FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Duarte Manuel Bento Dias

Attack Framework for SDN Networks and Protocols

Dissertation in the context of the Master in Cybersecurity, advised by Prof. Bruno Sousa and Prof. Tiago Cruz and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

Jan 2023

This page was intentionally left in blank.

"I know of no better life purpose than to perish in attempting the great and the impossible"
Nietzsche

This page was intentionally left in blank.

Acknowledgements

I would like to acknowledge the support of the project POWER (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

I also acknowledge the support of the project MH-SDVanet: Multihomed Software Defined Vehicular Networks (reference PTDC/EEI-COM/5284/2020).

This page was intentionally left in blank.

Abstract

The network paradigm is an ever-evolving technology. From OpenFlow to the more recent P4, the goal has always been to increase programmability whilst keeping the speed of execution. With greater freedom within switches, there is also an increased risk towards its compromise. From a security perspective, few mechanisms have already been designed to protect P4 networks (it is a recent technology), but we are yet to contemplate the idea that the attack may come from within. If the switch itself contains a program that can both perform a regular and hostile activity, it can go unnoticed for a long time. With this thesis, both sides of this problem are explored, first by analysing the corruption potential of P4 and lastly by creating a framework to detect the aforementioned attacks.

Keywords

P4, P4-INT, Software-Defined Networks, Exploitation, Data-Plane, SDN Framework, Grafana Monitoring

This page was intentionally left in blank.

Contents

1 Introduction **1**
1.1 Introduction & Motivation 1
1.2 Objectives 2
1.3 Contributions 2
1.4 Structure 3

2 Context and Related Work **5**
2.1 Historical Context 5
2.1.1 Traditional Networking 5
2.1.2 SDNs and Network Decoupling 6
2.1.3 OpenFlow 6
2.1.4 P4 7
2.2 SDN Architecture 7
2.2.1 SDN Security 8
2.3 Technical background 9
2.3.1 P4 evolution 9
2.3.2 Traditional Switch vs P4 Switch 10
2.3.3 Device Support 10
2.3.4 P4 Language pipeline 10
2.3.5 P4 Language Abstractions 11
2.3.6 Data Storage 16
2.3.7 Operator Precedence 16
2.3.8 Calling Convention (*in/out/inout*) 16
2.3.9 Reading Uninitialized Values 17
2.3.10 Threading 17
2.3.11 Annotations 18
2.3.12 Egress port special actions 18
2.4 Network monitoring and security 21
2.4.1 Network Monitoring 21
2.4.2 In-band Network Telemetry(INT) 22
2.5 Related Work 24
2.5.1 P4 bug exploitation 24
2.5.2 Static application security testing (SAST) 26
2.5.3 Dynamic application security testing (DAST) 29
2.5.4 Network monitorization 31
2.5.5 Other 31
2.5.6 Final Notes 32

3 Attack Development 35
3.1 Attacker Model 35
3.2 Testing environment 36
3.3 Exploit Development 36
3.3.1 Remote trigger 36
3.3.2 Network Scenario 37
3.3.3 Exploit Introduction 41
3.3.4 Traffic re-routing 41
3.3.5 Man in the Middle (MiTM) 44
3.3.6 Denial of Service 47
3.3.7 Combining all the attacks 50
3.4 Evaluation 51
3.4.1 Tools 51
3.4.2 Gauntlet 52
3.4.3 BF4 52
3.4.4 Static Evaluation 52
3.4.5 Conclusions 54

4 Mitigation Framework 57
4.1 P4-INT 57
4.1.1 The selected P4-INT Implementation 57
4.2 Creating a Testbed 58
4.3 Network Topology 59
4.3.1 INT Functionality 60
4.4 Data Collection and Visualization 61
4.5 Adding a controller 62
4.6 Generating Network Traffic 63
4.7 Extracting, processing and understanding the limitations of INT-Metrics 64
4.7.1 Calculating latency 64
4.7.2 Extracting metrics 65
4.7.3 Understanding Metrics 65
4.7.4 Displaying metrics 66
4.8 Results 69
4.8.1 Traffic Re-Routing 69
4.8.2 Man-In-The-Middle 69
4.8.3 Denial of Service (Single Host) 69
4.8.4 Denial of Service (Entire Switch) 70
4.9 Proposing mitigation solutions 70
4.9.1 Implementation example 72
4.9.2 Using the control plane to issue a reset 73
4.9.3 Attack and Mitigation visualization 73

5 Conclusion 77

Appendix A Understanding Mininet in the context of P4 Tutorials 87
A.1 Mininet 87
A.2 Python Mininet 88
A.3 Mininet and P4 88

A.4 BMv2 Architecture 88
A.5 Simulation Files 89
A.5.1 P4 File 89
A.5.2 The Control-Plane file 91
A.5.3 Topology File 92
A.5.4 Mininet Python Script File 94
A.5.5 Makefile 100
A.6 Wrap-up 102

This page was intentionally left in blank.

Acronyms

API Application Programming Interface.

ARP Address Resolution Protocol.

BMv2 Behavioral Model Version 2.

CLI Command Line Interface.

DAST Dynamic Application Security Testing.

DoS Denial of Service.

DPI Deep Packet Inspection.

gRPC Remote Procedure Call.

IDS Intrusion Detection System.

INT In-Band Network Telemetry.

JSON JavaScript Object Notation.

ML Machine Learning.

MLPC Multilayer Perceptron.

NAT Network Address Translation.

OEM Original Equipment Manufacturer.

ONF Open Networking Foundation.

ONOS Open Network Operating System.

P4 Programming Protocol-independent Packet Processors.

p4c P4 Compiler.

PISA Protocol Independent Switch Architecture.

PSA Portable Switch Architecture.

SAST Static application Security Testing.

SEFL Symbolic Execution Friendly Language.

TAP Test Access Port.

TCP User Datagram Protocol.

TCP Transmission Control Protocol.

TTL Time To Live.

List of Figures

2.1 SDN architecture (based on [Cabaj et al., 2014]) 8
2.2 P4 Architecture[Consortium, 2022] 12
2.3 P4-INT modes of execution (based on [Joshi, 2021]) 25

3.1 Remote trigger activation 36
3.2 Network scenario used for the exploits 39
3.3 Exploit 1 step-by-step 42
3.4 Exploit 2 step-by-step 45
3.5 Regular and cloned traffic flow chart 46
3.6 Visual Explanation of Exploit 2 48
3.7 Inner workings of switch 2 49

4.1 Testbed used for testing 59
4.2 Roles of INT in the testbed 61
4.3 Full View of INT applied in a leaf-spine network 62
4.4 ONOS' Graphical User interface 63
4.5 Monitorization panels in Grafana (part 1) 67
4.6 Monitorization panels in Grafana (part 2) 68
4.7 Jitter Plot when the testbed is under a traffic re-routing attack . . . 69
4.8 Effect of using this clone operation at mass 70
4.9 Effect of DoS switch 2 71
4.10 Transit for switch 1 under a DoS attack 71
4.11 Setting up an alert in Grafana 73
4.12 Grafana Alert Triggering 75
4.13 Timeline of an attack to switch 2 75

A.1 Triangle topology representation 93
A.2 Flow execution of the python script 95
A.3 Execution summary and needed files 103

This page was intentionally left in blank.

List of Tables

2.1 Default values for P4 types present in specification 17
2.2 Summary of the state of the art and their contributions 33

3.1 P4 Security tools and their availability 51

This page was intentionally left in blank.

Listings

2.1 $P4_{16}$ Header Example 12
2.2 $P4_{16}$ Parser Example 12
2.3 $P4_{16}$ Table 13
2.4 $P4_{16}$ Action 14
2.5 $P4_{16}$ Control Flow Example 15
2.6 $P4_{16}$ Extern example [Consortium, 2022] 15
2.7 $P4_{16}$ User-defined Metadata 15
2.8 $P4_{16}$ Intrinsic Metadata 16
2.9 Special actions metadata variables 19
2.10 $P4_{16}$ Unwanted drop behaviour prevention 19
2.11 $P4_{16}$ Cloning Example 20
2.12 $P4_{16}$ Metadata Preservation 20
2.13 $P4_{16}$ Resubmission Example 20
2.14 $P4_{16}$ Recirculation Example 20
3.1 Rudimentary remote activation switch 38
3.2 Upgraded remote activation switch 39
3.3 Removing the control block 40
3.4 Reroute control Block 41
3.5 Main control Block 43
3.6 Clone creation block 44
3.7 Clone deviation block 46
3.8 Control Plane entry for clone_t table 46
3.9 Drop control block 47
3.10 Resubmit control Block 50
3.11 Control Block for multiple attacks 50
3.12 BF4 evaluation of exploit 1's code 52
3.13 BF4 evaluation of exploit 2's code 53
3.14 BF4 evaluation of exploit 3's code 53
3.15 Gauntlet's evaluation 54
4.1 INT Table entry 60
4.2 Algorithm for generating network traffic float 63
4.3 Function used to write table entry to the switch 74
4.4 Main body of the reset implementation 74
A.1 $P4_{16}$ Simple Forwarding Example 89
A.2 Control plane file example 91
A.3 Topology File 92
A.4 Link dictionary format 96
A.5 Exercise control flow 96
A.6 Generating the network topology object 96

A.7 Configuring the switches 97
A.8 Configuring topology host and host links 97
A.9 Configuring topology switch links 98
A.10 Programming the Host 98
A.11 Programming the Switch 98
A.12 Instantiating the Mininet Command Line Interface (CLI) 99
A.13 $P4_{16}$ Switch Abstraction Library 100
A.14 Default Switch 100
A.15 Makefile 100
A.16 Short makefile which makes use of the library makefile 102

Chapter 1

Introduction

The introductory chapter provides an overview of the work developed. Most noticeably, this chapter begins with a small introduction, followed by an overview of the objectives, a listing of contributions, the motivation behind the work, and finally, a summarised version of the structure of the document.

Section 1.1 briefly introduces the thesis, mentioning the motivation behind the research.

Section 1.2 refers to the goals for this document.

Section 1.3 outlines the contributions created by this thesis.

Section 1.4 describes the structure taken by the entire document, aimed to help the reader navigate the different topics mentioned.

1.1 Introduction & Motivation

P4, short for Programming Protocol-Independent Packet Processors is a packet programming language developed by Bosshart in 2014 [Bosshart et al., 2014]. This language was created as an effort to efficiently replace OpenFlow by offering solutions to several known issues and challenges, including lack of programmability and protocol independence. By approaching network configuration programmatically P4 allows great flexibility in network management.

Technically speaking, P4 processes bit-streams rather than protocol-specific packets, allowing the administrator to specify protocol parsing programmatically. Regarding technological support, P4 aims to be target-independent by working with multiple hardware manufacturers and by separating the core of the language from its external components, which can be defined by the vendor.

The P4 Language Consortium and the Open Network Foundation currently maintain P4. The ONF is a "community-led non-profit consortium fostering and democratizing innovation in software-defined programmable networks" [Foundation]. Being backed by such an institution ensures its long-term support and

keeps the technology open-source and widely available. P4's current revision is $P4_{16}$ and, at the time of writing of this document, running version 1.2.4 Since the more recent update was distributed after the start of this thesis, the work developed uses $P4_{16}$ running version 1.2.3.

P4 has met the acceptance of both the industry and academia and is predicted to slowly gain more popularity[Liatifis et al., 2023]. It is vital for a language to be bug-free and with well-drawn limitations in order to be viable to professional-grade workloads.

The job of the network administrator usually encompasses the configuration of multiple tools or apps, ensuring proper network configuration. P4, as a network programming language, goes a level deeper in terms of customization, but with the caveat that it requires a greater effort to work with. For this reason, it is not expected that a network admin configures a P4 network from the core, but instead uses a middleware that abstracts P4 in the form of an app.

As such, an attack vector opens, by exploiting the code itself and the lack of analyzing tools available for P4, an attacker can manipulate network behavior. This does not imply that the attacker fully controls the switch, instead [Black and Scott-Hayward, 2021] proved that P4 code can be changed by infecting adjacent libraries. [Dumitru et al., 2020] proved that P4 has some data plane level vulnerabilities which can be exploited.

1.2 Objectives

This thesis' goals are:

- Study and provide an accurate state-of-the-art summary of P4's security landscape.
- Generate data plane exploits using [Black and Scott-Hayward, 2021] as its attack vector.
- Leverage P4-INT as a network visibility tool, understanding its results when applied to the created attacks.
- Proposing mitigation strategies, using P4-INT as a data-gathering framerwork.

1.3 Contributions

Throughout the execution of this task, numerous contributions have emerged.

In the work [Black and Scott-Hayward, 2021], an issue was reported regarding the code, resulting in a change in the repository (`https://github.com/conorblack/AdvExpP4DP/issues/1`).

Appendix A, which was created with the goal of explaining a missing piece in the official P4 tutorial repository [P4Team] was converted to markdown and submitted to the official repository, where it waits merging with the main branch after being reviewed.

Using information collected during this thesis a paper was written and submitted to the to NetSoft 2023 Conference, as a possible PhD Symposium. Unfortunately, the paper was not accepted, but valuable information was extracted from the feedback and used to improve the thesis.

1.4 Structure

This document considers the following structure.

Chapter 2 starts by conducting an analysis of controllable data planes. This takes into consideration the historical perspective, analyzing several technologies. Next, P4 is formally introduced, discussing the most important aspect of the language. Finally, relevant literature in the P4 security landscape is presented.

Chapter 3 is the core of the thesis. In it, exploits are developed, and tested in a simple network scenario. The chapter ends with a formal test on the developed attacks.

Chapter 4 focuses on creating a solid detection strategy, mitigating the described exploits.

Chapter 5 provides a wrap-up for the document. It revises all of the work, connecting the theoretical completion with the implementation and results.

Additionally, Appendix A provides a tutorial on Mininet Framework applied in the context of the P4 tutorial repository.

This page was intentionally left in blank.

Chapter 2

Context and Related Work

This chapter introduces the reader to a variety of concepts related to networking. It serves as a starting point for the thesis, presenting the reader with most of the concepts needed to understand the work developed in later chapters.

Section 2.1 presents a historical perspective of SDN networks. From traditional switches to the newer P4, this section explains the evolution of networks as well as some of their inherent flaws.

Section 2.2 first takes a look at the infrastructures that support SDN. It then takes a look at the SDN security panorama.

Section 2.3 presents the practical aspects of P4 the language, following the official language specification.

Section 2.4.1 discusses the current panorama regarding network monitoring, with a special focus on In-Band Network Telemetry (INT).

Section 2.5 reviews relevant academic material. The main focus of this section is to explore topics of attack (P4 bug finding) and defense (Static application Security Testing (SAST) and Dynamic Application Security Testing (DAST)).

2.1 Historical Context

The next section delves into the historical context of networking, exploring the evolution and milestones that have shaped the field.

2.1.1 Traditional Networking

Historically speaking, networks were designed very differently than in the modern day. The traditional switch was typically closed source, meaning that the Original Equipment Manufacturer (OEM) had full control over said device. Both scholar and network admins were not able to perform changes to the behavior of the switch outside the functionality defined by the manufacturer.

In other words, the consumer was largely limited by the OEM. This was far from an optimal situation. For instance, if the user wanted to create and manage their own protocol, the hardware would not be able to support it

From the manufacturer's standpoint, it is an understandable decision, as it wishes to keep control over its intellectual property. That aside, this decision also lent more control to the OEM, allowing more control over the update cycle, as well as better performance.

2.1.2 SDNs and Network Decoupling

As the internet grew larger, its requirements also changed. In an optimal situation, the network would be able to split packet processing (data plane) from rule generation (control plane), enabling fast dynamic management of the network. Considering a 2-plane separation:

- The **Control plane** contains most of the logic necessary for a switch/router to operate. It handles the logical side of routing, guiding the behavior of a packet that enters the switch. Depending on the implementation, the control plane will be more or less active, nevertheless, most of the heavy lifting is done within this plane (generating network graphs, calculating best routing scenario, etc).
- The **Data plane**, on the other hand, processes the network packets, complying with the rules previously established by the control plane.

Internet decoupling is as old as 2004, being first described in [Yang et al., 2004]. According to the authors, the document "defines the architectural framework for the ForCES (Forwarding and Control Element Separation) network elements, identifies the associated entities and their interactions.". The success of the framework was, however, limited by the community's views on new techniques to control the network.

A few years later, in 2008, a paper [McKeown et al., 2008] promised a technology "based on an Ethernet switch, with an internal flow-table, and a standardized interface to add and remove flow entries.". OpenFlow was an instant hit, as two years after its inception, OpenFlow was already being implemented by Google in their network.

2.1.3 OpenFlow

OpenFlow [McKeown et al., 2008], originally created in 2008, came to revolutionize the networking industry, directly answering the problems related to proprietary software and hardware posed by the manufacturers.

OpenFlow proxied network packets such that, according to their properties, they could be arranged in different flows. This would allow, not only to keep the

"regular" internet working (by having it default to the "old way" of forwarding packets) but also to separate packets into different flows.

2.1.4 P4

OpenFlow represented a big change in networking, nevertheless, it still had its limitations. For instance, what started with 12 fields, was quickly growing up in size, reaching upwards of 40. To accommodate more protocols more fields may be required.

To answer this problem, P4 was created. Instead of adopting well-known and widely implemented protocols, P4 parses incoming bitstream, allowing the administrator to program the behavior of the switch as desired. Even though more complex, adopting this paradigm allows more flexibility.

According to the original P4 document [Bosshart et al., 2014], P4 provides the following properties:

- **Reconfigurability**. Programmers should be able to change the way switches process packets once they are deployed.
- **Protocol independence**. Switches should not be tied to any specific network protocol.
- **Target independence**. Programmers should describe packet-processing functionality independently of the specifics of the underlying hardware.

In short, P4 abstracts the concept of network protocols processing raw bytes, leaving the programmer with the job of parsing said bytes according to their needs.

2.2 SDN Architecture

SDNs are complex and therefore there is a need to formalize architecture to avoid further confusion. This thesis follows the naming convention used in [Cabaj et al., 2014]. According to it, an SDN network consists of 3 + 1 layers (refer to2.1), which are further described below.

The **Application Layer** is the highest layer of abstraction, where the end-user applications such as Intrusion Detection System (IDS), load balancing, and firewalls are implemented.

The **Control Layer** is the level where controllers are situated. It is responsible for managing high-level policies and enforcing them in the data plane.

The **Infrastructure Layer** corresponds to the physical structure of the network. Raw data is processed at this level according to the policies established by the control layer.

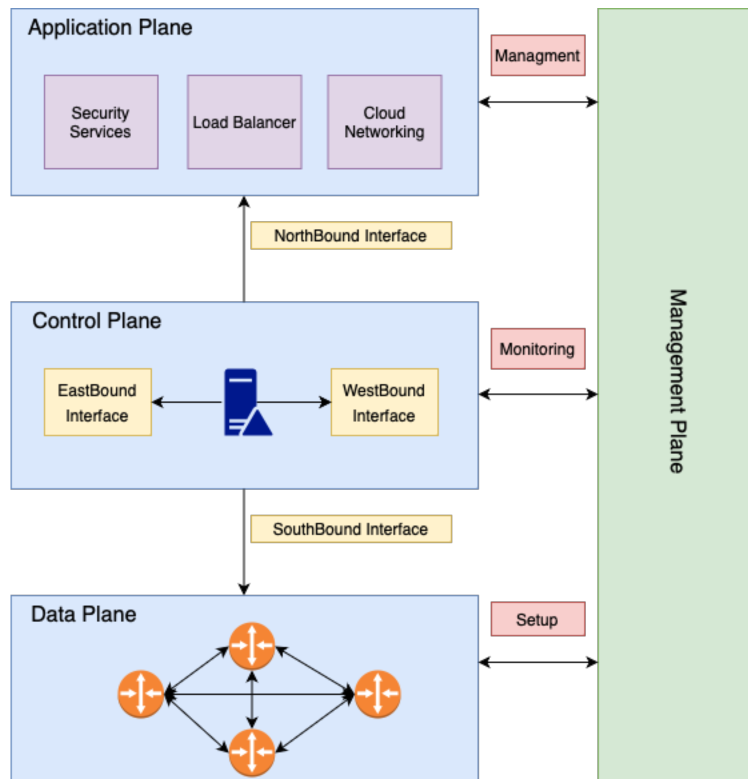

Figure 2.1: SDN architecture (based on [Cabaj et al., 2014])

Additionally, the **Management Layer** is hidden from execution but affects all layers. It represents all controls that are inherently hidden from the network.

Besides architectural layers, interfaces describe can be used to describe the interaction between components of the network. Based on [Alsmadi and Xu, 2015], 4 interfaces can be defined, representing 3 different flows, all centered around the controller.

The **Northbound** interface outlines the flow of communication between the application and control layers.

The **Eastbound and Westbound** interfaces represent the connection between the multiple controllers. Horizontal connection is at the heart of SDNs, allowing dynamic interaction between several network elements.

The **Southbound** interface describes the connection between the controller and data plane switches.

2.2.1 SDN Security

As SDNs became more and more popular, a deeper need for security research was necessary. Still a work in progress, various vulnerabilities have already been identified. Following the architectural model defined above, the SDN security panorama is analyzed.

The management plane can affect all other layers, as such, it is necessary to ensure

that it enforces not only strong passwords but strong access measures as a whole. It is also necessary to ensure that the communication channel is secure.

At the Application layer concerns with API abuse and impersonation should be taken into consideration. Regarding software, it can suffer from information leakage, third-party exploitation, or high-privilege exploitation. "Policy attacks", as defined in [Shaghaghi et al., 2020], define attacks that SDN's ability to define and store network policies. Bad policy management can heavily influence how traffic flows at a lower level

At the control layer, concerns focus on controller spoofing, man-in-the-middle attacks, information disclosure, and network manipulation. The control plane should also ensure that the circulating traffic is, to a degree, isolated from one another, to prevent infections from one network element to spread beyond the patient zero. Configurations must ensure coherency of application coherency and consistency, such that network rules do not contradict one another. In the southbound interface, must secure the communication between the switch and controller should be secure, otherwise, attacks such as man in the middle are possible. This thesis focuses on exploring the southbound interface and its problems.

At a data plane level, DoS attacks are most frequent. Other issues such as communication highjacking, and network manipulation are also possible. Besides DoS, [Gao et al., 2018] also mentions topology poisoning and side-channel attacks, the first consisting in poisoning the information collected by a controller and the second in utilizing the processing time of a control plane to learn network configurations.

2.3 Technical background

The next section explores the technical background of P4, shedding light on the underlying concepts and principles of this programmable data plane language. It was developed using the official manual[Consortium, 2022] (revision $P4_{16}$ version 1.2.3) as a reference as well as the official P4 repository[P4] for examples.

2.3.1 P4 evolution

P4 was originally created in 2014 [Bosshart et al., 2014], introducing its first formal language definition. After the first revision of the language, the old version was denominated $P4_{14}$, whilst the newer version $P4_{16}$.

When comparing both revisions side to side, $P4_{14}$ is a more complex language with a denser core. $P4_{16}$ makes several backward-incompatible changes in an effort to reduce the core of the language to its essential functionalities, decoupling hardware-based functions and constants into libraries.

With the help of a new construct, the **extern** (whose goal is to allow the usage of

the mentioned libraries), $P4_{16}$ contains a stronger and more stable core, leaving the libraries with more room for change

2.3.2 Traditional Switch vs P4 Switch

A traditional switch works independently from the remainder of the network. SDN switches on the other hand use a controller to define their behavior. The controller may be connected to multiple switches and can view the network as a whole, allowing for dynamic changes at runtime. A P4 switch is an SDN switch that uses the P4 technology, according to the specification "configured at initialization time to implement the functionality described by the P4 program" [Consortium, 2022].

2.3.3 Device Support

As mentioned, P4 compromises functionality and ease of implementation. As such, the manufacturer provides two elements:

- A P4 compiler;
- An accompanying architecture definition for the target.

The manufacturer needs not provide any further implementation details in order for P4 to work, maintaining the so-desired anonymity. P4 is supported by several architectures, including but not limited to NetFPGA [Zilberman et al., 2014], BMv2 [Consortium] and Barefoot Tofino [Networks](discontinued by Intel in January 2023).

Noticeably, P4 programs are not expected to be portable across multiple architectures, nevertheless, they should be portable across hardware that runs the same architecture.

2.3.4 P4 Language pipeline

The current version of P4, $P4_{16}$ uses the v1model, which is largely similar to Protocol Independent Switch Architecture (PISA). This is because the legacy version, $P4_{14}$, was designed for PISA, and upon the change to $P4_{16}$, the v1model took ideas from PISA. Figure 2.2 shows the v1 model and its core elements:

- The Parser;
- The Ingress Pipeline;
- The Egress Pipeline;
- The Deparser;

They are further detailed below.

