UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ALEXANDRE DA SILVEIRA ILHA

Towards a General Approach for Cyberattack Detection Using Programmable Data Planes

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Luciano Paschoal Gaspary

Porto Alegre
May 2022

CIP — CATALOGING-IN-PUBLICATION

da Silveira Ilha, Alexandre

Towards a General Approach for Cyberattack Detection Using Programmable Data Planes / Alexandre da Silveira Ilha. – Porto Alegre: PPGC da UFRGS, 2022.

94 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Luciano Paschoal Gaspary.

1. DDoS attacks. 2. Detection. 3. Mitigation. 4. Programmable data planes. 5. P4. 6. Entropy analysis. 7. Advanced persistent threats. 8. Network Intrusion Detection Systems. 9. Zeek. I. Paschoal Gaspary, Luciano. II. Título.

ACKNOWLEDGMENTS

I am grateful to Professor Luciano Paschoal Gaspary, Ph.D., for his priceless advice, friendship, patience, and support. It is an honor to study under your supervision.

Many thanks to Jonatas Adilson Marques, Ph.D., for the significant input and feedback during the development of this thesis; to Ângelo Cardoso Lapolli, M.Sc., whose work on DDoS detection keeps inspiring several research initiatives; to Lucas Nunes Alegre, Ph.D. candidate, for the long-time friendship and the technical support whenever the author got himself into an infinite loop; to Márcio Antônio Lawisch, M.Sc. student, whose interest on the EUCLID project motivated us to improve the code and documentation; and to Lucas Sonntag Hagen, B.Sc. student, for the fruitful discussions about RNA and automated code generation, not to mention the valuable implementation insights.

I dedicate this work to my parents (*in memoriam*), for having set the foundation for everything that followed; to my beloved wife, Grasiela, for her unconditional encouragement and support; and to my family, friends, and colleagues, for being patient about my relative absence during this research endeavor.

Nothing will ever be attempted if all possible objections must be first overcome. —

SAMUEL JOHNSON

ABSTRACT

Distributed Denial-of-Service (DDoS) and Advanced Persistent Threat (APT) are increasingly prominent and severe cyberattack categories that cause relevant damages and losses to Internet-connected organizations. DDoS attacks can compromise the availability of otherwise highly-resilient links and services. Stealthy APTs potentially lead to compromised information assets and public safety hazards. Existing defenses require frequent interaction between forwarding and control planes, making it difficult to reach a satisfactory trade-off between accuracy, resource usage, and defense response delay. Moreover, protection against APTs relies on Network Intrusion Detection Systems (NIDS), whose traffic inspection capabilities face scalability concerns related to the need to copy packet data from forwarding devices to the main memory of general-purpose computers. Recently, high-performance Programmable Data Planes (PDPs) enabled the development of a new generation of mechanisms to analyze and manage traffic at line rate. In this thesis, we investigate the potential of PDPs as a foundation for cybersecurity solutions. Our work has two iterations. In the first iteration, we propose EUCLID, a novel real-time DDoS attack detection and mitigation mechanism that can be executed entirely in a P4 forwarding device. Our experimental evaluation shows that our P4-based design has the potential to meet increasingly strict performance requirements in high-volume networks. In the second iteration, we pursue a general approach for cyberattack detection using PDPs. We introduce RNA, an innovative framework to offload NIDS-related operations from general-purpose CPUs to high-performance PDPs. RNA uses the mechanisms of a programmable switch to analyze traffic, summarize information about it, and send these summaries to a host-based component, which, in turn, translates these summaries into events the NIDS can handle. Using the BMv2 P4 switch and the Zeek Network Security Monitor as platforms, we built a proof-of-concept implementation of our framework. Through a series of examples and case studies, we demonstrated the feasibility of our design and its integration with Zeek. We showed that: (*i*) we can automate monitoring session setup, (*ii*) it is possible to offload lightweight packet inspection to the PDP, (*iii*) RNA can forward EUCLID alarms to Zeek, and (*iv*) we can filter traffic for Zeek in the PDP. We also concluded from these examples and studies that we can gradually add data plane support for more protocols and adapt our framework to identify higher-level network events. As RNA capabilities grow, we reduce the need for Zeek to do all the CPU-intensive packet analysis by itself.

Keywords: DDoS attacks. Detection. Mitigation. Programmable data planes. P4. Entropy analysis. Advanced persistent threats. Network Intrusion Detection Systems. Zeek.

Rumo a uma Solução Geral para Detecção de Ataques Cibernéticos baseada em Planos de Dados Programáveis

RESUMO

Distributed Denial-of-Service (DDoS) e *Advanced Persistent Threats* (APTs) são categorias de ataques cibernéticos cada vez mais proeminentes e graves, que causam danos e perdas relevantes a organizações conectadas à Internet. Os ataques DDoS podem comprometer a disponibilidade de *links* e serviços altamente resilientes. APTs furtivos potencialmente levam a ativos de informação comprometidos e a riscos à incolumidade pública. As defesas existentes exigem interação frequente entre os planos de encaminhamento e controle, dificultando a obtenção de um equilíbrio satisfatório entre precisão, uso de recursos e atraso na resposta da defesa. Além disso, a proteção contra APTs depende de Sistemas de Detecção de Intrusão de Rede (NIDS), cujos recursos de inspeção de tráfego enfrentam problemas de escalabilidade relacionados à necessidade de copiar dados (de pacotes) de dispositivos de encaminhamento para a memória principal de computadores de uso geral. Recentemente, Planos de Dados Programáveis (PDPs) de alto desempenho permitiram o desenvolvimento de uma nova geração de mecanismos para analisar e gerenciar tráfego em taxa de linha. Nesta dissertação, investiga-se o potencial dos PDPs como base para soluções de segurança cibernética. Este trabalho tem duas iterações. Na primeira iteração, propõe-se o EUCLID, um novo mecanismo de detecção e mitigação de ataques DDoS em tempo real que pode ser executado inteiramente em um dispositivo de encaminhamento P4. A avaliação experimental mostra que a solução tem potencial para atender a requisitos de desempenho cada vez mais rigorosos em redes de alto volume. Na segunda iteração, busca-se uma abordagem geral para detecção de ataques cibernéticos usando PDPs. Apresenta-se o RNA, uma estrutura inovadora para descarregar operações relacionadas ao NIDS de CPUs de uso geral para PDPs de alto desempenho. O RNA usa os mecanismos de um *switch* programável para analisar o tráfego, resumir informações sobre ele e enviar esses resumos para um componente baseado em *host*, que, por sua vez, traduz esses resumos em eventos que o NIDS pode manipular. Usando o *switch* P4 BMv2 e o Zeek Network Security Monitor como plataformas, construímos uma implementação de prova de conceito da estrutura proposta. Através de uma série de exemplos e estudos de caso, demonstra-se a viabilidade deste projeto e sua integração com o Zeek. Mostra-se que: (*i*) é possível automatizar a configuração da sessão de monitoramento, (*ii*) é possível descarregar a inspeção leve de pacotes para o PDP, (*iii*) o RNA pode encaminhar alarmes EUCLID para o Zeek e (*iv*) pode-se filtrar o tráfego para Zeek no PDP. Também conclui-se a partir desses

exemplos e estudos que é possível adicionar gradualmente ao plano de dados o suporte a mais protocolos e adaptar a estrutura para identificar eventos de rede de nível superior. À medida que os recursos do RNA crescem, reduz-se a necessidade de o Zeek fazer sozinho toda a análise de pacotes com uso intensivo de CPU.

Palavras-chave: Ataques DDoS. Detecção. Mitigação. Planos de dados programáveis. P4. Análise de entropia. Sistemas de Detecção de Intrusão de Redes. Zeek.

LIST OF ABBREVIATIONS AND ACRONYMS

ALC	Attack Life Cycle
API	Application Programming Interface
APT	Advanced Persistent Threat
AS	Autonomous System
ASIC	Application-Specific Integrated Circuit
CLI	Command-Line Interface
CPU	Central Processing Unit
CSIRT	Computer Security Incident Response Team
DDoS	Distributed Denial-of-Service
DPD	Dynamic Protocol Detection
DPI	Deep Packet Inspection
DPP	Data Plane Programmability
DRDoS	Distributed Reflective Denial-of-Service
EE	Event Engine
EWMA	Exponentially-Weighted Moving Average
EWMMD	Exponentially-Weighted Mean Deviation
FPR	False-Positive Rate
FSM	Finite State Machine
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IKC	Intrusion Kill Chain
IP	Internet Protocol
ISP	Internet Service Provider
IXP	Internet Exchange Point

LPI	Lightweight Packet Inspection
LPM	Longest-Prefix Match
MAC	Media Access Control
mRNA	RNA Message Format
NIC	Network Interface Card
NIDS	Network Intrusion Detection System
NIST	National Institute of Standards and Technology
NPU	Network Processing Unit
NTP	Network Time Protocol
OW	Observation Window
PAF	Packet Analysis Framework
PCF	Pre-Computed Function
PDP	Programmable Data Plane
PDU	Protocol Data Unit
PFD	Programmable Forwarding Device
PRE	Packet Replication Engine
PSI	Policy Script Interpreter
RAM	Random-Access Memory
RMT	Reconfigurable Match Table
RNA	Reconfigurable Network Analytics
SDN	Software-Defined Networking
SRAM	Static Random-Access Memory
TCAM	Ternary Content-Addressable Memory
TCP	Transmission Control Protocol
TPR	True-Positive Rate
UDP	User Datagram Protocol

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

Internet-connected systems have been increasingly targeted for various types of cyberattacks, which cause relevant damages and losses to corporate and governmental systems, including critical infrastructures. Two broad attack categories are at the forefront: Distributed Denial-of-Service (DDoS) and Advanced Persistent Threats (APTs). DDoS attacks remain the most severe threat to the security of networked systems (HUMMEL; HILDEBRAND, 2021). Increases in frequency and intensity of outbreaks constantly gain the headlines for causing outages even in large-scale online service providers (e.g., Yandex (MARROW; STOLYAROV, 2021), Cloudflare (YOACHIMIK, 2021), Microsoft Azure (WARREN, 2021), and Amazon Web Services (NICHOLSON, 2020)). Peak data rates generated during attack campaigns amount to several terabits per second and flood high-capacity links. Similarly, assaults reaching billions of packets or millions of requests per second can quickly inundate forwarding devices and network servers. Generating such digital tsunamis has typically required numerous attack sources. However, reflection and amplification techniques waived this requirement and gave rise to *Distributed Reflective Denial-of-Service* attacks (ROSSOW, 2014). The ongoing trend towards worse DDoS incidents has accelerated since early 2021. As the COVID-19 pandemic rushed organizations into often-insecure telecommuting arrangements, threat actors seized the opportunity to exploit the abundance of vulnerabilities and develop innovative and complex attack methods (HUMMEL; HILDEBRAND, 2021).

