Protocol Abuse Mitigation In SDN Programmable Data Planes

Abir Laraba

▶ To cite this version:

Abir Laraba. Protocol Abuse Mitigation In SDN Programmable Data Planes. Cryptography and Security [cs.CR]. Université de Lorraine, 2022. English. NNT : 2022LORR0168 . tel-03960197

HAL Id: tel-03960197
https://hal.univ-lorraine.fr/tel-03960197

Submitted on 27 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LORRAINE | BIBLIOTHÈQUES UNIVERSITAIRES

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact bibliothèque : ddoc-theses-contact@univ-lorraine.fr

(Cette adresse ne permet pas de contacter les auteurs)

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4

Code de la Propriété Intellectuelle. articles L 335.2- L 335.10

http://www.cfcopies.com/V2/leg/leg_droi.php

http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Protocol Abuse Mitigation In SDN Programmable Data Planes

THÈSE

présentée et soutenue publiquement le 11 octobre 2022

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Abir Laraba

Composition du jury

Président :	Stephan Merz	Directeur de recherche, Inria Nancy - Grand Est
Rapporteurs :	Maryline Laurent	Professeur, Télécom SudParis
	Guillaume Doyen	Professeur, IMT Atlantique
Examinateurs :	Daphné Tuncer	Chercheuse Senior, École des Ponts ParisTech
	Mohamed Faten Zhani	Professeur, ÉTS Montréal, Université du Québec
Encadrants :	Isabelle Chrisment (Directrice de thése)	Professeur, Université de Lorraine
	Jérôme François (Co−directeur de thése)	Chargé de recherche, Inria Nancy - Grand Est
	Raouf Boutaba (Co−directeur de thése)	Professeur, University of Waterloo

---

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

Je tiens à remercier en tout premier lieu mes trois encadrants pour leurs bienveillances. Ma directrice de thèse, Isabelle Chrisment, merci d'avoir été à l'écoute, de m'avoir reboosté quand j'en avais besoin et de m'avoir fait confiance. Mon co-directeur de thèse, Jérôme François, j'aimerais te remercier pour ta gentillesse, pour la qualité de tes conseils et pour m'avoir également accompagné tout au long de la thèse. Je tiens à remercier mon co-directeur de thèse, Raouf Boutaba pour sa disponibilité et son implication malgré la distance et pour son accueil au sein de son équipe à Waterloo.

Je tiens également à remercier Guillaume Doyen et Maryline Laurent d'avoir accepté d'être raporteur.ice de ma thèse, ainsi que Stephan Merz, Daphné Tuner et Mohamed Faten Zhani de faire partie de mon jury de thèse.

Une pensée pour tous les membres de l'équipe RESIST, Pierre-Marie, Mehdi, Philippe [1], Mohamed, Adrien, Karim, Nicolas, Omar, Jean philippe et bien d'autres. Une pensée particulière à mes collègues de bureau avec qui j'ai partagé mon quotidien, Lama [2], Matthews [3] et Ahmad. Merci aux permanents pour leurs convivialités. Merci à tous les collègues et amis de LORIA, Imene, Athénais, Laurine, Geuilherme, Jean-baptiste, Charifa et Karima [4]. Mes remerciements s'adressent également à toutes les personnes de LORIA/Inria qui m'ont témoigné leur sympathie et leur gentillesse, Isabelle (merci pour ton sourire au quotidien et tes délicieux desserts), Agnès, Caroline.

Mes plus profonds remerciements vont à ma famille et mes proches, mes mots ne seraient jamais à la hauteur de l'amour et l'affection que vous m'avez témoignée. Je remercie mon père 'Abi' pour m'avoir toujours soutenu et encouragé. Merci à mes deux frères, Khlaled et Amine, merci à mes deux chères sœurs, Ibtissem et Asma d'avoir toujours été là et pour m'avoir tant de fois écouté. Merci à ma chère nièce et ma petite sœur Yasmine 'Samouna' et à mes chers neveux 'Lokman', 'Ismail' et 'Yacine'. Je remercie particulièrement 'tata Djoumala' pour ses encouragements. Un grand merci à toutes mes amies d'avoir toujours été là au fil des années, 'les plus belles' Imenou, Ibti, Nouna, Safo, Imenouch, Tina, Lily. Je remercie mon chéri, celui qui m'a montré qu'aucune difficulté n'est insurmontable, merci d'avoir su me supporter tout au long de ces années. Ton soutien et encouragement constants m'ont été d'un grand réconfort et ont contribué à l'aboutissement de ce travail, merci pour tout [5].

À toutes et à tous, merci.

[1] je dois revenir à Nancy seulement pour la fameuse randonnée dans les Vosges
[2] nos rigolades féminines vont me manquer
[3] it was fun sharing the experience of writing the manuscript with you (P4, differents architectures :/ you know)
[4] merci pour tes nombreux conseils, nos discussions dans le couloir du bâtiment B vont me manquer
[5] c'est le mot 'chwetra' qui a fait toute la différence ;)

À ma mère 'Oumi' ...

"Persévérer, secret de tous les triomphes."
Victor Hugo

Résumé

L'émergence du paradigme des réseaux définis par logiciel (SDN) a favorisé le développement de nouveaux mécanismes de surveillance des réseaux grâce à leur programmabilité. Le premier objectif du SDN est de centraliser l'intelligence du réseau au niveau d'un plan de contrôle avec un plan de données sans état (*i.e.,* des éléments de réseau, des commutateurs). Par conséquent, les fonctions de surveillance du réseau nécessitent l'aide du contrôleur distant ou l'extension des protocoles de plan de données existants. Ces dernières années, des efforts ont été faits pour rendre le plan de données plus programmable et plus apte à supporter un traitement à état, ce qui permet de déployer des fonctions personnalisées et de décharger de nombreuses applications sur les éléments du réseau. Par exemple, la programmabilité du plan de données peut être assurée par le langage P4 qui définit la manière dont les paquets sont traités dans un pipeline d'un commutateur et qui prend en charge le traitement à état des paquets. Cependant, le langage P4 ne fournit pas d'abstractions à états intuitives pour modéliser les attaques comportementales. Il est donc nécessaire de reconsidérer des abstractions générales pour modéliser et surveiller un comportement complexe et à état dans le plan de données. Pour exploiter les opportunités offertes par un plan de données programmable, notamment le traitement des paquets à état et en temps réel, nous avons besoin de modèles suffisamment simples pour être déployés sur un commutateur programmable, tout en respectant les primitives existantes qui restent limitées, et en même temps, capables de capturer un comportement complexe.

Parallèlement, les attaquants exploitent les vulnérabilités présentes dans les protocoles utilisés au cœur de l'Internet, tels que TCP et DNS. Cependant, les solutions proposées pour détecter ces attaques nécessitent de modifier l'implémentation des protocoles au niveau des hôtes finaux ou ont un impact négatif sur les flux bénins. Par conséquent, corriger les vulnérabilités à l'échelle de l'Internet nécessiterait beaucoup de temps pour le déploiement, comme dans le cas de DNSSEC. Dans cette thèse, nous abordons ces lacunes en définissant une fonction de sécurité (c'est-à-dire une approche de détection et d'atténuation des attaques) qui peut être déployée dans un plan de données programmable de SDN. Nous proposons une abstraction reposant sur une machine à états finis étendue (EFSM) pour modéliser le comportement d'une attaque. Pour détecter des attaques sophistiquées telles que les attaques à étapes multiples, nous l'étendons avec un réseau de Petri pour synchroniser la détection d'un ensemble d'étapes d'attaque. Nous présentons comment ces modèles peuvent être mappés aux primitives P4 afin de pouvoir détecter et agir contre une attaque au sein du réseau. Nous présentons trois attaques de la couche 3, la couche 4 et la couche 7 du modèle OSI (Open System Interconnection), à savoir l'abus du protocole ECN (Explicit Congestion Notification), l'attaque Optimisitic ACK et la récente attaque DNS par empoisonnement du cache. Notre approche ne nécessite pas de modifier l'implémentation du protocole au niveau des hôtes finaux. En outre, notre solution s'appuie sur des plans de données programmables, permettant la surveillance des flux et la réaction contre les attaques en temps réel au sein du réseau.

Mots-clés: SDN, Plan de données programmable, Détection des attaques, Traitement avec état, P4, Abstractions, EFSM, Réseaux de Petri, TCP, ECN , Attaque Optimistic ACK , DNS, Attaque par empoisonnement du cache DNS

Abstract

The emergence of the Software-Defined Networking paradigm has supported the development of new network monitoring scheme thanks to network programmability. The first purpose of SDN is to centralize the network intelligence at the control plane with a stateless data plane (i.e., network elements, switches). As a result, the network monitoring functions require the help of the remote controller or the extension of existing data plane protocols. In recent years, efforts have been made to make the data plane more programmable and stateful, permitting customized functions deployment and offloading many applications to network elements (e.g., forwarding devices). For example, the data plane programmability can be enabled by the P4 language that defines how packets are processed in a switch pipeline and supports stateful packet processing. However, P4 does not provide intuitive stateful abstractions to model behavioral attacks. Therefore, rethinking on the general abstractions to model and track a complex and stateful behavior in the data plane is necessary. To exploit the opportunities offered by a programmable data plane, including stateful and real-time packet processing, we need models which are simple enough to be deployed on a programmable switch with respect to the existing primitives that remain limited and, at the same time, capable of capturing a complex behavior.

Meantime, attackers exploit vulnerabilities present in protocols used in the core of the Internet, such as TCP and DNS. However, the proposed solutions to detect these attacks require modifying the protocol implementation at the end-hosts or having a negative impact on benign flows. Therefore, patching at the scale of the Internet would require much time for deployment, such as in the case of DNSSEC. In this thesis, we address these shortcomings by designing a security function (i.e., an attack mitigation approach) that can be deployed in an SDN programmable data plane. We propose an abstraction based on an Extended Finite State Machine (EFSM) to model an attack behavior. To detect sophisticated attacks such as multi-step attacks, we extend it with Petri Net to synchronize the detection of a set of attack steps. We present how these models can be mapped to P4 primitives so that we can detect and react against an attack within the network. We present three attacks from Layer-3, Layer-4, and Layer-7 of the OSI model, namely the ECN protocol abuse, the Optimistic ACK attack, and the recent DNS multi-step cache poisoning attack. Our approach does not require modifying protocol implementation at the end hosts. Besides, our solution leverages programmable data planes, enabling flow tracking and reaction against attacks in real-time within the network.

Keywords: SDN, Programmable data plane, Attack detection, Stateful processing, P4, Abstractions, EFSM, Petri Nets, TCP, ECN , Optimistic ACK attack, DNS, DNS cache poisoning attack

Contents

Chapter 1
Introduction

1.1 Context . . . 2
1.2 Problem definition . . . 3
1.3 Contributions . . . 5
1.3.1 EFSM-based approach for attack detection in the programmable data plane and its applications . . . 5
1.3.2 Modular approach for multi-step attack detection in the programmable data plane and its application . . . 5
1.4 Thesis outline . . . 6

Chapter 2
State of the art

2.1 Introduction . . . 10
2.2 Programmable network history . . . 10
2.3 Programmable data plane architecture, language, and targets . . . 12
2.3.1 Programmable data plane architectures . . . 12
2.3.1.1 RMT architecture . . . 13
2.3.1.2 PISA architecture . . . 13
2.3.2 P4 language . . . 14
2.4 Stateful data plane . . . 19
2.4.1 Stateful data plane abstractions . . . 19
2.4.2 Stateful data plane processing . . . 25
2.4.3 State management . . . 26
2.5 Applications offloaded to the data plane . . . 27
2.5.1 Network monitoring . . . 27
2.5.2 Network security . . . 29
2.6 Summary . . . 32

Chapter 3
Stateful approach for detecting and mitigating attacks in programmable data planes

3.1 Introduction . . . 34
3.2 Protocol behavior monitoring in the data plane . . . 35
3.2.1 Overview . . . 35
3.2.2 Extended Finite State Machine abstraction . . . 37
3.2.3 Stateful behavior example : a DDoS detection EFSM . . . 38
3.2.4 From an EFSM to P4 programs . . . 39
3.3 Application to ECN . . . 44
3.3.1 Background and attack description . . . 44
3.3.2 EFSM model . . . 45
3.3.3 P4 implementation details . . . 47
3.3.4 Evaluation . . . 50
3.3.4.1 Bandwidth share and throughput evaluation . . . 50
3.3.4.2 Switch processing time evaluation . . . 53
3.4 Application to Optimistic ACK attack . . . 55
3.4.1 Attack description . . . 55
3.4.2 Attack EFSM model . . . 57
3.4.3 Evaluation . . . 58
3.4.3.1 Attack mitigation . . . 58
3.4.3.2 Throughput evaluation . . . 58
3.4.3.3 Switch processing time evaluation . . . 60
3.5 Design challenges . . . 61
3.5.1 TCP connection tracking . . . 61
3.5.2 Scalability and memory overhead . . . 62
3.5.3 Hash collisions . . . 63
3.6 Summary . . . 63

Chapter 4
Modular approach for detecting multi-step attacks in programmable data planes

4.1 Introduction . . . 66
4.2 Proposed approach . . . 67
4.2.1 Petri Net background . . . 67
4.2.2 Approach overview . . . 69

4.2.3 Models to compose attacks 70
4.2.4 From Petri Net model to P4 programs 72
4.3 DNS protocol 74
4.3.1 DNS architecture and component 74
4.3.2 DNS security weakness 75
4.3.3 History of DNS security and some proposed solutions 77
4.4 Application to the new multi-step DNS cache poisoning attack 79
4.4.1 Attack Petri Net 79
4.4.2 Port scan detection 81
4.4.3 DNS brute force detection 82
4.4.4 Evaluation 83
4.4.4.1 DNS cache poisoning attack mitigation 83
4.4.4.2 Switch processing time 85
4.5 Summary 86

Chapter 5
Conclusion

5.1 Achievements 90
5.2 Limitations, perspectives, and future work 91

Publications **93**

Glossary **95**

Bibliography **97**

Appendix **105**

Appendix A
Résumé en français

A.1 Introduction 107
A.2 Problématique 108
A.3 Abstraction pour détecter les attaques dans un plan de données programmable 109
A.4 Méthode pour détecter les attaques composées dans un plan de données programmable 110
A.5 Conclusion 111

Appendix B
Optimistic ACK attack detection P4 implementation details

Appendix C
DNS cache poisoning attack detection P4 implementation details

List of Figures **125**

List of Tables **129**

Chapter 1

Introduction

Sommaire

1.1 Context **2**
1.2 Problem definition **3**
1.3 Contributions **5**
1.3.1 EFSM-based approach for attack detection in the programmable data plane and its applications 5
1.3.2 Modular approach for multi-step attack detection in the programmable data plane and its application 5
1.4 Thesis outline **6**

1.1 Context

In recent years, networks have become complex and difficult to manage. Therefore, solutions have been proposed to make networks more programmable and simple to manage. Software Defined Network (SDN) technology was proposed; SDN is a network technology that separates the control plane, which decides how to handle network traffic from the data plane (*i.e.,* switches), which forwards traffic (Figure 1.1). In the first place, SDN was introduced to enable control plane programmability, where network control logic is centralized. Whereas the data plane is designed with limited capabilities (*i.e.,* possible actions) in order to process packets at line rate, in return, the control plane performs complex operations at a lower speed. The controller communicates with data plane devices using an API (Application Programming Interface). The SDN first proposal (*i.e.,* OpenFlow) lacks a persistent state in data plane elements. As a result, controllers send forwarding rules to data plane devices based on network behavior, which induces latency and network load. Although SDN has simplified network control, the central control and the stateless data plane introduce overhead and scalability issues due to the explicit involvement and frequent control of network elements and traffic (*i.e.,* as presented in Figure 1.1 some packets are sent to the control plane to configure the forwarding rules.)

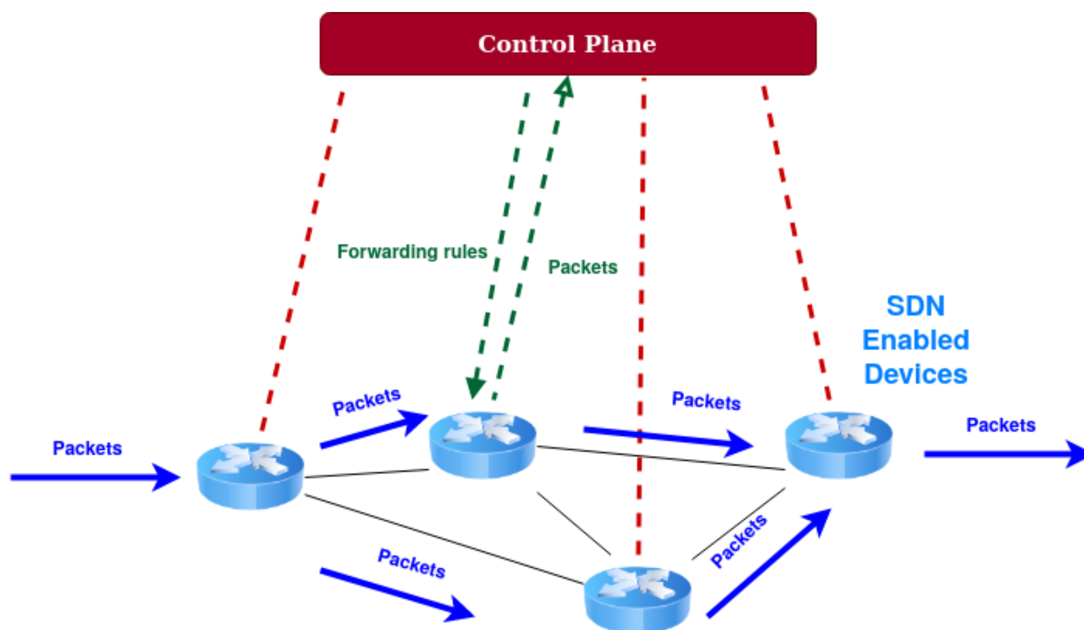


Figure 1.1: Software Defined Network architecture

Another technology, Network Function Virtualization (NFV) has been proposed to improve network agility. NFV virtualizes network services (routers, firewalls, load balancers, etc.) that traditionally run on proprietary hardware. These services are deployed in virtual machines on commodity hardware. As dedicated hardware is not needed for each network function, NFV improves agility by allowing operators to offer new network services and applications on-demand without additional hardware resources. However, the deployment of hardware with specific network service software introduces a high cost (i.e., adding and updating middleboxes). Network traffic must be routed to middleboxes, usually placed at the edge of the network, and

thus routing policies must be configured manually. Indeed, NFV-based software services are deployed on CPUs with limited speed compared to network interface speed.

SDN introduces the possibility of controlling the behaviors of network forwarding elements, although it introduces scalability and latency issues. At the same time, NFV proposes to use specific software functions instead of specific hardware, but with an extra cost. To address these drawbacks, significant efforts have been made to make the network data plane more programmable and stateful, allowing the offloading of complex and stateful operations to network elements (*i.e.,* a programmable switch would support in-network packet processing, allowing packets to be processed on the fly as they traverse the network).

A language called P4 has been proposed to program network elements. P4 is considered as an evolution of SDN. It allows programming the processing of individual packet by network elements (*i.e.,* switches), whether hardware or software. P4 supports not only the well-known standard protocol headers and actions, as traditional switches usually do, but also fully customized protocol headers and actions in contrast to OpenFlow. In addition, it proposes stateful structures (*i.e.,* to maintain the state inside the switches). It therefore allows customized, real-time and stateful packet processing within the network.

Meanwhile, our dependence on the services offered by the Internet is growing, and the massive use of the Internet has created a lot of traffic. This significant increase in Internet traffic and emerging technologies that use different network protocols is associated with an increased risk of network attacks. As a result, new attacks with various motivations have been developed and have become more sophisticated and costly. They can target the protocols used at the core of the Internet (e.g., TCP) and can cause significant damage to the global functioning of systems and services. Thus, identifying attacks at an early stage and reacting in real-time is important to eliminate attacks or reduce their impact.

1.2 Problem definition

Recent advances in programmable data planes have allowed offloading some functions to network elements (e.g., network monitoring and security functions). However, offloading stateful applications to the data plane is still an open question. Despite the powerful programmability and new primitives and functionalities proposed by the programmable data plane to support stateful processing, it does not have intuitive (*i.e.,* automatic) support for describing stateful applications to monitor flow traffic and capture particular behavior in the data plane. The proposed data plane language is limited in describing a stateful function that monitors the flow state (*i.e.,* track per-flow information). A common abstract model for stateful processing in the data plane would be necessary to support different applications for flow tracking over different data plane targets.

To address this challenge, efforts have been dedicated for realizing an OpenFlow-based stateful data plane abstraction. These solutions extend OpenFlow but are still limited to fixed header fields and actions (*i.e.,* switches are limited only to stateless processing), which requires a partial controller involvement. Other solutions proposed stateful abstraction in the fully programmable data plane, but are still hardware-specific. Thus, existing stateful abstractions are either limited by the stateless nature of OpenFlow or remain hardware-specific (*i.e.,* specific to one hardware type). A programmable data plane would allow users to deploy new solutions easily. Therefore,

a new stateful abstraction should rely on a common language that could be compiled to different targets instead of relying on specific hardware implementation details.

The flexibility and programmability of SDN have allowed supporting a wide range of applications, including security applications. They have also helped to replace middleboxes and their high cost. While programmable networks enable effective attack detection, an in-network solution would provide line-rate packet processing and allow real-time attack mitigation. However, OpenFlow-based solutions require the explicit involvement of a distant controller, which introduces additional delay and load. P4-based solutions are designed for specific attack detection without considering a common approach for different applications. At the same time, attacks target known and end-host protocols such as TCP and DNS (*i.e.,* implemented on end-hosts). As a result, the proposed solutions are designed on end-hosts (*i.e.,* end-host based approaches) and require the modification of protocol implementations on the end-host. The limitation of these solutions emerges when it comes to their deploying on an Internet-wide scale, this requires implementation changes at the Internet scale, which is a complex and time-consuming operation. A more efficient approach would be to deploy an attack mitigation solution within the network so that attacks are detected and mitigated in real-time in the traffic path before reaching the end-hosts (Figure 1.2).

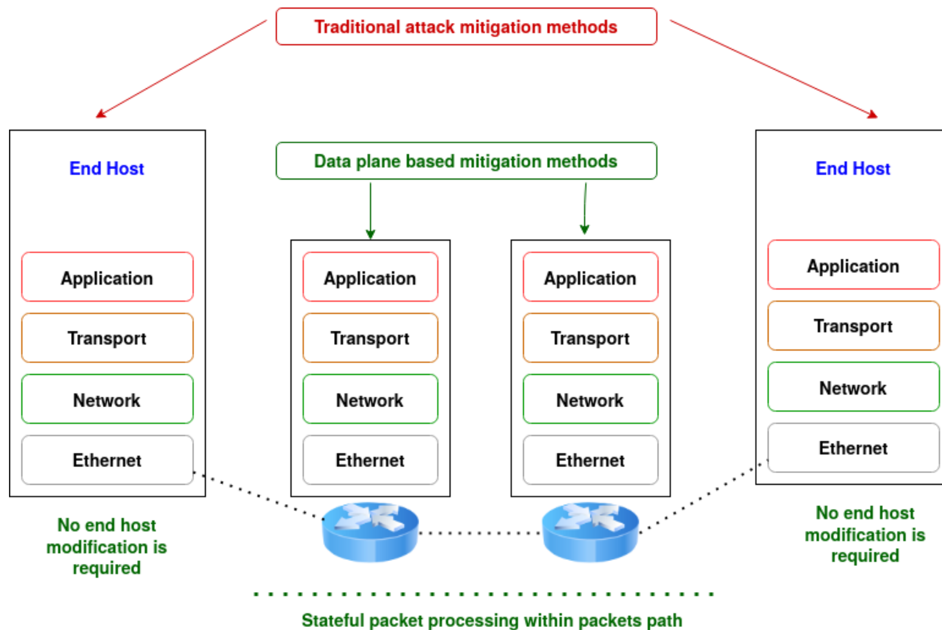


Figure 1.2: Attack mitigation solutions (end-host vs. data plane-based)

Research questions

In this thesis, we address the following research questions:

1. Can we design a stateful abstraction for a programmable data plane intended for stateful attack behavior tracking? The abstraction should be simple enough to be deployed on a programmable switch with respect to its limited primitives and, at the same time, must be capable of capturing complex behaviors ?

2. Can we propose a technique to systematically map this abstraction to programmable data plane primitives ?

3. How can this abstraction be applied to detect and mitigate different attacks from Layer 3, Layer 4 and Layer 7 ?

4. Can we combine a set of abstractions to ensure a modular system to handle sophisticated attacks such as multi-step attacks?

In summary, can we leverage the advantages of a programmable data plane, including real-time and stateful processing, and fill its lack in the definition of a stateful abstraction by proposing a generic and stateful abstraction intended for attack detection and mitigation? The abstraction should be generic enough to support different security functions and allow a user to focus on its definition and not on the data plane implementation details.

1.3 Contributions

In this thesis, two main contributions are presented. These contributions define new security functions to detect attacks in a programmable data plane:

1.3.1 EFSM-based approach for attack detection in the programmable data plane and its applications

The first proposal of the thesis (Figure 1.3 in green) is a stateful abstraction based on EFSM models capable of capturing and tracking different attack behaviors. EFSM can capture stateful and complex behaviors, and at the same time, its simplicity makes possible its mapping to switch primitives and thus enables offloading layer 3 and layer 4 attack detection functions to a programmable data plane. We present the approach based on an EFSM to model a behavior and a technique to map this model to P4 data plane primitives to support in-network flow tracking with real-time attack detection and mitigation. A P4-based security function is generic enough to be applied to different use cases and deployed on different targets, as the language is based on a target-independent architecture. Hence, our solution is agnostic to the type of the network. We have identified two attacks that exploit the IP and TCP protocol, namely the ECN attack and the Optimistic ACK attack. Unlike state-of-the-art solutions, we present the application of the proposed approach to detect these attacks within the network without requiring the protocol implementation modification.

1.3.2 Modular approach for multi-step attack detection in the programmable data plane and its application

The EFSM-based security function proposal detects one specific attacker behavior. However, network attacks are becoming more sophisticated and can pass through a set of steps before reaching the final goal. These attacks adopt a step-by-step approach to increase the possibility of getting undetected. Therefore, it is important to identify which step has been achieved and in which order the steps are performed to determine the severity of the attack and set the appropriate reaction dynamically and in real-time. The second proposal of this thesis (Figure 1.3 in grey) provides a solution that can monitor a compound attack within the network. The approach separates the detection module from the decision module, allowing operators to define decisions and reactions on attack progression at run-time (*i.e.,* using a controller API to change

the configuration on-the-fly). The approach detection module is based on an EFSM model, and the decision module relies on a Petri-Net that synchronizes the attack step detection and allows setting different mitigations at line-rate based on the composition of the attack steps. We demonstrate the efficiency of the approach in detecting the recent multi-step DNS cache poisoning attack entirely in the data plane, in contrast to existing solutions that require modifications of the DNS protocol and much time for deployment (*e.g.,* DNSSEC).

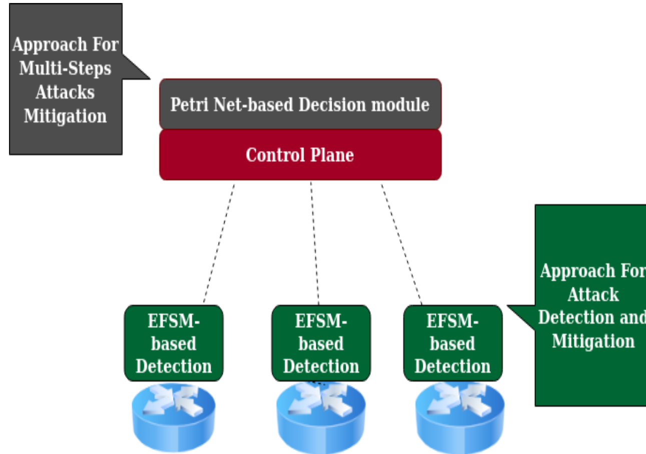


Figure 1.3: Thesis contributions

1.4 Thesis outline

In Chapter 2, in the first part, we present the necessary background for this thesis, the history of programmable networks, including active networks, SDN technology initially proposed with a programmable control plane, then evolved to a programmable data plane. In the second part, we study and analyze the different abstractions that have been proposed to perform stateful processing in data planes. From this analysis, we identify the shortcomings and limitations of each proposal. We also present the different applications that have been offloaded to the programmable data plane, focusing on applications for network monitoring and security. We show that the main limitation is that the proposed solutions are only dedicated for a specific application.

Chapter 3 is dedicated to our first contribution, a stateful abstraction for attack detection in programmable data planes. The solutions studied and analyzed in Chapter 2 have some limitations. In this chapter, we propose an approach based on a stateful model (EFSM) that will serve as a general model for attack behaviors. To achieve the goal of flow tracking at line-rate, we define a new technique for mapping an EFSM-based abstraction to a data plane programming language. We demonstrate the effectiveness of the proposed approach to detect two attacks that exploit the TCP protocol and evaluate the performance of our solution in detecting these attacks.

In Chapter 4, we present an approach to detect multi-step attacks. We propose the Petri-Net based solution that synchronizes the detection of the attack steps, and we present how the Petri Net model will be mapped to primitives supported by the P4 data plane. We validate our solution with an application to the DNS protocol.

At the end of this manuscript, we give the general conclusion and the perspectives of our work in Chapter 5.

Chapter 2

State of the art

Sommaire

2.1 Introduction **10**
2.2 Programmable network history **10**
2.3 Programmable data plane architecture, language, and targets **12**
2.3.1 Programmable data plane architectures 12
2.3.2 P4 language 14
2.4 Stateful data plane **19**
2.4.1 Stateful data plane abstractions 19
2.4.2 Stateful data plane processing 25
2.4.3 State management 26
2.5 Applications offloaded to the data plane **27**
2.5.1 Network monitoring 27
2.5.2 Network security 29
2.6 Summary **32**

2.1 Introduction

With the increase in Internet traffic and application requirements (*i.e.,* in terms of low latency and high throughput), network technologies aim to make networks more flexible, configurable, and programmable. The performance and availability of network services are strongly related to the underlying network architecture and technologies. The underlying architecture must provide scalability and real-time processing; it must process large traffic volumes at line rate and low latency. To support in-network applications with real-time and stateful constraints, we need models that can be deployed on network elements with respect to their limited primitives.

At the same time, there are many security concerns in networks, such as (D)DoS, scan, botnet, etc. Thus, depending on malicious behaviors, network failures, or congestion, the network should be able to react autonomously by rerouting, dropping, or modifying network traffic. To make networks more efficient, network devices should be more programmable to control their packet processing to meet network requirements, security, and flexibility needs.

Languages for controlling and defining packet processing logic and stateful abstractions have been proposed to support stateful processing, thus, enabling offloading network monitoring and attack detection functions to the data plane (*i.e.,* network elements, switches). This chapter summarizes the relevant background literature and several prior works that have been proposed to make networks more programmable and simple to manage. The first section addresses the history of programmable networks, including the background of the P4 language used to program network devices. This is followed by the literature support for stateful abstractions in the SDN data planes. Then we review the applications that have been offloaded to the data plane with a focus on network monitoring and security applications. This chapter concludes by discussing and analyzing proposed approaches and solutions shortcomings.

2.2 Programmable network history

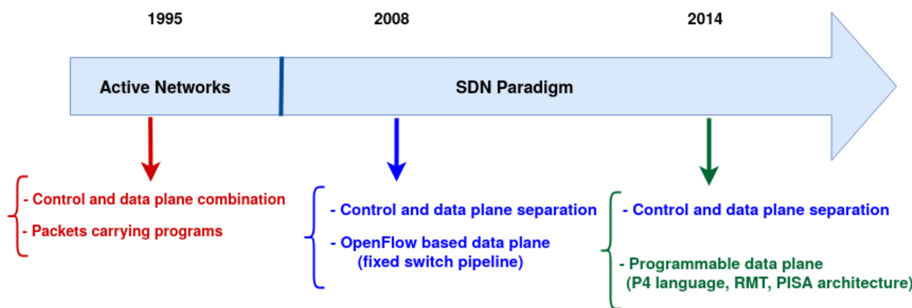


Figure 2.1: Programmable networks history

In networks, a plane represents the place where specific processing occurs; packets are processed using a control and a data plane. The roles of the two planes are as follows: the control plane defines the rules to be applied on ongoing packets (*i.e.,* how packets are forwarded on network paths), the control plane also perform complex and heavy functions (*e.g.,* arithmetic operations); in return, the data plane forwards packets according to the decisions made by the control plane.

Networks have been complex and challenging to manage; therefore, enabling network programmability was the key to simplify network management. In 1990, active network paradigm [1] was introduced by promoting a programming interface with storage, processing, and packet queue functions on the network elements, thus, giving the possibility to apply processing on packets passing through the network. In active networks, packets carry programs and network devices execute these programs upon their reception. In these networks, switches (*i.e.,* network elements) consist of the control plane and the data plane (*i.e.,* forwarding plane). However, associating the processing with network packets introduced a lot of overhead (*i.e.,* increased traffic volumes in the network) [2].

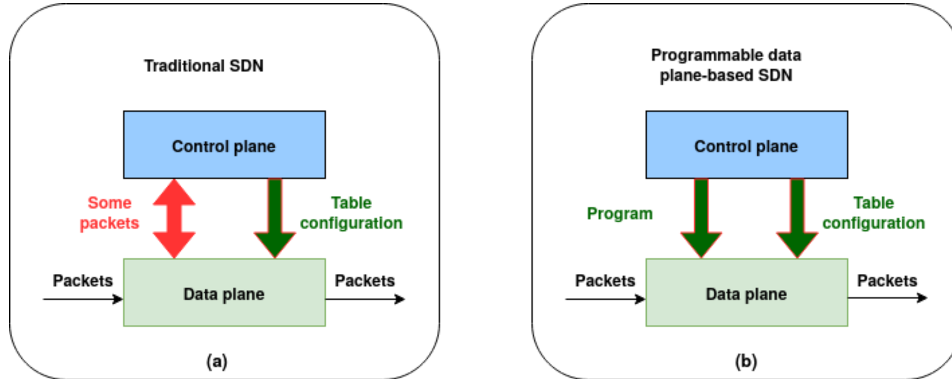


Figure 2.2: Traditional vs. Programmable SDN data plane

The increase in network traffic in 2000 had led to new approaches for network management [2]. Logically centralizing the network control was the main proposal (*i.e.,* enabling a centralized control and management of resources). Software-Defined Networking (SDN) was proposed [2], the basic idea behind SDN is to make the network programmable (*i.e.,* more capable of handling undergoing changes related to dynamic service demands, of controling network behavior, and configuring forwarding rules). It took an important step for network programming by separating the control role from the forwarding role (Figure 2.2 (a)). In order to reduce network complexity, the control plane is deployed out of the data plane element on distant servers, it represents the network intelligence allowing the control of the network's behavior as a whole, which facilitates its management and evolution. SDN implements a programming interface named OpenFlow [3] between the data plane and the control plane, thus, providing applications with the necessary means to control the network. The control plane has a global view of the network and is responsible for generating forwarding rules according to network traffic information (Figure 2.2 (a), table configuration). In return, data plane switches perform packet forwarding based on controller rules. Therefore, the control plane is responsible for network state management, while the data plane is responsible only for packet forwarding, which is qualified as the *stateless* data plane. Even if OpenFlow-based switches have new functionalities and primitives, they are still dumb while all the intelligence is placed at the control plane. It introduces a fixed match-action paradigm with fixed actions and packets header fields to match (*i.e.,* uses a small set of pre-defined header fields and actions which is more vendor-dependent). Thus, packet headers and actions cannot be extended, which reduces the custom processing capabilities of the packets. OpenFlow is limited in supporting stateful processing (*i.e.* limited to counters), since it lacks the possibility of maintaining state information inside switches. Therefore, the restricted expressiveness of OpenFlow limits the data plane's programmability and flexibility.

As a result, it cannot reach the aim of a fully programmable network (*i.e.,* support new protocol definitions, customized packet headers and actions definition, stateful packet processing in the data plane without the controller involvement). The stateless nature of the OpenFlow-based data plane and the separation between the control plane and the data plane result in frequent interactions between the two planes to report packets. As presented in Figure 2.2 (a), some packets are sent to the controller, which introduces an additional processing delay (*i.e.* the controller has an explicit involvement in stateful processing and/or forwarding rules updates).

Using OpenFlow, a stateful application requires continuous controller involvement. For example, analyzing the distribution of packet flows (*e.g.,* identifying the amount of data sent by a source) using Shannon entropy [4] would require a periodic packet report to the controller. The controller performs complex operations and state management since the data plane is stateless, thus introducing the overhead of periodic packet reporting to the controller.

The stateless nature of the OpenFlow data plane and its limitation of expressiveness have driven recent efforts to enhance data plane programmability and introduced the proposal of making the data plane more programmable and stateful (*i.e.,* more intelligent network elements to track and manage states in the data plane) to limit the explicit interaction with the control plane. A programming language (P4) [5] was proposed to program data plane functions and support user-defined packet processing functions and protocols. P4 supports stateful processing based on specific structures and enables the offload of a large number of functions onto the data plane (e.g., attack detection). As shown in Figure 2.2 (b), once programs are compiled, packets are not forwarded to the control plane. This language will be more detailed in the next section.