The parser

The parser's objective is to read incoming packets into the system (using the packet_in abstraction) and parse them. Parsing consists in validating headers (better explained in 2.3.5. It also sets standard metadata, creating a number of useful parameters, such as the ingress or egress port.

The Ingress Pipeline

The ingress pipeline contains a match-action pipeline (composed of several match-actions tables) used to modify the header structure. It is in this stage that most of the computations are done. Additionally, re-circulation and cloning are also done in this stage.

The Egress Pipeline

The egress pipeline is everything similar to the ingress pipeline, just differs in the timeline where it is found, which is just before the packet leaves the switch. The egress pipeline may revoke changes made by the ingress pipeline and make changes to the header as well. This stage is necessary since buffering may add some unpredictability to the switching process.

The Deparser

The deparser stage collects all the changes made into the packet, both from the ingress and egress pipelines, assembling them into the final packet, and releasing it to the network.

P4 also supports checksum verification. Whilst not formally defined as stages in processing the checksum verifier ensures that the checksum of arriving packet is valid, otherwise, the packet is discarded. When the packet is leaving the system the Checksum Updater updates the checksum such that changes made to the packet are reflected in the checksum.

Fig 2.2 represents the above-discussed pipeline.

2.3.5 P4 Language Abstractions

P4 is similar to a programming language, using many high-level concepts from other languages. Most similar to C, it noticeably lacks string-related control functions, this is because this functionality is not required for processing network packets. It also lacks loops, ensuring linear complexity for any packet traversing the system (though loops can technically be achieved by using recursion).

Below some core language abstractions are further explained. Most excerpts from this section were retrieved from the official P4 GitHub tutorial section [P4].

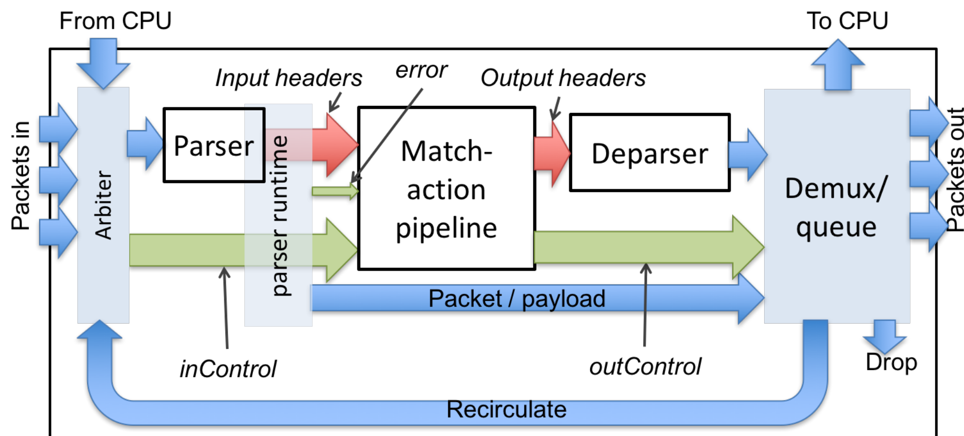

Figure 2.2: P4 Architecture[Consortium, 2022]

Header

The Header "describes the format (the set of fields and their sizes) of each header within a packet" [Consortium, 2022].

A header is similar to a C-struct, with the addition of a hidden validity bit. This bit is used the parsing and emission methods present in P4 to validate packets in and out of the system. 2.1 exemplifies a header.

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}
```

Listing 2.1: $P4_{16}$ Header Example

Parsers

The Parser "describes the permitted sequences of headers within received packets, how to identify those header sequences, and the headers and fields to extract from packets" [Consortium, 2022].

The P4 parser receives the packet using the packet_in primitive and parses the raw stream of bytes into a header. Parsing of a packet always starts with the "state start" block and ends with acceptance or rejection of the packet. The parser should verify all headers used in the ingress and egress pipelines. Without packet parsing, the ingress and egress pipelines may not work properly.

Listing 2.2 shows a sample parser implementation. Its function is to parse IPv4 packets, extracting and validating Ethernet and IPv4 information, respectively.

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {
    state start {
```

```
6           transition parse_ethernet;
7       }
8
9       state parse_ethernet {
10          packet.extract(hdr.ethernet);
11          transition select(hdr.ethernet.etherType) {
12              TYPE_IPV4: parse_ipv4;
13              default: accept;
14          }
15      }
16
17      state parse_ipv4 {
18          packet.extract(hdr.ipv4);
19          transition accept;
20      }
21  }
```

Listing 2.2: $P4_{16}$ Parser Example

Table

The table "associates user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex multi-variable decisions" [Consortium, 2022].

P4 Tables are special control blocks that perform actions based on key matches. There are 3 main elements to a table:

- The key indicates how the match-action comparison is performed. Three default comparison methods are considered:
  - LPM - longest prefix match;
  - Exact - direct comparison;
  - Ternary - matching a table entry where the field has all bit positions “don’t care” [Consortium, 2022];
- The actions define all the possible flows of execution that can be applied to the table.
- The default action decides which action is applied if no match is found.

Below, listing 2.3 illustrates a sample forwarding table, which takes a destination IP as a key and applies an action based on it.

```
1 table ipv4_lpm {
2           key = {
3               hdr.ipv4.dstAddr: lpm;
4           }
5           actions = {
6               ipv4_forward;
7               drop;
8               NoAction;
```

```
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

Listing 2.3: $P4_{16}$ Table

Notes regarding tables:

- Other elements can be added to a table (such as size);
- A table can have multiple keys.

Actions

Actions "are code fragments that describe how packet header fields and metadata are manipulated. Actions can include data, which is supplied by the control plane at runtime" [Consortium, 2022].

Actions are similar functions in programming languages. They represent a control block that is able to modify variables in the code. Listing 2.4 provides a sample forwarding action. It takes several parameters, which are used to change the IP alongside, setting the *egress* port, and the TTL.

```
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }
```

Listing 2.4: $P4_{16}$ Action

Control Flow Expressions

Control flow expressions are situated in the main body of the ingress or egress parsers. They cannot be used inside actions. According the official manual[Consortium, 2022], control flow is able to perform:

- Constructing of lookups keys from packet fields or computed metadata;
- Performing table lookups using the constructed key, choosing an action (including the associated data) to execute;
- Executing the selected action.

2.5 is an example of a control flow. It first checks the validity of the IP header and then executes two different tables, *emcp_group*, and *ecmp_apply*.

```
apply {
        if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 0) {
            ecmp_group.apply();
            ecmp_nhop.apply();
        }
    }
```

Listing 2.5: $P4_{16}$ Control Flow Example

Extern Objects

"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" [Consortium, 2022].

As mentioned in section 2.3.3, P4 is designed target-independent and, as such, its functionality is divided into its core (which is the same in all targets) and architecture-specific functions. The extern primitive defined methods that are not contained within P4 but instead, are contained within the architecture itself.

Listing 2.6 shows *Checksum16*. This method can be declared but is not directly implemented within the code.

```
extern Checksum16 {
    Checksum16();              // constructor
    void clear();              // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing
   checksum
    bit<16> get(); // get the checksum for the data added since
   last clear
}
```

Listing 2.6: $P4_{16}$ Extern example [Consortium, 2022]

User-defined metadata

"User-defined data structures associated with each packet" [Consortium, 2022].

Refers to the temporary storage variables available in a program. Commonly called metadata, it is defined by the user and may have different scopes within the pipeline (local or global). Listing 2.7 illustrates a user-defined variable, which is considered user-defined metadata.

```
struct metadata {
    bit<14> ecmp_select;
}
```

Listing 2.7: $P4_{16}$ User-defined Metadata

Intrinsic Metadata

"Metadata provided by the architecture associated with each packet, e.g., the input port where a packet has been received" [Consortium, 2022].

Usually declared under the *standard_metadata*. Represents data that can be accessed by the user but that is defined by the system. Listing shows*standard_metadata*parameter, contained in a parser definition. This variable is defined and filled by the system.

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata)
```

Listing 2.8: $P4_{16}$ Intrinsic Metadata

2.3.6 Data Storage

P4 defines 2 types of storage elements:

- **Stateless Elements** do not store data in permanent memory and only exist within the execution of each singular packet.
- **Stateful Elements** are global to the switch and maintain state between packet executions. Some examples of permanent storage include counters, meters, and registers.

2.3.7 Operator Precedence

Operator precedence follows the convention that has already been established for other languages, the only few differences include:

- The precedence of the bitwise operators & | and ^ is higher than the precedence of relation operators <, <=, >, >=.
- Concatenation (++) has the same precedence as infix addition.
- Bit-slicing a[m:l] has the same precedence as array indexing (a[i]).

2.3.8 Calling Convention (*in/out/inout*)

P4 defines the readability of variables by using an explicit representation of the read/write properties of the parameters. According to the official language manual [Consortium, 2022]:

- *in* parameters are read-only. It is an error to use an in parameter on the left-hand side of an assignment or to pass it to a callee as a non-in argument. The *in* parameters are initialized by copying the value of the corresponding argument when the invocation is executed.

- *out* parameters are, with a few exceptions listed below, uninitialized and are treated as l-values within the body of the method or function. An argument passed as an *out* parameter must be an l-value; after the execution of the call, the value of the parameter is copied to the corresponding storage location for that l-value. **Note**: any header-related *out* parameters will uninitialized and have their validity bit set to *False* by default.

- *inout* parameters are both in and out. An argument passed as an *inout* parameter must be an l-value.

2.3.9 Reading Uninitialized Values

Default values are of importance in this thesis as some exploits exploit architectural differences when handling default values. For such reason, the defaults values for the Behavioral Model Version 2 (BMv2) architecture are listed below:

Type	**Default Value**
int	0
int<N>	0
bit<N>	0
bool	false
error	error.NoError
string	empty string ("")
varbit<N>	string of 0 bits
enum with underlying type	0
enum without underlying type	first value in declaration
header	invalid
header stack	invalid and nextIndex is 0
header_union	all union elements are invalid
struct	default value per type
tuple	tuple with default value according to type

Table 2.1: Default values for P4 types present in specification

Note As mentioned above, this is only valid for the BMv2 architecture and will may differ in other architectures.

2.3.10 Threading

Concurrency in P4 consists of the usage of multiple simultaneous threads. Each packet is processed individually in its own local pipeline, maximizing throughput, and containing its own local resources (including the packet_in packet_out primitives.)

Global blocks, such as externs, share can be accessed by all threads. This behavior may lead to race conditions and other concurrency undesired behaviors. As such, P4 introduced the **@atomic** annotation, further explained in chapter 2.3.11.

2.3.11 Annotations

P4's annotations aid in the association of metadata with program elements, changing the runtime behavior of the program according to the compiler.

P4 established 2 main types of annotations:

- **Structured Annotations**, which have an optional body;
- **Unstructured Annotations**, which have a mandatory body.

User-created annotations aside, some annotations also are part of the standard. These are characterized by starting with a lowercase letter and defining behaviors specific to the standard library or architecture. Some of the most common are:

- **@optional**, indicates that an extern function, method, or object does not need a parameter specification;
- **@tableonly**, denotes actions that can only be used in a table;
- **@defaultonly**, just like the above, indicates that an action can only be used in a table, as the default action.
- **@name**, user for better API communication;
- **@hidden**, hides an entity from the control plane;
- **@atomic**, imposes the atomic execution of a control block or method;
- **@match**, specifies a other-than-default match_kind value.

2.3.12 Egress port special actions

P4 supports several actions that directly affect the flow of packets in the pipeline. These actions are natively supported by the system and include:

- Drop;
- Clone;
- Resubmission;
- Recirculation;

Since special actions create an abnormal packet flow, to identify such packets P4 uses a metadata variable as an indicator. For the BMv2 architecture, listing 2.9 illustrates the values of such constants.

Note: The remainder of this section uses excerpts from [Fingerhut].

```
const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_NORMAL        = 0;
const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_INGRESS_CLONE = 1;
const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_EGRESS_CLONE  = 2;
const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_COALESCED     = 3;
const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_RECIRC        = 4;
const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_REPLICATION   = 5;
const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT      = 6;
```

Listing 2.9: Special actions metadata variables

Dropping

Dropping is the action of not processing a packet anymore, removing it from the pipeline. This is done in code using the primitive *mark_to_drop()*.

Note: Dropping is only verified at the end of ingress or egress, meaning the *mark_to_drop()* can be overwritten manually. [Dumitru et al., 2020] talks about how this can be used as an exploit, using the term "resuscitated packet". Listing 2.10 illustrates how this behavior can be avoided by manually checking for drops.

```
ipv4_match.apply(); // Match result will go into nextHop if (
    outCtrl.outputPort == DROP_PORT) return;
check_ttl.apply();
if (outCtrl.outputPort == CPU_OUT_PORT) return;
dmac.apply();
if (outCtrl.outputPort == DROP_PORT) return;
smac.apply();
```

Listing 2.10: $P4_{16}$ Unwanted drop behaviour prevention

Cloning

Cloning duplicates a packet, re-ingressing it into a set port. When cloning a packet, both the clone and the original traverse the switch. This action can be called in both the ingress or egress stage, though the packet will only be cloned after the original packet finished traversing its egress pipeline. When cloning a packet several parameters need to be set:

- Clone type:
  - CI2E: defines an ingress-to-egress clone;
  - CE2E: defines an egress-to-egress clone.
- Session ID;
- User-defined metadata to be preserved.

A simple cloning scenario can be seen in listing 2.11.

```
//Simple Clone
clone();
//Preserving metadata
clone_preserving_field_list(CloneType.I2E, I2E_CLONE_SESSION_ID,
   CLONE_FL_1);
```

Listing 2.11: $P4_{16}$ Cloning Example

To clarify how metadata is preserved, 2.12 shows an example:

```
const bit<8> EMPTY_FL    = 0;
const bit<8> RESUB_FL_1  = 1;
const bit<8> CLONE_FL_1  = 2;
const bit<8> RECIRC_FL_1 = 3;

struct meta_t {
    @field_list(RESUB_FL_1, CLONE_FL_1)
    bit<8>  f1;
    @field_list(RECIRC_FL_1)
    bit<16> f2;
    @field_list(CLONE_FL_1)
    bit<8>  f3;
    @field_list(RESUB_FL_1)
    bit<32> f4;
}
```

Listing 2.12: $P4_{16}$ Metadata Preservation

Resubmission

Ressubmissiting can only be called from the ingress pipeline. Resubmitting makes the packet reenter the system. Resubmission can be useful to handle certain protocols which contain multiple layers of headers. Listing 2.13 shows an example of ressubmission.

```
 //Simple resubmit
resubmit();
//Preserving metadata
resubmit_preserving_field_list(RESUB_FL_1);
```

Listing 2.13: $P4_{16}$ Resubmission Example

Recirculation

Recirculation is largely similar to resubmission, only differing in the stage where the packet is recirculated. In this case, the original packet is sent back to reenter the system after traversing both the ingress and egress pipelines. For example, MLPS is a protocol that uses recirculation to remove the topmost label of the packet, regressing it into the switch afterward. Listing 2.14 shows an example of recirculation.

```
//Simple recirculate
recirculate();
```

```
//Preserving Metadata
recirculate_preserving_field_list(RECIRC_FL_1);
```

Listing 2.14: $P4_{16}$ Recirculation Example

2.4 Network monitoring and security

With an increasing number of networks, as well as their sizes and the number of connected devices, it has become mandatory for networks to ensure the safety of their communications. This task is by no means easy, yet network administrators must accomplish it on a daily basis. Below, several techniques for securing networks are described.

Device-related security practices affect the security of the devices in the network, including both network and user equipment. Depending on the characteristics of the network, the administrator may have more or less control over the use of equipment, which influences the measures that can be applied. A simple yet effective method to protect devices is to keep their software and firmware up to date and ensure all logins are properly secured with strong passwords.

When it comes to the network, implementing VLANs can restrict access and contain potential breaches. Segregating network elements can slow down attacks and increase the time necessary for the malware to spread. Third-party software can also be used to secure the network:

- Firewall controls incoming and outgoing traffic by applying predefined rules to filter potential threats.
- Intrusion Detection System (IDS) monitors the network and provides administrators with information.
- Intrusion Prevention System (IPS) builds upon IDS and takes automated decisions based on the network state.

Monitoring the system, whether it's an IDS or some other tool, and creating logs is a great way to understand past complications. Devices should generally have backups to prevent ransomware-like attacks from happening.

Moreover, education should be a central pillar of computer security. This helps to reduce the amount of malware circulating in the network and shorten the time required to apply an incident response plan.

2.4.1 Network Monitoring

As mentioned earlier, the security landscape of networking devices is comprised by many variables. With the goal of this thesis in mind, special emphasis is placed on network monitoring.

Monitoring is a key technique for preventing and mitigating network attacks. By continuously collecting and analyzing system metrics, anomalies can be identified, studied, and responded to quickly.

[Svoboda et al., 2015] describes several monitoring techniques. One technique is Traffic Duplication, which can occur by placing an inline device, such as a Test Access Port (TAP), or by port mirroring. However, duplication can be very costly on the network as it duplicates the circulating throughput. In cases where a physical device is used, it may also be dependent on the capabilities and restrictions of the device.

Packet capture is an alternative to traffic duplication. This approach involves two steps: creating the capture file and analyzing the capture file. While this methodology allows for a deeper search, including some data not present in live captures, it comes at a cost of storage and timely intervention. Furthermore, this workload can become burdensome for large amounts of data.

Deep Packet Inspection (DPI) is similar to packet capture, but it focuses on automated and sophisticated analysis. Implementing DPI is usually more complex than other approaches, and consistent tweaking may be necessary to maintain accurate analysis.

Flow observation, on the other hand, analyzes flows rather than individual packets. According to RFC 7011, "A Flow is defined as a set of packets or frames passing an Observation Point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties." This approach is more scalable because it outputs only a fraction of the monitored data. However, the analysis conducted is much more limited compared to other approaches.

2.4.2 In-band Network Telemetry(INT)

In-Band Network Telemetry (INT) is a lightweight solution for monitoring network devices, first developed by the P4 team. INT uses a "piggybacking" technique where data is added to packets flowing in the network. This methodology drastically reduces the traffic flowing in the network, albeit at the cost of additional processing in the sink and source routers. While INT can't collect information as detailed as in specialized devices, its speed can be very useful for fast detection.[Consortium, 2020]

Network Elements

INT is composed of multiple elements, below the most relevant for this thesis are described:

- **INT Source**. Where the INT packet is created. More specifically, data is prepended to a circulating packet, indicating that it now contains INT information.

- **INT Sink**. Where the INT packet is processed. More specifically, all INT-related data is stripped from the packet and processed.
- **INT Transit Hop**: A trusted entity that collects and reports telemetry for circulating INT packets.
- **INT Metadata**. Telemetry information is added to an INT packet at a source or transit node.
- **INT Header**. Packet header that defines the INT being used (XD, MX, or MD (detailed in section 2.4.2).
- **INT Domain**. A set of interconnected INT modes. Since this architecture makes use of sources, sinks, and transit nodes, there is a need to properly configure a domain so that nodes' functions are properly defined and both metadata and INT header do not leave the domain.

Metrics

According to the official documentation([Consortium, 2020], chapter 4), P4-INT version 2.1 measures the following items (Note: units are not added below, because they depend on vendor implementation of INT):

- **Node ID**: This is administratively assigned to the node and used for identification purposes.
- **Ingress Interface Identifier**: Reports the interface and port used to ingress the packet in the switch.
- **Ingress Timestamp**: Corresponds to the time the INT packet was processed by the egress pipeline.
- **Egress Interface Identifier**: Reports the interface and port used to egress the packet from the switch.
- **Egress Timestamp**: Corresponds to the time the INT packet was processed by the egress pipeline.
- **Hop Latency**: Time required for a packet to be switched within the device.
- **Egress Interface TX Link Utilization**: Lists the usage of the egress port.
- **Queue Occupancy**: Measures the units of traffic in the switch's queue (can be bytes, cells, or packets).

Modes of Operation

There are three variations of INT application modes that can be used depending on the desired level of packet modification and the type of metadata that needs to be collected.

INT-XD (eXport Data) directly exports metadata from the data plane to the monitoring system. This is done based on the INT instructions that are configured at the Flow Watchlists. By using this mode, no packet modification is needed, making it the most size-efficient option.

INT-MX (eMbed instruct(X)ions) embeds INT instructions in the packet header of the INT Source, which are then stripped at the INT Sink. In this mode, all nodes communicate with the Monitor. Packet modification is limited to the instruction header meaning the packet size does not grow as the packet traverses more Transit nodes.

INT-MD (eMbed Data) writes both INT instructions and metadata into the packet. This is the classic hop-by-hop INT, where the INT Source embeds instructions, the INT Source and Transit embed metadata, and the INT Sink strips the instructions processes the collected data and sends it to the monitoring system. The packet is most modified in this mode, but it there there is a reduced number of reports circulating in the network.

Selecting which mode to use is dependent on the scenario and needs of the network, meaning that there is no mode that is in every aspect when compared to other. Fig 2.3 visually explains the modes of execution of P4-INT.

2.5 Related Work

Understanding the state of the art provides a theoretical basis for the thesis, supplying relevant information about security developments in the P4 landscape. To provide a logical division of topics, four categories were considered: bug exploitation, static analysis (SAST), dynamic analysis (DAST), network monitoring.

2.5.1 P4 bug exploitation

Literature related to P4 bug exploitation is made of a mix of both practical and theoretical work. Structurally speaking documents in this category include a small theoretical explanation followed by a larger experimentation portion. In this thesis, this information is used for:

- Understanding the exploits and if they can be exploited using different methods;
- Leveraging the used methods to create new exploits;
- Developing a list of available vulnerabilities to help researchers in future work.

[Agape et al., 2018] offers a "systematic breakdown and approach to study the attack surface and security implications of emerging network architectures". This

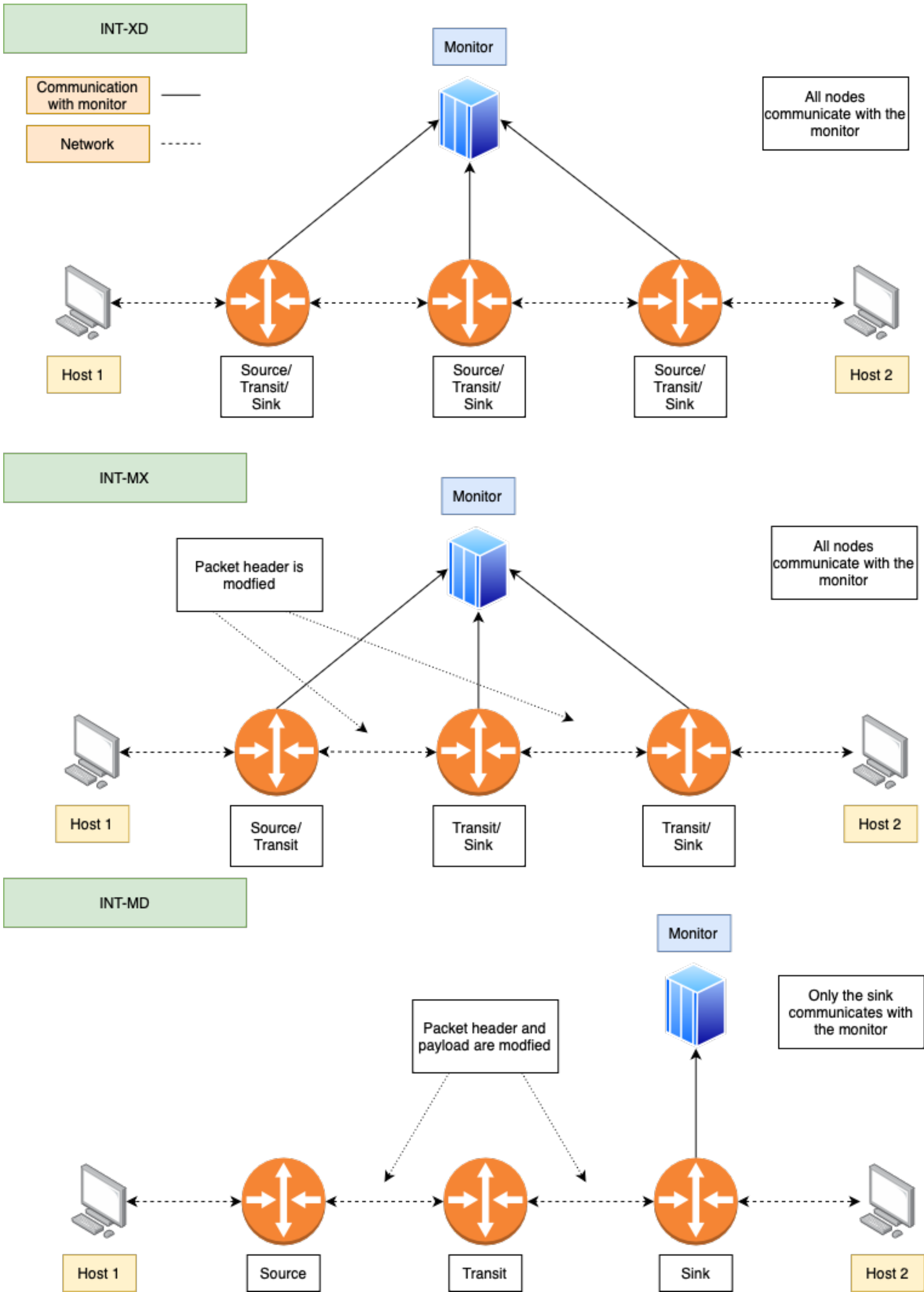

Figure 2.3: P4-INT modes of execution (based on [Joshi, 2021])

breakdown is used to investigate P4's attack surface. Several attacks are mentioned but man-in-the-middle and channel flooding stand out. Besides listing attacks, countermeasures are also detailed.