The Rise of APTs. Like DDoS attacks, APTs have become increasingly prominent over time (ALSHAMRANI et al., 2019). In contrast to blatant DDoS assaults, APT incursions are stealthier and go unnoticed for long periods, even years (MCWHORTER, 2013). Despite their secretive nature, APTs expose targets to long-lasting or even permanent damage—-which may include sabotage of cyber-physical systems. Threat actors typically seek access to critical, sensitive, or strategic data containing important information so that it becomes possible to exfiltrate, corrupt, or even destroy such data (CHEN; DESMET; HUYGENS, 2014). A typical APT campaign fits models such as the *Intrusion Kill Chain* (HUTCHINS; CLOPPERT; AMIN, 2011) or the *Attack Life Cycle* (MCWHORTER, 2013). Under these models, intruders initially strive to deploy a backdoor into a vulnerable network asset, thus establishing a foothold inside the target infrastructure. Then, operating from that entry point, gradually and surreptitiously, attackers expand their control to other network assets until obtaining enough privilege to accomplish their mission goals.

Problem Definition. Defense mechanisms must cater to the needs of present-day high-speed networks, whose data rates also reach the order of tens of terabits per second—especially in Internet Exchange Points (IXPs) and Service Providers (ISPs). It is a significant challenge for these mechanisms to defend such networks and their customers while meeting increasingly strict requirements for accuracy, latency, throughput, cost, and flexibility. Existing defense mechanisms pursue a satisfactory trade-off between these often-conflicting goals, typically relying on highly-specialized hardware or delegating functions to software on remote servers. Using specialized hardware, such as middleboxes based on fixed-function Application-Specific Integrated Circuits (ASICs), promotes high accuracy, low latency, and high throughput. However, this approach demands significant capital and operational expenditures (FEAMSTER; REXFORD; ZEGURA, 2014), besides potentially leading to vendor lock-in and requiring "forklift upgrades."

Conversely, software-based solutions are more flexible than custom-built hardware but require continuous interaction and coordination between servers and forwarding devices. Moreover, analyzing every forwarded packet in software would lead to unacceptably large overheads on processor time, memory allocation, and network management traffic. Hence, it is mandatory to diminish these overheads, which is commonly achieved by packet sampling (e.g., sFlow (PHAAL; PANCHEN; MCKEE, 2001)) and flow-based accounting (e.g., NetFlow (CLAISE, 2004) and OpenFlow (MCKEOWN et al., 2008)). Despite their benefits, we advocate that these approaches still fall short in either *accuracy* or *resource usage*, depending on analysis granularity (MOSHREF; YU; GOVINDAN, 2013). Moreover, the required coordination between data and control planes implies a long control loop, which leads to non-negligible *delays* in detection and mitigation.

The already intricate scenario of potential solutions becomes further convoluted when defending against APTs, a task that requires Network Intrusion Detection Systems (NIDSes) (LI et al., 2018). Even though NIDSes have been developed and researched for over 25 years, there is still a fundamental scalability problem with at least two facets: first, modern high-speed networks make it increasingly costly to copy packets from data planes to RAM buffers; second, NIDSes often require stateful inspection and packet payload analysis. Performing line-rate traffic analysis to uncover subtler clues of malicious activity remains a complex challenge (ZHAO et al., 2020).

Motivation. Recent technological advances present an unprecedented opportunity to tackle such a challenge. *Data Plane Programmability* (DPP) has emerged as a promising alternative to deal with the issues mentioned above by enabling the *in-network* execution

of novel packet processing algorithms (FEAMSTER; REXFORD; ZEGURA, 2014). This paradigm allows a programmer to express forwarding logic as code (with elementary primitives for header manipulation, memory access, and table lookup) delegated to forwarding devices across the network. DPP enables full-scale packet inspection directly within the data plane, thus facilitating low-latency and high-throughput network defense.

Several works leverage Programmable Data Planes (PDPs) to enhance the scalability of network management and monitoring functions. These solutions introduce essential building blocks, such as algorithms optimized for line-rate execution (e.g., Yu, Jose and Miao (2013), Liu et al. (2016), Yang et al. (2018)), stream processing query languages (e.g., Gupta et al. (2016), Narayana et al. (2017)), and defense primitive operations (e.g., Li et al. (2021)). Some of these works do consider and even present interesting security-related case studies. However, the general-purpose focus of these solutions results in monitoring constructs that do not meet the functionality demanded to instantiate sophisticated security-related detection and mitigation mechanisms. Nevertheless, the immense flexibility of PDPs makes it possible to devise solutions tailored and optimized for network defense.

Objectives. In this thesis, we explore the potential of programmable data planes as an underpinning for novel network defense solutions. Our work towards a security framework based on programmable networks has two iterations. In the first iteration, aiming to protect networks against volumetric DDoS attacks and to push the limits of PDPs, we propose EUCLID (ILHA et al., 2021), a full-fledged solution towards low-latency, fine-grained traffic analysis to detect and mitigate DDoS outbreaks. This work, which we detail in Chapter 3, builds upon an anomaly-based detection mechanism developed within our research group (LAPOLLI; MARQUES; GASPARY, 2019) that relies on IP address Shannon entropy to characterize legitimate traffic patterns and generate warnings about abnormal conditions. EUCLID further introduces a defense mechanism that responds to these warnings, thus integrating the *detection and mitigation* of attacks entirely into the data plane. As far as we are aware, our work was the first to offload this kind of anomaly detection and mitigation mechanism to programmable network devices. To meet the strict data plane time and memory constraints, EUCLID approximates frequencies using custom *count sketches* (CHARIKAR; CHEN; FARACH-COLTON, 2002) and performs compute-intensive mathematical operations with the aid of a memory-optimized longest-prefix match (LPM) table. Our method identifies suspect packets and enforces an arbitrary security policy (such as discarding, throttling, or detouring) to prevent suspicious traffic from disrupting networked services. We assess our method's efficacy through an extensive

experimental evaluation based on a proof-of-concept P4 prototype, to which we submit realistic workloads. We also compare the performance of our mechanism with that of well-established solutions.

In the second iteration, coming from the experience accumulated in our previous work, we investigate the possibility of stepping towards a general approach for attack and intrusion detection. We seek a solution that (*i*) considers CPU-intensive bulk traffic analysis and monitoring operations a NIDS requires, (*ii*) offloads such operations to programmable data planes, and (*iii*) enables the data plane to notify the NIDS about relevant events. To reach these objectives, we propose RNA, a framework that pushes bulk traffic processing away from NIDS hosts and into the network itself. Specifically, considering the Zeek Network Security Monitoring Tool (which includes NIDS capabilities) and the P4 programming language, we design a proof-of-concept implementation of our proposed solution, tailored for representative case studies.

Contributions. The main contributions of this thesis are described next.

1. We push the limits of data plane programming primitives and constructs to design an *in-switch* mechanism to protect networks against volumetric DDoS attacks.

2. In contrast to existing approaches and our previous work, we design an architecture that integrates DDoS attack detection and mitigation entirely in the data plane.

3. We thoroughly evaluate the performance advantages of offloading anti-DDoS attack solutions to programmable data planes.

4. We introduce a framework that offloads traffic analysis routines from CPU-based NIDS hosts to programmable forwarding devices.

5. We take meaningful steps towards a PDP-enhanced general solution for cyberattack detection.

Organization. This thesis is organized as follows: In Chapter 2, we provide an overview about DDoS attacks, APTs, network intrusion detection, and traffic analysis acceleration strategies, leading to an analysis of the related work. In Chapter 3, we introduce our design for DDoS attack detection and mitigation, discuss its implementation in a programmable switch, present our evaluation methodology, and discuss the results we obtained. In Chapter 4, we present and discuss our framework for traffic analysis offloading, its prototype implementation, and a series of case studies. In Chapter 5, we conclude the text with final remarks and perspectives for future work.

2 BACKGROUND AND STATE OF THE ART

In this chapter, the first section introduces and characterizes distributed denial-of-service attacks and advanced persistent threats. The following section discusses network intrusion detection using the Zeek Network Security Monitoring Tool. We then elaborate on the need to move away from CPUs and into programmable data plane-based acceleration for intrusion detection tasks. We conclude this chapter with an analysis of existing defenses and related work.

2.1 Distributed Denial-of-Service Attacks and Advanced Persistent Threats

Among several security threats, in this work, we focus on two main ones, namely, Distributed Denial-of-Service (DDoS) attacks and Advanced Persistent Threats (APTs). Next, we briefly revisit some of the concepts related to these threat categories.

DDoS Attacks. The term *distributed denial-of-service* encompasses massive cyberattacks against online services (MIRKOVIC; REIHER, 2004). Such attacks attempt to overload target systems through network congestion or computing resource saturation to degrade the quality or disrupt the availability of these services. DDoS attacks require that the attacker seizes control over a large number of hosts, commonly known as *bots* or *zombies*. Collectively, such hijacked hosts form a remotely-managed *botnet*. One can classify a DDoS attack according to the main exploited weakness as either *brute-force* (which floods the target with excessive requests) or *semantic* (which abuses implementation bugs or features to cause the target to operate in unintended modes).

Semantic DDoS Attacks. Semantic attacks are also known as protocol exploitation or state exhaustion attacks (SWAMI; DAVE; RANGA, 2019). State exhaustion attacks are generally low-rate and work by intentionally allowing otherwise legitimate transactions to time out, thus withholding and depleting resources on the target systems. Some notable instances are the HTTP-based slow DDoS attacks discussed by Muraleedharan and Janet (2017), such as Slow HTTP Headers (Slowloris), Slow HTTP POST (RUDY), and Slow Read. Defending against this type of attack requires tracking state information (at the transport or application layer) for an extended time.