Figure 2.1 summarizes programmable networks history from the active network where packets transfer the programs that the switches will execute, followed by the introduction of the SDN paradigm that separated the data plane from the control plane, the first proposal of the SDN paradigm introduced a data plane based on OpenFlow. Finally, the programmable data plane was introduced, enabled by the P4 switch programming language built on an architecture like RMT [6] and PISA to provide more flexibility, these architectures will be detailed in the next section.

2.3 Programmable data plane architecture, language, and targets

As mentioned in the previous section, the idea of making the SDN data plane more programmable was adopted to overcome OpenFlow-based SDN shortcomings. This section presents the proposed architectures to make network switches more programmable, the language used to program network switches, and the compatible hardware and software targets.

2.3.1 Programmable data plane architectures

The data plane architectures act as an intermediate layer between hardware targets and a switch language (*i.e.,* P4), They describe the common capabilities of the targets, either software or hardware. Therefore, programs developed for a generic and common architecture can be compiled on targets supporting it. This section describes the underlying target architecture of the P4 language, named RMT and PISA.

2.3.1.1 RMT architecture

To overcome the limitations of OpenFlow that only allow matching fixed header fields and propose limited actions, the Re-configurable Match Table (RMT) architecture [6] was defined to reconfigure switch pipelines (*i.e.,* Match Action Tables (MATs)). New header fields and new actions can be defined, the width of the match tables and queuing disciplines can be specified (*i.e.,* to place a packet in a specific queue). RMT is a flexible switch match-action pipeline architecture compared to the traditional OpenFlow-based rigid switch. It enables Match-Action Tables configuration in four ways: new fields can be added (*i.e.,* keys to match); the width and number of tables can be specified; new actions can be defined, the packet placement in queues with specific queuing disciplines can be specified. This architecture allows modifying the forwarding pipeline to define Match Action Tables by allowing a set of stages, each containing MATs with different widths.

2.3.1.2 PISA architecture

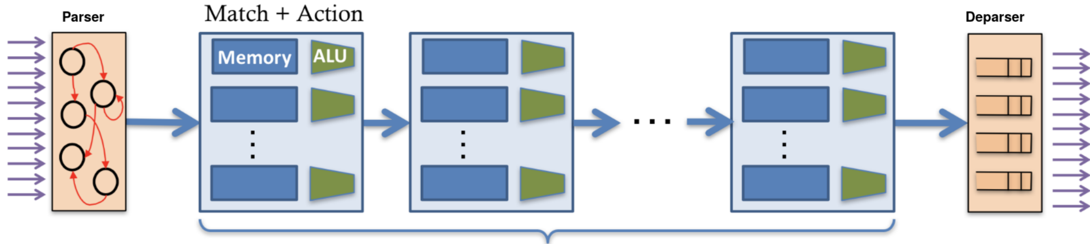


Figure 2.3: PISA architecture [7]

PISA (Protocol Independent Switch Architecture) is an architecture for high-speed programmable packet forwarding that describes the common capabilities of P4-based switch targets. PISA generalizes the RMT architecture, which gives more hardware details (*i.e.,* number of ALU, memory management, crossbars). Programs based on the PISA architecture are portable across many targets that support this architecture. To enable data plane programmability, the architecture consists of a programmable parser, deparser and a match action pipeline between the parser and deparser composed of multiple stages (Figure 2.3). PISA is the first proposed architecture related to the P4 specification. Recently, the P4 consortium defined a new architecture, called the PSA architecture [8]. PSA is similar to PISA with the possibility for vendors to implement custom extensions (*e.g.,* hash and random generator actions can be precisely specified). It provides more precision about multi-cast and packets re-circulation and gives the possibility to have custom bit widths. However, PSA is still a work in progress and remains a theoretical model, as the vendors are not providing a compiler for PSA-based programs to their targets [9]. In what follows, we detail the PISA architecture components.

Parser: it allows users to define customized headers to be extracted from incoming traffic. After a packet is parsed, the headers are converted to metadata. In a PISA pipeline, metadata consists of packet headers, intrinsic metadata, and user-defined metadata; only metadata are processed in a switch. These metadata are destroyed after each packet is processed. PISA components (parser, MAT, deparser) communicate using metadata. Two types of metadata are specified. First, *User-defined metadata* are defined by users to be used during the packet processing pipeline; these metadata are temporary and associated with the processing of each

packet (e.g., metadata to maintain a hash result). Second, *intrinsic metadata* which have the same logic as user-defined metadata (*i.e.,* are used in per-packet processing pipeline). However, these metadata are fixed and defined by the language or architecture (*i.e.,* vendor-specific such as queue occupancy, port utilization)

Match-action pipeline: it includes a set of match-action units. Each unit is composed of one or multiple MATs that match packet header fields and/or metadata and execute the corresponding action. MAT is composed of SRAM (static random-access memory) or TCAM (ternary content addressable memory) to maintain match lookup keys and actions. Action logic is implemented using ALUs (Arithmetic Logic Units) to perform header modification and arithmetic operations. Stateful structures that maintain states, such as registers, meters, and counters are implemented using SRAM.

Deparser: it reconstructs the outgoing packet by reassembling its headers.

PISA architecture can include a component that can be target-specific. For this reason, PISA proposes *externs* that are general enough to be supported across different targets.

2.3.2 P4 language

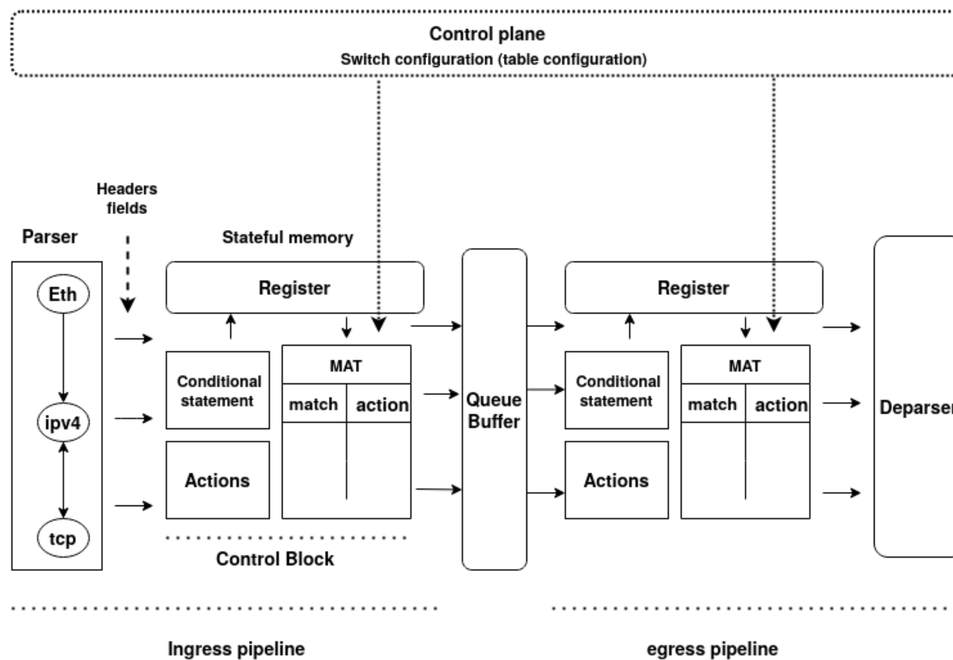


Figure 2.4: P4-based switch pipeline

P4 [5] (Programming Protocol Independent Packet Processor) was introduced in 2014 and is a data plane programming language based on the PISA architecture (*i.e.,* PISA is the P4 programming architecture). It defines how programmable forwarding elements process and handle packets (*i.e.,* software or hardware switches). Software or hardware targets have a control and a data plane; P4 specifies data plane functionalities and how the control plane can interact with the data plane (*e.g.,* P4 defines MAT (including match keys and actions), so that the control plane can populate these tables using specific CLI (Command Line Interface). Thus,

P4 allows programming a data plane target. In contrast to a traditional switch, in a P4-based switch, functionalities are defined by the program and not fixed in advance; tables and packet headers are not fixed, and new headers fields and actions can be defined. As a result, P4 gives the possibility to define and express a set of customized behaviors during packet forwarding. Programs are compiled into a P4-enabled switch. In the first place, a P4 program is converted into a high-level intermediate representation (HLIR). Second, this representation is converted into a target-independent program.

Upon receiving a packet by a P4 based switch, the P4 parser (Figure 2.4 parser) extracts the defined packet headers and intrinsic metadata (e.g., ingress timestamp, ingress port number). These metadata are defined in the parser with packet headers. Once the packet headers are parsed, the metadata and headers are used in the control block (Figure 2.4 control block including MAT, registers, conditional statement, and actions). During the control block pipeline processing, actions can be performed, such as headers and metadata modification. In the control block also conditional statement can be performed along with customized actions. Only parsed metadata are processed in the pipeline, and the packet payload goes separately to the deparser process. Once a packet leaves the pipeline, the deparser (Figure 2.4 deparser) gathers back the packet with the packet payload along with intrinsic metadata.

In what follows, we overview the P4 core primitives presented in Figure 2.4 (*i.e.,* packet processing pipeline logic) including the parser, the control block (MAT logic, conditional statement, actions) and the deparser.

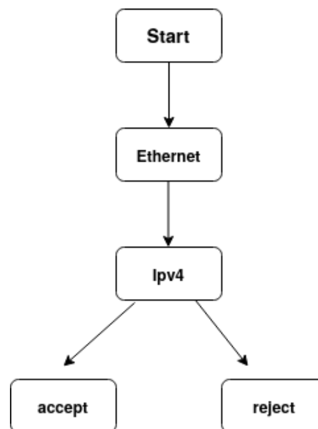


Figure 2.5: Parser FSM

Parser: P4 allows programmers to define customized packet headers to be parsed without a specific predefined format. A parser (Figure 2.4 Parser) describes header reception sequence, how headers are identified and which packet fields to extract. P4 represents the parser as a finite state machine FSM; the state machine execution starts with a start state, a set of intermediate states, and ends with an Accept or Reject state. In Figure 2.5 we present a P4 parser example for Ethernet and IPv4 headers, Figure 2.6 describes the corresponding P4 parser source code. The parser state machine starts from the *start* state and goes to the Ethernet state. In this state, the Ethernet header fields are extracted. If the Ethernet type field contains the IPv4 code, the parser state goes to the IPv4 state to parse the IP header fields, following the same logic, in the case of TCP or UDP packets. If the IP protocol field contains the TCP or UDP protocol, a transition to a TCP or UDP state is triggered to extract the TCP and UDP header fields.

```
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            16w0x800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
    state start {
        transition parse_ethernet;
    }
}
```

Figure 2.6: Parser program

Controls blocks: they are composed of conditional statements, actions, and MATs (as presented in Figure 2.4 Control Block).

- **conditional statement**: we note that P4 does not allow defining loops; instead, conditional statements can be used to express predicates.

- **actions**: they represent the program fragment to describe the header fields and metadata manipulation. Actions in P4 are functions that modify, read and write metadata and packet headers. P4 also has specific actions to read and write register values. These actions are called in the if statement or MAT actions. P4 defines a set of arithmetic and logical operations (e.g., addition +, subtraction − , concatenation ++)

- **Match Action Tables (MATs)**: each table contains a match key on packet headers or/and metadata fields and the corresponding action if the match succeeds; the programmer-defined keys are associated with actions in tables. Each table definition contains the table size in terms of the maximum number of entries and the list of possible actions that can be executed. Table entries are populated using a control plane API. Figure 2.7 presents an example of a MAT structure with lookup keys (*i.e.,* a lookup key can be a packet header fields and/or metadata such as the destination port number and the applied action can be a forward action). Figure 2.4 presents the MAT population by the control plane component.

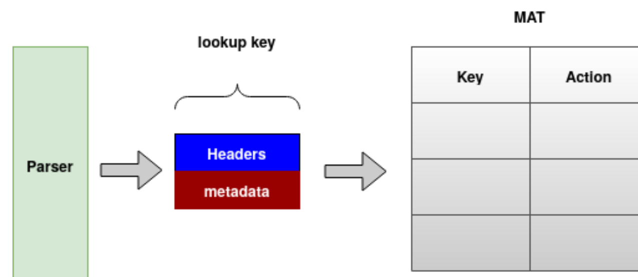


Figure 2.7: Match Action Table

P4 also proposes stateful memories named **registers** used to maintain state across packet flows. These registers are accessed or updated using actions in a control block. P4 gives the possibility to programmers to extend the available actions. It presents *Externs* objects that are architecture-specific and can be used in a P4 program, such as a pseudo-random number generator function. We note that since the latest version of P4 (*i.e.,* version $P4_{16}$), registers are considered as externs.

```
control ingress(inout headers hdr, inout metadata meta, inout standard_metadata_t
standard_metadata) {
    action _drop() {
        mark_to_drop(standard_metadata);
    }
    action set_nhop(bit<32> nhop_ipv4, bit<9> port) {
        meta.ingress_metadata.nhop_ipv4 = nhop_ipv4;
        standard_metadata.egress_spec = port;
        hdr.ipv4.ttl = hdr.ipv4.ttl + 8w255;
    }
    action set_dmac(bit<48> dmac) {
        hdr.ethernet.dstAddr = dmac;
    }
    table ipv4_lpm {
        actions = {
            _drop;
            set_nhop;
            NoAction;
        }
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        size = 1024;
        default_action = NoAction();
    }
    table forward {
        actions = {
            set_dmac;
            _drop;
            NoAction;
        }
        key = {
            meta.ingress_metadata.nhop_ipv4: exact;
        }
        size = 512;
        default_action = NoAction();
    }
    apply {
        if (hdr.ipv4.isValid()) {
          ipv4_lpm.apply();
          forward.apply();
        }
    }
}
```

Figure 2.8: P4 basic code

The control block contains an apply block to invoke actions, if statements, and MATs using apply() methods. A basic P4 code for packet forwarding is given in Figure 2.8, including a control block with actions, tables definition, and an apply block that defines an if statement and tables call for packet forwarding. Figure 2.8 code includes a control block with two tables, a table *forward* that contains two actions, a forward action named *set_dmac* that updates the MAC destination address, and also a drop action. The second table named *ipv4_lpm* performs the longest prefix match on the IP address and has two actions: the *set_nhop* action that specifies the egress port and increments the TTL and a drop action. In the apply block, both tables are invoked with the *apply*() method if the received packet is valid.

Deparser: packets are reassembled at the end of processing by the P4 deparser. Figure 2.9 shows a deparser code with the emit() method that reassembles the Ethernet and IPv4 header.

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

Figure 2.9: P4 deparser program

P4 is considered to be target independent and can be implemented on various targets because hardware or software manufacturers provide P4 compatible compilers (*i.e.,* based on the PISA architecture for their target) [5]. It provides a high-level and separate language to cover different hardware architectures. It presents a set of benefits, as it gives more flexibility for network programming by giving the possibility to express customized functions in contrast to traditional fixed-function forwarding policies. P4 enables portable functions across targets (*i.e.,* target independent) implementing the PISA architecture (*i.e.,* with the assumption of sufficient resources). P4 compiler hides the low-level details of the target, so each target manufacturer must make available a compiler for their target.

P4 targets can be software or hardware (*i.e.,* after compilation, the P4 program is executed on a hardware or software target). Software targets run on CPUs, in contrast to hardware targets. The most freely available software target for the P4 programs is the bmv2 switch [10], the Tofino ASIC is the hardware target.

2.4 Stateful data plane

As mentioned in the previous section, OpenFlow was proposed as a communication protocol between the control and data plane in an SDN architecture. However, OpenFlow is incapable of keeping information in the data plane and is incapable of performing customized packet processing. In recent years, works have been proposed to enable stateful processing in the data plane. This section reviews the most relevant works that have been achieved to improve data plane programmability and to support stateful processing by extending existing architectures or proposing stateful abstraction based on existing architectures. Before summarizing the existing work, we classify the states into two different kinds. A **per-packet state** is monitored for each packet and maintained in metadata, for example, port utilization or queue occupancy. A **per-flow state** is used to maintain flows state (*i.e.,* state of packets belonging to the same flow) and is maintained using a stateful structure such as registers and SRAM; for example, to track the number of packets belonging to a specific flow for a DDoS detection or the last seen acknowledgment number.

2.4.1 Stateful data plane abstractions

A major limitation of OpenFlow is its incapacity to support stateful flow processing in switches. To enable data plane programmability and stateful processing, the addition of abstractions (*i.e.,* abstractions that describe a stateful behavior) to maintain per-flow states inside the packet's path is required. In this section, we survey existing stateful abstractions in the data plane. Abstractions are based on state machines that can be used to model stateful behaviors (Finite State Machine (FSM), Extended Finite State Machine (EFSM)) or directly based on the OpenFlow match action pipeline. Briefly, FSM describes only input events (represented as Boolean inputs), and output events are determined by a current state. The passage from one state to another is directed by a *transition* triggered by *events*; *states* representing a given behavior are linked using *transitions*. FSM are not scalable; all states need to be explicitly defined. In contrast, an EFSM is not limited to states and events, variables can be defined, and transitions can occur based on conditions. These two models will be more detailed in the next chapter. In what follows, we summarize the proposed abstractions. In Table 2.1 we review the proposed solutions based on the model that is used (FSM or EFSM), whether the abstraction handles the per-flow state or not. We also categorize the proposed abstractions regarding a possible communication with the controller and the possible target (hardware (HW) or software (SW)). Finally, we summarize the limitations of each proposal.

The proposed abstractions named **FAST** [**11**], **SDPA** [**12**], **OpenState** [**13**] and **FlowBlaze** [**14**] extend a switch pipeline to map a state transition system (*i.e.,* to maintain state information inside a switch and based on state information and network events, perform actions). FAST, SDPA and OpenState extend the OpenFlow pipline, FAST and OpenState proposed a data plane abstraction based on FSM models, SDPA introduced a new component (*i.e.,* a processing unit in switches), whereas FlowBlaze proposed an EFSM based abstraction. These abstractions map a (state, event) pair into a (state, action) pair. The principle of stateful abstraction in a data plane can be described in two main aspects:

- Aspect 1 : maintaining state information within switches
- Aspect 2 : depending on the state information and network events (*i.e.,* header fields of incoming packets), actions are performed on packets, which is called *state transition.*

To support stateful processing inside switches, the abstractions use a sequence of MAT tables, mainly a table to maintain states (*a state table*) (Aspect 1), the table matches an identifier (*i.e.,* packet header fields) and the state. A second table is used to perform the state transition (*a state transition table*), which based on packet header fields and state labels, executes the corresponding action (Aspect 2).

Upon receiving a packet a stateful abstraction within a switch pipeline passes through a set of steps:

- Once a packet is received, a state lookup is performed to retrieve the state (*i.e.,* using the state table)
- In a second time, a match on the state and packet header fields is performed to find the corresponding action (*i.e.,* using the transition table)

A *state transition* is performed based on different table implementations that we will detail for each abstraction. In what follows, we provide the main differences between the proposed approaches.

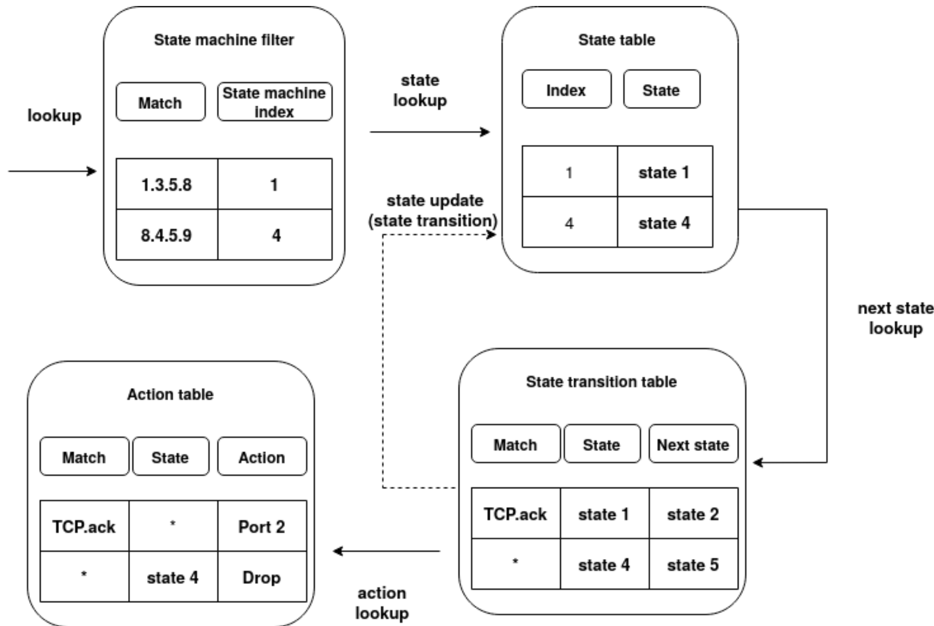


Figure 2.10: FAST tables [11]

FAST [11] proposes a hybrid system by coupling the control plane and the data plane to maintain the state. The control plane is composed of a compiler and a switch agent that communicate with switches during state machine execution and may perform a part of the state machine because switches have limited capabilities; thus, this agent receives packets and collects statistics from switches periodically. As shown in Figure 2.10 four tables are used to perform stateful processing. For state tracking (*i.e.,* Aspect 1) two tables are used, in addition to the state table that maintains states, a separate table is used to filter and identify packets, the first table associates packet fields to an index, and the second table associates the same

index to a state. The state transition is performed using two tables, the *state transition table* that based on the packet fields matches, and the state sends the received packet to the second table named *action table* to perform actions. **SDPA [12]** extended OpenFlow by introducing a new component to manage and maintain state, which is a co-processing unit in switches, that can be implemented using a CPU. The component consists of a *Forwarding Processor (FP)* block composed of three tables (represented in Figure 2.11), a state, a transition, and an action table. SDPA such as FAST uses two tables to perform the state transition (*i.e.,* one table to match packet header fields and the second table to execute actions). The design extends the OpenFlow instructions to direct packets between the OpenFlow forwarding table and the FP. **Openstate [13]** also such as FAST and SDPA extends OpenFlow with two tables (presented in Figure 2.12), the *State Table* along with an *XFSM table* that based on the states and events of received packets define the state transitions. Unlike FAST and SDPA, it uses a single table to perform the state transition and the action execution. In particular, OpenState forwards a packet to a specific port when a match occurs. In the same direction, **FlowBlaze [14]** was proposed to implement stateful processing on a specific hardware (*i.e.,* NetFPGA); therefore, it is hardware-specific. The abstraction is based on EFSM and uses a state table to maintain state (i.e., each entry in the table is associated to a timeout) and a transition table to implement the state machine, FlowBlaze introduces an update function to update state.

FAST is limited by the fact that it depends on a switch agent to perform complex operations (*e.g.,* arithmetic computations). The switch agent is an online component that manages the state machine and may perform part of it, which adds delay for the interaction with the agent based on the switch updates (*i.e.,* the switch agent receives packets periodically). Furthermore, FAST and SDPA need to use more than one table to perform state tracking and transition.

In the SDPA design [12], during runtime, the first packet of a flow is sent to the controller to identify which applications should process this flow. Besides, packets are redirected from the OpenFlow forwarding table to the FP which may add addional delay. SDPA extends the OpenFlow API to communicate messages between the controller and the FP including operation on state (*i.e.,* modification of state table entries (to add, modify, or remove states), therefore, the controller has full control over the FP. Furthermore, SDPA proposal is a hardware-specific implementation for FPGA.

OpenState, SDPA and FAST propose simple abstraction that are not sufficiently expressive and support limited actions (*i.e.,* limited by the OpenFlow primitives), therefore, not expressive enough to capture complex behavior. All states have to be explicitly enumerated (*i.e.,* due to the limited OpenFlow primitives and FSM-based abstractions that are not scalable, which may lead to a state explosion). If N states are possible, at least N table entries need to be created (*i.e.,* many table entries are required to track state). Another issue with OpenState and FAST is flow identification, the association of a state with a flow is not presented; only the per-packet state is tracked. With such limitation, bi-directional flow identification and monitoring become restricted.

FlowBlaze has been proposed to fill the gaps of OpenState and FAST, which did not define per-flow tracking, and that have been proposed to be deployed on an OpenFlow-based switch which is not enough expressive. FlowBlaze proposes an abstraction that focuses on how the abstraction could be mapped to specific hardware with its specific constraints. Thus, it discusses specific hardware constraints and performance. FlowBlaze is designed with a focus on

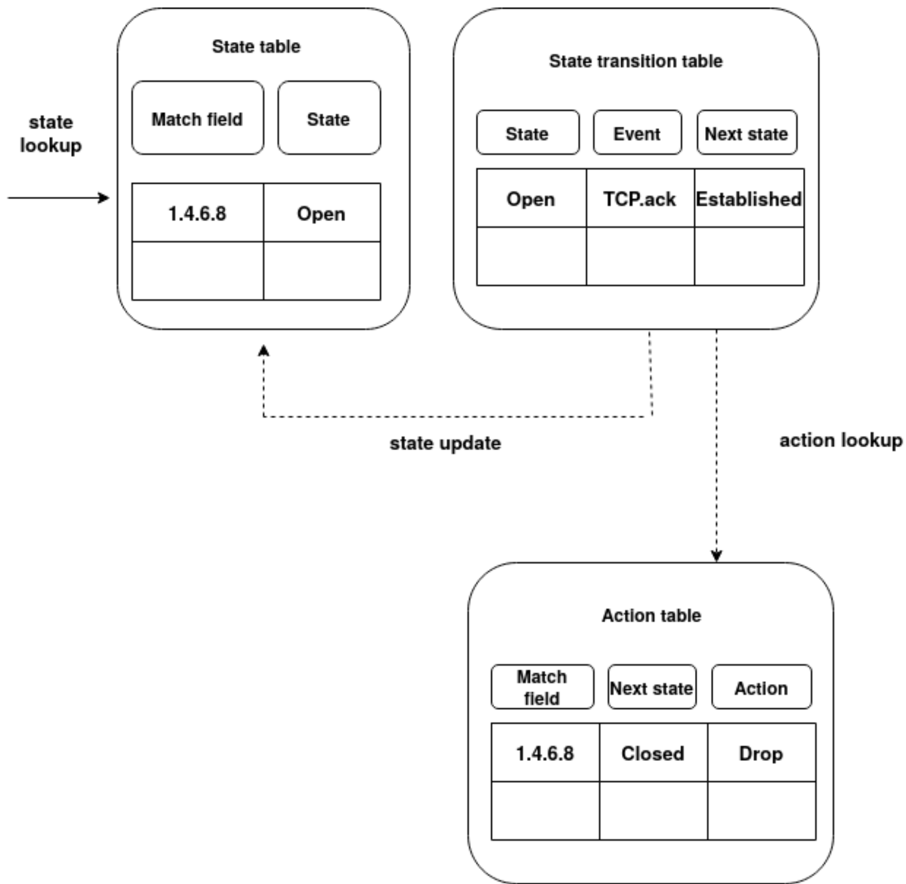


Figure 2.11: SDPA tables [12]

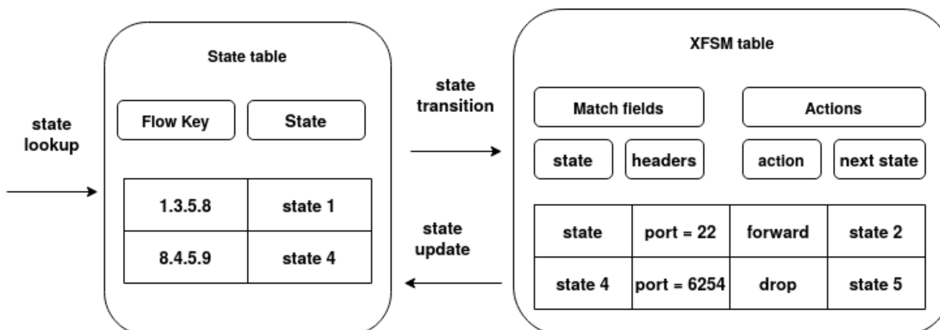


Figure 2.12: Openstate tables [13]

performance optimization of specific hardware without considering different and large hardware deployments. It proposes an abstraction on top of NetFPGA hardware, thus offering a vendor-specific abstraction (*i.e.,* requires some hardware design expertise).

Table 2.1: Stateful abstraction classification

Work	Stateful model	Per-flow state	Control plane communication	Target	HW/SW	Limitations
FAST	FSM	no	rule installation, state update, auxiliary storage and calculation	OpenFlow-based switch	SW (Open vSwitch)	state explosion (enumerate all state labels, no state update operation), communication with a switch agent that partially monitor state
SDPA	N/P	yes	state management	OpenFlow-based switch	SW (Open vSwitch), HW (ONetCard)	state explosion (enumerate all state labels, no state update operation), communication between the controller and the FP that maintain state information
OpenState	FSM	no	rules installation	OpenFlow-based switch	OpenFlow 1.3 Software Switch	state explosion (enumerate all state labels, no state update operation)
FlowBlaze	EFSM	yes	no communication	NetFPGA	HW (NetFPGA)	limited to a specific hardware platform

In summary, Table 2.1 presents different abstractions and OpenFlow extensions to support stateful packet processing in the data plane. FAST and OpenState propose an abstraction based on FSM for OpenFlow-based switches. However, the major limitation of FSM is the state explosion issue, these abstractions are limited because all states must be enumerated. Therefore, to pre-install state machine instances in switches, each state machine needs its own table entries. FAST requires an agent involvement in the control plane to perform a part of the state machine implementation or complex operations. Furthermore, the limited OpenFlow operations also require enumerating all existing states in the case of SDPA and the involvement of an agent in switches for state processing. However, FlowBlaze proposes an abstraction based on EFSM to support per-flow monitoring without a state explosion, yet FlowBlaze is dedicated to a specific hardware target.

2.4.2 Stateful data plane processing

In the previous section, we presented stateful abstractions based on finite state machines and OpenFlow Match-action tables. However, another field of work has proposed to program switches to perform particular stateful operations, relying on specific languages and tools, such as performing packet filtering (*e.g.,* using Berkeley Packet Filter (BPF)) or tracking time-related events (*i.e.,* timer function that fires based on a received event) or specific programs code based on specific architecture to support stateful processing. These works propose atomic primitives to implement a desired function and logic. In this section, we summarize these works.

Oko [15] extends the OpenFlow switch with a programmable Berkeley Packet Filter (BPF) to enable stateful processing in the data plane. Oko proposes a mechanism to support stateful NFs in the OpenFlow pipeline by proposing a stateful filtering in the flow processing pipeline. However, Oko is exclusively limited to Linux-based software switches.

XTRA [16] proposes to offload the transport protocol and L4 network functions to the data plane. It addresses the challenge of managing timer events in the data plane by introducing a timer in its design to support TCP-based applications (*i.e.,* it partially leverages the implementation of the FlowBlaze hardware and provides an API and its own language with timers to support transport layer functions inside switches). To do so, XTRA implements a calendar structure for timer scheduling actions and timer firing events. It can be used as a complement to work that processes only incoming packet-related events.

Domino [17] is a programming language that introduces an atomic isolation block for packet processing. Each packet sees only its own processing without interfering with other packets. DOMINO introduces a model (i.e., an architecture) named Banzai. This model claims that each state modification is accessible to the next packet, and states are not shared among different packet processing units (*i.e.,* state variables are not shared between atoms), so operations on one single packet are programmed. Therefore, an *atom* is the structure used to modify and store states (*i.e.,* an atom includes a local state and a circuit to modify state). Thus, the packet processing pipeline is composed of a vector of *atom*. However, P4 gives more advantages by supporting a programmable and flexible parser allowing programmers to define customized headers and protocols (*i.e.,* Banzai model assumes that the packets entering the model are already parsed [18]). Besides, P4 supports extern to support customized function, Domino is a theoretical model and does not have a hardware target that supports the banzai architecture.

2.4.3 State management

The abstractions presented in Section 2.4.1 are designed to be on top of one switch pipeline and target a single switch. However, enabling stateful processing requires managing state, yet monitoring a network in a distributed way remains a challenge. Also, the limited resources of the network elements are still an issue; so the distribution, replication, or migration of states between switches are required. Therefore, solutions that distribute, place state and describe how flows can be routed, or tools that migrate states if necessary, would ease network programming and the deployment of the abstractions presented in Section 2.4.1. In addition, programming the elements of the network in a distributed manner would help to manage complex networks and resource allocation. These solutions are presented in this section:

SNAP [19] is a stateful language that has been proposed to offer a high-level abstraction for state management. It gives a centralized programming model for stateful network-wide applications. In SNAP, a set of switches acts as one state maintainer, it allows users to program an application on the top of one theoretical switch, then a compiler places states across a set of switches. SNAP proposes a traffic routing and state placement algorithm to ensure network-wide stateful packet processing. The proposal is based on an intermediate representation called xFDD (extended Forwarding Decision Diagram). This representation defines the set of state variables and operations for each packet. The compiler is based on a mixed-integer linear program for state placement and routing. SNAP provides a network-wide programming model with a compiler to place programs among switches. SNAP is more oriented toward one big switch abstraction, in contrast to state machine-based abstraction that targets a single switch. Therefore, all abstractions invoked in Section 2.4.1 can be used as targets for SNAP programs.

Swing [20] proposes a state management framework to address memory limitation, it performs state migration among switches by piggybacking states to packets. The state is classified into two types: hard and soft states. Only the hard state is migrated, which can not be reconstructed from the incoming packet at the destination switch. The soft state can be recovered from the incoming packet, for example, a packet arrival time otherwise, the state is considered as hard. Therefore, Swing identifies the soft and hard states in a P4 program to migrate only hard state.

LODGE [21] presents a state synchronization mechanism to ensure consistency for network-wide applications. All switches maintain global states (*i.e.,* state shared among a set of flows), therefore, the mechanism is based on state replication. The state update is reported to all switches by generating and forwarding an update packet. However, the involvement of a controller to ensure state synchronization may introduce a lot of overhead.

LOADER [22] presents an abstraction to program distributed applications, LOADER as LODGE proposes a global state management in the data plane. It introduces consensus management in switches using a distributed algorithm. All switches are notified when the state is updated. Consensus management in switches improves scalability compared to consensus management in the control plane, such as in the case of LODGE.

2.5 Applications offloaded to the data plane

The emergence of the programmable data plane has motivated the offload of certain applications into network elements. There is a large body of applications which have been proposed to be offloaded to a programmable data plane; we will focus only on network monitoring and security applications. These works propose solutions for a specific problem without taking into account a high-level and generic abstraction or paradigm. We divide these applications into two categories (presented in Figure 2.13): first *network monitoring* including load balancing, performance monitoring and measurement, congestion monitoring, and scalable monitoring; second, *network security* including, heavy hitter, DDoS and flooding attacks detection. This section summarizes these use case-specific works implemented in the data plane. All the presented works in this section have been prototyped and written in P4.

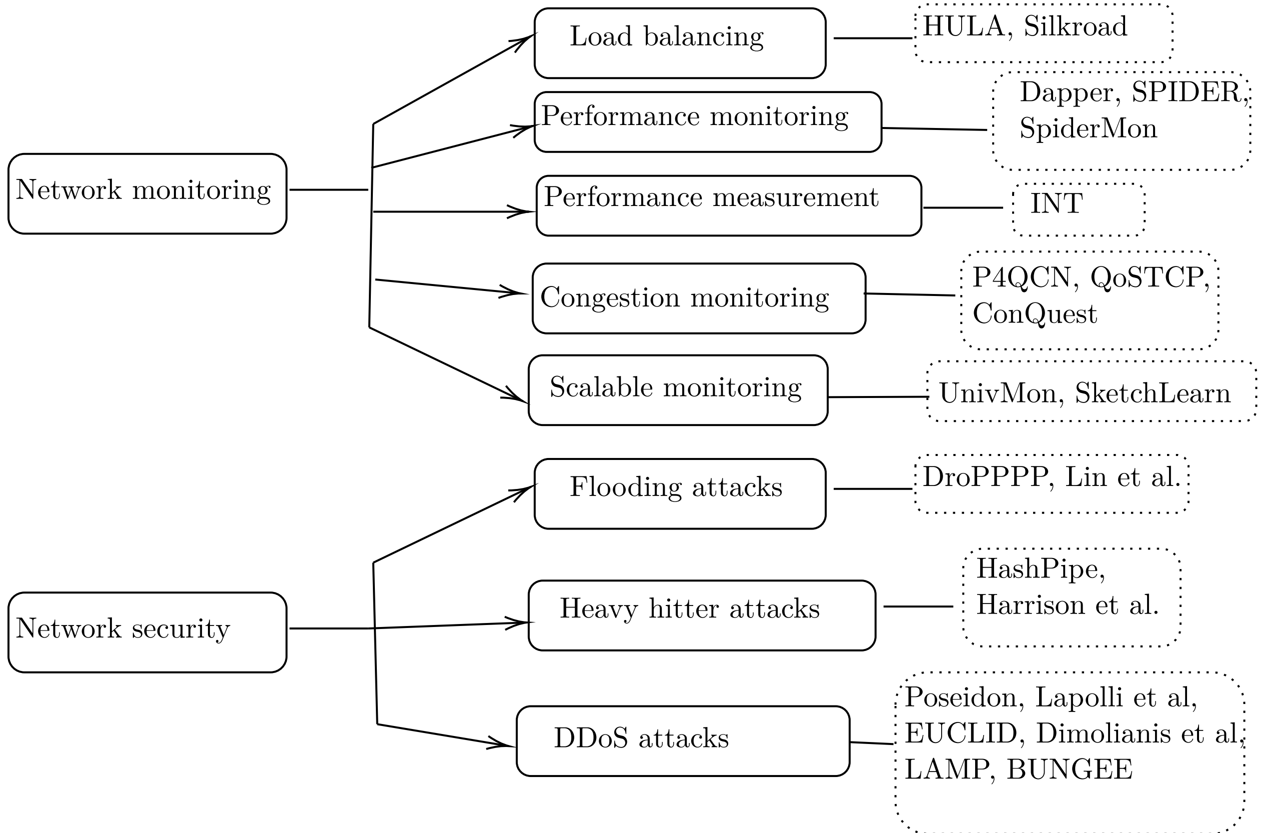


Figure 2.13: P4 applications

2.5.1 Network monitoring

As invoked previously, programmable and stateful data planes have made it possible to offload network monitoring applications to switches. Data centers have experienced a fast increase in traffic, offloading load balancing that performs real-time traffic monitoring to data planes is a good option to support low latency and improve scalability. In addition, congestion control mechanisms with real-time visibility in switches are required to meet the demands of high

throughput networks with low latency. Another field of work monitors network performance in real-time, including identifying bottlenecks and link failures. However, network monitoring applications require manipulating massive traffic statistics, per-flow and per-packet information. The literature leverages techniques to optimize resource utilization, such as sketch structures. In this section, we present these applications that have been offloaded to the data plane, including load balancing, real-time performance diagnostic, congestion control, and sketch-based solutions.