[Kang et al., 2019] describes an investigation on "sensitivity attacks". This research develops a system that produces an analysis of the behaviors of the source code, predicting malicious traffic patterns. Using probabilistic symbolic execution, all paths possible paths are generated, alongside their probabilities of occurrence. Said probabilities are used to create skew attacks (attacks where packets are crafted in such a way that their behavior is skewed), for which a working example is shown.

[Dumitru et al., 2020] researches P4 bug exploitation regarding architecture-specific undefined behaviors. Three different targets are tested: BMv2, P4-netFPGA, and Barefoot's Tofino (denote that each has a different, vendor-specific P4 implementation). According to the results from testing, the following vulnerabilities were found:

- Reading invalid headers;
- Writing invalid headers
- Infinite loops (Using recirculation and cloning);
- Packet resurrection;
- Implicit forward behavior;

With this information, the authors validate their proposition that P4 behavior is architecture dependent. Furthermore, the authors finish the paper by presenting their own exploits, based on previous results.

[Black and Scott-Hayward, 2021] proposes P4 exploitation in an adversarial manner. The first section is dedicated to discussing the P4 architecture from an adversarial perspective. The core of this work is based on exploiting a side-loaded library, which allows the interception of communications between the controller and switch. Intercepting the communications makes the following attacks viable:

- Manipulating Table Entries;
- Changing the P4 Program;

The paper also describes a novel defense strategy, named ADPv, whose main idea is to "exploit the programmability of the switches to implement a moving target defense that frequently changes the expected behavior of DP switches".

2.5.2 Static application security testing (SAST)

SAST focuses on performing formal verification of a P4 program, mainly using static code analysis. This type of analysis assumes full knowledge (white box)

and attempts to discover vulnerabilities and errors without executing the program. This method provides large coverage, is computationally less expensive than others, and can be used pre-deployment. Its caveats include the limited scope of action (static scope), time for completion, and a high amount of false positives.

[Lopes et al., 2016] focuses of this paper is on verifying the well-formedness of the P4 language, as well the verifying the compilation step. The aim of the investigation is to ensure the well-formedness of the process of serialization and de-serialization packets consistent across multiple implementations from different vendors. To achieve the proposed, P4 is converted to datalog and formally verified using this language.

[Kheradmand and Rosu, 2018] develops a tool, P4K, in an effort to formalize the language. Based on $P4_{14}$ and using the K framework (executable semantic framework [Framework]) the authors are the first to introduce this technique for the P4 language. Formalizing a language allows for its symbolic execution, translation validation (validating the behavior of a language after its architecture-specific compilation), model checking, and cross-program validation.

[Liu et al., 2018] is another effort to formalize the language, now using Guarded Command Language (GCL). Using this abstraction the authors look for potential bugs in areas such as header validity, header stack bounds, arithmetic overflow, and deparsing validity. Whilst easy to check manually, the authors believe creating automatic software can be very useful, especially when large programs are employed. Their implementation includes 8 core steps:

- **Parsing and type checking**. Performs the conversion from P4 to GCL.
- **Instrumentation**. Introduces a "zombie state" into the code that keeps track of information about execution.
- **Inlining**. "Uses a standard inlining algorithm to eliminate procedure calls and generate a GCL command that captures the semantics of the original P4 program."
- **Annotation**. Blends the control plane and data plane into the P4 program.
- **Passivization**. Converts the program into a "passive-form", to improve efficiency.
- **Optimizations**. Applies optimizations such as constant propagation and dead-code elimination.
- **Verifying conditions**. Using the Z3 SMT solver ([Moura and Bjørner, 2008]), the compatibility of the passive and optimized programs is verified.
- **Counter-example generation**. If the verification fails, the program is reconverted back to a human-readable form for manual verification.

To finalize a scalability test is conducted, demonstrating the validity of the approach for real-world scenarios with regard to performance.

[Stoenescu et al., 2018] once again aims to formalize the P4 language, in this case usingSymbolic Execution Friendly Language (SEFL). Using SEFL a tool is created, Vera, which can detect the following bugs in the code:

- Implicit drops;
- Table rules that match dropped packets;
- Invalid memory accesses;
- Header Errors;
- Scoping and unallowed writes;
- Out-of-bounds arrays accesses;
- Fields overflows and underflows;

[Kodeswaran et al., 2020] discusses tracking P4 path execution using the Ball-Larus algorithm. The Ball-Larus algorithm can discover all possible execution paths efficiently (using concurrent track execution for each sub-DAG), having little overhead. With path knowledge, a programmer can more easily identify the various steps in execution and backtrace the execution of packets for debugging and correction purposes.

[Dumitrescu et al., 2020] proposes a tool, bf4, which combines static and dynamic analysis. bf4 consists of the following stages:

1. Finding all the bugs supported by the tool, during compile time;
2. Using this information, controller annotations are inferred, avoiding controller-induced bugs;
3. Assuming most bugs are unreachable (due to the addition of the annotations), bf4 monitors runtime for the introduction of rules which do not respect the previously set annotations.

With this methodology, problems such as an invalid header or out-of-bounds array accesses can easily be detected. bf4 aims to be as automated as possible while keeping a competitive performance (when compared to other similar-purpose tools). A series of programs (including the large *switch.p4* are tested, acknowledging how bf4 successfully detects all intended bugs, furthermore fixing a percentage of the found bugs.

[Tian et al., 2021] produces a work similar to P4v ([Liu et al., 2018] and Vera ([Stoenescu et al., 2018]), under the name of Aquila. According to its creators, Aquila is an improvement to the mentioned frameworks, both in terms of speed and complexity. This paper is highly focused on the testing aspect, having experiments run in Alibaba's very own data center. From the given experiments, several bugs were detected:

- Unexpected program behaviors;
- Incorrect Tables entries;
- Incorrect service-specific properties;
- Wrong call sequence of multi-pipeline.

[Cao et al., 2022] develops a tool, Firebolt, which is characterized as a "black box testing tool designed to dig out faults in DP program generators". Firebolt follows many of the principles discussed in other papers regarding formal language specification and language translation. In more practical terms, Firebolt uses the Z3 theorem prover[Moura and Bjørner, 2008] using a python and C++ framework. This tool focuses on 2 main efforts: security vulnerability checking and intent violation. Within intent violation, an extensive list of bugs was found:

- Incorrect query combination;
- Missing/Incomplete table entries;
- Incorrect mask translation;
- Incorrect list comparer;
- Incorrect comparison operator;
- Missing table entries;
- Missing action parameters;
- Incorrect infinity valuation;
- Incorrect key storage.

2.5.3 Dynamic application security testing (DAST)

To complement the analysis of SAST frameworks, DAST frameworks are evaluated. DAST's goal is to test the runtime portion of the code, presenting a real-world scenario analysis. DAST is, however, not perfect, some of its significant problems include weaker coverage (some parts of code may not be reachable and therefore, never tested), harder automatization, and slower pace (due to the need for manual work).

[Freire et al., 2018] creates a tool, ASSERT-P4, which allows programmers to build assertions into their code, later symbolically executing them. This tool tackles two problems: code circumvention and control configuration. After annotating the original program, ASSERT-P4 performs a conversion from P4 to C, which is then executed by the symbolic engine.

[Shukla et al., 2019] create P4RL, a tool for automatically verifying switches at runtime. This paper also contributes with a novelty language, p4q, which aims

to "conveniently specify the intended properties, using simple conditional statements". This work uses Machine Learning (ML) to consistently generate packets that may cause bugs in the network. Testing the approach proves that P4RL generates better results than a random selection agent, verifying that P4RL can learn patterns of buggy code.

[Shukla et al., 2020] presents P4-Consist, a system to detect inconsistencies between control and data planes in P4 SDNs. In short, this approach contains 4 key modules: (1) Input traffic generator; (2) Data plane module, (3) Control plane module; (4) Analyzer. To check network consistency, this tool generates packets that travel the data plane, gather metadata, and finally return to their origin. After returning, these packets are compared to the expected flow graph, provided by the control plane, resulting in a "real vs expected" comparison. Only if the network is consistent between the control plane and the data plane the comparison is true. The paper ends by empirically validating the approach.

[Ruffy et al., 2020] proposes a tool, Gauntlet, which tests the runtime behavior of different P4 architectures. This tool focuses on finding 2 types of bugs: (1) Crash bugs; (2) Semantics bugs. Two types of testing are employed: (1) Differential testing (same program, 2 different compilers); (2) Metamorphic testing (same compiler, 2 similar programs). Using these methods, if the output is different, there is a guarantee that at least one of the compilers performs an incorrect behavior. Using Gauntlet, the authors found 96 distinct bugs across the P4c framework, which were distributed in the following categories:

- Ripple effects;
- Crashes in the type checker
- Handling side effects;
- Unstable code;
- Consequences of compiler changes;
- Specification changes;
- Invalid transformations.

[Agape et al., 2021] provides a P4Fuzz, a fuzzy logic P4 test-case generator and program tester. P4 fuzz is described by the authors as "a tool that generates syntactically and semantically valid P4 programs that stress the compilers in order to find bugs in their implementation". Built to compile to several architectures, P4Fuzz consists of 2 core mechanisms: (1) Test case generator; (2) Test case tester. The generator creates a program (architecture-specific), to which the tester verifies its compatibility. After guaranteeing that the program can run, using [Nötzli et al., 2018] packets are automatically generated and injected into the created program, searching for possible bugs. Finally using machine learning, all the collected bugs are grouped according to their type. The approach is formally tested, resulting in the discovery of several bugs in all stages of P4, especially in the back end.

[Shukla et al., 2021] produces a P6, a tool to detect, localize, and patch software bugs in P4 programs. P6 utility contains three main modules: (1) The Fuzzer (generates test packets); (2) Localizer (pinpoints the faulty lines of code); (3) Patcher (automates the patching of bugs). As a novelty improvement, P6 uses machine learning to improve code correction (Multilayer Perceptron (MLPC)). This tool correctly identifies the following bugs:

- Accepted wrong checksum;
- Generated wrong checksum;
- Incorrect IP version;
- IP IHL value out of bound;
- IP TotalLen value is too small;
- TTL 0 or 1 is accepted (PI) TTL not decremented;
- Clone not dropped;
- Resubmitted packet not dropped;
- Multicast packet not dropped.

2.5.4 Network monitorization

Network monitoring elicits the state-of-the-art for relevant information regarding network monitoring solutions for P4. In particular, this section focuses on INT as a tool for network visibility.

[Joshi, 2021] describes the implementation of INT) in a P4 network. Initially, INT is described from a theoretical standpoint, exposing the characteristics of the technology including modes of operation, packet structure, and architectural design. The latter part of the thesis tests the approach of a testbed composed of Tofino switches, comparing the different modes of operation.

2.5.5 Other

[Nötzli et al., 2018] is an open-source tool for generating test scenarios. Using well-established symbolic analysis techniques, this tool is able to automatically generate test cases for packets, table entries, and expected paths. By comparing differences between the execution of the same program in two different compiler implementations, one can ensure that at least of the implementations is faulty.

[Dumitrescu et al., 2019] creates a tool, netdiff, which goal is to "use symbolic execution to check the equivalence of two network data planes modeled in SEFL. Using this tool programmers can more easily debug their programs and prevent bugs from entering the network". According to the authors, equivalence can

roughly be described as "for any packet that is injected to any two programs, their output is the same". This piece of work served as the basis for many papers that used symbolic engines.

2.5.6 Final Notes

Programming Protocol-independent Packet Processors (P4) has already been the subject of several quality works regarding its security of data and control planes. From the presented studies, a comprehensive panorama of the P4 security landscape can be extracted. The chapter ends by providing a table (2.2 containing a summary of the related work, highlighting the available contributions provided in the literature.

Name	Category	Notes
[Agape et al., 2018]	Exploitation	Investigates the P4 attack surface.
[Kang et al., 2019]	Exploitation	Investigates malicious traffic patterns.
[Dumitru et al., 2020]	Exploitation	Exploits architecture-specific behaviors on uninitialized headers.
[Black and Scott-Hayward, 2021]	Exploitation	Exploits sideloading and channel. interception attacks.
[Lopes et al., 2016]	SAST	Verifies well-formedness of the compilation step using datalog.
[Kheradmand and Rosu, 2018]	SAST	Formalizes the P4 language using the K Framework.
[Liu et al., 2018]	SAST	Formalizes the P4 language using the GCL.
[Stoenescu et al., 2018]	SAST	Formalizes the P4 language using the SEFL.
[Kodeswaran et al., 2020]	SAST	Path execution tracking uses the Ball-Larus algorithm.
[Tian et al., 2021]	SAST	Improves[Liu et al., 2018] and ([Stoenescu et al., 2018].
[Cao et al., 2022]	SAST	Black box testing tool designed to dig out faults in DP program generators.
[Freire et al., 2018]	DAST	Assertion-based bug detection.
[Shukla et al., 2019]	DAST	A tool for automatically verifying switches at runtime using machine learning.
[Shukla et al., 2020]	DAST	A system to detect inconsistencies between control and data planes in P4 SDNs.
[Ruffy et al., 2020]	DAST	A tool for testing the runtime behavior of different P4 architectures.
[Agape et al., 2021]	DAST	A fuzzy logic P4 test-case generator and program tester.
[Shukla et al., 2021]	DAST	A tool to detect, localize, and patch software bugs in P4 programs.
[Joshi, 2021]	Network Monitoring	P4-INT implementation
[Nötzli et al., 2018]	Other	An open-source tool for generating test scenarios.
[Dumitrescu et al., 2019]	Other	A tool to check the equivalence of two network data planes.

Table 2.2: Summary of the state of the art and their contributions

Chapter 3

Attack Development

Armed with the knowledge gathered in the previous chapters, an attack framework is presented here.

Section 3.2 describes the testbed developed for testing. It defines, structuring, reasoning and implementation.

Section 3.3 presents the reader with the developed exploits. In this section 3 attacks are considered: traffic re-routing (3.3.4), traffic cloning (3.3.5) and Denial of Service (DoS) (3.3.6). These attacks were selected based on the knowledge about the programming language and architecture, as well as the selected attack target (switch functionality, refer to 2.5.1 for further detail).

Section 3.4 evaluated if the exploits developed can be caught using the tool described in the state of the art.

3.1 Attacker Model

To describe an accurate and realistic exploitation scenario, it is necessary to understand the capabilities of the attacker. Since the conducted work builds upon the research developed in [Black and Scott-Hayward, 2021], the attacker model follows is the same. According to [Black and Scott-Hayward, 2021], the attacker model "assumes that the attacker is able to intercept and edit data to/from the controller. In particular, we assume that the attacker can inject their code before calls to the switch SDK or drivers, allowing them to edit the arguments passed to the SDK or driver functions."[Black and Scott-Hayward, 2021].

In specific, the attack "Changing P4 Program - Controller initiated" is used. Assuming this attack is used, the program can be swapped without triggering any errors. Furthermore, since the P4RT server stores a copy of the P4Info File (which indicates the data structures present in the data plane) locally, the controller cannot detect new tables inserted in the code. To ensure that no errors are triggered, and also that the code maintains its original functionality, the attacker should add the exploit in such way that it does not affect original functionality (ensuring all

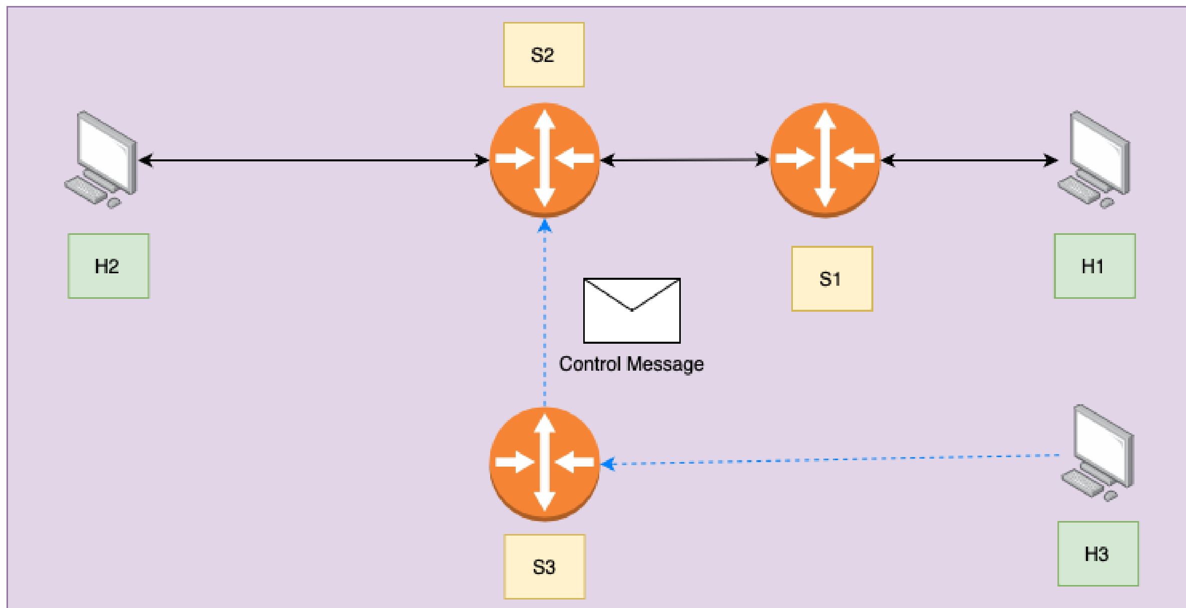

Figure 3.1: Remote trigger activation

control plane calls work correctly). The downside of this method is that it requires the controller to initiate the P4 program change.

Attack Formulation: If the attacker is able to create an exploit within the P4 file, using [Black and Scott-Hayward, 2021] injection tool, he can inject the attack into the switch. Furthermore, using the controller-initiated program change, no logs are triggered. In an optimal scenario, the attack can be remotely triggered, and that does not use the control plane, as the less the moving part (in this case, table entries) the less detectable the attack becomes.

3.2 Testing environment

A virtualized scenario was used to test implementation. Such a tool significantly reduces the time and hassle of implementation, which is vital for researching undeveloped and unimplemented technologies.

3.3 Exploit Development

3.3.1 Remote trigger

A remote trigger is a piece of code allowing the execution of another piece of code. In the context of this thesis P4 exploitation, a remote trigger allows for the exploit to be used at will. Image 3.1 visually explains how the trigger works.

For the trigger to work, it must persist beyond the processing of any singular packet. For this reason, the register data structure is used. As mentioned in 2.3.6,

the register is a stateful storage type. Other such as counters or meters can also be used, but since the register is more common, it also contributes to the stealth factor (many uses of the register are documented in [He et al., 2019]) Listing 3.1 demonstrates a rudimentary switch implemented in P4.

Lines 1 and 2 define the data structures used throughout the code. Next, *set_control* and *unset_control* modify the value of the *myReg* variable. Lines 29 to 40 define the main body of *ingress* control. Lines 30 to 32 ensure *myReg* has a default value and finally line 34 triggers the table which sets or unsets the remote trigger.

The *control_t* table takes the following parameters as key: source IP, IP identification field, and the current value of *myReg*. Source IP and IPv4 Id ensure only the packets crafted by the attacker match the table entries, the temp variable is used as a reference (required to flip the variable's value). For the key, other combinations of fields can also be used, this is especially relevant since some fields may be overwritten by the switch, such as IP or MAC in Network Address Translation (NAT) operations.

Listing 3.1 is verbose-heavy, using a table, multiple control plane entries, and actions. Despite being more flexible it is also easier to be detected. Listing 3.2 limits the number of resources achieving the same goal.

Listing 3.2 trades the ability to use IP ranges, by foregoing the use of tables. Line 8 once again ensures that only the desired packets trigger the mechanism. Lines 9 to 13 verify the current state of the *myReg* variable and change to the opposite.

One major improvement to the listing 3.2 is the creation of a control block to put the malicious code. In itself, the control block does not add any stealth properties to the code. Its main benefit is to remove the code from the main control block, reducing human detection.

In the example above (listing3.3), it is still easy to detect the addition of code. If this code was inserted inside a more realistic program such as *switch.p4* (a production-grade switch p4 program), using a harder-to-detect naming scheme detection would not be trivial.

switch.p4, was originally created for $P4_{14}$ and was later ported to $P4_{16}$. A ported implementation `https://github.com/jafingerhut/p4lang-tests/blob/master/v1.0.3/switch-2017-03-07/out1/switch-translated-to-p4-16.p4`. It contains over 80 tables and 100 actions, totaling over 5000 lines of code. This is not to say that a trained professional keenly looking at the code would not be able to detect a change such as explained in **??**, but chances are greatly reduced if those changes are spread out, and represent less than a 5% change on the original codebase. Section 3.4 researches the feasibility of automatically testing the codebase.