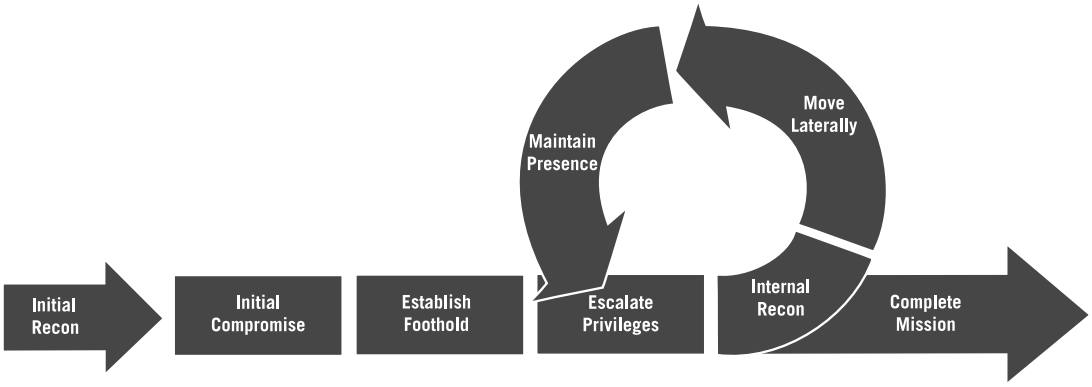
Reflective DDoS Attacks. The effectiveness of a brute-force, flooding DDoS attack depends largely on the source botnet size and aggregate bandwidth. Malicious actors can augment their firepower by combining brute-force and semantic tactics into

a strategy termed *Distributed Reflective Denial-of-Service* (DRDoS) (ROSSOW, 2014). DRDoS attacks exploit characteristics or weaknesses of widespread protocols and services (e.g., DNS and NTP) to turn vulnerable servers into high-volume traffic generators. In this type of assault, attackers command bots to flood servers with numerous but small service requests—spoofed to appear to have come from a given victim, which causes servers to "reflect" traffic back to that victim. When the disproportionally "amplified" volume of service responses arrives at the target, denial of service is a likely outcome.

Advanced Persistent Threats. An Advanced Persistent Threat (APT) is a covert, multi-step, and long-term cyberattack initiative against a well-defined target information technology infrastructure. The main characteristic of an APT is the inconspicuous movement of a group of malicious actors whose goal usually is target data exfiltration (AL-SHAMRANI et al., 2019; Cisco Systems, Inc., 2022). The US National Institute of Standards and Technology (NIST) defines an APT as a skilled and resourceful adversary able to use multiple attack vectors to gain, keep, and extend footholds within the target network, from which the attacker can launch further attacks (RADACK et al., 2011). Furthermore, the NIST considers that the APT (*i*) progresses on its mission over an extended timeframe, (*ii*) adapts to defense countermeasures, and (*iii*) commits to the level of effort required to accomplish its goals. According to Stojanović, Hofer-Schmitz and Kleb (2020), each word in "APT" points to a set of traits, wherein: *advanced* means stealthy, targeted, data-focused attacks, where malicious actors attempt to use multiple techniques to gain access; *persistent* refers to the extended timeframe during which attackers operate surreptitiously within the network to escalate their privileges; and *threat* indicates the possibility of significant damage to the target as a result of the attack mission.

APT Attack Models. APTs differ from automated campaigns in which malicious actors first scan the Internet searching for vulnerable assets and then exploit such vulnerabilities. In an APT, first, attackers select a high-value target and then carry out a sequence of steps, in a patient manner, attempting to accomplish their goals in the long run (SOOD; ENBODY, 2013). APT attack models help clarify and understand adversarial modes of operation and objectives (AL-MOHANNADI et al., 2016; STOJANOVIĆ; HOFER-SCHMITZ; KLEB, 2020). Two APT attack models commonly considered when analyzing APTs in enterprise-scale network scenarios are the *Intrusion Kill Chain* (IKC) (HUTCHINS; CLOPPERT; AMIN, 2011) and the *Attack Life Cycle* (ALC) (MCWHORTER, 2013).

Figure 2.1 – Attack Life Cycle



Source: McWhorter (2013).

The diagram in Figure 2.1 and the discussion in Stojanović, Hofer-Schmitz and Kleb (2020), which presents a systematic review of attack models and modeling techniques, help us explain the main steps of the ALC model:

1. Initial Reconnaissance (or *Recon*): information gathering through social engineering and open-source intelligence to find potential entry points to the target network.

2. Initial Compromise: attackers deliver malware through techniques such as spear phishing, watering hole attacks, and vulnerability exploitation of Internet-facing servers.

3. Establish Foothold: backdoor deployment to allow further connections.

4. Escalate Privileges: credential collection through brute-force attacks or dumping of hashes and passwords.

5. Internal Reconnaissance: intruders scan the internal networks searching for additional information, such as devices, running services, and privileged accounts.

6. Move Laterally: covert abuse of previously compromised accounts to gain access to more network assets, similarly to the Initial Compromise stage.

7. Maintain Presence: backdoor deployment across multiple systems (to prevent losing access should a backdoor be detected and removed), using legitimate credentials for remote access, and access to web-based systems.

8. Complete Mission: at this stage, attackers likely initiate data exfiltration, potentially followed by sabotage, data corruption, destruction, or encryption.

It is essential to notice that from the initial compromise upon mission completion, attackers generate varying degrees of extraneous events that, theoretically, a well-prepared Computer Security Incident Response Team (CSIRT) would be able to detect and handle. Additionally, given enough time, the CSIRT would be able to curtail the extent to which an APT has already compromised the information technology infrastructure. It is for these reasons that APT groups will make every effort to remain undetected.

Nevertheless, research literature and technical references provide numerous red flags and indicators of ongoing APT activities (e.g., Bhatt, Yano and Gustavsson (2014), Friedberg et al. (2015), Ahmed, Mahmood and Hu (2016), Marchetti et al. (2016), Strom et al. (2020), and MITRE (2022)). Well-studied APT offensives also provide insight into how APT groups operate. For instance, McWhorter (2013) exposes the Chinese APT1 group attacks between 2006 and 2013. Using the intrusion kill chain model, Chen, Desmet and Huygens (2014) analyze four different APT campaigns. In a broad study, Ussath et al. (2016) compare methods and techniques of 22 APT strikes. In their journal article, Alshamrani et al. (2019) scrutinize lateral movement tactics used in five incidents.

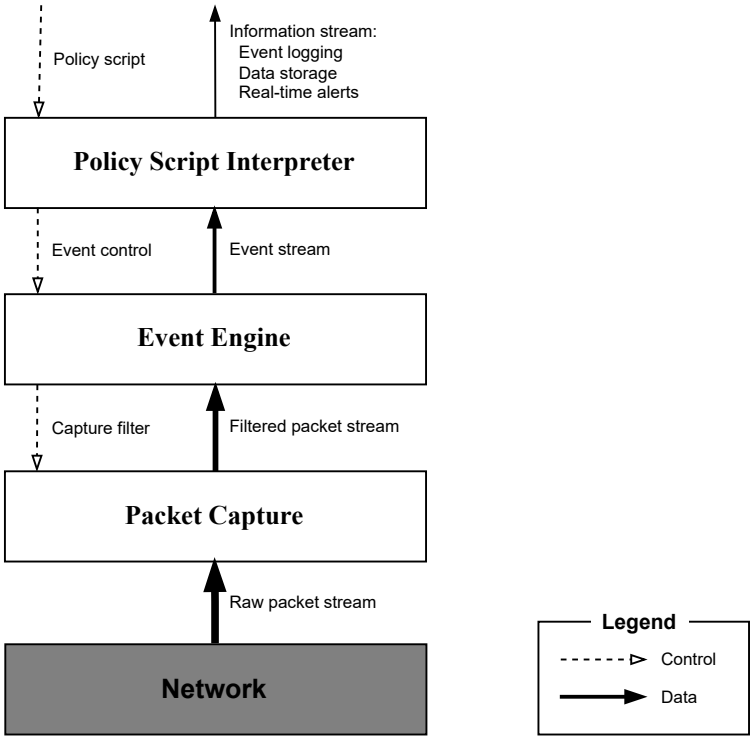
2.2 Intrusion Detection with the Zeek Network Security Monitoring Tool

According to Kurose and Ross (2017), an Intrusion Detection System (IDS) is a tool that generates alerts for network administrators when it observes potentially malicious traffic. As such, an IDS is of utmost importance for network defense against cyberattacks, particularly APTs (STOJANOVIĆ; HOFER-SCHMITZ; KLEB, 2020). Among several open-source IDS solutions, we can highlight Snort (Cisco Systems, 2022), Suricata (The Open Information Security Foundation, 2022), and Zeek (The Zeek Project, 2022). Henceforth, we focus on Zeek for two main reasons: first, it is widely deployed; second, we use it in our proof-of-concept implementation described in Chapter 4.

Formerly known as Bro (PAXSON, 1999), the Zeek Network Security Monitoring Tool is a passive IDS whose design emphasizes high-speed network monitoring, real-time notification, and extensibility (The Zeek Project, 2022). The tool pursues its performance goals through a *layered* and *distributed* architecture. The Zeek layers are depicted in Figure 2.2. Right at the bottom, the *Packet Capture*[1] layer filters the raw *packet stream* coming from the network. Next, the *Event Engine* (EE) reduces the packet stream into a semantically-enriched *event stream*. Finally, the *Policy Script Interpreter* (PSI) further summarizes the event stream, generating logs, storing data files, and sending real-time

[1] Originally termed `libpcap`.

Figure 2.2 – Zeek Layered Architecture



Source: adapted from Paxson (1999).

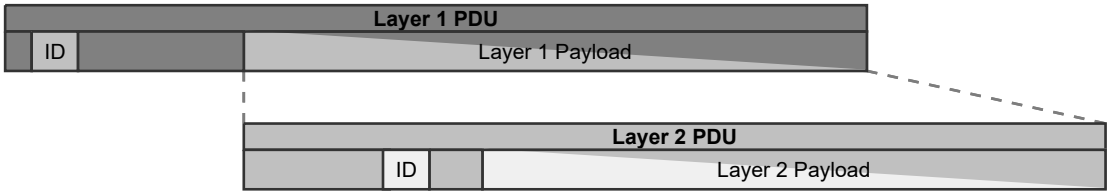
alerts to operators. The upward solid arrows indicate how the data streams are progressively distilled into more meaningful and manageable items until finally becoming an information source for human analysis. Next, we briefly explain each Zeek architectural layer.

Packet Capture. This layer typically uses the open-source library `libpcap` (JACOBSON; LERES; MCCANNE, 1994) and `tcpdump` filter expressions (JACOBSON; LERES; MCCANNE, 1989) to screen the packets coming from a switch port configured in "mirror" or "monitor" mode. These capture filters commonly include (*i*) network prefixes, (*ii*) ports and protocols, and (*iii*) header fields (e.g., TCP flags indicating connection control datagrams). The filtered packet stream enters the Event Engine.

