininet e outra equivalente com P4Pi. Estas redes operam com INT-MD e switches BMv2. Nestas redes foi também configurado um colector de INT com InfluxDB e um painel Grafana. Esta solução foi testada como um possível controlo de segurança na qual foram executados alguns cenários de ataque. Dado o valor de uma solução P4 com INT para a defesa, foi também testado como poderia ser atacada e defendida. Devido às vulnerabilidades encontradas nesta rede INT são propostas algumas melhorias.

// Palavras-Chave

SDN, segurança, P4, *data plane*, INT.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 3
1.3 Structure . . . 3

2 Background and Related Work **5**
2.1 SDN architecture . . . 5
2.1.1 Centralized SDN controller . . . 6
2.1.2 High Availability Controller architectures . . . 8
2.2 OpenFlow . . . 9
2.3 Control Plane security with OpenFlow . . . 10
2.3.1 Secure control channel . . . 11
2.3.2 DoS - flooding the controller . . . 11
2.4 P4 . . . 12
2.4.1 P4Runtime . . . 14
2.4.2 BMv2 . . . 15
2.4.3 P4 Inband Network Telemetry . . . 16
2.4.4 P4 language . . . 20
2.5 Control Plane security with P4 . . . 22
2.5.1 Controller fault tolerance . . . 23
2.5.2 Data plane consistency . . . 24
2.5.3 Port knocking security . . . 24
2.5.4 Runtime Verification . . . 26
2.5.5 Adversarial Data Plane Verification . . . 28
2.5.6 P4RT security . . . 29
2.6 Summary . . . 30

3 Research objectives and approach **31**
3.1 Research . . . 32
3.1.1 P4 tutorial . . . 32
3.1.2 P4 infrastructure by the University of South Carolina . . . 33
3.1.3 NG-SDN . . . 34
3.1.4 P4Pi . . . 36
3.2 Objectives . . . 36
3.3 Approach and use cases . . . 37
3.3.1 Compromised switch, tables manipulated . . . 37
3.3.2 Compromised switch, INT manipulated . . . 38
3.3.3 MITM attack against INT . . . 38

3.3.4 Mitigation of MITM attack against INT 39

4 Preliminary work 41
4.1 Simulate an INT platform 44
4.1.1 Packet source 44
4.1.2 Packet forwarding 45
4.1.3 INT source 45
4.1.4 INT transit 46
4.1.5 INT sink 47
4.1.6 Server listening 48
4.2 Collection of INT statistics and visualization 48
4.2.1 Collection 49
4.2.2 Visualization 50
4.3 Attacking the INT platform 50
4.3.1 INT eavesdropping 52
4.3.2 INT replay 55
4.3.3 INT manipulation 55

5 Implementation scenario 57
5.1 Topology 57
5.2 Network deployment 58
5.3 Packet source 58
5.4 Packet forwarding 60
5.5 INT source 61
5.6 INT transit 62
5.7 INT sink 63
5.8 Wireshark INT P4 dissector 64
5.9 Collection of INT statistics and visualization 64
5.9.1 Install and configure InfluxDB 65
5.9.2 Collection to InfluxDB 65
5.9.3 Install and configure Grafana 67
5.9.4 Visualization in Grafana 68

6 Results 69
6.1 Attack detection 69
6.1.1 Unauthorized flows 70
6.1.2 Attack visualization 70
6.2 Attacking INT 71
6.2.1 INT eavesdropping 71
6.2.2 INT replay 77
6.2.3 INT manipulation 77
6.3 Defending INT 79
6.4 Costs of INT 82
6.5 Summary 84

7 Conclusions 87
7.1 Initial research 87
7.2 Preliminary test cases 87
7.3 Final test cases 87

7.3.1 INT as security control 88
7.3.2 INT as a target 88
7.3.3 Defending INT 88
7.4 Future work 89

Appendix A Installation and configuration in P4Pi **103**
A.1 Pre-requisites 103
A.2 Service mode 104
A.3 Manual mode 105
A.4 Debugging mode 105
A.5 Default routes 106

List of Figures

2.1 The three-layer architecture of SDN [9]. 6
2.2 Roles of Multiple Controllers in the SDN 7
2.3 Fault-tolerant control plane design with HAC controller [25]. 9
2.4 OpenFlow 1.5.1 Matching and Instruction execution in a flow table [13]. 10
2.5 OpenFlow 1.5.1 flow entry [13]. 10
2.6 DoS attack on SDN [37]. 12
2.7 Evolution of network programmability[38]. 13
2.8 PISA architecture [39]. 13
2.9 P4 runtime, adapted from [44], ... 15
2.10 P4 INT-MD mode operation example [47]. 18
2.11 P4 INT operation modes [5]. 18
2.12 How P4 interacts with the targets [65]. 20
2.13 Example of a P4 switch configured as a firewall, managing most secure features [29]. 23
2.14 P4Consist architecture [71]. 24
2.15 Port knocking example [75]. 25
2.16 Port knocking options, adapted from [74]. 26
2.17 Port knocking attack [76]. 27
2.18 P4 KeySight overview [83]. 28
2.19 P4 compromised switch sending fake INT-XD data. 29

3.1 Gantt chart view of the activities performed in the first semester. . . 31
3.2 Gantt chart view of the activities completed in the second semester. 32
3.3 P4 tutorial scenario for the link monitor exercise. 33
3.4 Monitoring with a P4 probe, before and after simulating traffic with *iperf*. 34
3.5 Excerpt of ONOS GUI in NG-SDN VM. 35
3.6 Overview of an attack in which a compromised switch reports false P4 Port Knocking Ids to the controller [93]. 35
3.7 P4Pi overview. 36
3.8 Foe host gets access to the server and the controller is unaware. . . 37
3.9 Foe host gets access to the server and the controller is unaware as well as the INT collector. 38
3.10 Foe host listens and hides its activities with a MITM attack. 39

4.1 Standard Spine-Leaf architecture [96]. 41
4.2 INT early test environment and statistics collection with a crafted probe. 42

4.3 INT preliminary environment. 43
4.4 Example of processing an INT-MD packet. 44
4.5 Wireshark capture of a frame leaving h1. 45
4.6 *Wireshark* capture of a packet leaving s1 towards s2. 46
4.7 *Wireshark* capture of a packet leaving s2 towards s3. 47
4.8 *Wireshark* capture of a packet leaving s3 towards h4. 48
4.9 Wireshark capture of packet leaving s3 towards h2. 48
4.10 INT packed decoded by the collector script. 49
4.11 InfluxDB client, displaying INT measurements. 50
4.12 Grafana view of INT statistics. 51
4.13 INT statistics under high load. 51
4.14 The server network with the collector and the rogue host eavesdropping on the INT reports. 52
4.15 Failed ARP poisoning attempt. 53
4.16 INT detecting high load in s3, due to *ettercap* attack. 53
4.17 Spine-Leaf topology with s6 as a P4 L2-switch within a SOC. 54
4.18 h5 eavesdropping INT sent to h4. 54
4.19 INT replay attack, h1 port 80. 55

5.1 Diagram of the INT solution with three P4Pis. 58
5.2 INT lab with three P4Pis. 59
5.3 *Wireshark* view with a dissector of the INT-MD packet sent from the INT source to the INT transit. 65
5.4 *Wireshark* view with a dissector of the INT report packet sent from the INT sink to the collector. 66
5.5 Switch latency in P4PI. 68

6.1 Possible attack flows to the server and the flow of INT statistics to the collector. 70
6.2 Grafana dashboard evidencing unauthorized flows. 71
6.3 h1 accessing ports not authorized through a *nmap* scan. 72
6.4 Diagram of the INT solution with three P4Pi devices. 72
6.5 Diagram of the INT solution with three P4Pis and a router to a SOC. 74
6.6 INT lab with three P4Pis and one router. 75
6.7 ARP poisoning attack, view of an ARP reply with wireshark. 76
6.8 Scenario of INT replay attack. 77
6.9 INT replay attack, pre-captured data sent to the collector. 78
6.10 Scenario of an attack hidden from INT with an *ettercap* filter. 78
6.11 Unauthorized access to a SSH service is misreported as an authorized access to HTTP, using an *ettercap* filter. 79
6.12 CPU and memory usage of the three P4Pi at 20pps. 83
6.13 CPU and memory usage of the three P4Pi under excessive load with *iperf* at 770pps. 83
6.14 High load effect on the queues of the P4 switches and loss of reports. 84

7.1 Possible improvements of an INT platform: authentication of the switch, encrypted tunnel, encrypted data, and closed loop control. 89

List of Tables

6.1 Payload size along a INT-MD path. 82

Acronyms

ADPV Adversarial Data Plane Verification. 28, 29
AES Advanced Encryption Standard. 22
ALU Arithmetic Logic Units. 13
BFT Byzantine Fault Tolerance. 23
BMv2 Behavioral Model version 2. 15, 16, 19, 20, 33, 36, 42, 58, 60, 82, 83, 87
DAI Dynamic ARP Inspection. 73, 80
DDoS Distributed Denial-of-Service. 11, 22
DoS Denial-of-Service. 11, 12, 30, 71, 80
DPDK Data Plane Development Kit. 36
DTLS Datagram Transport Layer Security. 81, 82, 89
GCP Google Cloud Platform. 29
GRE Generic Routing Encapsulation. 19
HAC High Availability Controller. 8
IDS Intrusion Detection System. 12, 81, 85
INT Inband Network Telemetry. 2, 16
IPS Intrusion Prevention System. 81, 85
IPSec Internet Protocol Security. 81, 82, 89
LPM Longest Prefix Match. 45–47, 60, 62
MAT Match-Action Tables. 13, 14, 27, 45, 60
MITM man-in-the-middle. 29, 37–39, 43, 52, 55, 57, 73
mTLS Mutual TLS. 29
NIB Network Information Base. 6, 7
ONF Open Networking Foundation. 1, 2, 10
ONOS Open Network Operating System. 7
P4RT P4Runtime. 14, 23, 29
PBT Postcard-based Telemetry. 17
PEC Packet Equivalence Class. 28
PISA Protocol Independent Switching Architecture. 12
SDN Software-Defined Networking. xi, 1
SOC Security Operations Center. 53, 74
VM Virtual Machine. 32
VXLAN Virtual Extensible LAN. 19
WIPS Wireless Intrusion Prevention System. 81, 89
XFSM eXtended Finite State Machines. 11

Chapter 1

Introduction

1.1 Motivation

The Open Networking Foundation (ONF) defines Software-Defined Networking (SDN) as "the physical separation of the network control plane from the forwarding plane where a control plane controls several devices" [1].

The SDN controller serves as the control plane, directing traffic based on forwarding policies established by the network operator. This reduces the reliance on manual configurations of network devices. By separating the control plane from the network hardware and implementing it as software, the SDN controller enhances automated network management and enables seamless integration with business applications.

SDN fundamentally transforms network design and deployment by granting applications greater control over network infrastructure configuration. This paradigm shift enables highly automated and adaptive infrastructure that fulfils the specific requirements of applications.

SDN offers several advantages [2]:

1. **Increased control, speed and flexibility**: Developers can exercise control over traffic flows on the network by programming an open standard software-based controller.
2. **Customizable network infrastructure**: Administrators can configure network services and allocate virtual resources, allowing real-time changes to the network infrastructure through a centralized location.
3. **Robust security**: SDN provides comprehensive visibility into the entire network, offering a broad view of security threats. Operators can establish separate security zones for devices with different security requirements and promptly quarantine compromised devices to prevent the spread of infections..

These advantages translate into cost reduction, as efficient network management minimizes the need for additional devices and facilitates the use of general-purpose devices.

The SDN controller plays a crucial role in managing and configuring network devices by applying flow rules to control their behavior. In SDN, a flow represents a specific network traffic stream that is identified and governed by the SDN controller. It comprises a group of packets that share common attributes and are treated similarly by the network devices. SDN flows are defined based on various packet attributes, including source and destination IP addresses, port numbers, protocol type, and other packet header fields.

Given the separation of planes in SDN, the control plane must maintain communication with the forwarding or data plane. This communication is facilitated through a standard set of messages and rules, such as OpenFlow. OpenFlow was the *de facto* standard for the southbound interface between the SDN controller and switches during the Pre-P4 era. With OpenFlow the SDN controller takes information from applications and converts it into flow entries, which are then transmitted to the switches via OpenFlow. However, OpenFlow has a limitation in its fixed set of supported header fields. Any new version of the protocol must be approved by the ONF and implemented by hardware manufacturers.

SDN with P4 introduces a new set of possibilities, as it allows packet processing to be defined by the P4 program rather than the vendor. Using P4 language, developers can define the behavior of the data plane, specifying how switches process packets. P4 enables developers to define which headers a switch should parse, how tables match on each header, and which actions the switch should perform on each header. This programmability extends the capabilities of the data plane, including security features such as stateful packet inspection and filtering, thus offloading tasks from the control plane. Moreover, P4 programs can run on programmed devices at line speed, enhancing the performance of security functions.

Offloading security functions to the P4 data plane can help mitigate security risks in the P4 control plane. However, there is still a need to monitor the network behavior. As defined in the NIST Cyber Security Framework [3], the 'Detect' function must be assured, and P4 can assist in this aspect as well.

P4 Inband Network Telemetry (INT) is a framework that enables the data plane to collect and report network state without requiring the control plane's involvement [4]. INT can provide telemetry that aids in evaluating the network behavior per flow, including information such as the path taken by the flow, the reasons for choosing that path, and the duration of each hop.

The INT architecture is designed to be versatile and supports various high-level applications that offer valuable capabilities [5]. These include:

1. Network troubleshooting and performance monitoring: INT facilitates functionalities like traceroute, micro-burst detection, and packet history. These features assist in diagnosing network issues and monitoring performance metrics.
2. Advanced congestion control: The INT architecture enables advanced congestion control mechanisms, including network congestion management with feedback control loops. This allows for efficient handling of congestion scenarios within the network.

3. Advanced routing: INT also empowers advanced routing techniques such as utilization-aware routing with feedback loops. This enables routing decisions to consider the current network utilization, leading to optimized routing paths.

1.2 Objectives

Our objectives include two main aspects. Firstly, we aim to showcase the capabilities of a INT platform utilizing P4 devices. Additionally, we seek to assess the suitability of this platform to support security control mechanisms. However, it is essential to acknowledge that the INT framework itself may be susceptible to potential attacks. Therefore, we will also dedicate our efforts to evaluating potential vulnerabilities and exploring strategies for protecting INT from such threats.

The concepts to be designed and implemented are to be evaluated in high-fidelity environments based on Mininet [6] and then on hardware, in Raspberry [7] devices running P4Pi [8].

1.3 Structure

The remaining of this document is organized as follows:

- Chapter 2 - Provides an overview of the key topics related to this work, introducing concepts such as SDN, OpenFlow, P4, security features, and related work regarding these topics.
- Chapter 3 - Describes the research objectives and introduces the approach taken throughout the dissertation work.
- Chapter 4 - Describes the activities and preliminary results related to the first iterations.
- Chapter 5 - Describes the implementation of the proposed framework, that was tested in the Chapter 4.
- Chapter 6 - Presents the results obtained from the experimental work on the framework.
- Chapter 7 - Provides the main conclusions, a summary of the results, and the future work that can be followed.

Chapter 2

Background and Related Work

This chapter presents a comprehensive overview of the security features of SDN, covering both the classic OpenFlow as well as the P4 perspectives, with an overview of the relevant concepts deemed essential for grasping the topics discussed in this thesis.

The chapter is structured in a top-down manner, starting with introducing the fundamental SDN concepts. Subsequently, it explores OpenFlow and related aspects concerning the Control Plane. The discussion then proceeds to introduce P4, highlighting its features, limitations, and the available security controls to address potential vulnerabilities.

2.1 SDN architecture

A typical representation of the SDN architecture includes three layers: the application layer, the control layer, and the infrastructure layer, as illustrated in Figure 2.1:

- The **application layer** contains the typical network applications or functions, like intrusion detection systems, load balancing, and firewalls. A traditional network uses specialized devices, such as firewalls or load balancers, but a SDN network can replace these device with applications that use the controller to manage the data plane behavior. The applications communicate with the control layer using a northbound interface.
- The **control layer** is provided by the centralized SDN controller software. This controller resides on a server and manages policies and the flow of traffic throughout the network. The control layer communicates with the data plane using its southbound interface.
- The **infrastructure layer** is comprised by the SDN switches. The switch queries the controller for guidance as needed and provides the controller with information about the traffic it handles. All similar packets sent to the same host are treated in a similar manner and forwarded along the same path by the switch. This is the concept of the flow in SDN. With this model, once a centralized controller derives the desired forwarding behavior, forwarding instruc-

tions for packets are downloaded to the appropriate switches. The communication between the controller and the network devices can use some form of a standardized protocol to facilitate device programming. Thus the switching devices that may play the role of routers, switches, load balancers, firewalls, or virtual switches, don't require local intelligence.

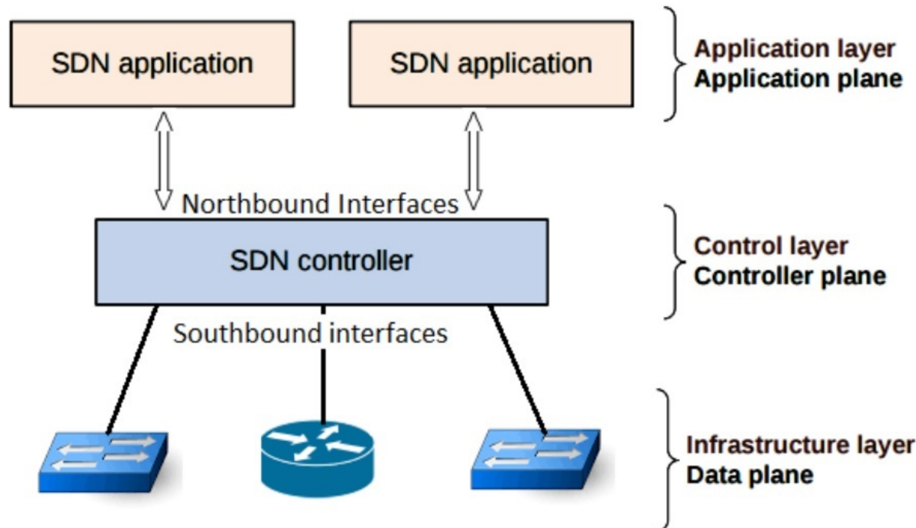

Figure 2.1: The three-layer architecture of SDN [9].

2.1.1 Centralized SDN controller

The main advantage of SDN is the ability to manage a network from a centralized system. This allows for centralized provisioning, enhanced scalability, centralized monitoring, and better resource management [10], but brings with it the increased risk of a single point of failure. A single controller system may not keep up with the growth of the network. It is likely to become overwhelmed via controller bottlenecks while dealing with an increasing concurrent number of requests from the switches and struggling to achieve the performance requirements [11].

In order to raise capacity and redundancy, the Openflow protocol - currently the most widely accepted industry standard for the southbound interface [12], between the SDN controller and the switch - supports multiple controllers. As such, the switch can continue to operate in OpenFlow mode if one controller or controller connection fails. Each controller can have one of the three following roles for a switch: master, equal, or slave [13]. Master and Equal controllers can both receive asynchronous messages (e.g., Packet-In) and modify switch states. Each switch can have a maximum of one master switch, but many equal or slave controllers. By default, slave controllers do not receive asynchronous messages and can only read switches' state [14].

The master-slave topology is often preferred [15][16], as it has proved to be more reliable in case of failure. There are however important considerations to address due to the balance between ensuring consistency in the Network Information Base (NIB), and performance when using a distributed data store [17].

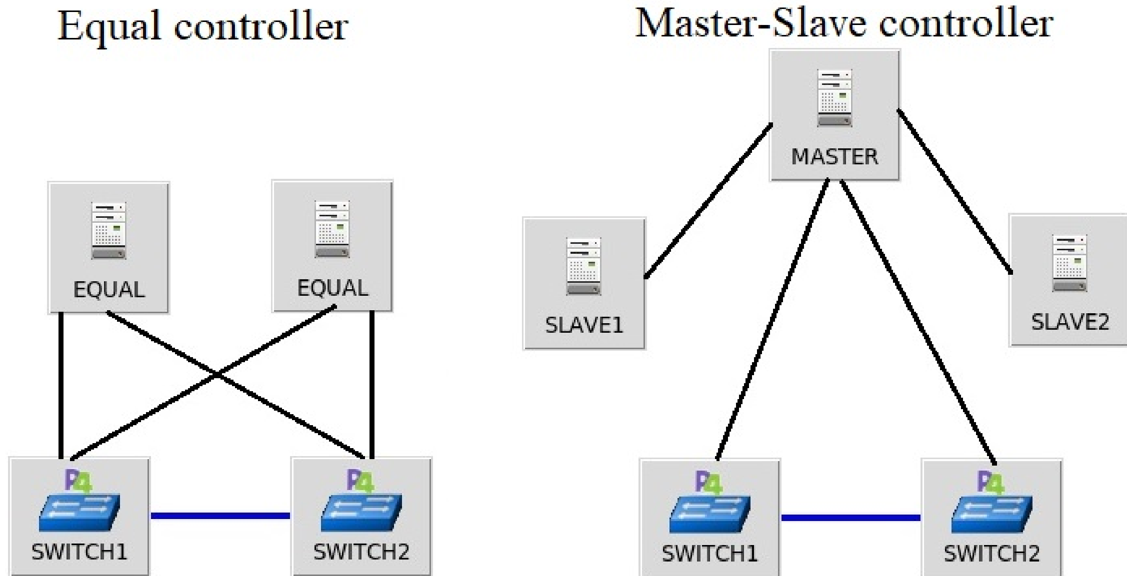

Figure 2.2: Roles of Multiple Controllers in the SDN

Some solutions for this issue are:

- FlowVisor [18], which assigns different controllers to different slices of the network;
- Onix [19], that partitions the NIB giving each controller instance responsibility for a subset of the NIB before sharing it between other Onix instances within the cluster.
- Hyperflow [11], a distributed event-based control plane for OpenFlow, that yet remains logically centralized [20]. When a controller failure is discovered, HyperFlow reconfigures the affected switches and redirects them to another nearby controller instance.
- Open Network Operating System (ONOS) [21], a flat-distributed controller platform composing a cluster, in which each instance is responsible for a subset of the network devices [11]. To achieve high availability ONOS control platform uses redundant secure channels between switch and controller instances, global network state distribution between controller instances, controller role distribution for each switch, and controller role changing procedure in case of primary controller failure. However, ONOS does not support load-balancing procedures between controller instances [22].
- Kandoo [23], that proposes a hierarchical distribution of controllers based on two layers: the bottom layer, a group of controllers with no interconnection and no knowledge of the network-wide state and the top layer, a logically centralized controller that maintains the network-wide state. Despite the scalability advantage, Kandoo does not support fault-tolerance and resiliency [11].
- Google's B4, also a two-level hierarchical control framework, that deploys robust reliability and fault-tolerance mechanisms at both levels of the control hierarchy in order to enhance the B4 system availability. [11].
- As a second iteration since 2017, Orion [24] solved B4's availability and scale problems via a distributed architecture in which B4's control logic is decoupled into micro-services with separate processes.

The flat-distribution approaches, despite their ability to distribute the control plane, impose a strong requirement: a consistent network-wide view in all the controllers. Thus they generate a large amount of control traffic among controllers, which is reduced in the hierarchical architectures [14].

Controllers may fail as result of two main causes:

- **Software or hardware failures** can be caused by bugs, attacks, or maintenance errors.
- **Network failures** leading to a loss of connectivity between a controller and a switch.

Controllers can discover the failure of their neighbors in a number of ways:

- **Heartbeat messages**, Each controller regularly sends heartbeat messages to neighbor controllers, if some consecutive messages are missed, it can consider that the neighbor controller has failed.
- **Failure message**. Controllers could fail in a graceful way by sending a failure message to their neighbors before totally shutting down. This can happen for example when a controller is being shut down manually for maintenance reasons.

2.1.2 High Availability Controller architectures

In order to avoid single point of failures in the control platform, High Availability Controller (HAC) architectures have been proposed with the following design of the control plane [25] as illustrated in Figure 2.3:

- Controller network services and application redundancy and synchronization with warm active/standby strategy.
- The primary controller periodically or conditionally pushes up snapshots of services and applications to all standby controllers.
- For controller data synchronization, the data is shared in a storage.

This setup was tested and exhibited an average controller failover time for a two-node HAC cluster of 40 to 50ms, which is under the maximum statistically acceptable delay for services [25].

Another approach described in the literature relies on the cluster paradigm, since it has a number of advantages over the master-slave paradigm [26]:

- No overhead of manual configuration for all controllers on each device.
- Increased High Availability as there are multiple controllers available if primary fails.
- A single IP address is enough to deploy multiple controllers in a cluster.

The proposed system consists of a High Availability architecture composed of a cluster of controllers with a single virtual IP, which will also be responsible for equal distribution of the traffic load [26].

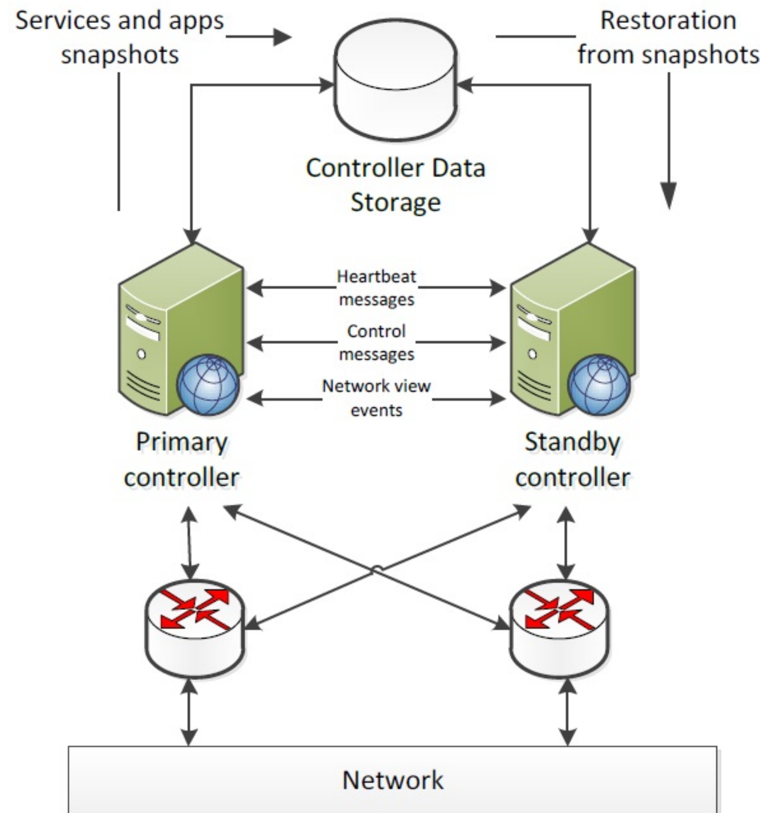

Figure 2.3: Fault-tolerant control plane design with HAC controller [25].

2.2 OpenFlow

An OpenFlow switch has one or more tables of packet-handling rules, named flow tables. Each rule matches a subset of the traffic and performs certain actions such as dropping, forwarding, or modifying the received packet, as illustrated in Figure 2.4.

Using the OpenFlow protocol, the controller is able to add, update, and delete flow entries in flow tables, both reactively (in response to packets) and proactively (predefined). The switch communicates with the controller and the controller manages the switch via the OpenFlow protocol [13]. Each flow table in the switch contains a set of flow entries as illustrated in Figure 2.5. Each flow entry consists of match fields, counters, and a set of instructions to apply to matching packets [13]:

- Match fields: for packet matching, which consists of ingress ports and optional matched fields or metadata specified from a previous table.
- Priority: is defined the precedence of the flow entry.
- Counters: updated when packets match.
- Instructions: to modify action sets or pipeline processing.
- Timeouts: maximum amount of time or idle time before flow is expired by the switch.
- Cookie: opaque data value chosen by the controller. May be used by the controller to filter flow entries affected by flow statistics, flow modification and

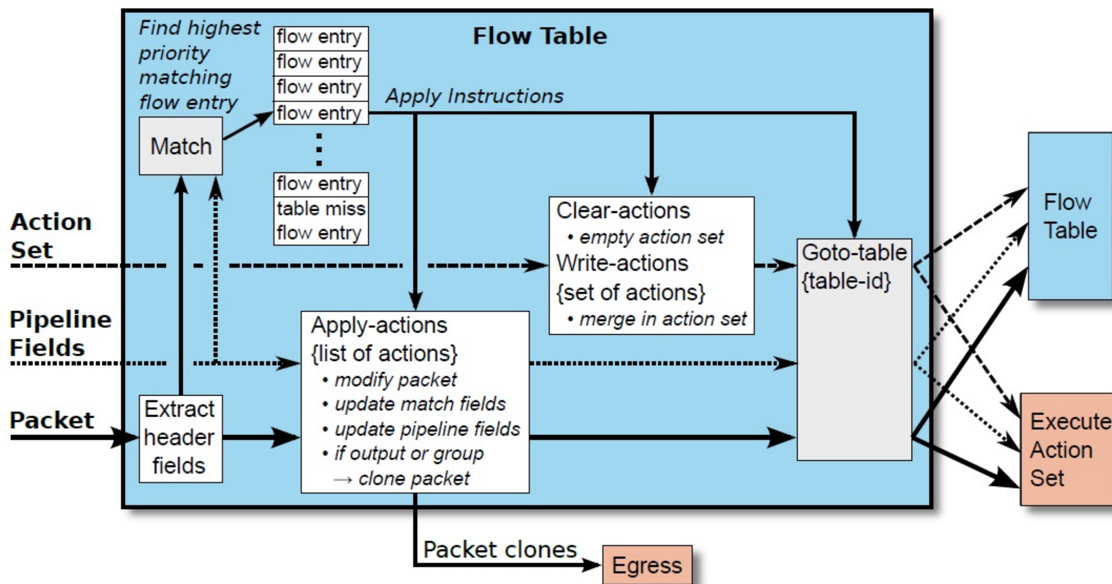

Figure 2.4: OpenFlow 1.5.1 Matching and Instruction execution in a flow table [13].

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Figure 2.5: OpenFlow 1.5.1 flow entry [13].

flow deletion requests. Not used when processing packets.
- flags: flags change the way flow entries are managed.

The main limitation of OpenFlow is the fixed set of header fields supported. Each new version of the protocol must first be approved by the Open Networking Foundation (ONF) and then be implemented by hardware manufacturers. Although more flexibility is available to the network operator in contrast to traditional networking, it is still limited by a fixed set of features of the OpenFlow protocol [27]. So, OpenFlow doesn't really control the switch behavior, rather it gives us a way to populate a set of well-known tables [28].

2.3 Control Plane security with OpenFlow

The different points of attack to OpenFlow-based platforms might be at either the data plane, the control plane, or the application plane.

As the brain of SDN, the control plane is responsible for mitigating the attacks, either by itself or with a third-party application, which operates at a lower speed than the data plane [29]. The offloading to the controller highlights the scalability and performance issues in stateful packet processing in SDN switches with OpenFlow. The data plane relies on the rules installed by the controller to forward or drop network traffic, and it introduces communication overhead of the control

plane and data plane. Due to this limitation, approaches such as *OpenState* [30] have been proposed. *OpenState* adds stateful data plane programming to OpenFlow, using eXtended Finite State Machines (XFSM) to provide some intelligence to the switch.

The SDN control plane is vulnerable to Distributed Denial-of-Service (DDoS) attacks, DNS cache poisoning, packet manipulation, route poisoning, API exploitation, and other types of attacks [31].

There has been considerable effort in securing the SDN control plane since it is central to ensuring the whole network's security including:

- development of frameworks that detect vulnerabilities such as *CONGUARD*, that can detect harmful race conditions [32];
- control plane isolation and recovery in case of security-compromising events [33]. The proposed mechanisms are the separation and isolation in the control plane (originally SDN only performs isolation at the data plane), and recovery mechanism that allows network components to be rolled back to a good-known state at regular intervals;
- anomaly-based intrusion detection and prevention by training the network controller with machine learning algorithms [34].

2.3.1 Secure control channel

The latest OpenFlow standard, v1.5.1 [13], proposes TLS1.2. The switch and controller mutually exchange certificates signed by a site-specific private key, multiple CAs and CRLs when dealing with multiple controllers [13]. However, OpenFlow still does not enforce TLS and offers plain TCP as an option, thus some switches and controllers don't support TLS, and some that do support haven't implemented authentication [35]. Thus, some attacks are possible spanning from simple eavesdropping to taking full control.

Some controllers, like OpendayLight and Floodlight, have some other vulnerabilities related to weak authentication mechanisms which have been already exploited [35]. In fact, there has been some development around enhancing the controllers' security. One example is the *SE-Floodlight* [36] that extends *Floodlight* with a security-enforcement kernel layer whose purpose is to mediate all data exchange operations between the application layer and the data plane. This kernel layer functions are also applicable to other OpenFlow controllers: authentication service, role-based authorization, inline flow-rule conflict resolution, and a security audit service.