- **Load balancing**

HULA [23] is a load balancing mechanism implemented in the data plane; it is based on the information collected from switches about the less congested paths. It performs a path congestion monitoring to balance traffic accordingly. HULA maintains the utilization of the network link to choose the next hop to a destination. Each switch maintains only the best next-hop to reduce memory consumption. Thus, only the best next hop is calculated instead of maintaining and calculating the entire path. To share link status, switches use probes, including the use of the minimal congested path (*i.e.,* the authors assumed that any P4 supported hardware could be used as a target). In the same direction, **Silkroad [24]** proposes a dynamic load balancing entirely in the data plane that is compiled to the Tofino switch.

- **Performance monitoring**

Dapper [25] introduces a data plane-based TCP performance monitoring to identify the causes of a bottleneck at the network edge (*i.e.,* near end-hosts) by analyzing packets in real-time. Dapper tracks TCP metrics and packet fields such as sequence, acknowledgment numbers, packet sizes, and timestamps to calculate rate loss and path latency and detect congestion. It first monitors simple metrics, second, only flows that require diagnostics are investigated. A controller gathers collected information to identify the limiting entity, the network, the receiver or the sender (*i.e.,* the authors assumed that the P4 code can be compiled to a compatible hardware switch). **SPIDER [26]** is a stateful packet processing pipeline that implements a link failure recovery to reroute flows. SPIDER is based on periodic link checking that allows switches to detect failures, which results in rapid traffic redirection (*i.e.,* the bmv2 switch has been used as a target). In the same direction, **SpiderMon [27]** presents a performance failure monitoring system that detects latency spikes. To do so, it verifies that the accumulated latency exceeds a specific threshold (*i.e.,* based on the queueing delay). The Tofino and bmv2 switches have been used as targets.

- **Performance measurement**

IN-band Network Telemetry (INT) describes the use of programmable data planes to collect hop-by-hop network information. It is a data plane implementation that enables collecting telemetry data while packets traverse the network [28]. INT is used to transfer QoS metrics across a network [29] or to detect latency spikes [30].

- **Congestion monitoring**

P4QCN [31] presents a congestion mechanism. Before forwarding packets, switches verify egress ports to detect possible congestion. If congestion is detected, the packets are forwarded back to the sender so that it can adjust its sending rate. **QoSTCP [32]** proposes an adapted TCP congestion window, the flow rate is limited when it exceeds a specific value. The mechanism is based on a marking approach. These two works are compiled to the bmv2 switch. **ConQuest [33]** introduces a buffer monitoring system that tracks the occupancy of the switch queue to identify large portions of the buffer occupation (*i.e.,* their P4 prototype is implemented on the Tofino switch).

- **Scalable monitoring**

The count-min sketch is a probabilistic structure that serves to identify the frequency of an event (*i.e.,* count the number of elements in a set). Several works implement sketch-based flow monitoring systems on P4 based switches. **Hashpipe** and **UnivMon [34]** propose a sketch-based solution for sampling traffic to identify the heaviest flows (*i.e.,* detect heavy hitter attacks). UnivMon implements their design on the bmv2 switch. **SketchLearn [35]** also introduces a sketch-based solution to track flow frequencies and it is deployed on the Tofino hardware switch. SketchLearn and UnivMon delegate traffic counting to the data plane, using hash tables to maintain summarized counters (*i.e.,* sketches). In return, the values of the sketches are periodically collected by a controller.

2.5.2 Network security

Per-flow and per-packet state tracking in the data plane has enabled the deployment of a set of security applications within traffic data paths, along with monitoring applications introduced in the previous section. Attacks have always been a major threat; flooding and volumetric attacks (*e.g.,* DDoS, heavy hitter), can cause significant degradation of network performance. Traditional attack mitigation solutions (*i.e.,* middleboxes based solutions) add cost and operational overhead. OpenFlow-based solutions require interaction with a distant controller, which incurs overhead and additional delay. Therefore, the mitigation solution should be deployed within the network to address the limitations of existing solutions. This section summarizes some offloaded attack detection applications on the data plane built using the P4 language. Table 2.2 reviews existing solutions to detect specific attacks. We classify them depending on whether they are designed for a specific use case (*i.e.,* with a generic abstraction or not), the detection technique that has been used, how flows are tracked and identified, the stateful structure used to maintain the states, the hardware or software switch targets, and some limitations.

Table 2.2: Stateful attack detection

Work	Use-case specific	Detection technique	Flow identifier	State management	Target	Limitations
DroPPPP	yes	compares addresses with pre-stored values	IP and MAC source address	N/P	bmv2	needs to maintain pre-calculated IP address
Lin et al.	yes	counts the number of SYN/ACK, ACK/FIN and verifies the threshold exceed attempt	N/P	register	bmv2	involves the controller to install dropping rules
HashPipe	yes	maintains flows packet count	5-tuple	hash tables	bmv2	tracks only a set of heavy hitter flows, to achieve high performance it sacrifices accuracy
Harrison et al.	yes	counts packet	source address	register	tofino	consumes memory to maintain per-address state
POSEIDON	yes	counts packet	N/P	count min sketch	tofino	uses external servers
Lapolli et al.	yes	calculates IP address entropy and compare it to a threshold	N/P	register	bmv2	provides complex code that is not feasible for real hardware implementations [36]
EUCLID	yes	tracks the entropy	IP source, destination	count min sketch	tofino, bmv2	depends on external server storage which adds latency
LAMP	yes	end-host adds a tag to packets when the attack is detected, switches drop packets with tags	N/P	N/P	bmv2	modifies end-hosts to manage tags
BUNGEE	yes	analyses the entropy and verifies the threshold exceeding	N/P	N/P	bmv2	sends alert notification to upstream switches which may add overhead

To detect flooding attacks **DroPPPP** [**37**] proposes a spoofed address detection mechanism in the data plane. DroPPPP uses IP source and MAC addresses hash to detect a possible spoofing by comparing each incoming address with a pre-stored hash. **Lin et al.** [**38**] presents a SYN flood and ARP flood attack detection mechanism. The method is based on counting SYN, ACK and FIN packets.

To detect heavy hitter attacks **HashPipe** [**39**] proposes an algorithm in the data plane which uses hash tables to count flows. Only a limited number of flows are maintained; if needed, the flow with the lowest counter is replaced. In the same direction, **Harrison et al.** [**40**] also detect heavy hitter attacks in the data plane.

Many works have been proposed to detect DDoS attacks in the data plane. **POSEIDON** [**41**] presents an approach to detect volumetric attacks entirely in the data plane. It introduces a policy language to define DDoS attack mitigation and defense primitives. It uses a controller to reconfigure the switch and generate a new defense policy. However, attack detection is distributed between switches and servers; POSEIDON relies on external hardware (controller and servers) to dynamically detect attacks. **Lapolli et al.** [**42**] propose to detect DDoS attacks in the data plane. Their mechanism passes three steps, namely, entropy estimation, traffic characterization, and anomaly detection. An entropy scheme is implemented using count-min sketch structures to address memory issues in switches. The attack is identified when a specific entropy threshold is triggered. This work is extended in **EUCLID** [**43**] by introducing a framework to react against the attack, thus integrating the detection presented in [42] and the reaction entirely in the data plane. **LAMP** [**44**] introduces a layer 7 DDoS detection, an end-server detects the attack (*i.e.,* scanning attempt) and sends the attack alert to switches. The IP option field is used to add a flag indicating the presence of the attack to the edge switch responsible for dropping all packets from the attacker flow. **BUNGEE** [**45**] performs an IP address entropy analysis on the data plane to detect a DDoS attack. Once the attack is identified on a switch, the upstream switch is informed to filter attacker packets to mitigate the attack closer to the source.

Although all of these solutions provide a data plane-based solution for a specific use case, including network monitoring and attack detection, a general monitoring approach that can be applied to diverse applications is not considered. The main limitation of these works is that the proposed solutions are for a specific application that monitors specific network traffic or defends against specific attacks. A practical solution for offloading applications to the data plane would be to provide a general abstraction and approach that would be applied to different application use cases.

2.6 Summary

In the literature, different solutions and abstractions are proposed to enable stateful processing in the data plane. Fast and OpenState share the same idea of using FSM to make the forwarding path more stateful, and thus support a stateful application. In the same way, FlowBlaze was proposed, which is a stateful packet processing architecture based on EFSM. It provides an abstraction for application implementation, focusing on one type of hardware as target. However, the extension of OpenFlow to enable stateful processing instead of stateless processing is not effective for flow-based monitoring applications, since it tracks only per-packet state and not per-flow state. Indeed, such an approach requires communication with the controller and generates overhead. Hardware-based abstraction is limited to specific hardware and fabric constraints. However, other works have proposed to offload different applications to the data plane, such as attack detection, with the limitation of one specific application without a general abstraction.

Following the existing solutions line of reasoning, state machines are a good choice to implement stateful processing in the data plane. A lighter solution is required in terms of simplicity (*i.e.,* based on a common language), per-flow tracking, and generality for deployment on a large number of hardware targets. To overcome existing solutions shortcomings, in this thesis, we propose an abstraction focusing on a high-level and a general approach. An EFSM-based abstraction would act as a high-level abstraction to be mapped to an intermediate and portable representation using a programming language such as P4. In addition, a P4-based abstraction can be common and applicable to many targets and general enough to be applied to different applications. The proposed approach would mask hardware constraints, since the P4 language can be compiled on any hardware that supports the PISA architecture.

A common model and a general abstraction for the data plane would be beneficial for implementing different and complex attack detection applications on different data plane targets. In this thesis we focus on network security functions, an attack detection solution entirely deployed in the data plane would benefit from programmable data plane advantages, including stateful processing, quick and real-time detection and reaction. In the next chapter, we present an abstraction to model a stateful attack behavior tracking to detect and react against an attack entirely in the data plane within the network. Enabling stateful abstraction inside switches allows offloading a large set of monitoring and security functions onto the data plane. As a result, this would reduce the need to rely on middlebox (*i.e.,* less flexible) and distant controllers. Indeed, the possibility to program and offload network functions regardless of the underlying target, hardware, or software would be the key to effective attack detection.

In summary, our proposition distinguishes from the existing works in different aspects. On the first side, our abstraction targets a mapping to the P4 language compared to FlowBlaze, which focuses on specific hardware target constraints. In contrast to an OpenFlow-based abstraction limited to a FSM abstraction, an EFSM-based abstraction avoids state explosion. Furthermore, its mapping to P4-based primitives aims to be generic to be deployed on many hardware targets. In this thesis, we focus on security applications; in contrast to works specifically aimed at a specific security use case such as DDoS and heavy hitter detection, we propose a generic abstraction for security applications that can be implemented to track different attacks behavior.

Chapter 3

Stateful approach for detecting and mitigating attacks in programmable data planes

Sommaire

3.1 Introduction **34**

3.2 Protocol behavior monitoring in the data plane **35**

3.2.1 Overview 35

3.2.2 Extended Finite State Machine abstraction 37

3.2.3 Stateful behavior example : a DDoS detection EFSM 38

3.2.4 From an EFSM to P4 programs 39

3.3 Application to ECN **44**

3.3.1 Background and attack description 44

3.3.2 EFSM model 45

3.3.3 P4 implementation details 47

3.3.4 Evaluation 50

3.4 Application to Optimistic ACK attack **55**

3.4.1 Attack description 55

3.4.2 Attack EFSM model 57

3.4.3 Evaluation 58

3.5 Design challenges **61**

3.5.1 TCP connection tracking 61

3.5.2 Scalability and memory overhead 62

3.5.3 Hash collisions 63

3.6 Summary **63**

3.1 Introduction

The P4 programming language has made it possible to support line-rate programmable operations in the data plane. This data plane programmability has enabled offloading security functions to the data path, allowing users to replace middlebox-based solutions and avoid their high costs. Using the P4 language, customized packet processing can be deployed to detect and mitigate attacks.

In parallel most protocols were not initially designed with security considerations. As a result, many attacks have exploited protocol vulnerabilities that are not yet patched. One of the vulnerable protocols is the Transport Control Protocol (TCP), that is used in a lot of primary Internet services. An attacker can manipulate the TCP congestion control mechanism to take more bandwidth than legitimate flows and cause an unfair bandwidth share. One example is when an ECN-enabled receiver ignores the congestion notifications to force the sender to keep the same transmission rate while the network is congested. Another example is an attacker that gives the illusion that the network is not congested by sending acknowledgements before segments are received, forcing the server to send data more faster; this misbehavior is called the Optimistic ACK attack. TCP is an end-to-end protocol; the end-to-end principle states that rather than placing intelligence at the network's core, it should be located at end-hosts. Accordingly, existing mitigation solutions [46, 47] propose modifying the protocol specification and end-host implementation to defend against protocol abuse. These end-host-based solutions are complicated from a practical and deployment perspective as they require modifications on the scale of the Internet. To limit the exploitation of protocol vulnerabilities, the solution must mitigate this kind of attack without changing the protocol implementations for fast deployment. On the other side, the solution should detect and mitigate an attack at line-rate and as fast as possible within the network before reaching the end-host. Therefore, a possible approach is to offload the mitigation function onto the traffic's data paths, thus enabling detection and reaction against attacks while the attack traffic is traversing the network and without requiring any protocol implementation change on the end-hosts.

To effectively mitigate attacks, an in-network detection is essential. The programmable data plane enabled by P4 offers advantages in supporting in-network functions such as line-rate packet processing. P4 enables stateful processing by introducing a structure to maintain state. However, it lacks the definition of a stateful abstraction to define a behavioral model (*i.e.,* it does not explicitly define how customized and stateful behavior processing can be supported). Therefore, to exploit the advantages offered by the programmable data plane to effectively detect attacks within the network, the main question to be addressed is how to describe an attack behavior within a switch. The proposed data plane architecture and language require rethinking on a high-level abstraction that could model attack behaviors and ensure three main requirements: complex, expressive, and stateful behavior tracking. We need to address a set of challenges to deploy a security function on the data plane. First, we have to use a powerful model that captures complex behavioral attacks while at the same time being simple enough to be mapped to a programmable switch supportive primitives that remain limited. Second, the abstraction should be general and expressive enough to support a variety of protocols and attack behavior tracking (*i.e.,* with respect to all packet information). Finally, the abstraction should rely on the stateful structure of P4 to support stateful processing.

Monitoring a network in a stateful manner with a simple and expressive abstraction and at the same time allowing tracking complex behaviors while being general enough to support different applications regardless of the underlying hardware and technologies is fundamental for an effective attack detection. We propose an abstraction based on Extended Finite State Machine (EFSM) that is simple enough to be mapped to a programmable switch primitives and at the same time can model a stateful processing and capture complex behaviors. A mapping of this abstraction to P4 based supportive primitives would enable supporting a stateful flow processing within the network. Also, an approach based on an EFSM and the P4 language is general enough and makes its application possible to many security use cases. This chapter introduces an EFSM-based approach for modeling the behavior of a protocol. We present a method to map the EFSM abstraction to P4-based supported primitives. We present two applications to detect and mitigate Layer 3 and Layer 4 attacks, namely the ECN attack and the Optimistic ACK attack.

3.2 Protocol behavior monitoring in the data plane

In this section, we overview the key concepts of the proposed approach. We address the problem of offloading an attack detection function on a programmable switch. We propose a stateful abstraction that decides how packets should be processed based on network events. The approach allows programming in the switch a security function that can be modeled as an EFSM. Figure 3.3 presents an overview of the proposed solution, from the modeling of a protocol with an EFSM to the deployment on a programmable switch. These steps will be discussed in more detail in Sections 3.2.2 and 3.2.4.

3.2.1 Overview

FSM are well suited to model state-based behavior and to verify the presence or absence of unexpected or expected behavior. However, as highlighted in Chapter 2 Section 2.4.1 one of the shortcomings of this model lies in the fact that the number of possible states increases with the complexity of the system. EFSM has been introduced as an extension of the simple FSM model to address this weakness. EFSM allows the use of variables that maintain persistent values; it allows abstracting several state variables into one state variable. Therefore, in our work the motivation behind the use of EFSM over regular FSM is the ability of EFSM to avoid state explosion.

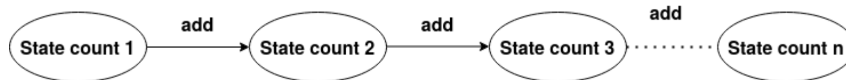


Figure 3.1: FSM counter example

For example, maintaining a counter in an FSM would require one state per counter value (Figure 3.1), whereas the same can be represented in an EFSM with one variable corresponding to the counter (Figure 3.2). Another example is when tracking sequence numbers, a regular FSM would require one state for each of the 2^{32} possible values compared to using just one context variable in an EFSM. Therefore, an EFSM has a smaller number of states than an equivalent abstraction based on FSM. This is because an EFSM uses context variables resulting in a state machine with less number of states. In addition, an FSM always performs a transition on a

boolean input (*i.e.,* an event) from one state to another. An EFSM allows the use of conditions on variables (e.g., $count > 3$ in Figure 3.2); and thus performs a transition when a given set of conditions have been satisfied and not only based on incoming events which simplifies the modeling of more complex behaviors.

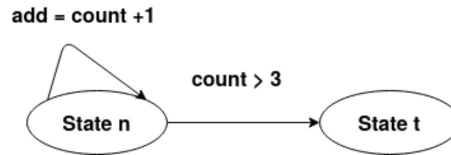


Figure 3.2: EFSM counter example

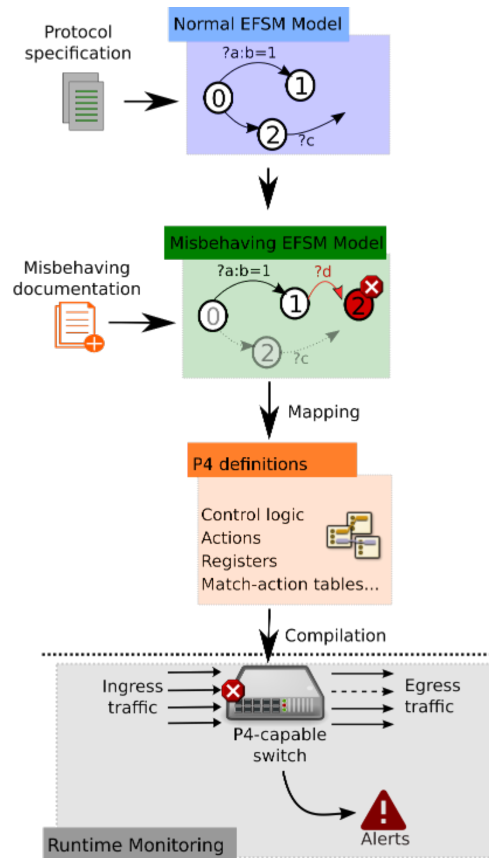


Figure 3.3: Approach overview

An EFSM abstraction is a good starting point for providing an effective abstraction to describe a stateful processing. The stateful propriety of EFSM is represented by the current state and variables that can be maintained in the state machine, read and written by all states. Each flow needs to be associated with an EFSM and the associated information with each EFSM is updated as packets are processed.

In our approach, an EFSM represents a protocol behavior; the first step in the approach is to derive an EFSM from a protocol specification. A protocol can have multiple interacting components, rules and steps. Therefore, multiple states can be produced in the protocol EFSM model, for simplicity, a set of states can be merged. Thus, the number of states to monitor is limited to the states needed to track the correct protocol behavior. Assuming protocols have well defined messages and sequence, we extend the EFSM model with possible misbehavior (*i.e.,* one or more possible misbehaviors). Therefore, the EFSM model represents a formal description of the protocol specification (*i.e.,* the EFSM abstraction represents the specification of the security function to detect a possible misbehavior).

The proposed approach gives the possibility to network operators to configure switches to take different actions when a misbehavior is detected, such as, dropping or rerouting the packet (*i.e.,* modify forwarding behavior), generating an alert and applying corrective actions such as modifying packet field(s), sending a copy of the packets to a remote server for further inspection and setting queue priorities. After the protocol state machine definition step, the next step maps the latter to a set of primitives supported by the programmable data plane. We describe the stage of mapping the EFSM model to a P4 program in more detail in Section 3.2.4. Once the P4 program is compiled, the input traffic is tracked on the fly and at line-rate. Therefore, each flow state (*i.e.,* across different packets) and additional information are maintained for the detection of possible misbehaviors (*i.e.,* each flow has an EFSM instance).

3.2.2 Extended Finite State Machine abstraction

Notation	Definition
S	States (stage of the behavioral process)
$I \subset S$	Initial states
E	Events (pattern matching applied to packet data)
$e[p]$	Value of parameter p extracted from event $e \in E$
V	Persistent context variables
A	Actions, $a \in A, a = \{modify, drop, forward, write(v)\}$ with $v \in V$
C	Conditions, $c \in C$ is a function combining numerical or logical condition operator applied to persistent variables ($\in V$) and event parameter values p
T	Transitions, $t \in T, t = s_1, c \rightarrow a, s_2$ ($s_1 \in S$, $s_2 \in S$, $c \in C$, and $a \in A$)

Table 3.1: Key notations used in the EFSM model

There are different ways to formally represent an EFSM [48, 49]. In our approach, we represent an EFSM formerly by a 7-tuple (S, I, E, V, A, C, T) (summarized in Table 3.1), where:

- S is the set of possible states. In our case, S consists of all the states of the protocol that need to be tracked for detecting misbehavior.
- I is the set of initial states.
- E is a finite set of events. An event is triggered by the content of the monitored packets, such as the presence of a TCP flag in a TCP packet. Events can be parametric [50] to store the values observed in events. We represent the value of the parameter p of the event e by $e[p]$. For example, if $TCP.ACK$ is an event that matches a TCP packet with an ACK flag, $TCP.ACK[AckNo]$ represents its corresponding acknowledgement number, (Figure 3.4).

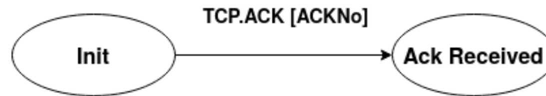


Figure 3.4: EFSM event example

- V is the set of context variables that are persistent and accessible by all states using read and write operations. One of the main advantages of EFSM is that the context variables can be leveraged to avoid creating numerous states.
- A is a finite set of actions to be performed. A includes modifications of context variables and other actions that can be performed within PISA switch capabilities such as modify, drop or forward a packet, or send a message to the control plane. Actions are divided into two categories: packet forwarding actions, and arithmetic and logic operations to update the variables in V or the packet headers.
- C is the set of conditions. Each condition $c \in C$ is defined from event parameters and context variables. This is another major advantage of EFSM compared to regular FSM where transitions are defined only on symbols. With EFSM, more complex conditions can be described. More precisely, each $c \in C$ is a conditional expression composed of operands (events and context variables) to express logical or numerical calculations (*e.g.*, addition, logical AND, OR) and comparisons (*e.g.*, less-than and equal).
- T is the set of transitions, where $t \in T$ is defined as $s_1, c \rightarrow a, s_2$ denotes a transition from state s_1 to s_2 when condition c is satisfied and that results in the action a to be performed.

3.2.3 Stateful behavior example : a DDoS detection EFSM

In this section, we illustrate an attack example to introduce the stateful processing necessity for stateful monitoring and demonstrate the adequacy of the EFSM model in regard to it. Suppose that we have a network with a set of connected end-hosts, switches, and flows. An attacker who wants to load a specific service would perform a DDoS attack by sending a packet burst. A DDoS detection solution needs to be stateful, which means being able to verify the history of received packets. In a stateful processing, the processing of each packet is based not only on the current packet, but also on the set of previous packets. To detect a DDoS attack (e.g., an application-level DDoS), a basic solution is to count the number of received packets. We illustrate an EFSM for the DDoS attack behavior tracking in Figure 3.5, each flow should be associated with an EFSM instance. In this example, we have four states ($Init, Established, State\ X, State\ Y$).

From the initial state *Init*, after receiving a SYN with a source address IP s, a SYN-ACK with a destination IP address s (i.e., the same IP address as in the SYN packet but in the reverse order) and an ACK packet, the connection is established. From the *Established* state, each received packet will increment the value of the context variable *count* that maintains the number of received packets and is associated with a flow identifier ($dstIP, dstPort$), once a first predefined threshold is exceeded TH_1 (i.e., a lower threshold) a transition to the state *StateX* occurs, an action *Send_Alert*() is applied. When a second predefined threshold is exceeded TH_2 (i.e., a higher threshold) a transition to the state *StateY* occurs and a *Drop*() action is applied since the reached threshold is critical compared to TH_1. The example shows that each transition is triggered by a matching on a certain variable value (*e.g.,* count) or on packet fields values (*e.g.,* $srcIP, srcPort$), which represent events. We note that the EFSM is instantiated for each flow.

Another example is a TCP connection tracking to identify packets belonging to the same flow. Implementing such a connection tracking requires monitoring packets information belonging to the same flow. To track TCP connection establishment, each connection needs to be identified in a bi-directional way, which means that, no matter the direction of the flow, it will be identified as the same flow. We need to track a connection establishment in a stateful manner since that we won't be able to identify the $SYN-ACK$ packet ($srcIP = d, srcPort = dp, dstIP = s, dstPort = sp$) without knowing that the packet previously received is a SYN packet ($srcIP \rightarrow s, srcPort \rightarrow sp, dstIP \rightarrow d, dstPort \rightarrow dp$) from the same flow. Similarly, we cannot identify bidirectional flows that belong to the same flow without being able to maintain the source address s and the source port sp in the SYN packet and verify in the $SYN-ACK$ packet that the source address and source port are the same as in the SYN packet but inverted ($srcIP = d, srcPort = dp$).

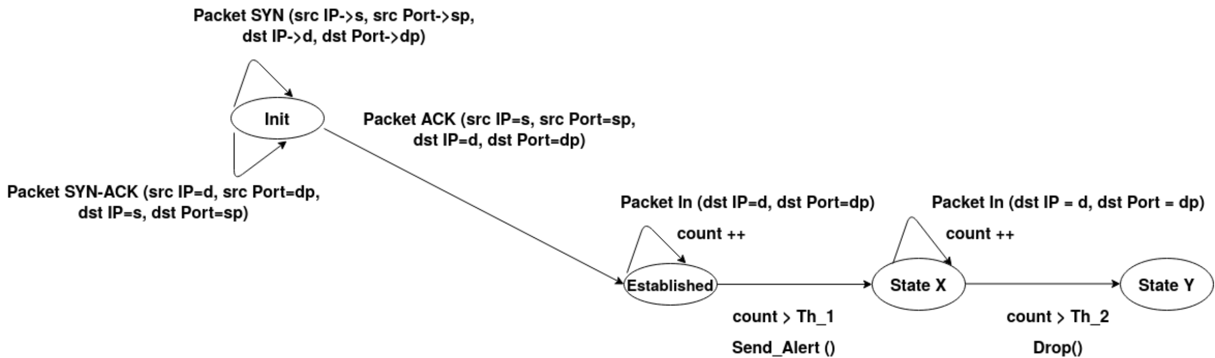


Figure 3.5: EFSM DDoS example

3.2.4 From an EFSM to P4 programs

We leverage the P4 language to track flows (*i.e.,* connections) in the data plane, our approach consists in instantiating an EFSM for each flow in the data plane. In what follows, we describe how the different components of an EFSM (*i.e.,* the EFSM 7-tuple) described in Section 3.2.2 are mapped to a P4 program. We also show how an EFSM abstraction is supported by data plane primitives, which is referred as the mapping process (Figure 3.6).

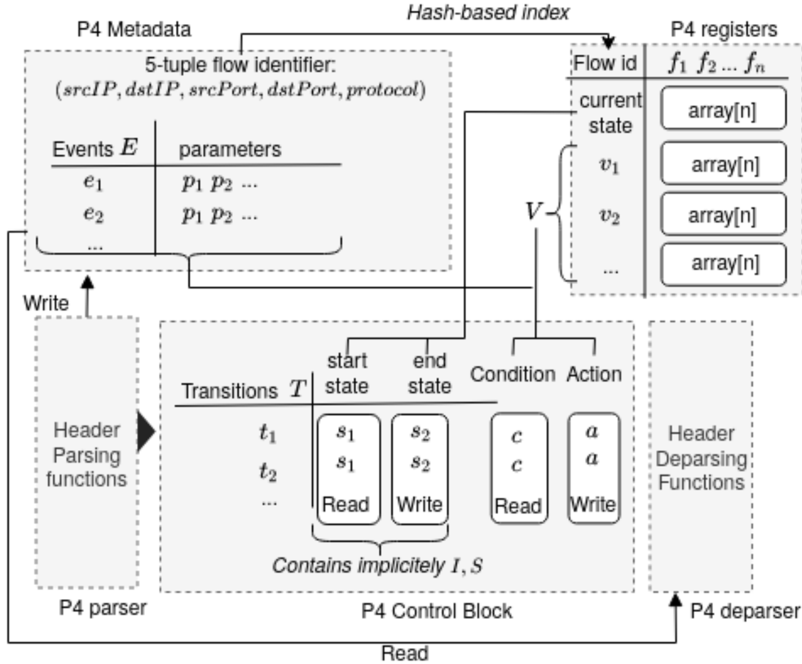


Figure 3.6: EFSM mapping to P4 elements and associated I/O operations

Maintaining persistent information

For each connection or flow, we need to maintain persistent information in the data plane, including the context variables of the EFSM and its current state. There is a one-to-one mapping between the number of executed EFSMs and the number of monitored flows. We formally define a flow as a set of packets $f = \{p_0, p_1, \ldots, p_n\}$ sharing the same set of attributes. Without loss of generality, we define a flow by the 5-tuple: $f' = (srcIp, dstIp, srcPort, dstPort, protocol)$, where $srcIp, dstIp, srcPort, dstPort, protocol$ represent the source IP address, the destination IP address, the source port, the destination port and the transport protocol, respectively. This 5-tuple also forms the *flow key*, which will be later used for computing an index in a hash table. As shown in Figure 3.6, the header fields in the 5-tuple flow key are defined as P4 metadata. As invoked in Chapter 2 metadata variables are associated with the processing of each packet and are used for carrying information across the pipeline stages. Metadata fields along with packet headers are declared in P4 and are set by the parsing function.

Once the flow key is extracted from a packet, we compute a hash function $hash(key)$ to retrieve the persistent information corresponding to that flow. The hash function $hash(key)$, takes as input the flow key and outputs a flow identifier. The current EFSM state of each flow, as well as the context variables from V are stored as entries in register arrays as shown in Figure 3.6. The flow identifier returned by the hash function acts as the indices of these arrays for retrieving the corresponding information (*i.e.*, the current state or the context variables of an EFSM). In total we need $|V| + 1$ register arrays (*i.e.*, V context variable and one for the EFSM current state), each with n entries for keeping track of the persistent information of the EFSMs of n flows. For example, in Figure 3.7 the hash flow identifier is 83, the current state of this flow is directly accessible with its identifier in the register.

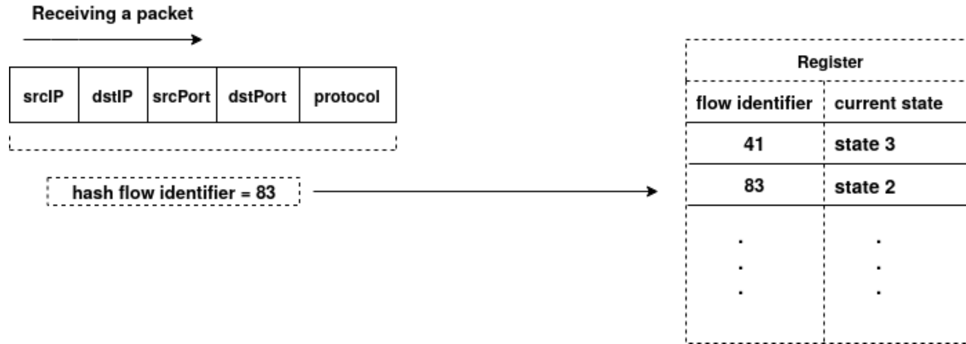


Figure 3.7: Accessing current state

In many cases, we need to monitor behavior for both directions of a flow. For example, as mentioned previously, monitoring the 3-way TCP handshake requires monitoring both directions of a TCP flow. One solution to this problem proposed in the literature is to form the 5-tuple flow key in the order (dstIp, srcIp, dstPort, srcPort, protocol) if (dstIP > srcIP); otherwise form the 5-tuple flow key in the reverse order (srcIp, dstIp, srcPort, dstPort, protocol) [25]. In this way, the source and destination IP address pair in the 5-tuple always remains the same for both directions of a flow. The same applies to the source and destination ports. We note that for applications where the forward and return paths may be different, the solution has to be deployed at the network edge to see both flow directions.

Event extraction

Events in E are retrieved by matching ingress packets against the patterns defining the events (*e.g.*, the presence of the ACK flag in the TCP header). Additionally, ingress packets are checked against the conditions defined on event parameters (*e.g.*, an event is detected if a condition on TCP sequence number is satisfied, $SEQNo = value$ (Figure 3.8)). The result of aforementioned pattern matching and condition evaluation on packet header fields are captured in metadata fields that we define for tracking the events. These metadata fields are populated during packet parsing and can be read when the parsed packet is deparsed at the end of processing. The protocol header fields that must be extracted to perform such pattern matching for event detection are identified based on the events $e \in E$ and *event parameters* $e[p]$ (*e.g.,* $ACKNo$ in Figure 3.4).

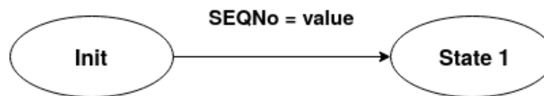


Figure 3.8: EFSM event example

Mapping of the state transition system

When a switch receives a new packet, the packet is processed according to Algorithm 1. While processing the packet, Algorithm 1 may trigger a state transition within the EFSM corresponding to the flow to which the packet belongs. In Algorithm 1, we assume that the current state and the context variable of the EFSMs are maintained in register arrays (P4 registers in Figure 3.6). Except for the events derived from ingress packets and persistent variables defined a priori,

Algorithm 1 Packet Processing

Definitions:

- n: number of monitored flows
- $Current[n]$: register array of the current states
- $\forall v \in V, Val[n]$: register array of each context variable

Input: packet *pkt*

```
1: p ← parse(pkt)
2: id ← hash(p.srcIp, p.dstIp, p.srcPort, p.dstPort,protocol)
3: current ← Current[id]
4: find t ∈ T, t = s1, c → a, s2 such that s1 = current and c.check(p, V) = TRUE
5: if t ≠ None then
6:     Current[id] ← s2
7:     a.apply(V,p)
8: end if
9: deparse(p)
```

actions, transitions and conditions are implicitly defined within P4 control block (P4 Control Block in Figure 3.6).

The control logic of the P4 program starts at line 1 of Algorithm 1 with the parsing of the newly arrived packet (P4 Parser in Figure 3.6). Then, the hash-based flow identifier (P4 Metadata in Figure 3.6) is computed from the flow key to retrieve the current state of the corresponding EFSM (lines 2-3). The algorithm then searches for a possible transition fulfilling two conditions (line 4). First, the transition origin must be the current state. Second, the transition condition computed using the function $c.check(p, V)$ (*i.e.,* the condition c is based on variables in V and event parameters p) must be evaluated to true. If a suitable transition is found in T, the transition is applied. The current state, the context variables, and the packet metadata are modified according to the actions of the applied transition (P4 Control Block in Figure 3.6).

The conditions and the action logic (C and A) are mapped to the P4 control block, this block includes conditional statement or/and MAT and actions (P4 Control Block in Figure 3.6). Read operations are used for checking if a transition can occur and the write operations apply modifications when applicable. It is worth mentioning that all states, S, including the initial states, I, are embedded implicitly in the transitions defined in the P4 processing logic, eliminating the need to define them. Packet modifications include packet header field changes as well as changes to P4 standard metadata. The EFSM model mapped actions (P4 Control Block Action in Figure 3.6) are performed sequentially with packet processing actions such as packet forwarding.