Event Engine. Once in the EE, packets traverse a chain of four major processing stages: (*i*) acquisition, (*ii*) packet analysis, (*iii*) session analysis, and (*iv*) application protocol parsing. In the **acquisition** stage, traffic enters Zeek through an `IOSource`, a software interface that abstracts packet acquisition mechanisms. In the **packet analysis** stage, acquired packets enter the Packet Analysis Framework (PAF)[2]. Packets typically consist of nested Protocol Data Units (PDUs), as illustrated in Figure 2.3. A layer-n PDU contains a header, which carries an identifier ("ID") for the layer-n + 1 protocol, and a payload, which is the layer-n + 1 PDU. Within the PAF, each analyzer module parses

[2]Documentation available at <https://docs.zeek.org/en/v4.0.5/frameworks/packet-analysis.html>.

Figure 2.3 – Nesting of Protocol Data Units (PDUs)



Source: adapted from The Zeek Project (2021).

the packet header, determines the analyzer module for the encapsulated protocol, and dispatches the payload to the next analyzer (The Zeek Project, 2022). This process is then repeated until the PAF reaches the network layer. The modular nature of the PAF allows adding support to new L2-L3 protocols. In the **session analysis** stage, Zeek constructs *sessions* for the *connections* it observes[3]. For valid packets (considering IPv4, IPv6, TCP, UDP, ICMPv4, and ICMPv6), Zeek uses the addresses and port numbers (or, for ICMP, query types and their counterparts) to look up its internal session state table and, as needed, creates a new session or updates an existing one. Zeek then generates the suitable events: for instance, TCP packets lead to events such as `connection_established`, `connection_rejected`, or `connection_finished`; similarly, UDP packets result in events such as `udp_request` and `udp_reply`. When session analysis is complete, the packet payload is dispatched to the next stage. The **application protocol parsing** stage deals with the selection of a suitable application-layer protocol parser. Initially, Zeek relied on well-known port numbers to pick the suitable parser. Later, Dynamic Protocol Detection (DPD) was introduced by Dreger et al. (2006). Throughout these stages, the Event Engine forwards any generated events to the PSI.

Policy Script Interpreter. As its name suggests, the PSI loads and executes a set of *policy scripts*, written in the high-level domain-specific language ZeekScript (The Zeek Project, 2022). These scripts specify a series of *event handlers*, which provide instructions on how to respond to each event (PAXSON, 1999). Handlers can keep and manipulate state information, synthesize additional events, export data files, log relevant entries, and send alerts to the operator. Zeek comes with several pre-built frameworks and protocol analyzers and a set of policy scripts for logging, diagnostics, and notification (BURAGLIO, 2015). Modularity facilitates extensibility: Zeek users can write security policy scripts and event engine components. For instance, one can write detection rules to help identify subtle indicators of compromise related to a potential APT. Similarly, one can develop scripts and protocol analyzers to enable Zeek to respond to attack indicators relayed by other network security tools (e.g., DDoS attack detection alarms).

[3] In this context, *session* and *connection* refer to Zeek internal data structures.

2.3 Packet Analysis Acceleration Strategies

This section presents a rationale for moving NIDS-related packet analysis away from general-purpose hosts into the network itself.

IDS Performance Bottleneck. As explained by Hu, Yu and Asghar (2020), obtaining sufficiently accurate intrusion detection on a high-speed network (e.g., whose transfer rates reach 100 Gbps or more) introduces several performance challenges related to system resource allocation, packet processing speed, and packet drop rate. For instance, to fully process a single 1 Tbps flow, a traffic analysis system would have to be able to process around 100 million packets per second. However, current open-source NIDS architectures can handle from 100 thousand to 1 million packets per second per CPU core (GUPTA et al., 2018; NARAYANA et al., 2017).

OS Kernel, NICs, and NPUs. Packet analysis performance can benefit from OS kernel-level mechanisms, such as the BPF (MCCANNE; JACOBSON, 1993), the eBPF (BROUER, 2016), and the XDP (KICINSKI; VILJOEN, 2017). Nevertheless, these kernel-space approaches still demand CPU time for packet processing. As such, these strategies require large numbers of general-purpose CPUs to process terabit-scale traffic, pushing the need for large NIDS clusters. Current technology introduces the opportunity to offload traffic analysis to specialized processors, such as enterprise-grade Network Interface Cards (NICs) and programmable Network Processing Units (NPUs). While NPUs offer a noticeable performance improvement over CPUs, they must be added to each traffic collector host, which leads to a potentially undesirable coupling between components. Ideally, we should be able to offload packet processing away from the host and *into the network itself*. Such a strategy requires "smarter" network substrates, such as Software-Defined Networks (SDN) and Programmable Data Planes (PDPs), which we will discuss in the following paragraphs.

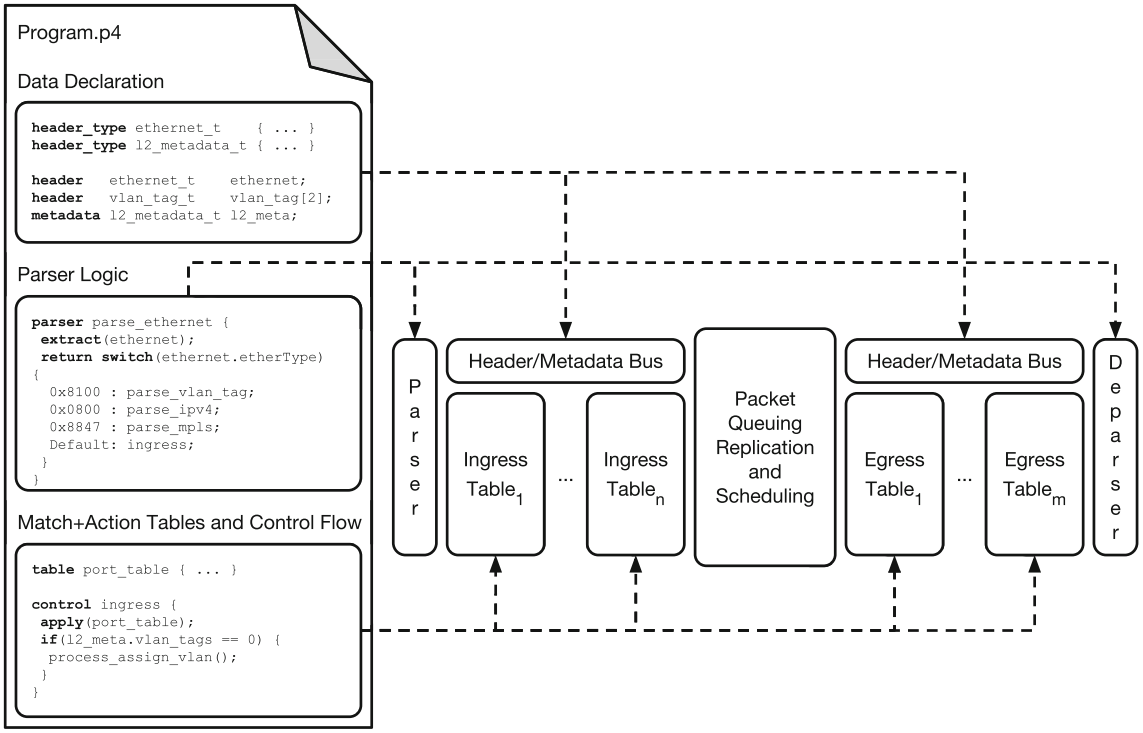
Software-Defined Networking. SDN is a relatively new paradigm that is increasingly important in both academia and industry (CORDEIRO; MARQUES; GASPARY, 2017). A key feature of SDN is the separation between the network control system - the *control plane* - and the forwarding devices such as switches and routers - the *data plane*. Such separation tears down the traditionally monolithic *vertical integration* of systems, in which both control and data planes reside in the same physical device, thereby improving network operation flexibility (KREUTZ et al., 2015). This flexibility gives SDN several advantages over traditional networking. First is the potential simplification of filtering, monitoring, routing, and troubleshooting tasks, which benefit from a global network view. Second, the reduction of the probability of inconsistent configurations among devices is

made possible by a centralized configuration base. Third, the potential for novel solutions for several classes of problems (such as intrusion detection) is facilitated by the separation between the control and data planes (DACIER et al., 2017). However, to obtain the most significant benefits from decoupling while assuring a reasonable degree of interoperability, we need open protocols, standards, and application programming interfaces (APIs), such as those provided by OpenFlow (The Open Networking Foundation, 2015).

Programmable Data Planes and P4. Despite the flexibility provided by standard SDN APIs, there remained certain shortcomings. For instance, each new version of OpenFlow introduced new features that often required replacing existing hardware due to the overdependence on fixed-function ASICs (Application-Specific Integrated Circuits). Notoriously, ASICs have a release cycle too long for the needs of research and development initiatives. As a response to these needs, *data plane programmability* emerged as a more flexible alternative to standard APIs (CORDEIRO; MARQUES; GASPARY, 2017). A prominent enabler of data plane programmability is P4 (BOSSHART et al., 2014). P4 is a high-level domain-specific language that enables *programming protocol-independent packet processors*–whence it derives its name. P4 provides an abstract forwarding model closely matching the *Reconfigurable Match Table* (RMT) architecture (BOSSHART et al., 2013). Nevertheless, P4 is target independent, enabling users to define the entire packet processing logic via software. P4 language specifications and its reference compilers define core functions that must be implemented by all compliant devices, leaving for manufacturers the task of writing platform-specific compiler backends.

According to Cordeiro, Marques and Gaspary (2017), a P4 program has three logical sections, namely (*i*) *Data Declaration*, (*ii*) *Parser Logic*, (*iii*) and *Match+Action Tables and Control Flow*, which are mapped to the abstract forwarding model (see Figure 2.4). In the Data Declaration section, we define packet header formats and specify metadata structures to be used throughout the packet processing stages. These definitions are mapped into the header/metadata bus. The Parser Logic section allows the programmer to specify rules to translate between packets and software-defined headers. These rules determine how the P4 Parser must extract header fields from incoming packets and how the P4 Deparser must serialize internal headers into outgoing packets. Finally, the Match+Action Tables and Control Flow section contains lookup tables that match on header fields to select and execute the appropriate actions. This section also specifies the control flow, which determines the table execution sequence. An action resembles a procedure in a common imperative language and allows programmers to write their packet-processing algorithms using P4 primitive instructions. This section is mapped to the Ingress Table and Egress Table pipeline stages in the abstract forwarding model.

Figure 2.4 – P4 code sections and mapping to the abstract forwarding model.



Source: Cordeiro, Marques and Gaspary (2017), adapted from Kim and Lee (2016).