3.3.2 Network Scenario

A triangle topology was used, using 3 switches and 3 hosts, as per figure 3.2. Switch 2, uses an altered version of a p4 program (that contains the exploit), *exploit.p4*, while the remainder runs a basic forwarding program *basic.p4*.

```
register<bit<32>>(128) myReg;
bit<32> temp;
(...)
action set_control(){
      myReg.write((bit<32>)REG_IDX, (bit<32>) REG_VAL);
    }

action unset_control(){
    myReg.write((bit<32>)REG_IDX, (bit<32>) REG_VAL_OTHER);
}
(...)

table control_t {
         key = {
          hdr.ipv4.srcAddr: lpm;
          hdr.ipv4.identification: exact;
          temp: exact;
        }
        actions = {
            NoAction;
            set_control;
            unset_control;
        }
        size = 1024;
      default_action = NoAction();
    }

(...)
 if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 0) {
        myReg.read(temp, (bit<32>) REG_IDX); // Reads the control
   variable
        if (temp != REG_VAL && temp != REG_VAL_OTHER)
            myReg.write((bit<32>) REG_IDX, REG_VAL_OTHER); //Set
   Default Value for the register

        control_t.apply();
        ipv4_lpm.apply(); // Persforms regular Forwarding

        myReg.read(temp, (bit<32>) REG_IDX); // Reads the control
   variable
        if(temp  ==  REG_VAL)
            <malevolent_action>.apply();
    }
```

Listing 3.1: Rudimentary remote activation switch

```
const bit<32> REG_IDX = 0x1;
const bit<32> REG_VAL = 0xFFFFFFFF;
const bit<32> REG_VAL_OTHER = 0xFAAAAAAA;
(...)
register<bit<32>>(128) myReg;
bit<32> temp;
(...)
if (hdr.ipv4.srcAddr == 0x0a000303 && hdr.ipv4.identification ==
    1337){
  myReg.read(temp, (bit<32>) REG_IDX); // Reads the control
    variable
    if(temp  ==  REG_VAL)
      myReg.write((bit<32>) REG_IDX, REG_VAL_OTHER); //Set Default
    Value for the register
    else
  myReg.write((bit<32>)REG_IDX, (bit<32>) REG_VAL);
}
```

Listing 3.2: Upgraded remote activation switch

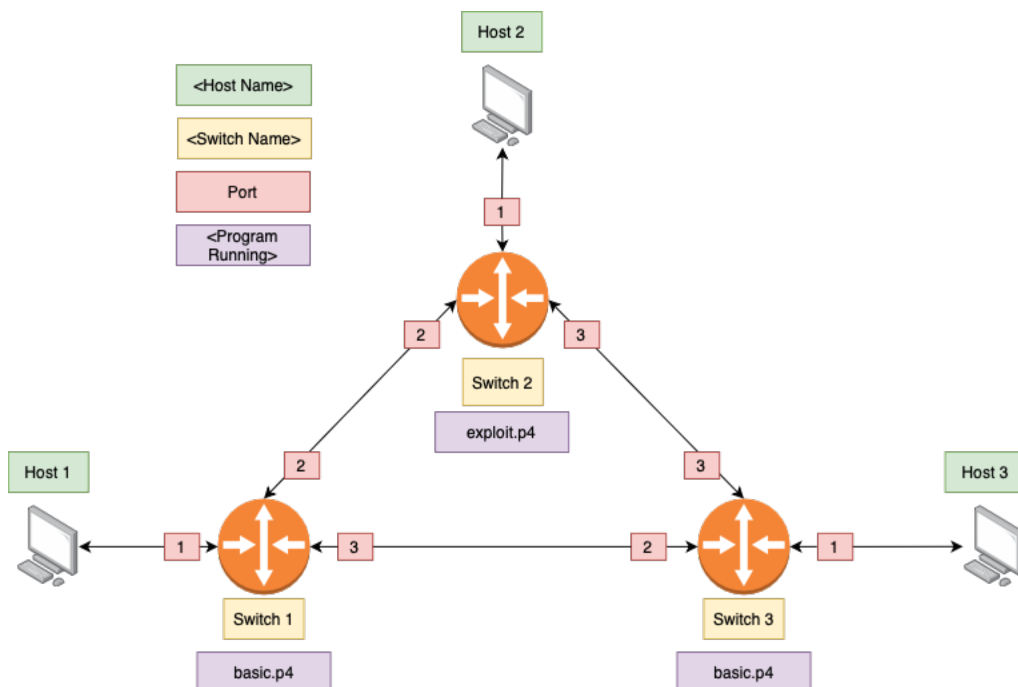

Figure 3.2: Network scenario used for the exploits

```
control Exploit(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata){
     register<bit<32>>(128) myReg;
     bit<32> temp;

     table recirc_t {
       key = {
           hdr.ipv4.srcAddr: lpm;
       }
       actions = {
           NoAction;
           recirc_packet;
         }
         size = 1024;
         default_action = recirc_packet();
       }
     apply {
       if (hdr.ipv4.srcAddr == 0x0a000303 && hdr.ipv4.identification
     == 1337){
           myReg.read(temp, (bit<32>) REG_IDX);
           if(temp  ==  REG_VAL)
           myReg.write((bit<32>) REG_IDX, REG_VAL_OTHER);
           else
             myReg.write((bit<32>)REG_IDX, (bit<32>) REG_VAL);
           }
       }

     }
(...)
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
@name(".Program")Exploit() exploit_0;
(...)
apply {
        if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 0) {
            exploit_0.apply(hdr, meta, standard_metadata);
            ipv4_lpm.apply();
        }
     }
```

Listing 3.3: Removing the control block

```
control Deviate(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata){
    bit<32> regVal;
    apply{
        myReg.read(regVal, (bit<32>) REG_IDX); // Reads the control
    variable
        if(hdr.ipv4.dstAddr == 0x0a000101 && regVal == REG_VAL){
            standard_metadata.egress_spec = 0x3;
            hdr.ethernet.dstAddr = 0x080000000300;
            hdr.ipv4.dstAddr = 0x0a000303;
        }
    }
}
```

Listing 3.4: Reroute control Block

3.3.3 Exploit Introduction

Using the attacker model as defined in 3.3.1, the following attacks are possible:

- **Traffic re-routing**, any traffic that enters the switch can be re-routed to another host;
- **Man-in-the-Middle (MiTM)**, using special egress actions (recall section 2.3.12), traffic is cloned, being received by both the attacker and the original receiver.
- **Denial of Service (DoS)**, severing the connection between switches and hosts is disrupting the service;

3.3.4 Traffic re-routing

Traffic re-routing changes the receiver of the traffic, normally a non-intended destination degrading communication and potentially stealing user data. Image 3.3 visually explains the exploit.

Implementation

To execute this exploit, assuming the usage of the remote trigger mentioned in 3.3.1, the code should be placed at the end of the pipeline, such that it overwrites regular forwarding behavior. According to the previously defined objectives, no extra tables are used. Listing 3.4 shows a control block that changes data whose destination is *10.0.2.2* to *10.0.3.3*.

Using this block, only 4 lines of code are added to the main ingress block, reducing detection chances.

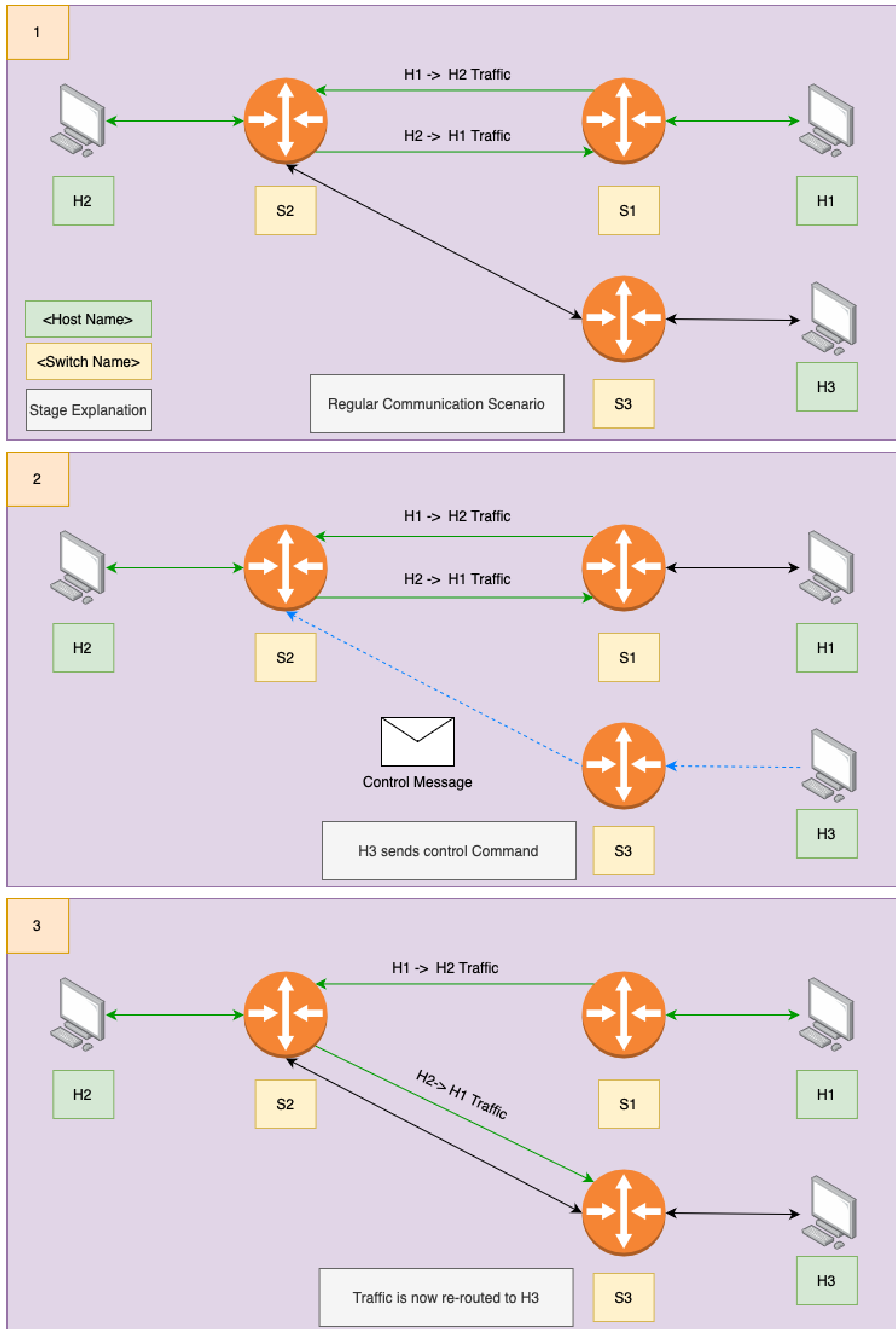

Figure 3.3: Exploit 1 step-by-step

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    @name(".Trigger")Trigger() trigger_0;
    @name(".Deviate")Deviate() deviate_0;

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 0 ) {
            trigger_0.apply(hdr, meta, standard_metadata); // Sets or unsets control.
            ipv4_lpm.apply(); // Performs regular forwarding
            deviate_0.apply(hdr, meta, standard_metadata);
        }
    }
}
```

Listing 3.5: Main control Block

3.3.5 Man in the Middle (MiTM)

MiTM extends the concept of re-routing, cloning the traffic to the attacking host whilst keeping regular traffic flowing. This technique does not raise any alarm to the end user, as traffic arrives normally. Image 3.4 visually explains the exploit.

From an architectural standpoint, figure 3.5 shows how packets are cloned internally by switch 2. The green arrow represents, "regular" traffic while the red arrow represents cloned traffic.

Implementation

The implementation uses the same underlying principle of section 2.3.12. Architecturally speaking, two types of clones are available in P4:*I2E* (ingress-to-egress) and *E2E* (egress-to-egress). Their difference resides in their timing of instantiation (ingress or egress), nevertheless, both are spawned and processed in the egress pipeline.

Three parts make this exploit: The remote switch, as described in section 3.3.1; The Clone instantiation block; The clone deviation block. Noticeably, the deviation block must be placed in the egress pipeline. Programatically speaking, clones are detected using system variable *standard_metadata.instance_type*, where it is defined as the type of packet (for the BMv2 architecture the values are listed in listing 2.9).

Listing 3.6 presents an implementation for the cloning block. The clone operation (line 8) is instantiated using the method *clone_preserving_field_list()*. It takes, as a parameter, the type of clone, the id of the clone, and fields to preserve. Before cloning a packet, 2 conditions are asserted: if the remote trigger is toggled and if the packet is not coming from the attacker (avoiding repeating data).

```
1  control Clone(inout headers hdr, inout metadata meta, inout
       standard_metadata_t standard_metadata){
2      const bit<32> CLONE_ID = 500;
3      bit<32> regVal;
4
5      apply{
6          myReg.read(regVal, (bit<32>) REG_IDX); // Reads the control
       variable
7          if(hdr.ipv4.srcAddr != 0x0a000303 && regVal == REG_VAL)
8          clone_preserving_field_list(CloneType.I2E, CLONE_ID,0);
9      }
10 }
```

Listing 3.6: Clone creation block

Listing 3.7 demonstrates the clone deviation block. This block is similar to the deviation block described in 3.4 with the added caveat that it first checks if the packet is a clone, only deviating such packets (line 6).

Finally, in the egress pipeline, the clone deviation block is first imported (line 5) and applied (line 7).

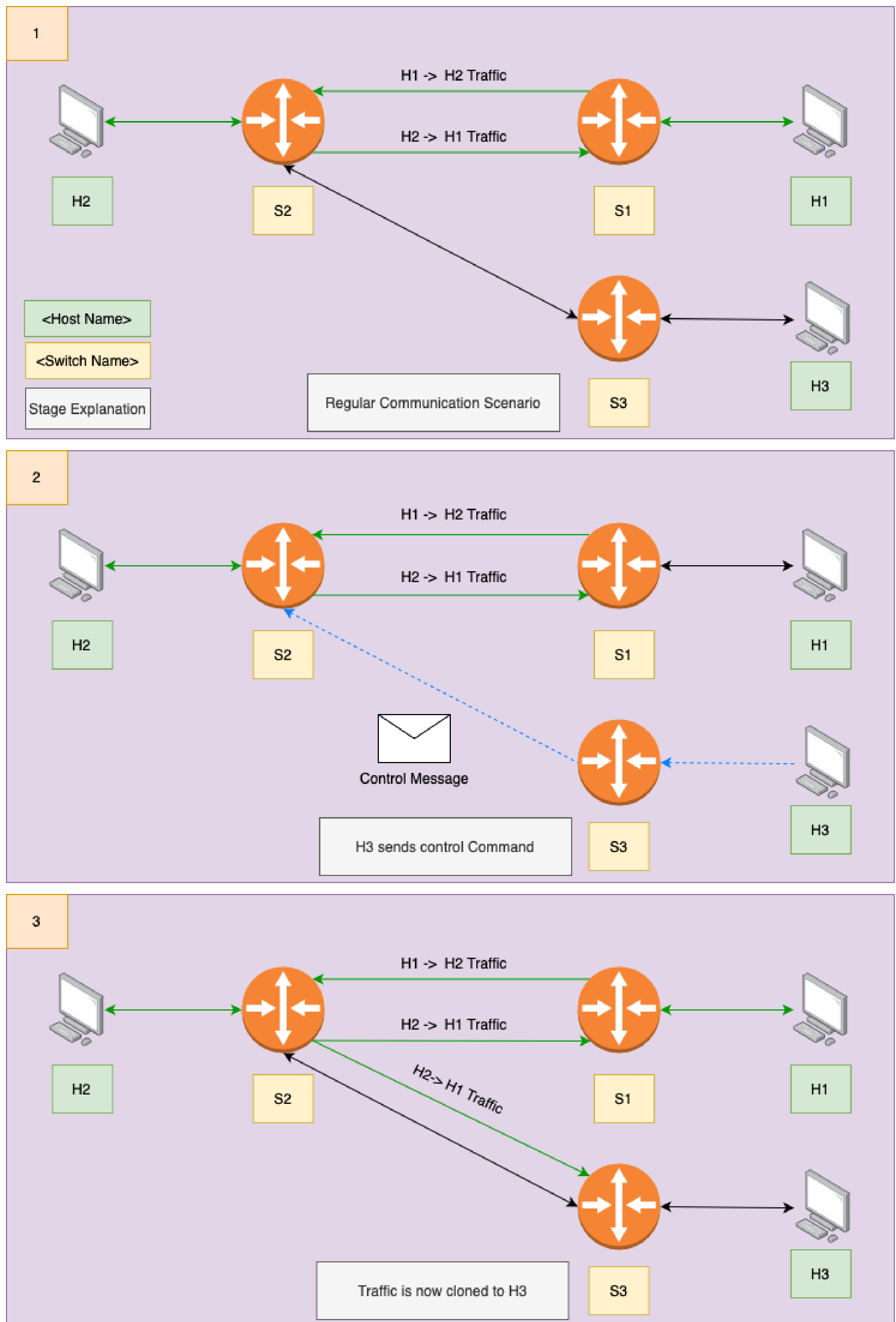

Figure 3.4: Exploit 2 step-by-step

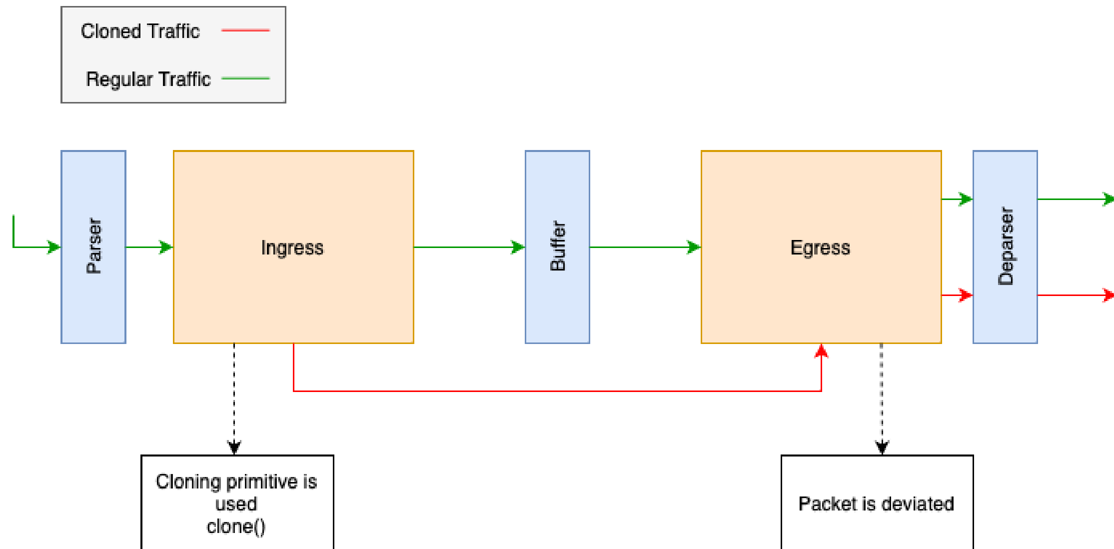

Figure 3.5: Regular and cloned traffic flow chart

```
1 control Deviate_clone(inout headers hdr, inout metadata meta, inout
     standard_metadata_t standard_metadata){
2     bit<32> regVal;
3     const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_INGRESS_CLONE = 1;
4     apply{
5         myReg.read(regVal, (bit<32>) REG_IDX); // Reads the control
     variable
6         if(hdr.ipv4.dstAddr == 0x0a000101 && standard_metadata.
    instance_type == BMV2_V1MODEL_INSTANCE_TYPE_INGRESS_CLONE &&
    regVal == REG_VAL){
7             standard_metadata.egress_spec = 0x3;
8             hdr.ethernet.dstAddr = 0x080000000300;
9             hdr.ipv4.dstAddr = 0x0a000303;
10        }
11    }
12 }
```

Listing 3.7: Clone deviation block

```
1 control MyEgress(inout headers hdr,
2                  inout metadata meta,
3                  inout standard_metadata_t standard_metadata) {
4
5         @name(".deviate_clone")Deviate_clone() deviate_clone_0;
6     apply {
7             deviate_clone_0.apply(hdr, meta, standard_metadata);
8             //Other operations
9      }
10 }
```

Listing 3.8: Control Plane entry for clone_t table

3.3.6 Denial of Service

DoS consists in making machine(s) or resource(s) unavailable. There are several methods to conduct DoS be it on the switch or the end user. DoS attacks have a long history in networking and more advanced networks are usually equipped to deal with such problems. This attack may still be effective if several pieces of equipment in a network are affected since it can be remotely triggered at the same time.

Figure 3.6 shows two attack scenarios. Both start with steps 1 and 2, which represent a regular network with an infected switch and the control server sending a control message to said switch, respectively. Step 3a describes an attack where DoS only happens for a particular host and step 3b describes an attack where the switch just stops responding to any communications (and requires a manual restart).

Figure 3.7 explains the traffic flow inside the switch for both depicted switches. "A" shows how resubmission creates an outer loop, while "B" shows how packets are dropped.

Implementation

DoSing a specific host

It is very simple to create a DoS attack on a specific host. It can be achieved by using the same code base as in the deviation attack (section 3.3.4) but replacing the forwarding action with a drop action. In P4, to drop a packet the *mark_to_drop()* action is used. This action should be put at the end of the processing pipeline, to avoid a bug (mentioned in [Dumitru et al., 2020]) where a packet is resurrected.

Codewise, listing 3.9 presents control that can turn off communications to a host.

```
1 control Drop(inout headers hdr, inout metadata meta, inout
     standard_metadata_t standard_metadata){
2     bit<32> regVal;
3     apply{
4         myReg.read(regVal, (bit<32>) REG_IDX);
5         if(regVal == REG_VAL && hdr.ipv4.dstAddr == 0x0a000101){ \\
     Ensure the remote trigger is on
6             mark_to_drop(standard_metadata);
7         }
8     }
9 }
```

Listing 3.9: Drop control block

DoSing the switch

The goal of this implementation is to make the switch unavailable (and therefore all its connections). In this particular case, this is achieved by resource exhaustion. This is done by applying the resubmit method to packets after they have been processed, immediately sending them back to the entry port of the same switch. This creates an outer loop (because packets from Egress-to-Ingress are technically

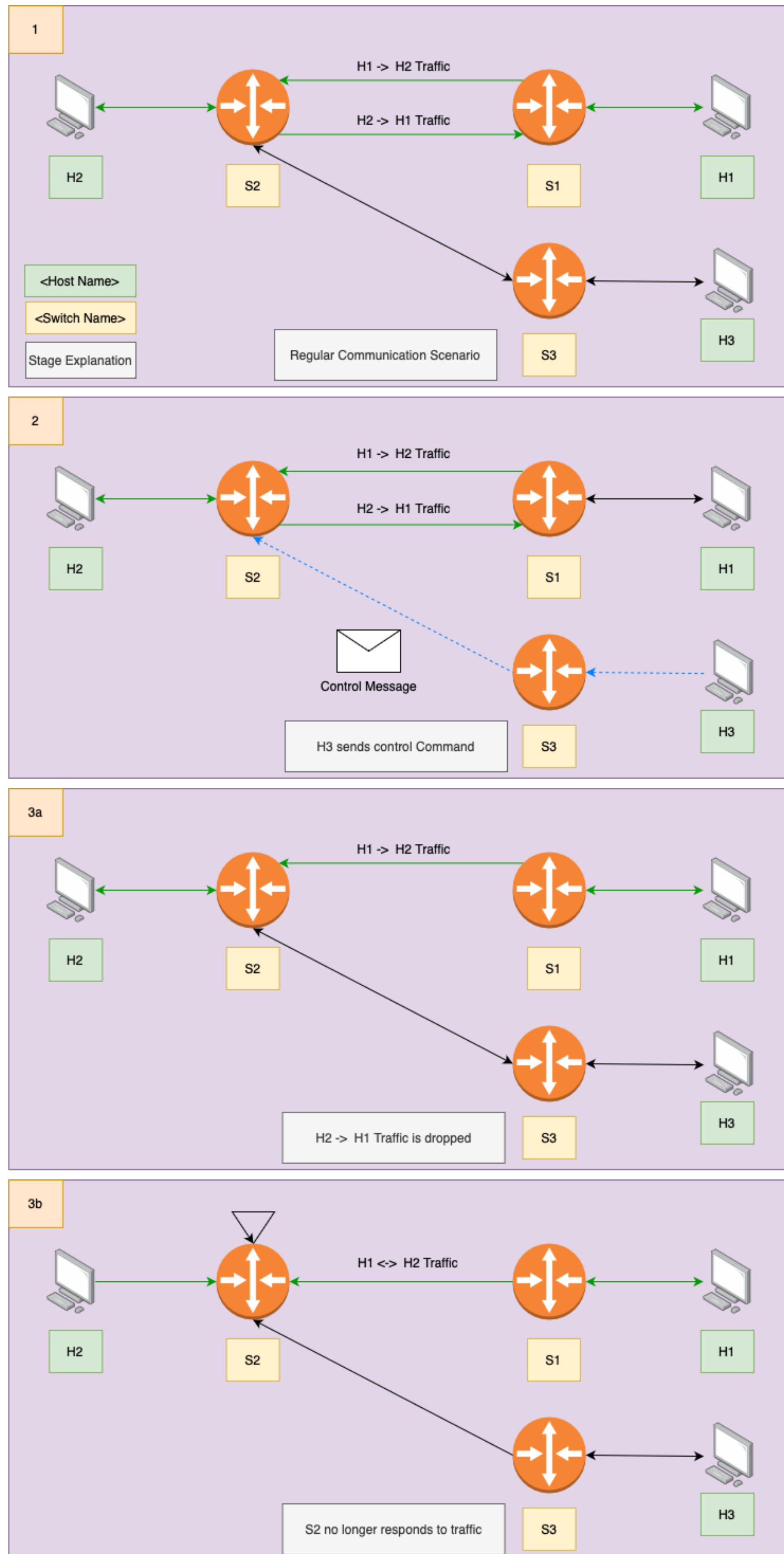

Figure 3.6: Visual Explanation of Exploit 2

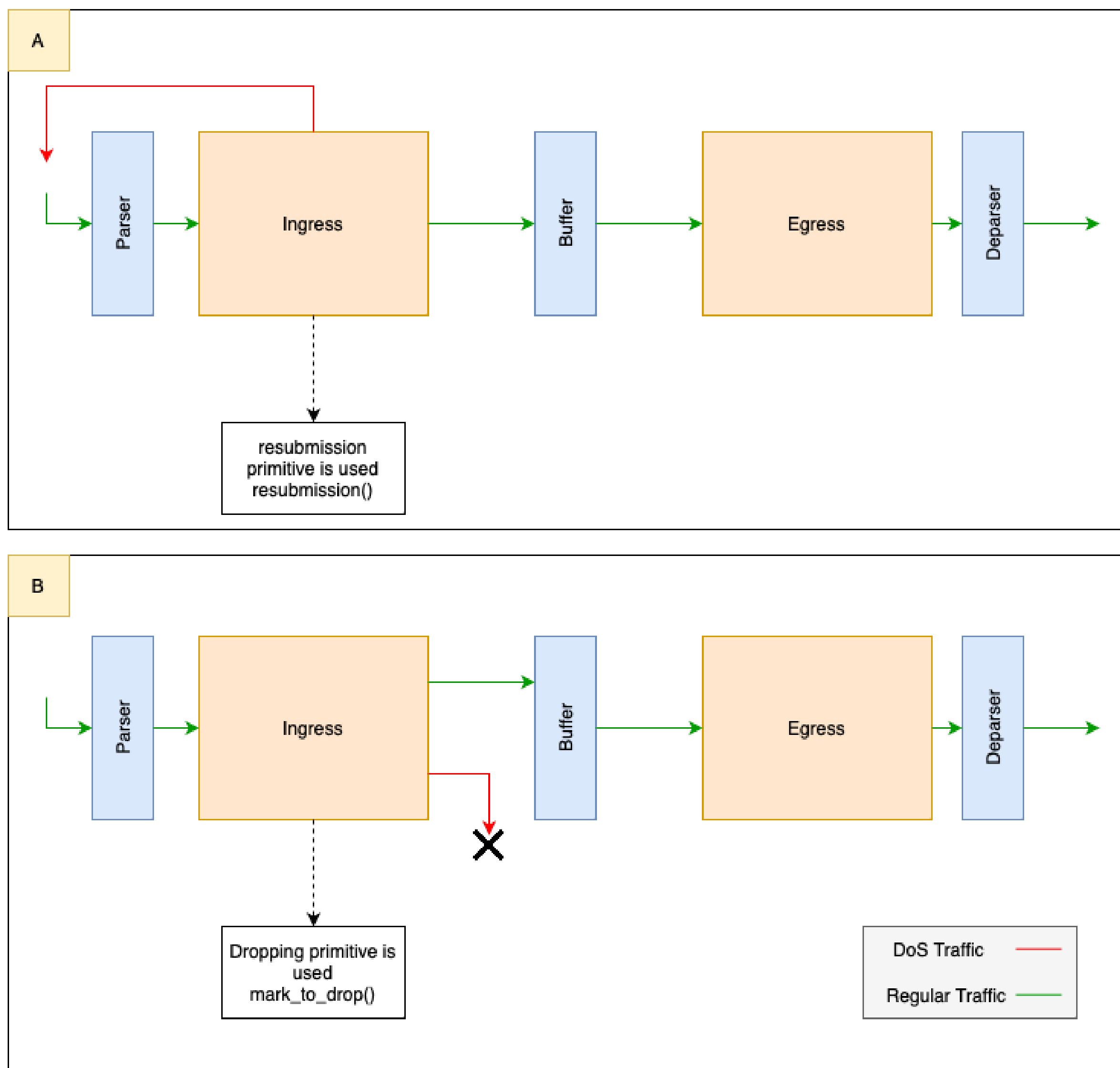

Figure 3.7: Inner workings of switch 2

out of the switch), increasing the CPU usage of the switch to 100%, limiting the acceptance of other packets.

P4 is a linear language, meaning no loops are inside the code. While this proposition is true, using the resubmission primitive (as mentioned in section 2.3.12) an outer loop is created. The loop occurs because the packet leaves the switch, with its destination being the same switch's ingress port, as per figure 3.5. This technique crashes the switch, making it not able to receive any communication before a hard reset.

Using the aforementioned control strategy, a loop can be started on command by having the ingress control block as listed in 3.10. As a note, the attacker must make sure the trigger message is not recirculated, else, the loop stops itself every iteration (and therefore, the additional verification in line 5).