Packets fields modification and checksum recalculation actions

Among the actions of the EFSM, there can be an action consisting of modifying a packet field. The action can be written as $mod_pkt(packet, x) = (packet', x')$ that takes as input a packet *packet* and outputs a new packet $packet'$ where *packet* has a field with the value x modified to the value x' in the same field in $packet'$ such as $x \neq x'$. Recomputing a checksum may be necessary depending on the protocol. For example, in the case of TCP protocol monitoring, the TCP checksum will need to be recomputed. Instead of performing an expensive full checksum

recalculation, the checksum can be incrementally updated using the following simple arithmetic function [51]:

$$HC' = HC - \sim m - m'$$

Where :

- HC: old checksum in the packet header,
- HC': new checksum in the packet header,
- m: old value field including the former packet value field,
- m': new value of field including the modified packet value field,
- $\sim x$: the one's complement of x.

3.3 Application to ECN

In this section, we introduce the application of our EFSM-based approach to detect and mitigate Explicit Congestion Notification (ECN) protocol abuse. We first introduce ECN and a description of the attack. Then we present the modeling of ECN with an EFSM. Afterward, we give some details of the implementation with the P4 language. Finally, we present the evaluation results that show the attack's impact on bandwidth share and throughput and the benefit of our approach in correcting the attacker abuse.

3.3.1 Background and attack description

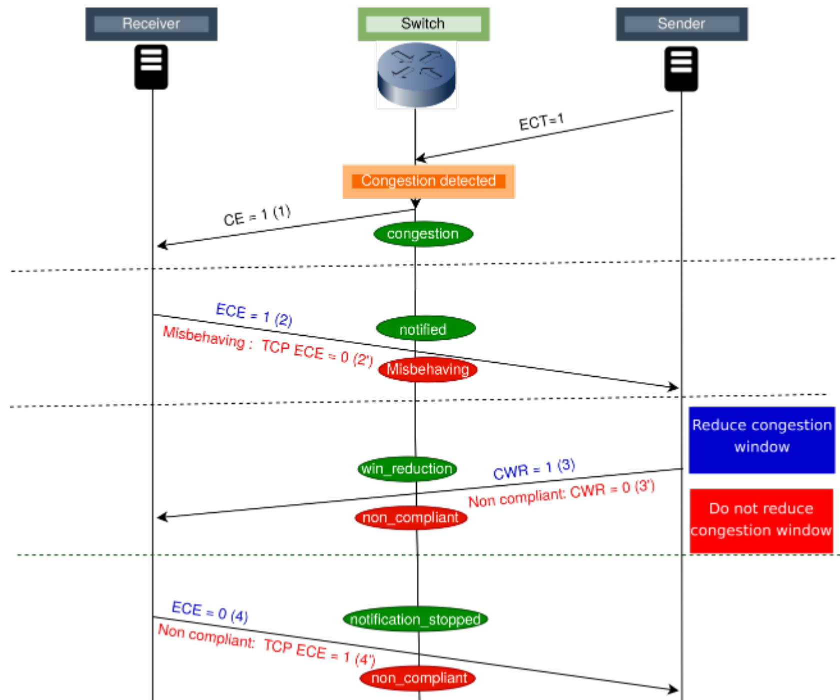


Figure 3.9: ECN (normal behavior in blue, misbehavior in red, ellipses represent EFSM state updates)

TCP end-hosts increase throughput as long as no packets are lost; when a packet is lost the throughput is decreased. TCP end-hosts use end-to-end congestion signals such as packet loss or round-trip-time for adjusting their congestion windows. ECN was proposed as a mechanism for network devices and an extension to the TCP; ECN allows network congestion to be notified before packet loss occurs. The ECN RFC [52] defines the following codepoints and fields for both IP and TCP protocols:

- Congestion Experienced (CE) and ECN-capable Transport (ECT) codepoints for the DSCP field of the IP header.
- ECN-Echo (ECE) and Congestion Window Reduced (CWR) flags in the TCP header.

The design of ECN (*i.e.*, explicit end-hosts signals) introduces the possibility of having misbehaving end-hosts in the network, *i.e.*, end-hosts that do not fully conform to the protocol

specification. We illustrate such misbehaviors (messages in red) along with the expected normal behavior (messages in blue) of ECN enabled TCP end-host in Figure 3.9. During the TCP three-way handshake phase (not shown in the Figure), TCP end-hosts negotiate the use of ECN. Following the TCP three-way handshake, the ECN protocol behaves as follows:

- A congested switch detects an ECN-capable TCP connection (ECT set in IP header) and marks the corresponding packet with CE ((1) in Figure 3.9);
- After receiving a packet with CE, the receiver becomes aware of the congestion and informs the sender by setting the ECE flag in the TCP header ((2) in Figure 3.9);
- Once the sender receives a packet with the ECE flag set, it reacts by reducing its congestion window. Then, the sender sets the CWR flag to inform the receiver ((3) in Figure 1), which in turn stops sending congestion notification by unsetting ECE ((4) in Figure 3.9).

ECN RFC defines a possible misbehavior of an end-host announcing itself as ECN-capable but ignoring congestion notification from the switch [52]. As shown in Figure 3.9, once a switch notifies about congestion, the receiving host can misbehave by not echoing back the congestion information to the sender, *i.e.*, set the ECE flag to 0 ((2') in Figure 3.9). As a result, the sender does not reduce the congestion window ((3') in Figure 3.9). Misbehaving flows can degrade network performance and create a denial service for the benign flows at the expense of their own loss in throughput. This is why the RFC recommends that such flows must be identified and handled.

For the sake of completeness with respect to the original ECN procedure, Figure 3.9 introduces two non-compliant behaviors (*i.e.,* not necessarily malicious behaviors, but behaviors that do not comply with the protocol specification). First, the receiver can keep sending packets with ECE set ((4') in Figure 3.9) even though the sender has already reduced its congestion window, forcing the sender to further reduce its congestion window. Second, the sender can ignore the notification sent back from the receiver ((2) in Figure 3.9) by not reducing the congestion window ((3') in Figure 3.9). Similarly the sender can reduce its congestion window without notifying through setting the CWR bit. In both cases, the switch cannot deduce if the congestion window has been really reduced. That is why these behaviors are qualified as non compliant rather than misbehaving. These non-compliant behaviors will allow us to evaluate our method in a more complex scenario.

The solution proposed in [46] is based on a nonce to detect receiver attempts to misrepresent the congestion signal. The sender generates a random nonce, the nonce value is sent by the sender in the first packet with the ECT field set ($ECT = 1$). When the congestion occurs, the switch set the CE field ($CE = 1$) and clears the nonce ($nonce = 0$) to indicate congestion. The receiver should set the ECE field ($ECE = 1$) and copies the nonce value to signal congestion. Therefore, the sender detects the possible misbehaving if the nonce is different from the initial value (i.e., is cleared ($nonce = 0$)) and the ECE field is unset ($ECE = 0$). The deployment of such solution requires TCP protocol modification. In what follows, we address the ECN misbehaving problem without requiring the protocol modification.

3.3.2 EFSM model

We model the expected and unexpected (misbehaving and non compliant) behaviors of ECN capable end-hosts in a single EFSM. Each time a packet is received, the state of the EFSM illustrated by an ellipse in Figure 3.9 is updated. Normal states are colored in green while all

unexpected states are colored in red. *init* represents the initial state of the EFSM, $I = \{init\}$. The EFSM model is instantiated into the switch and the transitions are triggered by events which are derived from ingress packets as explained in Section 3.2.4, notably ECN related flags in these scenarios. Based on these definitions, we illustrate the EFSM corresponding to Figure 3.9 in Figure 3.10. It is defined as follows:

- $S = \{init, congestion, notified, win_reduction, notification_stopped, misbehaving, noncompliant\}$;
- $I = \{init\}$;
- $E = \{ECT = 1, ECE = 1, ECE = 0, CWR = 1, CWR = 0\}$ with the following parameters: $IP.src$ and $IP.dst$ are respectively the source and destination IP address;
- $V = \{dstCE\}$;
- Annotations of arrows in Figure 3.10 represent A, C and T

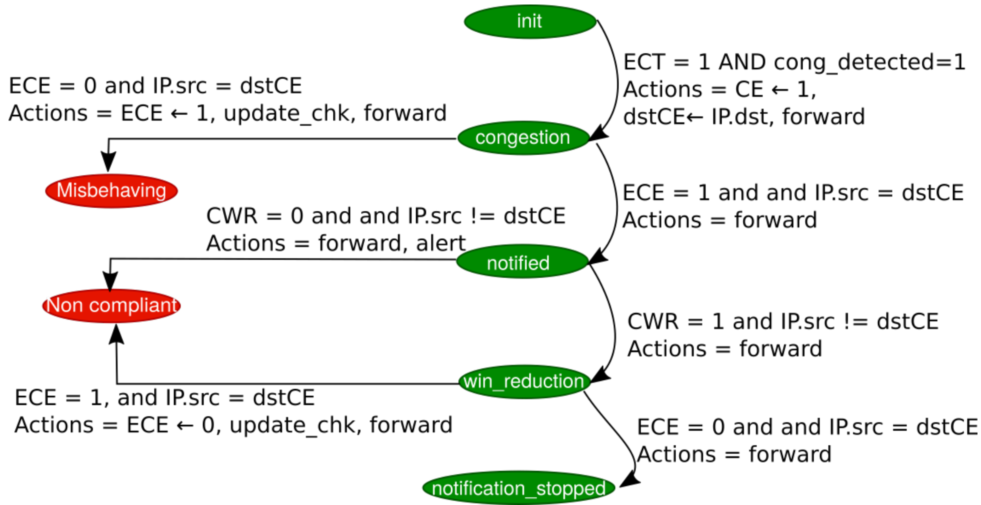


Figure 3.10: EFSM abstraction for ECN

No additional information must be maintained alongside the current state since the states themselves are self-contained in terms of necessary information. However, recognizing the direction of packets is still necessary. Therefore, we use a single context variable $dstCE$ for identifying the host that is supposed to relay the congestion notified by the switch (*i.e.*, the receiver in Figure 3.9). To properly set the $dstCE$ variable, source ($IP.src$) and destination ($IP.dst$) IP addresses are needed, which are anyway extracted to create the 5-tuple flow key. Once a misbehavior or non compliant behavior is detected, we take corrective actions when possible.

For the misbehaving and the second non compliant cases, the receiver must react according to the congestion signal (with either $CE = 1$ or $CWR = 1$). Therefore, the behavior can be corrected by setting or unsetting ECE accordingly. This also requires the TCP checksum to be updated (*action update_chk* as introduced in Section 3.2.4). Correcting the second non compliant case is also important because a receiver with an incorrect ECN implementation would penalize itself by forcing the sender to continue reducing its congestion window. Note that our

approach can also be employed to detect incorrect protocol implementation at end-hosts aside from being used for the security use cases.

For the first non compliant case, the switch cannot effectively verify that the sender has reduced its congestion window. As a result, it is impossible to deduce from the switch if the CWR flag must be really set to one in the first non compliant case. Therefore, we resort to sending an *alert* and $forward$ the packet. The exact definition of the *alert* depends on the switch capabilities and can be of different kinds such as creating entries in the log files of the switch, sending a packet to the controller, or generating a postcard [53, 54]. The exact details of such action is out of the scope of this work. In our prototype implementation, we let the packet pass through and change the EFSM state to non compliant.

It is worth mentioning that the congestion is detected by the switch itself enabling so the transition from *init* to *congestion*. It corresponds to the condition $cong_detected = 1$ in Fig. 3.10. In practice, this variable is derived from the occupancy level of the switch queues (intrinsic metadata). When the occupancy reaches a marking threshold, the packets are marked and *cong_detected* is set to 1.

We note that a packet ordering problem may arise due to network latency, a receiver may send a packet with ECE unset ($ECE = 0$) because the packet was sent before receiving the congestion notification (CE=1). In this case, our approach assumes that the receiver is malicious. However, we consider our approach to be proactive by reacting earlier to congestion.

3.3.3 P4 implementation details

This section describes how we track TCP flows in the data plane and discusses our solution using P4. We need to process the packet header information to track TCP flows in the data plane. Therefore, we need to extract these fields using the P4 parser. We need to maintain a state across packets belonging to the same flow using P4 registers and P4 metadata to maintain the state across packets processing stages. Operations must be performed on packets using P4 actions, and conditions are evaluated on packets headers and context variables.

Registers for maintaining per-flow state

We use registers to maintain per-flow state; registers are updated when packets are processed. Figure 3.11 illustrates the definition of a register; $Register5_flowState$ which is an array of values of 32 bit length with 500 entries. To access each register, the hash flow identifier is used ($hashFlowResult$); in our case, it is the 5 tuple hash result using the $crc32$ hash function, as shown in Figure 3.15. On the reception of each packet, the 5 tuple is hashed to find its index in the register. Registers can be updated using the write action and the hash identifier, as shown in Figure 3.12. However, the risk of hash collisions may occur, techniques to address this issue are presented in Section 3.5.3

```
register<bit<32>>(500) Register5_flowState;
```

Figure 3.11: Register declaration

```
Register5_flowState.write(meta.ingress_metadata.hashFlowResult,meta.ingress_metadata.currentState);
```

Figure 3.12: Register write

Metadata for maintaining per-packet state

Metadata are used to carry information between processing stages of the same packet and are used as temporary variables in the P4 program, metadata are not necessarily extracted from packet headers. In Figure 3.15, the metadata *hashFlowResult* is used as the output of the hash function that produces the hash flow identifier. This metadata is used to read and write registers. In the P4 code, the set of metadata used is defined with its size in bits along with the definition of packet headers.

As invoked in Chapter 2, P4 provides some specific metadata called the 'standard intrinsic metadata' (Figure 3.13) such as *enq_qdepth* that gives the number of packets in the switch queue which is necessary in our case to detect a possible congestion, if a specific threshold is exceeded, as a result, the IP *ECN* field (*i.e., the IP CE field*) is set to indicate congestion (Figure 3.14). Random Early Detection (RED) [55] is a congestion signaling and an active queue management mechanism that drops packets with a certain probability in anticipation of network congestion. In our case, we employ RED for probabilistic packets marking with congestion notification instead of dropping the packets. We have implemented the RED algorithm in P4 based on the implementation from [56]. We use the instantaneous outgoing queue occupancy (*standard_metadata.enq_qdepth*) made available by bmv2 as intrinsic metadata to react faster to traffic bursts [57]. Note that bmv2 provides queue occupancy in terms of the number of packets. Therefore, we slightly modified the original RED algorithm, which uses the number of bytes to determine the queue length [58]. However, we keep the packet sizes the same in our experiments; hence, the number of packets is directly proportional to the number of bytes. Since P4 targets do not support floating-point operations, we pre-calculate the probabilities, scale the values to be between 0 and 255, and store the probabilities in a match-action table (Figure 3.17), the table and action definition are presented in Figure 3.16 including the table *calc_red_drop_probability* with a match on the *standard_metadata.enq_qdepth* and the action *set_drop_probability* that sets the *drop_prob* metadata). For each incoming packet, the length of the queue is matched with the entries in this table to obtain the associated probability. The obtained probability *drop_prob* set in the action *set_drop_probability* in Figure 3.16 is then compared against a random number (*meta.ingress_metada.rand_val_test* in Figure 3.18) to decide whether the packet should be marked (*hdr.ipv4.ecn* flag set in Figure 3.18).

```
struct queueing_metadata_t {

        bit <48> enq_timestamp;
        bit <16> enq_qdepth;
        bit <32> deq_timedelta;
        bit <16> deq_qdepth;
}
```

Figure 3.13: Intrinsic metadata

```
if (standard_metadata.enq_qdepth >= 3){

                hdr.ipv4.ecn = 3 ;

        }
```

Figure 3.14: metadata usage

```
hash(meta.ingress_metadata.hashFlowResult,
            HashAlgorithm.crc32,
            32w0, {hdr.ipv4.srcAddr,hdr.tcp.srcPort, hdr.ipv4.dstAddr, hdr.tcp.dstPort,
hdr.ipv4.protocol}, 32w500);
```

Figure 3.15: hash result metadata

```
@name("set_drop_probability") action set_drop_probability ( bit<9> drop_probability ) {
        meta.ingress_metadata.drop_prob =  drop_probability;
    }

table calc_red_drop_probability {
        key = {
            standard_metadata.enq_qdepth : exact;
        }
        actions = {
            set_drop_probability;
        }
        const default_action = set_drop_probability(0);
        size = NUM_RED_DROP_VALUES;

}
```

Figure 3.16: RED MAT definition

```
table_add calc_red_drop_probability set_drop_probability 0 => 0
table_add calc_red_drop_probability set_drop_probability 1 => 0
table_add calc_red_drop_probability set_drop_probability 2 => 0
table_add calc_red_drop_probability set_drop_probability 3 => 0
table_add calc_red_drop_probability set_drop_probability 4 => 85
table_add calc_red_drop_probability set_drop_probability 5 => 170
table_add calc_red_drop_probability set_drop_probability 6 => 256
table_add calc_red_drop_probability set_drop_probability 7 => 256
```

Figure 3.17: RED MAT entries

```
    if (hdr.ipv4.isValid()) {

            calc_red_drop_probability.apply();

            random<bit<8>>(meta.ingress_metadata.rand_val_test, 0, 255);


    if ( (bit<9>) meta.ingress_metadata.rand_val_test < meta.ingress_metadata.drop_prob) {

            hdr.ipv4.ecn = 3 ;


      }
  }
```

Figure 3.18: ECN flag set

Condition and action for packet processing

Figure 3.19 shows a piece of P4 code to track the behavior of the ECN flows. In the P4 control flow we can evaluate conditions and apply actions, conditions are applied on packet

header (*e.g., hdr.tcp.ece*) fields or metadata (*e.g.,meta.ingress_metadata_currentState*). In Figure 3.19 we verify the *ece* TCP field and the current state values using the conditional if statement which corresponds to the transition from the state *congestion* to the state *notified* in the ECN EFSM in figure 3.10. In addition, as invoked in Section 3.2.4 we verify the flow direction by comparing the source and destination addresses, the current state is updated and wrote back to *Register5_flowState*. We note that the packet processing in the P4 control block can also be implemented using MAT where the match keys represent the *ece* flag and the current state.

```
if (hdr.tcp.isValid() && hdr.tcp.ece == 1 && meta.ingress_metadata.currentState == 0 ) {


     if( hdr.ipv4.srcAddr < hdr.ipv4.dstAddr){

     readFlowCurrentState1_dir1.apply();

     }else {

     readFlowCurrentState1_dir2.apply();
     }

       meta.ingress_metadata.currentState = meta.ingress_metadata.currentState +1;
       Register5_flowState.write(meta.ingress_metadata.hashFlowResult,meta.ingress_metadata.currentState);


}
```

Figure 3.19: Control Flow example

3.3.4 Evaluation

This section presents our results that evaluate the proposed approach for detecting ECN abuse. We present results demonstrating the attack impact on bandwidth share and how our solution restores bandwidth loss. The experiments were performed using the bmv2 software switch [10] with mininet [59], p4app docker container [60] and P4-16 [61].

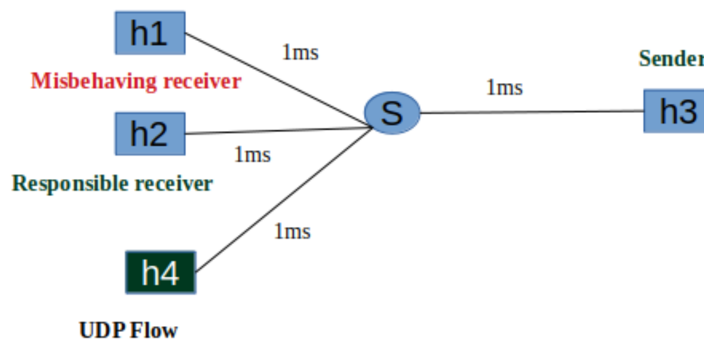


Figure 3.20: Experimental topology used in mininet

3.3.4.1 Bandwidth share and throughput evaluation

To assess the impact of the use or misuse of ECN, we evaluate if the bandwidth is correctly shared during congestion assuming the topology presented in Figure 3.20. It is composed of one

switch and four hosts (h1, h2, h3, h4). By default, all TCP hosts are ECN-capable and the link latency is set to 1ms. To create congestion, we limit the link capacity to 2000 packets/sec and a UDP flow between h4 and h3 is generated with 8 Mbits/sec to flood the network. The size of UDP packets is set to 400B to increase the number of packets in the network and in the switch queue.

We generate TCP flows using iperf between h1 and h3 and between h2 and h3. h1 acts as a misbehaving host by not echoing the congestion notification ((2') in Figure 3.9), (*i.e.* not setting the TCP ECE flag). h1-h3 and h2-h3 flows are thus qualified as misbehaving and normal, respectively. The switch queue capacity is set according to the Bandwidth Delay Product rule (BDP) [62]: $queue_capacity = RTT * Network_bottleneck_capacity = 8$ packets, in our setup since the lowest RTT between two hosts is 4ms and the bottleneck capacity is limited by the queue rate of 2000 packets/sec. We evaluate the misbehaving ECN use-case under different settings for the ECN marking threshold, *i.e.*, the number of packets in the switch queue for triggering the sending of congestion notification. First, we consider recommendations from the data center networking literature. Second, we consider the ECN with the active queue management mechanism RED [55], which represents a more generic scenario.

In the case of data center networks, the research literature proposes several ECN marking schemes, the majority of them being deterministic, *i.e.*, switches mark packets when switch queue length is higher than a pre-determined threshold. There are different recommendations for the ECN threshold in DCN. DCTCP [63] recommends an ECN marking threshold greater than $queue_capacity/7$. Another work on ECN presented in [64] recommends a marking threshold of: $queue_capacity/\sqrt{n}$, where n is the number of flows. In the first case, the threshold should be greater than 1.14 in our setting. In the second case, the marking threshold should be 4.61. However, we decided to vary the marking threshold from 2 to 7 to consider different values, including those calculated from the recommendations (2 and 5 obtained by rounding the numbers to the next integer).

In Figure 3.21, we present the throughput of the normal flow (h2-h3) and the misbehaving flow (h1-h3) side-by-side from the following scenarios:

- noMisbehaving-forward: the baseline scenario where all hosts follow ECN specification and the switch forwards packets without modification (neither detection nor reaction activated). Even the flow h1-h3 behaves as expected.
- Misbehaving-noReaction: h1 misbehaves but the switch continues to forward packets (neither detection nor reaction activated).
- Misbehaving-ECEreaction: similar to the previous case but the switch implements ECN misbehavior detection and applies corrective measures. The switch partially implements the EFSM of Figure 3.10 with $S = \{init, congestion, misbehaving\}$.

The bars in Figure 3.21 represent the average throughput of 10 emulation runs of 120 seconds each. In the case of no misbehaving flow (noMisbehaving-forward), a fair bandwidth share is observed regardless of the marking threshold (each flow is getting ≈10 Mbits/sec). However, the misbehaving flow takes the largest share of the available bandwidth for low marking thresholds (2, 3, 4, 5) when there is no detection. In that case, the misbehaving host ignores the congestion notification and increases its sending rate much higher than the benign flow. Indeed, the benign flow reduces its congestion window in response to congestion notification, leaving more queue space to the misbehaving flow. When increasing the marking threshold, congestion notifications

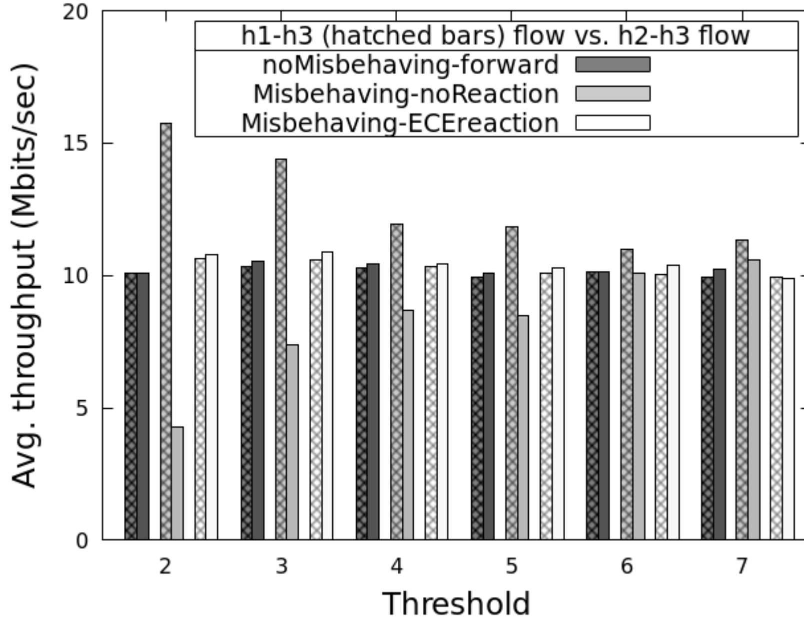


Figure 3.21: Misbehaving and normal flow throughput depending on the marking threshold

are sent later, (*i.e.*, when the queue is more occupied). Therefore, for higher ECN marking thresholds (6, 7) the impact of the misbehaving flow on the normal flow is reduced compared to low marking thresholds. Even with the most aggressive recommendation [64] with a threshold of 5, the misbehaving flow gets 11.8 Mbits/sec while there is only 8.4 Mbits/sec left for the normal flow (59% vs. 42% share). In the presence of the reaction (Misbehaving-ECEreaction), the bandwidth is properly distributed and an equitable bandwidth share is observed between the flows in all cases. This demonstrates the effectiveness of our solution in detecting and reacting to the attack.

In the case of using RED, when the average queue length in a switch is between two thresholds *minth* and *maxth*, an incoming packet is marked with a certain probability. The marking probability is a function of queue length and changes linearly between 0 and 1 [55]. However, if the average queue length becomes higher than *maxth*, then all the incoming packets are marked [55]. For setting the *maxth* and *minth* parameters of RED, we set *maxth* to at least twice *minth* following the recommendation in [55]. Since the queue capacity is 8 in our experiments, we used the following combinations of (*minth–maxth*) tuples: (2-4), (2-5), (2-6), (2-7), (3-6), and (3-7) for covering all the possible combinations in our experiments. We report the result of our experiments using RED in Figure 3.22. We again observe that applying our technique enables ensuring a fair bandwidth share when an attack takes place, reinforcing the effectiveness of our solution. Similarly to the deterministic threshold case, we observe a higher impact of the attack when the packets are marked at lower queue occupancy. In the case of RED, this scenario occurs for lower *minth* values, when packets are likely to be marked even when the queue is not substantially occupied. For example, for the combination (2-7) the misbehaving flow reaches $\approx$ 9.31 Mbits/sec, forcing the benign flow to reach only $\approx$ 6.48 Mbits/sec (58.96% vs. 41.03% share). Finally, we notice that each flow achieves $\approx$ 8 Mbits/sec throughput, which is lower than that reported in Figure 3.21. This is because applying RED also introduces additional per-packet processing overhead within bmv2, hence, reducing the throughput.

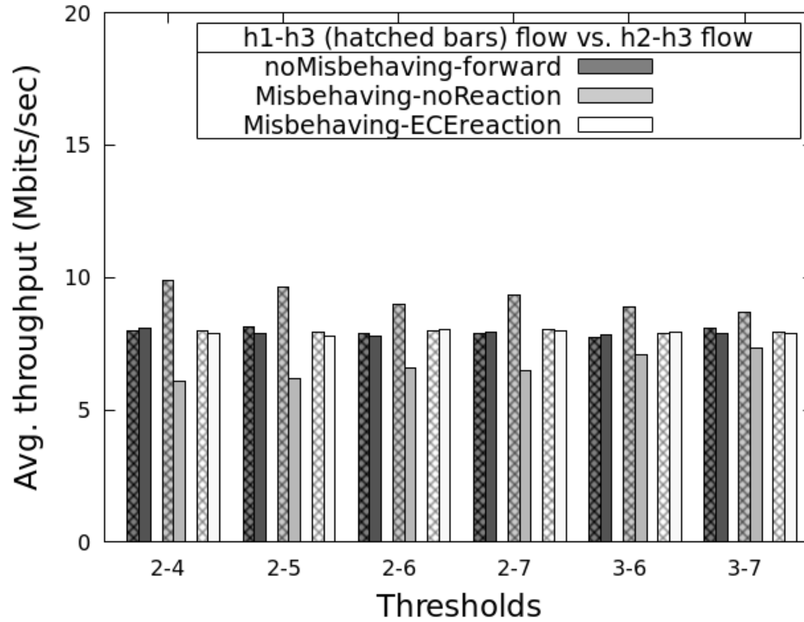


Figure 3.22: Misbehaving and normal flow throughput depending on the marking threshold using RED

3.3.4.2 Switch processing time evaluation

In this experiment, our goal is to evaluate the runtime overhead per packet. Mininet and bmv2 switches are not suitable for evaluating performance in a realistic setting (*i.e.* have limited performance), thus a baseline scenario for comparison is used in different experiments. We consider different configurations:

- L3: parsing is limited up to the IP header.
- L4: L3 + TCP header parsing.
- ECE-v: the state machine is partially implemented to model the normal behavior; states S are restricted to $\{init, congestion, notified\}$ in Figure 3.10.
- ECE-v-r: the state machine is partially implemented to monitor and react against the misbehavior, ((2') in figure 3.9); S is restricted to $\{init, congestion, notified, misbehaving\}$.
- full-v: the full state machine is implemented for misbehaving and non compliant flow verification except for the corrective actions and checksum recalculation.
- full-v-r: similar to full-v but including the reaction against the ECN misbehavior ((2') in Figure 3.9) by setting ECE to 1 and thus including the checksum recalculation.

In this evaluation, the full-v and full-v-r scenarios are considered to estimate the overhead induced for monitoring the whole ECN state machine in the data plane including misbehaving and non-compliant flows verification. We deploy a simple topology with a single switch and two hosts (server and client). We generate 1000, 5000 and 10000 flows between the hosts and report the processing time per packet from one emulation run in Figure 3.23. The median value is 2.7 ms, 2.7 ms, 3.5 ms, 3.7 ms, 4.6 ms and 5 ms for L3, L4, ECE-v, ECE-v-r, full-v and full-v-r

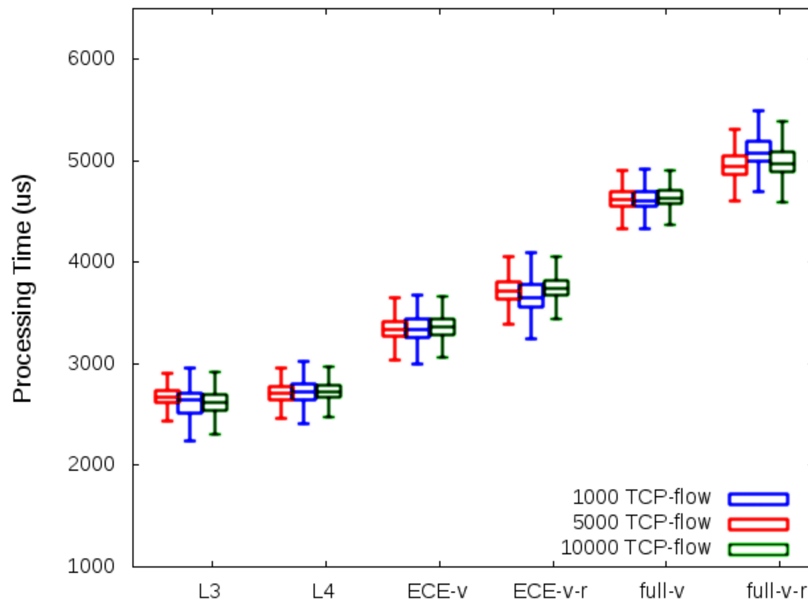


Figure 3.23: Switch processing time

scenarios, respectively. Parsing TCP does not add a significant cost compared to IP parsing. As expected, the more complex the EFSM, the higher the overhead. It is worth mentioning that corrective actions incur less overhead than monitoring using a more complex EFSM when comparing the increase between ECE-v and ECE-v-r and between full-v and full-v-r. Therefore, we recommend a partial implementation of the EFSM to react only against the misbehavior since the non compliant flows verification induce a substantial overhead. Therefore, before deploying a protocol compliance verification solution, it is necessary to verify its usefulness and its adequacy to network performance conditions. Finally, as shown in Figure 3.23, our approach is scalable with respect to the number of flows to monitor in parallel.

These results have been obtained using the deterministic marking threshold. This is because an additional active queue management mechanism for ECN marking is independent of our contribution, and including that for processing time evaluation will not give a real picture. Note that we also evaluated the processing time with RED included and found it to be ≈4.1 ms for the ECE-v-r scenario. This additional processing time explains the decrease in throughput between Figure 3.21 and Figure 3.22.

3.4 Application to Optimistic ACK attack

In this section, we describe how our approach can detect and mitigate a second use case of TCP protocol abuse, the Optimistic ACK attack. We present the attack followed by the EFSM model. Finally, we present our evaluation results, which demonstrate the impact of the attack on network throughput and show how our approach can correct the throughput sharing among flows.

3.4.1 Attack description

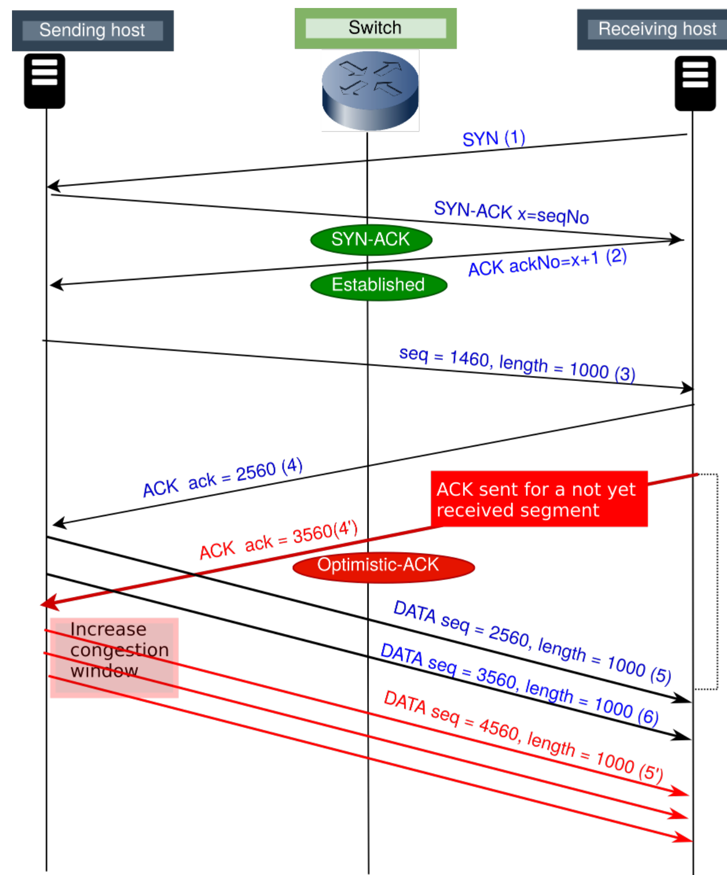


Figure 3.24: Optimistic ACK attack

The Optimistic ACK attack was introduced in [47] as a way to mislead end-hosts to increase their TCP congestion window. This is accomplished by malicious receivers who acknowledge not yet received TCP segments to the senders. In the context of TCP, the congestion window limits the number of TCP segments that can be sent by an end-host without receiving an acknowledgement (ACK). TCP congestion control mechanisms (*e.g.*, BIC, CUBIC [65]) typically start with a relatively small congestion window and keep increasing the window based on received ACKs during the slow start and congestion avoidance phases. Since receiving ACKs of in-flight segments is considered a sign of good network conditions, the malicious receivers in the Optimistic ACK attack will mislead the sender and trigger an increase in the congestion window. This in turn will cause the sender to increase its sending rate, which will have a detrimental impact on legitimate TCP connections sharing the same network.

We illustrate the Optimistic ACK attack through a sequence diagram in Figure 3.24. Here, a receiving host (*i.e.*, the attacker) is starting a normal TCP connection before abusing it. During normal operations at the beginning (in blue), the receiving host (the attacker) initializes the connection. After the 3-way hand-shake (step (2)), data transfer begins as usual. In this example, the sender sends a segment of 1000 bytes (step (3)) and the receiving host sends an ACK upon receiving the segment (step (4)). Then, the sender transmits two more segments in a row (step(5)-(6)) assuming a slow start.

The Optimistic ACK attack phase is highlighted in red in Figure 3.24. During the attack, the receiving host (the attacker) anticipates the reception of segments by sending ACK for segments that are not yet received (step (4')). The receiver (the attacker) sends these Optimistic ACKs in a way that they appear to be corresponding to "in-flight" segments. As a consequence, the sender increases its congestion window and sends the subsequent segments earlier than it should be (step (5')). In this example, the sender sends three segments in a row after receiving the Optimistic ACK from the receiver. As a consequence of this attack, the sender believes that network conditions are better than the reality, resulting in a transmission rate faster than it should be. This way, the malicious TCP connection can exhaust bandwidth on some or all of the intermediary links on the path between the sender and the receiver, causing a possible denial of service attack.

Optimistic ACK can be considered as an amplification attack because the attacker only needs to send many small ACKs while delegating the flooding of larger segments to a legitimate host, the victim. Note that the Optimistic ACKs may be received by the sender before the corresponding TCP segments are sent. However, in most of implementations, these anticipated ACKs are ignored [47]. There is indeed no verification of the matching between received ACKs and the sent segments as it is not necessary for normal operations and adds overhead.