2.3.2 DoS - flooding the controller

In the scope of Denial-of-Service (DoS) attacks, the controller is at risk of being flooded and not being able to manage the network. The main mitigation strategy is to distribute the control plane to be able to handle the load [22].

The DoS attacks were successfully tested with a tool named *sdn-toolkit*. This toolkit includes an application that can impersonate an OpenFlow switch, establishing relationships with the controller, exchanging *Hellos*, responding to *Feature Requests*, *Configurations Sets*, and *Configuration Gets*. It also includes other applications that can send packets with different header combinations towards the controller packets, thus reaching millions possible different flows [35].

This technique takes advantage of the fact that, for each new flow, the switch may be configured to query the controller, thus a DoS attack often includes sending an high-rate of new flows at the switches.

The security control mechanisms frequently act in two steps [37], as illustrated in Figure 2.6:

- **detection**, with classification via techniques such as entropy-base detection (information statistics), traffic pattern analysis, monitoring connection rate, and Intrusion Detection System (IDS) with signature-based, machine learning or deep learning.
- **mitigation**, usually using the detection mechanisms as input, involve packet dropping, port blocking, bandwidth throttling, or redirection. Other solutions involve using moving targets (e.g. frequently changing IP addresses).

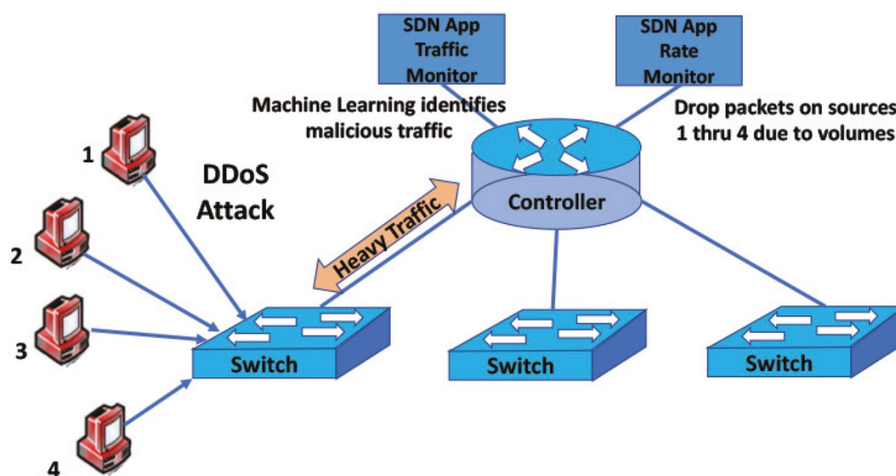

Figure 2.6: DoS attack on SDN [37].

2.4 P4

P4 (Programmable, Protocol-independent Packet Processor) defines the way the switches process packets, making the switches protocol-independent (not tied to a given protocol) and target-independent (independent of the hardware). So, this is a step forward in the evolution of the network programmability, as illustrated in Figure 2.7.

P4 is based on the Protocol Independent Switching Architecture (PISA) which incorporates the following key components:

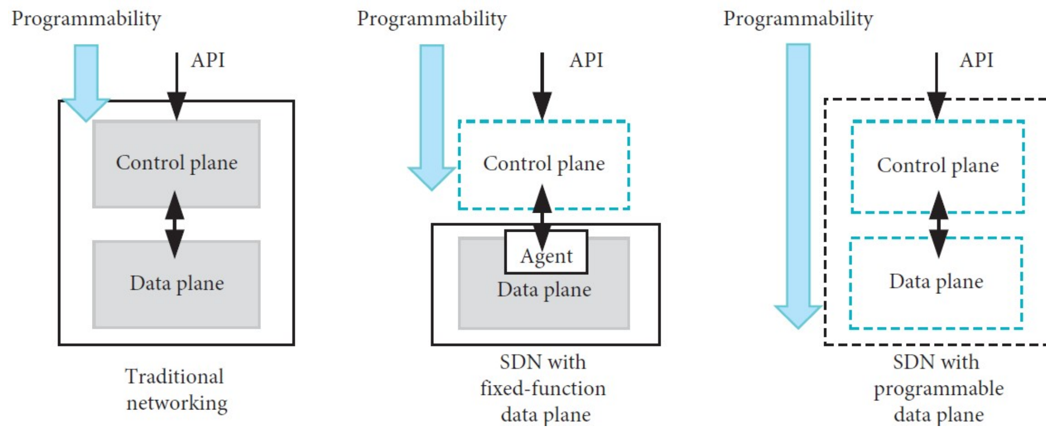

Figure 2.7: Evolution of network programmability[38].

- programmable parser, that determines which packet headers will be recognized by the data plane program;
- "Match-Action" stages, that match data against Match-Action Tables (MAT) that contains entries and execute a corresponding action. Each match-action stage has multiple memory blocks and Arithmetic Logic Units (ALU);
- programmable deparser, that re-assembles the packets back to transmission, into the queueing, replication engine, etc.

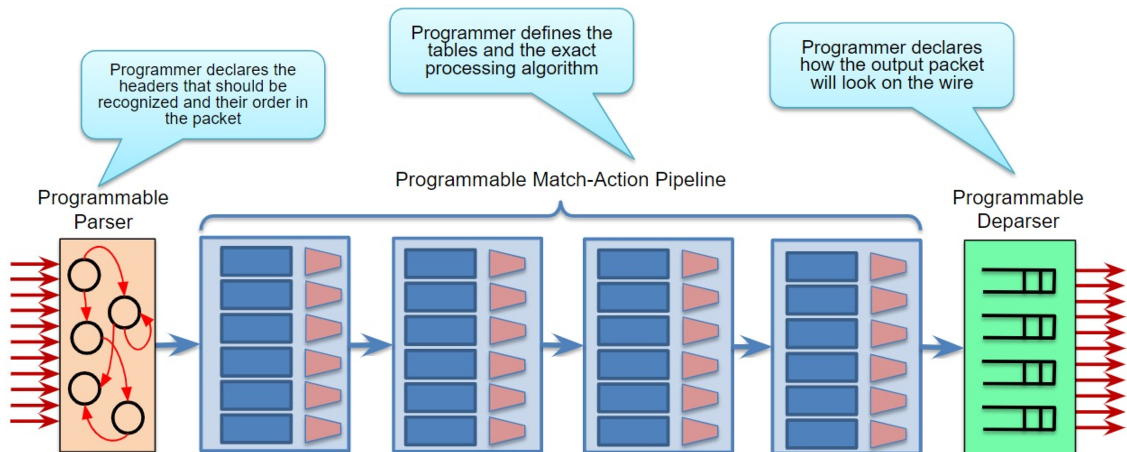

Figure 2.8: PISA architecture [39].

Besides tables, P4 contains an additional feature, named registers, for stateful processing. Unlike metadata, registers are persistent beyond the single iteration of packet processing. Each register is defined as a register type, an array that consists of multiple register entries.

Although P4 and OpenFlow are both focused on opening up the forwarding plane, P4 addresses a different requirement. P4 allows to define which headers a switch recognizes (or "parses"), how to match each header, and which actions the switch shall perform on each header. P4, therefore, lets us control switches "top-down" by first specifying their forwarding behavior, then populating the tables we have defined. Additionally, P4 compilers typically auto-generate the API needed to populate the tables [28].

This way, the data-plane functionality is no more defined by the switch vendor but rather by the P4 program, which allows to:

- define and parse new protocols,
- customize packet processing functions,
- measure events occurring in the data plane with high precision,
- offload applications to the data plane.

These programmable switches have been found to be 21% faster and 53% less power-hungry than comparable legacy switches [40].

P4 has became a project hosted by ONF with its own site [41], where we can already find an ecosystem including compilers, compatible hardware, Network Operating Systems, etc.

2.4.1 P4Runtime

The P4 data plane configuration is named data plane runtime [42]. The P4Runtime (P4RT) API is a control plane specification to manage the data plane elements of a device defined by a P4 program [4] which shields the hardware details of the data plane and is independent of the features and protocols the data plane supports [38].

P4 Runtime API has support for two main functionalities:

- Manages MATs: manages the data plane by adding, deleting, modifying, and displaying entries in MATs.
- Updates forwarding plane logic: updates the forwarding behavior of P4 programmable switch using new P4 code.

This way, devices based on different targets can be controlled by the same API.

This API is based in Google's Protobuf [43] to realize a language and platform-neutral mechanism for serializing structured data. The endpoints of a P4RT connection are in the controller and the switches as illustrated in Figure 2.9. The gRPC server on P4 targets interacts with the P4-programmable components via platform drivers. P4 compilers with support for P4Runtime generate a P4Runtime configuration. It consists of the target-specific configuration binaries and *P4Info* metadata.

P4info

The structure of the API calls to access P4 entities is described in the *p4runtime.proto*. It is part of the P4Runtime but developers can extend it to use custom data structures, e.g., to implement interaction with target-specific externs.

P4Info describes all P4 entities (MATs and externs) that can be accessed by controllers via P4Runtime. Then, the controllers establish a gRPC connection to the gRPC server on the P4 target. The target-specific configuration is loaded onto the P4 target and P4 entities can be accessed.

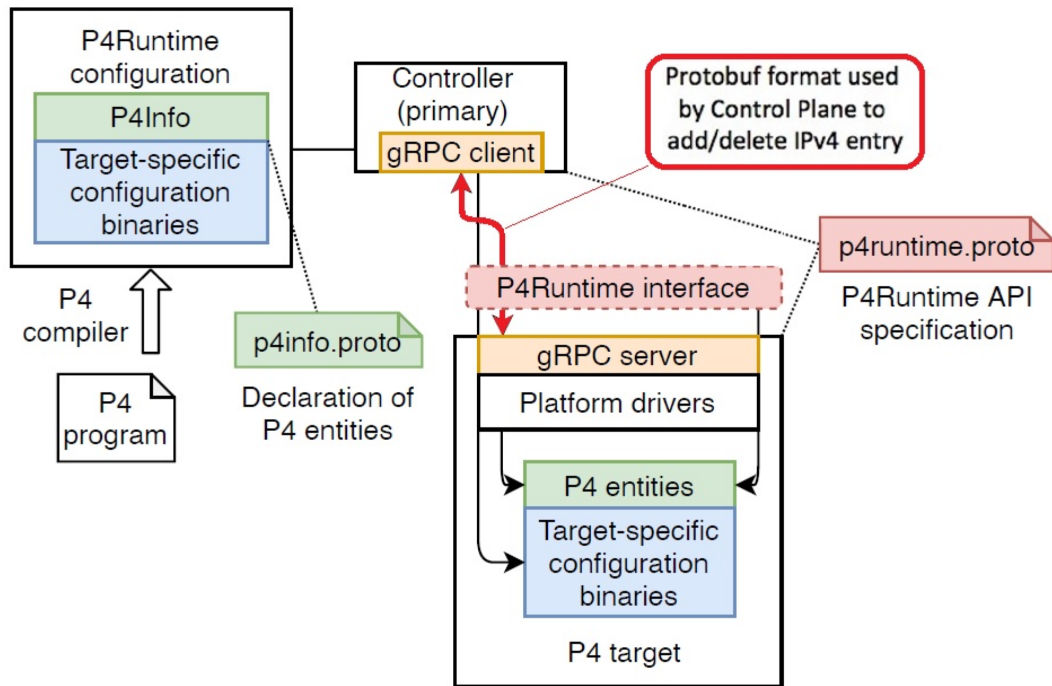

Figure 2.9: P4 runtime, adapted from [44],

Multiple controllers

P4Runtime provides support for multiple controllers, such as OpenFlow, allowing for enhanced flexibility and interoperability within software-defined networks. With this capability, network operators have the option to choose and integrate different controllers into their SDN infrastructure, leveraging the strengths and functionalities offered by each.

For every P4 entity, read access is provided to all controllers whereas write access is only provided to one controller. To manage this access, P4 entities can be arranged in groups where each group is assigned to one primary controller with write access while secondary controllers have only read access.

2.4.2 BMv2

The most popular P4 software target is the behavioral model Behavioral Model version 2 (BMv2) switch, which is written in C++. BMv2 is used to implement complex P4 programs and test new implementations, usually within a Mininet [6] virtual network. Even though BMv2 can run any P4 program, it lacks in performance as the latency and throughput has been compared with others like *Open vSwitch*.

simple_switch

simple_switch is the BMv2 target supporting all features from the $P4_{14}$ specification and the v1model architecture of $P4_{16}$.

simple_switch_CLI

BMv2 includes a command line interface (CLI) program to manipulate MATs and configure the multicast engine of the BMv2 P4 software target via this API.

p4c

The P4 compiler, *p4c* is written in C++ too. The compiler consists of a generic frontend that accepts both $P4_{14}$ and $P4_{16}$ code which may be written for any architecture. It furthermore has several reference backends for the BMv2, eBPF, and uBPF P4 targets as well as a backend for testing purposes and a backend that can generate graphs of control flows of P4 programs [45]. The compiler is developed and maintained by P4.org.

2.4.3 P4 Inband Network Telemetry

P4 Inband Network Telemetry (INT) is a framework for the data plane to collect and report network state without requiring work from the control plane [4].

INT may provide telemetry about:

- which path is taken by the flow;
- the reason why the path was chosen;
- how long does it stay in each hop;
- what other flows are sharing the same physical link.

INT allows network switches and endpoints to insert additional custom headers into the packets that cross the network. These additional headers carry network measurement information such as timestamps, queue occupancy, congestion, etc [46]. INT can handle events that occur on a microseconds scale, also known as *microbursts* [47].

INT allows instrumenting the metadata to be monitored without modifying the application layer. The metadata to be inserted depends on the use case. For example, to monitor congestion, the programmer inserts queue metadata and transit latency [47].

INT makes it possible to perform instant processing in the data plane after measuring telemetry data (e.g. reroute flows when a link is congested) without having to interact with the controller [47].

The INT approach creates an overhead that may become an issue. So to avoid

exceeding the MTU along the path and to ensure that packet processing by 'standard', i.e. non-INT-enabled nodes is not impacted, the MTU must be adjusted.

INT terminology

This telemetry is built from the collection of network state data such as:

- Switch ID, or Node Id: ID of the INT-enabled network node adding the new metadata information to the packet.
- Ingress information:
  - ingress interface identifier: local ID of the port where the packet was received.
  - ingress timestamp: local node time when the packet was received.
- Egress information:
  - egress interface identifier: local ID of the port the packet will be sent out of.
  - egress timestamp: local node time when the packet left the node.
  - hop latency: time taken for the packet to traverse that specific node.
  - egress interface TX link utilization: current utilisation of the egress port via which packet will be sent out.
  - queue occupancy: ID of the queue in which the packet was temporarily stored, and measurement of the occupancy of that queue.
  - buffer occupancy: ID of the buffer in which the packet was temporarily stored and measurement of that buffer's occupancy.

An INT-enabled network has the following entities, as illustrated in the Figure 2.10 :

- INT source switch: a switch that sets the initial metadata that must be added into the packet by other devices;
- INT transit switch: a switch adding its own metadata to an INT packet after examining the INT instructions inserted by the INT source;
- INT sink switch: a switch that extracts the INT headers in order to keep the INT operation transparent to upper-layer applications;
- INT collector: a device that receives and processes INT packets.

INT Modes

INT provides different operation modes [5], as illustrated in Figure 2.11:

- **INT-XD** (eXport Data), also known as Postcard-based Telemetry (PBT): INT-enabled nodes directly export metadata from the data plane. No packet modification is needed. The postcard solution is an approach to minimize the overhead between the controller and the other nodes.
- **INT-MX** (eMbed instruct(X)ions): The INT Source node embeds INT instructions in the packet header, then the INT Source, each INT Transit, and the INT sink directly send the metadata to the monitoring system by following the instructions embedded in the packets. The INT Sink node strips the instruction header before forwarding the packet to the receiver. Packet modification is limited to the instruction header, so packet size does not grow as the packet

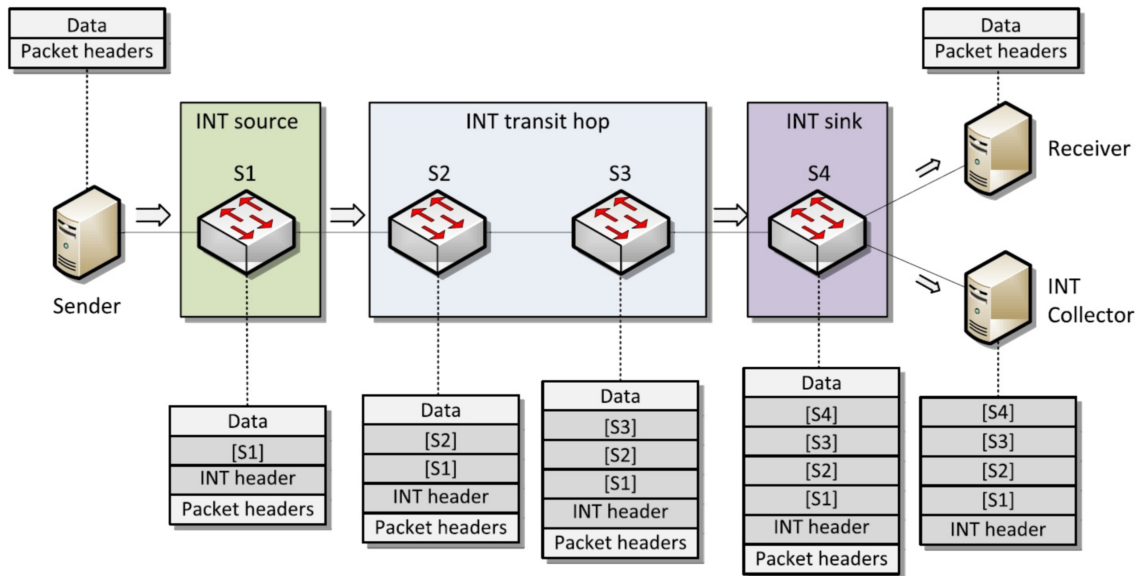

Figure 2.10: P4 INT-MD mode operation example [47].

traverses more Transit nodes. This mode requires the server receiving INT reports to correlate multiple postcards of a single packet passing through the network, to form the packet history at the monitor [47].

- **INT-MD** (eMbed Data): both INT instructions and metadata are written into the packets. This is the classic hop-by-hop INT illustrated in Figure 2.10.

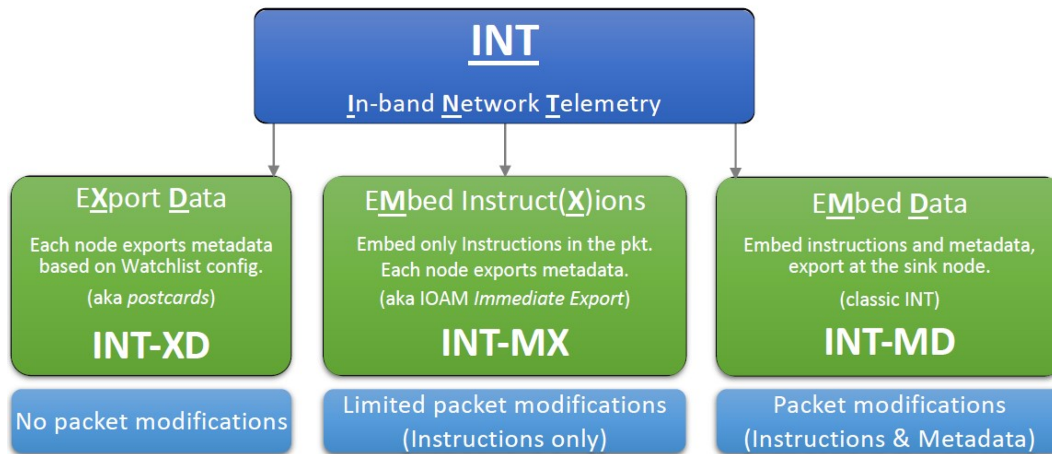

Figure 2.11: P4 INT operation modes [5].

INT collection

Not all data may need to be monitored, hence we have a pre-programmed set of flows named ***watchlist***. This flow *watchlist* is a table in the INT source switch that defines which flows the switch must match and apply the INT actions. Also important to note that the P4 switches in a network require to be configured as part of an **INT domain** to make sure they handle the INT flows as expected.

The INT packets are forwarded through a predetermined path, hence named the

INT path. These INT paths are used by the controller to continuously assess the data plane. However, there are conflicting requirements about INT paths:

- INT packets must traverse all nodes. If there are many paths, the controller receives many INT packets. If the number is minimized, the controller may receive only a few packets with too much information.
- INT packets latency must be similar. Or else, the controller does not have a complete view at a given time.
- INT packets must not overlap nodes, or else the redundant information also causes extra size and delay.

There has been some research on this topic that led to solutions such as using the Euler method or *GPINT* based on graph partitioning algorithm [48].

INT implementation

To allow for greater flexibility to the developers, the location of an INT header in the packet is intentionally not enforced in the specifications document [5]. For example, it can be inserted as a payload on top of Generic Routing Encapsulation (GRE), TCP, UDP, and Virtual Extensible LAN (VXLAN) [47] [5]:

- INT over IPv4/GRE - INT headers are carried between the GRE header and the encapsulated GRE payload.
- INT over TCP/UDP - A shim header is inserted following TCP/UDP header. INT Headers are carried between this shim header and TCP/UDP payload.
- INT over VXLAN - VXLAN generic protocol extensions are used to carry INT Headers between the VXLAN header and the encapsulated VXLAN payload.
- INT over Geneve - Geneve is an extensible tunneling framework, allowing Geneve options to be defined for INT Headers.

There are diverse implementations of INT. However we selected just the ones we found with code and suitable for our labs with BMv2 switches:

- **TCP-INT** [49]: Lightweight In-band Network Telemetry for TCP, is implemented in the TCP header as a new TCP option with three fields:
  - INTval: the link utilization (or queue depth if utilization is 100pct);
  - HopID: the ID of most congested switch (the packet TTL at the switch);
  - HopLat: the sum of latencies experienced across all hops.

  Each field has a corresponding echo-reply field for the receiver to echo the telemetry back to the sender. The switch-side P4 and control plane implementations are provided in a special release of Intel P4 Studio (aka Tofino SDE) so not suitable for Mininet.
- **ONOS-P4-INT** [50], with ONOS and eBPF, collecting INT with Prometheus [51] and Grafana [52], is fully documented [53].
- **INT MRI** [54], is a a scaled-down version of INT, named Multi-Hop Route Inspection (MRI). MRI allows users to track the path and the length of queues that every packet travels through. The program appends an ID and queue length to the header stack of every packet and at the destination, the sequence of switch IDs correspond to the path, and each ID is followed by the queue

length of the port switch.
- **INT-P4 ML** [55], for detecting intrusion with INT and ML models, seems well coded, but has no descriptions. It is similar to the MRI implementation (Multi-Hop Route Inspection), very similar to [54]. Uses a controller with specific Python dependencies.
- **Host INT** [56], measures packet loss and one-way packet latency between enabled hosts in the network, independently for each application flow. This project was ran by Intel but was discontinued early 2023.
- **Link Monitoring** [57], enables a host to monitor the utilization of all links in the network. This P4 program processes a source routed probe packet such that it is able to pick up the egress link utilization at each hop and deliver it to a host for monitoring purposes. This probe packet may be sent in a way to go through the whole network and back to the source.
- **GEANT INT-MD** [58] is the collaboration of European National Research and Education Networks (NRENs). This implementation is fully documented [59] and available for BMv2 and Tofino switches.
- **INT MD, XD, and MX** [60] also fully documented in [61], implemented a INT framework for Tofino switches.
- **INT-MD** [62], has minimal documentation, but is ready to run with minimum requirements with BMv2.

2.4.4 P4 language

The original proposal of the P4 programming language was written in July 2014 [63]. P4 is currently in its release 16 published in May 2017, but with latest review in May 2023 as version 1.2.4 [64]. $P4_{16}$ is intentionally a statically-typed, strongly-typed and memory-safe programming language: it has no support for pointers, dynamic memory allocation, floating-point numbers, and recursive functions [46].

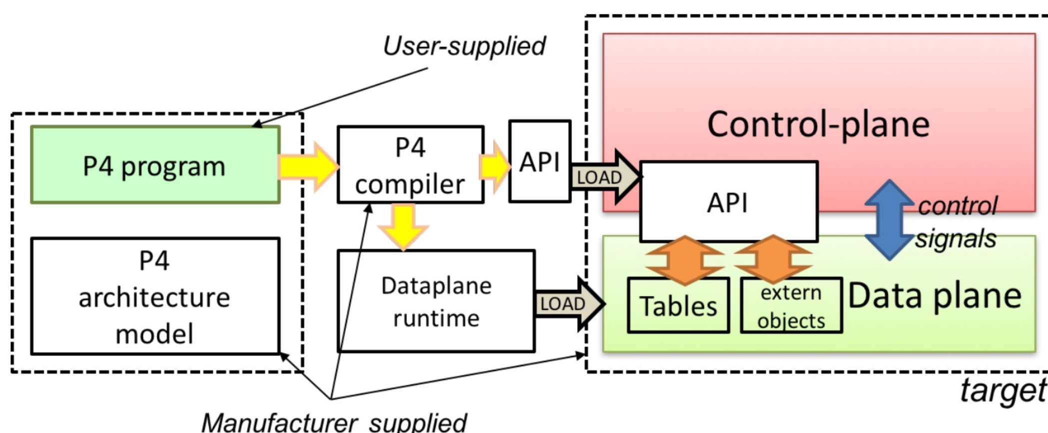

Figure 2.12: How P4 interacts with the targets [65].

As illustrated in Figure 2.12, the compiler maps the P4 program to the specific platform. The compiler, the architecture model, and the target device are supplied by the manufacturer. The P4 source code is supplied by the user. The compiler generates two packets after compiling:

- **data plane configuration**, data plane runtime, that implements the forwarding logic specified in the P4 input program. This configuration includes the instructions and resource mappings for the target.
- **runtime APIs**, that are used by the control plane to interact with the data plane. The APIs contain the information needed by the control plane to manipulate tables and objects in the data plane, such as the identifiers of the tables, fields used for matches, keys, action parameters, and others.

The P4 program is usually organized in several parts, including:

- **headers**: contains the packet headers and the metadata definitions.
- **parser**: contains the implementation of the programmable parser. The corresponding header is extracted from the packet and the values are then made available to the other routines.
- **ingress**: contains the ingress control block that includes match-action tables.
- **egress**: contains the egress control block.
- **deparser**: contains the deparser logic that describes how headers are emitted from the switch.
- **checksum**: contains the code that verifies and computes checksums.

P4$_{16}$ language limitations

Some of the P4$_{16}$ language features also contribute to its limitations [46]:

- **There is no iteration construct in P4**. Loops can only be created by the parser state machine.
- **There is no support for recursive functions**. In consequence, the work performed by a P4 program depends linearly only on the header sizes.
- **There is no dynamic memory allocation in P4**. P4 language does not provide mechanisms for allocating and deallocating memory dynamically at runtime. In other words, P4 does not support the creation or destruction of memory objects during program execution.
- **There are no pointers or references**.
- **There is no support for multicast or broadcast**. These must be achieved by means external to P4. The typical way a P4 program performs multicast is by setting a special intrinsic metadata field to a "broadcast group". This triggers a mechanism that is outside of P4, which performs the required packet replication.
- **P4 has no built-in support for queueing, scheduling or multiplexing**. P4 is unsuitable for deep-packet inspection. In general, due to the absence of loops, P4 programs cannot do complex processing of the packet payload.
- **P4 offers no support for processing packet trailers**. All the state in a P4 program is created when a packet is received and destroyed when the processing is complete. To maintain state across different packets (e.g., per-flow counters) P4 programs must use extern methods.

P4$_{16}$ externs

P4$_{16}$ supports *extern* functions or methods, which are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is hard-wired (e.g. checksum units) and hence not programmable using P4. There is currently an effort to standardize a set of such methods. However, each P4 target platform can provide additional *extern* methods, e.g., to model hardware accelerators. Invoking *extern* methods is one way that P4 programs can perform otherwise impossible tasks [46].

2.5 Control Plane security with P4

Compared to OpenFlow, the P4 programmable data plane has three key advantages when dealing with security: per-packet visibility, scalability, and high-speed processing capability [38]:

- **Visibility** of each packet means that attack detection algorithms can be developed in the switch hardware and applied to each individual packet instead of sampling towards the controller.
- **Scalability** means that since the defense is located directly in the switch, it will expand with the network scale and speed, making the centralized controller no longer a bottleneck. Most of the defense measures are completed by the data plane, so the controller only receives and maintains a small number of statistical results, distributes flow tables, etc., which can reduce the complexity of the network and ensure the scalability of the defense scheme.
- **High-speed processing** capability means that a lot of tasks can be done at line rate. The programmable data plane can execute local policies, process packets at line rate, and respond quickly to attack behavior. Once an attack is detected, measures can be taken immediately to mitigate the attack on the switch without causing a round-trip time delay to the remote controller. Functions originally implemented by software can be offloaded to the data plane, so that the data plane transmits the results of execution to the controller instead of raw data.

This way, P4 can further assist in enforcing policies in a programmable data plane, where the software that describes how the packets are processed helps to detect anomalies with no latency at line-rate. Functions such as access control, privacy, encryption assuring availability, and integrated defense are mostly assured by the programmable data plane, thus offloading the controller [38].

One example is the DDoS detection and mitigation solutions fully deployed in the data plane such as *EUCLID*, which relies on a statistical analysis of Shannon entropy to characterize legitimate traffic [66]. Other is the establishment of secure channels between switches by implementing Diffie-Hellman key exchange and then using Advanced Encryption Standard (AES), all programmed with P4 in the data plane [67]. Another example, as illustrated in Figure 2.13, is a firewall concept named P4Guard [68]. In this concept, the controller is only required

to translate the high-level firewall policies into match-action table rules on the switch or to detect various flooding attacks from the collected statistics [29].

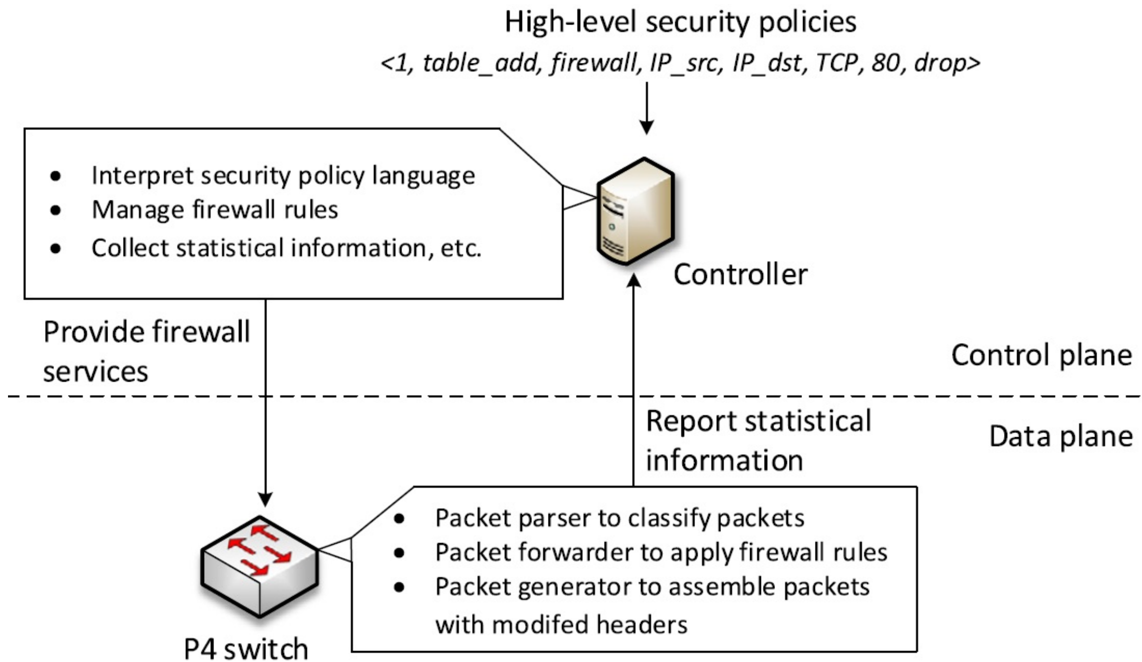

Figure 2.13: Example of a P4 switch configured as a firewall, managing most secure features [29].

A consequence of having the switches taking care of security features is that it becomes difficult to confirm if the data plane behavior corresponds to the network policies. As the current testing focuses on finding bugs in the programs, there is an implicit trust in the devices [69]. As such, there is a need to evaluate in run time if the network is behaving as expected, resorting to several strategies as:

- Controller fault tolerance;
- Data plane consistency;
- Port knocking security;
- Runtime verification;
- Adversarial data plane verification;
- Securing P4RT.