Among the core P4 switch mechanisms is the *Packet Replication Engine* (PRE). The PRE is responsible for cloning packets, when directed by primitive instructions which select cloning modes. These include *ingress to egress* and *egress to egress*, which must be invoked from the corresponding pipeline control block. Before invoking an appropriate cloning primitive, one must first set up a *mirror session*, which binds an identifier to a user-selected output port. Packet mirroring is especially useful for traffic analysis tasks which require processing by external entities, e.g., a NIDS.

2.4 Existing Defenses and Related Work

Distributed Denial-of-Service (DDoS) attacks and general strategies to defend networks against them have been extensively discussed in several highly-cited surveys (e.g., Mirkovic and Reiher (2004), Peng, Leckie and Ramamohanarao (2007), Zargar, Joshi and Tipper (2013), Hoque, Bhattacharyya and Kalita (2015)). As for DDoS, there are numerous relevant studies about Advanced Persistent Threats (APTs) (e.g., Tankard (2011), Sood and Enbody (2013), Chen, Desmet and Huygens (2014), Friedberg et al. (2015), Alshamrani et al. (2019), Stojanović, Hofer-Schmitz and Kleb (2020)). It is a relevant challenge to defend networks against DDoS and APT campaigns cost-effectively, i.e., balancing requirements for performance, defense latency, and operational flexibility. One

of the main concerns of defense systems is determining an adequate placement of security functions (such as attack detection, attack source identification, and attack reaction (PENG; LECKIE; RAMAMOHANARAO, 2007)). At one extreme, ordinary switches would directly forward all traffic to off-path middleboxes for scrubbing (attack detection and filtering). At the other extreme, switches with advanced functionality would perform on-path traffic scrubbing by themselves without depending on middleboxes. In between the extremes lie architectures that distribute functionalities on both off-path mechanisms and forwarding devices.

Middlebox-based Solutions. When defense depends on middleboxes, these devices must handle the high volume of traffic that flows through the switches, which leads to performance concerns. A fine-grained approach, in which all traffic traverses a middlebox, demands significant processing and storage resources. These resource demands can be reduced by using monitoring primitives such as packet sampling (e.g., sFlow (PHAAL; PANCHEN; MCKEE, 2001)) and flow-based aggregate accounting (e.g., NetFlow (CLAISE, 2004) and OpenFlow (The Open Networking Foundation, 2015)). However, sampling and aggregate accounting also diminish defense accuracy (MOSHREF; YU; GOVINDAN, 2013). Alternatively, middleboxes based on fixed-function application-specific integrated circuits (ASICs) can achieve the desired accuracy, latency, and throughput levels. Nevertheless, this approach demands significant expenditures and leads to vendor lock-in, perpetuating the so-called *network ossification* (FEAMSTER; REXFORD; ZEGURA, 2014).

While software-based solutions running on general-purpose CPUs offer the best possible flexibility, this strategy has its own limitations. The growth in forwarding and link speeds has outpaced the increase in CPU performance (MCKEOWN, 2020; VAHDAT, 2020), which means that scaling out by adding extra general-purpose processors may become unsustainable in the long term. Numerous research efforts have sought to solve this apparent impasse by exploring the potential of software-defined networking and programmable data planes (as we innovatively do in our work) as enablers of a new generation of security services.

SDN-based Solutions. Recent investigations (e.g., Swami, Dave and Ranga (2019), Valdovinos et al. (2021)) have surveyed SDN-based DDoS defense mechanisms, some of which we analyze next. Following the OpenFlow model, Xu and Liu (XU; LIU, 2016) proposed a mechanism to detect DDoS attacks and identify the source and destination hosts. Their solution executes a machine-learning algorithm in the control plane to classify packet flows according to volume and rate asymmetry. The controller periodically fetches raw measurements from flow tables in the switches. Considering that the total storage

area for flow tables is constrained to a few thousand ternary content-addressable memory (TCAM) entries per switch, the controller dynamically adapts data aggregation granularity to optimize memory usage while enabling zooming into abnormal traffic patterns. This adaptive process requires multiple application programming interface (API) calls, resulting in a non-negligible delay (in the order of several seconds) to detect an ongoing attack and close in on the attacking sources. Aiming to avoid the multi-second delay in detection due to cross-plane operations, StateSec (BOITE et al., 2017) is a DDoS attack detection and mitigation mechanism that offloads all monitoring functions to the data plane. StateSec analyzes flow features (i.e., source and destination hosts and ports) through a set of in-switch finite-state machines based on extended OpenFlow tables (BIANCHI et al., 2014). The network controller fetches data plane-generated statistics and uses an entropy anomaly-based algorithm to detect attacks. However, this approach requires a table entry for each flow, potentially leading to memory saturation. Furthermore, StateSec performs mitigation by installing on-demand, new flow rules in the data plane to drop, queue, or deep-inspect suspect traffic. Consequently, StateSec is subject to the same delays as the method by Xu and Liu (2016).

As the proposals we mentioned exemplify, SDN allows security systems to evolve. However, there is still a dependency on frequent communication between the control and data planes to carry out traffic accounting and security function processing, which results in significant overhead and delays. Security decisions are taken in the control plane time scale, requiring hundreds of milliseconds (or even whole seconds). These delays are undesirable in DDoS defense, for which early detection and mitigation are paramount (ALCOZ et al., 2022). As opposed to these OpenFlow-based approaches, the anti-DDoS solution we present in Chapter 3 can perform policy enforcement in the time scale of nanoseconds.

Programmable Data Plane-Based Solutions. Recently-published works (e.g., Dalmazo et al. (2021), AlSabeh et al. (2022)) present systematic literature reviews about PDP-based security mechanisms. In contrast to the OpenFlow-based mechanisms we described, approaches such as OpenSketch (YU; JOSE; MIAO, 2013), UnivMon (LIU et al., 2016), and Elastic Sketch (YANG et al., 2018) fully delegate traffic accounting to the data plane. In these approaches, forwarding devices maintain summarized traffic counters in sets of hash tables, known as *sketches* (CHARIKAR; CHEN; FARACH-COLTON, 2002; KRISHNAMURTHY et al., 2003), whose values are periodically collected by the control plane. Choosing an adequate polling interval is a challenge. On one extreme, short intervals increase the network management overhead and the CPU usage in the control plane. On the other, long intervals increase the delay between the occurrence of the attack and its detection. Therefore, despite being highly accurate, these solutions are subject to

a trade-off between reaction time and management overhead. Nevertheless, sketches are a powerful low-footprint tool for calculating statistics on packet streams. The resource efficiency of sketches is further explored by SkyShield (WANG et al., 2018), which compares pairs of sketches related to pre-and under-attack conditions to infer the identity of malicious sources. However, being a CPU-based security architecture, SkyShield is unsuitable for high-throughput scenarios.

Aiming to offload even more monitoring logic (as compared to sampling and aggregate statistics) to the data plane, Sonata (GUPTA et al., 2016) provides a language for specifying packet stream filtering queries. According to operator-defined queries, programmable switches conditionally forward only the traffic of interest to external stream processors. The query language used by Sonata abstracts packet headers as tuples of field values, which can be used to define filtering and sampling rules to be executed by the data plane. Based on a similar concept, Marple (NARAYANA et al., 2017) introduces a query language and compiler that target programmable forwarding devices. It also provides a new key-value store construct that enables in-network execution of functions over aggregations of packets. Marple also uses programmable switches to measure traffic features; however, analyzing such metrics requires processing in external servers, which implies additional detection delay.

Other investigations have explored defenses against specific DDoS attack types by implementing prevention techniques on programmable networks. For instance, the most common technique involves intercepting session initiation packets in the data plane and responding to them with challenges to authenticate client hosts (via the three-way handshake) before allowing them to contact servers inside a network (SHIN et al., 2013; AFEK; BREMLER-BARR; SHAFIR, 2017). However, this technique penalizes the connection time of all clients, even without an ongoing attack in the network. Seeking to avoid such delays and thus improve user quality of experience, Tavares and Ferreto (2019) proposed requiring authentication only of clients trying to connect to servers that are under attack. To pinpoint these servers, the authors estimate application-layer statistics (e.g., the number of half-opened TCP sessions) by implementing count sketches in programmable switches. Similarly, Paolucci et al. (2019) developed a P4-based method to detect TCP SYN flood and port-scanning attacks at edge switches. In their mechanism, upon the detection of an attack, packets identified as malicious can be dropped or steered for further inspection on an external stateful firewall. Another example of session-based defense is FrameRTP4 (BONFIM et al., 2020), whose anti-DDoS component uses *count-min sketches* (CORMODE; MUTHUKRISHNAN, 2005) to find heavy-hitter flows and access-control lists to block traffic related to these flows.

The main limitation of the prevention techniques we mentioned is their limited applicability. They resemble signature-based systems, observing specific field values, and detecting particular types of attacks, such as protocol exhaustion at the transport or application layers. The sketch-based approaches diminish monitoring overhead by delegating accounting to data plane devices, but they still require control plane decisions. In turn, streaming analytics (as Sonata and Marple perform) can go beyond aggregate statistics, but security functions also depend on external stream processors. To sum up, both solution classes require frequent interaction between the control and the data plane, hindering attack detection timeliness. Unlike the PDP-based approaches we analyzed, EUCLID can be categorized as an anomaly detection system, detecting (and, very importantly, mitigating) different variations of volumetric DDoS attacks. Moreover, this process can be entirely executed in the data plane at the network line rate.

Architectures and Frameworks. We have recently seen the emergence of research on security architectures and frameworks that approach in-network defense more abstractly. For instance, Xing, Wu and Chen (2019) propose FastFlex, an architecture that implements a "multimode" data plane whose security mechanisms are enabled only when needed. Under normal conditions, switches forward data according to standard routing policies without additional latency. When under attack, switches engage their defense mechanisms to mitigate the threat. Poseidon, initially proposed in Zhang et al. (2020) and later extended in Li et al. (2021), introduces a high-level language comprising a set of instructions for network monitoring and traffic management. Its users can specify defense strategies and security policies in this language and deploy the resulting configurations on programmable data planes. The Poseidon runtime environment can reconfigure network devices so they better adapt to changes in attack characteristics which may demand different detection strategies.

We acknowledge these solutions and consider their contributions as enablers for the composition of security mechanisms that implement a range of defense techniques. While the authors present use cases, their focus is not on developing novel defense techniques. Instead, they show how existing techniques can be mapped to constructs provided by their frameworks. We emphasize that generic approaches, such as FastFlex and Poseidon, while good for flexibility, present similar problems to those that occur with CPU-based solutions. For instance, in the case of Poseidon, one needs a large amount of memory to store all the sketches demanded. In contrast, EUCLID memory requirements are minimal.