Solutions have been proposed to react against this attack. The authors in [47] present a nonce-based solution that requires changing the TCP implementation at the end-hosts. The TCP sender needs to fill each sent segment with a unique random value, the TCP receiver and sender maintain the cumulative nonce sum of all acknowledged segments, each time a receiver sends an ACK it echoes the nonce value sent by the sender. In that case, the sender can verify that the ACKs sent by the receivers have been really sent. Another solution was proposed in [66] to mitigate the Optimistic ACK attack; the solution randomly drops segments at the sender side. Therefore, when the sender gets an Optimistic ACK for one of the intentionally dropped segments, it can identify the receiver as a misbehaving one. In addition to requiring a change in the TCP implementation, this solution also penalizes the legitimate receivers. Jero *et al.* introduced an offline mechanism for misbehaving TCP end-host detection, it collects logs from end-hosts and compares the resulting TCP performance with the expected performance obtained from a testing environment [67]. This solution is effective for attack forensics; however, it is ineffective for online detection. Since some of the misbehaviors are attributed to incorrect implementations, Kothari *et al.* proposed to use symbolic execution techniques for verifying the latter [68]. However, this approach is limited to a specific operating system, implementation, and language.

Referring to all these works, we can see that to mitigate this attack protocol modifications are required, or the detection is not performed in real time. In contrast to these solutions, we propose to leverage data plane programmability for real-time detection and mitigation of the attack without any end-host and protocol specification or implementation modification.

3.4.2 Attack EFSM model

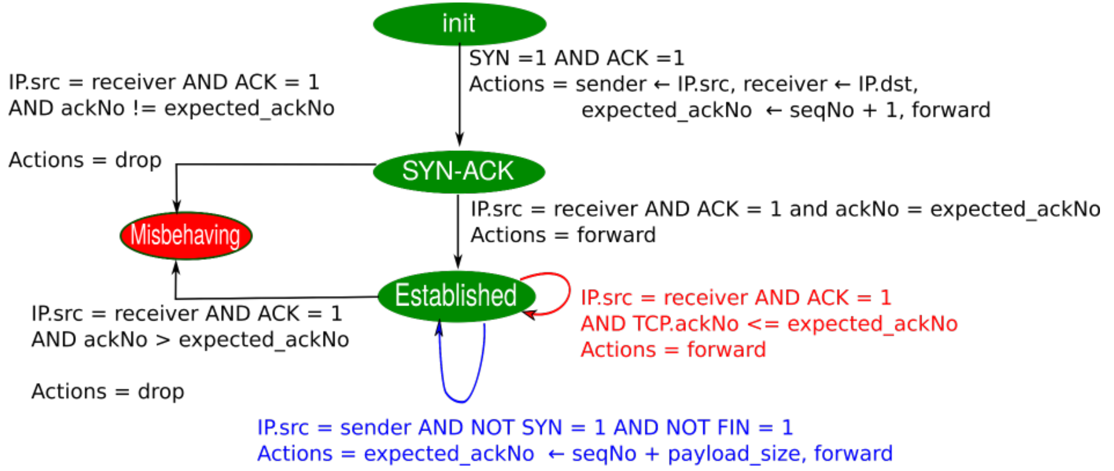


Figure 3.25: EFSM for Optimistic ACK detection and reaction

We model the normal TCP behavior and the possible Optimistic ACK attack in a single EFSM. We use the states represented in Figure 3.24 as the states in the EFSM in Figure 3.25 with an additional initial state called *init*. We define the events from the relevant TCP flags shown in Figure 3.24. Once the TCP connection is established, we use the conditions $SYN = 1$ and $FIN = 1$ to identify and ignore a possibly re-transmitted SYN and FIN (end of connection) segments for the tracked connection. The detection of the attack requires observing the TCP segments in both *sender-to-receiver* and *receiver-to-sender* directions. Therefore, we monitor bi-directional flows and determine the direction of a flow by checking if the source IP address of a packet, $IP.src$, is from the receiver (attacker) or the sender. The $SYN - ACK$ state allows us to monitor the beginning of the connection/flow, initiate its monitoring and identify the receiver and sender in the bi-directional flow. The identified receiver and sender IP addresses are saved into context variables, *receiver* and *sender*, since they are required for monitoring the connection.

To detect the attack, we have to observe a deviation between the maximum acceptable acknowledgement number (*i.e.*, $expected_ackNo$) and the acknowledgement number sent by the potential attacker (receiver). We compute $expected_ackNo$ from the last seen sequence number $TCP.SeqNo$ and the payload size $payload_size$. The $payload_size$ in turn is calculated from IP header length ($IP.ihl$), IP total length ($IP.len$) and TCP header length ($TCP.dataOffset$) fields extracted during parsing:

$$payload_size = IP.len - ((IP.ihl + TCP.dataOffset) \times 4)$$

All these per-packet data are stored as packet metadata in P4 while $expected_ackNo$ is tracked using the context variable since this needs to persist across the TCP segments of a connection (Section 3.2.2). For this attack, we drop the packet when the attack is detected.

Therefore, the Optimistic ACK attack EFSM model can be represented as follows:

- $S = \{init, SYN - ACK, Established, Misbehaving\}$;
- $I = \{init\}$;

- $E = \{SYN = 1, ACK = 1, FIN = 1\}$ with the following parameters: $TCP.ackNo$ and $TCP.SeqNo$ are the acknowledgement and sequence number of the current packet and $IP.src$ and $IP.dst$ are respectively the source and destination IP address;
- $V = \{expected_ackNo, sender, receiver\}$;
- Annotations of arrows in Figure 3.25 represent A, C and T

It is worth mentioning that the EFSM in Figure 3.25 does not represent future actions after the attack is detected. This depends on the operator policy and is orthogonal to the EFSM model for detecting misbehavior. One possible action can be to drop all subsequent opportunistic ACKs for the malicious TCP connection. We note that we did not introduce the possibility of re-transmission occurrences in the state machine. Still, it is a simple functionality to add at the time of deployment by comparing the number of sequence numbers with those received before to check the presence of a retransmission. However, this will have no impact on the detection in the current presentation of the state machine, the expected acknowledgment number will be recalculated in the presence of a retransmission. With the addition of the retransmission verification functionality, a recalculation will not be performed.

3.4.3 Evaluation

This section evaluates our solution to detect and react against the Optimistic ACK attack.

3.4.3.1 Attack mitigation

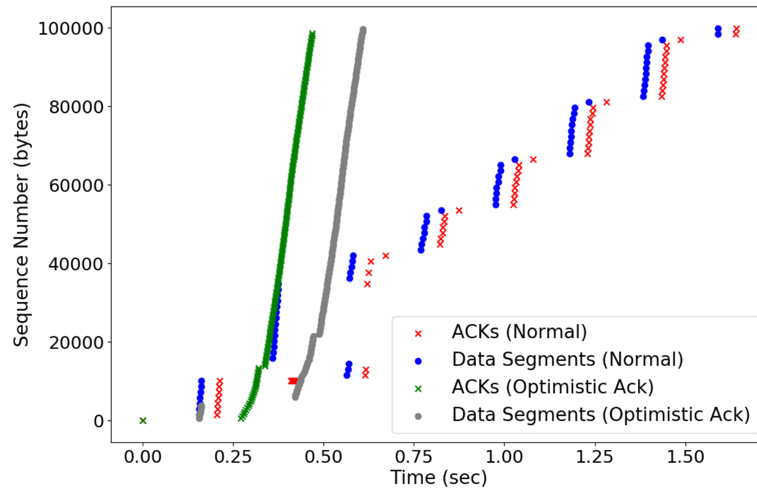
In the first experiment, we evaluate the effectiveness of our approach in mitigating the Optimistic ACK attack from the data plane. To accomplish this, we perform an experiment similar to that done in [47]. We consider a simple topology with two hosts (a sender and a receiver) connected to the same switch. We perform the experiment both with and without detection and mitigation in the switch program. We generate the attack using the tool available in [69]. We note that the traffic is captured on the attacker's side so that we can track the Optimistic ACKs sent before being dropped at the switch.

Figure 3.26 (a) illustrates how the attack is performed by comparing the sequence numbers of data segments and the acknowledgment numbers assuming a normal client or an attacker. After the first few data segments, the attacker starts its malicious behavior at 0.25s by sending a stream of Optimistic acknowledgements (plotted in green). As a consequence, the sender keeps increasing its congestion window and increases the speed of sending new segments (plotted in grey). As we can see, the server completes its transmission (plotted in grey) significantly earlier than it would normally do without the presence of Optimistic ACK attack (plotted in blue).

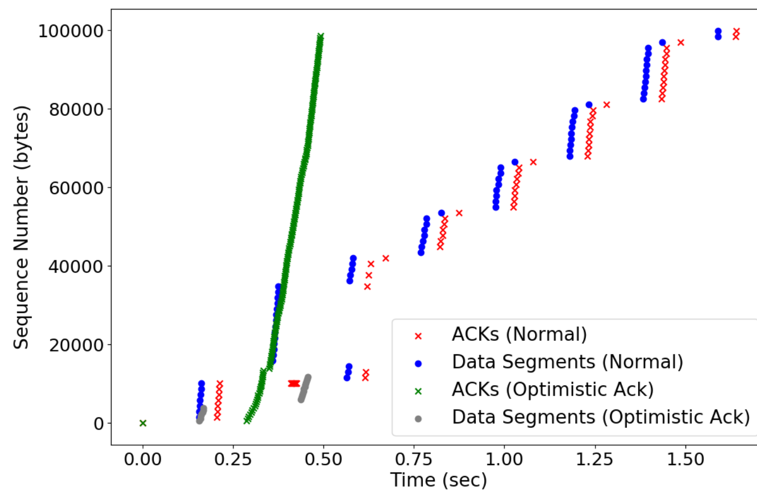
Figure 3.26 (b) demonstrates how our monitoring strategy effectively detects the Optimistic ACK attack and mitigates its impact. When the switch detects Optimistic ACKs, the EFSM corresponding to that TCP connection transitions to the misbehaving state. Consequently, the anticipated ACKs are dropped at the switch and no longer reach the server. Therefore, the server's congestion window does not incorrectly grow, hence, it does not significantly increase the data transmission rate.

3.4.3.2 Throughput evaluation

In this experiment, we evaluate the impact of the Optimistic ACK attack flow on the achieved throuhgput of a normal (non misbehaving) flow sharing the same path. The topology for this



(a) No reaction



(b) Reaction

Figure 3.26: Optimistic ACK in action

experiment is composed of two switches and seven hosts (Figure 3.27). Here, host h1 is the sender (the attacked server). We vary the number of attackers between 1 and 4 (hosts h4 to h7) (*i.e.,* to vary the number of flows from 2 to 5 (one normal flow with 1 to 4 attack flows)). We generate a normal flow between host pair (h2, h3) using iperf and measure the impact of the attack on its throughput. The link between the two switches is shared by all flows and is thus the bottleneck. We consider two scenarios: *Reaction*, where the switch applies the mitigation process and *noReaction*, otherwise. The results are presented in Figure 3.28.

Each box in Figure 3.28 represents the distribution of results obtained from 10 emulation runs of 60 seconds each. For a first baseline experiment, $iperf + Normal$, we have a non misbehaving flow alongside the iperf flow between h2 and h3 (*i.e.,* two non misbehaving flows).

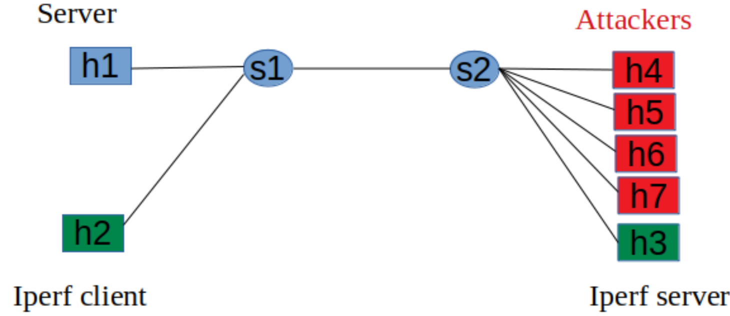


Figure 3.27: Topology

In this case, we observe similar behavior for both the *noReaction* and *Reaction* scenarios. The observed throughput variation due to the EFSM-based monitoring is very low.

Once we have a baseline, we incrementally increase the number of attackers. First, with a single attacker ($iperf + 1$ *attacker*), the throughput degrades in the *noReaction* scenario compared to the baseline, $iperf + Normal$. When we activate the reaction process, the throughput improves and gets even better than the $iperf + Normal$ case. This can be explained by the fact that the attacker's Optimistic ACKs are dropped, causing the server to stop transmitting further data segments. Consequently, the bandwidth of the bottleneck link becomes fully available for the iperf flow. When the number of attackers increases, the impact of the attack becomes higher as expected. In case of 2 attackers, (*i.e.*, 66% of the hosts being attackers), the reaction process recovers the throughput back to approximately the same level as the $iperf + Normal$ scenario. Starting from three attackers, the throughput is not completely recovered as we can see from the box-plot for the *Reaction* scenario. In this case, switch s2's ingress queues are flooded with Optimistic ACKs (on the link to the attackers). Although the Optimistic ACKs are dropped and their impact is mitigated on the shared link s1 − s2, the ACKs still need to enter switch s2 for processing. This consumes the switch resources, impacting the throughput recovery. Note that 3 attackers correspond to having 75% of the hosts being attackers. This is a very aggressive scenario and it becomes even worse with four attackers.

3.4.3.3 Switch processing time evaluation

In this experiment, we evaluate the overhead of running our mitigation solution in the data plane by measuring the increase in packet processing time in the switch. The results of this experiment are presented in Figure 3.29. The two groups of box-plot represent when forwarding is performed with monitoring disabled ($forward$) or enabled ($reaction$).

We deployed a topology with a single switch and two hosts, a sending and a receiving host. We generated 1000, 5000 and 10000 TCP flows and measured the per packet processing time from one emulation run. As we can observe from Figure 3.29, the median value is 2.9 ms for the $forward$ configuration and 3.3 ms for the $reaction$ configuration. The switch processing time overhead of our reaction process is 0.4 ms (≈14% additional processing time compared to $forward$). As shown also in Figure 3.29, our solution has no significant change in processing time overhead with respect to the number of flows to monitor. Note that both the processing time and the overhead will be significantly improved for a hardware target.

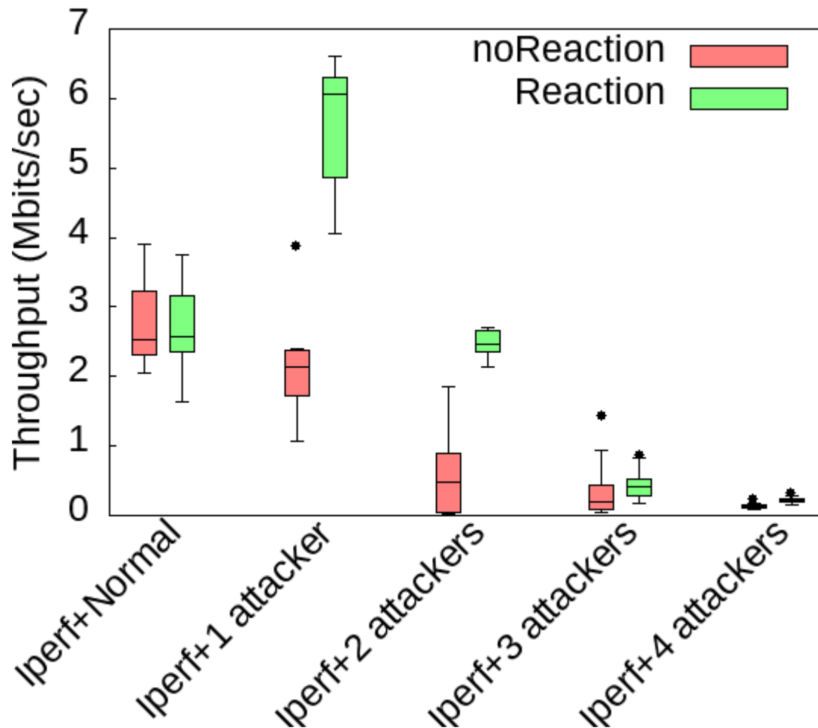


Figure 3.28: Optimistic ACK impact on the normal flow

3.5 Design challenges

The recent emergence of programmable switches has enabled the adaptation of attacks detection tasks. However, the programmable data plane has some hardware implementation challenges. This section discusses some scalability, hash collisions and memory issues.

3.5.1 TCP connection tracking

We track each individual TCP connection in the data plane as a separate flow. For maintaining per-flow persistent data across packets of a flow, we rely on hashing the 5-tuple flow key. TCP connection tracking through flow key hashing can lead to hash collisions that we address in Section 3.5.3. After a TCP connection terminates we do not update the associated EFSM anymore. The exact mechanism of how connection termination is detected is out of the scope of our work. A hardware switch can rely on a variety of mechanisms such as detecting FIN and RST flags in TCP segments [70] or using a timeout mechanism for evicting terminated TCP connections when a new flow with the same hash occurs. For the latter, the hardware target must be capable of precise timestamping (available in P4 hardware switches [71]). Furthermore, additional register entries per tracked connection would be required for storing the last seen timestamp as a context variable.

Another issue that may arise for TCP connection tracking in the data plane is sequence number wrap-around, (*i.e.*, when sequence number reaches its upper bound $(2^{32}-1)$), it restarts from 0. This is particularly problematic for use-cases where we need to track the sequence numbers in each flow such as in the Optimistic ACK EFSM. The TCP sequence number wrapping issue can be solved by employing techniques such as serial number arithmetic [72] or by enabling

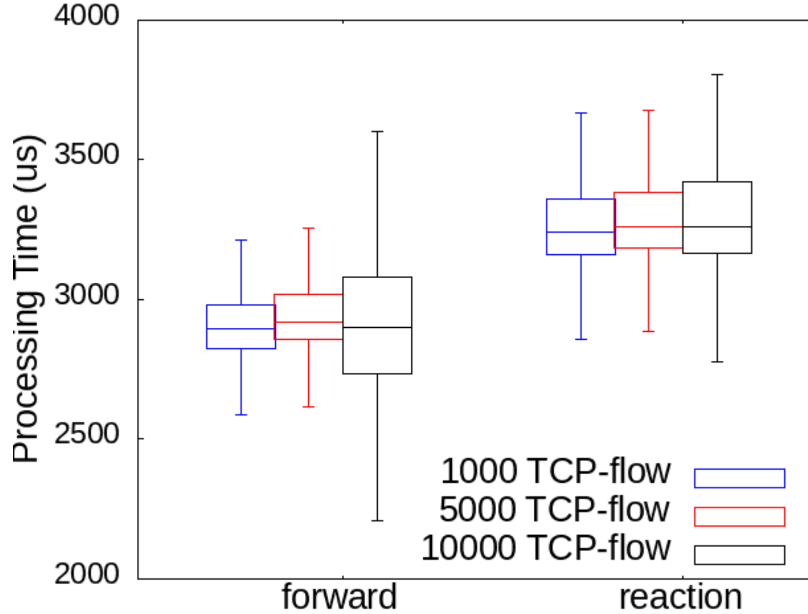


Figure 3.29: Switch processing Time

the TCP segment timestamp option [73]. Note that the latter will require tracking the last seen timestamp of each TCP connection as in the timeout based flow tracking scenario described above. Both of these techniques can be implemented in the P4 language and programmable hardware target constraints [61].

3.5.2 Scalability and memory overhead

A possible implementation issue is the support of P4 externs such as registers, necessary for maintaining persistent data across the segments of tracked TCP connections. Although P4 leaves the externs to be target-specific, registers are the most widely used externs and are expected to be supported by most P4 targets. For example, in [74], authors propose a P4 compiler for various FPGA hardware including stateful objects. The implementation of our solution on programmable hardware raises the issue of memory constraint. Although metadata variables that carry information between tables has a negligible bit overhead [6], the scalability of our method is limited by the available register memory in the hardware targets. As highlighted in Section 3.2, we need $|V| + 1$ registers per flow, one for maintaining the current state of the EFSM and one for each context variable. For n flows, our method requires $n \times (|V| + 1)$ register entries. For example, the EFSMs for the ECN and the Optimistic ACK attacks have $|V| = 1$ and $|V = 3|$, respectively. Assuming 32 bit values in registers, tracking 10,000 TCP connections will lead to allocate less than 64kB and 128kB, respectively, for these two use-cases. In comparison, state-of-the-art programmable hardware provides sufficiently large memory to accommodate state registers capable of holding tens of thousands of active TCP connections. For example, P4FPGA [74] can implement up to a 288 bits key for TCAM or hash-based memory and can fit up to 93K entries. Indeed, memory requirement will be higher with a higher $|V|$. In that case, we can employ optimizations such as lowering the size of register entries below 32 bits. In the current version, the number of states can be up to $2^{32} - 1$, sufficient to support any use case.

3.5.3 Hash collisions

The limited hardware resources in the data plane can pose challenges for tracking per-flow state [24, 25]. We follow the approach presented in state-of-the-art such as [24, 25] and maintain the hash of the 5-tuple flow key rather than maintaining the flow key itself to reduce the width of flow key used in hardware. This approach creates the risk of multiple flow keys getting mapped to the same hash value, (*i.e.*, hash collision). For a real deployment we can leverage techniques such as those described in [24, 25] for tackling the hash collision issue. Using multiple hash functions in conjunction with multiple hash tables reduces the number of collisions [25]. However, collisions are often unavoidable and therefore must be detected by employing methods such as checking if the sequence of keys stored against a hash value is correct [24]. Some existing hardware switches include mechanisms for handling hash collisions, however, these mechanisms are hardware-specific and cannot be generalized.

3.6 Summary

This chapter proposed an EFSM-based approach to detect attacks in the data plane. We demonstrated that an abstraction based on EFSM is expressive enough to capture attacks behaviors. We presented how this abstraction is mapped to the data plane P4 primitives. We presented a solution to mitigate TCP protocol misuse (*i.e.,* ECN misuse and Optimistic ACK attack). It can be deployed at low cost in the programmable data plane, we show that it is possible to mitigate such misbehaviors leveraging emerging programmable data planes while not requiring any end-host or protocol modifications unlike the state-of-the-art solutions. We showed that the throughput loss caused by misbehaving flows can be fully or partially restored.

EFSM are well suited for behavior monitoring. However, one of the shortcomings of these models is the fact that they do not allow to model concurrent behaviors (*e.g.,* parallelism). Indeed, attacks are becoming more sophisticated and it is sometimes difficult to identify the severity of an attack without considering a set of sequential or parallel attacker actions. To overcome this weakness, it is possible to use finite EFSM in combination with other models. We propose to extend our approach with a Petri Net model that can cope with competition and parallelism.

Chapter 4

Modular approach for detecting multi-step attacks in programmable data planes

Sommaire

4.1 Introduction **66**
4.2 Proposed approach **67**
4.2.1 Petri Net background 67
4.2.2 Approach overview 69
4.2.3 Models to compose attacks 70
4.2.4 From Petri Net model to P4 programs 72
4.3 DNS protocol **74**
4.3.1 DNS architecture and component 74
4.3.2 DNS security weakness 75
4.3.3 History of DNS security and some proposed solutions 77
4.4 Application to the new multi-step DNS cache poisoning attack **79**
4.4.1 Attack Petri Net 79
4.4.2 Port scan detection 81
4.4.3 DNS brute force detection 82
4.4.4 Evaluation 83
4.5 Summary **86**

4.1 Introduction

In Chapter 3, we proposed an approach based on an EFSM abstraction to detect and mitigate individual attacks. However, network attacks are becoming more complex and pass through a set of steps before achieving their ultimate goal and impacting the final target. These multi-step attacks are more difficult to be identified, especially since some of the steps are not too dangerous by themselves but only when combined with other steps. Attackers adopt a step-by-step methodology (*i.e.,* with intermediate objectives); unlike traditional attacks, individual actions have a minor impact. A multi-step attack has at least two different actions, for example, an attacker in a first step performs a port scan to identify system vulnerabilities, and in a second step, exploits these vulnerabilities to perform a DDoS attack. Another example is an attacker that first conducts an IP-spoofing and then launches a hijack attack. The attacker can evolve, change, or refine its behavior over time; thus, the composition of the attack steps can also evolve. Therefore, the level of the impact of an attack step can be used to determine the severity of another attack step.

In a sophisticated multi-step attack, the success of the attack depends on the success of the intermediate steps, so the degree of the security violation of a system must be identified and categorized based on the successful result an attacker may achieve in its different steps. Therefore, composing and recomposing the detection of the attack steps would be more beneficial to the flexibility and responsiveness of an attack detection system. For this purpose, in this chapter, we address the challenge of efficiently monitoring a multi-step and composed attack. Therefore, we propose to separate a decision module from the detection module (EFSM-based detection proposed in Chapter 3 or based on other models). The proposed approach allows specifying the appropriate mitigation level according to different composition of the attack behaviors. For this reason, the decision module is based on a Petri Net that allows synchronizing the set of detection modules. J.P McDermott [75] has introduced the application of the Petri Net for attack modeling and has demonstrated how the model is efficient for modeling concurrency, attack progress, intermediate and final objectives. Therefore, a Petri Net-based approach gives the possibility of composing behaviors at runtime and defining a set of reaction levels based on the composition of behaviors from a detection module.

Our method is defined to be deployed on data plane switches to run at line-rate and avoid the modification of the attacked protocol target on end-hosts in contrast to state-of-the-art-solutions. To do so, we take advantage of the flexibility given by the RMT Match Action Tables (MATs) that gives more flexibility since it can be configured at line-rate with a control plane. This chapter presents a Petri Net-based approach that can be mapped to a switch MAT representing a decision module; switch programs representing detection modules are synchronized using the Petri Net MAT. Once the solution is deployed on a switch, we can maintain and track flow states and attack steps in the data plane, allowing different levels of reactions at line-rate based on the composed detected behavior.

Historically, DNS has been prone to attacks; the cache poisoning attack is one of these attacks. *Kaminsky* [76] has discovered the DNS cache poisoning attack, demonstrating that the DNS cache can be poisoned by an attacker injecting spoofed DNS responses using brute force on the DNS *Transaction Identifier.* To mitigate the attack, the source port has been randomized. However, *Man et al.* [77] demonstrate how by dividing the DNS poisoning attack into several steps, the attack can be successful. State-of-the-art solutions (*e.g.,* DNSSEC) often require modifications of the protocol or require a lot of time for deployment due to compatibility issues.

This chapter is organized as follows. We present an overview of the Petri Net-based approach, followed by some background on Petri Nets. We then provide an overview of the proposed system and its basic modules in Section 4.2. We demonstrate the effectiveness of the proposed approach with an application to a Layer 7 DNS protocol attack. We also provide some DNS background and security history in Section 4.3. We describe how the approach detects and identifies the multi-step DNS cache poisoning attack and evaluate the effectiveness of our approach in Section 4.4. In Section 4.5 we summarize and conclude this chapter.

4.2 Proposed approach

This section presents the fundamental concepts of the proposed approach to detect multi-step attacks.

4.2.1 Petri Net background

Carl Adam Petri first introduced Petri Nets in his thesis in 1939 [78]. Petri Net is a general mathematical tool for describing relationships between conditions, events and describing dynamic systems behavior. Therefore, this model is used for modeling parallel and synchronized behaviors.

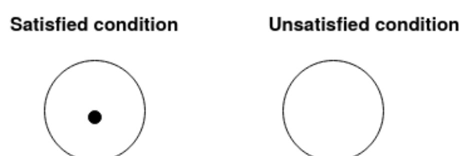


Figure 4.1: Satisfied condition

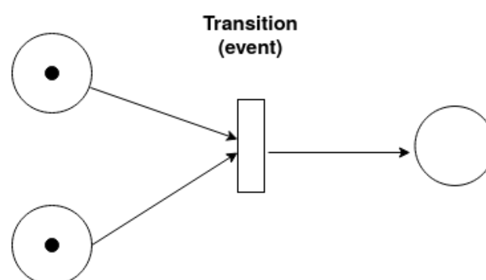


Figure 4.2: Example of Petri Net events

The Petri Net is represented by a graph with mainly two types of nodes: places and transitions. The places described by circles represent conditions. A transition indicates a computation or an event to be performed. The places contain tokens; a satisfied condition is presented by the presence of a token in a place (*i.e.,* drawn as black dots on places, Figure 4.1). An event occurs if all prior conditions are met (Figure 4.2). The firing of a transition (calculation step or event) corresponds to the consumption and production of tokens from one place to another. The places are connected to the transitions with incoming arcs (inputs) indicating the tokens that must be consumed, and outgoing arcs (outputs) indicating the tokens that should be produced in the place. The state of the Petri Net is represented by the current contents of the places (i.e.,

presence of tokens). An example of a Petri Net representing a connection establishment is given in Figure 4.3. We have two possible states of the connection: closed or open. The firing of the transition "open connection" makes it possible to pass from the closed state to the open state by consuming the token in the "closed" place and producing a token in the "open" place (Figure 4.3 (a)). The firing of the transition "close connection" consumes a token from the "open" place and produces a token in the "closed" place (Figure 4.3 (b)).

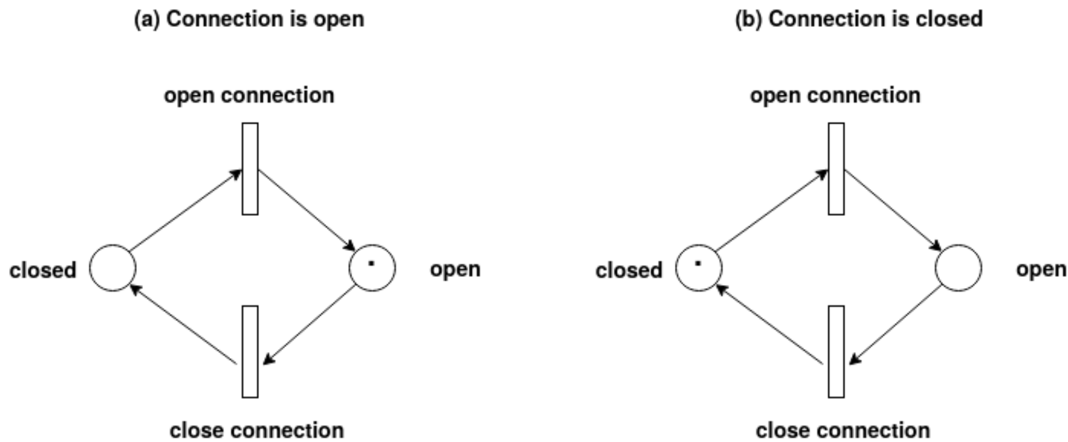


Figure 4.3: Petri Net connection establishment example

Petri Nets are very simple and may be utilized in different ways; they are a major model of concurrent systems, and their simplicity makes them an excellent choice for representing concurrency or parallelism. Synchronization is easily modeled using tokens when necessary. Therefore, Petri Nets may be the ideal choice for modeling systems of multiple processes occurring simultaneously or requiring synchronization [79]. The use of tokens represents the dynamic of the Petri Net (*i.e.,* moving tokens from one place to another allows tracking of simultaneous or parallel behaviors). Different transitions in the Petri Net may be enabled simultaneously, and so events can occur independently of each other. A Petri Net is composed of two parts: a static part and a dynamic part. The static part defines the Petri Net model, where places (conditions) and transition (events) are represented. The dynamic part defines the state of the Petri Net and the progression of its behavior according to the moving of tokens. Therefore, the state of a Petri Net is characterized by its marking [78].

Works have shown that the Petri Nets and attack tree models can be used to describe attack behavior and capture attack information. The attack tree approach adopts a tree representation of the dependencies among the actions performed by attackers [80]. In an attack tree, the attacker's final goal is represented by the root; the branches represent possible actions to complete the attack goal. The attack tree is limited due to the fact that it sequentially proceeds the attack steps, it focus on a single goal and a single attacker. In contrast, Petri Nets have been presented to be more suitable for modeling more complex attack behavior with more expressiveness. The utility of using Petri Nets for attack modeling was introduced by McDermott [75] as an alternative to attack trees for more expressiveness; it was observed that Petri Nets are more practical at capturing concurrent actions in an attack progression. The Petri Net places represent the attack steps, and the transitions represent actions performed by the attackers. Therefore, the Petri Nets approach provides more flexibility and expressiveness in capturing multiple and simultaneous actions [81]. Furthermore, in threat modeling [82], Petri Nets have also been used

to combine both attack attributes and system vulnerabilities in the same model. The ability of Petri Nets to model concurrent behavior has made it an appropriate tool to ensure the security of cyber-physical, industrial systems, and smart grid [83] [81]. In the field of cryptographic protocol verification [84] Petri Nets has gained a lot of interest in the research community as it gives a graphical presentation of the protocol, which is used to ensure that a security property is satisfied by a given cryptographic protocol. These solutions have shown that the Petri Nets can be used to describe attack behaviors.

In Petri Nets, tokens are not distinguishable. For this reason, Colored Petri Nets have been proposed to provide values and variables for the data using colors; these Nets are used to facilitate modeling and make a Petri Net more expressive, giving the possibility to describe complex attacks [85] [86]. However, for large and complex systems, the size of the Petri Net may be complex to manage and not scalable. The Petri Net needs to be extended with a structure of tokens to carry additional information, such as High-level Petri Nets (HLPN) [87].

All these works demonstrate that when managing security functions, attack modeling is helpful since it describes how attacks evolve. In a composed attack many objectives are implied, thus, an attack detection system needs multi-objective detection cooperation. Since the attack behavior is composed and the attack steps are synchronized, the detection model needs to define the notion of re-composition. Therefore, Petri Net is a good choice to model the detection and composition of a multi-step attack, identify and mitigate the attack based on the composition of suspicious behaviors.

4.2.2 Approach overview

In this section, we overview the proposed approach illustrated in Figure 4.4. Then we present the formal description of our model.

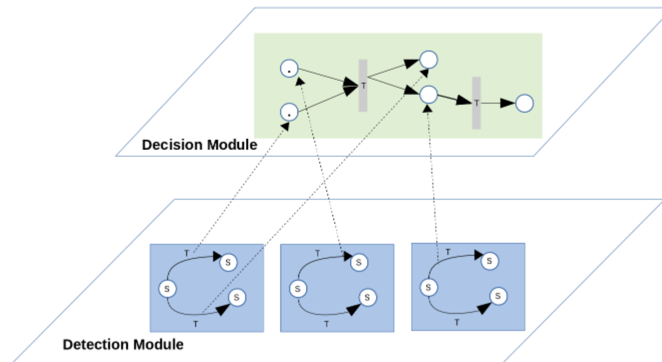


Figure 4.4: Approach overview

Multi-step and sophisticated attacks are based on a combination of behaviors. Each individual step contributes to the effectiveness of the attack. These steps can be sequential, parallel, optional, or alternative. We assume that each attack step aims at a particular sub-goal or stage. Based on the attack progression and reached sub-goals, we can define different levels of alerts and reactions. Figure 4.4 presents the proposed approach based on two layers modeling the relations between attack sub-goals and their compositions. The first layer which represents a

detection module (*i.e.,* lower layer) relies on a set of detectors in charge of monitoring individual sub-goals or stages of an attack. The second layer represents a *decision module* (*i.e.,* higher layer), which is synchronized with the set of behaviors detected from the first layer modules and defines possible decisions and alert levels. This high-level module verifies whether individual detectors have reached their sub-goals to decide on the attack progression based on a user-given attack representation. Indeed, our approach allows the user to compose the different behaviors according to his or her own needs.

- **First layer (Detection modules):** this layer includes a set of individual EFSMs models. In Chapter 3 we demonstrated that EFSMs could be mapped to P4 primitives within the programmable logic or using MATs. However, representing an EFSM with only MATs consumes many tables entries and is less efficient than a compiled program running the EFSM model. Compiled programs discard the possibility of modifying the EFSM model at run-time. So, every detector has to represent an EFSM of an individual attack stage or sub-goal (*e.g.,* scan attempt). It can then be pre-deployed on the switch and leveraged by the user through the combination of behaviors in the second layer. It is worth mentioning that non EFSM models could also be programmed and compiled with the P4 programmable logic to define other types of attack detection modules, but this is out of the scope of our work.

- **Second layer (Decision module):** in this layer, a Petri Net models the possible reactions and alerts based on the behavior combination information gathered from the detection modules. The relations between the first layer elements (composition, sequence, parallelism) are defined. The main advantage of the Petri Net model is its simplicity, which can be implemented using Match-Action Tables (MATs). As invoked in Chapter 2, the MATs are reconfigurable at runtime, so they support the flexibility of our approach. Therefore, a user can define and redefine the composition of the behaviors on the fly. The Petri Net describes the semantics of a multi-step attack events combination, synchronization, and composition. Petri Net places can represent attack sub-goals, and depending on the detected behavior at each EFSM, tokens are placed on places to indicate whether a certain sub-goal has been achieved. In the next section, we will detail how Petri Nets can be used to track a multi-step attack behavior.

The fundamental advantage of the proposed approach is that a Petri Net enables modular monitoring by mapping it into a MAT that is more configurable at run-time via a control plane. The MAT is configurable based on the attack risk levels and the composition of the set of an attack's sub-goals.