```
1 control Resubmit(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata){
2   bit<32> temp;
3   apply {
4     myReg.read(temp, (bit<32>) REG_IDX); // Reads control variable
5     if(temp  ==  REG_VAL && hdr.ipv4.srcAddr != 0x0a000303) //
    Starts recirculation loop
6         resubmit_preserving_field_list(0);
7     }
8 }
```

Listing 3.10: Resubmit control Block

3.3.7 Combining all the attacks

Throughout this chapter, several techniques have been used to exploit P4's code. Noticeably, all use the same basic remote activation technique, using a trigger as described in section 3.3.1.

As such, it is possible to combine multiple techniques in a single code block, giving flexibility to the attacker while also making the attack harder to diagnose. To combine various techniques inside the P4 code, the remote trigger is changed from a binary variable but to a numeric identifier. Before, 0 would represent no attack, whereas 1 would indicate the code to activate. Now, 1 performs the deviation attack, 2 the cloning attack, and 3 for the DoS attack. Listing 3.11 shows an updated version of the code where the control block can execute multiple exploits. The key difference is that the comparison performed by register now changes variables according to the attack executed (*0xA*, *0xB*,*0xC*, *0xD*).

```
1 control C2(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata){
2   bit<32> regVal;
3   apply{
4       myReg.read(regVal, (bit<32>) REG_IDX);
5       if(regVal == 0xB && hdr.ipv4.dstAddr == 0x0a000101){ //Drop
    Packet
6           mark_to_drop(standard_metadata);
7       }
```

```
8         if(regVal == 0xC && hdr.ipv4.dstAddr == 0x0a000101){
9           standard_metadata.egress_spec = 0x3;
10          hdr.ethernet.dstAddr = 0x080000000300;
11          hdr.ipv4.dstAddr = 0x0a000303;
12        }
13        if(regVal == 0xD && hdr.ipv4.srcAddr != 0x0a000303)
14          clone_preserving_field_list(CloneType.I2E, CLONE_ID,0);
15
16        if(regVal == 0xE && hdr.ipv4.srcAddr != 0x0a000303) { //
      Starts recirculation loop
17            resubmit_preserving_field_list(0);
18        }
19  }
```

Listing 3.11: Control Block for multiple attacks

3.4 Evaluation

This section is dedicated to the study of the available tools and frameworks that provide defense against the attacks described previously.

3.4.1 Tools

According to the research conducted in section 2.5, several tools can be used to detect bugs on P4-enabled switches: P4K, P4V, Vera, Aquila, Firebolt, P4RL, P4-Consist, Gauntlet, P4Fuzz, P6, and BF4; Unfortunately, most of them are either closed-source, only support old syntax-unsupported versions or are outright, not available. To better test the developed exploits, the authors of the above-mentioned frameworks have been contacted but no response was retrieved successfully. Table 3.1 summarizes the availability of security-related P4 contributions.

Tool Name	Contribution
P4K	https://github.com/kframework/p4-semantics
P4V	No response from author
Vera	No response from author
Aquila	No response from author
Firebolt	No response from author
Assert-P4	https://github.com/LucasMFreire/assert-p4
P4ML	https://gitlab.inet.tu-berlin.de/apoorv/P4ML[1]
P4-Consist	https://gitlab.inet.tu-berlin.de/apoorv/P4CONSIST[2]
Gauntlet	https://p4gauntlet.github.io/
P4-Fuzz	Could not find author information
P6	https://gitlab.inet.tu-berlin.de/apoorv/P6[3]

Table 3.1: P4 Security tools and their availability

Of the listed above, Gauntlet and BF4 have been successfully installed and used for testing.

3.4.2 Gauntlet

Per the official P4 repository[Ruffy et al., 2020], "Gauntlet is a set of tools designed to find bugs in programmable data-plane compilers. More precisely, Gauntlet targets the P4 language ecosystem and the P4-16 reference compiler (p4c). The goal is to ensure that a P4 compiler correctly translates a given input P4 program to its target-specific binary.", meaning that the tool does not target contexts where the code may contain rogue behavior. Furthermore, it does not provide an easy way for an analyst to list all system behaviors as the tool merely ensures that translation between the P4 code and binary-specific architecture is successful.

3.4.3 BF4

BF4 is "an analysis backend for P4. It translates P4 code (for the moment V1Model) into a Context Free Grammar (CFG), performs optimization passes, and then converts it into a verification condition which is checked using Z3". It performs a combination of SAST + DAST, but it is not properly directed at security validation and optimization.

3.4.4 Static Evaluation

Evaluation using BF4

There is no standardized method to accurately compare both approaches regarding their security evaluation capabilities. Therefore, to evaluate both tools, the output is analyzed manually. This is a disadvantage since manual evaluation usually takes longer than automated evaluation, yet if the evaluation proves to be accurate, automation should be the next developing step.

Analysis with Bf4 is done by running the command *p4c-analysis*. Analyzing the output for the first exploit(listing 3.5) file yields the result present in 3.12.

```
1  starting frontend
2  /usr/local/share/p4c/p4include/v1model.p4(31): [--Wwarn=unknown]
      warning: Unknown annotation: metadata
3  @metadata @name("standard_metadata")
4   ^^^^^^^^
5  /usr/local/share/p4c/p4include/v1model.p4(59): [--Wwarn=unknown]
      warning: Unknown annotation: alias
6      @alias("queueing_metadata.enq_timestamp")
7       ^^^^^
8  /usr/local/share/p4c/p4include/v1model.p4(442): [--Wwarn=unknown]
      warning: Unknown annotation: pipeline
9  @pipeline
10  ^^^^^^^^
```

```
11 /usr/local/share/p4c/p4include/v1model.p4(460): [--Wwarn=unknown]
    warning: Unknown annotation: deparser
12 @deparser
13  ^^^^^^^^
14 exploit4v2/exploit.p4(157): [--Werror=type-error] error: ipv4_lpm_0
    .apply: Passing 1 arguments when 0 expected
15             ipv4_lpm.apply();
```

Listing 3.12: BF4 evaluation of exploit 1's code

The analysis of 3.12, lists 5 potential issues in the code. The first 4 are related to unknown annotations. Such annotations are related to libraries used in the compilation of the code, and not the created code itself. But why are these annotations flagged? Several reasons can explain this issue. First, P4's documentation does not have a concrete list of annotations, causing confusion with the creators of BF4. Secondly, these annotations may have been added after the last update made to BF4 (with the last feature-relevant update being added in July 2019, where the remainder 6 only contain small bug fixes). The last detection occurs in line 157 and it is related to the method *ipv4_lpm*. Once again it seems to be a system bug rather than a problem in the code. Furthermore a since `basic.p4` file, which is used by the official P4 repository, yields the same result. This indicates that BF4 does not detect any rogue functionality present in the code.

The analysis for exploits 2 and 3 are shown in 3.13 and 3.14 respectively, while exploit 4 does not have any different results and exploit 1.

```
1 exploit2v2/clone.p4(42): [--Wwarn=unknown] warning: Unknown
    annotation: field_list
2     @field_list(0)
3      ^^^^^^^^^^^
4 exploit2v2/clone.p4(121): [--Werror=not-found] error:
    clone_preserving_field_list: Not found declaration
5         clone_preserving_field_list(CloneType.I2E, CLONE_ID,0);
```

Listing 3.13: BF4 evaluation of exploit 2's code

```
1 exploit3v2/exploit.p4(59): [--Wwarn=unknown] warning: Unknown
    annotation: field_list
2     @field_list(0)
3      ^^^^^^^^^^^
4 exploit3v2/exploit.p4(133): [--Werror=not-found] error:
    resubmit_preserving_field_list: Not found declaration
5         resubmit_preserving_field_list(0);
```

Listing 3.14: BF4 evaluation of exploit 3's code

Interestingly, besides the bugs mentioned in 3.12, BF4 also reports the presence of the functions *clone_preserving_field_list*, *resubmit_preserving_field_list* and the annotation for the field list *@field_list(0)*.

According to the information provided in the official P4 repository (`https://github.com/jafingerhut/p4-guide/blob/master/v1model-special-ops/README.md`), "On 2021-Dec-06, the p4c compiler's bmv2 back end with v1model architecture was changed in how you specify what user-defined metadata fields to preserve for the resubmit, recirculate, and clone operations.". BF4 is not prepared

for these changes resulting in a false positive *–Werror=not-found* for the functions *clone_preserving_field_list* and *resubmit_preserving_field_list*. This is almost a bug-turned-feature since the developed code uses this method to perform the exploits. This problem can be avoided by using the legacy functions *clone* and *resubmit*, dropping the ability to preserve metadata but conserving the stealth properties of the attack.

Evaluation using Gauntlet

After thoroughly evaluating Gauntlet's documentation (which can be found in its respective GitHub repository `https://github.com/p4gauntlet/gauntlet`), it becomes clear that Gauntlet is not suitable to perform a security evaluation for the designed exploits.

Gauntlet implements several methods for testing: a fuzz tester, which generates random P4 programs, a translation validator, which compares compiler passes for potential discrepancies, and a model-based tester, which infers the input and output for P4 programs (thus testing the randomly generated programs). Furthermore, the authors of Gauntlet have announced that the tool is currently being ported to C++, making it not feature-complete.

Gauntlet, therefore, was designed with the idea of automatically generating syntactically correct programs, converting said programs to an intermediate language, and testing if said programs would compile correctly. This program covers very specific implementation bugs, which differ completely from the approach of this work. Even so, to ensure the analysis was valid, the translation validation tool provided by Gauntlet was used against the developed exploits. Results are presented in listing 3.15.

```
1 ~/gauntlet/bin/validate_p4_translation ../exploit2v2/exploit.p4
2 Using the compiler binary "/home/tldart/gauntlet/modules/p4c/
    extensions/toz3/validate/../../../../p4c/build/p4test".
3 Analyzing "../exploit2v2/exploit.p4"
4 P4 file did not generate enough passes.
```

Listing 3.15: Gauntlet's evaluation

3.4.5 Conclusions

After extensive research on tools that can evaluate the data-plane portion of P4 code, it can be concluded that a reduced number of tools exist, only a small of them are available, and none detect the kinds of attacks described.

The only two tools that were successfully installed were Gauntlet and BF4. Gauntlet, although created for bug finding, does fit the purpose of the analysis. BF4, on the other hand, presented some interesting results. Because BF4 has not been updated since its release, it does not support the newest methods of cloning and resubmission, flagging them. This makes the tool, obsolete for any code that uses an updated P4 version, but also can be bypassed by attackers because the legacy methods of cloning a resubmission do not affect the end result of the attack.

To conclude, after a thorough analysis of the state of the art, complemented with practical testing of the available tools, it can be concluded that no currently available tool is able to detect the attack by statically analyzing the code.

This page was intentionally left in blank.

Chapter 4

Mitigation Framework

This chapter focuses on applying network-based detection to the attacks created in the previous chapter, as well as leveraging P4-INT as a technique to collect data from the network.

Section 4.1 briefly describes the InBand Network Telemetry technology, referring to possible implementation solutions.

Section **??**

4.1 P4-INT

InBand Network Telemetry (also known as INT) is a technique originally developed for the P4 programming language, with the goal of monitoring network metrics, in a lightweight format. INT is essentially different from other, older, methods of network monitoring. INT does not require extra packets to circulate on the network, as the data required collected circulates, attached to packets currently traveling in the network. Furthermore, there is no need to implement a new protocol, as the data is directly handled by the network devices. INT also achieves greater granularity than traditional monitoring solutions, as it can ensure per-packet granularity.

In summary, INT does not replace traditional solutions, which measure network performance, and security monitoring, amongst other functions. It instead is used to grant visibility over the network and to guarantee fast detection and response to incidents. It is worth noting that INT is a new concept with a lot of room to discover and specific hardware may be required.

4.1.1 The selected P4-INT Implementation

It goes beyond the scope of this thesis to implement INT from scratch, and therefore an implementation of the INT was searched for. There are several available implementations of INT online:

- GEANT's INT [Geantonso]

- ONOS' INT [ONOS, a];

- LaoFan's INT [Fan].

With the options listed above in mind, the implementation used in this work LaoFan's INT implementation. The criteria used to select this work over the others take into consideration a couple of reasons. To use Open Network Operating System (ONOS)' INT implementation it was expected that the testbed used ONOS as a controller, as further detailed in 4.5 this takes is a complex task. Comparing both remaining approaches and taking into consideration that both are incomplete in different aspects, [Fan] is the one updated more recently and therefore contains the more recent version of int (version 2.1, while [Geantonso] implements versions 0.4 and 1.0).

Based on the information found in the repository, the current version only supports User Datagram Protocol (TCP), which is enough for testing but not for real-world implementation. Additionally, from the collected metrics (which go in line with the metrics collected by INT implementation description (check 2.4.2) it does not support egress throughput and queue occupancy.

4.2 Creating a Testbed

For the telemetry monitoring testbed, the initial goal was to use real switches, namely Intel's Tofino switches. Tofino is a specialized architecture developed by Intel that is optimized for use in networking devices, such as switches and routers, designed to provide high performance and low latency.

Unfortunately, the laboratory could not provide the tools required to emulate such a network. In regards to Tofino switches, Intel also halted the production and development[Fool]. Other solutions were also considered, but again, the laboratory could not provide a significant number of switches to run the emulation scenario.

A possibility would be to emulate the switch virtually. The most popular solution using this method is the Open Virtual Switch (OVS). The official website[P4-OvS] mentions that some features are not yet available, including the much important register, without which the attack can't be run.

Without a clear solution that would be feasible to implement the testbed on, Mininet once again was used. This solution is not optimal, but given the constraints shown above and the timeframe to conclude the work, it was the best available.

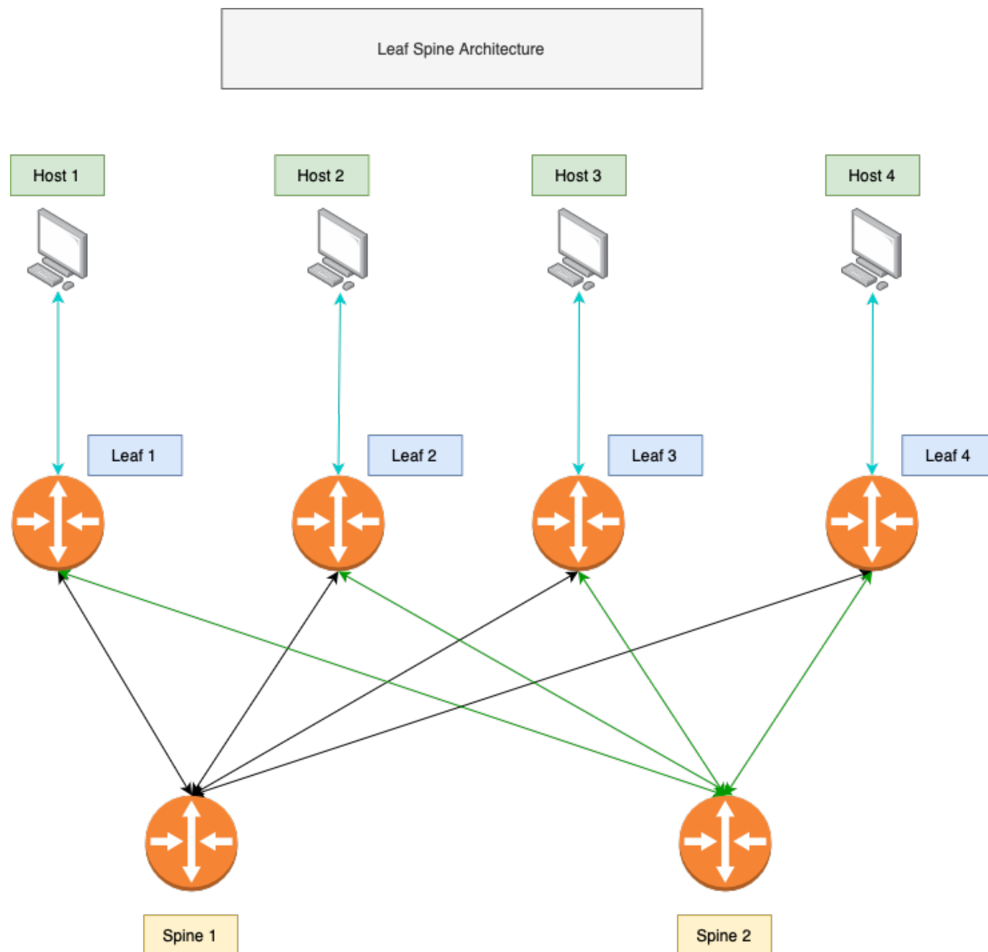

Figure 4.1: Testbed used for testing

4.3 Network Topology

Selecting a network topology is vital to ensure that experiments are relevant to the analysis. After a lot of consideration, a leaf-spine architecture was chosen. A leaf-spine architecture is a network design approach commonly used in data center environments to provide high-bandwidth, low-latency connectivity between devices. This architecture is moderately complex to implement and very scalable. It has also seen a rise in popularity, giving adding to the testbed's effort to emulate reality.

The topology created uses a total of 6 switches, 2 spines, and 4 leaves. It also contains 4 hosts (1 per leaf). Leaf switches are responsible for connecting end-user devices to the network and are typically located at the edge of the network. Spine switches are responsible for connecting the leaf switches together. They are typically located in the core of the network and are responsible for providing high-speed connectivity. Figure 4.1 describes the leaf-spine architecture used

```
{
  "table": "MyIngress.process_int_source.tb_int_source",
  "match": {
    "hdr.ipv4.src_addr": ["10.0.0.0", 4294901760],
    "hdr.ipv4.dst_addr": ["10.0.0.0", 4294901760],
    "local_metadata.l4_src_port": [0,1],
    "local_metadata.l4_dst_port": [0,1]