Next, we discuss each one in more detail.

2.5.1 Controller fault tolerance

In SDN, controller replicas are distributed and their state is replicated for high availability purposes. Malicious controller replicas, however, may destabilize the control plane and manipulate the data plane, thus requiring fault tolerance. One such approach is theByzantine Fault Tolerance (BFT), which is a decentralized, permissionless system capable of successfully identifying and rejecting fake or faulty information. In deployments where application flows share the same infrastructure as the control flows, the traffic arriving from controller replicas im-

poses some overhead. As such, solutions such as *P4BFT* [70] leverage an optimal strategy to decrease the total amount of messages transmitted to the switches.

2.5.2 Data plane consistency

SDN operates under the premise that the logically centralized control plane holds an accurate representation of the actual data plane state. Unfortunately, bugs, misconfigurations, faults, or attacks can introduce inconsistencies between the network control and the data plane that can undermine the correct operation at runtime.

With the aim to verify the control-data plane consistency, *P4CONSIST*[71] detects inconsistencies between control and data plane in P4 SDNs. *P4CONSIST* generates active probe-based traffic continuously or periodically as an input to the P4 SDNs to check whether the actual behavior on the data plane corresponds to the expected control plane behavior.

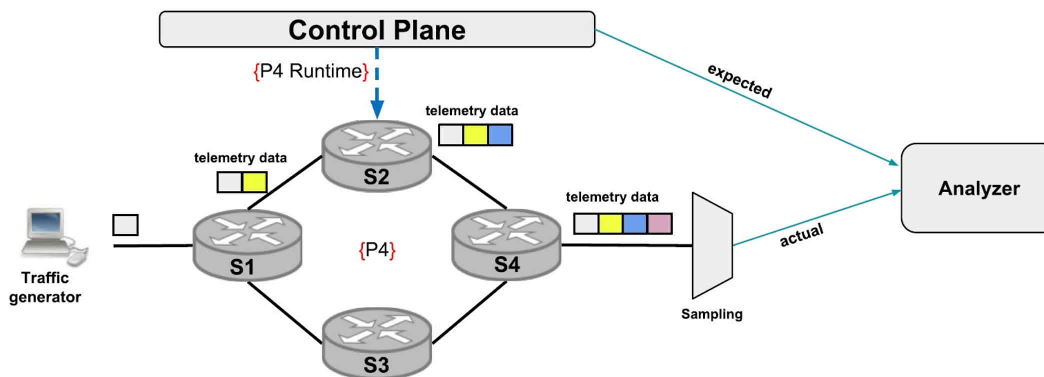

Figure 2.14: P4Consist architecture [71].

This solution may create some load in the control plane, hence *P4Update* [72] proposes partial offload of the consistency control and most of the routing update logic to the data plane. *P4Update* enables switches to locally verify and reject inconsistent updates thus reducing control plane preparation time and improving its scalability.

2.5.3 Port knocking security

Port knocking, introduced by [73], is an authentication mechanism used to hide services from unauthorized users and avoid undesired connection attempts. This way, a service port appears to be closed until the user generates a connection attempt on a preconfigured set of closed ports [74]. In the example in Figure 2.15, the client sends three TCP SYN packets to 3 ports, upon which the server enables the SSH port 22.

This concept has been used within P4 to offload the hosts from dealing with unintended traffic. Instead of being enabled in the hosts, this service can be part

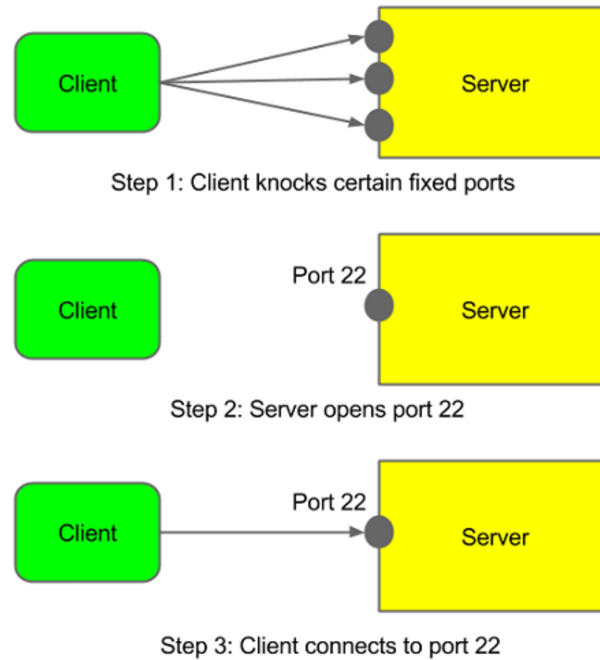

Figure 2.15: Port knocking example [75].

of the P4 switches as proposed by *P4Knocking* [74]. *P4Knocking* requires minimal configuration since the control plane can expose an interface to let applications define the knock sequence for a given destination address [74].

To be able to track the sequence of ports, *P4Knocking* relies on registers to track the state of the knock sequence for a given source IP address.

The control plane involvement depends on the possible implementations, as illustrated in Figure 2.16:

- **Full data plane offloading**, makes the switch responsible for statefully keeping track of knocks and approving clients to connect to the destination server. However, being able to keep a register per existing IP address may require a huge register size.
- **Hybrid control and data plane offloading**, when a new client initiates the knock sequence, the P4Runtime server (switch) sends a *Packet In* message to the controller via a gRPC stream. Upon receiving the packet, the controller assigns the first available ID to the corresponding IP address. To facilitate this assignment, the controller inserts a rule into a table that maps the source IP to the assigned ID. This table is applied and matched each time the switch receives a new packet. If a match is found, the switch statefully tracks the knock sequence, similar to the previous case. The controller continues assigning IDs to source IPs as long as there are available IDs. It also keeps track of the oldest assignments to ensure that new clients can still be assigned an ID, thus enabling the reuse of IDs.
- **Main control plane and minimal data plane offloading**, delegates most of the tasks to the control plane. Instead of statefully keeping track of the knock state, the controller is in charge of this task. The benefit of offloading most of the tasks to the control plane is that in terms of memory, the *P4Knocking* application

running at the controller would be able to allocate only the necessary resources in a much more efficient way.

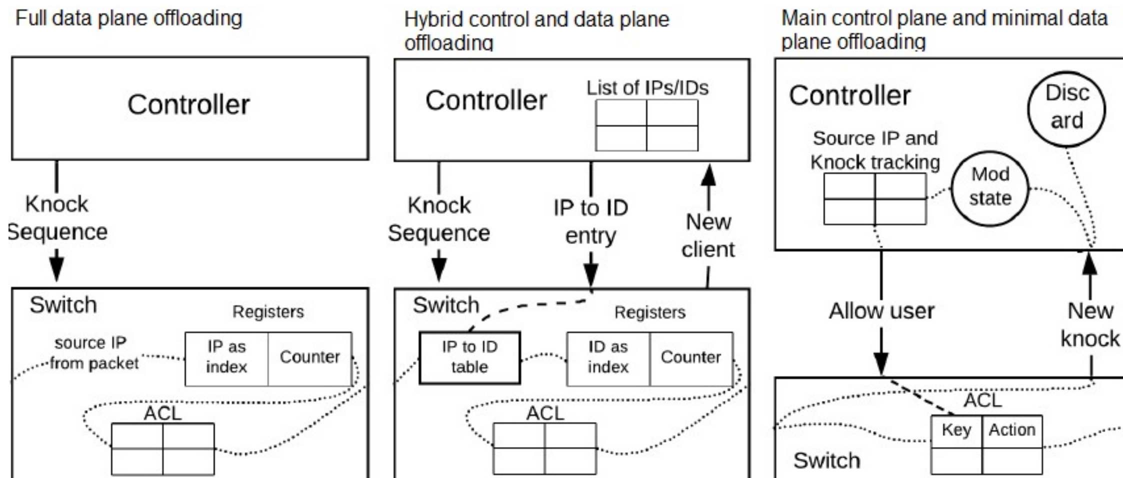

Figure 2.16: Port knocking options, adapted from [74].

Each of these three modes has its own advantages and disadvantages but the controller is always required. This solution is also vulnerable to MITM attacks due to [76]:

- unsecured transfer of the port knocking sequences between the SDN controller and hosts, and
- lack of host identity verification mechanisms post port-knocking authentication.

An attacker could eavesdrop on the port-knocking messages from a host and then spoof and replay it, as illustrated in Figure 2.17.

A possible solution is *P4-sKnock* [76], a P4-based two-level host authentication and access control mechanism:

- The first level requires that the hosts and the controller have public keys. The controller uses the host public key to encrypt the port-knocking sequence and sends it to the host via the switch. The host decrypts with its own private key and sends the port-knocking code back, encrypted with the controller's public key. If something is wrong, the host is put under quarantine.
- The second level assures host identification, with a challenge-response mechanism with the controller sending an encrypted nonce to the host. The controller confirms if the host is able to decrypt and sends the nonce back, and only then instructs the host to create the switch to forwarding rules. If something is wrong, the host is put under quarantine.

2.5.4 Runtime Verification

The P4 runtime verification aims to assure the dynamic forwarding rules provided by the control plane and the static rules deployed with the P4 program.

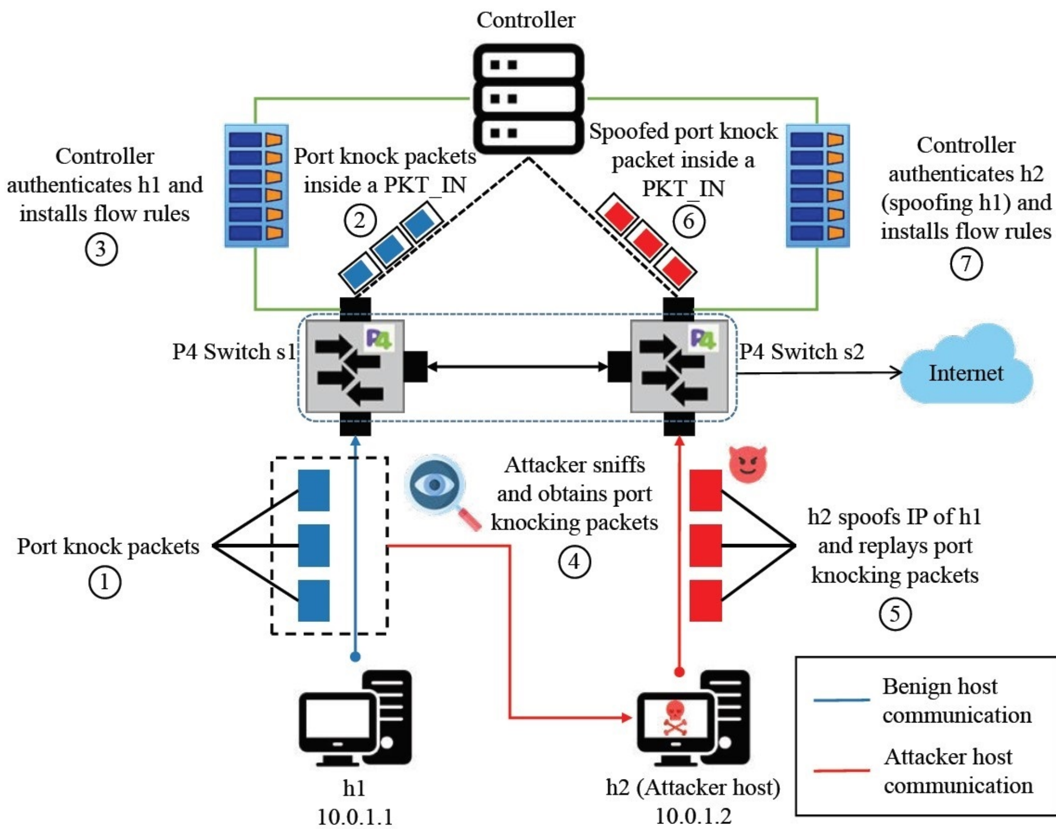

Figure 2.17: Port knocking attack [76].

Bugs can happen anywhere: in the P4 program, the controller installing rules into tables, or the compiler that maps the P4 program into the switches. Most of these bugs appear after specific sequences of packets with specific combinations of rules in the tables. Some testing can be made automatically such as with *p4pktgen* [77], a tool that creates test cases automatically and can be used in the development phase.

Nevertheless, as the contents of the Match-Action Tables (MAT) are not known until they are populated by the control plane at run time, other solutions are required.

One such solution is fuzz testing, which generates semi-valid, random inputs which trigger abnormal program behavior. For instance *P4RL* [78] relies on fuzz testing with reinforced learning. The feedback is generated using the control plane configuration and the queries are defined with a proposed query language *paq*. This query language allows to specify the expected network behavior and compare it with the actual behavior.

Another approach [79] uses the Ball-Larus algorithm to test the switches in runtime or *p4v* [80], which was confirmed to scale well (a known limitation of symbolic execution test approaches) . Similarly, *DBVal* [81] compares the intended execution path with the observed path.

Another solution is *P4DB* [82], which helps to find bugs. However, this solution resides in the controller, which may lead to congestion the control channel and

exhaust the CPU.

Some of these solutions track packet behaviors through postcards, but under run-time this could impact the bandwidth and switching capacity.

To assure full coverage and high scalability, *KeySight* aims at aggregating packets with identical packet behavior[83] [84]. The new equivalent postcard is named Packet Equivalence Class (PEC) and reduces the number of postcards by one to three orders of magnitude while monitoring all behaviors with minor false positives. As illustrated in Figure 2.18, at runtime every packet traversing the P4 pipeline gets a postcard according to the PEC representation. Then, *KeyTracker* checks postcards and determines whether to report the postcard that has never been seen by *KeyTracker* before. After collecting postcards from every switch, *KeyVisor* conducts troubleshooting tasks and makes the other components transparent to operators with troubleshooting service APIs.

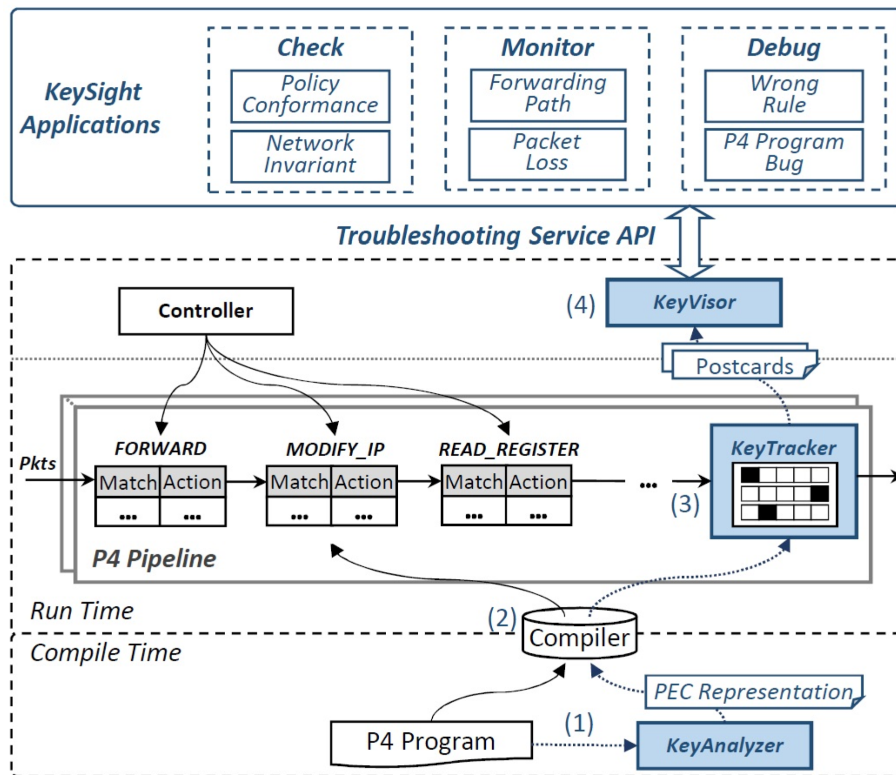

Figure 2.18: P4 KeySight overview [83].

2.5.5 Adversarial Data Plane Verification

Unlike regular program testing, adversarial testing stresses low-probability edge cases in a program.

These mechanisms help to detect anomalous behavior from potentially compromised switches with Adversarial Data Plane Verification (ADPV) solutions. These solutions differ from non-adversarial verification mechanisms in that their designs implicitly assume that an attacker may attempt to hide their activities from

the Control Plane, meaning that data received from an individual switch is not automatically trusted [69].

One such ADPV is *P4wn*[85], a program profiler that can analyze program behaviors of stateful data plane systems. *P4wn* takes in the source code of a data plane system as input and performs program analysis to generate stateful sequences to trigger all program behaviors in a fully automated manner.

If the switches are attacked, and their behavior changes, it is possible to falsify runtime statistics and hide attack patterns from the controller. So, we could envision a scenario, as illustrated in Figure 2.19, in which a compromised switch does not send the corresponding attack telemetry as it should. The postcards (INT-XD) sent to the INT collector are what would be expected if there were no attacks.

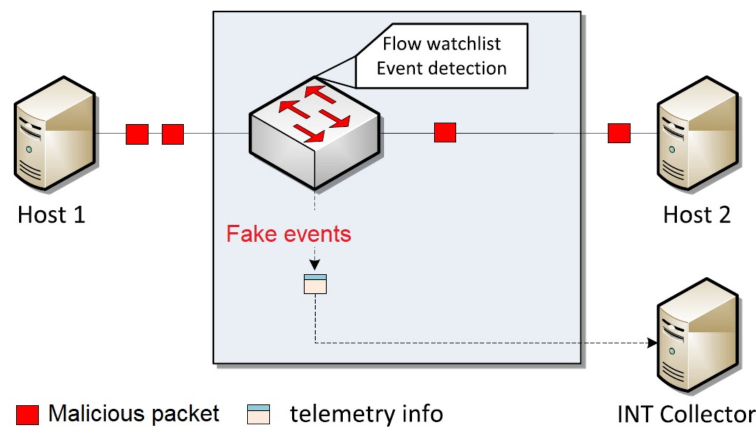

Figure 2.19: P4 compromised switch sending fake INT-XD data.

The actual compromising of the switch may be the result of an attack in which the table entries are manipulated by intercepting the P4 runtime communication as described in [69].

2.5.6 P4RT security

As P4RT is supported in gRPC, this communication benefits from the following built-in mechanisms [86]:

- TLS to authenticate the server, and encrypt all the data exchanged between the client and the server. Optional mechanisms are available for clients to provide certificates for mutual authentication.
- ALTS, as a transport security mechanism, if the application is running on Google Cloud Platform (GCP).

So, using TLS to authenticate and encrypt the gRPC channel can prevent man-in-the-middle (MITM) attacks between the server and the client. Mutual TLS (mTLS) may be used to facilitate the authentication of the client by the server and vice-versa.

2.6 Summary

In this chapter we introduced the main concepts of SDN, starting with its splitted architecture between the application, control and infrastructure layers. Then we discussed the SDN controller conceptual role and operation, as well as the strategies to mitigate the risk of the single point of failure as well as to improve its availability.

OpenFlow was also addressed as it is the current *de facto* standard in the controller southbound interface, along with its main limitation: fixed set of headers fields. Then discussed some of the most important topics about securing the control plane: securing the OpenFLow channel and protecting against DoS.

The P4 language was introduced as an improvement in SDN, as the data plane can be programmable, which unleashes several benefits such as defining new protocols, customizing packet processing functions, offloading applications to the data plane, having these functions work at line speed, and all of this with potentially lower costs. Then we discussed the P4 ecosystem components including the Southbound interface named P4Runtime, P4 telemetry and its language.

Finally, we addressed the control plane security with P4, with focus on controller fault tolerance, data plane consistency, port knocking, runtime verification, adversarial data plane verification, and P4 runtime security.

Chapter 3

Research objectives and approach

This chapter describes the research carried out in this work, the main objectives, the adopted approach and use cases.

In Section 3.1, we present the research approach and outcomes, and in Section 3.2, we describe the research objectives. Following, in Section 3.3, we cover the approaches and use cases adopted to accomplish the objectives listed, including some preliminary results.

The work carried out in the first semester can be summarized in Figure 3.1.

Figure 3.1: Gantt chart view of the activities performed in the first semester.

During the research phase, we have identified a number of important topics which could be further analyzed, so we have gone through a process of prioritization as described in Section 3.3.

The research phase led to the planning of the activities for the second semester with further focus on P4 INT. Our work focused on P4 INT, how it can be used as a security control, how it could be compromised, and how it could be protected. These tasks are summarized in Figure 3.2.

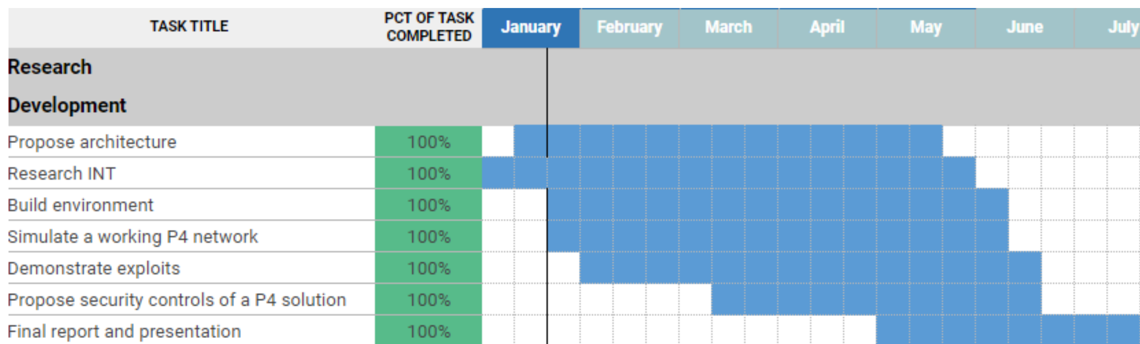

Figure 3.2: Gantt chart view of the activities completed in the second semester.

3.1 Research

In our initial research, we have gone through the SDN evolution and security risks. Then we focused on P4, for which the main sources are academic articles. In parallel, we have gone through hands-on labs with P4. Some of these labs include exercises and tutorials.

As part of the research, we have worked on several environments, such as:

1. P4 tutorial, as the main entry point;
2. P4 infrastructure by Univ South Carolina, as a basis of a course, and a week-long workshop;
3. NG-SDN, including ONOS;
4. P4Pi, P4 running on Raspberry Pi.

In these environments, the network is usually simulated with *Mininet* [87]. *Mininet* is a network emulator which creates a network of virtual hosts, switches, controllers, and links. These environments are deployed as a single Virtual Machine (VM) in VirtualBox [88] - a software virtualization package that installs on an operating system as an application. VirtualBox allows additional operating systems to be installed on it, as a Guest OS, and run in a virtual environment.

It became apparent that the simulation of all parts working together is hard to achieve and maintain. A complete P4 lab must simulate controllers, switches, INT collectors, P4RT (gRPC and protobuf), and be able to demonstrate an attack. Also, the environment must be easily replicable by others so that the demonstrations can be reproduced.

In the P4Pi ecosystem, the network is not simulated but rather comprised by the hosts that connect to its WiFi interface.

3.1.1 P4 tutorial

The P4 tutorial [89] is the main starting point to learn P4. It includes several exercises and an ecosystem packed with *Mininet* plus P4. The exercises help to understand, test and simulate several examples with P4 in a *Mininet* environment, from which we highlight the P4Runtime exercise, in Section 3.1.1, and the link

monitor exercise, in Section 3.1.1. These exercises offered information that were later used for the implementation of a lab.

P4Runtime exercise

The *P4Runtime* lab [90] provides a demonstration of the *P4RT*, *gRPC*, ingress and egress counters, and a simple controller in Python language. This controller establishes a *gRPC* connection to the switches so the tables can be dynamically created via P4RT.

Link monitoring exercise

The link monitoring lab introduces the concept of probe packets. This is the concept of the INT path packet as described in Section 2.4.3.

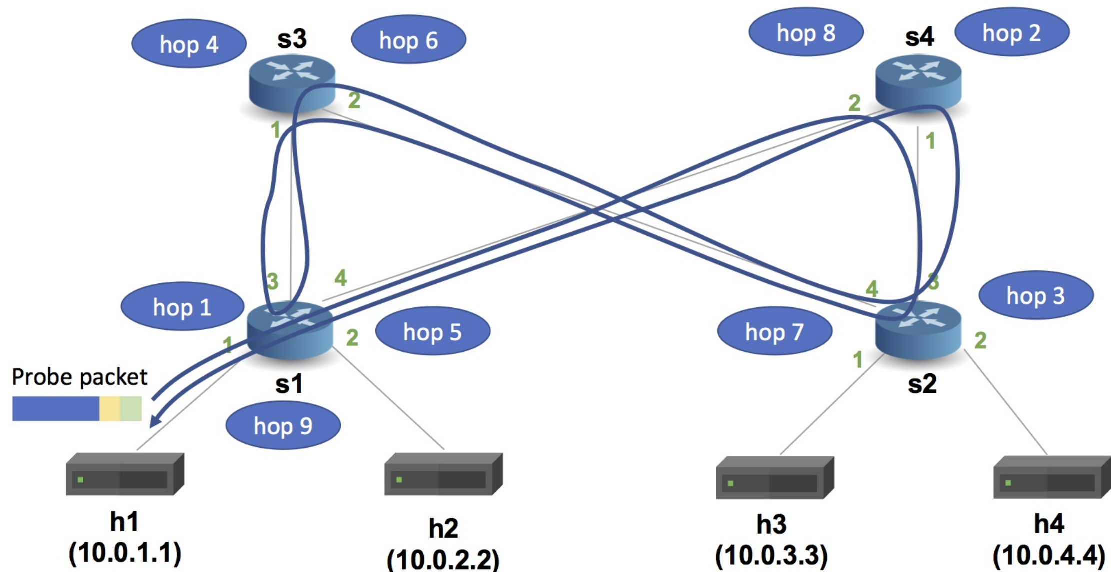

Figure 3.3: P4 tutorial scenario for the link monitor exercise.

With this exercise we identified the effect of traffic in the measurement of the used bandwidth in the network, as detailed in the Figure 3.4.

3.1.2 P4 infrastructure by the University of South Carolina

The University of South Carolina offers two virtual platforms for cyber training purposes [91], some tutorials and exercises along its own set of VMs. These VMs include *Mininet* + P4, *MiniEdit* and other tools:

- Virtual Labs on P4 Programmable Data Plane Switches (BMv2). The lab series explains topics that include parsing, match-action tables, checksum verification, and others.

```
Switch 1 - Port 1: 0.001 Mbps
Switch 4 - Port 2: 0.001 Mbps
Switch 2 - Port 3: 0.001 Mbps
Switch 3 - Port 2: 0.001 Mbps
Switch 1 - Port 3: 0.001 Mbps
Switch 3 - Port 1: 0.001 Mbps
Switch 2 - Port 4: 0.0 Mbps
Switch 4 - Port 1: 0.0 Mbps
Switch 1 - Port 4: 0.0 Mbps

Switch 1 - Port 1: 0.081 Mbps
Switch 4 - Port 2: 0.001 Mbps
Switch 2 - Port 3: 0.001 Mbps
Switch 3 - Port 2: 0.001 Mbps
Switch 1 - Port 3: 0.001 Mbps
Switch 3 - Port 1: 0.079 Mbps
Switch 2 - Port 4: 0.079 Mbps
Switch 4 - Port 1: 2.479 Mbps
Switch 1 - Port 4: 2.464 Mbps
```

Figure 3.4: Monitoring with a P4 probe, before and after simulating traffic with *iperf*.

- Virtual Labs on P4 Programmable Data Planes: applications, stateful elements, and custom packet processing. The lab series explains topics that include metadata, registers, counters, meters, advanced parsing, and others. We appreciate the support of Prof. Jose Gomez who granted us the access to this environment.

These VMs are very stable but it was not possible to install additional software. So, these may not suitable only for a PoC, not for the test platform.

3.1.3 NG-SDN

The NG-SDN environment [92] is available within a single VM that include:

- *Mininet* with P4;
- Data plane programming and control via P4 and P4Runtime;
- Configuration via YANG, OpenConfig, and gNMI;
- Stratum switch OS;
- ONOS SDN controller.

The ONOS GUI provides an interesting user experience as depicted in Figure 3.5.

We found that the environment has many moving parts, leading to some instability, thus making this environment unsuitable for a test platform.

This environment is also used as a base for the exploit described in [69], which however requires extensive preparation as described in the author's GitHub [93]. We were not able to reproduce the exploit as the VM frequently becomes unstable and the work is lost. However this attack is quite interesting as the controller is oblivious to the events, as described in Figure 3.6.

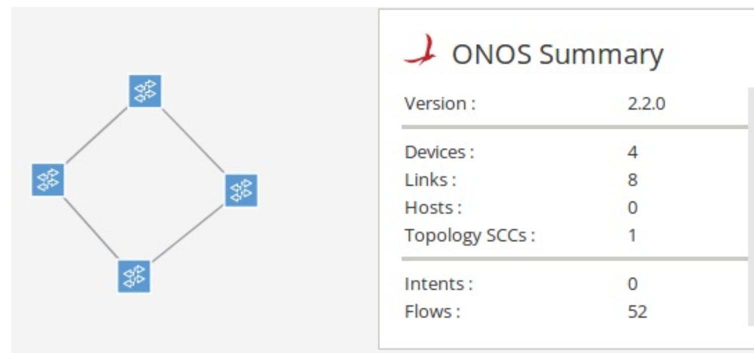

Figure 3.5: Excerpt of ONOS GUI in NG-SDN VM.

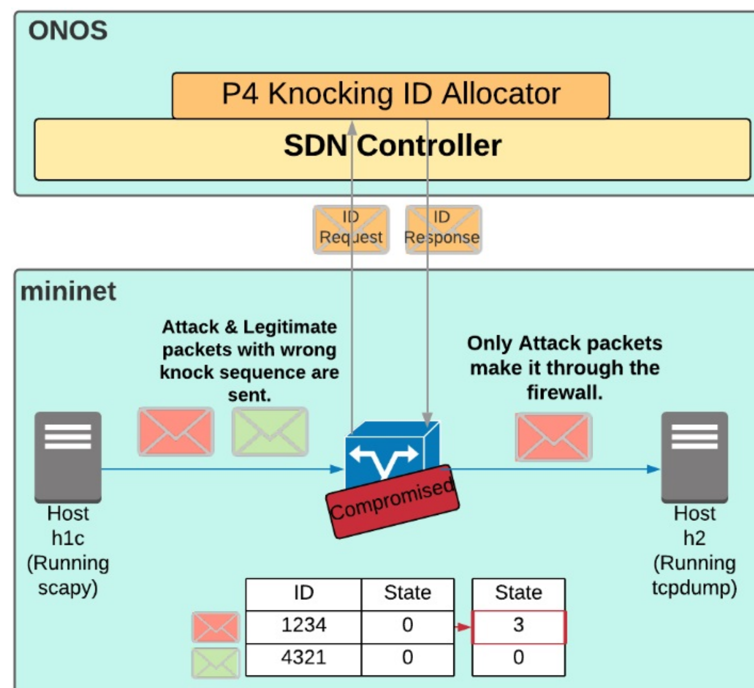

Figure 3.6: Overview of an attack in which a compromised switch reports false P4 Port Knocking Ids to the controller [93].

3.1.4 P4Pi

P4Pi [94] is a low cost, open source platform for computer networks teaching and research, based on the Raspberry Pi board and P4 programming language. P4Pi is also supported by ONF with its own resources online [95]. P4Pi supports t4p4s (compiler for the software-based Data Plane Development Kit (DPDK) switch) and BMv2 switches (open source P4 switch).

The Figure 3.7 describes how the control and data planes are separated in the P4Pi architecture.

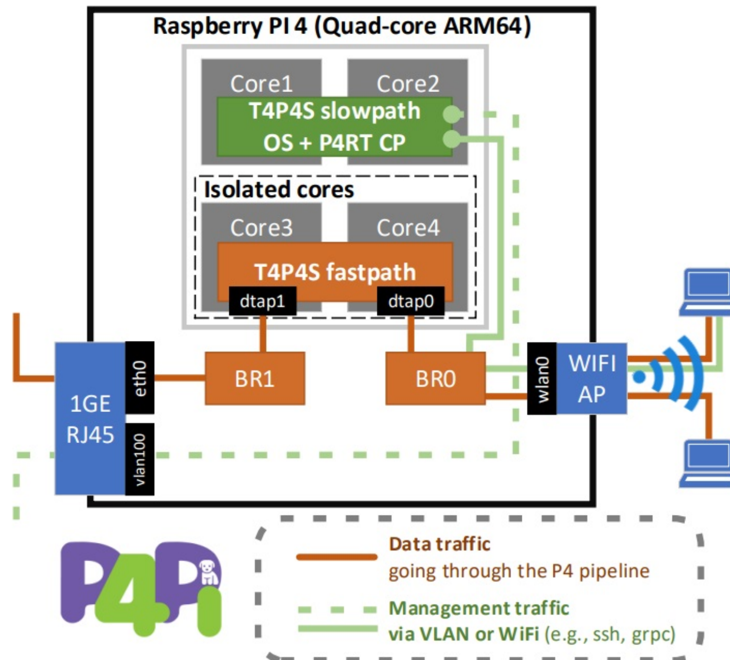

Figure 3.7: P4Pi overview.