4.2.3 Models to compose attacks

We suppose that a multi-step attack is composed of a set of sub-goals $SG = \{sg_1, sg_2, .., sg_n\}$ and stages $AS = \{as_1, as_2, .., as_m\}$; each stage represents the composition of a set of sub-goals to achieve an intermediate objective and is associated with a reaction action. The set of sub-goals and stages are combined to reach a final attacker objective. Therefore, an attack ATK is defined as follows: $ATK = \{SG\} \cup \{AS\}$. Figure 4.5 illustrates an example of a Petri Net model, where an attacker can reach five sub-goals, $SG = \{1, 2, 3, 4, 5\}$, and five stages, $AS = \{A, B, C, D, E\}$.

We define a Petri Net PN by a 5-tuple $PN = (P, T, M, M_F, R)$ with:

- P, the set of places, represents the set of the attack's sub-goals SG and stages AS, such that, $\forall sg_i \in SG : \exists p_i \in P$, $\forall as_i \in AS : \exists p_i \in P$.

  A set of reaction actions, namely *decision*, can be associated with the set of places representing an attack stage AS, such that, $\forall p_i \in P\ \exists decision = \{\emptyset, drop, alert, .., etc.\}$. The decision can be, for example, doing nothing (empty), an alert, or a drop action. It is worth mentioning that this set of actions is not fixed and depends on the switch capabilities, which can be leveraged as actions in a MAT, (e.g., packets placement in different QoS queues or packet modifications). However, this assumes the reaction type is available (either native or implemented beforehand, similar to EFSMs). In the example in Figure 4.5 $P = \{1, 2, 3, 4, 5, A, B, C, D, E\}$, the achievement of sub-goals 1, 2 and 3, triggers the application of the decision associated with the stage C

- M a set of tokens, the presence of one token at a place indicates that an attack sub-goal $sg_i \in SG$ or an attack stage $as_i \in AS$ has been achieved.

- A marking function $M_F : P \times M$ denotes the marking of a place p with a token. This action is triggered by a module of the first layer (e.g., $EFSM$ model) to inject tokens into the Petri Net model when executed. This is different from the basic Petri Net models (Section 4.2.1), where tokens are initiated with an initial marking. In our case, tokens can be injected externally and the execution of the Petri Net can be changed based on external events (*i.e.,* modeled by EFSM).

- A transition $t \in T$ is connected to the input and output places with arcs. A transition is enabled if each input place contains at least one token and it produces one token in each output place. The transition represents a conditional set of an attack's sub-goals or stages to be validated to reach another stage. In our model, unlike the semantics of transitions in common Petri Nets (Section 4.2.1), a token is not consumed from a place by a transition to another place. Hence, when the attacker reaches an attack sub-goal or stage, this information is voluntary made persistent over time. In the example in Figure 4.5, a token will be added in place A if there is at least one token in places 1 and 3.

- An arc $r \in R$ connects places and transitions and is formally defined as the couple (i, o) where $i \in P$ and $o \in T$ or $i \in T$ and $o \in P$ exclusively. Hence, places cannot be connected directly.

Once the Petri Net model PN is defined, the execution of its transition system will change tokens through places. The state of its execution can be represented by the presence of tokens in each place and is formally denoted as $PN' = (p_1(m), p_2(m), \ldots, p_n(m))$. By analogy, $SG' = \{sg_1(status), sg_2(status), \ldots, sg_n(status)\}$ represents the different status of each sub-goal sg_i where $sg_i = 1 \iff p_i(m) > 0$ (sg_i is binary).

The Petri Net-based decision module is independent of the detection modules because the marking function M_F can be instantiated by any type of modules. However, in our work, we adopt the EFSM abstraction for modeling the detection of each sub-goal related to an attack. We have shown in Chapter 3 that this abstraction supports the right level of expressiveness to capture stateful attacks without the explosion of the number of states and, most of all, it can be mapped into the P4 primitives. At the switch level, packets are parsed and the extracted

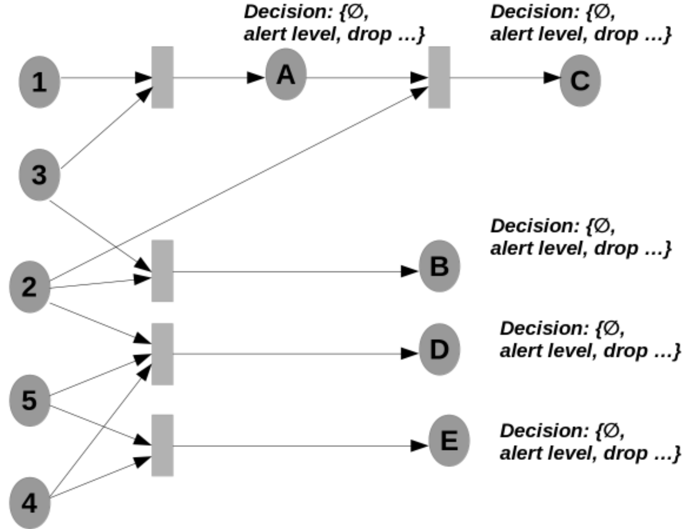


Figure 4.5: Petri Net Example

information from headers is used to change the current state of the EFSM and apply actions (*e.g.* drop a packet, modify packet header, etc.). To cope with the proposed layered model, the decision module (Petri Net) must synchronize with the detection module (the set of EFSMs). Hence, the EFSM action set is augmented with a marking function to add a token m to the Petri net at a specific place p.

4.2.4 From Petri Net model to P4 programs

As discussed in Chapter 2 switches can rely on MAT to process packets. This section elaborates on our design to run Petri Nets on switches using MAT.

To map and execute the Petri Net model in a P4 programmable pipeline, the Petri Net model needs specific constructs. As highlighted in Section 4.2.3, a transition is triggered when all input places are marked. It is worth mentioning that a transition requires at least one token in the input places; in our case, tokens are maintained and made persistent once set in a place. Therefore, keeping track of the number of tokens can be omitted. For the same reason, the attack stages do not need to be maintained because they can be recovered from the persistent tokens in places representing sub-goals.

A single MAT is used to model the Petri Net, where the lookup keys represent the places of the sub-goals $p \in SG$, one bit for each $p \in SG$. For example, 8 places are represented as a single byte and 10010000 is the match key when tokens are present on p_0 and p_3 (Figure 4.6).

To keep in memory the state of the Petri Net, (*i.e.* tokens), we rely on the P4 registers. However, they cannot be used directly in MAT lookup keys. As highlighted in Chapter 2, when processing a packet, P4 allows the use of metadata to get contextual information about the switch (*i.e.,* queue occupancy) but there is also user-defined metadata. Thus, it is possible to read register values as metadata similarly when parsing a packet header structure and save the corresponding metadata in a register. Therefore, we define a set of metadata representing the places $p \in SG$ and use the registers to maintain them persistent over the processing of packets subsequently:

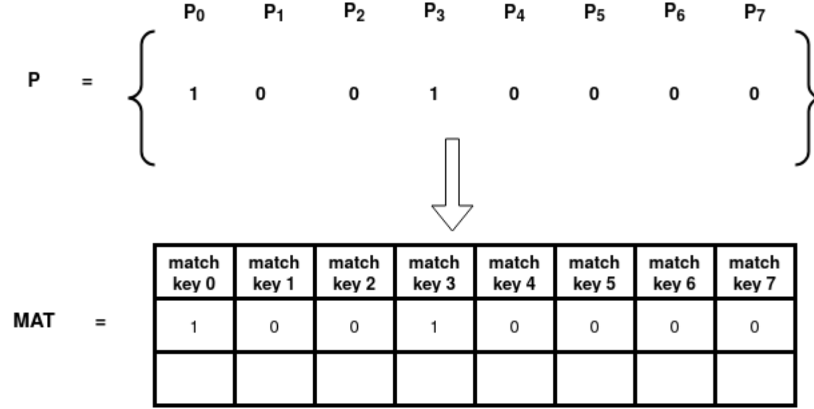


Figure 4.6: Places representation

$Meta = \{meatadata_1, metadata_2, .., metadata_n\}$ where $\forall sg_i \in SG : \exists metadata_i \in Meta$ with $metadata_i = 1$ when sg_i has been reached (a token is in the place), 0 otherwise. Each entry in the MAT with match values ($\in Meta$) set to 1 represents the set of sub-goals composing a stage achievement $as_i \in AS$, and so the action of this match is the reaction defined for as_i. Table 4.1 corresponds to the Petri Net depicted in Figure 4.5:

- The first line represents the achievement of the sub-goals 1 and 3, the applied action (e.g., alert level 4) is the one associated with the stage $A \in AS$
- The fourth line represents the achievement of the stage D composed of the sub-goals 2, 4 and 5
- The third line represents the stage C corresponding to the achievement of the sub-goal 2 and stage 2, which in turn corresponds to sub-goals 1 and 3

As highlighted previously, tokens are persistent, thus, the attack stages are easily recovered from sub-goals without maintaining the individual states of each stage.

Table 4.1: Petri Net as a MAT

meta 1	meta 2	meta 3	meta 4	meta 5	actions
1	0	1	0	0	$decision \in A$ (e.g., alert level 4)
0	1	1	0	0	$decision \in B$ (e.g., alert level 3)
1	1	1	0	0	$decision \in C$ (e.g., drop)
0	1	0	1	1	$decision \in D$ (e.g., drop)
0	0	0	1	1	$decision \in E$ (e.g., alert level 1)

To detect and react against a composed attack, the detection and decision modules need to be synchronized. Algorithm 2 describes how the first and second layer are synchronized. Lines 2-4 define the synchronization part of the first layer, if an attack sub-goal $sg_i \in SG$ is achieved (line 2), a token is set on a place p_i by the action $setToken(p_i)$ triggered by the EFSM, that uses the marking function. In the data plane, marking tokens on places is represented by setting a $metadata_i \in Meta$ to 1 to indicate the presence of a token on a place p_i (line 6). The Petri Net tuple is abstracted to a MAT in the data plane, the set of match fields are the set of metadata $\in Meta$ and the *decision* is a predefined action with respect to the achieved attack stage (line 7).

Algorithm 2 EFSM and Petri Net synchronization algorithm

Definitions:

- n: number of the attack sub-goals SG
- $setToken(p)$: function triggered by an external model to set tokens

1: **Synchronization between** $EFSM$ **and** PN
2: **if** $sg_i(status) = achieved$ **then**
3: $setToken(p_i) \Longrightarrow M_F : P \times M$
4: **end if**
5: **Synchronization between** PN **and** MAT
6: $M_F : P \times M \Longrightarrow metadata_i = 1$
7: $PN : (p_1(m), p_2(m), .., p_n(m)) \Longrightarrow table(match_1(metadata_1), match_2(metadata_2),$
$..., match_n(metadata_n) \mid decision)$

4.3 DNS protocol

We have chosen DNS as the application protocol for our approach due to the recently identified DNS cache poisoning attack that passes through a set of steps [77]. In this section, we provide a background on DNS, including the main components and functioning of the protocol. We invoke the history of DNS security issues, vulnerabilities of this protocol, and some proposed solutions.

4.3.1 DNS architecture and component

The Domain Name System (DNS) is an essential part of the Internet to access different services. The purpose of DNS is to provide a translation service called name resolution. It is responsible for translating a name (easier to remember) to an IP address and vice versa (*i.e., reverse DNS*). Two DNS entities are used for name resolution; the name server (authoritative server) maintains information on domain names; its primary function is to answer queries using data from its domain names file. The resolver (recursive server), which is the server that asks questions to authoritative servers and memorizes the answers to answer client requests, Figure 4.7 represents the interaction between an authoritative server and a resolver.

One of the essential criteria of a resolver is to minimize (or even eliminate) the delays due to the network and to the resolution of the queries; the rapidity of response is indeed an essential criteria for DNS servers. To ensure a fast response, name servers and resolvers have a cache that stores the records they have recently accessed. If the answer is available, the use of a cache avoids sending a request on the network and thus decreases the network load and the response time.

The DNS message format [88] is presented in Table 4.2. We only detail the fields that we use in the rest of the chapter. These fields are the Question and Answer fields, which include the requested and answered domain names, the DNS identifier (transaction ID) and the message type (QR).

Domain name composition and format: questions and answers contain domain names (*i.e.,* QNAME, NAME in the question and answer fields, respectively). A domain name is a list of labels; a label is a set of characters preceded by the length of each label. The length of each

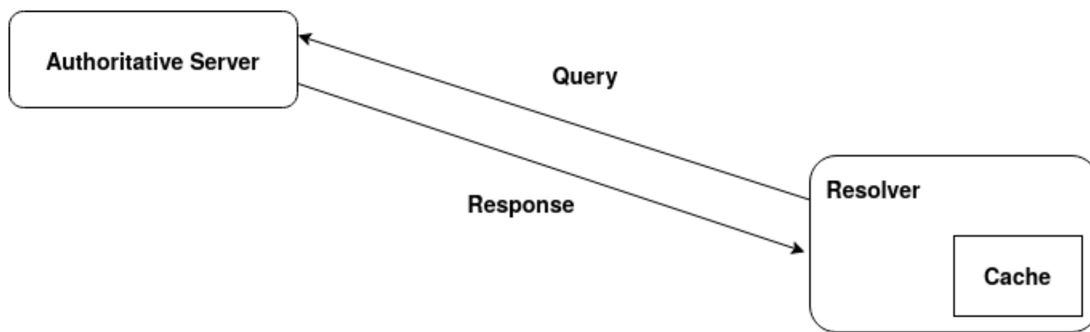


Figure 4.7: DNS Query Response

Table 4.2: DNS message

Identifier
Parameters (QR, OPCODE, AA, TC, RD, RA, UNUSED, RCODE)
QDcount
Ancount
NScount
ARcount
Question (QNAME ...)
Answer (NAME ...)
Authority
Additional

label is between 0 and 63 bytes. The last label is 0 byte long; it is the only label whose length can be 0 byte. A domain name has a maximum total length of 255 bytes. Each label is encoded in ASCII, with a "." character between each label. The domain name "www.example.fr" contains three labels, "www", "example" and "fr".

Transaction ID (Identifier): coded on 16 bits, this field must be recopied from the DNS question during the DNS response, allowing the sender to identify the DNS response. Therefore, these first two bytes must be identical in the question and the response.

Flags: as part of the set of DNS header flags, the "QR" code is on a 1 bit; this field allows to indicate if the packet is a DNS request (QR = 0) or a response (QR = 1).

4.3.2 DNS security weakness

When DNS was first designed, security was not prioritized. It is exposed to many potential attacks, which mainly use brute force strategies or corruption of the information exchanged between resolvers and name servers. Among the most well-known attacks is the DNS cache poisoning, whose purpose is to redirect users to a fake website. For example, a user types "real.mail.com" (i.e., having the IP address 1.1.1.1) into a web browser to check his email box. Because DNS has been poisoned, it is not the "real.mail.com" page that is displayed, but a spoofed page chosen by the attacker (*i.e.,* fake website having the IP address 2.2.2.2)) (Figure 4.8).

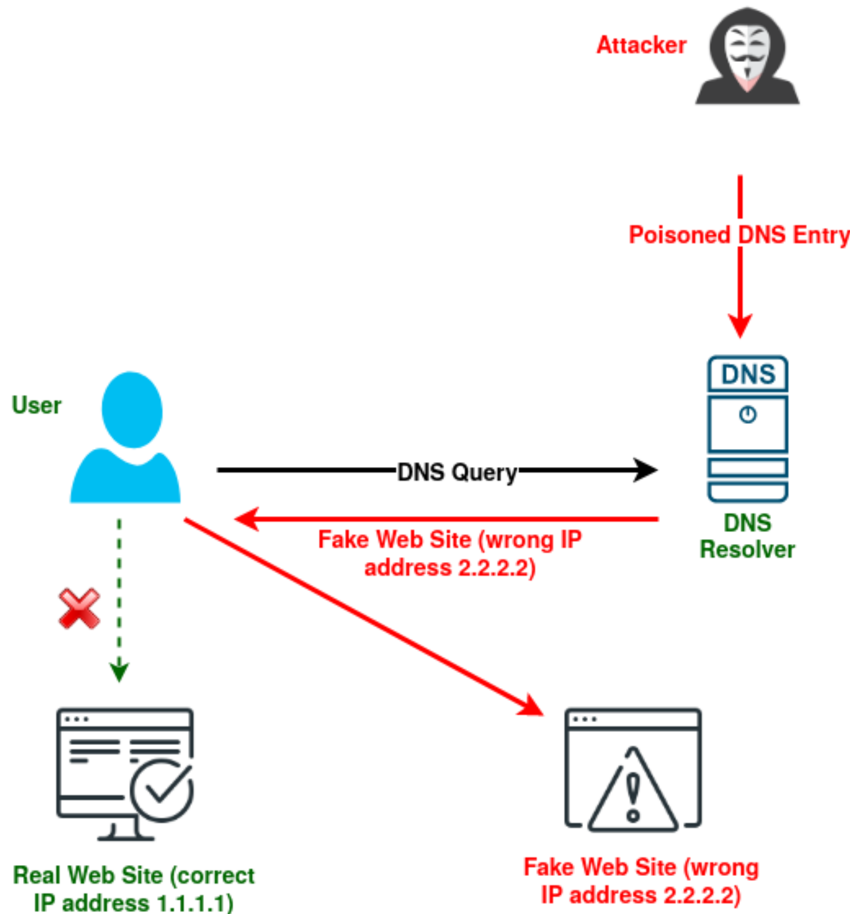


Figure 4.8: DNS cache poisoning overview

To carry out this attack, the DNS caching system is explored. The trust given by the server to the data it has placed in its own cache makes it a particularly interesting target for attacks. When the authoritative name server sends a response to the recursive resolver, the latter can not verify the legitimacy of the response (*i.e.,* the DNS query transaction ID and the port number). Only the IP address of the response that is the same as the one to which the request was sent can be verified by the resolver. However, relying on the source IP address of a response is not a strong authentication mechanism since that the IP address can easily be spoofed. From the way DNS was originally designed, a resolver cannot easily detect a forged response to one of its queries. Therefore, an attacker can easily impersonate the authoritative server to which the resolver sent the original query by spoofing a response that seems to originate from that authoritative server. This means that users can be redirected to a malicious website without noticing. Recursive resolvers cache the DNS data they receive from authoritative name servers to speed up the resolution process. If a resolver requests a domain name that the recursive resolver holds in its cache, the recursive resolver can respond immediately without first sending a query to the authoritative servers. However, there is a downside of using the cache; if an attacker sends a poisoned DNS response accepted by a recursive resolver, the attacker has poisoned the recursive resolver's cache. The resolver will then resend the poisoned DNS records to other users who have made the DNS request.

In a DNS cache poisoing attack (presented in Figure 4.9), the attacker must know or guess the DNS transaction ID, which is randomly generated by the DNS resolver. The attacker knows the UDP source port of the DNS response, which is 53 (the standard DNS port), and so only the destination port is unknown (nominally 16 bits, practically about 16 bits of entropy), which is randomly generated by the resolver. Thus, the attacker must predict or guess the UDP destination port and the transaction ID to poison the resolver cache (*i.e.,* generate a correct response corresponding to the request forcing a name server to cache a false response and thus poisoning its cache). Detecting DNS cache poisoning attack is complicated, poisoned data can remain in the cache until the TTL, which may take several days to correct the problem.

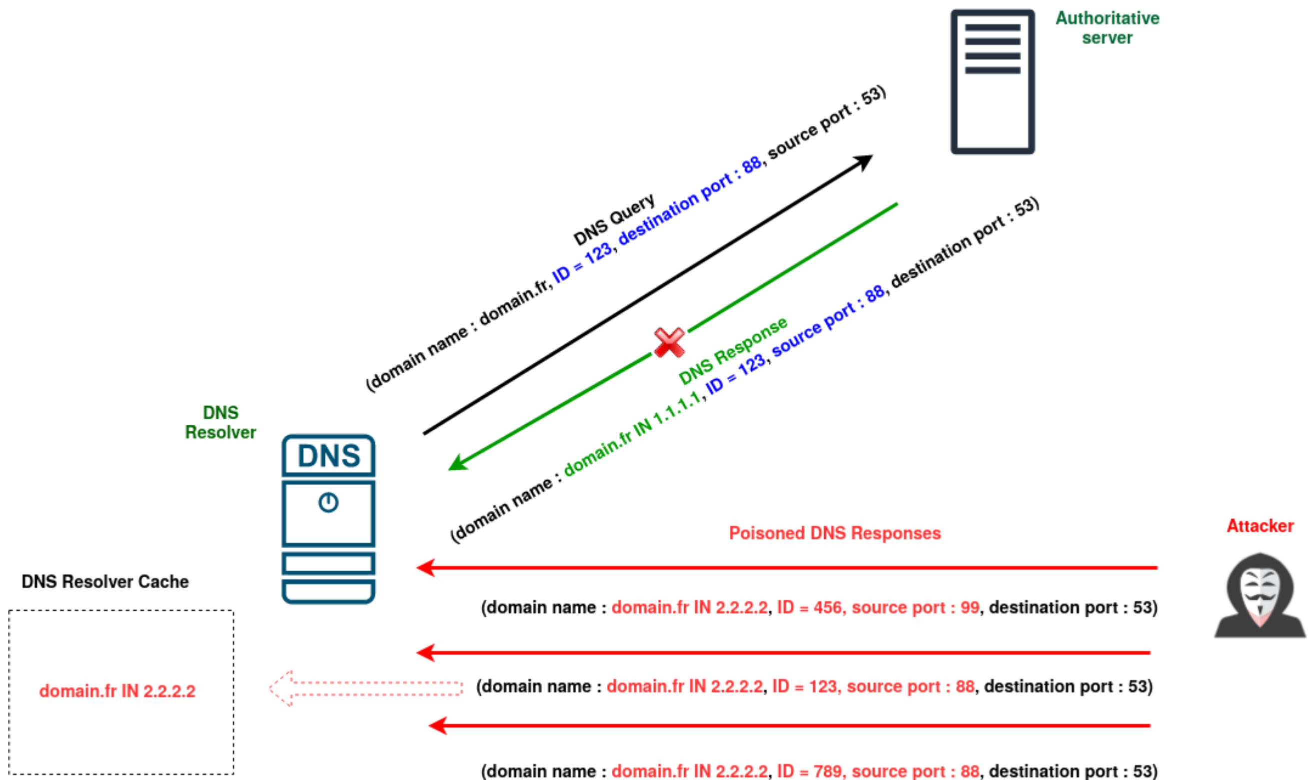


Figure 4.9: DNS cache poisoning attack

4.3.3 History of DNS security and some proposed solutions

In 2008, a security researcher Daniel Kaminsky revealed a major vulnerability concerning the DNS system, allowing an attacker to send authoritative responses to domains for which the server was not authoritative and thus impersonate any website. The attack was called the "Kaminsky attack". All that was needed was to brute-force the transaction ID by sending many responses that varied this field. This identifier is encoded in 16 bits, which means that there were at most 65,536 possibilities. If the attacker is fast enough, the real response arrived too late and could be rejected. After this attack, DNS resolvers now choose the source port of the query at random, which is called source port randomization. Therefore, an attacker would have to brute-force not only the transaction ID (16 bits) but also the source port (16 bits). As a result, the number of possibilities is now too high (*i.e.,* 32 bits) for the attack to be easily feasible but not impossible.

Recent techniques along with source port randomization have been proposed to prevent DNS cache poisoning attacks. Defense methods such as 0x20 encoding, and DNSSEC were proposed.

DNSSEC (Domain Name System Security Extensions) is a standardized protocol that secures DNS data and allows securing the authenticity of the DNS response. By default, the DNS data is not encrypted, letting an attacker to intercept or even modify a transaction, mainly to direct a user to a fake server, without his knowledge. DNSSEC does not encrypt the connection, but signs the response using asymmetric cryptography, allowing the client to confirm its validity. However, DNSSEC relies on DNS itself to operate as a Public Key Infrastructure (PKI) to distribute public keys [89]. DNSSEC has a few deployment compatibility problems [90]; it requires a significant modification to the current DNS design and extensive collaboration from all parties involved (i.e., the modification of all DNS hierarchy levels, the introduction of security functions to all operating systems and the introduction of PKI operations that incur significant overhead since PKI operations are known as computationally expensive). Although it has been in development for over ten years, its large-scale deployment remains complicated. In addition, recently [91] discovered that vulnerable DNSSEC keys are used to sign many domain names.

Another solution is the 0x20 encoding [92], which uses upper and lower case letters in the question domain name to obtain randomness. These random bits in the request prevent spoofing attempts. With this technique, the DNS server response must match not only the domain name in the query but also the case of each letter in the domain name; for example, the two domain names, www.GoOgLe.CoM and wWW.gOoGle.COm are not equivalent. This may add some entropy to queries which would be effective for spoofing prevention. However, DNSSEC and 0x20 encoding are still far from being widely deployed [89] due to compatibility issues and the need to modify the DNS server.

A study on DNS cache poisoning has been conducted on different networks [93] (*i.e.,* published in 2017), including networks of large Internet Service Providers (ISPs), public DNS service operators, and enterprise networks. The study evaluated injection vulnerabilities in DNS resolution platforms that allow cache poisoning. The evaluation revealed significant security issues; more than 92% of DNS platforms are vulnerable and can be poisoned with some efforts. On the other hand, RFC5452 [94] notes that NAT devices significantly reduce the effects of source port randomization. A resolver behind a NAT uses source port randomization for all DNS queries, when the query crosses the NAT, the NAT translates the source port number into a new source port from the NAT's local port number pool. NAT devices are not considered as part of the DNS hierarchy thus have not been considered by defenses against DNS attacks.

Moreover, there may be vulnerabilities in the system that will facilitate DNS cache poisoning attacks. In [95] a weakness has been discovered in the pseudo-random number generator (PRNG) used for generating UDP source ports in the Linux/Android kernel. This vulnerability can speed up the DNS cache poisoning attack by a factor of 3000 to 6000. A different class of DNS poisoning attacks is described in [96]. The attacker initiates the attack on the client cache using a malware. The attack was conducted on Windows, Mac OS, and Ubuntu Linux systems. The attacker adopts a local port reservation forcing the server to pick one available port; the attack is effective even behind a NAT. In [97], the authors propose to duplicate DNS queries and send them to multiple resolvers for verification purposes, but this entail a significant overhead. In [98] a randomly generated 4-bits with the random port and transaction ID is used. These two solutions require the modification of the DNS protocol.

Recently, Man *et al.* [77] have introduced a new attack capable of poisoning the DNS server cache. The new method is called Side Channel DNS Attack. The authors have found weaknesses that allow an attacker to divide the attack by guessing the DNS server source port using a port scan in the first stage and then guessing the DNS transaction ID in the second stage. The authors analyzed 14 public resolvers, including Google, Cloudflare, Dyn, and OpenDNS. Only two were not vulnerable to this attack, namely Tencent DNS and Ali DNS. The authors have also analyzed more than 138,000 so-called "open" resolvers (freely available but not well known). Of those, 34 % were vulnerable. The Linux kernel can be patched using rate-limiting methods to counter this attack. However, these methods have a collateral impact on collocated traffic; DNSSEC can as well mitigate this attack.

These works demonstrate that the DNS cache poisoning attack can still be conducted and is a valid problem to consider. Unfortunately, many resolvers will be vulnerable until one proposed solution is fully applied and deployed. Referring to all these works, we can see that even large organizations are vulnerable to different types of attacks, and we can note from these examples the evolution of attacks over time. In what follows, we present a solution that does not require any protocol modification as it reacts within the network, in contrast to state-of-the-art proposed solutions that require the modification of the DNS protocol and require a considerable time for deployment.

4.4 Application to the new multi-step DNS cache poisoning attack

In this section, we describe the recent DNS multi-step attack invoked in [77]. We present the Petri Net-based model as the decision module and the EFSM-based models as the detection modules to detect and mitigate this attack. We evaluate our approach with this use case in Section 4.4.4.

In the recent DNS cache poisoning attack, an attacker starts by sending a DNS request to the DNS server victim as shown on the left of Figure 4.10 (step 1). This server is in charge of the recursive resolution by requesting other DNS servers within the Internet (step 2). The attacker's objective is to reply first with a valid answer to one of these requests. To be valid, an answer must match the domain initially requested, the source port and query ID used by the DNS server. As highlighted in Section 4.3.3 in the past, the source port was not randomized, and it was feasible to brute force the ID [99]. Due to this flaw, the UDP source port is now randomized, making the brute-forcing complicated (quadratic complexity). An attacker can alleviate this issue by performing a UDP port scan and expecting the DNS server to respond with ICMP unreachable messages for all closed ports except for the open one (steps 3 and 4). Using this strategy, the attacker can first guess the source port and then brute-force the query ID in step 5 (linear complexity).

4.4.1 Attack Petri Net

We decompose the DNS cache poisoning attack into 3 sub-goals $SG = \{sg_1, sg_2, sg_3\}$: sg_1 corresponds to the reception of a valid response (valid domain name and valid ID) that matches the outgoing server query and the port used in the response is identified as open; sg_2 represents the UDP port scan; sg_3 is the DNS brute force attack.

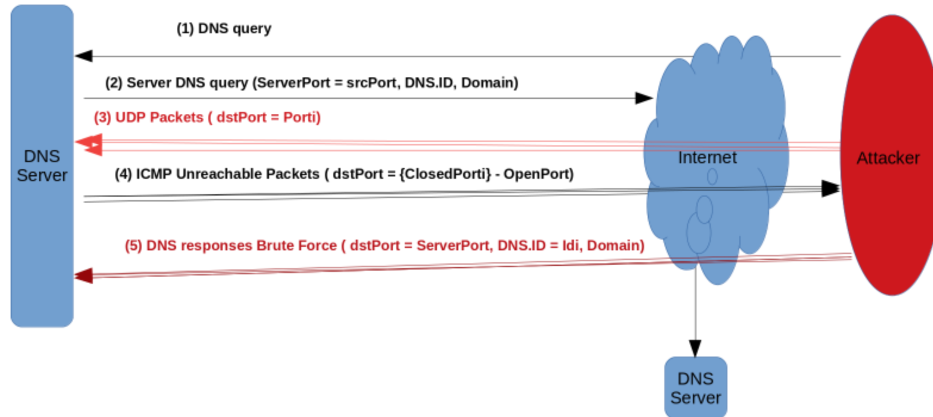


Figure 4.10: Multi-step DNS cache poisoning attack

As highlighted in Figure 4.11, there are three sub-goals (*i.e.,* represented as places) and three attack stages with different alert severity levels (i.e., *decision*). For example, when an attacker performs a port scan sg_2 and a DNS brute force sg_3, the Petri Net enters into attack stage 5 (alert level 5). However, to be effective, the attacker has also to send a DNS valid response sg_1. So, if all sub-goals are reached, all attack stages are reached too, including the most critical with the action *Drop*.

In this attack, a valid answer is also expected from the legitimate DNS server. Hence, observing sg_1 only is not helpful for distinguishing an attacker, so we must rely on observing other side attacks. We have to jointly monitor three types of traffic: DNS, UDP and ICMP traffic. We have to observe both the brute force and a valid answer in the DNS traffic. Therefore, sg_1 and sg_3 are monitored by a DNS-specific module while sg_2 is handled separately.

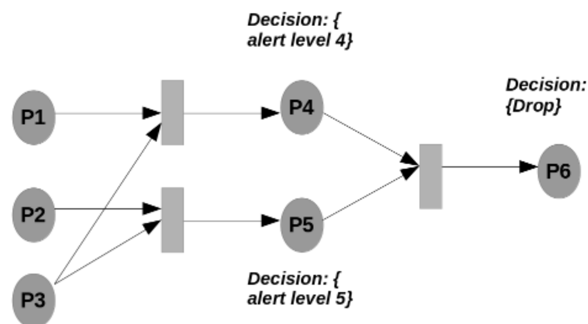


Figure 4.11: Petri Net of DNS cache poisoning

It is worth mentioning that the different detection modules require the sharing of some information to be effective. For instance, valid answers can be emitted by a DNS server, our second layer model relies on detecting the port scan to distinguish between a benign and malicious valid answer. This implicitly assumes that the port scan sub-goal has been achieved (sg_2) and that the attacker has discovered the port used for the valid answer (sg_1). Independently of the second layer performing Petri Net execution, the first layer modules may need to share information.

Like token management, data sharing between detection modules is performed using registers denoted as context variables. Table 4.3 illustrates the corresponding DNS cache poisoning Petri Net in Figure 4.11. The attack model is as follows: $ATK : SG = \{p1, p2, p3\} \times AS = \{p4, p5, p6\}$ with the set of the attack sub-goals SG, possible attack stages AS with possible associated reaction actions. The DNS cache poisoning Petri Net is abstracted to a MAT with a match key on metadata $meta1, meta2, meta3$ corresponding to the presence of tokens on places $p1, p2, p3$.

Table 4.3: Attack MAT

meta 1	meta 2	meta 3	actions
1	0	1	alert level 4
0	1	1	alert level 5
1	1	1	Drop

We define the set of alert levels based on the severity of the attack achievement. The first line, $meta\ 1 = 1$ and $meta\ 3 = 1$ represents the combined achievement of sg_1 and sg_3, which is associated with an alert of level 4. In the second line, $meta\ 2 = 1$ and $meta\ 3 = 1$ represent the combined achievement of sg_2 and sg_3, associated with an alert of level 5. The last line denotes the achievement of the attack three sub-goals $SG = (sg_1(achieved), sg_2(achievd), sg_3(achieved)) \Longrightarrow PN = (p_1(m), p_2(m), p_3(m)) \Longrightarrow table(match(meta1 = 1), match(meta2 = 1), match(meta3 = 1)$ with a drop as an action from $p6$'s set of *decision*.

We present the EFSM model to detect the set of the attack sub-goals in the next sections.

4.4.2 Port scan detection

This section describes the detection module based on an EFSM abstraction for the port scan attack sub-goal detection. Figure 4.12 represents the EFSM-based detection module. The EFSM is defined as follows:

- $S = \{ListenUDPQueryICMPResponse, ScanAttempt\}$;
- $E = \{IP.dst = DNSserverIP, IP.protocol = 17, ICMP.type = 3\}$
- $V = \{MetaQR, DNSserverIP, THscan\}$;

This EFSM verifies if an attacker performs a port scan. Therefore, the EFSM maintains a counter of UDP packets sent from the potential attacker to the server, stateful variables used in EFSM are mapped to registers. A register, $CountScan$, is used to track this value in a stateful manner using the DNS server IP address as an identifier (potential victim to be protected). The source IP addresses are not considered in our model as we assume that the attacker can perform the attack from distributed hosts.

When a predefined threshold is reached, a transition to the state $s = ScanAttempt$ is triggered. Once this attack sub-goal sg_2 has been met, a token in the Petri Net is set on the place $p2$ representing the attack sub-goal ($setToken(p2)$).

To track open ports, we introduce the context variable $MetaQR$, which is a stateful variable maintained in a register with the UDP port as a key. When a UDP packet is received, the $MetaQR$ value is set to 1, meaning that the port is assumed open until an ICMP unreachable reply is received (*i.e.,* type 3). In that case, $MetaQR$ is set to 2 (*i.e.,* port closed).

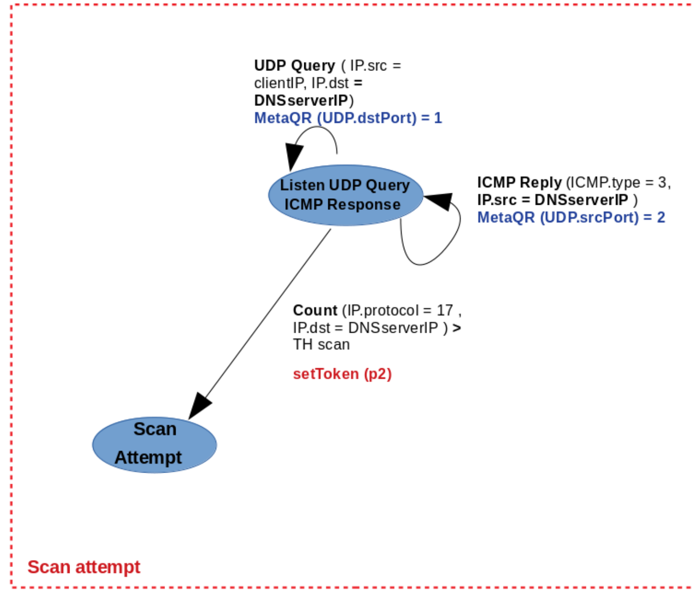


Figure 4.12: Port scan EFSM

4.4.3 DNS brute force detection

In this section we describe the detection module based on an EFSM abstraction for the brute force attack stage detection. Figure 4.13 describes the attack stage detection module composed of the sub-goals sg_1 and sg_3, the EFSM is defined as follows:

- $S = \{ListenDNSQueryResponse, Bruteforceconfirmed, MatchedResponse, Attackconfirmed\}$;
- $E = \{IP.dstIP = DnsServerIP, IP.srcIP = DnsServerIP, UDP.dstPort = 53, UDP.srcPort = 53, DNS.ID, DNS.Domain\}$
- $V = \{MetaServerQR, DNSserverIP, THbrute\}$;

In the DNS cache poisoning attack, a set of invalid responses are expected before a legitimate response that matches the outgoing server query. Therefore, counting unmatched queries and comparing this counter with a specific threshold can be effective for attack detection. To detect the match and mismatch of DNS responses, a context variable $MetaServerQR$ maintained in a register and associated with a key is used to track server queries, (*i.e.,* the key is based on the port number used in the DNS query and response, the transaction ID, and the requested domain name, these three fields are sufficient to identify the DNS request or response). Below we explain the detection steps of the attack:

- When a server query is received, $MetaServerQR$ is set to a value (e.g., set to 1) with a hash key identifier composed of $(UDP.srcPort, DNS.ID, DNS.domain)$.
- When a DNS response is received, the value of the $MetaServerQR$ is verified using the reverse port in the key $(UDP.dstPort, DNS.ID, DNS.domain)$ to check the match with the server query sent previously (*i.e.,* $MetaServerQR = 1$).