Table 2.1 summarizes the preceding discussion about related works in terms of solution category, applicability, placement (for traffic accounting and attack detection), management traffic overhead, memory footprint, and reaction timescale.

Table 2.1 – Related Work Summary

Reference	Category	Applicability	Traffic Accounting	Attack Detection	Management Traffic	Memory Footprint	Reaction Timescale
Xu and Liu (2016)	SDN	DDoS	DP[4]	CP[5]	High[6]	Low[7]	Seconds[8]
Boite et al. (2017)	SDN	DDoS	DP	CP	Medium[9]	High[10]	Seconds[11]
Yu, Jose and Miao (2013)	PDP	DDoS	DP	CP	Medium[12]	Low	Milliseconds[13]
Liu et al. (2016)	PDP	DDoS	DP	CP	Medium	Low	Milliseconds
Yang et al. (2018)	PDP	DDoS	DP	CP	Medium	Low	Milliseconds
Wang et al. (2018)	CPU	Monitoring	CP	CP	High[14]	Low	Milliseconds
Gupta et al. (2016)	PDP	Monitoring	DP+CP	CP	Medium	Low	Seconds
Narayana et al. (2017)	PDP	Monitoring	DP+CP	CP	Medium	Low	Seconds
Shin et al. (2013)	SDN	DDoS	DP	DP	Low	Low	Microseconds[15]
Afek et al. (2017)	PDP	DDoS	DP	DP	Low	Low	Microseconds
Tavares and Ferreto (2019)	PDP	DDoS	DP	DP	NA	Low	Microseconds
Paolucci et al. (2019)	PDP	DDoS	DP	DP	NA	High	Nanoseconds
Bonfim et al. (2020)	PDP	DDoS	DP	DP[16]	Medium	High[17]	Nanoseconds
Xing, Wu and Chen (2019)	PDP	DDoS	DP	DP	Low	NA	Nanoseconds
Li et al. (2021)	PDP	DDoS	DP	DP	Medium	High	Nanoseconds
Zeek (PAXSON, 1999)	CPU	Security	CP	CP	High	High	High
This Work (Chapter 3)	PDP	*DDoS*	DP	DP	*Negligible*	*Low*	*Low*
This Work (Chapter 4)	PDP	*Security*	DP	*NA*	*Low*	*Low*	*NA*

Towards a General Solution. So far, we discussed several *ad-hoc* security solutions. We observed a trend towards moving increasingly sophisticated functionality to "smarter" network devices. The programmable data plane is a versatile, powerful, and efficient building block for many of the security solutions we mentioned. Moreover, recently-developed architectures and frameworks show the beginning of a trajectory towards integrating "traditional" CPU-based solutions (especially in the case of intrusion detection) into the network itself. However, there is still a relevant research gap. On one side, the functionalities of existing solutions are somewhat limited compared to a full-fledged solution against DDoS attacks or APTs. On the other side, we have frameworks that do not leverage the full potential of PDPs. What adds depth to such a wide gap is the duplication of efforts in writing, rewriting, and debugging fundamental code (e.g., protocol parsers, data structures, and routing functions) for a plethora of different devices. We seek to explore how to bridge this gap by developing an automated method to translate security policies into code executed on arbitrary hardware.

[4]Data Plane.
[5]Control Plane.
[6]CP fetches raw measurements.
[7]Memory usage reduced by adaptive granularity.
[8]CP-DP interaction is required to adjust granularity.
[9]CP fetches statistics.
[10]DP requires one table entry for each flow.
[11]CP must install flow rules for mitigation.
[12]Management traffic volume depends on the desired detection accuracy.
[13]CP-based detection.
[14]CP needs to observe full flows.
[15]Prevention requires connection authentication techniques.
[16]Hits on DP-based ACLs are assumed malicious, but the ACLs are populated by the CP.
[17]Needed for the ACLs.

3 A FULLY IN-NETWORK, P4-BASED APPROACH FOR REAL-TIME DDOS ATTACK DETECTION AND MITIGATION

In this chapter, we present EUCLID (ILHA et al., 2021), our approach for fully in-network DDoS attack detection and mitigation. Our proposal builds upon and extends the work by Lapolli, Marques and Gaspary (2019), which introduced the DDoS attack detection mechanism used by EUCLID.

3.1 Foundations of DDoS Attack Detection and Mitigation

This section provides the groundwork for our proposed in-network DDoS defense mechanism. We begin by describing the attack scenario and the threat model we address (§ 3.1.1). Next, we present the foundations of the attack detection strategy (§ 3.1.2). Finally, we explain the underpinnings of the mitigation technique (§ 3.1.3).

3.1.1 Attack Scenario and Threat Model

The term *distributed denial-of-service* (DDoS) refers to a broad class of attack strategies that seek to degrade the quality or disrupt the availability of services on the Internet (as introduced in Section 2.1). In this work, we consider a threat model whose attack vector is a large set of globally-distributed computers (e.g., a botnet) controlled by an attacker, sending illegitimate service requests to a single target host (e.g., a web server). Moreover, the attacker uses spoofing techniques in an attempt to evade defense measures.

The attack scenario just described makes it challenging to deploy detection and mitigation mechanisms close to the attack sources—since the malicious traffic originates in several different locations, the widespread adoption of source-based defenses would require collaboration among several Internet service providers around the globe. Conversely, defenses installed on the victim's infrastructure may not be effective against the aggregated malicious traffic, which may have already saturated on-path and local network resources. Thus, we expect our proposed mechanism to be deployed in an intermediate position within the autonomous systems (ASes) that are closest to the victim. These transit ASes typically have high-throughput forwarding devices and a privileged vantage point. Such characteristics facilitate traffic scrubbing in a timely fashion before service degradation or disruption occurs. In order to prevent congestion of lower-capacity links, our mechanism should be installed on border routers, where it can process inter-AS traffic.

3.1.2 Traffic Characterization and Anomaly Detection

Strict performance constraints make it a significant challenge to engineer hardware for programmable switches at a reasonable cost. As a result, current programmable data planes impose strict constraints on time and memory, as well as make available only a reduced set of instructions (BOSSHART et al., 2013). Hence, defense mechanisms built upon this type of device must be simultaneously memory-efficient and implementable in terms of the existing programming primitives. We advocate that concepts and constructs that have already been successfully applied in the context of traffic flow analysis can help meet these goals. This is the case of entropy measurements (e.g., Liu et al. (2016), Boite et al. (2017), Yang et al. (2018) and sketches (e.g., Yu, Jose and Miao (2013), Liu et al. (2016), Yang et al. (2018), Wang et al. (2018)). In this subsection and the next, we discuss these concepts and their application in our proposed solution.

From an information-theoretic standpoint, a DDoS incident induces anomalies in the *Shannon entropy* (SHANNON, 1948) of the IP address frequency distribution. These anomalies result from increases in the total number and spread of source addresses (both legitimate and malicious) and from the concentration of traffic towards the destination address of the target host—which leads to a skewed traffic profile (HOQUE; BHATTACHARYYA; KALITA, 2015). Thus, by accurately distinguishing between normal and abnormal traffic patterns, it is possible to detect DDoS attacks reliably (LAKHINA; CROVELLA; DIOT, 2005; BHUYAN; BHATTACHARYYA; KALITA, 2015).

In this work, traffic characterization and anomaly detection begin with *frequency measurements*: our research group's attack detection mechanism (LAPOLLI; MARQUES; GASPARY, 2019) aggregates incoming packets in fixed-length *observation windows* (OWs), each containing m packets. During each OW, our mechanism counts the number of occurrences of every distinct source and destination IP address. Considering X the set of IP addresses within a total of m packets, and $f_1, f_2, ..., f_N$ (where $N = |X|$) the frequencies of each distinct address, the Shannon entropy of X, denoted by $H(X)$, is defined by

$$H(X) = \log_2(m) - \frac{1}{m} \sum_{x=1}^{N} f_x \log_2(f_x), \tag{3.1}$$

where the summation is the *entropy norm* of X, defined by

$$S(X) = \sum_{x=1}^{N} f_x \log_2(f_x). \tag{3.2}$$

As such, Equation 3.1 can be rewritten as

$$H(X) = \log_2(m) - \frac{1}{m}S(X), \tag{3.3}$$

which highlights the negative relation between the entropy norm and the entropy itself. The minimum entropy $H(X) = 0$ occurs when all addresses are the same such that $S(X) = m\log_2(m)$. Dispersed distributions result in higher entropy values reaching the maximum $H(X) = \log_2(m)$ when all addresses are distinct, i.e., $S(X) = 0$.

Our detection mechanism calculates entropies separately for the sets of source and destination IP addresses. During a DDoS attack, it is expected that the entropy of the set of source IP addresses increases as malicious packets introduce new values to the frequency distribution. Conversely, it is expected that the entropy of the set of destination IP addresses decreases with the victim becoming more frequent as a destination. On the one hand, these variations are only observable when the total number of packets (m) encompasses a sufficiently robust representation of the address frequency distributions. However, large values of m lead to higher attack detection delays. On the other hand, small values of m may render attack-related changes indistinguishable from short-term fluctuations of legitimate traffic. The mechanism seeks to address this trade-off by scaling the entropy measurements of the source and the destination IP addresses considering a preset value of m (the entropy is proportional to $\log_2(m)$).

Anomaly-based attack detection has the advantage of being able to uncover attacks of unusual behavior and various intensities, although typically requiring a bootstrapping (or training) phase. In order to find relevant anomalies, our mechanism first needs to go through a training phase, during which it characterizes normal (not anomalous) traffic. The training consists of monitoring the network for a certain number of successive *observation windows* (OWs), independently calculating source and destination IP address entropies for each OW, and summarizing these values in terms of indices of central tendency and dispersion. Our mechanism uses exponentially-weighted moving averages (EWMAs) and mean deviations (EWMMDs) (ROBERTS, 1959) to summarize measurements. The source and destination entropy EWMAs are defined by:

$$M_{src,n} = \alpha H_{src,n} + (1-\alpha)M_{src,n-1}, \tag{3.4a}$$

$$\text{with} \quad M_{src,1} = H_{src,1}, \text{ and}$$

$$M_{dst,n} = \alpha H_{dst,n} + (1-\alpha)M_{dst,n-1}, \tag{3.4b}$$

$$\text{with} \quad M_{dst,1} = H_{dst,1},$$

where $M_{src,n}$ and $M_{dst,n}$ are the source and destination entropy EWMAs at the observation window $n \in \mathbb{N}^*$. The factors $H_{src,n}$ and $H_{dst,n}$ are, respectively, the source and destination address entropies at OW n. The value $\alpha \in (0, 1)$ is the *smoothing coefficient*—a parameter that allows us to filter short-term fluctuations while giving prominence to the most recent entropy measurements. The averages are initialized with the first entropy measurements.