This environment requires some effort to be able to simulate interesting security-related use cases. As such, we planned to use *Mininet* in the preliminary architecture and then test a similar P4 deployment in the P4Pi devices.

3.2 Objectives

The broad objectives of this work are to research the security vulnerabilities in SDN with P4 and how to mitigate them: given the huge extent of security topics that could be addressed, we focused on the P4 INT framework as a security control.

Along with this research, we learned that with the enhanced offloading of security features to the data plane, the control plane may become unaware of some attacks.

This opens up a few interesting lines of investigation:

1. considering a compromised switch, can the tables be manipulated without the controller being aware?
2. can we use INT as a security control?
3. is it possible to eavesdrop INT?
4. would an attacker be able to do a INT replay attack?
5. can P4 INT be manipulated, so that the telemetry don't show any misbehavior?
6. is it possible to manipulate the INT with MITM?
7. how to protect against MITM attacks?

These questions led to some more analysis and tests as described in Section 3.3.

3.3 Approach and use cases

Following the line of investigation described above, we have gone through some tests as described in the following sub chapters.

3.3.1 Compromised switch, tables manipulated

This use case is similar to what is described in [69]. The demonstration is detailed in the author's GitHub [93] but the process is difficult and the VM became unstable.

However it could be beneficial to focus on a more stable environment. A proposed scenario would be, as described in Figure 3.8:

- H1 is approved to access the server;
- S1 is compromised;
- S1 MATs are changed and foe host can access the server;
- controller reads the correct MATs from S1.

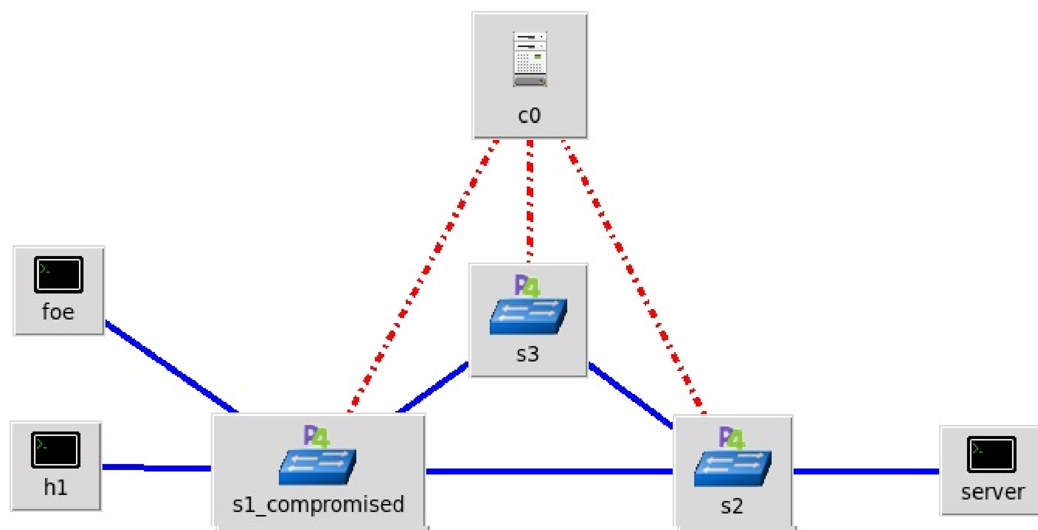

Figure 3.8: Foe host gets access to the server and the controller is unaware.

Given the instability of the NG-SDN environment, and the requirement to have the switch pre-compromised, we considered this scenario of low priority and didn't pursue on this.

3.3.2 Compromised switch, INT manipulated

This use case builds on top of the previous one: the INT data generated in the compromised switch don't include data from the foe host, as described in Figure 3.8:

- H1 is approved to access the server;
- S1 is compromised;
- S1 MATs are changed and foe host can access the server;
- The controller reads the correct MATs from S1;
- INTC host is an INT collector which will not get any information from the foe accessing the server.

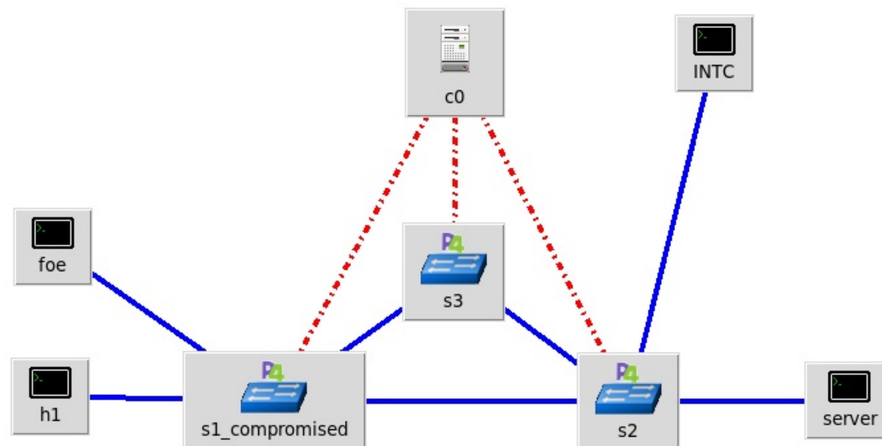

Figure 3.9: Foe host gets access to the server and the controller is unaware as well as the INT collector.

Given the difficulties with the NG-SDN environment, and the requirement to have the switch pre-compromised, we consider this scenario of low priority too.

3.3.3 MITM attack against INT

A MITM attack against P4RT is not feasible due to the gRPC specifications described in Section 2.5.6. However, a MITM attack against INT is probably possible, as described in Figure 3.10:

- INTC is an INT collector which gets information from traffic towards the server;
- Foe host acts as INT MITM and hides its own traffic.

This scenario looks feasible and highlights a potential security issue related to the lack of authentication and encryption of this traffic. We will focus on this scenario, exploring INT as a potential security control. As such, P4 INT may be

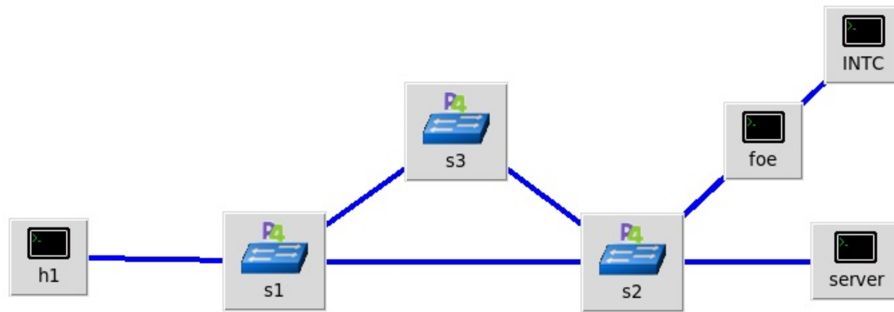

Figure 3.10: Foe host listens and hides its activities with a MITM attack.

a potential victim of a malicious actor via a MITM attack. We will explore these attacks as well as mitigation solutions to protect the INT platform.

3.3.4 Mitigation of MITM attack against INT

After demonstrating a MITM attack against INT, the next step is to propose mitigation approaches. These mitigation techniques shall address the protection of the network, authentication of the INT sink and INT collector, as well assurance of the integrity of the content.

Chapter 4

Preliminary work

This chapter presents the activities and results related to the first iterations. As described in the Section 3.3, we selected an architecture without a controller, as the focus is evaluating attacks through INT.

As a way to mimic a standard topology in a data center, we have chosen the Spine-Leaf architecture as described in Figure 4.1. The spine layer serves as the backbone of the network and the leaf switches connect to end devices.

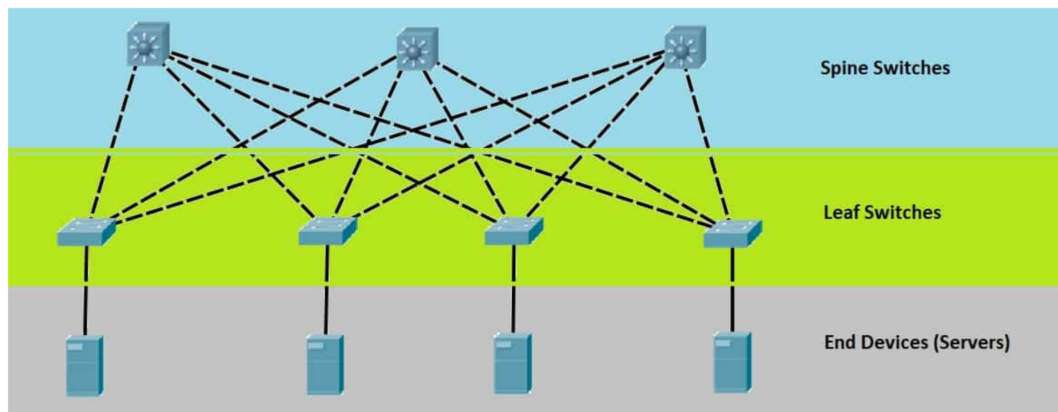

Figure 4.1: Standard Spine-Leaf architecture [96].

As a way to create such scenario, we used Mininet [87] and *MiniEdit* [97].

Starting from P4 tutorials such as the basic routing [98], then including link monitoring [99] and ARP [100] support, we built a code that could inject probes and thus collect egress port statistics in each hop. These probes enable a host to monitor the utilization of all links in the network. This P4 program processes a source routed probe packet such that it is able to pick up the egress link utilization at each hop and deliver it to a host for monitoring purposes. This probe packet may be programmed to go through a predetermined sequence of hosts, e.g. go through the whole network and back to the source, as referred in 3.1.1.

This served as a PoC to assess the *MiniEdit* and collect a basic collection of statistics per egress port:

- byte_cnt - counts the number of bytes transmitted out of each port since the

last probe packet was transmitted out of the port;
- last_time - stores the last time that a probe packet was transmitted out of each port.

At the collector, these statistics are collected with a Python script that does the necessary math per egress port to calculate the bandwidth: 8 * byte_cnt / (current_time - last_time).

We explored this Proof of Concept in the topology illustrated in Figure 4.2.

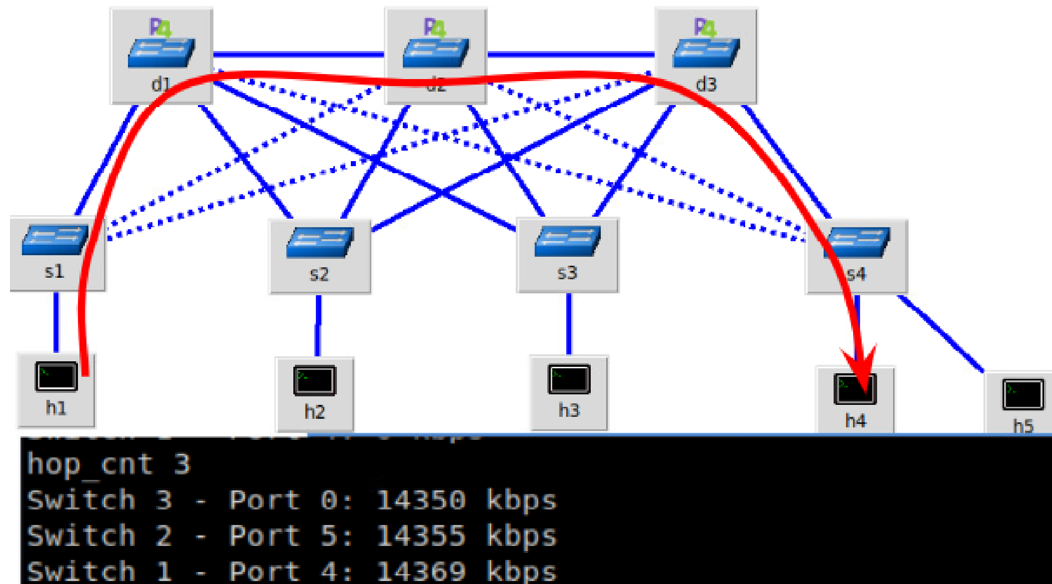

Figure 4.2: INT early test environment and statistics collection with a crafted probe.

The VMs we used with Miniedit posed difficulties to install additional software so we moved on to Mininet and searched for open-source implementations. We analyzed some INT implementations with collection and visualization and selected three from GitHub:

1. **INT MD, XD, and MX** [60] also fully documented in [61], implemented a INT framework for Tofino switches. The repository refers to have been tested with Tofino switches only, so we didn't choose this code.
2. **GEANT INT-MD platform** [58], also documented in [59], is a INT-MD implementation for BMv2 and Tofino switches. This code is out outdated and not supporting INT v2.1.
3. **INT-MD** [62], has minimal documentation, but is ready to run with minimum requirements with BMv2 switches.

Although undocumented, the last implementation in the above list worked seamlessly. We used this code and included the ARP support from a P4PI example [100]. This code is available in our P4-INT for Mininet GitHub repository [101].

With these P4 source code, we built a virtual network with Mininet running P4 BMv2 switches supporting INT-MD.

The scenario in the Figure 4.3 aims at simulating a P4 network in which two

clients request data from a protected server. The INT framework collects data from the data flowing through this network. The host h4 is an INT collector that also runs a Influxdb [102] database and a Grafana [52] system for enhanced visualization. A rogue host, h5, controlled by an adversary, is able to act as a MITM and eavesdrop or manipulate the INT data stream, hence able to hide an ongoing attack.

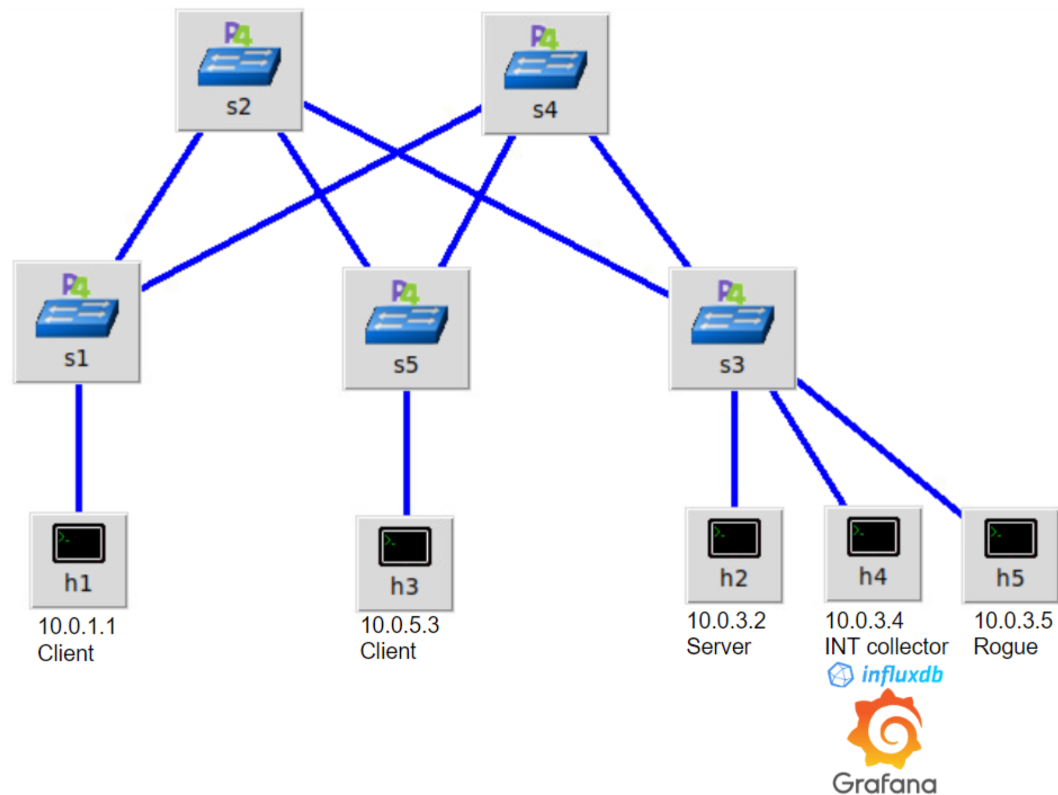

Figure 4.3: INT preliminary environment.

The INT operation is similar to what is described in Figure 2.11. In this scenario, the INT flow can be described in the following steps:

1. Source data: one host, e.g. client h1 or h3, sends some data to a server, h2, through a P4 network.
2. INT source: if the data sent from the clients matches the pre-programmed watchlist, then the switch s1 or s5, adds the INT header and statistics to this packet.
3. INT transit. The transit switches, s2 or s4, add their INT statistics to the same packet.
4. INT sink. The sink switch, s3 strips all added INT payload and sends the original data to the server h2. Then creates a new packet with the retrieved INT data of the previous switches and adds its own statistics. The INT information, of the 3 switches, is encapsulated in a new UDP packet towards the INT collector.

This scenario can thus be split in the following parts:

1. simulate an INT platform;

2. collection of INT statistics and visualization;
3. attacking the INT platform;

4.1 Simulate an INT platform

This Mininet network must create INT statistics and send those to the collector. In this scenario, if the data sent by h1 matches the watch lists in s1 and s3, we can describe the INT operation as in Figure 4.4.

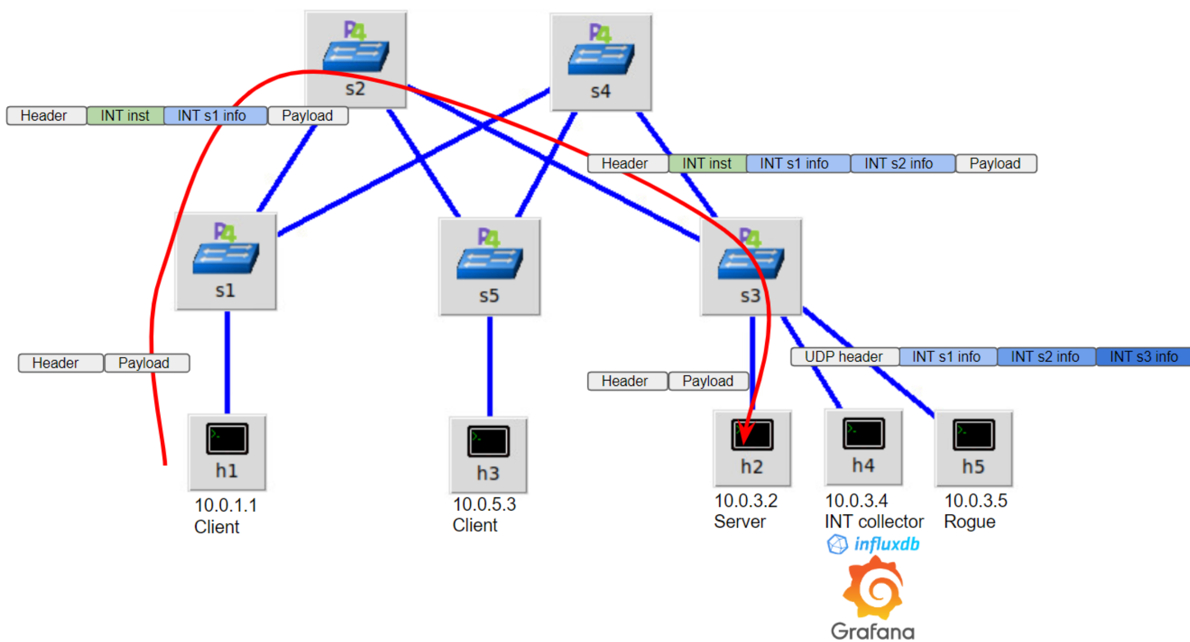

Figure 4.4: Example of processing an INT-MD packet.

As part of the scenario, the h2 server is simulating three services: PostgreSQL, HTTPS and HTTP. So, the switches s1 and s5 are pre-configured as INT source and also are pre-configured with the match list for source and destination IPs and L4 ports: 5432 for PostgreSQL, 443 for HTTPS and 80 for HTTP.

All code and details are documented in our INT for Mininet GitHub repository [101].

4.1.1 Packet source

INT packets are only generated if a specific packet matches the *watchlist*. So, we used the *scapy* library within a Python script to craft the packets. This is a Python script that takes as input parameters the destination IP, the L4 protocol UDP/TCP, the destination port number, an optional message and the number of frames to be sent. This script was adapted from [62].

Additionally, we included a command to simulate recurrent accesses to the server, e.g., every 5 seconds access to HTTPS, from the h1 and h3 hosts' CLI:

```
watch -n 5 python3 send.py --ip 10.0.3.2 -l4 udp --port 443 --m
    INTH1 --c 1
```

This frame is only carrying the content "INTH1" as captured at the exit of h1 interface in Wireshark, displayed in the Figure 4.5.

```
▸ Frame 23: 47 bytes on wire (376 bits), 47 bytes captured (376 bits) on interface s1-eth1, id 0
▸ Ethernet II, Src: OmronTat_00:01:01 (00:00:0a:00:01:01), Dst: CisTechn_00:01:01 (00:01:0a:00:01:01)
▸ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.3.2
▸ User Datagram Protocol, Src Port: 64980, Dst Port: 443
▾ Data (5 bytes)
    Data: 494e544831
0000  00 01 0a 00 01 01 00 00  0a 00 01 01 08 00 45 00   ········ ······E·
0010  00 21 00 01 00 00 40 11  62 c9 0a 00 01 01 0a 00   ·!····@· b·······
0020  03 02 fd d4 01 bb 00 0d  19 ab 49 4e 54 48 31      ········ ··INTH1
```

Figure 4.5: Wireshark capture of a frame leaving h1.

4.1.2 Packet forwarding

The L3 forwarding tables are pre-configured in the switches with a MAT using Longest Prefix Match (LPM). So, the MACs of the hosts h1, h2, h3 are pre-configured in each switch's MAT as e.g. for s2:

```
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.1/32 => 00:00:0a
    :00:01:01 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.2/32 => 00:00:0a
    :00:03:02 2
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.5.3/32 => 00:00:0a
    :00:05:03 3
```

The hosts h4 and h5 are not required to have routing. h4 is a protected host just for the collection, and h5 is not known by the network.

4.1.3 INT source

The INT source switch must identify the flows via its *watchlist*. When there is a match, the switch adds the INT header and its INT data accordingly. In this lab, the source switches are s1 and s5. The code below is the configuration of the switch s1, which defines the switch ID, the INT domain and the *watchlist*.

```
//set up ipv4_lpm table
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.1/32 => 00:00:0a
    :00:01:01 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.2/32 => 00:00:0a
    :00:03:02 2
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.5.3/32 => 00:00:0a
    :00:05:03 3
//set up switch ID
table_set_default process_int_transit.tb_int_insert init_metadata 1
//set up process_int_source_sink
table_add process_int_source_sink.tb_set_source int_set_source 1 =>
//matchlist h1 to h2, HTTP 80 Hex50
table_add process_int_source.tb_int_source int_source 10.0.1.1&&&0
    xFFFFFFFF 10.0.3.2&&&0xFFFFFFFF 0x00&&&0x00 0x00508&&&0xFFFF\
```

```
=> 11 10 0xF 0xF 10
//matchlist h1 to h2, HTTPS 443 Hex1BBB
table_add process_int_source.tb_int_source int_source 10.0.1.1&&&0
   xFFFFFFFF 10.0.3.2&&&0xFFFFFFFF 0x00&&&0x00 0x01BBB8&&&0xFFFF\
=> 11 10 0xF 0xF 10
//matchlist h1 to h2, PostGreSQL 5432 Hex1538
table_add process_int_source.tb_int_source int_source 10.0.1.1&&&0
   xFFFFFFFF 10.0.3.2&&&0xFFFFFFFF 0x00&&&0x00 0x1538&&&0xFFFF\
=> 11 10 0xF 0xF 10
```

The last line configures the:

- source-ip, source-port, destination-ip, destination-port defines 4-tuple flow which will be monitored using INT functionality;
- int-max-hops - how many INT nodes can add their INT node metadata to packets of this flow;
- int-hop-metadata-len - INT metadata words are added by a single INT node;
- int-hop-instruction-cnt - how many INT headers must be added by a single INT node;
- int-instruction-bitmap - instruction mask defining which information (INT headers types) must added to the packet;
- table-entry-priority - general priority of entry in match table (not related to INT);

The packet leaving s1 has now the s1 INT statistics, as captured at the exit of s1 interface in *Wireshark*, displayed in the Figure 4.6.

```
Frame 1: 107 bytes on wire (856 bits), 107 bytes captured (856 bits) on interface s1-eth2, id 0
Ethernet II, Src: CisTechn_00:01:01 (00:01:0a:00:01:01), Dst: OmronTat_00:03:02 (00:00:0a:00:03:02)
Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.3.2
User Datagram Protocol, Src Port: 53372, Dst Port: 443
Data (65 bytes)
0000  00 00 0a 00 03 02 00 01  0a 00 01 01 08 00 45 5c   ········ ······E\
0010  00 5d 00 01 00 00 3f 11  63 31 0a 00 01 01 0a 00   ·]····?· c1······
0020  03 02 d0 7c 01 bb 00 49  47 03 10 0e 00 00 20 00   ···|···I G····· ·
0030  0b 09 ff 00 00 00 00 00  00 00 00 00 00 01 00 01   ········ ········
0040  00 02 00 00 03 f8 00 00  00 00 00 00 00 04 71 f3   ········ ······q·
0050  37 ab 00 00 00 04 71 f3  3b a3 00 00 00 01 00 00   7·····q· ;·······
0060  00 02 00 00 00 00 49 4e  54 48 31                  ······IN TH1
```

Figure 4.6: *Wireshark* capture of a packet leaving s1 towards s2.

4.1.4 INT transit

The INT transit switch searches if there is a INT packet embedded in a packet, and then reads the instructions encoded in the INT header and adds its own INT data. Then forwards as specified by the LPM MAT. In this lab, the transit switches are s2 and s4. The code below is the configuration of switch s2, which specifies the LPM and the the switch ID.

```
//set up ipv4_lpm table
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.1/32 => \
00:00:0a:00:01:01 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.2/32 => \
00:00:0a:00:03:02 2
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.5.3/32 => \
```

```
00:00:0a:00:05:03 3
//set up switch ID
table_set_default process_int_transit.tb_int_insert init_metadata 2
```

The packet leaving s2 has now the s1 + s2 INT statistics, as captured at the exit of s2 interface in *Wireshark*, displayed in the Figure 4.7.

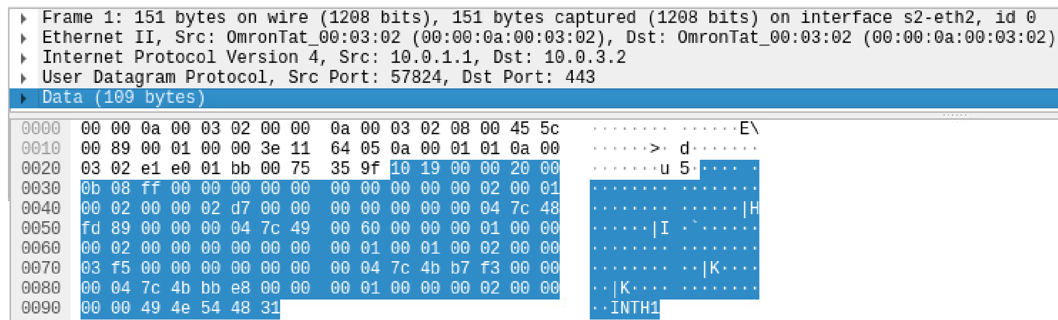

Figure 4.7: *Wireshark* capture of a packet leaving s2 towards s3.

4.1.5 INT sink

The INT sink switch detects the INT header in the packets and reads the instructions. Then adds its own INT data and creates a new packet as defined in the table below, towards the INT collector. This new packet is mirrored to the port where the INT collector is. Then extracts the INT data and restores the packet as originally sent towards the destination host. The code below is the configuration of the switch s3 which includes the following:

- mirrored port for the INT collector;
- LPM MAT;
- INT domain;
- source and destination, L2 addresses and L3 addresses, and L4 port;
- switch ID.

```
//creates a mirroring ID to output port specified
mirroring_add 500 2
//set up ipv4_lpm table
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.1/32 => \
00:00:0a:00:01:01 4
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.2/32 => \
00:00:0a:00:03:02 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.5.3/32 => \
00:00:0a:00:05:03 6
//set up process_int_source_sink
table_add process_int_source_sink.tb_set_sink int_set_sink 1 =>
//INT report setup towards the INT collector
table_add process_int_report.tb_generate_report
   do_report_encapsulation =>\
00:01:0a:00:03:07 00:00:0a:00:03:04 10.0.3.254 10.0.3.4 1234
//set up switch ID
table_set_default process_int_transit.tb_int_insert init_metadata 3
```

Now s3 adds its own statistics, so we have a new packet with s1 + s2 + s3 INT statistics, as captured at the exit of s3 interface towards h4 in Wireshark, displayed in the Figure 4.8.

```
▸ Frame 1: 252 bytes on wire (2016 bits), 252 bytes captured (2016 bits) on interface s3-eth2, id 0
▸ Ethernet II, Src: CisTechn_00:03:05 (00:01:0a:00:03:05), Dst: OmronTat_00:03:04 (00:00:0a:00:03:04)
▸ Internet Protocol Version 4, Src: 10.0.3.254, Dst: 10.0.3.4
▸ User Datagram Protocol, Src Port: 1234, Dst Port: 1234
▸ Data (210 bytes)

0000  00 00 0a 00 03 04 00 01  0a 00 03 05 08 00 45 00   ········ ······E·
0010  00 ee 00 00 00 00 40 11  00 00 0a 00 03 fe 0a 00   ······@· ········
0020  03 04 04 d2 04 d2 00 da  00 00 20 40 0c aa 00 00   ········ ·· @····
0030  00 03 13 00 00 20 00 00  00 00 00 00 00 00 00 00   ····· ·· ········
0040  0a 00 03 02 00 00 0a 00  03 02 08 00 45 5c 00 b5   ········ ····E\··
0050  00 01 00 00 3e 11 63 d9  0a 00 01 01 0a 00 03 02   ····>·c· ········
0060  fa c3 01 bb 00 a1 1c bc  10 24 00 00 20 00 0b 07   ········ ·$·· ···
0070  ff 00 00 00 00 00 00 00  00 00 00 03 00 04 00 01   ········ ········
0080  00 00 04 3a 00 00 00 00  00 00 00 04 85 ed ab a0   ···:···· ········
0090  00 00 00 04 85 ed af da  00 00 00 04 00 00 00 01   ········ ········
00a0  00 00 00 00 00 00 00 02  00 01 00 02 00 00 03 02   ········ ········
00b0  00 00 00 00 00 00 00 04  85 f0 da 33 00 00 00 04   ········ ···3····
00c0  85 f0 dd 35 00 00 00 01  00 00 00 02 00 00 00 00   ···5···· ········
00d0  00 00 00 01 00 01 00 02  00 00 04 68 00 00 00 00   ········ ···h····
00e0  00 00 00 04 85 f3 94 2b  00 00 00 04 85 f3 98 93   ·······+ ········
00f0  00 00 00 01 00 00 00 02  00 00 00 00               ········ ····
```

Figure 4.8: *Wireshark* capture of a packet leaving s3 towards h4.

4.1.6 Server listening

Finally, the simulated server in h2 is preferably listening to the data sent from h1 and h3, so we used *netcat* to listen to the pre-determined services:

```
while true; do nc -ul -p 80; done
while true; do nc -ul -p 443; done
while true; do nc -ul -p 5432; done
```

The packet leaving s3 to the server is stripped of the INT statistics, as captured at the exit of s3 interface towards h2 in Wireshark, displayed in the Figure 4.9. This packet has the same data as the initial packet sent from the host, but has now different L2 metadata.

```
▸ Frame 1: 47 bytes on wire (376 bits), 47 bytes captured (376 bits) on interface s3-eth1, id 0
▸ Ethernet II, Src: OmronTat_00:03:02 (00:00:0a:00:03:02), Dst: OmronTat_00:03:02 (00:00:0a:00:03:02)
▸ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.3.2
▸ User Datagram Protocol, Src Port: 58237, Dst Port: 443
▸ Data (5 bytes)

0000  00 00 0a 00 03 02 00 00  0a 00 03 02 08 00 45 00   ········ ······E·
0010  00 21 00 01 00 00 3d 11  65 c9 0a 00 01 01 0a 00   ·!····=· e·······
0020  03 02 e3 7d 01 bb 00 0d  34 02 49 4e 54 48 31      ···}···· 4·INTH1
```

Figure 4.9: Wireshark capture of packet leaving s3 towards h2.