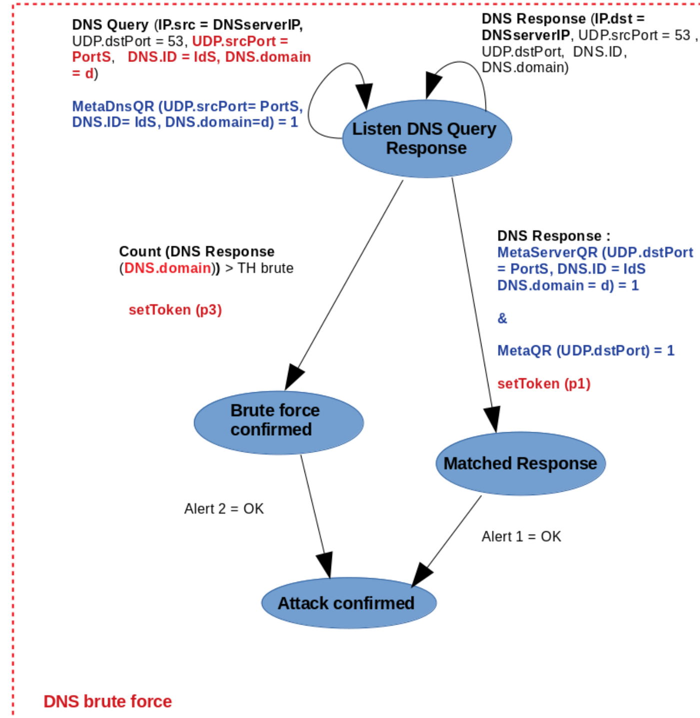


Figure 4.13: DNS brute force EFSM

- Receiving a valid response that matches the server query and the port identified previously in a scan attempt (*i.e.*, $MetaQR = 1$ (shared context variable)) results in setting a token in the Petri Net at the place p1 ($setToken(p1)$) associated with the sub-goal sg_1.
- Reaching a threshold of invalid responses results in setting a token on the place $setToken(p3)$ corresponding to the sub-goal sg_3. Similarly to port scan, a register $CountDNSResponse$ is used to maintain the number of the received DNS responses to detect the brute force.

4.4.4 Evaluation

In this section, we present our results that demonstrate the effectiveness of the proposed approach. First, we evaluate the efficiency of the MAT abstraction in detecting the attack. Then, we evaluate the switch processing time when we increase the number of the attack sub-goals. The evaluation environment is the same as the experiments in Chapter 3.

4.4.4.1 DNS cache poisoning attack mitigation

In this experiment, we assume a topology composed of one switch, one server, and a single attacker (Figure 4.14). We evaluate the effectiveness of the proposed approach when changing the Petri Net MAT configurations:

- (a) only drop packets when all sub-goals are achieved (P6 in Figure 4.11),
- (b) when at least sg_1 and sg_3 are achieved (P4 or P6 in Figure 4.11),
- (c) when at least sg_3 and sg_2 are achieved (P5 or P6 in Figure 4.11).

Therefore, the Petri Net-related MAT is configured accordingly.

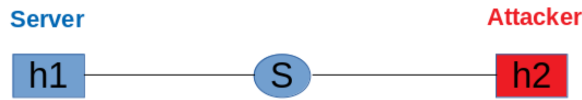


Figure 4.14: Topology

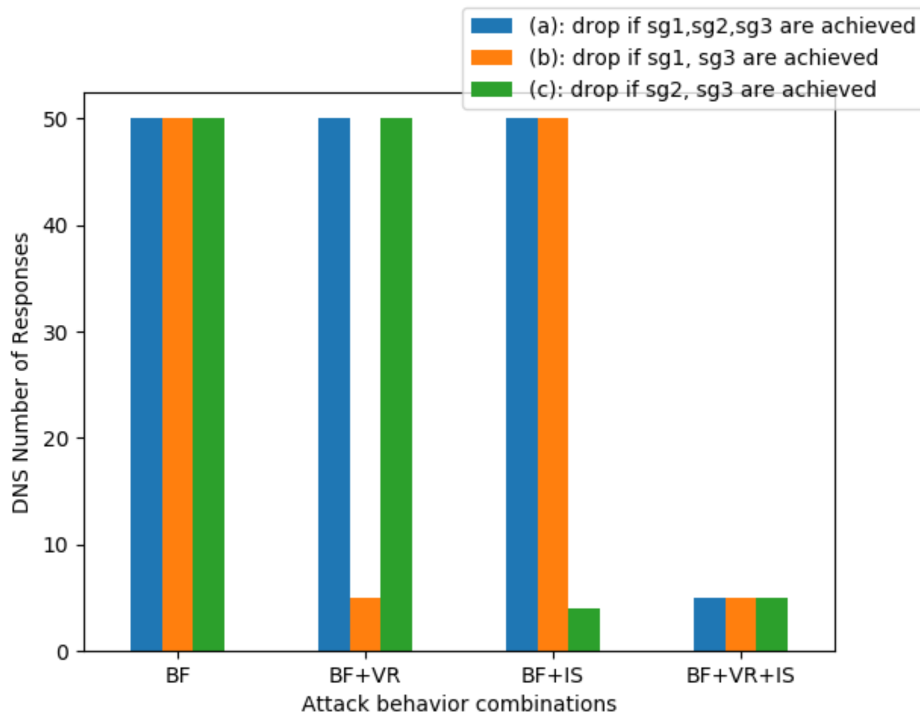


Figure 4.15: DNS number of responses for different behavior combinations

To assess its proper functioning, we also make the assumption that an attacker performs different behaviors as shown in Figure 4.15:

- BF: only a DNS brute force attack is performed (sg_3)
- BF+VR: a DNS brute force attack combined with the reception of a response that matches the server query ($sg_3 + sg_1$)
- BF+IS: a DNS brute force attack combined with a port scan ($sg_3 + sg_2$)
- BF+VR+IS: full malicious behaviors composed of a port scan, brute force and a valid response ($sg_1 + sg_3 + sg_2$)

The bars in Figure 4.15 represent the number of DNS responses. According to the different attacker behaviors and the MAT configurations, the number of packets to filter varies. For example, the configuration (a) only drop the traffic if and only if all attack stages have been reached (P6 in Figure 4.11). This results in dropping packets only when the most complex

behavior is monitored. In case of $BF + IS$, a token is added in P2 and P3 and so a transition to P5 occurs. Assuming configuration (b), packets are not dropped in that case as confirmed in Figure 4.15.

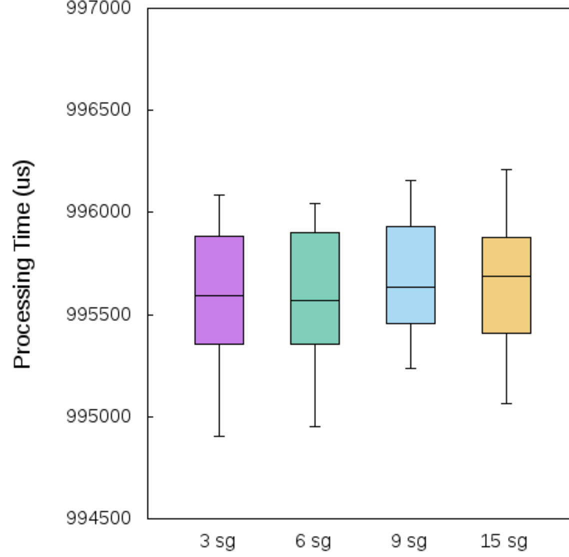


Figure 4.16: Switch processing time vs. Number of attack sub-goals

4.4.4.2 Switch processing time

Mininet and bmv2 switches are limited in providing an evaluation performance that will be representative of a deployment onto a real programmable switch. However, we can still compare different scenarios in our case. In this experiment, we evaluate the switch processing time when artificially increasing the number of attack sub-goals (3, 6, 9 and 15), given that one bit per goal is necessary to perform the lookup in the MAT. Therefore, 15 bits are used at most in our case. 100 DNS packets are generated and Figure 4.16 reports the processing time per packet.

We can see that the median value is approximately 995 ms for all scenarios. This shows that the switch processing time does not significantly increase with respect to the number of an attack's sub-goals to be detected in parallel.

In the next experiment, we evaluate the impact of the position of a MAT entry to be matched (1, 10, 20 or 32) on the switch processing time.

- Entry 1 : the matched entry in the table is the first one
- Entry 10: the matched entry in the table is the tenth
- Entry 20: the matched entry in the table is the twentieth
- Entry 32: the matched entry in the table is the thirty second

As highlighted in Figure 4.17, the switch average processing time is approximately 996 ms for all entries match positions. We can see that the position of an entry that is matched does not significantly impact the switch processing time. It is worth mentioning that the number of MAT entries is bounded by the number of Petri Net places, $|P|$.

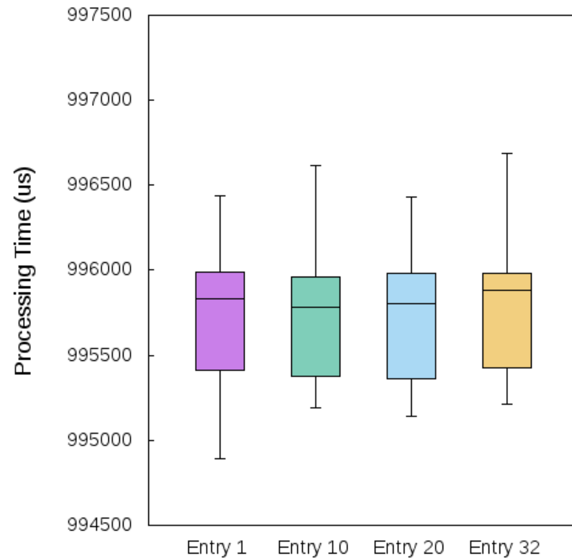


Figure 4.17: Switch processing time vs. table entry position

4.5 Summary

Because attacks are becoming more sophisticated and complex, we have to rely on suitable and flexible abstractions for attack detection. In this chapter, we introduced a Petri Net-based approach capable of detecting and mitigating composed attacks that pass through a set of steps in the data plane; we showed that a Petri Net abstraction could capture and synchronize a composed behavior. We demonstrated how this abstraction could be mapped to a data plane supportive primitive (*i.e.,* MAT). We described how the proposed approach could detect the new DNS multi-step cache poisoning attack. Our evaluation shows that the solution can detect the multi-step DNS attack, and the processing time is stable as the number of the attack steps increases (*i.e.,* sub-goals). Unlike existing solutions, our solution does not require the modification of the DNS protocol. Using the proposed approach, users can offload programs in the data plane to mitigate multi-step attacks and set different reaction levels based on the attacks progression and risk levels.

Chapter 5

Conclusion

SDN has simplified network management and enabled network programmability. Now users can define new protocols and implement new monitoring and security functions. OpenFlow was proposed to support communication between a controller and forwarding elements. However, the separation between the data plane and the control plane and the stateless nature of an OpenFlow-based data plane introduced much overhead in the network due to the communication between the two planes. Furthermore, OpenFlow is limited by pre-defined packet headers and actions, limiting the definition of custom functions. Users would also like to program customized in-network monitoring and attack detection functions running at line-rate. To this end, data plane programmability has been promoted to make the data plane more programmable and stateful. The P4 language has been proposed to define how packets are processed in a switch pipeline. New protocols and customized actions can be defined, and stateful structures named registers are used to maintain the state. In addition, P4 is based on a general architecture PISA that various hardware are expected to support. However, it has not been described how stateful processing can be supported based on a general abstraction hiding the implementation details.

Recent works have proposed solutions to provide a stateful data plane abstraction. Solutions that extend OpenFlow are limited by its primitives (i.e., including state explosion, limited definition of new headers and protocols) or depend on a controller to perform certain calculations. Alternatively, the solution is limited to a specific hardware target.

At the same time, attacks are more sophisticated. Some of them exploit vulnerabilities present in protocols used in the core of the Internet, such as TCP and DNS. An attacker can ignore the ECN congestion notification to give the illusion that the network is not congested. In this way, the sender maintains the same transmission rate during congestion. Another misbehaving is a TCP receiver that acknowledges packets before their reception, forcing the sender to transmit faster. These two misbehaving cause an unfair bandwidth share. Since TCP is an end-to-end protocol, the existing solutions propose to change the TCP implementation at end-hosts, which is complicated for a deployment at the Internet scale. In addition, it has been recently demonstrated that DNS cache poisoning can succeed if the attacker divides the attack into a set of steps. The attacker can perform a port scan to identify the port that has been used in the DNS query and then perform a brute force on the transaction ID. However, to mitigate the recent attack, the proposed solution has a negative impact on the collocated traffic.

5.1 Achievements

We have presented in the thesis manuscript our two main contributions; each of them defines a new solution to detect attacks and misbehaviors in a programmable data plane. For both contributions, we have proposed a model and a technique to map this model to a P4 data plane supportive primitives to exploit its benefits, including stateful and line-rate processing. We have evaluated the proposed solutions using different use cases.

First, in Chapter 3 we presented an abstraction based on an EFSM to model a protocol behavior to monitor in a P4 programmable data plane. We showed that the abstraction could describe stateful flow behaviors and model a misbehavior tracking. We presented a method to map the EFSM abstraction to P4 primitives, allowing the compilation of programs to many targets that support the PISA architecture. The main advantage is that the same abstraction can be used to describe different security functions. We demonstrated how the proposed approach could detect and mitigate two attacks exploiting the TCP protocol vulnerabilities named ECN abuse and Optimistic ACK attack directly within the network without any protocol modification. We developed an implementation of this solution through these two attack use cases in order to demonstrate the feasibility of the solution and evaluate the performance of the proposed method. Our results have shown that our solution can partially or fully restore the throughput loss caused by misbehaving end-hosts while ensuring a fair share of bandwidth among flows. Regarding the switch resource usage, our solution does not add significant processing time compared to a baseline when the switch only ensures the forwarding function.

Second, since attacks can be sophisticated to avoid their detection (*i.e.,* an attacker can divide the attack into a set of steps), in Chapter 4 we proposed a new method to detect multi-step attacks based on Petri Nets to model the attack. The proposed approach synchronizes the detection of a compound attack by combining a set of EFSM-based models to detect the steps separately. The Petri Net is in charge of synchronizing the detection between the set of EFSM models. To demonstrate the feasibility of our proposed approach, we have identified a recent attack, named the multi-step DNS cache poisoning attack, which was recently introduced as a new possible attack if the attacker divides the attack into several steps. We modeled the detection of the attack using a Petri net and performed an evaluation to demonstrate the effectiveness of our approach. We have shown that the attack steps can be detected by changing the configuration of the attack step composition, thus, demonstrating that the proposed approach can capture the dependencies between the attack steps which can be reconfigured within a single MAT representing the Petri Net. Moreover, the results show that the switch processing time does not significantly increase when the number of the attack steps increases.

To summarize, the EFSM-based abstraction limits state explosion and is simple enough along with Petri Net models to be mapped to P4 primitives. P4 is based on the PISA architecture that is supported by several hardware targets without worrying about hardware details (*i.e.,* a P4 compatible compiler is provided). Therefore, the proposed in-network security function can be deployed within the network to react against attacks in real-time without modifying the end-host implementations. Furthermore, to the best of our knowledge, we are the first to propose a solution to be deployed directly within the network for the ECN misuse, Optimistic ACK, and the DNS cache poisoning attacks without protocol modification.

5.2 Limitations, perspectives, and future work

This thesis opens up many perspectives and future work in the field of stateful processing abstraction and attack detection in a programmable data plane.

Automatic design and deployment across networks switches

In this thesis, we explored programmable data planes to enable stateful flow monitoring, allowing the offload of security functions to the data plane to track end-host behaviors and possible misbehaviors for line rate detection and reaction. We proposed an abstraction to model a behavior and to compose a set of behaviors. We presented how these models can be mapped to the P4 language-based supportive primitives. However, the modeling of protocols behavior and the mapping to data plane primitives is done manually (*i.e.,* manual design and deployment). As a perspective for our work, new tools and approaches can be developed to automate this mapping stage to the data plane (*i.e.,* to provide a systematic approach via a language that describes the EFSM and Petri Net models and a compiler to translate it to P4 primitives). This tool can also consider hardware target resources to deploy the solution in a distributed manner while respecting the available resources of each switch, and at the same time consider state migration mechanisms.

Complex applications

P4 have enabled the offloading of many function to the data plane. However, deploying application-level tasks on the data plane is challenging. With the emergence of web applications, identifying anomalies based on patterns present in the packet payload and not only based on packet headers fields would be helpful for network operators in monitoring application-level traffic. However, P4 has some restrictions and lacks primitives needed for payload processing, the P4 parser supports pre-defined headers field length parsing with one possible field having a variable length, so we cannot parse variable lengths header fields (*i.e.,* we can not process packet payload but only headers fields with fixed lengths). Another perspective for our work is to explore the possibilities of processing Layer 4 and above protocols, including payload processing to support complex application use cases in the data plane. Integrating payload processing into P4 would simplify the deployment of complex security functions into the flows data paths.

Additional evaluation

Stateful packet processing allowed the offload of different application to the data plane. However, the amount of per-flow data maintained in switches is limited by the target resources. Our evaluations are done on a software switch, the processing time and overhead will be significantly improved on a hardware switch compared to a software switch. As a future work, exploring an evaluation on a hardware switch can be considered. Since that our evaluations were done using synthetic traffic because there are no real traffic traces available from real scenarios, a possible extension will be an evaluation using real traffic traces to see how big these misbehaviors are.

Publications

The contributions presented in this thesis have led to several publications:

- **Defeating Protocol Abuse with P4: Application to Explicit Congestion Notification [100]** published in the IFIP Networking Conference (Networking 2020) presented in Chapter 3 Section 3.3
- **Mitigating TCP Protocol Misuse With Programmable Data Planes [101]** published in IEEE Transactions on Network and Service Management (TNSM 2021), presented in Chapter 3
- **Detecting Multi-Step Attacks: A Modular Approach for Programmable Data Plane [102]** published in IEEE/IFIP Network Operations and Management Symposium (NOMS 2022), presented in Chapter 4

Glossary

API: Application Programming Interface
CLI: Command Line Interface
DNS: Domain Name System
DNSSEC: Domain Name System Security Extensions
DDoS: Distributed Denial-of-Service
EFSM: Extended Finite State Machine
ECN: Explicit Congestion Notification
FSM: Finite State Machine
IP: Internet Protocol
NFV: Network Function Virtualization
OSI: Open System Interconnection
PKI: Public Key Infrastructure
P4: Programming Protocol Independent Packet Processors
PISA: Protocol Independent Switch Architecture
RMT: Reconfigurable Match Table
RED: Random Early Detection
SDN: Software Defined Network
TCP: Transport Control Protocol

Bibliography

[1] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, pp. 80–86, 1997.

[2] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, p. 8798, 2014.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, 2008.

[4] R. Neres Carvalho, J. Luiz Bordim, and E. Adilio Pelinson Alchieri, "Entropy-based dos attack identification in sdn," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 627–634.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, 2014.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of ACM SIGCOMM*, 2013.

[7] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "Qpipe: quantiles sketch fully in the data plane," in *CoNEXT 19: Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 12 2019, p. 7 pages.

[8] *P4 Language Consortium. Portable Switch Architecture (PSA)*, 2021. [Online]. Available: https://p4.org/p4-spec/docs/PSA.html

[9] J. Gomez, E. Kfoury, J. Crichigno, and G. Srivastava, "A survey on tcp enhancements using p4-programmable devices," *Computer Networks*, 05 2022.

[10] *P4 Language Consortium. 2018. Behavioral Model (BMv2)*, 2018. [Online]. Available: https://github.com/p4lang/behavioral-model

[11] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014, pp. 61–66.

[12] C. Sun, J. Bi, H. Chen, H. Hu, Z. Zheng, S. Zhu, and C. Wu, "Sdpa: Toward a stateful data plane in software-defined networking," *IEEE/ACM Transactions on Networking*, 2017.

[13] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, “Openstate: Programming platform-independent stateful openflow applications inside the switch,” *ACM SIGCOMM Comput. Commun. Rev.*, 2014.

[14] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano, “Flowblaze: Stateful packet processing in hardware,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[15] P. Chaignon, K. Lazri, J. François, T. Delmas, and O. Festor, “Oko: Extending open vswitch with stateful filters,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3185467.3185496

[16] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, and S. Pontarelli, “Xtra: Towards portable transport layer functions,” *IEEE Transactions on Network and Service Management*, pp. 1507–1521, 2019.

[17] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, “Packet transactions: High-level programming for line-rate switches,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1528. [Online]. Available: https://doi.org/10.1145/2934872.2934900

[18] E. Kaljic, A. Maric, P. Njemcevic, and M. Hadzialic, “A survey on data plane flexibility and programmability in software-defined networking,” *IEEE Access*, vol. 7, pp. 47 804–47 840, 2019.

[19] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, “Snap: Stateful network-wide abstractions for packet processing,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 2943. [Online]. Available: https://doi.org/10.1145/2934872.2934892

[20] S. Luo, H. Yu, and L. Vanbever, “Swing state: Consistent updates for stateful and programmable data planes,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 115121. [Online]. Available: https://doi.org/10.1145/3050220.3050233

[21] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, and G. Bianchi, “Lodge: Local decisions on global states in programmable data planes,” in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 257–261.

[22] S. German, B. Marco, T. Angelo, G. Paolo, B. Andrea, and B. Giuseppe, “Local decisions on replicated states (loader) in programmable dataplanes: Programming abstraction and experimental evaluation,” *Computer Networks*, vol. 184, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128620312597

[23] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, “Hula: Scalable load balancing using programmable data planes,” in *ACM Symposium on SDN Research (SOSR)*, 2016.

[24] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[25] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," in *ACM Symposium on SDN Research (SOSR)*, 2017.

[26] C. Cascone, D. Sanvito, L. Pollini, A. Capone, and B. Sanso, "Fast failure detection and recovery in sdn with stateful data plane," *International Journal of Network Management*, vol. 27, 11 2016.

[27] W. Wang, P. Tammana, A. Chen, and T. S. E. Ng, "Grasp the root causes in the data plane: Diagnosing latency problems with spidermon," *Proceedings of the Symposium on SDN Research*, 2020.

[28] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band network telemetry: A survey," *Computer Networks*, vol. 186, p. 107763, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128620313396

[29] F. Cugini, P. Gunning, F. Paolucci, P. Castoldi, and A. Lord, "P4 in-band telemetry (int) for latency-aware vnf in metro networks," in *2019 Optical Fiber Communications Conference and Exhibition (OFC)*, 2019, pp. 1–3.

[30] C. Kim, A. Sivaraman, N. P. K. Katta, A. Bas, A. A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," 2015.

[31] J. Geng, J. Yan, and Y. Zhang, "P4qcn: Congestion control using p4-capable device in data center networks," *Electronics*, vol. 8, no. 3, 2019. [Online]. Available: https://www.mdpi.com/2079-9292/8/3/280

[32] C. Chen, H.-C. Fang, and M. S. Iqbal, "Qostcp: Provide consistent rate guarantees to tcp flows in software defined networks," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–6.

[33] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, "Fine-grained queue measurement in the data plane," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1529. [Online]. Available: https://doi.org/10.1145/3359989.3365408

[34] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 101–114.

[35] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 576–590.

[36] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno, "Towards a unified in-network ddos detection and mitigation strategy," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, 2020, pp. 218–226.

[37] G. Simsek, H. Bostan, A. Sarica, E. Sarikaya, A. Keles, P. Angin, H. Alemdar, and E. Onur, *DroPPPP: A P4 Approach to Mitigating DoS Attacks in SDN*, 01 2020, pp. 55–66.

[38] T.-Y. Lin, J.-P. Wu, P.-H. Hung, C.-H. Shao, Y.-T. Wang, Y.-Z. Cai, and M.-H. Tsai, "Mitigating syn flooding attack and arp spoofing in sdn data plane," in *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2020, pp. 114–119.

[39] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 164176. [Online]. Available: https://doi.org/10.1145/3050220.3063772

[40] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3185467.3185476

[41] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Proceedings of NDSS*, 2020.

[42] C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 19–27.

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

[44] G. Grigoryan and Y. Liu, "Lamp: Prompt layer 7 attack mitigation with programmable data planes," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, 2018, pp. 1–4.

[45] L. A. Q. González, L. Castanheira, J. A. Marques, A. Schaeffer-Filho, and L. P. Gaspary, "Bungee: An adaptive pushback mechanism for ddos detection and mitigation in p4 data planes," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 393–401.

[46] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson, "Robust congestion signaling," in *International Conference on Network Protocols (ICNP 2001)*, 2001.

[47] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "Tcp congestion control with a misbehaving receiver," *ACM SIGCOMM Comput. Commun. Rev.*, 1999.

[48] V. S. Alagar and K. Periyasamy, *Specification of Software Systems*. Springer Publishing Company, Incorporated, 2011.

[49] Kwang-Ting Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *30th ACM/IEEE Design Automation Conference*, 1993.

[50] K. El-Fakih, N. Yevtushenko, M. Bozga, and S. Bensalem, “Distinguishing extended finite state machine configurations using predicate abstraction,” *Journal of Software Engineering Research and Development*, vol. 4, pp. 1–26, 2016.

[51] T. Mallory and A. Kullberg, “Incremental updating of the internet checksum,” Internet Requests for Comments, RFC, 1990.

[52] K. Ramakrishnan, S. Floyd, and D. Black, “The addition of explicit congestion notification (ecn) to ip,” *RFC 3168*, 2001.

[53] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, “I know what your packet did last hop: Using packet histories to troubleshoot networks,” in *USENIX NSDI*, 2014, pp. 71–85.

[54] T. P. A. W. Group, “In-band Network Telemetry (INT) data plane specification,” June 2020. [Online]. Available: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf

[55] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” *IEEE/ACM Transactions on Networking*, pp. 397–413, 1993.

[56] *Traffic control*, 2019. [Online]. Available: https://github.com/PIFO-TM/ns3-bmv2/tree/master/traffic-control

[57] W. Bai, K. Chen, L. Chen, C. Kim, and H. Wu, “Enabling ecn over generic packet scheduling,” ser. CoNEXT ’16, 2016, p. 191204. [Online]. Available: https://doi.org/10.1145/2999572.2999575

[58] B. B. et al, “Recommendations on queue management and congestion avoidance in the internet,” *RFC 2309*, 1998.

[59] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, “Reproducible network experiments using container-based emulation,” in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. ACM CoNEXT 12.

[60] “p4app,” https://github.com/p4lang/p4app, accessed: 2020-01-09.

[61] *P4 Language Consortium. P4-16 Language Specification*, 2018. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html

[62] A. Dhamdhere and C. Dovrolis, “Open issues in router buffer sizing,” *SIGCOMM Comput. Commun. Rev.*, p. 8792, 2006.

[63] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *Proceedings of ACM SIGCOMM*, 2010.

[64] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang, “Tuning ecn for data center networks,” in *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2012.

[65] E. B. M. Allman, V. Paxson, “Tcp congestion control,” Internet Requests for Comments, Network Working Group, RFC, 2009.

[66] R. Sherwood, B. Bhattacharjee, and R. Braud, "Misbehaving tcp receivers can cause internet-wide congestion collapse," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[67] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *ACM SIGCOMM Conference*, 2011.

[68] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in tcp congestion control using a model-guided approach," in *Proceedings of the Applied Networking Research Workshop*. ACM, 2018.

[69] V. Ramesh, "misbehaving-receiver," 2016. [Online]. Available: https://github.com/rameshvarun/misbehaving-receiver

[70] C.-H. He, B. Y. Chang, S. Chakraborty, C. Chen, and L. C. Wang, "A zero flow entry expiration timeout p4 switch," in *Symposium on SDN Research (SOSR)*. ACM, 2018.

[71] R. Kundel, F. Siegmund, J. Blendin, A. Rizk, and B. Koldehofe, "P4sta: High performance packet timestamping with programmable packet processors," in *Network Operations and Management Symposium (NOMS)*. IEEE, 2020.

[72] R. Bush and R. Elz, "Serial number arithmetic," *RFC 1982*, 1996.

[73] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, "Tcp extensions for high performance," *RFC 7323*, 2014.

[74] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *Proceedings of the Symposium on SDN Research*, ser. SOSR 17. Association for Computing Machinery.

[75] J. P. McDermott, "Attack net penetration testing," in *Proceedings of the 2000 Workshop on New Security Paradigms*, ser. NSPW '00. New York, NY, USA: Association for Computing Machinery, 2001, p. 1521. [Online]. Available: https://doi.org/10.1145/366173.366183

[76] Y. Musashi, M. Kumagai, S. Kubota, and K. Sugitani, "Detection of kaminsky dns cache poisoning attack," in *2011 4th International Conference on Intelligent Networks and Intelligent Systems*, 2011, pp. 121–124.

[77] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, "Dns cache poisoning attack reloaded: Revolutions with side channels," in *SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. ACM, 2020.

[78] E. Smith, "Carl adam petri," in *Springer Berlin Heidelberg*, 2015.

[79] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, no. 3, p. 223252, sep 1977. [Online]. Available: https://doi.org/10.1145/356698.356702

[80] I. Nai Fovino, M. Masera, and A. De Cian, "Integrating cyber attacks within fault trees," *Reliability Engineering & System Safety*, vol. 94, no. 9, pp. 1394–1402, 2009, eSREL 2007, the 18th European Safety and Reliability Conference. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0951832009000337

[81] T. M. Chen, J. Sánchez-Aarnoutse, and J. Buford, "Petri net modeling of cyber-physical attacks on smart grid," *IEEE Transactions on Smart Grid*, vol. 2, pp. 741–749, 2011.

[82] D. P. Mirembe and M. Muyeba, "Threat modeling revisited: Improving expressiveness of attack," in *2008 Second UKSIM European Symposium on Computer Modeling and Simulation*, 2008, pp. 93–98.

[83] K. N. Junejo and J. Goh, "Behaviour-based attack detection and classification in cyber physical systems using machine learning," ser. CPSS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2899015.2899016

[84] I. Al-Azzoni, D. G. Down, and R. Khedri, "Modeling and verification of cryptographic protocols using coloured petri nets and design/cpn," *Nordic J. of Computing*, vol. 12, no. 3, jun 2005.

[85] X. Li and D. Li, "A network attack model based on colored petri net," *J. Networks*, vol. 9, pp. 1883–1891, 2014.

[86] R. Wu, W. Li, and H. Huang, "An attack modeling based on hierarchical colored petri nets," in *2008 International Conference on Computer and Electrical Engineering*, 2008, pp. 918–921.

[87] I. S. I. 15909, "High-level petri nets," October 2000.

[88] P. Mockapetris, "Domain names - implementation and specification," Internet Requests for Comments, Network Working Group, RFC, 1987.

[89] L. Yuan, K. Kant, P. Mohapatra, and C.-n. Chuah, "Dox: A peer-to-peer antidote for dns cache poisoning attacks," in *2006 IEEE International Conference on Communications*, vol. 5, 2006, pp. 2345–2350.

[90] W. Lian, E. Rescorla, H. Shacham, and S. Savage, "Measuring the practical impact of DNSSEC deployment," in *Security Symposium*. USENIX Association, 2013.

[91] H. Shulman and M. Waidner, "One key to sign them all considered vulnerable: Evaluation of DNSSEC in the internet," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 131–144. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/shulman

[92] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee, "Increased dns forgery resistance through 0x20-bit encoding: Security via leet queries," ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 211222. [Online]. Available: https://doi.org/10.1145/1455770.1455798

[93] A. Klein, H. Shulman, and M. Waidner, "Internet-wide study of dns cache injections," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[94] R. v. M. A. Hubert, "Measures for making dns more resilient against forged answers," Internet Requests for Comments, IETF, RFC, 2009.

[95] A. Klein, "Cross layer attacks and how to use them (for DNS cache poisoning, device tracking and more)," *CoRR*, vol. abs/2012.07432, 2020. [Online]. Available: https://arxiv.org/abs/2012.07432

[96] F. Alharbi, J. Chang, Y. Zhou, F. Qian, Z. Qian, and N. Abu-Ghazaleh, “Collaborative client-side dns cache poisoning attack,” in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1153–1161.

[97] H.-M. Sun, W.-H. Chang, S.-Y. Chang, and Y.-H. Lin, “Dependns: Dependable mechanism against dns cache poisoning,” in *Cryptology and Network Security*, J. A. Garay, A. Miyaji, and A. Otsuka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 174–188.

[98] J. Mohan, S. Puranik, and K. Chandrasekaran, “Reducing dns cache poisoning attacks,” in *2015 International Conference on Advanced Computing and Communication Systems*, 2015, pp. 1–6.

[99] N. Alexiou, S. Basagiannis, P. Katsaros, T. Dashpande, and S. A. Smolka, “Formal analysis of the kaminsky dns cache-poisoning attack using probabilistic model checking,” in *IEEE 12th International Symposium on High Assurance Systems Engineering*, 2010.

[100] A. Laraba, J. François, I. Chrisment, S. R. Chowdhury, and R. Boutaba, “Defeating protocol abuse with p4: Application to explicit congestion notification,” in *IFIP Networking 2020*, 2020.

[101] A. Laraba, J. François, S. R. Chowdhury, I. Chrisment, and R. Boutaba, “Mitigating tcp protocol misuse with programmable data planes,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 760–774, 2021.

[102] A. Laraba, J. Franx00E7;ois, I. Chrisment, S. R. Chowdhury, and R. Boutaba, “Detecting multi-step attacks: A modular approach for programmable data plane,” in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–9.

[103] J. Kim, “Meta4: Analyzing Internet Traffic by Domain Name in the Data Plane,” Princeton University, Tech. Rep., 2021.

Appendix

Appendix A

Résumé en français

A.1 Introduction

Ces dernières années, les réseaux sont devenus complexes et difficiles à gérer, des travaux ont donc été réalisés pour rendre les réseaux plus programmables et plus simples à monitorer. La technologie SDN (Software Defined Network) a ainsi été proposée ; elle sépare le plan de contrôle, qui décide comment traiter le trafic réseau, du plan de données, qui achemine le trafic. En premier lieu, elle a été introduite pour permettre la programmabilité du plan de contrôle, où la logique de contrôle et l'intelligence du réseau sont centralisées. Alors que le plan de données est conçu avec des capacités limitées (c'est-à-dire des actions fixes et prédéfinies) afin de traiter les paquets à des débits élevés, le plan de contrôle effectue des opérations complexes. Le contrôleur communique avec les éléments du plan de données à l'aide d'une API appelée OpenFlow ; le plan de données basé sur OpenFlow n'a pas d'état persistant dans les commutateurs. Par conséquent, les contrôleurs envoient des règles d'acheminement aux éléments du plan de données en fonction du comportement du réseau, ce qui rajoute plus de délai et de charge. Le contrôle central et le plan de données sans état introduisent des problèmes de charge et d'évolutivité en raison de l'implication explicite et fréquente du contrôleur sur les éléments du réseau. La nature sans état du plan de données basé sur OpenFlow et sa limitation en termes d'expressivité ont suscité des efforts pour rendre le plan de données plus programmable et à états afin de limiter l'interaction explicite avec le plan de contrôle.

Le langage de programmation P4 a été proposé pour programmer les éléments du réseau et considérer des fonctions de traitement de paquets définies par l'utilisateur. P4 supporte non seulement les actions standards bien connues comme celles des commutateurs traditionnels, mais aussi des actions entièrement personnalisées. Il peut prendre en charge le traitement avec états reposant sur des structures spécifiques, limitant ainsi l'interaction avec le plan de contrôle et permettant le déchargement d'un grand nombre de fonctions sur le plan de données, par exemple, la détection d'attaques.

Entre-temps, notre dépendance à l'égard des services offerts par l'Internet s'accroît, et leur utilisation massive a généré un trafic important. Cette augmentation significative du trafic Internet et des technologies émergentes qui utilisent différents protocoles de réseau est associée à un risque d'attaques. En conséquence, des attaques aux motivations diverses se sont développées et sont devenues plus sophistiquées et plus coûteuses. Elles ciblent principalement les protocoles utilisés au niveau de l'Internet, par exemple, TCP. Elles peuvent donc causer des dommages

importants sur le fonctionnement global des systèmes et des services et rendre le réseau indisponible pour les utilisateurs légitimes. Il est donc important d'identifier les attaques à un stade préliminaire et de réagir en temps réel pour les éliminer ou réduire leurs impacts. Lorsqu'il s'agit d'attaques sophistiquées et multi-étapes, nous devons identifier quelle étape a été réalisée et dans quel ordre les étapes de l'attaque ont été effectuées. En outre, lorsqu'une attaque composée est identifiée, il est important de déterminer la gravité de l'attaque et de définir la réaction appropriée de manière dynamique et en temps réel.