The source and destination EWMMDs are defined by:

$$D_{src,n} = \alpha|M_{src,n} - H_{src,n}| + (1 - \alpha)D_{src,n-1}, \tag{3.5a}$$
$$\text{with} \quad D_{src,1} = 0, \text{ and}$$
$$D_{dst,n} = \alpha|M_{dst,n} - H_{dst,n}| + (1 - \alpha)D_{dst,n-1}, \tag{3.5b}$$
$$\text{with} \quad D_{dst,1} = 0,$$

where $D_{src,n}$ and $D_{dst,n}$ are the source and destination EWMMDs at the OW n. The factors $M_{src,n}$ and $M_{dst,n}$, respectively, are the source and destination EWMAs at OW n. The value $\alpha \in (0, 1)$ is the smoothing coefficient. The mean deviations are initialized with zero.

After having obtained a model of normal traffic under safe conditions, we use its EWMAs and EWMMDs to decide whether an entropy measurement is anomalous, in which case the mechanism triggers a DDoS attack alarm; otherwise, it updates the traffic model. An entropy anomaly occurs when any of the following inequalities hold:

$$H_{src,n} > M_{src,n-1} + kD_{src,n-1}, \tag{3.6a}$$
$$H_{dst,n} < M_{dst,n-1} - kD_{dst,n-1}. \tag{3.6b}$$

The value k is a configurable parameter called the *sensitivity coefficient*, which scales the detection threshold. Since k multiplies the index of dispersion, its effect is directly proportional to the variability in traffic patterns. Lower values of k allow the detection of subtler attacks, at the cost of a lower *specificity* (i.e., a higher number of legitimate fluctuations incorrectly interpreted as attacks). Conversely, higher values of k result in higher statistical specificity at the cost of letting less-relevant attacks remain unnoticed. It is the responsibility of the network administrators to choose k values according to the intended level of accuracy (i.e., high sensitivity and high specificity). Experimental results (LAPOLLI, 2019) indicate that when k is in the ideal operating range, our mechanism can accurately (i.e., $\approx$ 98% sensitivity and $\approx$ 90% specificity; see § 3.3.3) detect and signal the occurrence of DDoS attacks. Such a high accuracy allows us to use these detection alarms as trustworthy inputs for our mitigation mechanism—which we discuss in the following subsection.

3.1.3 Inferring Intent from Frequency Variation Anomalies

We use the output of the anomaly detection component as a trigger to (*i*) identify which packet sources are the likely culprits of an attack and (*ii*) apply a suitable countermeasure. We design our mechanism following the observation that Shannon entropy anomalies are likely caused by addresses whose frequencies have excessively diverged from baseline measurements (PENG; LECKIE; RAMAMOHANARAO, 2007).

The difference between the frequency variations of legitimate and malicious addresses is significant enough to allow the accurate classification of packets. By finding an adequate threshold, we can obtain acceptable results for *sensitivity* (i.e., the true-positive rate or proportion of correctly-identified malicious packets) and *specificity* (i.e., the true-negative rate or proportion of correctly-identified legitimate packets).

Since we are interested in comparing safe and unsafe network conditions, we need to keep track of these. We consider a network *safe* when none of its nodes is undergoing a detectable DDoS attack; otherwise, we consider the network *unsafe*. When the anomaly detection module indicates that the network is unsafe, the mechanism enters a *defense readiness* mode, in which it will remain until the network has been safe for a predetermined *cooldown interval*—which avoids premature defense de-escalation.

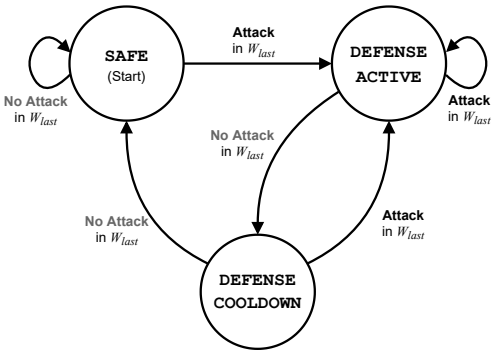
We model defense readiness as a finite state machine (FSM) that controls the operation of the attack mitigation mechanisms. At the end of each observation window, the mechanism updates the FSM state according to the safety of the network in W_{last} (the OW whose accounting has just finished). Figure 3.1 illustrates the defense-readiness FSM. It starts in the `SAFE` state, in which detection is active, but mitigation is dormant. Whenever an attack is detected in W_{last}, the FSM transitions to the `DEFENSE ACTIVE`, in which mitigation is active. The FSM remains in the `DEFENSE ACTIVE` until no attack is detected in $Wlast$, in which case there is a transition to `DEFENSE COOLDOWN`. Once in cooldown, mitigation remains active; however, if no attack has been detected for a predetermined number of OWs (not shown in the figure), the machine transitions to `SAFE`.

Once in any of the `DEFENSE` states, the defense pipeline calculates *frequency variation* for each incoming packet. Frequency variation is denoted by V and defined by

$$V = V_{dst} - V_{src}, \tag{3.7}$$

where V_{src} and V_{dst} measure the changes in frequencies of the source (src) and destination (dst) addresses of the packet. During an attack, we expect relevant changes in V_{src} and V_{dst}. The intuition behind this is that variations related to legitimate traffic will be proportional

Figure 3.1 – Defense-Readiness State Machine.



Source: the author (2022).

for both source and destination addresses. However, for the malicious traffic, this pattern changes: while the total traffic grows, the frequencies of the legitimate source addresses will vary disproportionately to the frequencies of the malicious addresses. Relative frequencies for source hosts tend to decrease (many hosts). Relative frequencies for attack targets tend to increase. By subtracting V_{src} from V_{dst}, we expect to obtain larger values of V for malicious packets than for legitimate ones. We calculate the variations between the last observation window (W_{last}) and the observation window used as a baseline of a safe network condition (W_{safe}). The values of the V-terms are given by

$$V_{src} = f_{src,last} - f_{src,safe} \text{ and} \tag{3.8a}$$

$$V_{dst} = f_{dst,last} - f_{dst,safe}, \tag{3.8b}$$

where $f_{src,last}$ and $f_{src,safe}$ are the frequencies of src in W_{last} and W_{safe}, respectively. Similarly, $f_{dst,last}$ and $f_{dst,safe}$ refer to the destination address.

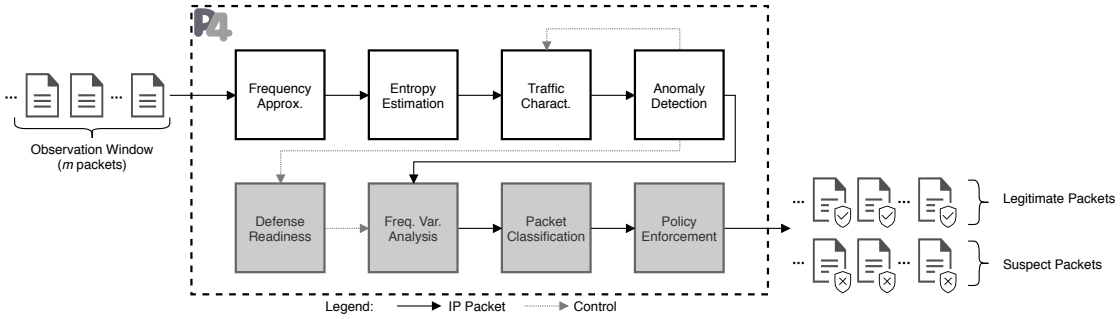
After calculating the frequency variation, classification follows a mitigation threshold t and annotates (by setting metadata) each packet as *legitimate* or *suspect*:

$$V \leq t \Rightarrow \text{packet is legitimate;} \tag{3.9a}$$

$$V > t \Rightarrow \text{packet is suspect.} \tag{3.9b}$$

Once packets have been annotated, the mechanism is ready to enforce an operator-configurable security policy, such as *discarding*, *throttling*, or *detouring*. Discarding is straightforward: we immediately drop the suspect packet. Throttling consists of forwarding suspect packets to a different egress queue, with a limited number of entries and a predefined dispatch rate, thus limiting the volume of suspect traffic allowed to reach their target. Detouring enables various scenarios: packets forwarded to a different path can undergo, for instance, stateful, light, or deep inspection.

Figure 3.2 – Anti-DDoS Attack Mechanism Top-Level Scheme.



Source: the author (2020).

3.2 Our Design for In-Network DDoS Attack Detection and Mitigation

In this section, we discuss the design and implementation of the EUCLID packet processing pipeline, which materializes the defense strategy whose foundations we presented in the previous section.

We implement EUCLID in $P4_{16}$ (BOSSHART et al., 2014). Using this language to specify our mechanism, we facilitate its deployment on compatible P4-programmable switches suitable for high-speed, high-throughput packet forwarding. Notwithstanding the versatility of the language, P4 programs must operate under strict constraints, such as a reduced set of instructions and a limited amount of memory—in the order of megabytes of static random-access memory (SRAM) and kilobytes of ternary content-addressable memory (TCAM). Consequently, we need to consider these architectural constraints to develop our real-time anti-DDoS defense. Hence, we designed EUCLID with resource-efficiency in mind: its operation requires, for each 1-Gbps link, less than 80 KB of SRAM and 2 KB of TCAM. We discuss the requirements for scaling up our mechanism to meet the needs of faster links in § 3.3.3.2.

We present an overview of EUCLID in Figure 3.2 . The top row portrays the attack detection components proposed by Lapolli, Marques and Gaspary (2019), published as open-source software, which we integrated in our own approach. Throughout its operation, EUCLID partitions the stream of incoming packets into fixed-size *observation windows* (OWs). During the processing of each OW, the *frequency approximation* component tallies the frequency (number of occurrences) of every source and destination IP address (§ 3.2.1.1). Next, these frequencies are used as inputs by the *entropy estimation* logic (§ 3.2.1.2). At the end of the OW, the *traffic characterization* component uses the entropy estimates to build and update a statistical model of normal traffic conditions (§ 3.2.1.3). Then, the *anomaly detection* stage employs the traffic model to check for abnormal changes in IP address entropies, in which case it issues attack alarms (§ 3.2.1.4).