  },
  "priority": 10,
  "action_name": "MyIngress.process_int_source.int_source",
  "action_params": {
    "hop_metadata_len": 11,
    "remaining_hop_cnt": 10,
    "ins_mask0003": 15,
    "ins_mask0407": 15
  }
    },
```

Listing 4.1: INT Table entry

4.3.1 INT Functionality

LaoFan's implementation of P4 uses INT-MD, as described in section 2.4.2, meaning that each switch should be assigned a functionality (source, sink, and transit). INT packets can only be created in the Source nodes and can only be processed at the Sink nodes.

Transit nodes only add telemetry to the packet. In terms of configuration, Source nodes need to be configured in regard to the flows they are monitoring. This can be done by adding one or more table entries, to the source int table. Listing 4.1 shows an example of a table entry. Lines 4 to 7 define the flows to be monitored. The number 4294901760 (present in lines 4 and 5) converts to OxFFFF0000, and, in combination with the IPs present in the same line, covers all flows ranging from 10.0.1.1 to 10.0.254.254. In terms of action parameters, line 13 refers to the max number of nodes the telemetry data can travel, and line 14 is the current number of traveled nodes (since it is the sink, the number equals the max -1). Finally, lines 15 and 16 define OxF (or 15 if converted to an integer) to define the instruction mask.

Implementation-wise, after the packet reaches the sink, it creates a clone of the original packet: the original packet is stripped of all int-related headers and content and proceeds normally; the cloned packet is processed and sent to the CPU port (a port used for management, control protocols and etc). Using this port is ideal as it removes telemetry and isolates the data gathered from the remaining circulating traffic. Noticeably, it is also necessary that all nodes are provided with an ID.

Regarding the action of forwarding packets on the network, the spines perform Layer-3 routing, while the leaves perform Layer-2 routing. For the sake of simplicity, it is assumed that the L2 routers know the MAC address of the host for a given IP, bypassing the usage of the Address Resolution Protocol (ARP) and ARP

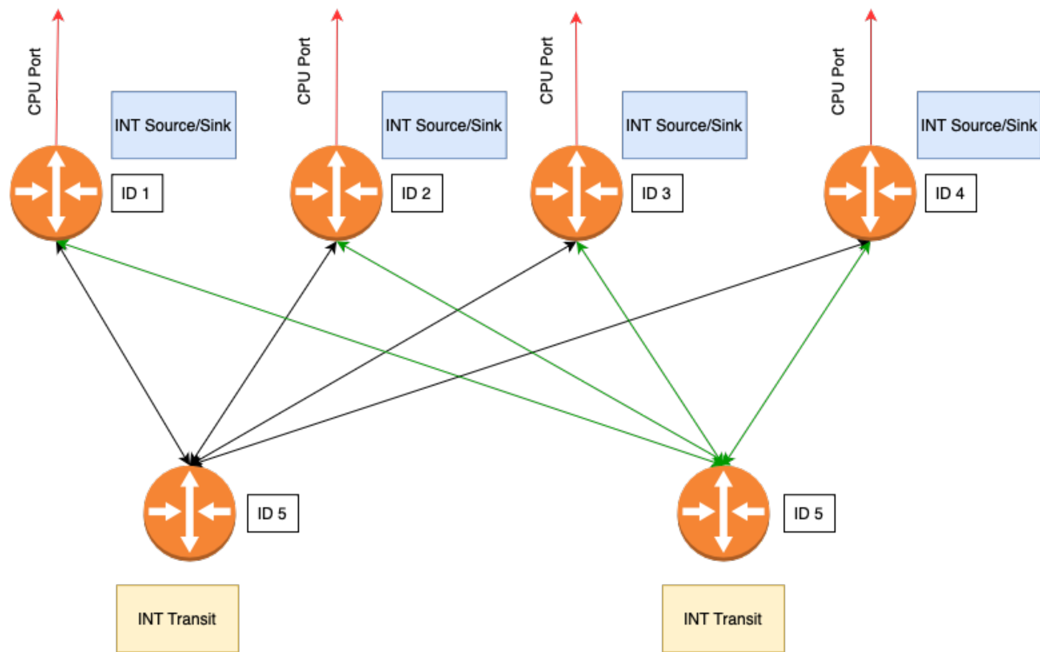

Figure 4.2: Roles of INT in the testbed

Tables.

Figure 4.2 demonstrates the different INT roles in the network.

4.4 Data Collection and Visualization

As mentioned in 2.4.2 INT produces a small set of metrics of the flows present in the network. It is of total interest to the network administrator to collect and display such metrics. Fortunately, LaoFan's implementation already provides a framework that, upon a configuration step, does just this.

As a database solution, InfluxDB[InfluxDB] was selected. InfluxDB is a time-series database commonly used for storing and querying large amounts of time-stamped data, such as network traffic data.

As a display and monitoring solution Grafana was selected. Grafana[Graphana] is a popular open-source platform for data visualization and analysis. It enables users to create customize dashboards displaying data from a range of sources.

The original implementation uses a *Stratum_bmv2* switch, this thesis, on the other hand, uses a more updated *P4RuntimeSwitch*. It is possible to open a CPU port using the *P4RuntimeSwitch* but for simplicity and consistency of implementation (with LaoFan's work), the CPU port is replaced by creating another regular network interface, which does not circulate any traffic besides the allotted telemetry measurements.

To collect the data, the monitor is actively listening to this generated CPU port processing the incoming data, inputting it to the InfluxDB Database.

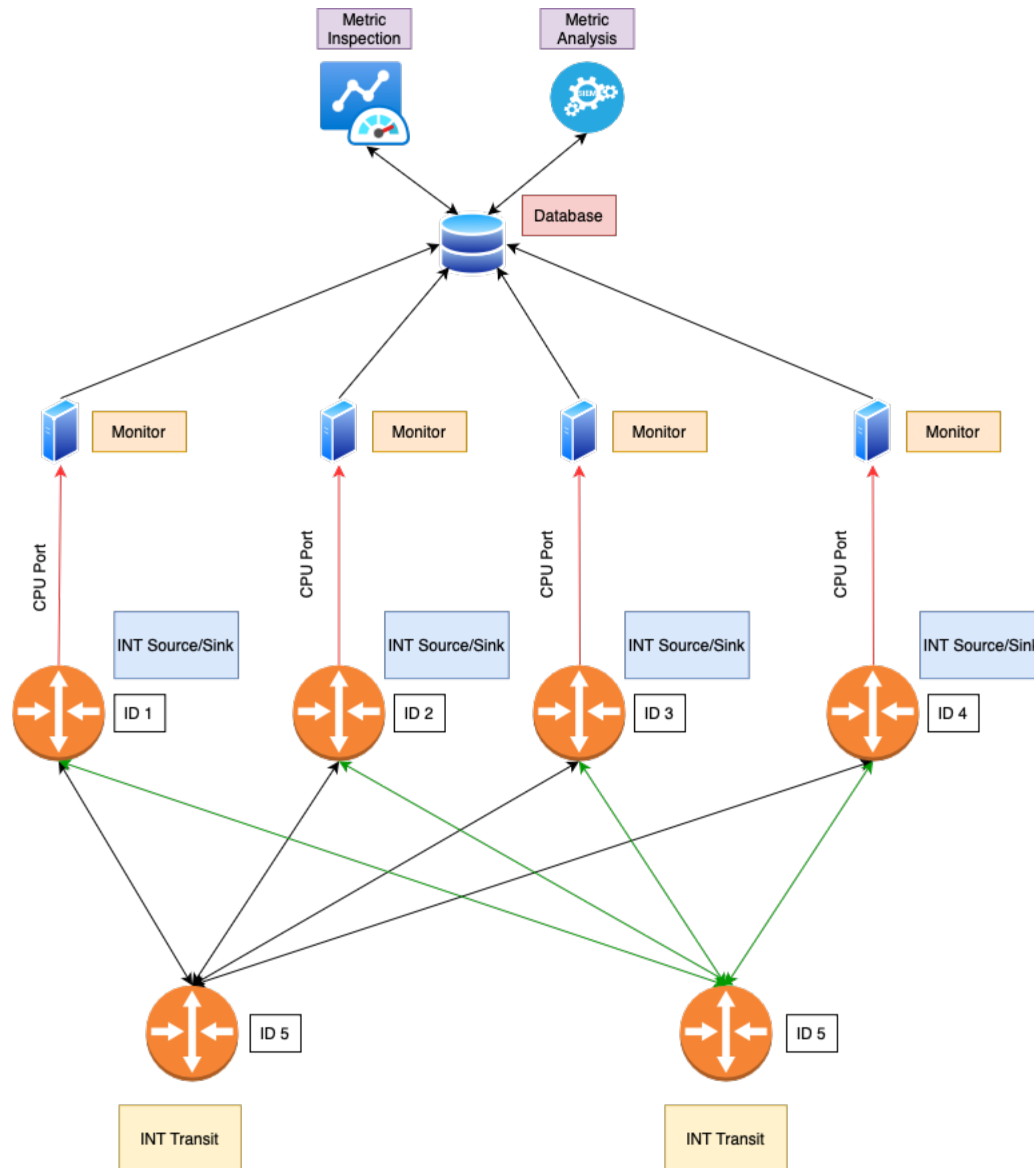

Figure 4.3: Full View of INT applied in a leaf-spine network

Finally, Grafana can be configured to retrieve data from InfluxDB (in which the monitor inputs data) and to display it in a dashboard. Figure 4.3 show the complete panorama regarding the underlying IT framework (function, collection, and display of data).

4.5 Adding a controller

To better simulate a real-world scenario, a controller was added to the simulation. The initial idea was to add the ONOS controller to the simulation. This controller, besides actions related to network management, also possesses a graphical user interface, allowing one to view the topology using an internet browser (such as the one shown in figure 4.4).

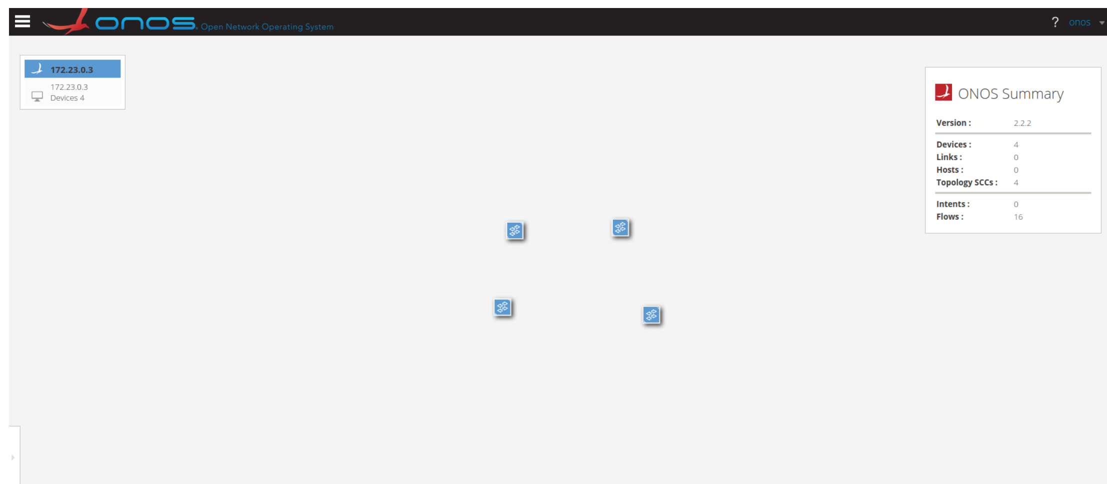

Figure 4.4: ONOS' Graphical User interface

Unfortunately, the documentation is really sparse regarding the installation and configuration of the ONOS controller. Its official website[ONOS, b] is, at the time of writing offline. From the remaining resources found, one working example uses the Stratum_bmv2 switch, which, while also running the P4 language uses different drivers.

For such reasons, the fallback controller used is the P4Runtime Controller. It does not possess a GUI like ONOS but since it directly uses the Python library *p4runtime_lib* it is of much easier management.

4.6 Generating Network Traffic

According to the constraints of the testbed, the network traffic generator, must work with Mininet and generate TCP packets. There are several packet generators available online.

The first intent was to use an established tool for network traffic generation. For this reason, PacketSender[PacketSender] was selected. Upon further experimentation, it was concluded that PacketSender could not be used as it cannot be started inside each Mininet Host. A suitable alternative is Scapy [Scapy], which is a packet library used for packet manipulation. Using this library, a Python script was created, which runs inside every host.

Generating traffic in a realistic manner goes using a Python script goes beyond the scope of this thesis, and for such reason, a simpler approach was taken. First, as mentioned in 4.1.1, since this implementation of INT only works for UDP packets, only such types of packets are generated. The algorithm used is described in4.2 as pseudocode. The goal of this algorithm is to randomly select a receiver for a packet in such a way that after a new receiver is selected the probability of changing receiver linearly increases starting from 0.

```
1 input: list host_list, int bias, int decay, int interval
2 output: None
```

```
3  begin
4      val_bias = bias
5      while True:
6         if random.integer(0,bias) > val_bias:
7              receiver = random.element(host_list)
8              val_bias = bias
9          else:
10             bias = bias - decay
11
12         packet = create_network_packet()
13         response = send_network_packet(packet, receiver)
14         sleep(interval)
15     return None
16 end
```

Listing 4.2: Algorithm for generating network traffic float

4.7 Extracting, processing and understanding the limitations of INT-Metrics

According to the limitations presented in 4.1.1, the following metrics are collected by the system:

- Protocol Type;
- Source IP and Destination IP;
- Source Port and Destination Port;
- Switch ID;
- Queue Occupancy (Queue ID is not supported in Bmv2);
- Ingress and Egress Timestamp (in microseconds since switch start[Martínek et al.], section 4.1);
- Hop Latency, which is the subtraction of the Ingress to Egress Timestamp;

4.7.1 Calculating latency

With the values of all timestamps between destination nodes, should it be possible to calculate the latency of a packet?

No, in fact, packet latency measures the delay end to end, meaning host to host. The tool *ping* calculates latency by sending an ICMP message from the host and waiting for a reply measuring the time taken. INT was not developed in such a way that the switches communicate back and forth. At most, it would be possible to calculate the latency between the first and the last switch, not end to end. But even this task must assume that the clocks between the switches are synchronized.

In our case and according to [Martínek et al.], since BMv2 counts time according to the start of the switch and the switches cannot be ensured to have started at the exact same time, the only latency that can be calculated is hop latency, which is calculated by subtracting timestamps inside the same switch.

4.7.2 Extracting metrics

Even if latency (except hop latency) cannot be calculated, valuable data can still be collected from the timestamps. Instead of describing latency, the timestamps describe Jitter. Jitter refers to the variation in the delay of received packets or data packets' arrival time at the destination and can be calculated despite synchronization problems of timestamps. Since the source of asynchrony relates to the time taken for a switch to start, since the said value is constant, Jitter can be calculated as an absolute difference between two timestamps.

Transit Jitter refers to the variation in latency between the source and sink nodes, and it is calculated by computing the standard deviation of the subtraction between the Egress Timestamp of the sink with the Ingress Timestamp of the Source.

Link Jitter refers to the jitter in the link for any given two nodes, it can be calculated by computing the standard deviation of the subtraction between the Ingress Timestamp of Node 2 with the Egress Timestamp of Node 1.

4.7.3 Understanding Metrics

Now that all collectible metrics have been detailed, their significance is detailed below.

Note: INT is not intended to be the replacement of a Firewall, it is meant to be used as a network visibility tool, providing near real-time information about the circulating packets.

Protocol Type can be used to derive network quality based on the sampling ratio. For example, a network containing a number of ARP packets may have some problem, be it configuration or an attack such as an ARP flood. Furthermore, on a configured network, any circulating packet on a protocol that is not whitelisted such be flagged and investigated.

Queue Occupancy can be used to understand if there are buffering problems in a switch. Paired with **Link Latency**, it works as an indicator of the health of the switch as its circulating packets. If the queue starts increasing it means that there is a problem with the inflow of the packets out of the network, indicating a possible link Exhaustion.

Link Latency refers to the time required to process data inside the switch (Ingress to Egress). If this value increases it means that the switch is taking a longer time to process a packet. This can happen if the rate of packets entering the switch is larger than the switch's processing capabilities, leaving the packet waiting to be processed in the buffer.

Link Jitter refers to the variation or inconsistency in the delay of data transmission across a network link. It specifically relates to the irregularity in the arrival time of packets over a network link. Jitter an is indicator that there is a problem in the connection. High Link Jitter can result in degraded performance on real-time applications, inconsistent data transmission, delay, and impact overall network reliability.

Transit Jitter refers to the variation or inconsistency in the delay of data packets as they traverse through multiple network hops or transit points in a network path.

4.7.4 Displaying metrics

As mentioned before, Grafana was used to display the data collected from INT. To introduce the reader to Grafana's UI, figures 4.5 and 4.6 display the panels used to track the network status. Both figures represent a state where no attack is applied to the network.

Figure 4.5 focuses on real-time metrics of the system, presenting data every few seconds. It is important to notice that a network administrator may not have an infinite amount of screen estate to display metrics and therefore the most relevant should be presented first.

In this figure, there are a total of 4 plots: Plot 1 displays the switch latency in microseconds and it is configured such that if any value goes over a pre-determined limit, the gauge indicator flashes red. It is difficult to extract significance from the value presented beyond the "it's relatively low" since the testing environment (Mininet) may incur some consistency issues; Plot 2 show the progress of the Flow Jitter over time, a view using a time window can help detect patterns present in the flow. Plot 3 shows the Jitter associated with the link in microseconds, plotting a small time series below; Plot 4 shows the queue occupancy per queue in the network. It is only possible to display such a plot because this network contains only 5 queues (since each switch only supports a single queue). Once again if the value goes over a pre-determined limit, the gauge indicator flashes red.

Figure 4.6 shows the second portion of displayed metrics. This figure shows on focusing the overall temporal evolution of the metrics presented above: Link Jitter (plot 5), Queue Occupancy (plot 6), and Switch Latency (plot 7). Using such plots helps in a second analysis if the gauge indicator shows an alarm indication, by analyzing if say behavior is only temporary. Finally, plot 8 displays the quantification of the circulating packets. This metric on its own does not say much about the network, as they do not follow any predictable distribution of packets. Nevertheless, if there is an increase in traffic for a host without a particular reason behind it, it can work as a warning.

Figure 4.5: Monitorization panels in Grafana (part 1)

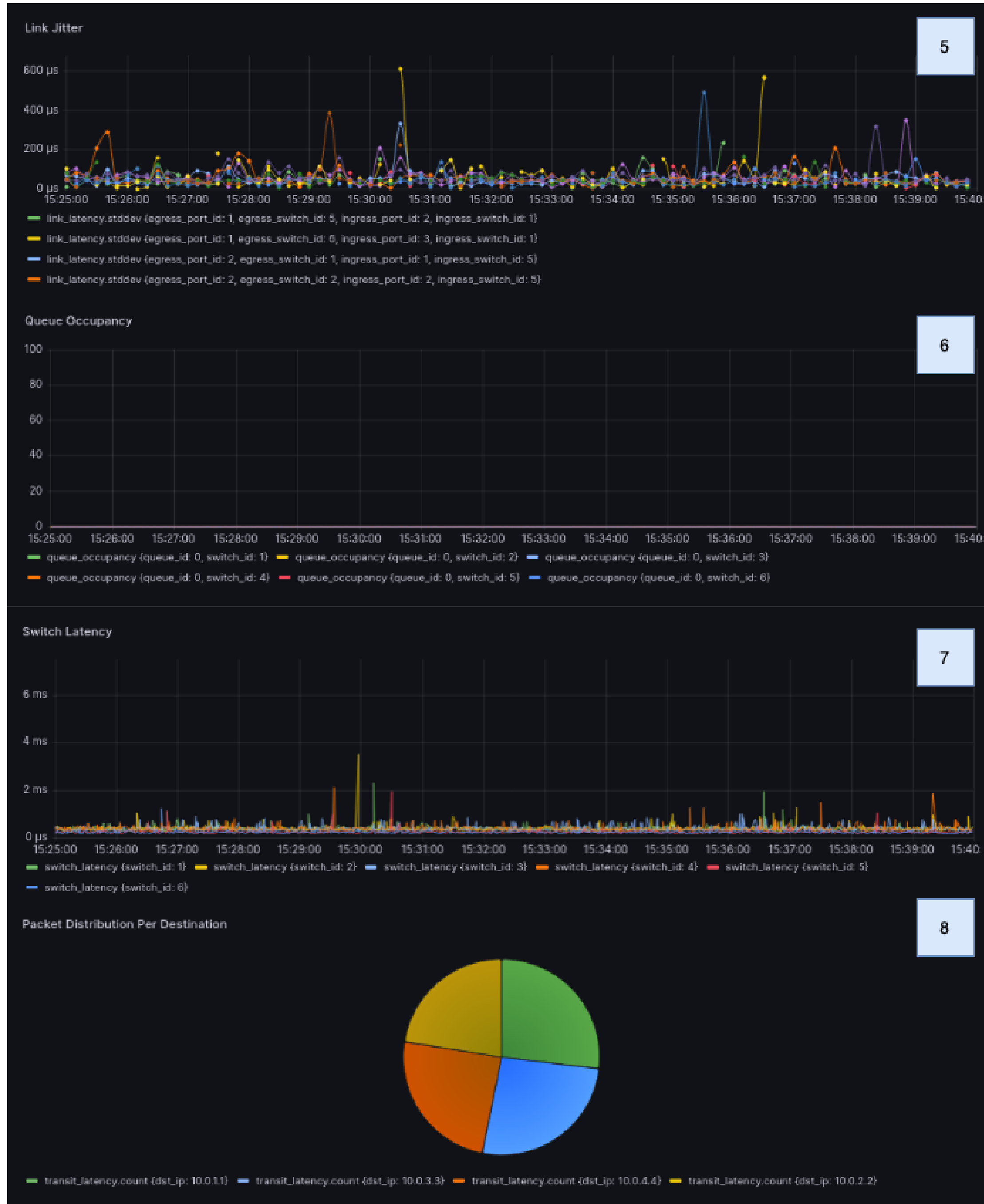

Figure 4.6: Monitorization panels in Grafana (part 2)

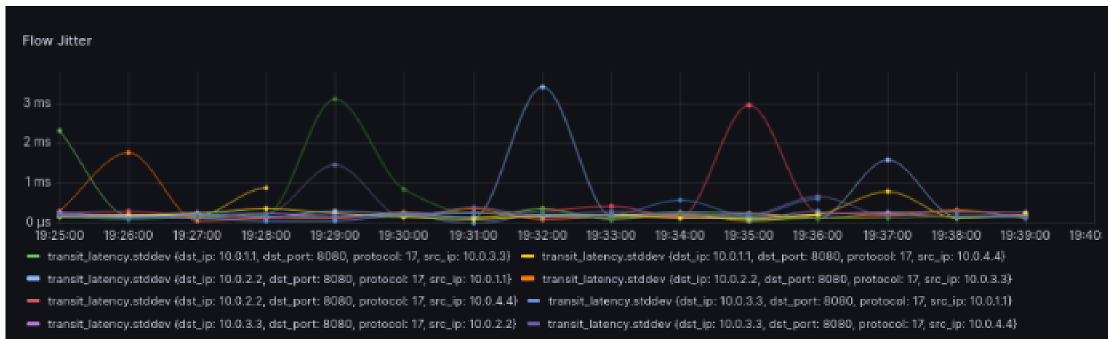

Figure 4.7: Jitter Plot when the testbed is under a traffic re-routing attack

4.8 Results

In this section, the attacks developed in chapter 3 are tested against the INT testbed. Since INT is only a visibility tool that can only collect a limited number of metrics, it is not expected that it can discriminate attacks.

4.8.1 Traffic Re-Routing

As previously explained, this exploit changes reroutes traffic from *10.0.2.2* with destination *10.0.1.1* to *10.0.3.3*.

Using the Flow Jitter Plot as an example (figure 4.7), the data shown by the plots is not enough to infer that an attack is happening in the network. Yet, since INT is configured to track all circulating flows, it could detect a rogue flow to destination *10.0.3.3*. However, this is not scalable, as in a network with thousands of flows, the probability of manual detection is very low.

4.8.2 Man-In-The-Middle

For attack number, the scenario focuses on intercepting data sent by *10.0.2.2* to *10.0.1.1* and cloning those packets to *10.0.3.3*. In this case, there is indicative data that there might be a problem with the system. Figure 4.8, shows the immediate and overall values of Switch Latency when the attack is occurring. This data informs that the cloning operation performed at a large scale in a limited environment such as the one created really takes a toll on the system's resources.

4.8.3 Denial of Service (Single Host)

Attack number 3, creates a denial of service able to shut off the switch completely, disturbing all flows that traverse that switch. This is achieved by recirculating all packets sent to the switch, creating a loop. Figure 4.10 shows the effect of the attack. The green line on the topmost plot represents a regular flow. As can be seen 3 minutes of working regularly, all metrics related to switch 2 (be it Switch

Figure 4.8: Effect of using this clone operation at mass

latency, link, or flow jitter) stop being recorded as the switch is overwhelmed and cannot properly packet to the network (and its respective INT statistic).

4.8.4 Denial of Service (Entire Switch)

The final attack describes an attack that turns off a specific connection, rather than the entire switch. Curiously, this attack is not detected by the INT framework. This is due to the implementation of INT. INT is configured to export metadata according to the packets that enter the switch, not the packets leaving the switch. It means that even if a packet is posteriorly dropped by the switch if it reaches the switch in the first place, it is reported as that. It is concluded then that P4-INT cannot detect a DoS attack on leaf nodes of an infrastructure.

4.9 Proposing mitigation solutions

While not a mitigation solution, INT can be used as a network mitigation measure (note, it should be combined with other tools, for example, a firewall).

As INT as monitoring a visualization tool, INT allows for detail (do not mistake with deep) packet inspection. This means that it is able to visualize all circulating flows (with proper configuration), alongside some of its characteristics, with little added overhead. This behavior can be used to track and report any traffic that is reaching a blacklisted destination. For example, imagine a private network system contains both a firewall and P4-INT, if the firewall is misconfigured and

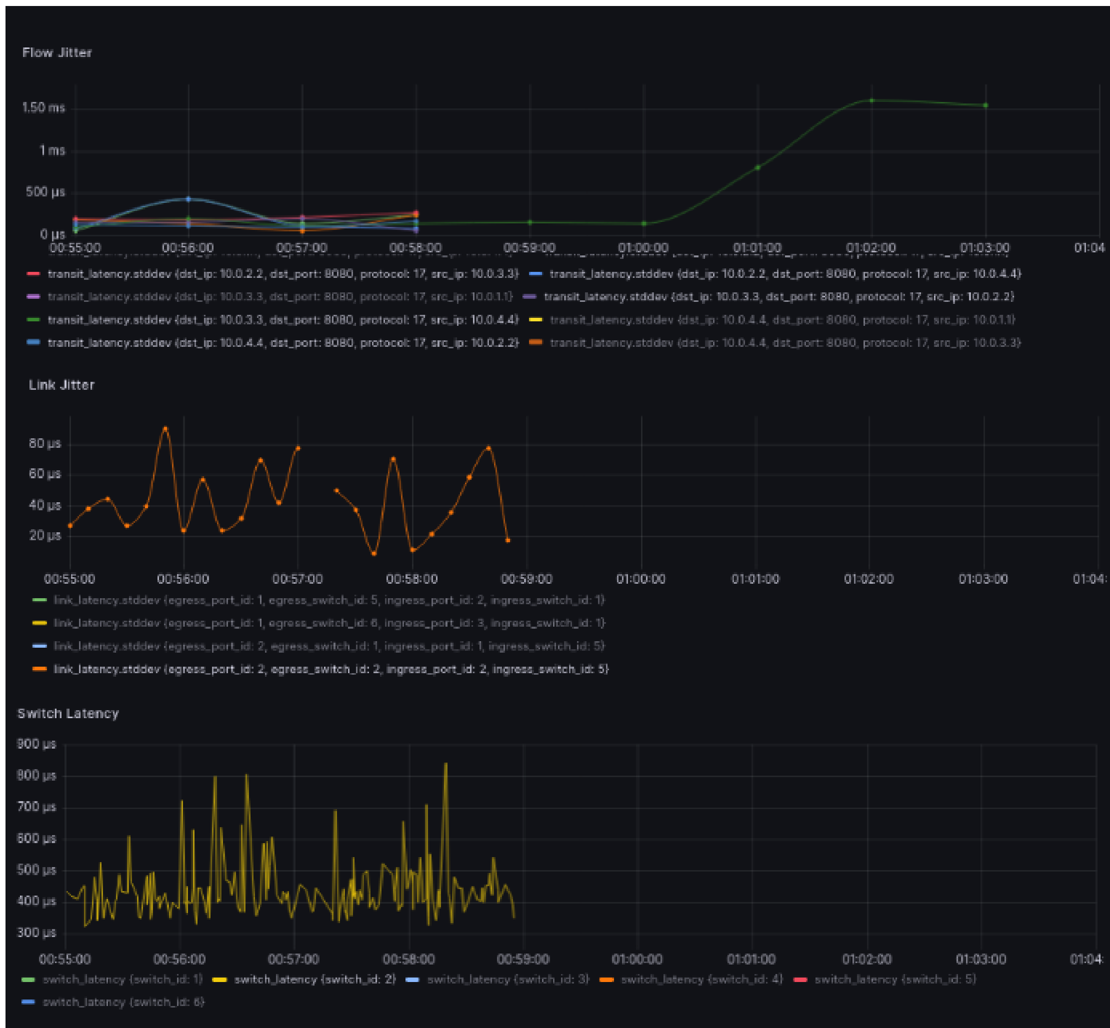

Figure 4.9: Effect of DoS switch 2

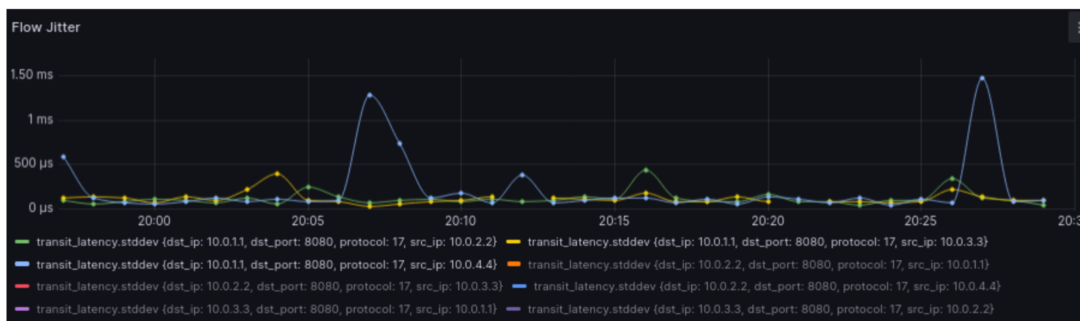

Figure 4.10: Transit for switch 1 under a DoS attack

allows for packets to leave the system, P4-INT will inform the network admin that an outbound connection to a blacklisted location is being made, which will prompt the admin to fix the problem with the firewall.

As a traffic shaping tool INT can be used to measure the circulating traffic in regards to packet number and average throughput (if the metric is available). With this information, manual or automatic decisions can be triggered such that measures are taken to improve the quality of the network. For example, if a link in the network is constantly overloaded, P4-INT can detect such a problem and report it. Such information can be used by a controller to find all available paths and redirect traffic using different links (latency and QoS should be taken into consideration).

As shown in section **??**, even with limited information,INT can be used to detect and report attacks in the network. Its capability of mass processing network metadata in real-time allows the collection and analysis of traffic patterns that show anomalies. An example would be detecting ARP-based DoS attacks, as INT would detect a massive increase in ARP traffic directed at a single host.

Most importantly, INT is able to process and insert network traffic in a database, which can be combined with most tools available, giving great flexibility to admins to set up the analysis and response protocols.

4.9.1 Implementation example

To conclude this section, a practical implementation example is provided. Noticeably, the P4 Runtime Controller Implementation is available as a Python package nothing including many features.

This scenario follows the testbed used for validation of P4-INT metrics (presented in figure 4.3). More specifically, it is running the same scenario as the man-in-the-middle attack (section 4.8.2). In this case, the switch experiences high latency due to the cloning operation employed by the attack. To fix this problem, the switch is first restored to factory settings, followed by a recovery of the control plane rules. While this solution may be considered extreme, it can be useful in cases where the network administrator does not understand the root cause of the faulty switch. In such cases, it might be faster to factory restore the device rather than diagnose the problem. (Note that this solution does not require the switch to be restarted, reducing the amount of time needed to conclude the operation.)

Setting up a Grafana alert

The process of fixing this problem, is a human-in-the-loop scenario, meaning a human decision is taken in the process, rather than being an automated process. Therefore, the first step is to set up a Grafana alert, this alert notifies the admin that there is a problem in the network and action must be taken.

Figure **??**, details the configuration page for a Grafana Alert. The image shows

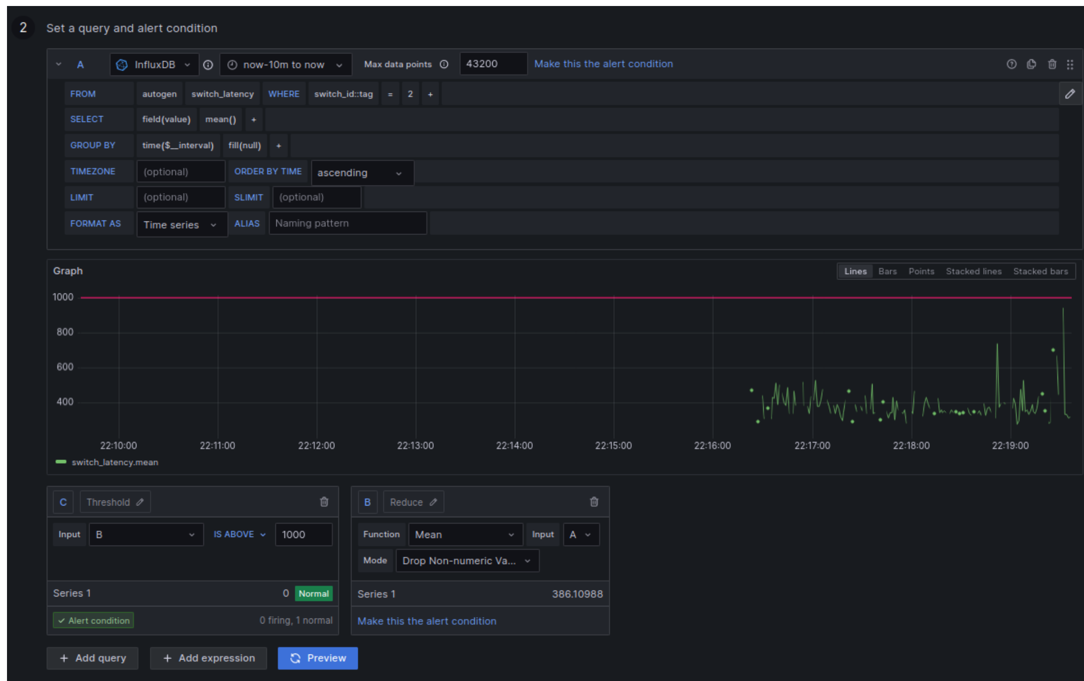

Figure 4.11: Setting up an alert in Grafana

the alert condition as well as a visual representation of the threshold line. The alert is configured to notify the administrator if the latency of switch 2 gets over 1000 microseconds (1 millisecond).

4.9.2 Using the control plane to issue a reset

After analyzing the problem in the network there are several possible solutions according to the situation. In this example, the admin elects to reset the switch and re-insert the control rules. First, listing 4.3 shows two functions, *bulk_write* takes in an entry list separates then one by one. Function *writeTunnelRules* parses a table entry into a structure which can be understood by the switch and sends it to the respective pipeline.

Listing 4.4 first loads the configuration file (line) and then P4Infofile (line). It then tries to establish and connection with the device (lines x-x) and install the P4 program (line). Finally it call the *bulk_write* method.

4.9.3 Attack and Mitigation visualization

To conclude, the flow of the attack is described below. Figure 4.12 shows the Grafana trigger in action. Once triggered, the trigger sends a notification through the appropriate method.

Figure 4.13 illustrates an example flow of attack and respective mitigation using switch latency as an identifying metric. Initially, the system is functioning normally. However, after the attack begins, the switch's latency rapidly increases