Dans cette thèse, nos contributions portent sur la détection des attaques dans un plan de données programmable. Dans la section 2, nous introduisons la problématique et présentons l'impact de ces attaques ciblant les vulnérabilités des protocoles. Dans la section 3, nous présentons une solution pour détecter et remédier à ces attaques, en particulier celles qui ciblent le protocole TCP. Dans la section 4, nous décrivons une approche pour détecter les attaques composées. Enfin, nous terminons par une conclusion générale et les perspectives de recherche.

A.2 Problématique

Les plans de données programmables ont permis de décharger certaines fonctions vers les éléments de réseau, par exemple, les fonctions de surveillance et de sécurité. Cependant, le déchargement d'applications à états vers le plan de données reste un défi. Malgré les nouvelles primitives et fonctionnalités proposées par le plan de données programmable, il n'y a pas de support intuitif pour un traitement avec états. Une abstraction avec états pour surveiller et capturer le comportement des flux dans le plan de données est toujours une partie manquante.

Le langage proposé pour la programmation du plan de données est limité dans la description d'une fonction qui surveille l'état des flux. Un modèle abstrait commun pour un traitement avec états dans le plan de données serait nécessaire pour prendre en charge différentes applications de suivi de flux sur différentes cibles du plan de données (commutateur matériel ou logiciel). Pour relever ce défi, des efforts ont été consacrés à la réalisation des abstractions à états dans le plan de données basées sur OpenFlow. Ces solutions étendent OpenFlow mais sont limitées en termes d'actions supportées, ce qui nécessite une implication partielle du contrôleur. D'autres solutions proposent des abstractions avec états dans le plan de données entièrement programmable, mais sont toujours spécifiques aux matériels cibles. Ainsi, les abstractions existantes avec états sont limitées par la nature sans état d'OpenFlow ou restent spécifiques à un type de matériel. Un plan de données programmable devrait permettre aux utilisateurs de déployer facilement de nouvelles solutions. Par ailleurs, une abstraction avec états devrait reposer sur un langage commun qui pourrait être compilé sur différentes cibles au lieu de s'appuyer sur des détails spécifiques d'implémentation matérielle.

En parallèle, la communauté a fait plusieurs efforts autour de la sécurité des réseaux et a pris conscience des risques liés aux attaques visant des protocoles connus. Cependant, les attaques ciblent les protocoles implémentés sur les hôtes finaux. Les solutions proposées sont donc déployées sur ces hôtes, avec des difficultés lorsqu'il s'agit de les mettre en œuvre car elles nécessitent des changements d'implémentation, ce qui reste une opération complexe. Une solution plus rapide consisterait à déployer une solution d'atténuation des attaques au sein du réseau, de sorte que l'attaque soit détectée et atténuée en temps réel, le plus tôt possible, au sein du réseau, avant d'atteindre les hôtes finaux.

A.3 Abstraction pour détecter les attaques dans un plan de données programmable

Les protocoles de réseau peuvent faire l'objet d'attaques qui exploitent leurs vulnérabilités. Par exemple, pour le protocole TCP, un attaquant peut ignorer une notification de congestion pour maintenir un taux de transmission élevé, amplifiant ainsi cette congestion. Un autre exemple est celui d'un attaquant qui donne l'illusion que le réseau n'est pas congestionné en envoyant des accusés de réception avant que les segments ne soient reçus, ce qui oblige l'émetteur à envoyer plus de données ; ce mauvais comportement est appelé l'attaque Optimisitic Ack. Pour détecter ces attaques, les solutions proposées consistent à modifier la spécification du protocole TCP, ce qui est compliqué d'un point de vue déploiement. Pour limiter l'exploitation de ces vulnérabilités, une approche est d'atténuer ce type d'attaque sans modifier les implémentations des protocoles pour un déploiement plus rapide.

En parallèle, le langage de programmation P4 a permis de supporter des opérations personnalisées avec des débits élevés dans le plan de données. Ainsi, il est possible de décharger une fonction d'atténuation des attaques sur le chemin du trafic, ce qui permet de détecter et de réagir aux attaques pendant que le trafic d'attaque traverse le réseau et sans nécessiter une modification d'implémentation du protocole sur les hôtes finaux. Utiliser une abstraction à états expressive, permettant le suivi de comportements complexes tout en étant suffisamment générale pour supporter différentes applications, indépendamment du matériel et des technologies sous-jacentes, peut permettre une détection efficace des attaques dans le plan des données.

Nous proposons une abstraction reposant sur des Machines à États Étendues (EFSM) qui peuvent être traduites en primitives programmables et simultanément modéliser un traitement à états et capturer des comportements complexes. La correspondance de cette abstraction vers des primitives permettrait de prendre en charge le traitement des flux dans le réseau. En outre, une approche associant une EFSM et le langage P4 est suffisamment générale pour supporter de nombreuses applications de sécurité.

Dans notre contribution, une EFSM représente un comportement d'un protocole. La première étape consiste à dériver une EFSM à partir de la spécification de protocole. Un protocole peut avoir plusieurs composants, règles et étapes qui interagissent ; par conséquent, plusieurs états peuvent être produits dans une EFSM. Un ensemble d'états pouvant être fusionné, le nombre d'états à surveiller est limité aux états nécessaires pour suivre le comportement correct du protocole. En supposant que les protocoles ont une séquence de messages définis, nous étendons le modèle EFSM avec les mauvais comportements possibles. L'abstraction EFSM représente ainsi la spécification de la fonction de sécurité pour détecter un éventuel mauvais comportement.

L'approche proposée donne la possibilité aux opérateurs de réseau de configurer les commutateurs pour qu'ils prennent différentes mesures lorsqu'un mauvais comportement est détecté, comme le rejet ou le réacheminement des paquets, la génération d'une alerte, l'envoi d'une copie des paquets à un serveur distant pour une inspection plus approfondie, la définition des priorités de la file d'attente ou l'application des actions correctives telles que la modification des champs d'entête des paquets. Une fois la machine à états du protocole définie, l'étape suivante consiste à réaliser la correspondance vers un ensemble de primitives P4 supportées par le plan de données programmable. Lorsque le programme P4 est compilé, le trafic est suivi en temps réel ; chaque

état de flux et des informations supplémentaires sont maintenus dans le plan de données pour la détection d'éventuels mauvais comportements (c'est-à-dire que chaque flux correspond à une instance EFSM).

Nous avons implémenté deux applications pouvant être délocalisées vers un commutateur programmable. La première application implémente la détection de l'utilisation abusive du mécanisme ECN (Explicit Congestion Notification) de TCP. Elle détecte le placement ou non des drapeaux TCP et les corrige dans le cas d'une attaque. Pour détecter l'attaque Optimistic Ack, la deuxième application implémente un limiteur d'acquittement optimiste ; elle vérifie si un acquittement TCP a été reçu pour un segment envoyé.

Nous avons effectué des évaluations pour montrer l'impact de ces attaques et la manière dont notre solution peut les détecter et y réagir. Nous avons aussi évalué le coût de notre solution en termes de temps de traitement à l'intérieur d'un commutateur.

A.4 Méthode pour détecter les attaques composées dans un plan de données programmable

Les attaques deviennent de plus en plus sophistiquées, avec par exemple un ensemble d'étapes permettant d’atteindre efficacement un objectif final. Il est parfois difficile de reconnaître le niveau de risque de l'attaque, uniquement en composant l'ensemble de ses étapes.

Pour détecter ce type d'attaque, nous proposons une abstraction reposant sur un réseau de Petri pour synchroniser l'accomplissement des attaques composées, et une méthode pour dériver cette abstraction vers des primitives supportées par le plan de données programmable, qui offre des avantages en termes de traitement des paquets à des débits élevés.

Notre approche vise à modéliser les relations entre les sous-objectifs d'une attaque et leurs combinaisons. Elle repose sur une modélisation multi-couches. Une première couche est constituée d'un ensemble de modules basés sur des EFSM ; ces modules se chargent de détecter les étapes d'une attaque (module de détection). Une deuxième couche représente un module de décision qui est synchronisé avec les modules de la première couche. Ce module définit les décisions possibles et les niveaux d'alerte selon les comportements détectés par la première couche. En effet, l'utilisateur peut composer différents comportements d'une attaque et définir des niveaux de réaction en fonction de ses propres besoins. Notre contribution est définie pour être déployée sur des commutateurs P4. Nous utilisons principalement des *Match-Actions table (MAT)* qui font correspondre des clés de recherche avec des champs de paquets ou métadonnées. Une fois la correspondance établie, des *actions* prédéfinies par le commutateur ou personnalisées sont appliquées. Les *métadonnées* sont des données générées ou calculées durant l’exécution d’un programme P4. Pour maintenir les données dans le commutateur à travers les paquets du même flux, nous utilisons des *registres* qui sont des structures à états écrites et lues par les actions P4.

Nous supposons qu'une attaque multi-étapes est composée d'un ensemble de sous-objectifs et de stades d'attaque. Chaque stade représente la composition d'un ensemble de sous-objectifs pour atteindre un objectif intermédiaire. L'ensemble des sous-objectifs et des stades sont combinés pour atteindre l'objectif final. Dans un réseau de Petri, les places représentent les sous objectives d'une attaque. La progression de l'attaque est représentée par la présence des jetons sur les places ; les jetons sont maintenus et rendus persistants une fois placés sur une place.

Nous avons implémenté une solution pour détecter les attaques d'empoisonnement de cache de DNS. Pour empoisonner le cache DNS, un attaquant commence par envoyer une requête DNS au serveur DNS victime, qui est en charge de la résolution récursive en demandant à d'autres serveurs DNS sur Internet. L'objectif de l'attaquant est de répondre le premier avec une réponse valide à l'une de ces requêtes. Pour être valide, une réponse doit correspondre au domaine initialement demandé, mais aussi au port source et à l'identifiant de la requête utilisée par le serveur DNS. Dans le passé, le port source n'était pas randomisé et il était possible de dériver l'identifiant par force brute. En raison de cette faille, le port source UDP est désormais randomisé. Un attaquant peut contourner ce problème en effectuant un port scan UDP et en attendant que le serveur DNS réponde par des messages 'ICMP unreachable' pour tous les ports fermés sauf celui ouvert. Grâce à cette technique, l'attaquant peut d'abord déterminer le port source et, par la suite, l'identifiant de la requête par force brute. Nous décomposons l'attaque en 3 sous-objectifs : la réception d'une réponse valide qui correspond à la requête sortante du serveur (nom de domaine et identifiant valides) et le port utilisé dans la réponse est identifié comme ouvert ; une tentative de port scan avec UDP et un force brute de l'identifiant de la transaction DNS. Les sous-objectifs sont combinés pour détecter l'attaque.

Nous avons évalué le temps de traitement du commutateur en augmentant le nombre de sous-objectives d'une attaque.

A.5 Conclusion

Cette thèse ouvre de nombreuses perspectives et des travaux futurs dans le domaine du traitement avec états et de la détection d'attaques dans un plan de données programmable.

Dans cette thèse, nous avons exploré les plans de données programmables pour assurer la surveillance avec états des flux. Les fonctions de sécurité sont déchargées vers le plan de données programmable afin de suivre les comportements des flux et les éventuels mauvais comportements pour une détection et réaction en temps réel. Nous avons proposé une abstraction pour modéliser un comportement composé ou non d'attaques et des techniques pour traduire ces modèles vers des primitives reposant sur le langage P4. Cependant, la modélisation du comportement des protocoles et la mise en correspondance avec les primitives du plan de données sont effectuées manuellement (c'est-à-dire, conception et déploiement manuels).

Dans la perspective de notre travail, de nouveaux outils et approches peuvent être développés pour automatiser cette étape de correspondance vers le plan de données, c'est-à-dire fournir une approche systématique via un langage qui décrit l'EFSM et le réseau de Petri et via un compilateur pour les traduire en primitives P4. Cet outil peut également prendre en compte les ressources matérielles cibles pour déployer la solution de manière distribuée, tout en respectant les ressources disponibles de chaque commutateur, et en même temps, considérer les mécanismes de migration d'état.

P4 a permis de décharger de nombreuses fonctions sur le plan des données. Cependant, le déploiement des fonctions de la couche application sur le plan des données est un défi. Avec l'émergence des applications web, l'identification d'anomalies exploitant la charge utile des paquets et pas seulement les champs des en-têtes de paquets serait utile aux opérateurs de réseau pour surveiller le trafic au niveau applicatif. Cependant, P4 a quelques restrictions et manque de primitives nécessaires pour le traitement de la charge utile. Une autre perspective pour notre

travail consiste à explorer les possibilités de traitement des protocoles de la couche transport et des couches supérieures, y compris le traitement de la charge utile pour prendre en charge des applications complexes dans le plan de données. L'intégration du traitement des charges utiles dans P4 simplifierait le déploiement des fonctions de sécurité complexes directement dans le réseau.

Appendix B

Optimistic ACK attack detection P4 implementation details

As invoked in Section 3.4.2 to detect the Optimistic ACK attack, we need to track the TCP expected acknowledgment numbers. Figure B.1 represents the declaration of the register $Register1_seq_state$ used for this purpose. Each entry is 32 bit length and has 10000 entries. As explained in section 3.4.2 the expected ACKno is calculated based on the $IP.len$, $IP.ihl$ and $TCP.dataOffset$ fields. We associate these fields with a set of P4 metadata, $meta_ihl, meta_dataoffset, meta_totallen$. Figure B.2 shows how $payload_size$ is calculated as explained in Section 3.4.2 using fields metadata. Figure B.3 shows the P4 program to calculate the $expected_ackNo$; a metadata $hashcorrectACK1$ is used to maintain the $expected_ackNo$. The calculation is based on the sequence number $meta_seq$ and the calculated payload length in Figure B.2 (*i.e.,* the $expected_ackNo$ equals to the sequence number plus the payload length of the packet).

To track TCP flows in the switch, a hash flow identifier is necessary; we use the 5-tuple combined with the $expected_ackNo$. The output of the hash function is maintained in a metadata $hashFlow$ (Figure B.4).

For each received packet in the direction $sender - to - receiver$ (*i.e.,* $(srcAddr < dstAddr)$), the $expected_ackNo$ is calculated in the $hashcorrectACK1$ metadata and maintained in $Register1_seq_state$ (as shown in Figure B.5). The hash flow identifier used is the $hashFlow$ calculated in Figure B.4.

In the reverse direction, from $receiver - to - sender$ (*i.e.,*$(srcAddr > dstAddr)$), the hash flow identifier is maintained in a metadata $hashFlowDirVerify$; this metadata is calculated based on the inverted addresses and ports in the 5-tuple combined with the $tcp.ackNo$ (as shown in Figure B.6). This allows us to verify if the received acknowledgement number is the one we calculated in the first direction ($sender - to - receiver$) which corresponds to the sequence number plus the payload length.

Hence, as shown in Figure B.7 in the reverse direction $receiver - to - sender$, to retrieve the previously calculated $expected_ackNo$, we read the $Register1_seq_state$ using the hash flow identifier maintained in a metadata $hashFlowDirVerify$ (*i.e.,* based on the reverse IP addresses and ports, calculated in Figure B.6). The read value is maintained in a metadata $meta_ack_verify$. The Optimistic ACK attack is detected if $meta_ack_verify == 0$ (*i.e.,*

the expected acknowledgment number was not calculated and maintained in the register). Thus, the acknowledgment number is considered as not having already been sent

```
register<bit<32>>(10000) Register1_seq_state;
```

Figure B.1: expected_ackno register declaration

```
meta.ingress_metadata.multi_4 = 4;


meta.ingress_metadata.meta_add1 = (meta.ingress_metadata.meta_ihl +
meta.ingress_metadata.meta_dataoffset);
meta.ingress_metadata.meta_add2 =  (int<16>)(bit<16>) (bit<4>) meta.ingress_metadata.meta_add1 *
meta.ingress_metadata.multi_4;


meta.ingress_metadata.meta_len = (bit<16>) meta.ingress_metadata.meta_totallen - (bit<16>)(int<16>)
meta.ingress_metadata.meta_add2;
```

Figure B.2: *payloadsize* calculation

```
       meta.ingress_metadata.meta_hashcorrectACK1 = (bit<32>) meta.ingress_metadata.meta_seq +
(bit<32>)(bit<16>)meta.ingress_metadata.meta_len;
```

Figure B.3: *expected_Ackno* calculation

```
             hash(meta.ingress_metadata.hashFlow,
             HashAlgorithm.crc32,
             32w1, {hdr.ipv4.srcAddr,hdr.tcp.srcPort, hdr.ipv4.dstAddr, hdr.tcp.dstPort,
hdr.ipv4.protocol,meta.ingress_metadata.meta_hashcorrectACK1}, 32w6000);
```

Figure B.4: Hash flow calculation

```
if (hdr.ipv4.srcAddr < hdr.ipv4.dstAddr ){

    meta.ingress_metadata.meta_hashcorrectACK1 = (bit<32>) meta.ingress_metadata.meta_seq + (bit<32>)
(bit<16>)meta.ingress_metadata.meta_len;



       if (hdr.tcp.syn ==1 && hdr.tcp.ack == 1 || hdr.tcp.seqNo ==1 && hdr.tcp.ackNo ==1 )
{
            meta.ingress_metadata.meta_hashcorrectACK1 = hdr.tcp.seqNo+1; ;
           }


         Register1_seq_state.write(meta.ingress_metadata.hashFlow,
meta.ingress_metadata.meta_hashcorrectACK1);

         }
```

Figure B.5: sender-to-receiver control flow

```
hash(meta.ingress_metadata.hashFlowDirVerify,
            HashAlgorithm.crc32,
            32w1, {hdr.ipv4.dstAddr, hdr.tcp.dstPort, hdr.ipv4.srcAddr, hdr.tcp.srcPort,
hdr.ipv4.protocol,hdr.tcp.ackNo}, 32w6000);
```

Figure B.6: receiver-to-sender hash flow identifier calculation

```
      if (hdr.ipv4.srcAddr > hdr.ipv4.dstAddr && hdr.tcp.syn != 1){

Register1_seq_state.read(meta.ingress_metadata.meta_ack_verify,meta.ingress_metadata.hashFlowDirVerify);


         if (meta.ingress_metadata.meta_ack_verify == 0 || meta.ingress_metadata.meta_ack_verify <
hdr.tcp.ackNo && hdr.tcp.syn != 1 ){


        drop_packet1.apply();

            }
      }
```

Figure B.7: receiver-to-sender control flow

Appendix C

DNS cache poisoning attack detection P4 implementation details

Parser and header definition

To detect the ICMP responses triggered by the UDP packets with brute force, we need to parse the ICMP packets. The ICMP header to parse includes the ICMP response, and the UDP packet sent (*i.e.,* including the IP header) that triggered the ICMP response. In Figure C.1 we have the definition of the ICMP header combined with the IP and UDP headers that are included in the ICMP message response.

```
header icmp_t {
    bit<8>  type;
    bit<8>  code;
    bit<16> hdrChecksum;
    bit<32> unused;
    bit<4>  version;
    bit<4>  ihl;
    bit<6>  dscp;
    bit<2>  ecn;
    bit<16> totalLen;
    bit<16> identification;
    bit<3>  flags;
    bit<13> fragOffset;
    bit<8>  ttl;
    bit<8>  protocol;
    bit<16> hdrChecksumipv4;
    bit<32> srcAddr;
    bit<32> dstAddr;
    bit<16> srcPort;
    bit<16> dstPort;
    bit<16> length_;
    bit<16> checksum;
}
```

Figure C.1: ICMP header definition

Detecting DNS cache poisoning attack requires manipulating DNS domain names that are variable in length [103]. Parsing variable length fields in a programmable data plane is challenging since the parser is designed to parse fixed-length values. As highlighted in Section 4.3.1 in the DNS header, the question field separates the domain name into a set of labels, each preceded by its length in terms of character in bytes, the last label is followed by a `0x00` byte, which specifies

the end of the domain name. This separation into labels makes possible the parsing of domain names; for example a domain name '*example.com*' is composed of two labels and is represented as follows: $(0x07)example(0x03)com$

The DNS header, along with different label length headers, is defined in Figure C.2. The fields with exact length are defined with their length (e.g., DNS_ID with 16 bits length, *isQueryResponse* with 1 bit, which is the fields that identify the type of the DNS packet if either it is a query or a response (i.e., the QR field)).

Since the domain name labels are preceded by one octet that defines the length of the label in terms of characters. We define a header *dns_domain_length_label* with 8 bit representing the length of the label preceding each label. Labels may have variable length; we define three possible labels length headers with 8 bits, 16 bits, and 32 bits in order to parse labels with one byte, two bytes, three bytes or combine different lengths (e.g., parse a label of 4 bytes by combining two headers of 2 bytes).

In order to parse the domain name with its variable length, we need to define different parser states based on different lebel lengths (*i.e.,* from 1 byte to 20 bytes) as proposed in [103]. Given this restriction of the P4 parser (*i.e.,* can not parse variable lengths), we suppose that each domain name is composed with a maximum of 4 labels, each label has 20 bytes length at most. In Figure C.3, a parser state ($parse_dns_name_label1_length$) is defined to parse different domain name labels lengths (from 1 to 20 bytes), each transition refers to a state (presented in Figure C.4) to extract a specific label length. For example, suppose that we want to parse the domain name first label '*example*' with a length of 7 bytes. The parser first parses the $hdr.length_label1$ header, which is 7 *octet*, the parser goes to the state $parse_dns_label1_len7$ (in Figure C.4) to extract the label with seven bytes by extracting 1 byte, 2 bytes and 4 bytes headers. Then the parser reads the next length to parse the next label.

Set of registers

To track DNS flows for DNS cache poisoning mitigation, we use four registers (Figure C.7). The register $Register_ICMPQueryReply$ is used to track UDP packets and ICMP responses. A register $Register_CountICMPGlobal$ is used to track the number of UDP packets for the detection of the scan attempt for each potential victim (*i.e.* IP addresses of DNS servers).

The DNS non-legitimate responses mismatch the DNS Transaction Identifier (DNS.ID), the Port Number (UDP.dstPort), and match the Domain Name (DNS.domain) of the outstanding query. Therefore, to detect the possible DNS brute force, we use a register $Register_DNSRequestResponse$ to track the match and the mismatch between DNS queries and responses. A register $Register_CountDNSID$ is used to maintain the number of received DNS responses to detect a possible brute force.

Set of actions

Figure C.5 describes the P4 program to track port scan attempts, each port is associated with a context variable $MetaQR$ (*QueryReply* in the code) to define an open or closed port when receiving an ICMP unreachable message (e.g., a value of 1 for open (UDP packet received) and 2 for closed (ICMP unreachable packet received)). We maintain the metadata *QueryReply* value of 1 for each received UDP packet in the $Register_ICMPQueryReply$ using the hash of the DNS server IP address as a key (metadata *hashdstPortResult*). In the

```
header dns_t {
    bit<16> DNS_ID;
    bit<1> isQueryResponse;
    bit<4> opcode;
    bit<1> auth_answer;
    bit<1> trunc;
    bit<1> recur_desired;
    bit<1> recur_avail;
    bit<1> reserved;
    bit<1> authentic_data;
    bit<1> checking_disabled;
    bit<4> resp_code;
    bit<16> q_count;
    bit<16> answer_count;
    bit<16> auth_rec;
    bit<16> addn_rec;
}
header dns_domain_length_label {

 bit<8> length_label;

}
header dns_domain_label_1byte{
    bit<8> label;
}

header dns_domain_label_2byte {
    bit<16> label;
}

header dns_domain_label_4byte {
    bit<32> label;
}
```

Figure C.2: DNS header definition

$Register_CountICMPGlobal$, we maintain a count for each received UDP packet for the same destination server IP address hash (metadata $hashResultICMPQueryCount$). The set of UDP packets is sent to trigger the ICMP response packets. Upon receiving an ICMP unreachable message packet, the *QueryReply* metadata is updated to the value 2 using the port number returned in the ICMP response *hdr.icmp.dstPort* (program in Figure C.6).

For each DNS domain name, non-legitimate responses mismatch the DNS ID, the UDP port number (UDP.dstPort), and match the DNS domain name (DNS.domain) of the outstanding query. Therefore, we track the match and mismatch between DNS queries and responses to detect the possible DNS brute force. For each received query, a metadata $DNSRequestResponse$ is set to 1 and maintained in $Register_DNSRequestResponse$ per hash key ($hashResultDNSIDRequestResponse$), which is the hash result of the UDP source port $udp.srcPort$, the DNS transaction identifier $dns.DNS_ID$ and the set of domain name labels (e.g., $.label1_1byte.label, label1_2byte.label$) (Figure C.8). Upon receiving a response, the register entry is verified using the reverse port ($udp.dstPort$), the DNS transaction identifier, and domain name key to verify the match with the server query (i.e., MetaServerQR = 1).

As explained in Section 4.2.3 the Petri model is mapped to a MAT. Figure C.9 presents the definition of the MAT representing the Petri Net with the set of metadata representing the attack sub-goals as match keys. For each confirmed attack sub-goal, a token is set in the Petri Net that is associated with setting a metadata $\in Meta$ to 1. Achieving the set of sub-goals sg_1, sg_2, sg_3 results in setting the values of the corresponding metadata $meta1, meta2, meta3$, to 1 respectively (Figure C.10).

```
    @name("parse_dns") state parse_dns{
        packet.extract(hdr.dns);
        transition select(hdr.dns.isQueryResponse == 0 || hdr.dns.isQueryResponse == 1) {
        true : parse_dns_name_label1_length;
        false: accept;
        default: accept;
    }

}


   @name("parse_dns_name_label1_length") state parse_dns_name_label1_length{
       packet.extract(hdr.length_label1);


        transition select(hdr.length_label1.length_label) {
            0: accept;
            1: parse_dns_label1_len1;
            2: parse_dns_label1_len2;
            3: parse_dns_label1_len3;
            4: parse_dns_label1_len4;
            5: parse_dns_label1_len5;
            6: parse_dns_label1_len6;
            7: parse_dns_label1_len7;
            8: parse_dns_label1_len8;
            9: parse_dns_label1_len9;
            10: parse_dns_label1_len10;
            11: parse_dns_label1_len11;
            12: parse_dns_label1_len12;
            13: parse_dns_label1_len13;
            14: parse_dns_label1_len14;
            15: parse_dns_label1_len15;
            16: parse_dns_label1_len16;
            17: parse_dns_label1_len17;
            18: parse_dns_label1_len18;
            19: parse_dns_label1_len19;
            20: parse_dns_label1_len20;
            default: accept;
        }

}
```

Figure C.3: DNS parser definition

```
state parse_dns_label1_len1 {
     packet.extract(hdr.label1_1byte);
     transition parse_dns_name_label2_length;
 }

 state parse_dns_label1_len2 {
     packet.extract(hdr.label1_2byte);
     transition parse_dns_name_label2_length;
 }

 state parse_dns_label1_len3 {
     packet.extract(hdr.label1_1byte);
     packet.extract(hdr.label1_2byte);
    transition parse_dns_name_label2_length;
 }

 state parse_dns_label1_len4 {
     packet.extract(hdr.label1_4byte);
     transition parse_dns_name_label2_length;
 }

 state parse_dns_label1_len5 {
     packet.extract(hdr.label1_1byte);
     packet.extract(hdr.label1_4byte);
     transition parse_dns_name_label2_length;
 }

 state parse_dns_label1_len6 {
     packet.extract(hdr.label1_2byte);
     packet.extract(hdr.label1_4byte);
     transition parse_dns_name_label2_length;
 }

state parse_dns_label1_len7 {
     packet.extract(hdr.label1_1byte);
     packet.extract(hdr.label1_2byte);
     packet.extract(hdr.label1_4byte);
     transition parse_dns_name_label2_length;
 }

state parse_dns_label1_len8 {

     packet.extract(hdr.label1_4byte_1);
     packet.extract(hdr.label1_4byte_2);
     transition parse_dns_name_label2_length;
 }
```

Figure C.4: DNS parser transition definition

```
hash(meta.ingress_metadata.hashdstPortResult,
            HashAlgorithm.crc32,
            32w0, {hdr.udp.dstPort}, 32w500);



meta.ingress_metadata.QueryReply = 1;
Register_ICMPQueryReply.write(meta.ingress_metadata.hashdstPortResult, meta.ingress_metadata.QueryReply);


hash(meta.ingress_metadata.hashResultICMPQueryCount,
            HashAlgorithm.crc32,
            32w0, {hdr.ipv4.dstAddr}, 32w500);


Register_CountICMPGlobal.read(meta.ingress_metadata.CountICMPGlobal,
meta.ingress_metadata.hashResultICMPQueryCount);

meta.ingress_metadata.CountICMPGlobal = meta.ingress_metadata.CountICMPGlobal +1;

Register_CountICMPGlobal.write(meta.ingress_metadata.hashResultICMPQueryCount,
meta.ingress_metadata.CountICMPGlobal);
```

Figure C.5: Tracking UDP flow

```
hash(meta.ingress_metadata.hashICMPdstPortResult,
            HashAlgorithm.crc32,
            32w0, {hdr.icmp.dstPort}, 32w500);


 meta.ingress_metadata.QueryReply = 2;
Register_ICMPQueryReply.write(meta.ingress_metadata.hashICMPdstPortResult, meta.ingress_metadata.QueryReply);


}
```

Figure C.6: Maintain UDP flow

```
register<bit<2>>(500) Register_ICMPQueryReply;
register<bit<2>>(500) Register_DNSRequestResponse;
register<bit<32>>(500) Register_CountICMPGlobal;
register<bit<32>>(500) Register_CountDNSID;
```

Figure C.7: DNS cache poisoning register declaration

```
hash(meta.ingress_metadata.hashResultDNSIDRequestResponse,
            HashAlgorithm.crc32,
            32w0, {hdr.udp.srcPort,  hdr.dns.DNS_ID, hdr.label1_1byte.label, hdr.label1_2byte.label,
hdr.label1_4byte.label, hdr.label1_4byte_1.label, hdr.label1_4byte_2.label,  hdr.label2_1byte.label,
hdr.label2_2byte.label, hdr.label2_4byte.label, hdr.label2_4byte_1.label, hdr.label2_4byte_2.label,
hdr.label3_1byte.label, hdr.label3_2byte.label, hdr.label3_4byte.label ,hdr.label3_4byte_1.label,
hdr.label3_4byte_2.label, hdr.label4_1byte.label, hdr.label4_2byte.label,
hdr.label4_4byte.label ,hdr.label4_4byte_1.label, hdr.label4_4byte_2.label}, 32w500);

 meta.ingress_metadata.DNSRequestResponse = 1;
Register_DNSRequestResponse.write(meta.ingress_metadata.hashResultDNSIDRequestResponse,
meta.ingress_metadata.DNSRequestResponse);
```

Figure C.8: Maintain DNS Queries

```
@name("DNS_Petri_table") table DNS_Petri_table {

actions = {

            drop_me;
            AlertDNSBruteForce;
            NoAction;
        }
 key = {


           meta.ingress_metadata.meta3:exact;
           meta.ingress_metadata.meta2:exact;
           meta.ingress_metadata.meta1:exact;

        }
        size = 512;
        default_action = NoAction();

}
```

Figure C.9: DNS cache poisoning Petri Net MAT

```
if (meta.ingress_metadata.CountICMPGlobal > 4) {

meta.ingress_metadata.meta1 = 1;
}

if ( meta.ingress_metadata.OutScannedPort2 == 1 &&
          meta.ingress_metadata.DNSRequestResponse == 1) {

meta.ingress_metadata.meta3 = 1;

Register_MatchedDnsResponse.write (meta.ingress_metadata.hashServerIP ,  meta.ingress_metadata.meta3);

}


Register_MatchedDnsResponse.read(meta.ingress_metadata.meta3, meta.ingress_metadata.hashServerIP);

if ( meta.ingress_metadata.CountDNSID > 4 ) {

meta.ingress_metadata.meta2 = 1;

}
```

Figure C.10: DNS cache poisoning detection flow control

Algorithm 3 DNS cache poisoning attack detection algorithm

Definitions:

- n: number of monitored packets
- $DNSRequestResponse[n]$: register array of the DNS Query and responses to maintain $MetaServerQR$ metadata
- $ICMPQueryResponse[n]$: register array of the ICMP responses to maintain $MetaQR$ metadata
- $CountScan[n]$: register array of UDP packet count
- $CountDNSResponse[n]$: register array of DNS responses count
- $PetriNetTable[m]$: table of Petri Net abstraction

1: p ← parse(pkt)
2: id1 ← hash(p.udp.dstPort)
3: id2 ← hash(p.icmp.dstPort)
4: id3 ← hash(p.ip.dstIP)
5: id4 ← hash(p.udp.srcPort, p.dns.id, p.dns.domain)
6: id5 ← hash(p.udp.dstPort, p.dns.id, p.dns.domain)
7: id6 ← hash(p.dns.domain)
8: **while** $pkt = UDP$ packet **do**
9: $ICMPQueryResponse[id1] \leftarrow 1$
10: $CountScan[id3] \leftarrow CountScan[id3] + 1$
11: **end while**
12: **while** $pkt = ICMP$ packet and $p.icmp.type = 3$ **do**
13: $ICMPQueryResponse[id2] \leftarrow 2$
14: **end while**
15: **while** $pkt = DNS$ Query **do**
16: $DNSRequestResponse[id4] \leftarrow 1$
17: **end while**
18: **while** $pkt = DNS$ Response **do**
19: **if** $p.udp.srcPort == 53$ and $p.dns.id > 0$ and $DNSRequestResponse[id5]! = 1$ **then**
20: $CountDNSResponse[id6] \leftarrow CountDNSResponse[id6] + 1$
21: **end if**
22: **end while**
23: **if** $ICMPQueryResponse[id1] == 1$ and $DNSRequestResponse[id5] = 1$ **then**
24: $setToken(p1) \implies meta1 = 1$
25: **end if**
26: **if** $CountScan[id3] > TH\ scan$ **then**
27: $setToken(p2) \implies meta2 = 1$
28: **end if**
29: **if** $CountDNSResponse[id6] > TH\ brute$ **then**
30: $setToken(p3) \implies meta3 = 1$
31: **end if**

List of Figures

1.1 Software Defined Network architecture 2
1.2 Attack mitigation solutions (end-host vs. data plane-based) 4
1.3 Thesis contributions 6

2.1 Programmable networks history 10
2.2 Traditional vs. Programmable SDN data plane 11
2.3 PISA architecture [7] 13
2.4 P4-based switch pipeline 14
2.5 Parser FSM 15
2.6 Parser program 16
2.7 Match Action Table 16
2.8 P4 basic code 17
2.9 P4 deparser program 18
2.10 FAST tables [11] 20
2.11 SDPA tables [12] 22
2.12 Openstate tables [13] 22
2.13 P4 applications 27

3.1 FSM counter example 35
3.2 EFSM counter example 36
3.3 Approach overview 36
3.4 EFSM event example 38
3.5 EFSM DDoS example 39
3.6 EFSM mapping to P4 elements and associated I/O operations 40
3.7 Accessing current state 41
3.8 EFSM event example 41
3.9 ECN (normal behavior in blue, misbehavior in red, ellipses represent EFSM state updates) 44
3.10 EFSM abstraction for ECN 46
3.11 Register declaration 47
3.12 Register write 47
3.13 Intrinsic metadata 48
3.14 metadata usage 48
3.15 hash result metadata 49
3.16 RED MAT definition 49
3.17 RED MAT entries 49
3.18 ECN flag set 49

3.19 Control Flow example 50
3.20 Experimental topology used in mininet 50
3.21 Misbehaving and normal flow throughput depending on the marking threshold . 52
3.22 Misbehaving and normal flow throughput depending on the marking threshold using RED 53
3.23 Switch processing time 54
3.24 Optimistic ACK attack 55
3.25 EFSM for Optimistic ACK detection and reaction 57
3.26 Optimistic ACK in action 59
3.27 Topology 60
3.28 Optimistic ACK impact on the normal flow 61
3.29 Switch processing Time 62

4.1 Satisfied condition 67
4.2 Example of Petri Net events 67
4.3 Petri Net connection establishment example 68
4.4 Approach overview 69
4.5 Petri Net Example 72
4.6 Places representation 73
4.7 DNS Query Response 75
4.8 DNS cache poisoning overview 76
4.9 DNS cache poisoning attack 77
4.10 Multi-step DNS cache poisoning attack 80
4.11 Petri Net of DNS cache poisoning 80
4.12 Port scan EFSM 82
4.13 DNS brute force EFSM 83
4.14 Topology 84
4.15 DNS number of responses for different behavior combinations 84
4.16 Switch processing time vs. Number of attack sub-goals 85
4.17 Switch processing time vs. table entry position 86

B.1 expected_ackno register declaration 114
B.2 *payloadsize* calculation 114
B.3 *expected_Ackno* calculation 114
B.4 Hash flow calculation 114
B.5 sender-to-receiver control flow 114
B.6 receiver-to-sender hash flow identifier calculation 115
B.7 receiver-to-sender control flow 115

C.1 ICMP header definition 117
C.2 DNS header definition 119
C.3 DNS parser definition 120
C.4 DNS parser transition definition 121
C.5 Tracking UDP flow 121
C.6 Maintain UDP flow 122
C.7 DNS cache poisoning register declaration 122
C.8 Maintain DNS Queries 122
C.9 DNS cache poisoning Petri Net MAT 122

C.10 DNS cache poisoning detection flow control 123

List of Tables

2.1 Stateful abstraction classification 24
2.2 Stateful attack detection 30

3.1 Key notations used in the EFSM model 37

4.1 Petri Net as a MAT 73
4.2 DNS message 75
4.3 Attack MAT 81