4.2 Collection of INT statistics and visualization

There are some good examples of the visualization of INT statistics all based on InfluxDB for the database and Grafana for the visualization: [59], [61], and [103].

These examples demonstrate that the INT statistics can be exported to a service and fed to a InfluxDB. The DB is queried by the Grafana service and the data is available to build dashboards.

4.2.1 Collection

The collection of the INT data is achieved with a Python script [104] that listens to the data incoming to h4 and filters the packets with the predefined destination port and INT header. In this case these packets are predefined as UDP on port 1234, as per the example of the packet captured in *Wireshark* in Figure 4.8.

The Python script parses through the INT packet and extracts the collected information across the switches and appends to the InfluxDB database measurements:

- **Flow latency**: source IP, destination IP, source port, destination port, protocol, and the time when it was collected.
- **Switch latency**: switch ID, latency in its hop, and the time when it was collected.
- **Link latency**: egress switch ID, egress port ID, ingress switch ID, ingress port ID, latency, and the time when it was collected. The latency is calculated as the difference between the time of the egress and the time of ingress on each switch.
- **Queue latency**: switch ID, queue ID, occupancy of the flow, and the time when it was collected.

These measurements output the latency in micro-seconds.

The script also outputs to the screen as shown in Figure 4.10:

```
rpires@P4RP3:~/INT_v5/report_collector$ sudo python3 collector_influxdb.py
sniffing on s3-eth2
<influxdb.client.InfluxDBClient object at 0x7ff38917b820>
********* Receiving Telemetry Report ********
src_ip 10.0.1.1
dst_ip 10.0.3.2
src_port 55561
dst_port 5432
ip_proto 17
hop_cnt 3
flow_latency 2820
switch_ids [3, 2, 1]
l1_ingress_ports [4, 1, 1]
l1_egress_ports [1, 2, 2]
hop_latencies [1164, 673, 983]
queue_ids [0, 0, 0]
queue_occups [0, 0, 0]
ingress_tstamps [1414525256, 1414785488, 1415002607]
egress_tstamps [1414526420, 1414786161, 1415003590]
l2_ingress_ports [4, 1, 1]
l2_egress_ports [1, 2, 2]
egress_tx_utils [0, 0, 0]
```

Figure 4.10: INT packed decoded by the collector script.

These measurements are appended to a Influx database running on the host ma-

chine. We can see the measurements as in Figure 4.11.

```
rpires@P4RP3:~/INT_v5/report_collector$ influx
Connected to http://localhost:8086 version 1.6.4
InfluxDB shell version: 1.6.4
> show databases
name: databases
name
----
_internal
int
> use int
Using database int
> show measurements
name: measurements
name
----
flow_latency
link_latency
queue_occupancy
switch_latency
```

Figure 4.11: InfluxDB client, displaying INT measurements.

As a pre-requisite, InfluxDB must be installed and the INT database created before starting the collector.

4.2.2 Visualization

The visualization of the INT packets in Grafana offers quick insights of the behavior of the network. As shown in the screenshot in Figure 4.12, we can:

- display the link latency of the flows from h1 or from h3;
- display the flow mean flow latency;
- display the flow latency per service. In this case the HTTP, HTTPS or PostgreSQL;
- display the same flow latency per source host. In this case h1->h2 or h3->h2;
- display the switch latency overall and per switch;

As an example, when flooding the server with traffic from one host, we could easily identify the cause. In the case highlighted in Figure 4.13, the traffic came from h3 and the switch with high latency is the s5, INT source, in this flow.

4.3 Attacking the INT platform

The INT statistics can be an important security asset as the data may be used by the network administrators for assessing the network and troubleshooting any issues. So, it is a valuable target for a malicious adversary.

We consider in this scenario that an adversary is controlling a rogue host in the network next to the collector, as described in the Figure 4.14.

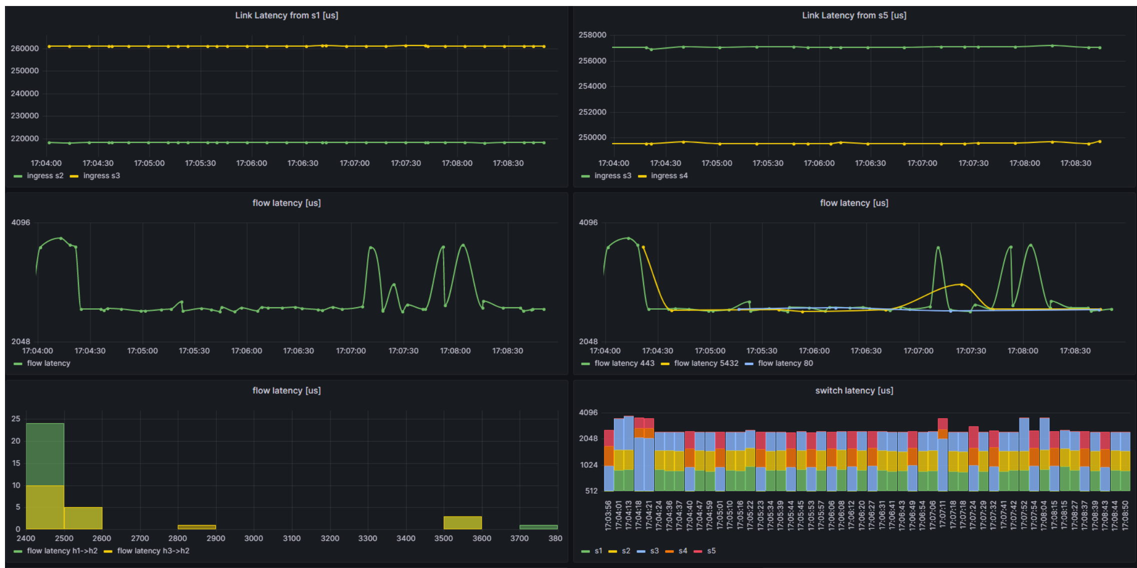

Figure 4.12: Grafana view of INT statistics.

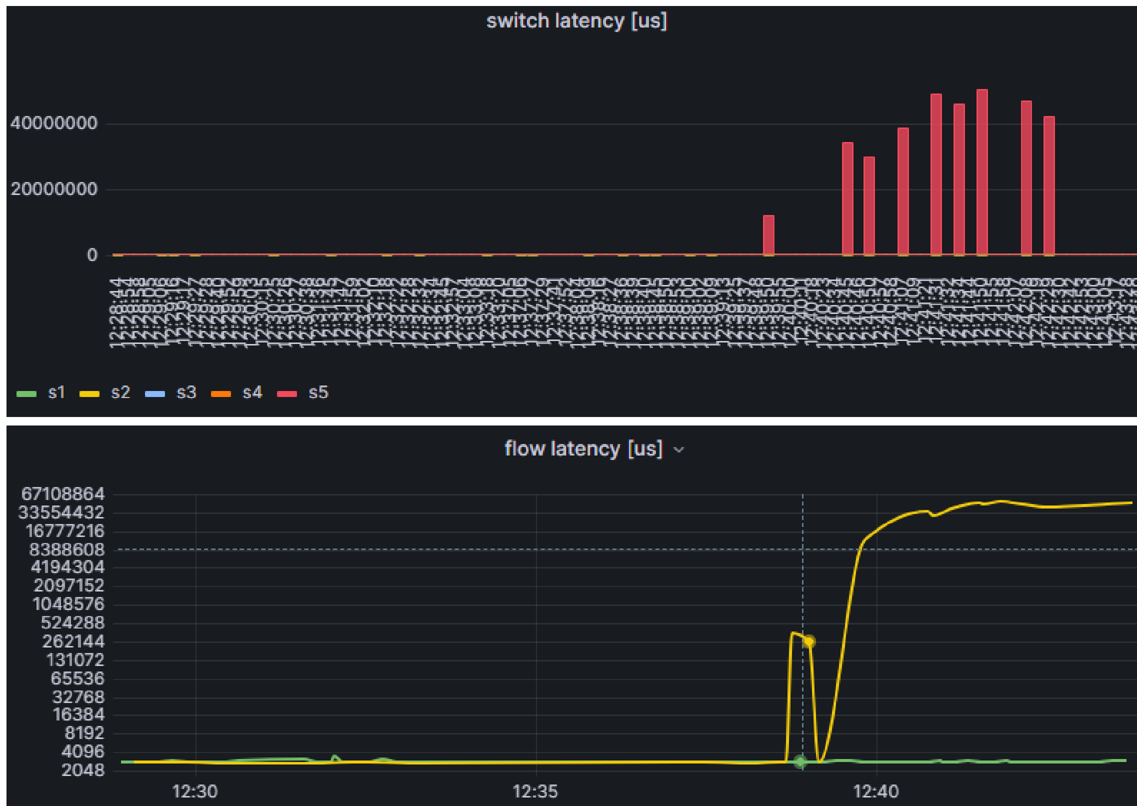

Figure 4.13: INT statistics under high load.

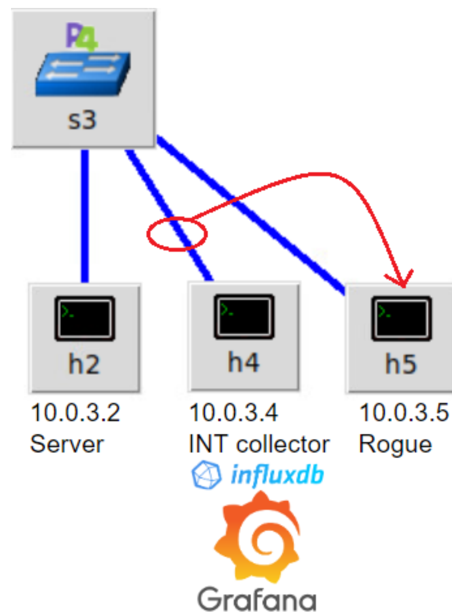

Figure 4.14: The server network with the collector and the rogue host eavesdropping on the INT reports.

There are several possible attacks that we tested such as:

- INT eavesdropping;
- INT replay;
- INT manipulation.

4.3.1 INT eavesdropping

As defined in the MITRE ATT&CK framework [105], an attack starts with the reconnaissance phase.

In this scenario, the adversary tries to listen to the traffic using tools like *ettercap* [106]. We used *ettercap* to do the ARP poisoning which misleads both the switch and the host ARP table. This is a MITM attack, that starts with eavesdropping:

```
ettercap -Ti h5-eth0 -M arp:oneway //10.0.3.254/ //10.0.3.4/
```

As in this current P4 code the ARP tables are static, the s3 and h4 ARP tables can't be poisoned. In the Figure 4.15 we illustrate the initial h4 ARP table and that after each poisoning message from h5, s3 replies with a gratuitous ARP message.

We chose to specify the static ARP because if there are no answers to the initial ARP spoofing, then *ettercap* fails with the message:

```
FATAL: ARP poisoning needs a non empty hosts list.
```

We also tried the attack with port stealing, and confirmed the INT platform identified such attack. The port stealing technique is useful to sniff in a switched environment when ARP poisoning is not effective (for example where static mapped ARPs are used). It floods the LAN with ARP packets and as such there are impacts in the network as evidenced in the Figure 4.16.

```
"Node: h4"
root@P4RP3:/home/rpires/intv8# arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.3.254               ether   00:01:0a:00:03:05   CM                    h4-eth0
10.0.3.2                 ether   00:00:0a:00:03:02   CM                    h4-eth0
10.0.3.5                 ether   00:00:0a:00:03:05   CM                    h4-eth0
```

h5 to s3 attempt poisoning

```
00:00:0a:00:03:05    00:01:0a:00:03:05    ARP     42 10.0.3.4 is at 00:00:0a:00:03:05
00:01:0a:00:03:05    00:00:0a:00:03:05    ARP     42 Gratuitous ARP for 10.0.3.254 (Reply)
```

s3 to h5 gratuitous ARP reply

Figure 4.15: Failed ARP poisoning attempt.

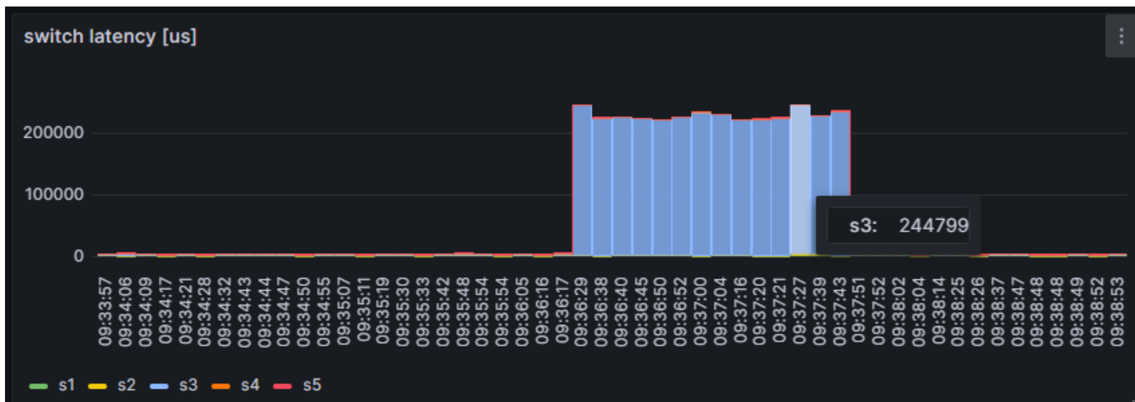

Figure 4.16: INT detecting high load in s3, due to *ettercap* attack.

These ARP poisoning attacks may be successful if the collector is not directly connected to the P4 switch. In this scenario, the INT packets are created by the INT sink switch with the destination MAC and IP.

So, we considered a slightly different scenario, in which we have a Security Operations Center. The SOC is a centralized solution within an organization that continuously monitor and helps to prevent, detect, analyze, and respond to cybersecurity incidents. As such, there would be some more network devices between the INT sink and the collector. So, we tested a new scenario with a new P4 switch between s3 and the hosts, as described in Figure 4.17. This new P4 switch is not part of the INT domain and was programmed as a simple L2 switch.

As a proof of concept, we additionally considered that an advanced attacker could have tweaked the static rules to add a mirroring port towards the rogue host, h5:

```
table_add MyIngress.forwarding MyIngress.forward 3 => 1
table_add MyIngress.forwarding MyIngress.forward 1 => 2
table_add MyIngress.forwarding MyIngress.forward 2 => 1
//table_add MyIngress.forwarding MyIngress.forward 3 => 2

//creates a mirroring ID 1 to output port specified
mirroring_add 1 2
```

Now h5 was able to eavesdrop as evidenced in Figure 4.18.

We will use this captured data for a replay attack.

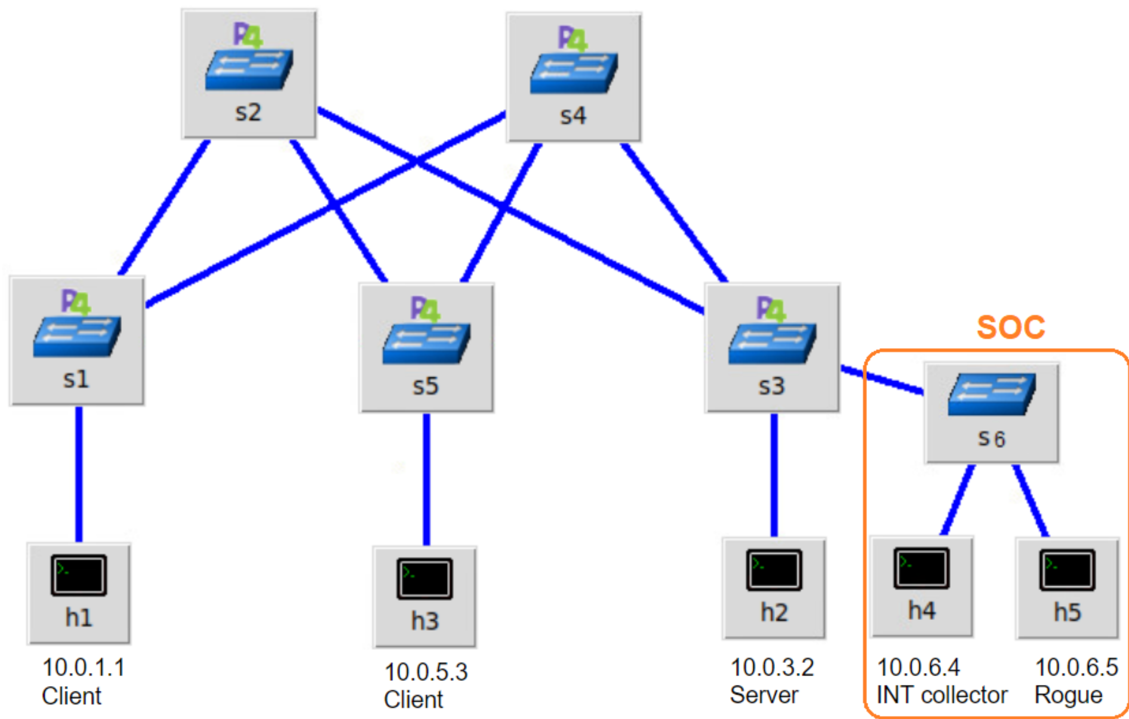

Figure 4.17: Spine-Leaf topology with s6 as a P4 L2-switch within a SOC.

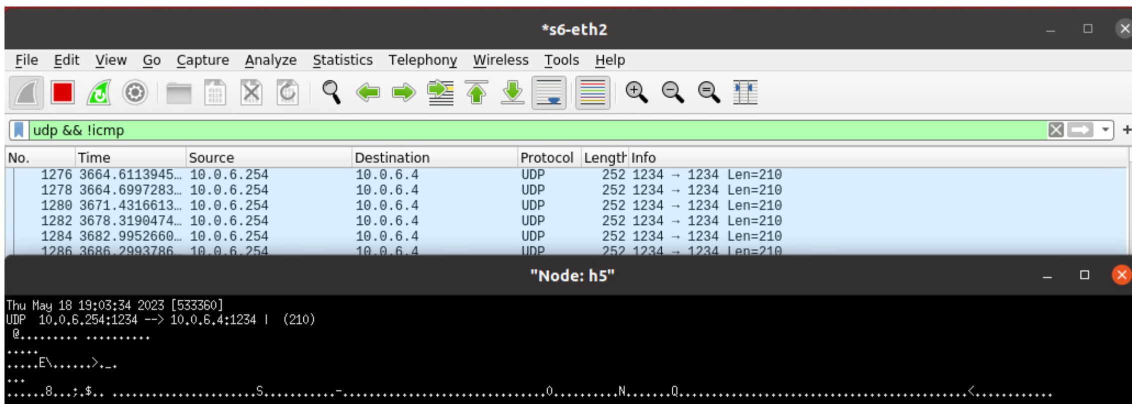

Figure 4.18: h5 eavesdropping INT sent to h4.

4.3.2 INT replay

An attacker could do a replay attack by sending fake data towards the INT collector:

1. collect a previous INT message or craft INT stats;
2. spoof the IP source as if it would be the s3 gateway;
3. send towards the collector;

In this case we have used the previously captured INT message and included into a Python script with *scapy* as the payload. This script was adapted from [62].

This replay simulated a flow coming from h1 to h2 towards the HTTP port, hence the attacker could use it to simulate a normal working status and thus hide other attacks. In the Figure 4.19 we can identify the fake statistics for h1 on port 80.

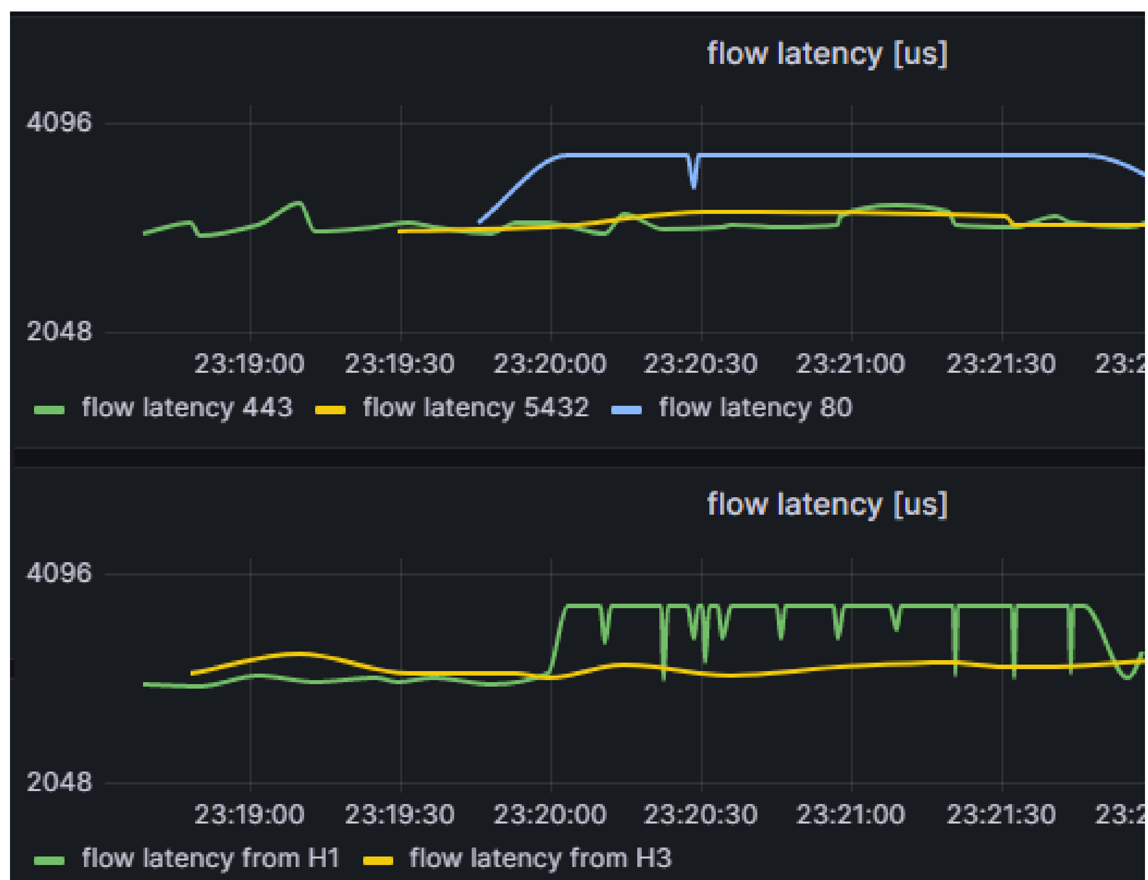

Figure 4.19: INT replay attack, h1 port 80.

4.3.3 INT manipulation

With *ettercap* and *etterfilter*, we could also change the traffic in transit, however it was not possible due to the issue already identified in 4.3.1: static ARP tables do not allow ARP poisoning. The architecture referred in the Figure 4.17 also do not allow the classical MITM manipulation because the switching is static and the

ARP tables of h4 and s6 are not changing either.

The next step is to try to use the same P4 code in a real environment with P4PI as described in chapter 5.

Chapter 5

Implementation scenario

This is the final scenario built after evaluating the test scenarios described in Chapter 4. We built this lab with P4Pi, as initially described in section 3.1.4.

All code and details are documented in our P4INT for P4Pi GitHub repository [107].

5.1 Topology

The scenario is similar to the scenario in the Figure 4.3:

- a P4-enabled network in which a client machine is at an unprotected network;
- the server is at a protected network, reachable through three P4 switches;
- the P4 network generates INT statistics of the flows between the host and the server;
- the INT collector is connected to the INT sink;
- the INT collector host that also runs a Influxdb database and a Grafana system for enhanced visualization;
- a rogue host, controlled by an adversary, is next to the collector;
- the rogue host attacks the INT solution with a MITM attack by eavesdropping or manipulating the INT reports.

For the sake of simplicity and due to the shortage of Raspberry PIs, we streamlined the topology to three P4 switches. The WiFi interfaces are configured as APs, so we may regard those as L2 switches. We used 2 laptops with several WiFi interfaces to simulate the hosts: h1 (client), h2, hc (collector), ha (attacker) and hs (server). We used the following Raspberry Pis as described in the Figure 5.1:

1. Raspberry Pi 3 Model B, Rev 1.2, 1GB of RAM. This device will act as INT source.
2. Raspberry Pi 4 Model B, Rev 1.5, 8GB of RAM. This device will act as INT transit.
3. Raspberry Pi 4 Model B, Rev 1.1, 4GB of RAM. This device will act as INT sink.

As the Raspberry Pi only has one on-board Ethernet and one WiFi interface, we

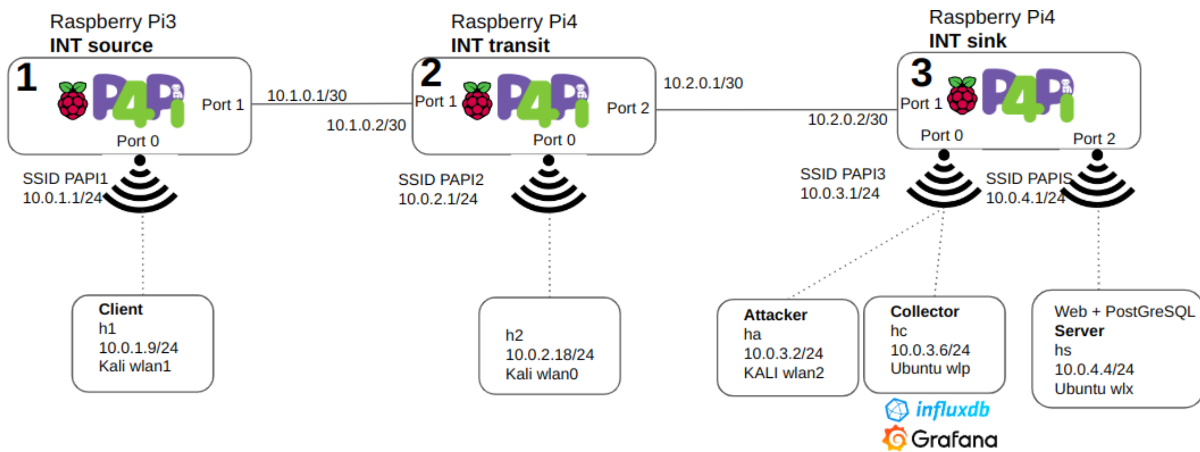

Figure 5.1: Diagram of the INT solution with three P4Pis.

added the necessary USB devices as in the picture of the Figure 5.2.

5.2 Network deployment

We deployed the P4Pi network as described in the Appendix A:

- P4Pi installed in the SD cards;
- configured the interfaces, static IP, DHCP service, and WiFi;
- copied the P4 code, as tested in the Section 4.1;
- configured the BMv2 service and started;
- uploaded the static tables to the running BMv2 switches.

As an option to using the BMv2 service, it is also possible to run in manual mode or debugging mode as described in the Appendix A.3 and A.4.

5.3 Packet source

The INT packets are only generated if a specific packet matches the watchlist. So, we used the *scapy* library within a Python script to craft the packets. This is a simple Python script that takes as input parameters the destination IP, the L4 protocol UDP/TCP, the destination port number, an optional message and the number of packets sent.

Additionally, we included a command to simulate recurrent accesses to the server, every few seconds access to HTTP, HTTPS, and PostgreSQL from h1:

```
watch -n 15 python3 sendwlan1.py --ip 10.0.4.4 -l4 udp --port 80 --
    m INTH1 &
watch -n 25 python3 sendwlan1.py --ip 10.0.4.4 -l4 udp --port 443
    --m INTH1 &
watch -n 35 python3 sendwlan1.py --ip 10.0.4.4 -l4 udp --port 5432
    --m INTH1 &
```

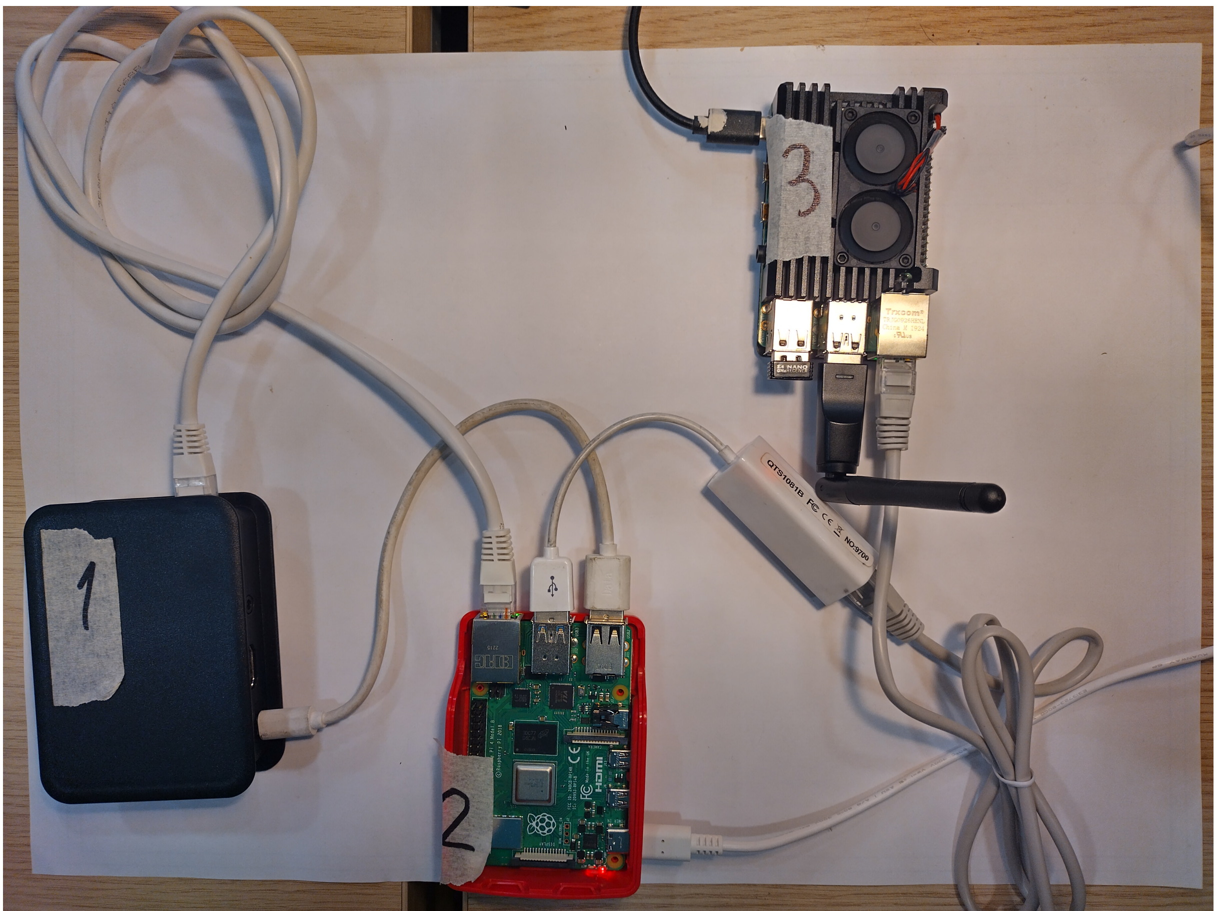

Figure 5.2: INT lab with three P4Pis.

These packets are only carrying the content "INTH1" which is confirmed in the captured data with tcpdump at the ingress of s1:

```
pi@p4pi1:~$ sudo tcpdump -e -X -i br0 udp
38:a2:8c:90:60:6c (oui Unknown) > 3e:90:87:5f:dc:af (oui Unknown),
ethertype IPv4 (0x0800), length 47: kali.p4pi1.65116 > 10.0.4.4.
    https:
UDP, length 5
0x0000:  4500 0021 0001 0000 4011 61bf 0a00 0109  E..!....@.a.....
0x0010:  0a00 0404 fe5c 01bb 000d 1819 494e 5448  .....\......INTH
0x0020:  31
```

5.4 Packet forwarding

We initially made sure there was connectivity between all devices, so we setup static routes for the networks in the Raspberry Pis. These can be deleted as soon as the BMv2 switch is running and well configured.

The L3 forwarding tables are pre-established in the switches with MAT using Longest Prefix Match (LPM). So, the networks and hosts h1, h2 and hs are pre-registered in each switch's MAT:

```
#s1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.0/24=>d8:3a:dd
    :11:7a:de 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.4.0/24=>d8:3a:dd
    :11:7a:de 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.2.0/24=>d8:3a:dd
    :11:7a:de 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.9/32=>38:a2:8c
    :90:60:6c 0
#s2
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.0/24=>b8:27:eb
    :83:45:95 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.0/24=>dc:a6
    :32:40:1b:03 2
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.4.0/24=>dc:a6
    :32:40:1b:03 2
table_add l3_forward.ipv4_lpm ipv4_forward
    10.0.2.18/32=>60:67:20:87:81:4c 0
#s3
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.0/24=>00:e0:4c
    :53:44:58 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.2.0/24=>00:e0:4c
    :53:44:58 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.6/32=>e4:a4:71:cd
    :52:99 0
```

The hosts hc and ha are not required to have routing defined in the switches. The collector, hc, is planned to be secured and as such no traffic should be sent to it from the unprotected network. The attacker host, ha, is not known to the network.