```
def bulk_write(pipeline, entry_list, p4info):
    for entry in entry_list:
        writeTunnelRules(p4info, pipeline, entry)

def writeTunnelRules(p4info_helper, pipeline, data):
    table_entry = p4info_helper.buildTableEntry(
        table_name=data['table'],
        default_action = data.get('default_action', False),
        match_fields= data.get('match', None),
        action_name= data.get('action_name', None),
        action_params= data.get('action_params', None),
        priority= data.get('priority', None))

    pipeline.WriteTableEntry(table_entry)
    print("Installed ingress tunnel rule on %s" % pipeline.name)
```

Listing 4.3: Function used to write table entry to the switch

```
def main(cfg_file):
    with open(cfg_file, 'r') as f:
        data = json.load(f)

    p4info_helper = p4runtime_lib.helper.P4InfoHelper(data['p4info'
   ])

    try:
        leaf2 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
            name='leaf2',
            address='127.0.0.1:50052',
            device_id=1,
            proto_dump_file='logs/leaf2-p4runtime-
   requests_controller.txt')
        leaf2.MasterArbitrationUpdate()
        leaf2.SetForwardingPipelineConfig(p4info=p4info_helper.
   p4info,
                                          bmv2_json_file_path=data['
   bmv2_json'])
        bulk_write(leaf2, data['table_entries'],p4info_helper)
```

Listing 4.4: Main body of the reset implementation

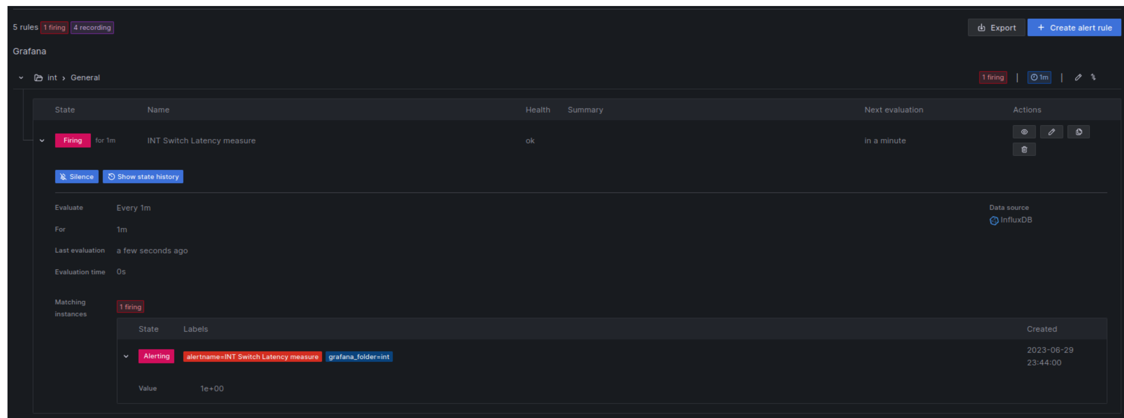

Figure 4.12: Grafana Alert Triggering

Figure 4.13: Timeline of an attack to switch 2

erratically. After a few minutes of this behavior, a control plane reset is issued. Noticeably, during this time, there is no data since the switch is being reinstalled and entries are being re-inserted. Finally, after the measures are applied, the system returns to its normal state. Note that the timeline is not representative of the least amount of time needed to complete a mitigation strategy in a real-world scenario; it should be seen as an upper bound.

This page was intentionally left in blank.

Chapter 5

Conclusion

To conclude this document, a summary of the work is presented below.

Chapter 1 serves as the introductory chapter to the work, providing a formal introduction, motivation, objectives, and contributions.

Chapter 2 begins by providing an allusion to the historical context of networking, detailing the evolution from traditional networks to SDN-based Openflow and P4 networks. Additionally, a brief overview of SDN security is provided.

The remainder of the chapter heavily focuses on the technical and theoretical perspective of P4. This includes a formal overview of the language, using the official P4 specification [Consortium, 2022], a review of INT, and a review of the state of the art. The state of the art section can be divided four topics: P4 bug exploitation, SAST and DAST analysis, and network monitoring.

Chapter 3 marks the beginning of the developmental portion of this thesis. The first section discusses the attacker model, which focuses on staying hidden until the attack is initiated. It is assumed that the attacker follows the same model as described in [Black and Scott-Hayward, 2021], using the "Changing P4 Program - Controller initiated" attack as the entry point. A remote trigger architecture is then formed.

The goal of the attack is to remain hidden for as long as necessary without disrupting regular traffic. Three attack methods are achieved by changing the data-plane code: traffic re-routing, Man-In-the-Middle (MitM), and Denial-of-Service (DoS) for both the entire switch and a single connection.

Finally, SAST tests are run using both gauntlet and BF4, with no successful detection observed based on the current state of the art.

Chapter 4 presents a detection and mitigation framework. The framework makes use of a testbed (using a leafspine architecture) to test the collection of metrics against created attacks, by using an implementation of INT available online [Fan]. Python3 is used for INT processing, InfluxDB for data storage, and Grafana for data display.

The results show that INT can detect some forms of attack (MitM and Switch-

Wide DoS). Finally, a mitigation solution is proposed that involves setting up Grafana alerts to notify the system admin of abnormalities in the network. The admin can then reset the device, reapply all control plane rules, and restore traffic to its original, pre-attack phase.

References

1. Andrei-Alexandru Agape, Madalin Claudiu Danceanu, René Rydhof Hansen, and Stefan Schmid. Charting the security landscape of programmable dataplanes. *arXiv preprint arXiv:1807.00128*, 2018.

2. Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. P4fuzz: Compiler fuzzer fordependable programmable dataplanes. In *International Conference on Distributed Computing and Networking 2021*, pages 16–25, 2021.

3. Izzat Alsmadi and Dianxiang Xu. Security of software defined networks: A survey. *Computers & security*, 53:79–108, 2015.

4. Conor Black and Sandra Scott-Hayward. Adversarial exploitation of p4 data planes. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 508–514. IEEE, 2021.

5. Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

6. Krzysztof Cabaj, Jacek Wytrebowicz, Slawomir Kuklinski, Pawel Radziszewski, and Khoa Truong Dinh. Sdn architecture impact on network security. In *FedCSIS (Position Papers)*, pages 143–148, 2014.

7. Jiamin Cao, Yu Zhou, Chen Sun, Lin He, Zhaowei Xi, and Ying Liu. Firebolt: Finding bugs in programmable data plane generators. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 819–834, 2022.

8. The P4 Consortium. Behavioural model. (accessed: 29.11.2022). URL `https://github.com/p4lang/behavioral-model`.

9. The P4 Language Consortium. In-band network telemetry (int) dataplane specification, version 2.1. 2020.

10. The P4 Language Consortium. $P4_{16}$ language specification, version 1.2.3. 2022.

11. Rogério Leão Santos de Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6, 2014. doi: 10.1109/ColComCon.2014.6860404.

12. Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 683–698, 2019.

13. Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: towards bug-free p4 programs. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 571–585, 2020.

14. Mihai Valentin Dumitru, Dragos Dumitrescu, and Costin Raiciu. Can we exploit buggy p4 programs? In *Proceedings of the Symposium on SDN Research*, pages 62–68, 2020.

15. Lao Fan. (accessed: 20.6.2023). URL `https://www.fool.com/investing/2023/01/29/intel-exits-another-non-core-business/`.

16. Andy Fingerhut. Resubmit examples. (accessed: 6.12.2022). URL `https://github.com/jafingerhut/p4-guide/blob/master/v1model-special-ops/README.md`.

17. The Motley Fool. (accessed: 15.6.2023). URL `https://www.fool.com/investing/2023/01/29/intel-exits-another-non-core-business/`.

18. The Open Networking Foundation. (accessed: 13.12.2022). URL `https://opennetworking.org/`.

19. K Framework. (accessed: 22.12.2022). URL `https://kframework.org/`.

20. Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

21. Shang Gao, Zecheng Li, Bin Xiao, and Guiyi Wei. Security threats in the data plane of software-defined networks. *IEEE network*, 32(4):108–113, 2018.

22. Geantonso. (accessed: 20.6.2023). URL `https://github.com/GEANT-DataPlaneProgramming/int-platforms`.

23. Graphana. (accessed: 15.6.2023). URL `https://grafana.com/`.

24. Mu He, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. Toward consistent state management of adaptive programmable networks based on p4. In *Proceedings of the ACM SIGCOMM 2019 Workshop on Networking for Emerging Applications and Technologies*, pages 29–35, 2019.

25. InfluxDB. (accessed: 15.6.2023). URL `https://www.influxdata.com/`.

26. Mandar Joshi. Implementation and evaluation of in-band network telemetry in p4, 2021.

27. Qiao Kang, Jiarong Xing, and Ang Chen. Automated attack discovery in data plane systems. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019.

28. Ali Kheradmand and Grigore Rosu. P4k: A formal semantics of p4 and applications. *arXiv preprint arXiv:1804.01468*, 2018.

29. Suriya Kodeswaran, Mina Tahmasbi Arashloo, Praveen Tammana, and Jennifer Rexford. Tracking p4 program execution in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 117–122, 2020.

30. Bob Lantz. (accessed: 30.12.2022). URL `https://github.com/mininet/mininet`.

31. Athanasios Liatifis, Panagiotis Sarigiannidis, Vasileios Argyriou, and Thomas Lagkas. Advancing sdn from openflow to p4: A survey. *ACM Comput. Surv.*, 55(9), jan 2023. ISSN 0360-0300. doi: 10.1145/3556973. URL `https://doi.org/10.1145/3556973`.

32. Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, pages 490–503, 2018.

33. Nuno Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. Automatically verifying reachability and well-formedness in p4 networks. *Technical Report, Tech. Rep*, 2016.

34. Tomáš Martínek, Mauro Campanella, Federico Pederzolli FBK, and Joseph Hill. White paper: Timestamping and clock synchronisation in p4-programmable platforms.

35. Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

36. Mininet. (accessed: 11.1.2023), a. URL `http://mininet.org/api/annotated.html`.

37. Mininet. (accessed: 5.1.2023), b. URL `https://github.com/mininet/mininet/wiki/FAQ#assign-macs`.

38. Mininet. (accessed: 11.1.2023), c. URL `https://pypi.org/project/p4runtime/`.

39. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

40. Barefoot Networks. Barefoot tofino. (accessed: 29.11.2022). URL `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html`.

41. Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4pktgen: Automated test case generation for p4 programs. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

42. ONOS. (accessed: 20.6.2023), a. URL `https://github.com/opennetworkinglab/onos/tree/master/pipelines/basic/src/main/resources`.

43. ONOS. (accessed: 21.6.2023), b. URL `https://wiki.onosproject.org/`.

44. P4. (accessed: 1.1.2023). URL `https://github.com/p4lang`.

45. P4-OvS. (accessed: 20.6.2023). URL `https://github.com/osinstom/P4-OvS`.

46. P4Lang. (accessed: 11.1.2023), a. URL `https://github.com/p4lang/behavioral-model/blob/main/targets/README.md`.

47. P4Lang. (accessed: 31.12.2022), b. URL `https://github.com/p4lang/behavioral-model`.

48. P4Team. (accessed: 28.5.2023). URL `https://github.com/p4lang/tutorials`.

49. PacketSender. (accessed: 15.6.2023). URL `https://packetsender.com/`.

50. Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 683–699, 2020.

51. Scapy. (accessed: 21.6.2023). URL `https://scapy.net/`.

52. Arash Shaghaghi, Mohamed Ali Kaafar, Rajkumar Buyya, and Sanjay Jha. *Software-defined network (SDN) data plane security: issues, solutions, and future directions*, pages 341–387. Springer, Springer Nature, United States, 2020. ISBN 9783030222765. doi: 10.1007/978-3-030-22277-2_14.

53. Apoorv Shukla, Kevin Nico Hudemann, Artur Hecker, and Stefan Schmid. Runtime verification of p4 switches with reinforcement learning. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 1–7, 2019.

54. Apoorv Shukla, Seifeddine Fathalli, Thomas Zinner, Artur Hecker, and Stefan Schmid. P4consist: Toward consistent p4 sdns. *IEEE Journal on Selected Areas in Communications*, 38(7):1293–1307, 2020.

55. Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Artur Hecker, Stefan Schmid, and Anja Feldmann. Fix with p6: Verifying programmable switches at runtime. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

56. Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 518–532, 2018.

57. Jakub Svoboda, Ibrahim Ghafir, Vaclav Prenosil, et al. Network monitoring approaches: An overview. *Int J Adv Comput Netw Secur*, 5(2):88–93, 2015.

58. Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 17–32, 2021.

59. Lily Yang, Ram Dantu, Terry Anderson, and Ram Gopal. Forwarding and control element separation (forces) framework. Technical report, 2004.

60. Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5): 32–41, 2014. doi: 10.1109/MM.2014.61.

This page was intentionally left in blank.

Appendices

Appendix A

Understanding Mininet in the context of P4 Tutorials

Mininet is a widely used network simulator. This technology is used in the P4 tutorial repository [P4]. Whilst long tutorials have been created and presented over the years, none focused on explaining how P4 is used in the example repository for the P4 language. This contribution aims to fix just this problem, explaining in detail how the simulation works in the context of P4's tutorial repository.

Section A.1 introduces the technology.

Section A.2 describes the integration between Mininet and Python.

Section A.3 describes the integration between Mininet and P4.

Section A.4 details information about switch architecture.

Section A.5 explains the functionality of the different files required for the simulation.

Section A.6 summarizes all the contents provided in the document.

A.1 Mininet

Originally created by Bob Lantz, Mininet [Lantz] is a very complete network emulation tool. Using Mininet, a user can quickly deploy a full network in minutes. As mentioned in [de Oliveira et al., 2014], "the possibility of sharing results and tools at zero cost are positive factors that help scientists to boost their researches despite the limitations of the tool in relation to the performance fidelity between the simulated and the real environment."

Some drawbacks of Mininet include line-rate fidelity and processing power, which are only of limited relevancy given the purpose of the technology. Mininet, much like P4, is supported by the Open Networking Foundation (ONF), ensuring its long-term support.

A.2 Python Mininet

Mininet supports both python2 and python3 through its API [Mininet, a].Since python2 was deprecated in 2020, it is not considered for the rest of the document.

By using this API, networks can be instantiated from a python script, speeding up the initialization process. Most Mininet abstractions are also usable from within the python API:

- Links, such as OVSLink or the TCLink;
- Switches, such as the OVSSwitch, IVSSwitch, and P4Switch;
- Controllers, such as NOX, OVS, or Ryu;
- NAT and Linux bridges;

A.3 Mininet and P4

To abstract a P4-enabled switch, the P4Runtime library [Mininet, c] is used . Using the *P4RuntimeSwitch* abstraction, the switch can be emulated within Mininet.

A.4 BMv2 Architecture

Several architectures are supported by P4. Since most are closed-source, Mininet only supports a smaller subset. For this reason, Behavioral Model Version 2 (BMv2) [P4Lang, b] was is used. BMv2 is an openly available implementation of the P4 Switch, using C++11. It makes use of JavaScript Object Notation (JSON) format files (which are compiled using the provided P4 compiler (P4c)) to generate switch behavior. The BMv2 behavioral model supports several different versions (also known as targets):

- **simple_switch**. The main target for the software switch. Can execute most $P4_{14}$ and $P4_{16}$ programs. Uses the v1model architecture and can run on most general-purpose CPUs.
- **simple_switch_grpc**. Based on the simple switch, but with support for Transmission Control Protocol (TCP) connections (Remote Procedure Call (gRPC)) from a controller (uses the P4 Runtime Application Programming Interface (API)).
- **psa_switch**. Based on the simple switch, but instead of using the more recent v1model, uses the Portable Switch Architecture (PSA).
- **simple_router and l2_switch**. Implemented as a proof of concept and largely incomplete[P4Lang, a]. Should not be used over the *simple_switch*.

A.5 Simulation Files

Before executing, the following files are required:

- **P4 file(s)**. Defines the behavior of the switch(es).
- **Control plane P4 file(s)**. Provides the entries used to fill the data plane tables (inserted by the control plane).
- **Topology file**. Describes the network topology, connections, switches, and hosts.
- **Makefile**. Automates network generation process.

Note: The files and code used in this document are taken fully or abridged from the official P4 GitHub repository [P4]. Further information on the respective P4 file can be found in `https://github.com/p4lang/tutorials/tree/master/exercises/basic`

A.5.1 P4 File

The P4 file defines the data-plane behavior and it is stored directly in the switch. Its structure is fixed while the switch is running, only allowing modifications to entries of its tables.

Listing A.1 illustrates a simple P4 program, whose goal is to forward an incoming packet to the correct host.

This simple program contains 4 stages (as mentioned in 2.2): Parser (line 2); Ingress (line 32); Egress (line 69); Deparser (line 79). This program constitutes a basic implementation of the forwarding script. Its functionality is: set the *egress.port* (line 40), decrement the Time To Live (TTL) (line 43), and change the source and destination IPs. Note: For the sake o readability, some parts of the code have been removed.

```
1  /(... Omitted ...)/
2  /************* P A R S E R  *************************/
3
4  parser MyParser(packet_in packet,
5                  out headers hdr,
6                  inout metadata meta,
7                  inout standard_metadata_t standard_metadata) {
8
9      state start {
10         transition parse_ethernet;
11     }
12
13     state parse_ethernet {
14         packet.extract(hdr.ethernet);
15         transition select(hdr.ethernet.etherType) {
16             TYPE_IPV4: parse_ipv4;
```

```
17              default: accept;
18          }
19      }
20
21      state parse_ipv4 {
22          packet.extract(hdr.ipv4);
23          transition accept;
24      }
25
26  }
27
28  /(... Omitted ...)/
29
30  /*********  I N G R E S S   P R O C E S S I N G   ***********/
31
32  control MyIngress(inout headers hdr,
33                    inout metadata meta,
34                    inout standard_metadata_t standard_metadata) {
35      action drop() {
36          mark_to_drop(standard_metadata);
37      }
38
39      action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
40          standard_metadata.egress_spec = port;
41          hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
42          hdr.ethernet.dstAddr = dstAddr;
43          hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
44      }
45
46      table ipv4_lpm {
47          key = {
48              hdr.ipv4.dstAddr: lpm;
49          }
50          actions = {
51              ipv4_forward;
52              drop;
53              NoAction;
54          }
55          size = 1024;
56          default_action = drop();
57      }
58
59      apply {
60          if (hdr.ipv4.isValid()) {
61              ipv4_lpm.apply();
62          }
63      }
64  }
65  /(... Omitted ...)/
66
67  /*********  E G R E S S   P R O C E S S I N G   ************/
68
69  control MyEgress(inout headers hdr,
70                   inout metadata meta,
71                   inout standard_metadata_t standard_metadata) {
72      apply {  }
73  }
74
```

```
/(... Omitted ...)/

/*********   D E P A R S E R  *****************/

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}

/(... Omitted ...)/
```

Listing A.1: $P4_{16}$ Simple Forwarding Example

A.5.2 The Control-Plane file

The control-plane file uses JSON format and completes the data-plane file, inserting entries corresponding to the respective data-plane tables. For example, entries for the corresponding ipv4_lpm table are filled in this file. Such tables are filled at startup.

Listing A.2 illustrates a control-plane file example. For each entry for a given table, 4 pieces of information can be modified:

- **table**. Defines to which table the entry corresponds.
- **match**. Defines the matching criteria.
- **action_name**. Defines which action is taken in a successful match.
- **action_params**. Defines the value for the action parameters.

Note: Each parameter should match its respective data-plane counterpart, meaning that for an entry to be inserted, then a table must exist, for an action to be called it must be defined in the data-plane, etc.

Listing A.2, exemplifies a control plane file. Since the example only uses one table in the data-plane counterpart, all entries are defined for said table. Its behavior can be described as: for any given IP (key), the switch performs the *MyIngress.ipv4_forward* action, which sets both the (*dstAddress* and *Port*).

```
{
  "target": "bmv2",
  "p4info": "build/basic.p4.p4info.txt",
  "bmv2_json": "build/basic.json",
  "table_entries": [
    {
      "table": "MyIngress.ipv4_lpm",
      "default_action": true,
      "action_name": "MyIngress.drop",
      "action_params": { }
    },
```

```
12      {
13        "table": "MyIngress.ipv4_lpm",
14        "match": {
15          "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
16        },
17        "action_name": "MyIngress.ipv4_forward",
18        "action_params": {
19          "dstAddr": "08:00:00:00:01:11",
20          "port": 1
21        }
22      },
23      {
24        "table": "MyIngress.ipv4_lpm",
25        "match": {
26          "hdr.ipv4.dstAddr": ["10.0.2.2", 32]
27        },
28        "action_name": "MyIngress.ipv4_forward",
29        "action_params": {
30          "dstAddr": "08:00:00:00:02:22",
31          "port": 2
32        }
33      },
34      {
35        "table": "MyIngress.ipv4_lpm",
36        "match": {
37          "hdr.ipv4.dstAddr": ["10.0.3.3", 32]
38        },
39        "action_name": "MyIngress.ipv4_forward",
40        "action_params": {
41          "dstAddr": "08:00:00:00:03:00",
42          "port": 3
43        }
44      },
45      {
46        "table": "MyIngress.ipv4_lpm",
47        "match": {
48          "hdr.ipv4.dstAddr": ["10.0.4.4", 32]
49        },
50        "action_name": "MyIngress.ipv4_forward",
51        "action_params": {
52          "dstAddr": "08:00:00:00:04:00",
53          "port": 4
54        }
55      }
56    ]
57  }
```

Listing A.2: Control plane file example

A.5.3 Topology File

The topology file uses JSON format and contains information regarding the network and its composition. For a simple triangle topology (as depicted in Figure A.1), the topology file takes the format given in listing A.3.

```
1  {
```

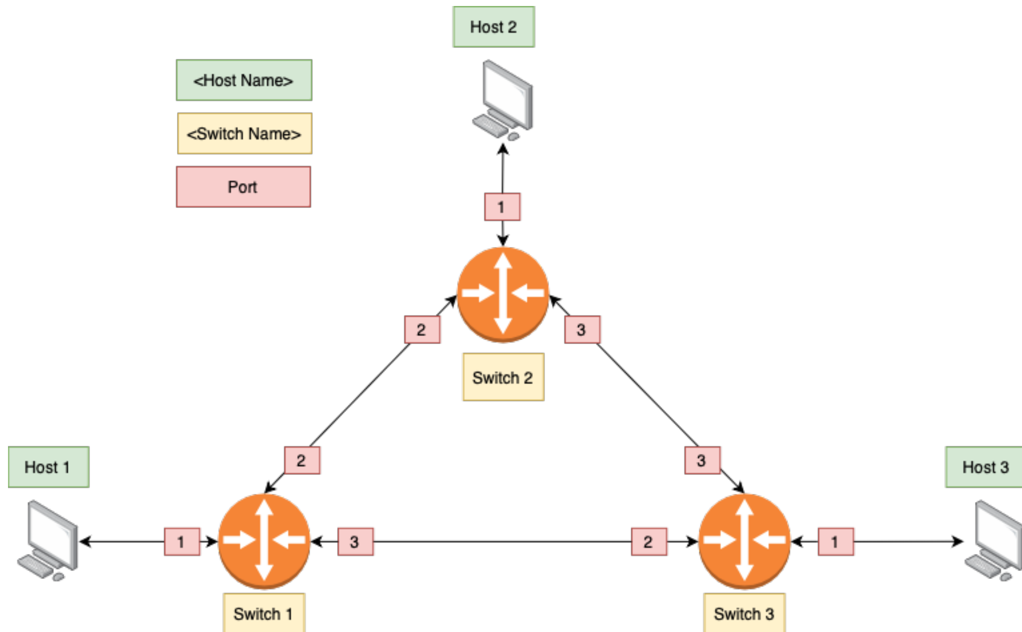

Figure A.1: Triangle topology representation

```
2  "hosts": {
3  "h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
4         "commands":["route add default gw 10.0.1.10 dev eth0",
5                     "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"]},
6  "h2": {"ip": "10.0.2.2/24", "mac": "08:00:00:00:02:22",
7         "commands":["route add default gw 10.0.2.20 dev eth0",
8                     "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00"]},
9  "h3": {"ip": "10.0.3.3/24", "mac": "08:00:00:00:03:33",
10        "commands":["route add default gw 10.0.3.30 dev eth0",
11                    "arp -i eth0 -s 10.0.3.30 08:00:00:00:03:00"]}
12 },
13 "switches": {
14     "s1": { "runtime_json" : "triangle-topo/s1-runtime.json" },
15     "s2": { "runtime_json" : "triangle-topo/s2-runtime.json" },
16     "s3": { "runtime_json" : "triangle-topo/s3-runtime.json" }
17 },
18 "links": [
19     ["h1", "s1-p1"], ["s1-p2", "s2-p2"], ["s1-p3", "s3-p2"],
20     ["s3-p3", "s2-p3"], ["h2", "s2-p1"], ["h3", "s3-p1"]
21 ]
22 }
```

Listing A.3: Topology File

In this file, the following elements can be found:

- **hosts**. Defines the hosts instantiated by Mininet. Parameters such as IP and MAC address and, startup commands are also defined in this section. In the example, the commands used are:
  - *route add*, which adds a static route to the default gateway (the switch).
  - *arp*, which manipulates the system ARP cache (adds an entry for the switch's MAC address).

- **switches**. Defines switch behavior. Three parameters can be inserted here:
  - *program*. Defines the program inserted into the switch (data plane). Note: If this parameter is not set, then the execution assumes the default P4 file (passed on startup)
  - *runtime_json*. Defines the path for the control plane file (similar to the one presented in section A.5.2).
  - *runtime_cli*. Also defines the path for the control plane file. This file is directed at actions that are only supported by the switch_cli interface (such as setting up mirroring).
- **links**. Defines the links between network nodes. The following list format is used: *[Node1, Node2, Latency, Bandwidth]*, where nodes can be defined as *<Hostname>*, for hosts, and *<SwitchName>-<SwitchPort>* for switches. Both *latency* and *bandwidth* is optional, where *latency* is an integer defined in milliseconds(ms) and *bandwidth* is a float defined in megabytes (MB).

Note: IPs need not be attributed to switches, as they make use of the Linux Networking Stack[Mininet, b].

A.5.4 Mininet Python Script File

The most intricate piece of code presented in this document is the Python file. Its goal is to guide the execution, using the files mentioned before, such that a network is established without manually setting it up. The original file can be found in `https://github.com/p4lang/tutorials/blob/master/utils/run_exercise.py`.

Figure A.2, presents an infographic chart of the program's flow of execution. To understand the chart, it should be read in numerical order where **3a** occurs after **3** but before **4**. Instead of a single line of execution, the chart was created so that it alludes to the different methods present in the code, bearing a resemblance to the underlying code structure.

The following subsections analyze the different stages of execution (1 to 8) represented in the infographic.

1. Exercise instantiation;
2. Running the Exercise;
3. Creating the Mininet Network;
4. Starting the Mininet Network;
5. Programming the Hosts;
6. Programming the Switches;
7. Instantiating the Mininet Command Line Interface (CLI);
8. Stopping the Mininet Network.

Figure A.2: Flow execution of the python script

Exercise Instantiation

The initializer intakes the arguments passed by execution and performs an initial formal parsing. Most notably it converts the links from the given <*Node*>-<*Node*> format to a Python dictionary. This dictionary (shown in listing A.4) contains the 4 elements mentioned in topology file A.5.3.

```
#(... Omitted ...)#
link_dict = {'node1':s,
             'node2':t,
             'latency':'0ms',
             'bandwidth':None
            }
#(... Omitted ...)#
```

Listing A.4: Link dictionary format

Running the Exercise

The class *Exercise*, is created to help manage all data and manage the execution flow. Listing A.5 describes the *run_exercise()* method. Lines 4 and 5 create the network. Lines 9 and 10 program the network elements. Line 15 launches the user interface.

```
#(... Omitted ...)#
def run_exercise(self):
# Initialize mininet with the topology specified by the config
    self.create_network()
    self.net.start()
    sleep(1)

    # some programming that must happen after the net has started
    self.program_hosts()
    self.program_switches()

    # wait for that to finish. Not sure how to do this better
    sleep(1)

    self.do_net_cli()
    # stop right after the CLI is exited
    self.net.stop()
#(... Omitted ...)#
```

Listing A.5: Exercise control flow

Creating the Mininet Network

Creating the network is the most complex stage. It is in this stage that switches, links, and hosts are added to the network. To instantiate the network object, a class, *ExerciseTopo*, is used. This class inherits from the *Topo* Class native to Mininet. Listing A.6 shows the initialization process of the *ExerciseTopo* class.

```
#(... Omitted ...)#
```

```
2 self.topo = ExerciseTopo(self.hosts, self.switches, self.links,
    self.log_dir, self.bmv2_exe, self.pcap_dir)
3
4 class ExerciseTopo(Topo):
5     """ The mininet topology class for the P4 tutorial exercises.
6     """
7     def __init__(self, hosts, switches, links, log_dir, bmv2_exe,
    pcap_dir, **opts):
8         Topo.__init__(self, **opts)
9         host_links = []
10        switch_links = []
11  #(... Omitted ...)#
```

Listing A.6: Generating the network topology object

Next, listing A.7 illustrates the configuration of the switches. Using the method *configureP4Switch* ensures the switch is created using the correct architecture, (*simple_switch* or *simple_switch_grpc*).

If no program is specified, the switch follows the default implementation (as noted in A.14). As mentioned, since the switches use the Linux Network Stack, no IPs need to be provided.

```
1 #(... Omitted ...)#
2 for sw, params in switches.items():
3     if "program" in params:
4         switchClass = configureP4Switch(
5                 sw_path=bmv2_exe,
6                 json_path=params["program"],
7                 log_console=True,
8                 pcap_dump=pcap_dir)
9     else:
10        # add default switch
11        switchClass = None
12    self.addSwitch(sw, log_file="%s/%s.log" %(log_dir, sw), cls=
    switchClass)
13 #(... Omitted ...)
```

Listing A.7: Configuring the switches

As seen in A.8, the penultimate step is to generate the hosts and the host-to-switch links. Using the information provided in the topology file (recall section A.5.3) and the methods *addHost* and *addLink*, configurations are directly translated to the Mininet Network.

```
1 #(... Omitted ...)
2 for link in host_links:
3     host_name = link['node1']
4     sw_name, sw_port = self.parse_switch_node(link['node2'])
5     host_ip = hosts[host_name]['ip']
6     host_mac = hosts[host_name]['mac']
7     self.addHost(host_name, ip=host_ip, mac=host_mac)
8     self.addLink(host_name, sw_name,
9                  delay=link['latency'], bw=link['bandwidth'],
10                 port2=sw_port
11 #(... Omitted ...)
```

Listing A.8: Configuring topology host and host links

Finally, listing A.9 details the creation of links between switches. Using the data structure presented in A.4, a parsing method splits the switch name and port, creating links based on such properties.

```
#(... Omitted ...)
for link in switch_links:
    sw1_name, sw1_port = self.parse_switch_node(link['node1'])
    sw2_name, sw2_port = self.parse_switch_node(link['node2'])
    self.addLink(sw1_name, sw2_name,
                 port1=sw1_port, port2=sw2_port,
                 delay=link['latency'], bw=link['bandwidth'])
#(... Omitted ...)
```

Listing A.9: Configuring topology switch links

Starting the Mininet Network

Starting the Mininet network can be achieved by using the *start* method of the *network* object. In the example, the line of code is *self.net.start()*

Programming the Hosts

After starting the network, runtime commands are executed. Listing A.10, demonstrates how console commands are applied to the hosts created in the network. First, the host is retrieved from the network (using its name as key) and then, using the method *<host>.cmd(<command>)*, commands are executed.

```
#(... Omitted ...)#
def program_hosts(self):
    for host_name, host_info in list(self.hosts.items()):
        h = self.net.get(host_name)
        if "commands" in host_info:
            for cmd in host_info["commands"]:
                h.cmd(cmd)
#(... Omitted ...)#
```

Listing A.10: Programming the Host

Programming the Switches

Much like the hosts, the switches also have a runtime counterpart. The method *program_switches* divides execution according to the switch type (*simple_switch* or *simple_switch_grpc*) and runs the respective architecture-specific commands (Note: since the *simple_switch_grpc* extends the (*simple_switch* the configuration may make use of both CLI and runtime methods). Listing A.11 denotes the sub-branch for the configuration of the *simple_switch_grpc*. Most notably, this method makes use of the control plane file (referred to in section A.5.2).

```
#(... Omitted ...)#
def program_switch_p4runtime(self, sw_name, sw_dict):
    sw_obj = self.net.get(sw_name)
```

```
    grpc_port = sw_obj.grpc_port
    device_id = sw_obj.device_id
    runtime_json = sw_dict['runtime_json']
    self.logger('Configuring switch %s using P4Runtime with file %s
    ' % (sw_name, runtime_json))
    with open(runtime_json, 'r') as sw_conf_file:
        outfile = '%s/%s-p4runtime-requests.txt' %(self.log_dir,
    sw_name)
        p4runtime_lib.simple_controller.program_switch(
            addr='127.0.0.1:%d' % grpc_port,
            device_id=device_id,
            sw_conf_file=sw_conf_file,
            workdir=os.getcwd(),
            proto_dump_fpath=outfile,
            runtime_json=runtime_json
        )
#(... Omitted ...)#
```

Listing A.11: Programming the Switch

Instantiating the Mininet CLI

The final step of execution is presenting the user with a usable interface. This tool is called the "Mininet CLI" and lets the user perform several operations in the network (such as ping hosts, instantiating command lines inside the switches, etc). In the example, listing A.12 presents *do_net_cli* method, which first prints information about the network and then calls the *Mininet CLI*.

```
def do_net_cli(self):
    #(... Omitted ...)#
    print('=================================================')
    print('Welcome to the BMV2 Mininet CLI!')
    print('=================================================')
    print('Your P4 program is installed into the BMV2 software
    switch')
    print('and your initial runtime configuration is loaded. You
    can interact')
    print('with the network using the mininet CLI below.')
    print('')
    #(... Omitted ...)#
    CLI(self.net)
```

Listing A.12: Instantiating the Mininet CLI

Stopping the Mininet Network

Stopping the Mininet network is similar to starting, it can be achieved by calling the "stop" method. In the example, the line of code is *self.net.stop()*

Other Notes

Switch Abstraction

There might be some confusion regarding the origin of the *P4RuntimeSwitch* class. This class materializes via library import. Listing A.13 details this process.

```
from p4_mininet import P4Host, P4Switch
from p4runtime_switch import P4RuntimeSwitch
```

Listing A.13: $P4_{16}$ Switch Abstraction Library

Default Switch

As mentioned previously, if no program is provided, the default switch is used instead. This switch is defined in A.14 and is largely similar to the program-enabled one. The difference in the program running inside the switch is defined at startup.

```
defaultSwitchClass = configureP4Switch(
                                sw_path=self.bmv2_exe,
                                json_path=self.switch_json,
                                log_console=True,
                                pcap_dump=self.pcap_dir)
```

Listing A.14: Default Switch

A.5.5 Makefile

Make is a very useful tool when it comes to automation. A Makefile is used by the Make utility for configuration. Listing A.15 shows the Makefile used in the tutorials.

```
BUILD_DIR = build
PCAP_DIR = pcaps
LOG_DIR = logs

P4C = p4c-bm2-ss
P4C_ARGS += --p4runtime-files $(BUILD_DIR)/$(basename $@).p4.p4info
    .txt

RUN_SCRIPT = ../../utils/run_exercise.py

ifndef TOPO
TOPO = topology.json
endif

source = $(wildcard *.p4)
compiled_json := $(source:.p4=.json)

ifndef DEFAULT_PROG
DEFAULT_PROG = $(wildcard *.p4)
endif
DEFAULT_JSON = $(BUILD_DIR)/$(DEFAULT_PROG:.p4=.json)

# Define NO_P4 to start BMv2 without a program
ifndef NO_P4
run_args += -j $(DEFAULT_JSON)
endif

```

```
27 # Set BMV2_SWITCH_EXE to override the BMv2 target
28 ifdef BMV2_SWITCH_EXE
29 run_args += -b $(BMV2_SWITCH_EXE)
30 endif
31
32 all: run
33
34 run: build
35 	sudo python3 $(RUN_SCRIPT) -t $(TOPO) $(run_args)
36
37 stop:
38 	sudo mn -c
39
40 build: dirs $(compiled_json)
41
42 %.json: %.p4
43 	$(P4C) --p4v 16 $(P4C_ARGS) -o $(BUILD_DIR)/$@ $<
44
45 dirs:
46 	mkdir -p $(BUILD_DIR) $(PCAP_DIR) $(LOG_DIR)
47
48 clean: stop
49 	rm -f *.pcap
50 	rm -rf $(BUILD_DIR) $(PCAP_DIR) $(LOG_DIR)
```

Listing A.15: Makefile

This Makefile has 3 main modes of execution):

- **run**. Build and runs the main program.
- **stop**. Stops the execution of the current Mininet network.
- **clean**. First calls stop, then cleans all files generated by execution.

Both **stop** and **clean** are straightforward. **Stop** calls the Mininet command to stop the network *sudo mn -c*, while clean additionally removes all execution files using "*rm -f *.pcap*" and "*rm -rf $(BUILD_DIR) $(PCAP_DIR) $(LOG_DIR)*".

As for **run**, follows the steps mentioned below:

1. Creates all the directories needed for execution: build, pcap, and log directories.
2. Convert all .p4 files into JSON files using the P4 Compiler (p4c) utility. These files are placed in the build directory.
3. Runs the python script. The following arguments are used: the topology file (topology.json by default); the default JSON file; the switch architecture type.

The main make file, A.15, is meant to be a part of other Makefiles (working as a library). In this way, the user just needs to set the required variables for execution and call the "main" Makefile. An example of a Makefile that uses the "library Makefile" can be found in listing A.16.

```
BMV2_SWITCH_EXE = simple_switch_grpc
TOPO = pod-topo/topology.json

include ../../utils/Makefile
```

Listing A.16: Short makefile which makes use of the library makefile

A.6 Wrap-up

This document's goal was to educate the reader about the inner workings of the infrastructure behind the P4 tutorial repository [P4]. This document covers all the files required to run a simulation, how to alter in order to them to create a customized simulation, and a thorough explanation of the python script used to run the simulation. To conclude, a summary chart (Figure A.3 is presented, detailing all necessary steps and files to run a simulation of a P4 virtual network in the context of the P4 tutorial repository.

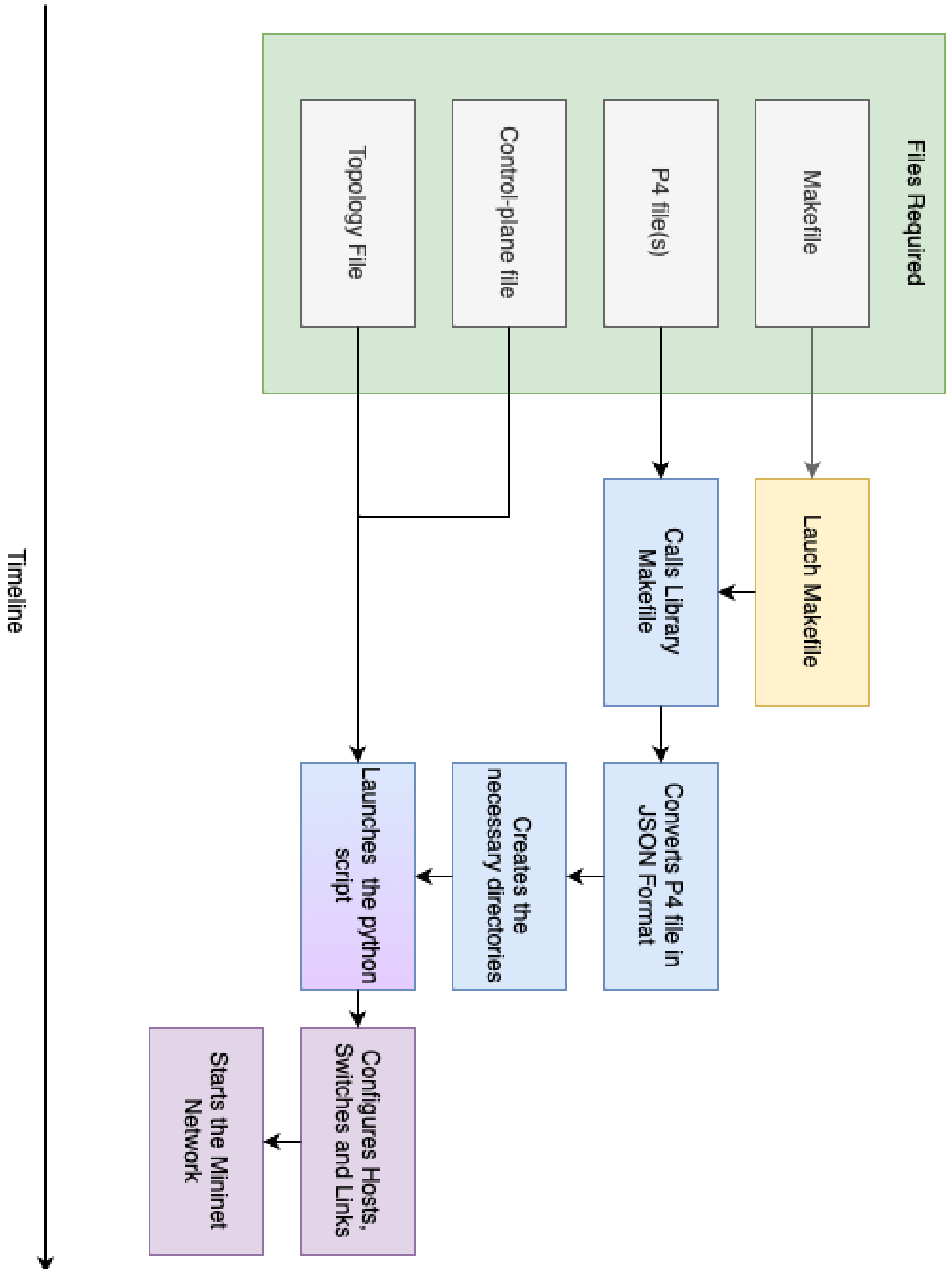

Figure A.3: Execution summary and needed files

This page was intentionally left in blank.