5.5 INT source

The INT source switch must identify the flows via its watchlist. When there is a match, the switch adds the INT header and its INT data accordingly. In this lab, the INT source switch is s1 so the code below is the content to be uploaded the switch tables:

```
#L3 forwarding
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.0/24=>d8:3a:dd
    :11:7a:de 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.4.0/24=>d8:3a:dd
    :11:7a:de 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.2.0/24=>d8:3a:dd
    :11:7a:de 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.9/32=>38:a2:8c
    :90:60:6c 0
#set up process_int_source_sink
table_add process_int_source_sink.tb_set_source int_set_source 1 =>
#set up switch ID
table_set_default process_int_transit.tb_int_insert init_metadata 1
#ARP
table_add arpreply.arp_exact arp_reply 10.1.0.1 => b8:27:eb
    :83:45:95
table_add arpreply.arp_exact arp_reply 10.0.1.1 => 3e:90:87:5f:dc:
    af
table_add arpreply.arp_exact arp_reply 10.0.1.9 => 38:a2:8c:90:60:6
    c
#port PostGreSQL 5432
table_add process_int_source.tb_int_source int_source 10.0.1.9&&&0
    xFFFFFF00 10.0.4.4&&&0xFFFFFFFF 0x00&&&0x00 0x1538&&&0xFFFF =>
    11 10 0xF 0xF 10
#port HTTPS 443
table_add process_int_source.tb_int_source int_source 10.0.1.9&&&0
    xFFFFFF00 10.0.4.4&&&0xFFFFFFFF 0x00&&&0x00 0x01BB&&&0xFFFF =>
    11 10 0xF 0xF 10
#port HTTP 80
table_add process_int_source.tb_int_source int_source 10.0.1.9&&&0
    xFFFFFF00 10.0.4.4&&&0xFFFFFFFF 0x00&&&0x00 0x0050&&&0xFFFF =>
    11 10 0xF 0xF 10
#any port
table_add process_int_source.tb_int_source int_source 10.0.1.9&&&0
    xFFFFFF00 10.0.4.4&&&0xFFFFFF00 0x00&&&0x00 0x00&&&0x00 => 11 10
     0xF 0xF 10
```

The packet leaving s1 has now the s1 INT statistics, as captured at the exit of s1:

```
pi@p4pi1:~$ sudo tcpdump -e -X -i enxb827eb834595 udp
d8:3a:dd:11:7a:de (oui Unknown) > d8:3a:dd:11:7a:de (oui Unknown),
ethertype IPv4 (0x0800), length 107: kali.p4pi1.58928 >
10.0.4.4.https: UDP, length 65
0x0000:  455c 005d 0001 0000 3e11 6327 0a00 0109  E\.]....>.c'....
0x0010:  0a00 0404 e630 01bb 0049 3045 100e 0000  .....0...I0E....
0x0020:  2000 0b09 ff00 0000 0000 0000 0000 0001  ................
0x0030:  0001 0001 0000 0a3f 0000 0000 0000 000a  .......?........
0x0040:  eded 8034 0000 000a eded 8a73 0000 0001  ...4.......s....
0x0050:  0000 0001 0000 0000 494e 5448 31         ........INTH1
```

Detect unauthorized flows

We chose to configure the masks in a way to create data when there are unauthorized flows. As the authorized flows are the ones from the host client to the server to the ports 80, 443 and 5432, we added the last line to capture unauthorized ones. This solution may overload the network, so it must be weighted carefully by the network administrator.

5.6 INT transit

The INT transit switch identifies that there is a INT packet embedded within the packet, so reads the instructions encoded in the INT header and adds its own INT data. Then forwards as specified by the LPM MAT. In this lab, the transit switch is s2 so the code below is the content to be uploaded the switch tables:

```
#L3 forwarding
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.0/24=>b8:27:eb
   :83:45:95 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.0/24=>dc:a6
   :32:40:1b:03 2
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.4.0/24=>dc:a6
   :32:40:1b:03 2
table_add l3_forward.ipv4_lpm ipv4_forward
   10.0.2.18/32=>60:67:20:87:81:4c 0
#set up switch ID
table_set_default process_int_transit.tb_int_insert init_metadata 2
#ARP
table_add arpreply.arp_exact arp_reply 10.0.2.1 => 8e:8a:0a:2c:17:
   de
table_add arpreply.arp_exact arp_reply 10.0.2.18 =>
   60:67:20:87:81:4c
table_add arpreply.arp_exact arp_reply 10.1.0.2 => d8:3a:dd:11:7a:
   de
table_add arpreply.arp_exact arp_reply 10.2.0.1 => 00:e0:4c
   :53:44:58
```

The packet leaving s2 has now the s1 + s2 INT statistics, as captured at the exit of s2:

```
pi@p4pi2:~$ sudo tcpdump -e -X -i enx00e04c534458 udp
dc:a6:32:40:1b:03 (oui Unknown) > dc:a6:32:40:1b:03 (oui Unknown),
ethertype IPv4 (0x0800), length 151:
10.0.1.9.56972 > 10.0.4.4.5432: UDP, length 109
x0000:  455c 0089 0001 0000 3c11 64fb 0a00 0109  E\......<.d.....
0x0010:  0a00 0404 de8c 1538 0075 246c 1019 0000  .......8.u$l....
0x0020:  2000 0b08 ff00 0000 0000 0000 0000 0002  ................
0x0030:  0002 0002 0000 1354 0000 0000 0000 000b  .......T........
0x0040:  a1e3 8201 0000 000b a1e3 9555 0000 0002  ...........U....
0x0050:  0000 0002 0000 0000 0000 0001 0001 0001  ................
0x0060:  0000 0a7c 0000 0000 0000 000b 2d74 0ebf  ...|........-t..
0x0070:  0000 000b 2d74 193b 0000 0001 0000 0001  ....-t.;........
0x0080:  0000 0000 494e 5448 31                   ....INTH1
```

5.7 INT sink

The INT sink switch detects the INT header in the packets and reads the instructions. Then adds its own INT data and creates a new packet as defined in the table below, towards the INT collector. This new packet is mirrored to the port 0 towards the INT collector. Then extracts the INT data and restores the packet as it was originally and sends to the server.

The code below is the configuration of the switch s3:

```
#creates a mirroring ID to output port specified
mirroring_add 500 0
#L3 forwading
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.1.0/24=>00:e0:4c
    :53:44:58 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.2.0/24=>00:e0:4c
    :53:44:58 1
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.3.6/32=>e4:a4:71:cd
    :52:99 0
#set up process_int_source_sink
table_add process_int_source_sink.tb_set_sink int_set_sink 0 =>
#INT report setup towards the INT collector PAPI3 veth0 to INTC in
    ubuntu
table_add process_int_report.tb_generate_report
    do_report_encapsulation => \
9a:4b:89:ad:8a:59 e4:a4:71:cd:52:99 10.0.3.1 10.0.3.6 1234
#set up switch ID
table_set_default process_int_transit.tb_int_insert init_metadata 3
#ARP
table_add arpreply.arp_exact arp_reply 10.0.3.1 => 9a:4b:89:ad:8a
    :59
table_add arpreply.arp_exact arp_reply 10.0.3.6 => e4:a4:71:cd
    :52:99
table_add arpreply.arp_exact arp_reply 10.1.0.3 => dc:a6:32:40:1b
    :03
```

Now s3 adds its own statistics, so we get a new package with s1 + s2 + s3 INT statistics. This is confirmed in the *tcpdump* capture at the exit of s3 interface towards hc:

```
pi@p4pi3:~$  sudo tcpdump -e -X -i br0 udp
9a:4b:89:ad:8a:59 (oui Unknown) > e4:a4:71:cd:52:99 (oui Unknown),
ethertype IPv4 (0x0800), length 252: 10.0.3.1.1234 > 10.0.3.6.1234:
UDP, length 210
0x0000:  4500 00ee 0000 0000 4011 0000 0a00 0301  E.......@.......
0x0010:  0a00 0306 04d2 04d2 00da 0000 2041 416d  .............AAm
0x0020:  0000 0003 1300 0020 0000 0000 0000 0000  ................
0x0030:  dca6 3240 1b03 dca6 3240 1b03 0800 455c  ..2@....2@....E\
0x0040:  00b5 0001 0000 3c11 64cf 0a00 0109 0a00  ......<.d.......
0x0050:  0404 e023 1538 00a1 22d5 1024 0000 2000  ...#.8.."..$....
0x0060:  0b07 ff00 0000 0000 0000 0000 0003 0001  ................
0x0070:  0000 0000 1e07 0000 0000 0000 0013 5aba  ..............Z.
0x0080:  ae25 0000 0013 5aba cc2c 0000 0001 0000  .%....Z..,......
0x0090:  0000 0000 0000 0000 0002 0002 0002 0000  ................
0x00a0:  16cb 0000 0000 0000 000b d958 2be8 0000  ...........X+...
0x00b0:  000b d958 42b3 0000 0002 0000 0002 0000  ...XB...........
0x00c0:  0000 0000 0001 0001 0001 0000 0a54 0000  .............T..
```

```
0x00d0:  0000 0000 000b 64e8 82a8 0000 000b 64e8  ......d.......d.
0x00e0:  8cfc 0000 0001 0000 0001 0000 0000       ..............
```

The packet sent to the server is stripped from INT data and its payload is back to the original sent:

```
pi@p4pi3:~$  sudo tcpdump -e -X -i wlx38a28c80c2ee udp
38:a2:8c:80:c2:ee (oui Unknown) > 34:60:f9:c9:ee:84 (oui Unknown),
ethertype IPv4 (0x0800), length 47: 10.0.1.9.61786 > 10.0.4.4.5432:
UDP, length 5
0x0000:  4500 0021 0001 0000 3c11 65bf 0a00 0109  E..!....<.e.....
0x0010:  0a00 0404 f15a 1538 000d 119e 494e 5448  .....Z.8....INTH
0x0020:  31                                       1
```

5.8 Wireshark INT P4 dissector

The INT packets can be analyzed in Wireshark, but it is helpful to have an appropriate decoder for these special packets. This decoder is called a dissector and needs to be built specifically for each implementation, as shown in Figures 5.3 and 5.4. We built these dissectors and made them available in our GitHub repository [101].

We built a dissector for the first hop, as shown in Figure 5.3. In this dissector we can identify the original payload as well as the INT data that is sent to the controller. The INT data uses 44B of data. We can confirm some of the data in the headers, as defined in the INT specifications [5]:

- INT Shim Header for UDP, 4B, INT type is INT-MD (1);
- INT MD Header, 12B, Type. INT version is INT-MD (2);
- INT metadata, 44B, with the swithc ID, ports, latency, etc;
- original payload, at the end of the packet.

The dissector for the report sent to the controller shows the INT data of the three devices in the path: INT source, INT transit, and INT sink. As highlighted in Figure 5.4, we can identify the flow as well the INT metadata of each hop:

- Telemetry data including the flow identification: INT-MD type, source and destination addresses and ports, etc.
- each hop uses 44B of data. This is the INT metadata referred in the INT-MD packet.

5.9 Collection of INT statistics and visualization

As described in 4.2, both InfluxDB and Grafana need to be installed and configured.

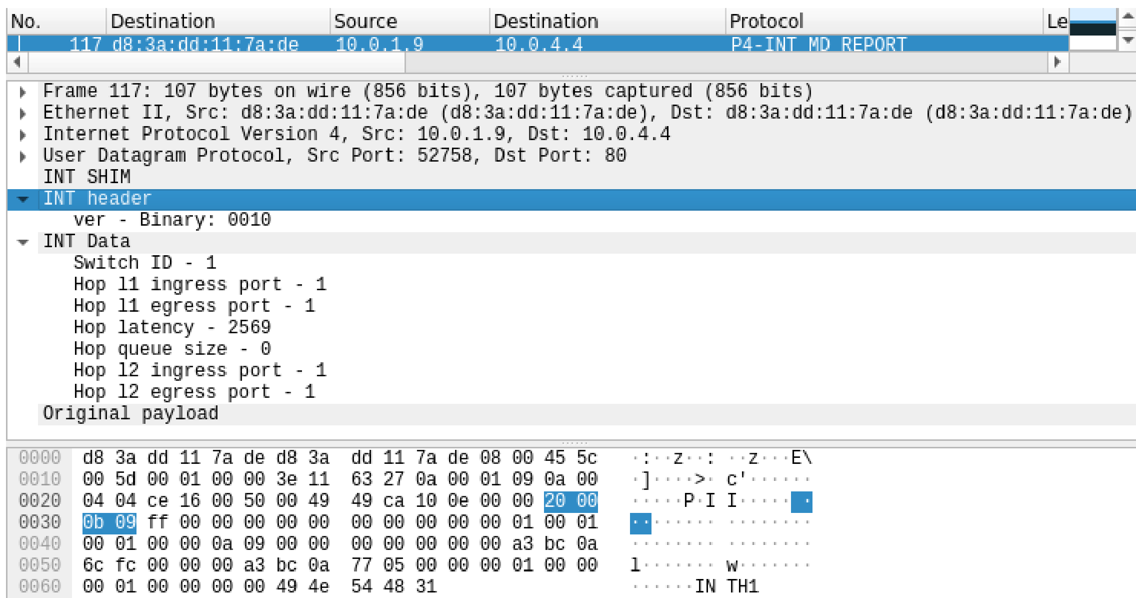

Figure 5.3: *Wireshark* view with a dissector of the INT-MD packet sent from the INT source to the INT transit.

5.9.1 Install and configure InfluxDB

Install InfluxDB with the influx online instructions, configure the service and install InfluxDB Python libraries:

```
sudo apt-get update && sudo apt-get install influxdb
sudo systemctl unmask influxdb.service
sudo systemctl start influxdb
sudo pip3 install influxdb
```

Create the int database:

```
influx
Connected to http://localhost:8086 version 1.8.10
InfluxDB shell version: 1.8.10
> show databases
name: databases
name
----
_internal
> create database int with duration 2h
> use int
Using database int
> show measurements
```

No measurements are there yet. These will be created when the data is uploaded.

5.9.2 Collection to InfluxDB

Similarly to what was implemented in 4.2.1, the script was configured to listen to the hc interface, in this case the Ubuntu machine in the interface wlp1s0. We got the real-world statistics:

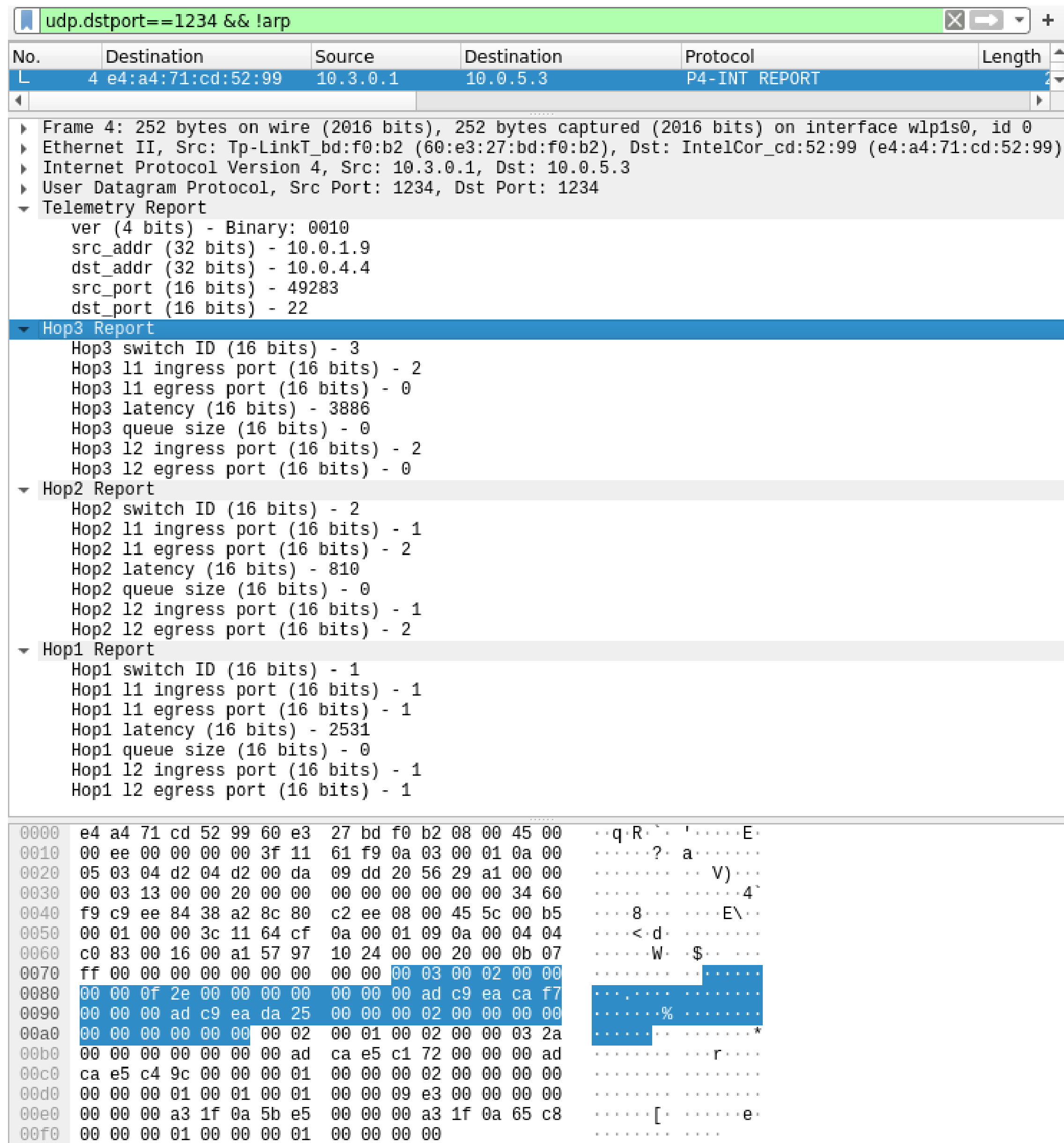

Figure 5.4: *Wireshark* view with a dissector of the INT report packet sent from the INT sink to the collector.

```
pi@p4pi3$ sudo python3 collector_influxdb.py
src_ip 10.0.1.9
dst_ip 10.0.4.4
src_port 54782
dst_port 443
ip_proto 17
hop_cnt 3
flow_latency 16530
switch_ids [3, 2, 1]
l1_ingress_ports [2, 2, 1]
l1_egress_ports [0, 2, 1]
hop_latencies [11094, 2835, 2601]
queue_ids [0, 0, 0]
queue_occups [0, 0, 0]
ingress_tstamps [915441488, 52685864513, 50732366655]
egress_tstamps [915452582, 52685867348, 50732369256]
l2_ingress_ports [2, 2, 1]
l2_egress_ports [0, 2, 1]
egress_tx_utils [0, 0, 0]
```

These measurements are appended to a InfluxDB database running on the host machine. We can now connect to InfluxDB and check if these were uploaded.

```
~$ influx
Connected to http://localhost:8086 version 1.8.10
InfluxDB shell version: 1.8.10
> use int
Using database int
> show measurements
name: measurements
name
----
flow_latency
link_latency
queue_occupancy
switch_altency
> select * from flow_latency
name: flow_latency
time         dst_ip   dst_port protocol src_ip   src_port value
----         ------   -------- -------- ------   -------- -----
16833879867 10.0.4.4   80       17      10.0.1.9 57347    3666
```

You may also check the logs with:

```
sudo journalctl -u influxdb.service | grep ‘‘POST /write’’
```

5.9.3 Install and configure Grafana

Installewith the guide Grafana from apt [108], then add the InfluxDB datasource in the Grafana web interface, default localhost:3000/:

1. Go to Configuration > Data sources, select InfluxDB;
2. Select the database int;
3. Test and if all is ok, you get the message "datasource is working. 4 measurements found".

Import the dashboard in the Grafana web interface. Go to Home > Dashboards > Import dashboard and upload the Grafana dashboard *json* from our P4INT for P4Pi GitHub repository [107]. This is optional, as you can build your own dashboard.

Note: make sure the collector is synchronized with an NTP or rather manually sync with the date command.

5.9.4 Visualization in Grafana

The visualization of the INT packets in Grafana offers quick insights of the behavior of the network. Additionally to the examples we provided in Section 4.2.2, we now have data from a real environment.

We can immediately confirm that there is more latency in the sink switch than in the transit switch as shown in Figure 5.5.

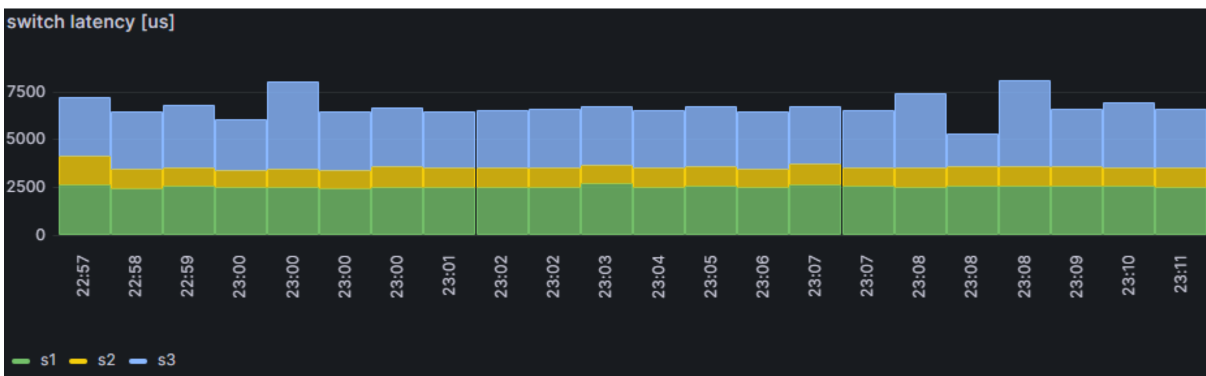

Figure 5.5: Switch latency in P4PI.

Chapter 6

Results

Following the final implementation described in Chapter 5, we can present some results.

As defined in the NIST Cyber Security Framework [3], the five functions included in the Framework are: Identify, Protect, Detect, Respond, and Recover. In this simulated environment we may consider the following:

- **Identify:** there is only one server and one client, and only three approved flows.
- **Protect:** the other traffic flows are not authorized.
- **Detect:** the INT solution can help to identify an attack.
- **Respond:** the INT solution can help other tools to respond to an attack.
- **Recover:** the server, the network and the INT solution shall recover after an attack.

Thus an INT platform may be used for the detection and respond functions.

INT may be used as a security control but can also be attacked, thus we analyzed the following use-cases:

1. **Detection of attacks:** how can INT detect an attack, and what kind of attacks can it detect?
2. **Attacking INT:** eavesdropping, replay or manipulation.
3. **Defending INT:** how can we improve the reliability of the platform?

6.1 Attack detection

This INT platform simulates an use case in which a server at a secure network is accessed from a client. INT is deployed to collect statistics that may help to protect the server from attacks coming from the client.

INT collects statistics of flows from the Port 0 of the INT source to the Port 2 of the INT sink, as described in the Figure 6.1.

The Grafana dashboard helps to detect some attacks which could even be used to

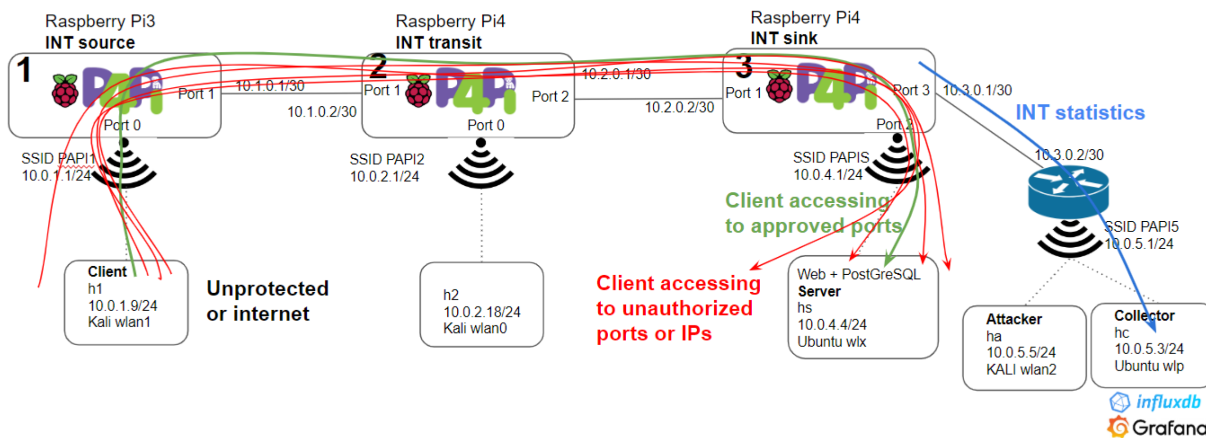

Figure 6.1: Possible attack flows to the server and the flow of INT statistics to the collector.

create attack signatures. We already discussed that if an attack causes high load to any P4 switch, then this will be detected as referred in Figure 4.16.

6.1.1 Unauthorized flows

As discussed in Section 5.5, if the administrator already knows what are the authorized flows then the unauthorized flows can be identified in the dashboard. As highlighted in the Figure 6.2, we can identify three types of unauthorized flows:

1. Unauthorized destination ports: only the L4 ports 80, 443 or 5432 should be reached. Anything else may point to an attack, e.g., attempts to connect to a stream of ports suggest it is a port scan.
2. Unauthorized destination IP ports: only the server should be reached. These attempts suggest some IP scan of the network.
3. Unauthorized source IPs: only the client should reach the server. This access suggest some rogue client is trying to reach out the server.

We created a dashboard to highlight unauthorized flows and then crafted some flows to get it populated. In Figure 6.2 we display in the bar graph the time and quantity of each type of unauthorized flows. Below the graph, we have a table providing more details per each flow.

6.1.2 Attack visualization

We can use the Grafana dashboard to help to detect some further attacks.

We added a report in the Grafana dashboard that counts the number of unauthorized attempts to a port not included in the approved services, as explained in the Section 5.5.

Considering the setup in Figure 6.1, if a malicious actor does a *nmap* port scan from the client, it can be detected as evidenced in Figure 6.3. This scan also filled

Time	dst_ip	dst_port	protocol	src_ip
2023-07-05 23:38:45	10.0.4.4	22	17	10.0.1.9
2023-07-05 23:38:45	10.0.4.4	22	17	10.0.1.9
2023-07-05 23:38:45	10.0.4.3	80	17	10.0.1.9
2023-07-05 23:38:45	10.0.4.3	80	17	10.0.1.8
2023-07-05 23:38:29	10.0.4.3	80	17	10.0.1.8

Figure 6.2: Grafana dashboard evidencing unauthorized flows.

the switch forwarding queues due to the high scan rate, as highlighted in the report of the queue occupancy table. The collector script often crashed under heavy-load *nmap* scanning, so having as a side effect a DoS.

6.2 Attacking INT

The INT statistics can be an important security asset as the data may be used by the network administrators for assessing the network and troubleshooting any issues. So, it is a valuable target for a malicious adversary.

We consider in this scenario, described in Figure 6.4, that an adversary is controlling a rogue host, *ha*. There are several possible attacks that we will try against INT such as eavesdropping, replay and manipulation, as described in the next sub chapters.

6.2.1 INT eavesdropping

As defined in the *MITRE ATT&CK* framework [105], the attacks start with the **reconnaissance phase**. In the reconnaissance phase, the adversary is trying to actively or passively gather information they can use to plan future actions [109]. Such information may include details of the victim infrastructure to be later leveraged by the adversary to aid in other phases of the adversary life cycle, such as

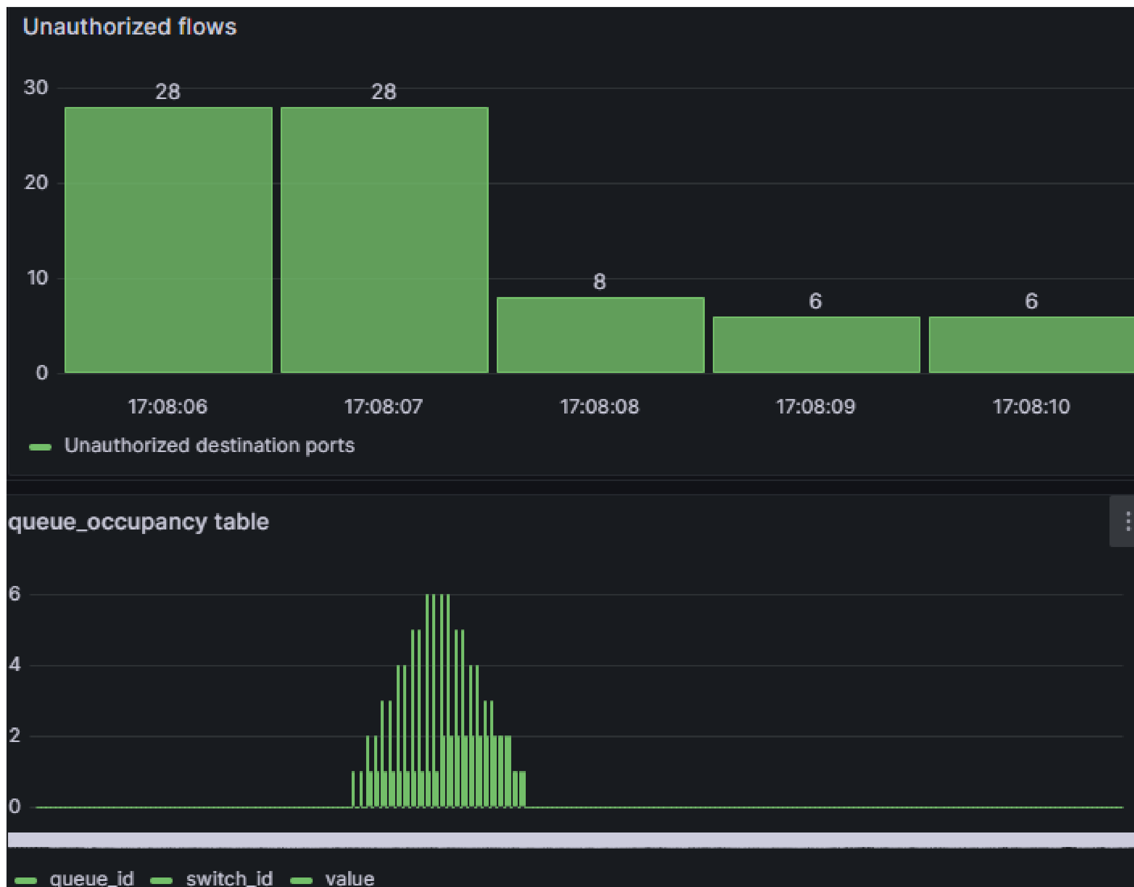

Figure 6.3: h1 accessing ports not authorized through a *nmap* scan.

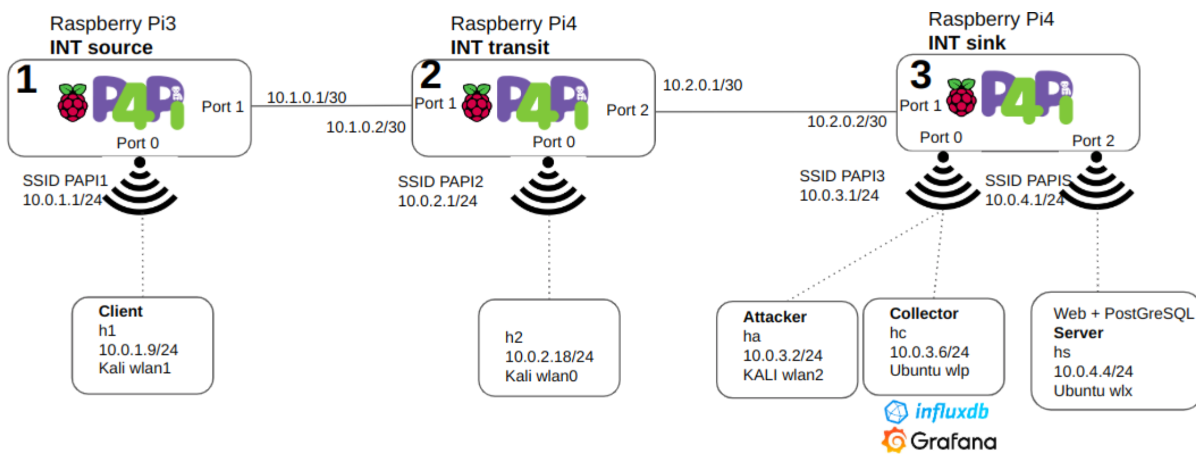

Figure 6.4: Diagram of the INT solution with three P4Pi devices.

using gathered information to plan and execute **Initial Access**. Thus, an adversary starts with eavesdropping the communications.

An earlier step consisting in **Active Scanning** [110] from the client was detected by the INT in the dashboard, as described in the Figure 6.2. However this INT infrastructure is not able to analyze the traffic from an attacker close to the INT collector. So, considering the attacker has taken over an host, or connected to the SSID PAPi3, where the collector is, the adversary may launch a *nmap* scan and identify the other hosts in this subnet: 10.0.3.6 and 10.0.3.1. The next step is to try to eavesdrop with ARP poisoning.

ARP poisoning

In this scenario, the adversary will try to listen to the traffic using tools like *ettercap* [106]. We used *ettercap* to do the ARP poisoning and thus mislead the switch to send the data to the rogue host. This is a simple MITM attack, that starts with eavesdropping, between the P4Pi3 port 0 and the collector:

```
sudo ettercap -T -M arp:oneway //10.0.3.1/ //10.0.3.6/ -i wlan2
Listening on:
 wlan2 -> 00:0F:00:7B:5C:E1
          10.0.3.2/255.255.255.0
2 hosts added to the hosts list...
ARP poisoning victims:
 GROUP 1 : 10.0.3.1 66:2F:C3:F0:B1:F2
 GROUP 2 : 10.0.3.6 E4:A4:71:CD:52:99
```

The expected result is to poison the current ARP table of P4Pi3 for the br0 interface:

```
root@p4pi3:/home/pi# arp -n
Address           HWtype  HWaddress           Flags Mask     Iface
10.0.3.2          ether   00:0f:00:7b:5c:e1   C              br0
10.0.3.6          ether   e4:a4:71:cd:52:99   C              br0
```

We confirmed that we could collect traffic between the P4 switch and the collector but not any of the INT reports. This attack was not successful because the packet sent to the INT collector is already created with the destination L2 address. In fact, the table loaded to the P4 switch defines the L2 address, as we can see from the excerpt below:

```
table_add process_int_report.tb_generate_report
   do_report_encapsulation => 9a:4b:89:ad:8a:59 e4:a4:71:cd:52:99
   10.0.3.1 10.0.3.6 1234
```

MAC spoofing

The attacker can try to attack by changing its own MAC address. This attack is only possible if the switch does not have port security or Dynamic ARP Inspection (DAI) in its Ethernet ports, as explained in the Section 6.3.

However in a WiFi environment this is easily exploited. We used the *macchanger* [111] with the following commands:

```
sudo ifconfig  wlan2 down
sudo macchanger -m e4:a4:71:cd:52:99 wlan2
Current MAC:   00:0f:00:7b:5c:e1 (Legra Systems, Inc.)
Permanent MAC: 00:0f:00:7b:5c:e1 (Legra Systems, Inc.)
New MAC:       e4:a4:71:cd:52:99 (unknown)
sudo ifconfig  wlan2 up
```

We immediately got the INT packets in the attacker machine and we were able to use these packets for a follow-up replay attack:

```
sudo tcpdump -e -i wlan2 udp
9a:4b:89:ad:8a:59 > e4:a4:71:cd:52:99, ethertype IPv4 (0x0800),
   length 252: 10.0.3.1.1234 > 10.0.3.6.1234: UDP, length 210
```

When doing this, the observed behavior was that the P4PI3 AP started sending the packets to only this attacker. So, the collector stopped getting data. This is not the stealth attack that one attacker would prefer. However an attacker could now send replays towards the collector, and minimize the interruption. We then reverted the changes with the same tool with the commands:

```
sudo ifconfig  wlan2 down
sudo macchanger -p wlan2
Current MAC:   e4:a4:71:cd:52:99 (unknown)
Permanent MAC: 00:0f:00:7b:5c:e1 (Legra Systems, Inc.)
New MAC:       00:0f:00:7b:5c:e1 (Legra Systems, Inc.)
sudo ifconfig  wlan2 up
```

ARP poisoning, collector in a SoC

We confirmed that the ARP poisoning was not successful because the packet is crafted with the MAC defined in the P4 table and not from a learned ARP table.

Considering a scenario in which the collector would be in a SOC, there would be some L3 switching to it. So, we added a router to the network as described in the diagram of Figure 6.5. This is a more complex setup as evidenced in the picture

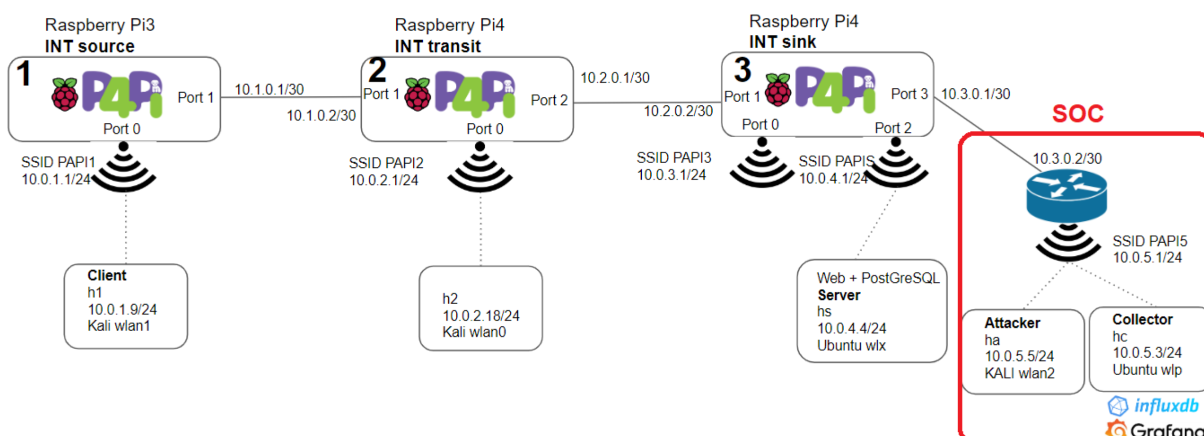

Figure 6.5: Diagram of the INT solution with three P4Pis and a router to a SOC.

of Figure 6.6. As the Raspberry Pi only has one on-board Ethernet and one WiFi interface, we added one new USB-to-Ethernet device.

Figure 6.6: INT lab with three P4Pis and one router.

We also had to update the INT sink interfaces, tables and routing:

- s3 needs a new interface in the P4 switch command;
- s3 needs to forward the packets towards 10.0.5.0/24;
- s3 needs to send the mirrored packets through the new interface;
- s3 needs new L2 and L3 addresses for the reports;
- h1 and the server need the new static routes.

```
#updated service parameters
sudo simple_switch_grpc -i 0@veth0 -i 1@eth0 -i 2@wlx38a28c80c2ee -
   i 3@enx7cc2c6484f4b ${BM2_WDIR}/bin/${P4_PROG}.json
#new L3 switching
table_add l3_forward.ipv4_lpm ipv4_forward 10.0.5.3/32 => e4:a4:71:
   cd:52:99 3
#new port for the mirroring
mirroring_add 500 3
#new mirroring L2 and L3 addresses
table_add process_int_report.tb_generate_report
   do_report_encapsulation => 7c:c2:c6:48:4f:4b 60:e3:27:bd:f0:b3
   10.3.0.1 10.0.5.3 1234
```

The last line defines the INT report sent to the collector. Now, the L2 addresses are different: from the new USB-to-Ethernet interface to the Ethernet interface of

the router. This way it is up to the router to create a new L2 frame in the segment towards the controller. In this case it is WiFi, but it would behave the same way in a cabled Ethernet:

1. The router receives the frame, and confirms the destination IP is not itself.
2. The router checks its ARP table for the destination IP and creates a new frame.
3. The router sends the new frame with its own source MAC and the destination MAC of the collector.

We confirmed this new setup to be working fine, so we tested again the ARP poisoning attack from the host *ha*:

```
sudo ettercap -T -M arp //10.0.5.3/ -i wlan2 -t udp
Listening on:
 wlan2 -> 00:0F:00:7B:5C:E1
          10.0.5.5/255.255.255.0
ARP poisoning victims:
 GROUP 1 : 10.0.5.3 E4:A4:71:CD:52:99
 GROUP 2 : Any
```

As confirmed in Wireshark, captured in Figure 6.7, the attacker sends unsolicited ARP replies with the spoofed source MAC address of the collector to the router.

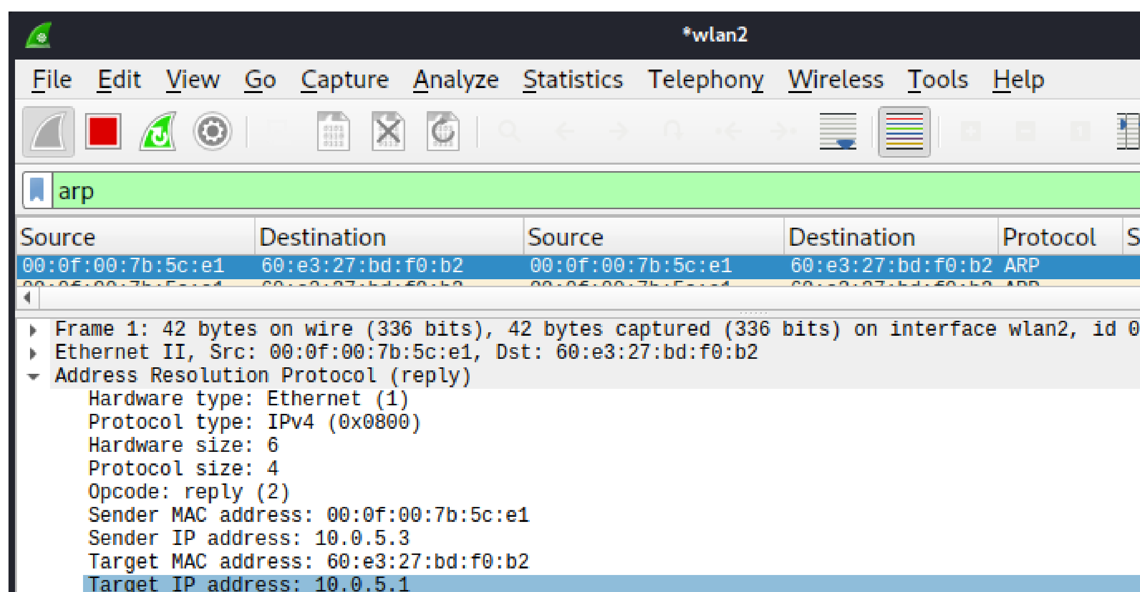

Figure 6.7: ARP poisoning attack, view of an ARP reply with wireshark.

As such, the ARP table of the router becomes compromised and the router instead of sending the frames to the collector, sends to the attacker, who now can receive them:

```
#ettercap view
UDP  10.3.0.1:1234 --> 10.0.5.3:1234 |  (210)

#tcpdump view
60:e3:27:bd:f0:b2 > 00:0f:00:7b:5c:e1 , ethertype IPv4 (0x0800),
length 252: 10.3.0.1.1234 > 10.0.5.3.1234: UDP, length 210
```

6.2.2 INT replay

An attacker could do a replay attack by sending fake data towards the INT collector:

- collects a previous INT message or craft INT stats;
- spoofs the MAC and IP sources as the P4Pi3 gateway;
- sends towards the collector;

In this case we used a previously captured INT message with the *macchanger* attack and included into a Python script, referred in Section 4.1.1 which requires the *scapy* library. The captured INT messaged is sent as the payload, with forged source IP to simulated as if it was sent by the INT sink switch, as described in Figure 6.8.

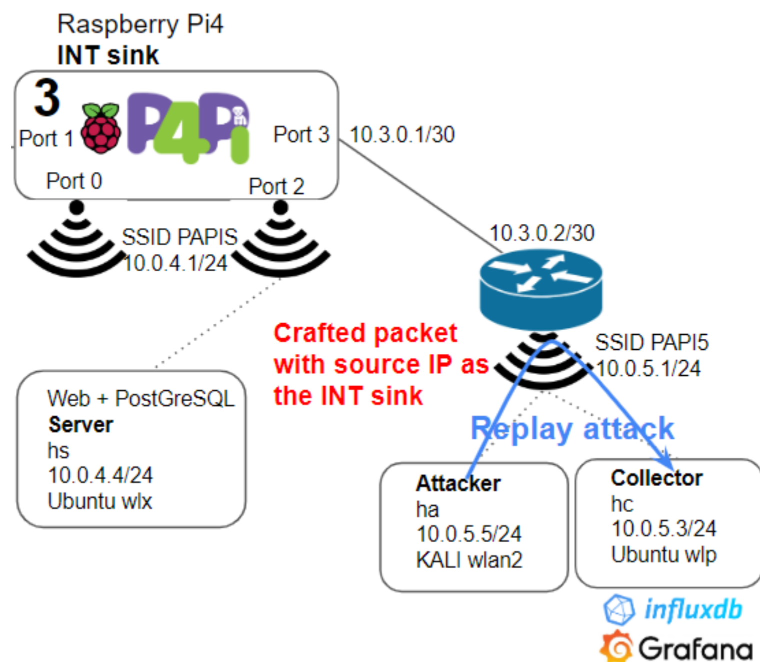

Figure 6.8: Scenario of INT replay attack.

This replay simulated a flow coming from h1 to h2 towards the HTTP port, hence the attacker could use it to simulate a normal working status and thus hide other attacks. At this point, the attacker could stop the P4 switching but the fake statistics could continue flowing to the collector.

In Figure 6.9 we can observe the fake statistics. In this case, the replay attack was as simple as sending the same data, but the adversary could generate different data in order to avoid detection due to the non-variability of the data.

6.2.3 INT manipulation

The manipulation of the INT reports can be used to hide some attacks. This is a typical *MITRE ATT&CK* **defense evasion** [112] which consists of techniques that

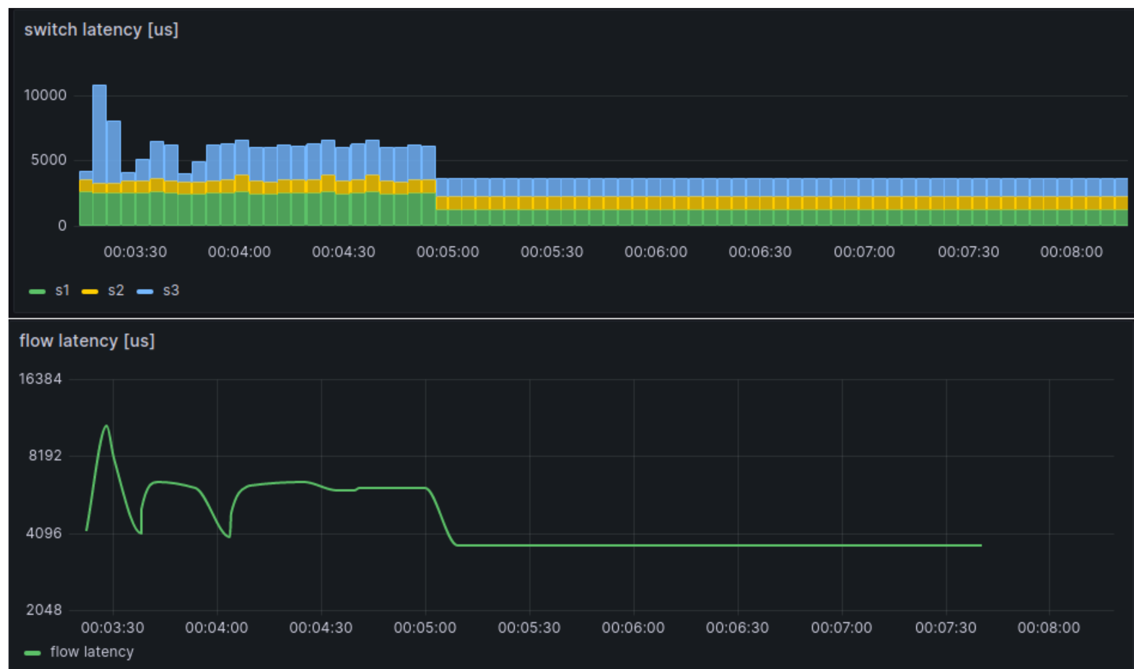

Figure 6.9: INT replay attack, pre-captured data sent to the collector.

adversaries use to avoid detection throughout their compromise.

In this case, there is an abusive usage of the server from the client. The unauthorized access to the ssh service is reported by INT but the attacker is able to manipulate the statistics by changing the data so that the telemetry reports HTTP. This scenario is described in the Figure 6.10.

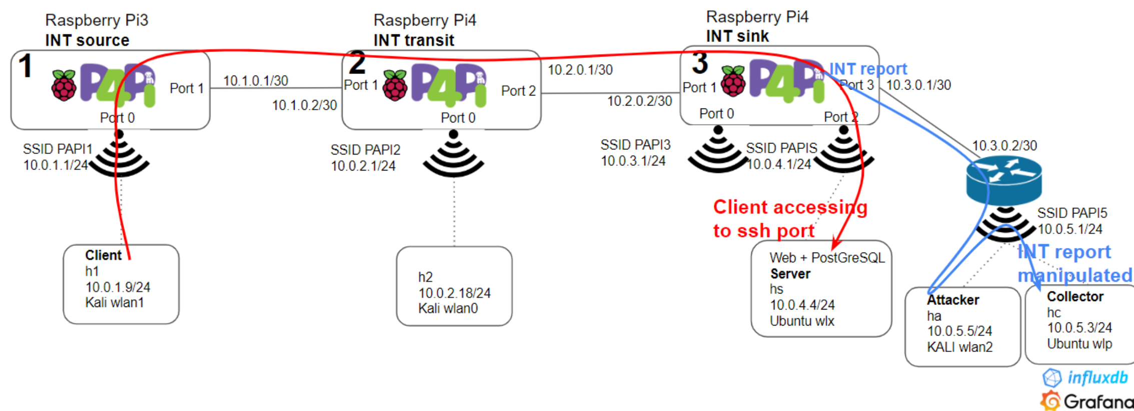

Figure 6.10: Scenario of an attack hidden from INT with an *ettercap* filter.

As such, this unauthorized access is hidden by using an *ettercap* filter designed to replace that specific information (port 22) in the INT report with http (port 80):

```
#P4Pi_etterfilter
if (ip.proto == UDP && udp.dst == 1234) {
  if (DATA.data + 56 == "\x00\x16") {
    msg("Access to SSH 22, replace with 80!\n");
    DATA.data + 56 = "\x00\x50";
  }
}
```

```
#compile the filter
etterfilter P4Pi_etterfilter -o P4Pi_etterfilter.ef

#launch the attack
sudo ettercap -T -M arp //10.0.5.3/ -i wlan2 -F P4Pi_etterfilter.ef
     -t udp

#test
sudo python3 sendwlan1.py --ip 10.0.4.4 --l4 udp --port 22 --m "
    attack"
```

As expected, there is nothing detected, except some more measurements related to access on port 80, as we can confirm in the Figure 6.11.

Figure 6.11: Unauthorized access to a SSH service is misreported as an authorized access to HTTP, using an *ettercap* filter.

6.3 Defending INT

We were able to perform the replay attacks but had difficulties with the eavesdropping and the manipulation of INT. These attacks were partially blocked due to the static ARP network. However, as simulated with a more complex network

(in Figure 6.5), these attacks are effective and easy to conceal. Given that the attacks are possible and their impacts, the INT platform must be itself protected.

Static ARP

Having a network doing switching with static ARP protects from some attacks but it becomes unmanageable and not scalable. This is highly effective in preventing ARP Poisoning attacks but adds a tremendous administrative burden. Any change to the network will require manual updates of the ARP tables across all hosts, making static ARP tables unfeasible for most organizations. Still, in situations where security is crucial, carving out a separate network segment where static ARP tables are used can help to protect critical information. This could be applicable in an high security part of network such as the one in our scenario where we have the server and the INT collector.

However it is as easy to spoof MAC addresses as it is to spoof IPs, hence it is not a good practice to rely only on the MAC addresses of the devices.

L2 switch protection

In a real network there would be other L2/L3 devices with ARP-learning between s3 and the hosts. These L2/L3 devices shall include security features such as Dynamic ARP Inspection (DAI), Port Security or DHCP snooping:

- **DAI:** known as DAI, evaluate the validity of each ARP message and drop the packets that appear suspicious or malicious. DAI determines the validity of an ARP packet based on valid IP-to-MAC address bindings stored in a trusted database, the DHCP snooping binding database. DAI can also be configured to limit the rate at which ARP messages can pass through the switch, effectively preventing DoS attacks [113].
- **Port Security:** allows to define a single MAC address on a switch port, depriving an attacker the chance to maliciously assume multiple network identities.
- **DHCP snooping:** drops DHCP traffic determined to be unacceptable. DHCP Snooping prevents unauthorized (rogue) DHCP servers offering IP addresses to DHCP clients. This is a functionality that can also be built in P4 switches [31].
- **IP Source Guard:** source IP address is filtered on the port, forwarding non-DHCP traffic only if the client IP is established. Packets with any spoofed IP addresses will be dropped since the IP will not be found in the DHCP bindings table. This is a functionality that can also be built in P4 switches [31].

Physical security

Additionally, it is a good security practice to deploy **physical security** policies (e.g. controlling physical access to the network infrastructure). As ARP messages are not routed beyond the boundaries of the local network, the attackers must

be in physical proximity to the victim network or already have control of a machine on the network. In the case of wireless networks, there are other security measures applicable such as a Wireless Intrusion Prevention System (WIPS).

IDS or IPS

An Intrusion Detection System (IDS) or Intrusion Prevention System (IPS) can also be installed in the collector. One such example is the Snort [114] that can alert attacks such as the ARP poisoning.

The ARP poisoning can be detected with specific Snort *preprocessors* [115]. We installed Snort in the collector host and included, in the Snort configuration file, the protected hosts' IP and MAC of the collector and the gateway:

```
#snort.conf
preprocessor arpspoof: -unicast
preprocessor arpspoof_detect_host: 10.0.5.3 e3:a4:71:cd:52:99
preprocessor arpspoof_detect_host: 10.0.5.1 60:e3:27:bd:f0:b2
```

We tested the *ettercap* attack and Snort immediately identified the event with the alarms:

```
[112:4:1] (spp_arpspoof) Attempted ARP cache overwrite attack [**]
[112:2:1] (spp_arpspoof) Ethernet/ARP Mismatch request for Source
   [**]
```

Protecting the INT data

Given the importance of the INT reports and how easy it is to do a replay attack, the integrity and the authenticity of the sender must be assured. One way to achieve these security requirements would be to use TLS over the UDP flows of the INT reports.

DTLS Datagram Transport Layer Security (DTLS) is a communication protocol providing security to UDP flows preventing eavesdropping, tampering, or message forgery. The DTLS protocol is based on the stream-oriented Transport Layer Security (TLS) protocol and is intended to provide similar security guarantees. However the DTLS protocol introduces some drawbacks:

- **DTLS handshake.** This imposes am heavy process with exchange of cookies, certificates, key and cipher totalling more than 15 messages. Additionally some more latency.
- **DTLS CPU and memory costs.** The CPU and memory requirements may not be an issue for the collector, but may be an issue for the P4 switch.
- **DTLS overhead.** After the handshake, the data is sent encrypted with non-negligible overhead.

IPSec It was already demonstrated that it is feasible to develop a host-to-site tunnel over Internet Protocol Security (IPSec) with a P4-switch, as documented

by [116]. It was confirmed that within a BMv2 setup it was possible to reach a throughput close to 50Mbps which exceeds the requirements for a typical telemetry solution. Also IPSec introduces similar drawbacks as DTLS due to its own negotiation sequence, CPU, and memory requirements and packet overhead.

6.4 Costs of INT

As discussed, INT can act as a security control addressing the detect and respond functions, but with some costs:

- **overhead on the packets** between the P4 switches.
- **overhead on the resource usage** of the CPU and memory of the P4 switches.
- **overhead on the collection** of the reports.

Overhead on the packets

As described in Figure 2.10, in the INT-MD mode of operation the packets carry the original payload along with the INT statistics. We confirmed this overhead by crafting small packets and then analyzing its contents along the path, as described in Table 6.1.

Where	Payload size [Bytes]
Client	5
INT source	65
INT transit	109
Collector	210
Server	5

Table 6.1: Payload size along a INT-MD path.

This is inline with what is defined for the INT-MD specifications [5]: 4 bytes of the INT Shim header, 12 bytes of the INT header and 44 bytes of the INT payload. As such, we can calculate the payload to increase along a network of n nodes of P4 switches as 60 + 44 x (n-1).

As referred back in Section 2.4.3, this overhead must be taken in account given the Ethernet limitation of 1518 bytes. We have tested the platform sending successfully packets up to 1368 bytes of payload (Ethernet frames of 1410 bytes). The impact of the overhead depends on the size of the packets but it was estimated to reach up to 15% on average [117].

Overhead on resource usage

We tested the maximum possible cadence of messages with the *Python* script at 20 packets per second, and confirmed all packets were reported. We confirmed

the Raspberry Pi devices to do not go over 3% of CPU and 250MB, as evidenced from the *htop* screenshots of the Figure 6.12.

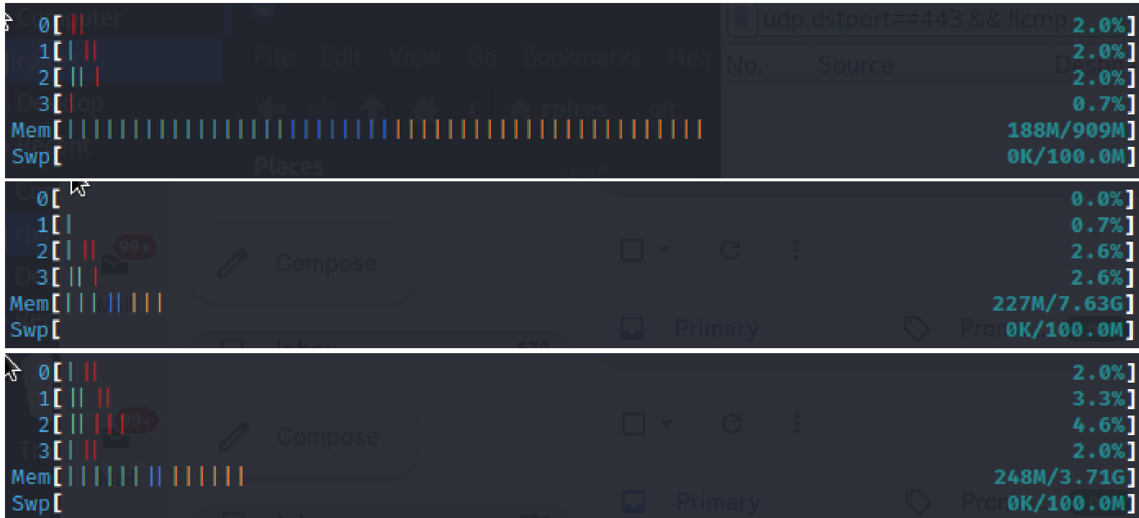

Figure 6.12: CPU and memory usage of the three P4Pi at 20pps.

```
%removed the delay, so it sends as fast as possible
sudo python3 sendwlan1.py --ip 10.0.4.4 --l4 udp --port 80 --m
   INTH1 --c 500
```

The BMv2 open source software switch is not designed for high throughput, as referred in the Section 2.4.2, but we anyhow tested it with *iperf*. We confirmed the CPU of Raspberry Pi devices to go up to 30% but without further requirements of memory, as evidenced from the *htop* screenshots of Figure 6.13.

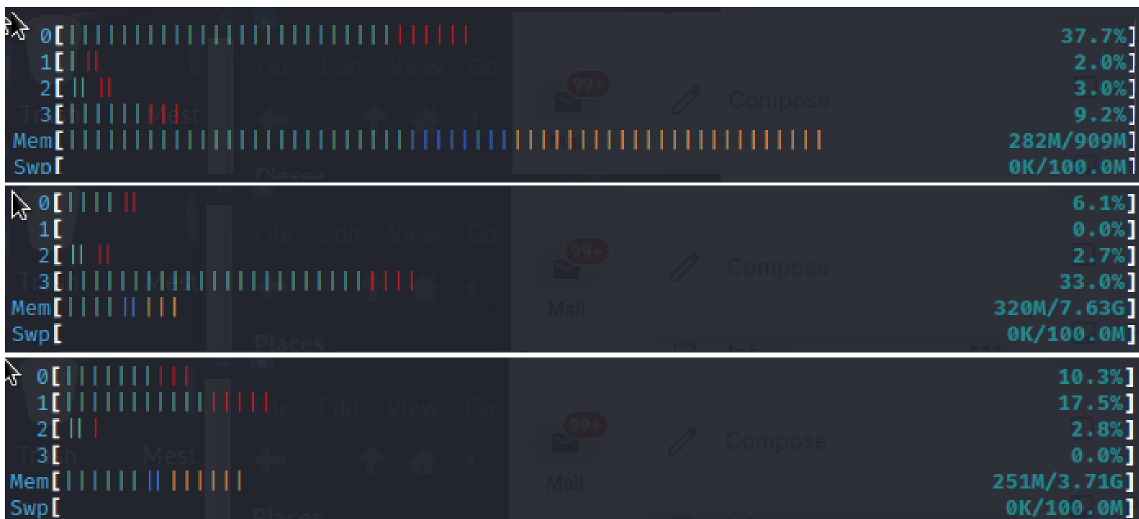

Figure 6.13: CPU and memory usage of the three P4Pi under excessive load with *iperf* at 770pps.

```
in the server: sudo iperf -s -u -p25
in the client; sudo iperf -c 10.0.4.4 -u -p 25 -l 5 -b 10000 -t 60
this provides 60s of 5B payload to udp/25, 10kbps, ~770pps
```

Overhead on the collection of reports

We did some stress tests with *iperf* simulating access to the HTTPS port of the server and confirmed that we could collect statistics up to about 25 reports per second. We then tried to go over that limit, sending more traffic, but we lost many corresponding measurements, as evidenced in the Figure 6.14. After analysis with *tcpdump* we verified that the P4 switches can't withstand the high throughput and don't generate all the reports.

The collector Python script often crashes under high load too.

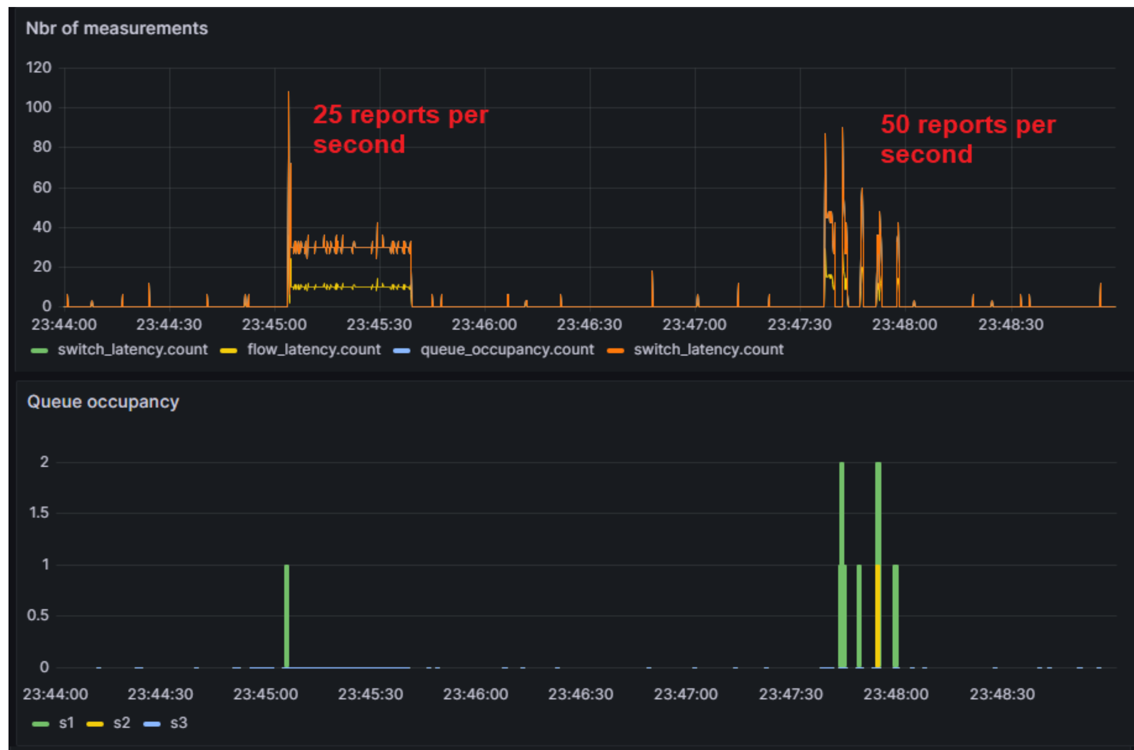

Figure 6.14: High load effect on the queues of the P4 switches and loss of reports.

6.5 Summary

We have confirmed that an INT solution can effectively work as a security control by supporting the detect and respond functions. We demonstrated some network attacks that could otherwise be unnoticed: IP scans, port scans or any other that causes high load.

Given the value of INT, we also addressed how it could be attacked. We considered a rogue host in the protected network starting with reconnaissance, up to the actual exploit by manipulating the data. Such attacks could conceal other attacks such as the one demonstrated in Figure 6.10.

As such, we also addressed the possible solutions to protect the INT platform. These solutions should follow a layered approach:

- physical security: access to the network infrastructure must be safeguarded.
- L2 protection: the switches in the protected network must be secured.
- IDS or IPS: other attacks can be detected or avoided with a solution like Snort.
- INT data: the authenticity of the sender and the integrity of the data must be assured.

We finally covered the costs of the INT solution regarding overhead of the packets, resource overhead and collection.

Chapter 7

Conclusions

This chapter presents the main conclusions from our work, reflecting on the progress achieved during the workplan execution, discussing the tradeoffs and obtained results, as well as pointing future research and development directions.

7.1 Initial research

Under the initial research that covered the first semester and parts of the second semester we have gone through the research about SDN, P4 and P4-INT. We analyzed possible scenarios to simulate and test our implementations. We also tested some tutorials and adapted according to our needs.

7.2 Preliminary test cases

In this phase, we went through other labs and built our own implementation of INT in a P4 environment. Due to the limitations of the ready-made VMs with *Miniedit*, we built our own environment with *Mininet* and adapted the necessary code to create a INT-MD solution with five BMv2 switches within a *Mininet* Spine-Leaf network. In this phase we were able to simulate a INT solution where we could create INT telemetry for the selected flows, with collection for the 3 switches in the path: source, transit and sink.

The telemetry is collected by a InfluxDB database which is queried and the data is displayed in a web-based grafana dashboard.

7.3 Final test cases

In this phase we downsized the topology to three P4 switches due to the shortage of Raspberry Pis. We used the P4Pi open source platform and used the same code as in the *Mininet* labs of the preliminary test cases. We also used the same solution

for the collection and analysis of the INT reports. We used two laptops with two WiFi interfaces each to simulate as they were different hosts in the network.

This network behaved similarly to the preliminary test cases, and we noticed the high latency close to 8ms, as confirmed in Figure 5.5.

7.3.1 INT as security control

We confirmed that INT as a telemetry solution can detect flooding attacks and identify the traffic sources such as the highlighted in Figure 4.13. Other possible attacks such as a port stealing are also recognizable, due to the flooding of ARP packets to the network as evidenced in the Figure 4.16. Other unforeseen attacks can also be detected as a trade-off of security versus performance. We added generic matching rules in the INT source and as such, all traffic is logged. With that, we created a report that identified some attack behaviours used in the reconnaissance phase [109]: port scans or IP scans. This report can help the network administrator to detect these attacks and identify the source, as evidenced in Figures 6.2 and 6.3.

7.3.2 INT as a target

We confirmed that the INT statistics can be an important security asset as the data may be used by the network administrators for assessing the network and troubleshooting any issues. So, it is a valuable target for a malicious adversary.

We tested some attacks, starting with eavesdropping, then replay and manipulation. **Eavesdropping** may be easy to achieve if the attacker can control an host close to the collector, and then spoof its own MAC address or by poisoning the ARP cache. The **replay** attack can be used to simulate a normal working status and thus hide other attacks as illustrated in Figure 6.9. The INT **manipulation** is one step further as it can be used to mislead the network administrator to take wrong actions or hide an attack such as accessing a forbidden service but that being reported as normal. We exemplified this attack, a typical Defense Evasion [112], with accessing a SSH service but INT reporting as an standard access to the http service, as illustrated in Figure 6.11.

7.3.3 Defending INT

Given that the attacks are possible and with impact to the network, the INT platform must itself be protected. We discussed several possible mitigations, that could be applied as a defense in depth approach:

- **Static ARP:** is highly effective in preventing ARP Poisoning attacks but adds a administrative burden, which may be applicable to separate networks such as where an INT collector would be placed. However it is as easy to spoof MAC addresses as it is to spoof IPs, hence it is not a good practice to rely on the MAC

addresses to authenticate the devices.

- **L2 switch protection:** such as DAI, Port Security, DHCP snooping or IP source guard: these controls help to avoid spoofed IPs, multiple MACs in the same port, or fake DHCP servers.
- **physical security:** as e.g. controlling physical access to the network infrastructure, or deploying a WIPS on a WiFi network.
- **assure integrity and the authenticity of the INT sink:** with DTLS or IPSec, however these solutions add costs to the latency, CPU, memory and packet overhead.

7.4 Future work

A future enhancement of an INT solution could focus on the NIST Cyber Security Framework [3] **respond** function. This enhancement could use the data in InfluxDB or the Snort alerts to create new tables in the P4 switches, as described in the Figure 7.1. One example could be the detection of port scans from a host, by querying the InfluxDB, and immediately create drop rules for that specific host in the switches. Optionally, reroute that traffic to a honeypot, for analysis of the adversary methodology.

A future enhancement of an INT solution should focus on some vulnerabilities we explored, such as implementing DTLS or IPSec to make sure the INT data is not eavesdropped, replayed or manipulated.

Other possible improvement would be to enhance the collection capabilities with e.g. deploying collectors running as micro-services or deploying the collector server in a High Availability cluster.

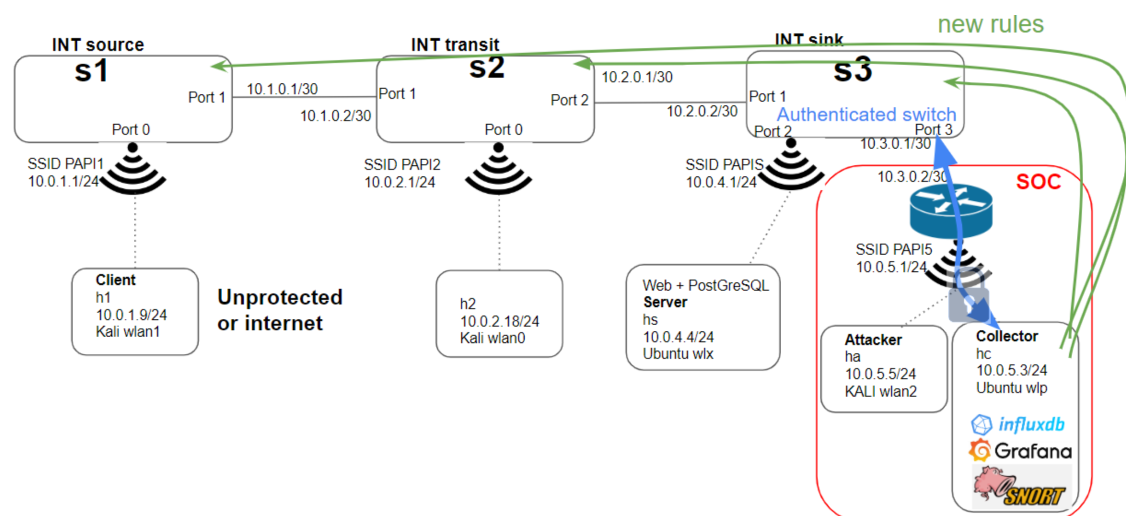

Figure 7.1: Possible improvements of an INT platform: authentication of the switch, encrypted tunnel, encrypted data, and closed loop control.

References

[1] O. N. Foundation. (2022) Software-defined networking (SDN) definition. [Online]. Available: https://opennetworking.org/sdn-definition/

[2] (2022) What is software-defined networking (SDN)? | VMware glossary. [Online]. Available: https://www.vmware.com/topics/glossary/content/software-defined-networking.html

[3] nist. (2023) Mnist cyber security framework. [Online]. Available: https://www.nist.gov/cyberframework

[4] noauthor. (2022) Specifications « p4 – language consortium. [Online]. Available: https://p4.org/specs/

[5] (2022) INT_v2_1.pdf. [Online]. Available: https://p4.org/p4-spec/docs/INT_v2_1.pdf

[6] (2022) MININET. [Online]. Available: https://opennetworking.org/mininet/

[7] R. P. Foundation. (2023) Teach, learn, and make with the raspberry pi foundation. [Online]. Available: https://www.raspberrypi.org/

[8] (2023) Getting started. Original-date: 2021-06-28T20:39:39Z. [Online]. Available: https://github.com/p4lang/p4pi

[9] K. Özdinçer and H. A. Mantar, "SDN-based detection and mitigation system for DNS amplification attacks," in *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2019, pp. 1–7.

[10] T. Keary. (2018) Software-defined networking (SDN) guide: SDN advantages. [Online]. Available: https://www.comparitech.com/net-admin/software-defined-networking/

[11] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN control: Survey, taxonomy, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 333–354, 2018, conference Name: IEEE Communications Surveys & Tutorials.

[12] Z. Latif, K. Sharif, F. Li, M. M. Karim, S. Biswas, and Y. Wang, "A comprehensive survey of interface protocols for software defined networks," *Journal of Network and Computer Applications*, vol. 156, p. 102563, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804520300370

[13] (2022) Openflow 1.5.1 switch specification. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[14] M. Obadia, M. Bouet, J. Leguay, K. Phemius, and L. Iannone, "Failover mechanisms for distributed SDN controllers," in *2014 International Conference and Workshop on the Network of the Future (NOF)*, vol. Workshop, 2014, pp. 1–6.

[15] School of Computer Applications, Lovely Professional University, Phagwara, India., D. K. Ryait, D. M. Sharma, and School of Computer Applications, Lovely Professional University, Phagwara, India., "To eliminate the threat of a single point of failure in the SDN by using the multiple controllers," *International Journal of Recent Technology and Engineering (IJRTE)*, vol. 9, no. 2, pp. 234–241, 2022. [Online]. Available: https://www.ijrte.org/portfolio-item/B3433079220/

[16] L. Sidki, Y. Ben-Shimol, and A. Sadovski, "Fault tolerant mechanisms for SDN controllers," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, pp. 173–178.

[17] A. J. Gonzalez, G. Nencioni, B. E. Helvik, and A. Kamisinski, "A fault-tolerant and consistent SDN controller," in *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2016, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7841496/

[18] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "FlowVisor: A network virtualization layer," *FlowVisor: A Network Virtualization Layer*, 2009.

[19] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: a distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. USENIX Association, 2010, pp. 351–364.

[20] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," *HyperFlow: A Distributed Control Plane for OpenFlow*, p. 6, 2022.

[21] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*, ser. HotSDN '14. Association for Computing Machinery, 2014, pp. 1–6. [Online]. Available: https://doi.org/10.1145/2620728.2620744

[22] V. Pashkov and R. Smeliansky, "On high availability distributed control plane for software-defined networks," in *2018 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)*, 2018, pp. 1–10.

[23] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of the first workshop on Hot topics in software defined networks*, ser. HotSDN '12. Association for Computing Machinery, 2012, pp. 19–24. [Online]. Available: https://doi.org/10.1145/2342441.2342446

[24] A. D. Ferguson, S. Gribble, C.-Y. Hong, C. Killian, W. Mohsin, H. Muehe, J. Ong, L. Poutievski, A. Singh, L. Vicisano, R. Alimi, S. S. Chen, M. Conley, S. Mandal, K. Nagaraj, K. N. Bollineni, A. Sabaa, S. Zhang, M. Zhu, and A. Vahdat, "Orion: Google's Software-Defined networking control plane," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 83–98. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/ferguson

[25] V. Pashkov, A. Shalimov, and R. Smeliansky, "Controller failover for SDN enterprise networks," *2014 International Science and Technology Conference (Modern Networking Technologies) (MoNeTeC)*, 2014, pages: 6.

[26] A. Rao, S. Auti, A. Koul, and G. Sabnis, "High availability and load balancing in SDN controllers," *International Journal of Trend in Research and Development*, vol. 3, p. 5, 2016.

[27] A. Liatifis, P. Sarigiannidis, V. Argyriou, and T. Lagkas, "Advancing SDN: from OpenFlow to p4, a survey," *ACM Computing Surveys*, p. 3556973, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3556973

[28] P4.org. (2016) Clarifying the differences between p4 and OpenFlow. [Online]. Available: https://opennetworking.org/news-and-events/blog/clarifying-the-differences-between-p4-and-openflow/

[29] A. AlSabeh, J. Khoury, E. Kfoury, J. Crichigno, and E. Bou-Harb, "A survey on security applications of p4 programmable switches and a STRIDE-based vulnerability assessment," *Computer Networks*, vol. 207, p. 108800, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128622000287

[30] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014. [Online]. Available: https://doi.org/10.1145/2602204.2602211

[31] N. Narayanan, G. C. Sankaran, and K. M. Sivalingam, "Mitigation of security attacks in the SDN data plane using p4-enabled switches," in *2019 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, 2019, pp. 1–6, ISSN: 2153-1684.

[32] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the SDN control plane," *USENIX Security 2017*, p. 19, 2022.

[33] T. Sasaki, A. Perrig, and D. E. Asoni, "Control-plane isolation and recovery for a secure SDN architecture," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, pp. 459–464.

[34] G. Abhilash and G. Divyansh, "Intrusion detection and prevention in software defined networking," in *2018 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, 2018, pp. 1–4, ISSN: 2153-1684.

[35] G. Pickett, "Abusing software defined networks," *https://www.blackhat.com/docs/eu-14/materials/eu-14-Pickett-Abusing-Software-Defined-Networks-wp.pdf*, p. 14, 2014.

[36] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software defined network control layer," in *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/securing-software-defined-network-control-layer/

[37] Z. J. Haas, T. L. Culver, and K. Sarac, "Vulnerability challenges of software defined networking," *IEEE Communications Magazine*, vol. 59, no. 7, pp. 88–93, 2021, conference Name: IEEE Communications Magazine.

[38] Y. Gao, Z. Wang, and S.-B. Tsai, "A review of p4 programmable data planes for network security," *Mobile Information Systems*, vol. 2021, 2022. [Online]. Available: https://doi.org/10.1155/2021/1257046

[39] noauthor, "P4_tutorial," 2022, online. [Online]. Available: https://docs.google.com/presentation/d/1zliBqsS8IOD4nQUboRRmF_19poeLLDLadD5zLzrTkVc

[40] V. Gurevich, "Introduction to p4 and data plane programmability," *Introduction to P4 and Data Plane Programmability*, p. 46, 2022.

[41] (2023) P4 – language consortium. [Online]. Available: https://p4.org/

[42] (2022) P4runtime specification. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[43] (2022) protocolbuffers/protobuf: Protocol buffers - google's data interchange format. [Online]. Available: https://github.com/protocolbuffers/protobuf

[44] N. McKeown. (2017) P4 runtime - putting the control plane in charge of the forwarding plane. [Online]. Available: https://opennetworking.org/news-and-events/blog/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane/

[45] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network*

and Computer Applications, vol. 212, p. 103561, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804522002028

[46] (2017) Programming networks with p4. [Online]. Available: https://octo.vmware.com/programming-networks-with-p4/

[47] 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*, vol. 9, pp. 87 094–87 155, 2021, conference Name: IEEE Access.

[48] G. Simsek, D. Ergenç, and E. Onur, "Efficient network monitoring via in-band telemetry," in *2021 17th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2021, pp. 1–6.

[49] (2023) TCP-INT: Lightweight in-band network telemetry for TCP. Original-date: 2022-06-16T18:27:53Z. [Online]. Available: https://github.com/p4lang/p4app-TCP-INT

[50] M. L. A. Syafii. (2022) ONOS-p4-INT. Original-date: 2022-09-10T07:55:16Z. [Online]. Available: https://github.com/assyafii/ONOS-P4-INT

[51] (2023) Prometheus - monitoring system & time series database. [Online]. Available: https://prometheus.io/

[52] (2023) Grafana: The open observability platform | grafana labs. [Online]. Available: https://grafana.com/

[53] (2022) 10. lab p4-INT (in-band network telemetry) using ONOS and eBPF :: M. luthfi as syafii. [Online]. Available: https://luthfi.dev/posts/lab-p4-int-in-band-network-telemetry-with-onos-from-scratch/

[54] (2023) tutorials/exercises/mri at master · p4lang/tutorials. [Online]. Available: https://github.com/p4lang/tutorials

[55] M. S. Towhid. (2023) int_p4. Original-date: 2023-02-06T01:23:40Z. [Online]. Available: https://github.com/shamimtowhid/int_p4

[56] (2023) Host-INT* for packet-telemetry. Original-date: 2021-05-17T22:15:47Z. [Online]. Available: https://github.com/intel/host-int

[57] (2023) tutorials/exercises/link_monitor at master · p4lang/tutorials. [Online]. Available: https://github.com/p4lang/tutorials

[58] (2023) Common p4-based INT implementation for bmv2-mininet and tofino platforms. Original-date: 2021-02-24T07:57:16Z. [Online]. Available: https://github.com/GEANT-DataPlaneProgramming/int-platforms

[59] M. Campanella, T. Martinek, J. Hill, M. Gerola, J. Kabat, M. Demolianis, and T. Vasilopoulos, "In-band network telemetry using data plane programming," *IEEE Access*, 2022.

[60] Mandar. (2023) In-band network telemetry implementation in p4. Original-date: 2021-06-03T07:18:53Z. [Online]. Available: https://github.com/mandaryoshi/p4-int

[61] M. Joshi, "Implementation and evaluation of inband network telemetry in p4," *IEEE Access*, 2021.

[62] laofan. (2023) P4-INT. Original-date: 2022-07-11T09:34:10Z. [Online]. Available: https://github.com/laofan13/P4-INT

[63] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "Programming protocol-independent packet processors," 2014. [Online]. Available: http://arxiv.org/abs/1312.1719

[64] (2023) P4-16-v1.2.4 specification. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.4.pdf

[65] (2022) P4-16-v1.2.3 specification. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.3.pdf

[66] A. d. S. Ilha, A. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Euclid: A fully in-network, p4-based approach for real-time DDoS attack detection and mitigation," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3121–3139, 2021, conference Name: IEEE Transactions on Network and Service Management.

[67] I. Oliveira, E. Neto, R. Immich, R. Fontes, A. Neto, F. Rodriguez, and C. E. Rothenberg, "dh-aes-p4: On-premise encryption and in-band key-exchange in p4 fully programmable data planes," in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2021, pp. 148–153.

[68] R. Datta, S. Choi, A. Chowdhary, and Y. Park, "P4guard: Designing p4 based firewall," in *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, 2018, pp. 1–6, ISSN: 2155-7586.

[69] C. Black and S. Scott-Hayward, "Adversarial exploitation of p4 data planes," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 508–514, ISSN: 1573-0077.

[70] E. Sakic, N. Deric, E. Goshi, and W. Kellerer, "P4bft: Hardware-accelerated byzantine-resilient network control plane," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–7, ISSN: 2576-6813.

[71] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, and S. Schmid, "P4consist: Toward consistent p4 SDNs," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 7, pp. 1293–1307, 2020, conference Name: IEEE Journal on Selected Areas in Communications.

[72] Z. Zhou, M. He, W. Kellerer, A. Blenk, and K.-T. Foerster, "P4update: fast and locally verifiable consistent network updates in the p4 data

plane," in *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '21. Association for Computing Machinery, 2021, pp. 175–190. [Online]. Available: https://doi.org/10.1145/3485983.3494845

[73] H. Xie and J. Zhao, "A lightweight identity authentication method by exploiting network covert channel," *Peer-to-Peer Networking and Applications*, vol. 8, 2014.

[74] E. O. Zaballa, D. Franco, Z. Zhou, and M. S. Berger, "P4knocking: Offloading host-based firewall functionalities to the network," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 7–12, ISSN: 2472-8144.

[75] (2021) Port knocking. [Online]. Available: https://4pfsec.com/port-knocking/

[76] A. Bhattacharya, R. Rana, S. Datta, and V. U., "P4-sKnock: A two level host authentication and access control mechanism in p4 based SDN," in *2022 27th Asia Pacific Conference on Communications (APCC)*, 2022, pp. 278–283, ISSN: 2163-0771.

[77] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "p4pktgen: Automated test case generation for p4 programs," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. Association for Computing Machinery, 2018, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3185467.3185497

[78] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime verification of p4 switches with reinforcement learning," in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, ser. NetAI'19. Association for Computing Machinery, 2019, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3341216.3342206

[79] S. Kodeswaran, M. T. Arashloo, P. Tammana, and J. Rexford, "Tracking p4 program execution in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '20. Association for Computing Machinery, 2020, pp. 117–122. [Online]. Available: https://doi.org/10.1145/3373360.3380843

[80] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 490–503. [Online]. Available: https://dl.acm.org/doi/10.1145/3230543.3230582

[81] K. S. Kumar, R. K, P. S. Prashanth, M. T. Arashloo, V. U., and P. Tammana, "DBVal: Validating p4 data plane runtime behavior," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, ser. SOSR '21. Association for Computing Machinery, 2021, pp. 122–134. [Online]. Available: https://doi.org/10.1145/3482898.3483352

[82] Y. Zhou, J. Bi, C. Zhang, B. Liu, Z. Li, Y. Wang, and M. Yu, "P4db: On-the-fly debugging for programmable data planes," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1714–1727, 2019, conference Name: IEEE/ACM Transactions on Networking.

[83] Y. Zhou, J. Bi, T. Yang, K. Gao, C. Zhang, J. Cao, and Y. Wang, "KeySight: Troubleshooting programmable switches via scalable high-coverage behavior tracking," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018, pp. 291–301, ISSN: 1092-1648.

[84] Z. Xia, J. Bi, Y. Zhou, and C. Zhang, "KeySight: A scalable troubleshooting platform based on network telemetry," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. Association for Computing Machinery, 2018, pp. 1–2. [Online]. Available: https://doi.org/10.1145/3185467.3190787

[85] Q. Kang, J. Xing, Y. Qiu, and A. Chen, "Probabilistic profiling of stateful data planes for adversarial testing," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. Association for Computing Machinery, 2021, pp. 286–301. [Online]. Available: https://doi.org/10.1145/3445814.3446764

[86] (2022) Authentication. Section: docs. [Online]. Available: https://grpc.io/docs/guides/auth/

[87] (2023) Mininet overview - mininet. [Online]. Available: http://mininet.org/overview/

[88] Oracle. (2023) Oracle VM VirtualBox. [Online]. Available: https://www.virtualbox.org/

[89] (2023) P4 tutorial. Original-date: 2015-09-30T19:17:28Z. [Online]. Available: https://github.com/p4lang/tutorials

[90] (2023) tutorials/exercises/p4runtime at master · p4lang/tutorials. [Online]. Available: https://github.com/p4lang/tutorials

[91] U. of South Carolina. (2022) Cybertraining: CI. [Online]. Available: http://ce.sc.edu/cyberinfra/cybertraining.html

[92] (2022) Next-gen SDN tutorial (advanced). Original-date: 2019-08-30T02:43:43Z. [Online]. Available: https://github.com/opennetworkinglab/ngsdn-tutorial

[93] conorblack. (2022) Adversarial exploitation of p4 data planes. Original-date: 2021-01-15T11:26:14Z. [Online]. Available: https://github.com/conorblack/AdvExpP4DP

[94] S. Laki, R. Stoyanov, D. Kis, R. Soulé, P. Vörös, and N. Zilberman, "P4pi: P4 on raspberry pi for networking education," *ACM SIGCOMM Computer Communication Review*, vol. 51, no. 3, pp. 17–21, 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3477482.3477486

[95] (2023) Introduction to p4pi · p4lang/p4pi wiki. [Online]. Available: https://github.com/p4lang/p4pi/wiki/Introduction-to-P4Pi

[96] Daniel. (2023) What is spine and leaf network architecture? [Online]. Available: https://study-ccna.com/spine-and-leaf-architecture/

[97] (2023) Mininet how to manual. [Online]. Available: https://www.brianlinkletter.com/2015/04/how-to-use-miniedit-mininets-graphical-user-interface/

[98] (2023) P4 tutorial basic routing. [Online]. Available: https://github.com/p4lang/tutorials/tree/master/exercises/basic

[99] (2023) P4 tutorial link monitor. [Online]. Available: https://github.com/p4lang/tutorials/tree/master/exercises/link_monitor

[100] (2023) P4 tutorial arp and icm responder. [Online]. Available: https://github.com/p4lang/p4pi/wiki/Example-%232---ARP-and-ICMP-responder-with-Bmv2-target

[101] R. Pires. (2023) P4int_mininet. Original-date: 2023-05-10T15:31:40Z. [Online]. Available: https://github.com/ruimmpires/P4INT_Mininet

[102] (2023) InfluxDB times series data platform | InfluxData. [Online]. Available: https://www.influxdata.com/

[103] N. V. Tu, J. Hyun, G. Y. Kim, J.-H. Yoo, and J. W.-K. Hong, "INTCollector: A high-performance collector for in-band network telemetry," in *2018 14th International Conference on Network and Service Management (CNSM)*, 2018, pp. 10–18, ISSN: 2165-963X.

[104] laofan. (2022) P4-INT. Original-date: 2022-07-11T09:34:10Z. [Online]. Available: https://github.com/lifengfan13/P4-INT

[105] mitre. (2023) Mitre att&ck® framework. [Online]. Available: https://attack.mitre.org/

[106] ettercap. (2023) ettercap - linux man page. [Online]. Available: https://linux.die.net/man/8/ettercap

[107] R. Pires. (2023) P4int_p4pi. [Online]. Available: https://github.com/ruimmpires/P4INT_P4PI

[108] (2023) Install grafana on debian or ubuntu | grafana documentation. [Online]. Available: https://grafana.com/docs/grafana/latest/setup-grafana/installation/debian/

[109] mitre. (2023) Mitre att&ck® framework reconnaissance phase. [Online]. Available: https://attack.mitre.org/tactics/TA0043/

[110] ——. (2023) Mitre att&ck® framework active scanning phase. [Online]. Available: https://attack.mitre.org/tactics/T1595/

[111] (2023) macchanger | kali linux tools. [Online]. Available: https://www.kali.org/tools/macchanger/

[112] mitre. (2023) Mitre att&ck® framework defense evasion phase. [Online]. Available: https://attack.mitre.org/tactics/TA0005/

[113] Cisco. (2023) Cisco catalyst6500 dynamic arp. [Online]. Available: www.cisco.com/content/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SXF/native/configuration/guide/swcg/dynarp.html.xml

[114] (2023) Snort ips. [Online]. Available: https://www.snort.org/

[115] (2023) Snort arpspoof manual. [Online]. Available: http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node17.html#SECTION00321500000000000000

[116] F. Hauser, M. Häberle, M. Schmidt, and M. Menth, "P4-IPsec: Site-to-site and host-to-site VPN with IPsec in p4-based SDN," *IEEE Access*, vol. 8, pp. 139 567–139 586, 2020. [Online]. Available: http://arxiv.org/abs/1907.03593

[117] R. T. Tiburski, L. A. Amaral, E. De Matos, D. F. G. De Azevedo, and F. Hessel, "Evaluating the use of TLS and DTLS protocols in IoT middleware systems applied to e-health," in *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2017, pp. 480–485. [Online]. Available: http://ieeexplore.ieee.org/document/7983155/

Appendices

Appendix A

Installation and configuration in P4Pi

All code and details are documented in our P4-INT for P4Pi GitHub repository [107].

A.1 Pre-requisites

We started with following the steps in the P4Pi wiki, downloaded the image, burnt it to the SD-cards and then booted and configure each one. The example steps below are for s3, the INT sink. The other switches have a similar configuration except for the IP addresses, interfaces and tables.

- connect via WiFi, pi/raspberry;
- ssh, pi/raspberry, sudo su, apt update, apt install nano (use an ethernet cable to your LAN);
- change static IPs, br0, eth0, and the second WiFi interface:

```
nano /etc/dhcpcd.conf
  interface br0
    static ip_address=10.0.3.1/24
    nohook wpa_supplicant
  interface eth0
    static ip_address=10.2.0.2/30
  interface wlx38a28c80c2ee
    static ip_address=10.0.4.1/24
    nohook wpa_supplicant
```

- change hostname

```
nano /etc/hosts
  127.0.0.1   p4pi3
nano /etc/hostname
  p4pi3
```

- change DHCP

```
nano /etc/dnsmasq.d/p4edge.conf
  interface=br0
```

```
  dhcp-range=set:br0,10.0.3.2,10.0.3.10,255.255.255.0,24h
  domain=p4pi3
  address=/gw.p4pi3/10.0.3.1

  interface=wlx38a28c80c2ee
  dhcp-range=set:wlx38a28c80c2ee
   ,10.0.4.2,10.0.4.10,255.255.255.0,24h
  domain=p4pis
  address=/gw.p4pis/10.0.4.1
```

- change WiFi

```
cp /etc/hostapd/hostapd.conf /etc/hostapd/wlan0.conf
cp /etc/hostapd/hostapd.conf /etc/hostapd/wlx38a28c80c2ee.conf
nano /etc/hostapd/wlan0.conf
  ssid=p4pi3
nano /etc/hostapd/wlx38a28c80c2ee.conf
  interface=wlx38a28c80c2ee
  ssid=p4pis
  #bridge=br0
```

- enable the new hostapd services

```
systemctl disable hostapd
systemctl enable hostapd@wlan0
systemctl enable hostapd@wlx38a28c80c2ee
```

- reboot
- connect via WiFi, papi3/raspberry
- ssh, pi/raspberry, sudo su
- add static routing as needed

```
ip r add 10.0.1.0/24 via 10.2.0.1
ip r add 10.0.2.0/24 via 10.2.0.1
```

- copy all the P4 code to /root/bmv2/intv8/. Be sure to name the main p4 file the same as the folder.

A.2 Service mode

- configure the bmv2 service and change as needed:

```
nano /usr/bin/bmv2-start
    #!/bin/bash
    export P4PI=/root/PI
    export GRPCPP=/root/P4Runtime_GRPCPP
    export GRPC=/root/grpc
    BM2_WDIR=/root/bmv2
    P4_PROG=intv8
    T4P4S_PROG_FILE=/root/t4p4s-switch
    if [ -f "${T4P4S_PROG_FILE}" ]; then
        P4_PROG=$(cat "${T4P4S_PROG_FILE}")
    else
        echo "${P4_PROG}" > "${T4P4S_PROG_FILE}"
    fi
```

```
    rm -rf ${BM2_WDIR}/bin
    mkdir ${BM2_WDIR}/bin
    echo "Compiling P4 code"
    p4c-bm2-ss -I /usr/share/p4c/p4include --std p4-16 --
   p4runtime-files \
    ${BM2_WDIR}/bin/${P4_PROG}.p4info.txt -o ${BM2_WDIR}/bin/${
   P4_PROG}.json \
    ${BM2_WDIR}/${P4_PROG}/${P4_PROG}.p4
    echo "Launching BMv2 switch"
    sudo simple_switch_grpc -i 0@veth0 -i 1@eth0 -i 2
   @wlx38a28c80c2ee \
    ${BM2_WDIR}/bin/${P4_PROG}.json -- --grpc-server-addr
   127.0.0.1:50051
```

- configure the switch

```
echo intv8 > /root/t4p4s-switch
```

- stop and disable t4p4s, stop bmv2, enable bmv2:

```
systemctl stop t4p4s | systemctl disable t4p4s
systemctl stop bmv2 | systemctl enable bmv2
```

- start the bmv2 service and check its status:

```
systemctl start bmv2
systemctl status bmv2
```

  Confirm the service is running and the code was successfully compiled before the next step.
- load the static tables into the P4 switch

```
simple_switch_CLI < /root/bmv2/intv8/r3commands.txt
```

A.3 Manual mode

You may stop the *bmv2* service and run manually with the commands:

```
cd /root/bmv2/intv8/
p4c /root/bmv2/intv8/intv8.p4
simple_switch_grpc -i 0@veth0 -i 1@eth0 -i 2@wlx38a28c80c2ee \
  /root/bmv2/intv8/intv8.json -- --grpc-server-addr 127.0.0.1:50051
simple_switch_CLI < /root/bmv2/intv8/r3commands.txt
```

A.4 Debugging mode

If you find issues with the P4 behavior, you may stop the *bmv2* service and run with the logging enabled:

```
cd /root/bmv2/intv8/
p4c /root/bmv2/intv8/intv8.p4
simple_switch -i 0@veth0 -i 1@eth0 /root/bmv2/intv8/intv8.json \
```

```
  --nanolog ipc:///tmp/bm-log.ipc
simple_switch_CLI < /root/bmv2/intv8/r3commands.txt
python3 /usr/lib/python3/dist-packages/nanomsg_client.py
```

A.5 Default routes

The hosts require some default routes or static routing, e.g. in h1:

```
ip route add 10.0.2.0/24 via 10.0.1.1
ip route add 10.0.3.0/24 via 10.0.1.1
ip route add 10.0.4.0/24 via 10.0.1.1
```