Also, in Figure 3.2, the bottom row lays out the *attack mitigation* components we presently introduce. Attack alarms trigger transitions in a *defense-readiness* state machine, which activates and coordinates the operation of the remaining components (§ 3.2.2.1). When attack mitigation is enabled, every packet undergoes three stages. First, EUCLID analyzes the address accounting history to measure the *frequency variation* of the IP addresses (§ 3.2.2.2). Next, the *packet classification* section decides whether there have been unwarranted changes in frequency, in which case it labels packets as suspects (§ 3.2.2.3). Last, *policy enforcement* applies network-operator-defined rules to determine the adequate destination for the packet (§ 3.2.2.4). In the remainder of this section, we detail the implementation of each attack detection and mitigation component.

3.2.1 Attack Detection

Our research group's anomaly-based attack detection strategy depends on traffic characterization, which requires measuring the Shannon entropies of the sets of source and destination IP addresses in each OW (§ 3.1.2). The entropies, in turn, depend on statistics about the frequency of every source and destination IP address. Keeping this kind of accounting is potentially computing- and storage-intensive, especially when exact measurements are required. To make it feasible to obtain these quantities under the processing constraints of a programmable data plane, our detection components build these statistics through the *frequency approximation* and *entropy estimation* pipeline, which we detail next. Figure 3.3 illustrates the entire pipeline.

3.2.1.1 Frequency Approximation

We approximate address frequencies by employing count-sketches (CHARIKAR; CHEN; FARACH-COLTON, 2002), which we briefly discussed in Section 2.4. Count-sketches are data structures that provide a compressed representation of frequency tables of *events* in data streams. The frequency approximation component generates two separate events upon the arrival of each packet: one for the source IP address and one for the destination. The two resulting data streams consist of all the events within a given observation window. The total size needed for a count-sketch to represent events without compression grows sublinearly in m (CORMODE; MUTHUKRISHNAN, 2005). Nonetheless, sketches provide unbiased frequency estimates within parameterizable probability and tolerances.

Formally, a count-sketch is an abstract data type represented by a tuple (C, X, F_h, F_g) and two operations, UPDATE and ESTIMATE, defined as follows. Let X be the set of all

possible IP addresses and C be a two-dimensional matrix of counters with depth d and width w (i.e., $C \in \mathbb{Z}^{d \times w}$), where $C_{i,j}$ denotes the counter at row i and column j (see Figure 3.3). We define two sets of independent hash functions $F_h = \{h_1, ..., h_d\}$ and $F_g = \{g_1, ..., g_d\}$, where each ordered pair $(h_i, g_i) \in F_h \times F_g$ is associated with a matrix row $i \in \{1, ..., d\}$. All hash functions take an IP address $x \in X$ as a parameter. Hash function h_i maps addresses to column numbers in row i (i.e., $h_i : X \mapsto \{1, ..., w\}$). Hash function g_i determines whether the counter $C_{i,h_i(x)}$ should be incremented or decremented (i.e., $g_i : X \mapsto \{-1, 1\}$).

The count-sketch operations are summarized in Algorithms 1 and 2. UPDATE(C, x) counts an occurrence of x by updating exactly one entry in each of the d depth levels of the sketch C. ESTIMATE(C, x) returns an estimate of the frequency count of x, which we denote as $\hat{f}_x$ (see Figure 3.3).

Algorithm 1 Count-sketch UPDATE Operation

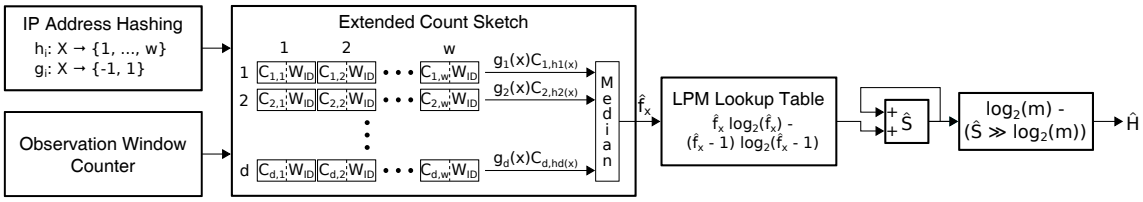
Input: C, x ▷ C: count-sketch; x: IP address.
1: **for** $i \in [1, 2, ..., d]$ **do** ▷ For each sketch row.
2: $\quad C_{i,h_i(x)} \leftarrow C_{i,h_i(x)} + g_i(x)$
Output: C

Algorithm 2 Count-sketch ESTIMATE Operation

Input: C, x ▷ C: count-sketch; x: IP address.
1: $\hat{f}_x \leftarrow median(\{g_i(x)C_{i,h_i(x)} \mid \forall i \in [1, 2, ..., d]\})$
Output: $\hat{f}_x$

The count-sketch uses the hash functions g_i to treat h_i collisions for multiple distinct IP addresses. Expectedly, when collisions occur, some addresses will increase the counter, and others will decrease it, which would result in inconsistent estimates. However, when considering all d counters for a given IP address, counters whose values are affected by collisions become outliers. By calculating the median (which eliminates outliers) of the values stored in all rows, the count-sketch avoids generating biased frequency estimates. It is essential to notice the need to implement a median operator whose number of inputs equals the sketch depth (d). Consequently, d directly influences the complexity of the median calculation, which requires $\mathcal{O}(d^2)$ execution steps to compare all inputs. In P4, we specify the count-sketch matrices as stateful registers. We implement IP address hashing operations as custom hash functions. In our design, hash functions are homomorphic to $h(x) = (a_i x + b_i) \bmod p$, where a_i and b_i are co-prime coefficients, and p is a prime number. This class of functions is suitable for deployment in programmable data planes, as previous work demonstrates (SIVARAMAN et al., 2017).

Figure 3.3 – Entropy Estimation Pipeline.



Source: Lapolli, Marques and Gaspary (2019).

Our design requires calculating independent frequency approximations for each OW, which mandates resetting all count-sketches before the first usage within a window. To avoid bursty processing overheads, we resort to *extended count-sketches*. In our implementation, we associate an additional register to each sketch entry (see Figure 3.3), which stores the index of the OW in which it was last updated (W_{ID}). This index, in turn, comes from a P4 stateful counter. Whenever the mechanism reads an extended count-sketch, outdated entries are presumed zero and updated accordingly.

Once the detection mechanism has processed an incoming packet and updated its frequency approximation, it can proceed to the entropy estimation step.

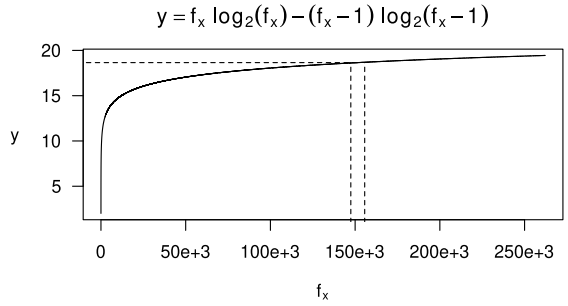
3.2.1.2 Entropy Estimation

Considering P4 has no support for floating-point arithmetic, our research group's solution stores and handles measurements in a fixed-point format, which allows obtaining fractional precision using only integer operations. Given that P4 also lacks instructions to calculate binary logarithms, we need to simplify Equation 3.1 (p. 36). For the first term, we set the observation window size m to a fixed (parameterizable) value so that $\log_2(m)$ becomes a constant. As a result, the real-time entropy estimation processing requires only the calculation of the second term, i.e., the entropy norm, as we explain next.

Entropy norm estimation. The second term of Equation 3.1 is given by $S(X) = \sum_{x=1}^{N} f_x \log_2(f_x)$. This term is a function of the frequencies of each unique IP address observed in the window (recall that the calculations are separate and independent for source and destination IPs). After the pipeline reads an IP address and updates its approximate frequency $\hat{f}_x$, it is ready to compute the corresponding term in the entropy norm estimate $\hat{S}$ (to simplify notation, we omit the parameter X). As each IP address reappears in each OW, the pipeline updates $\hat{S}$ by adding to it the difference between the newly-computed term and its previous value. This is done for $\hat{f}_x > 1$, as follows:

$$\hat{S} \leftarrow \hat{S} + \underbrace{\hat{f}_x \log_2(\hat{f}_x)}_{\text{newly-computed term}} - \underbrace{(\hat{f}_x - 1) \log_2(\hat{f}_x - 1)}_{\text{previous term value}}. \tag{3.10}$$

Figure 3.4 – LPM lookup table pre-computed function. The dashed lines illustrate how f_x values can be aggregated to a single table entry with reduced approximation error.



Source: Lapolli, Marques and Gaspary (2019).

To calculate Equation 3.10, we define a pre-computed function (PCF), which we implement as a longest prefix match (LPM) lookup table. Our LPM table contains values for $\hat{f}_x \log_2(\hat{f}_x) - (\hat{f}_x - 1)\log_2(\hat{f}_x - 1)$. Unlike an exact lookup table, which would require an entry for each domain value, the LPM maps variable-length intervals of domain values to a single entry. As an LPM is typically implemented in a switch by a ternary content-addressable memory (TCAM), we eliminate the need for real-time multi-step operations, replacing them with a single-step TCAM lookup.

We plot our PCF in Figure 3.4. The dashed lines represent the aggregation of the domain interval $[147\,456, 155\,647]$ to a single entry whose image is $y = 18.65214$. In this case, the maximum approximation error is ≈ 0.04 when $f_x = 147\,456$. In general, the magnitude of the error is directly proportional to dy/df_x, i.e., $\log_2(f_x) - \log_2(f_x - 1)$, whose value decreases as f_x grows. Consequently, aggregation intervals can be larger for higher frequencies. Lapolli (2019) proposes an algorithm to populate the LPM while meeting an adequate trade-off between entry count and maximum error.

Throughout the operation of our P4-based design, every incoming packet x triggers UPDATE(C, x) and $\hat{f}_x \leftarrow$ ESTIMATE(C, x) (for source and destination IPs). Our mechanism uses $\hat{f}_x$ as a key to look up the increments to the entropy norms. When our mechanism reaches the end of each observation window, it uses $\hat{S}$ to estimate the entropy $\hat{H}$ (for conciseness, we omit the parameter X), as shown below.

Entropy measurement. Seeking to further diminish the processing requirements, we constrain the operation window size m to a fixed power of two. Thus, $\log_2(m)$ yields an integer constant, which makes it possible to calculate $\hat{S}/m$ as an arithmetic shift. The resulting expression for the entropy estimate is:

$$\hat{H} \leftarrow \log_2 m - (\hat{S} \gg \log_2 m), \tag{3.11}$$

where $\gg$ denotes an arithmetic shift. We store the value $\log_2(m)$ in a register to allow the parameterization of m at runtime.

In a recently-published work, Ding, Savi and Siracusa (2020) introduced algorithms to calculate logarithmic functions and entropy estimates in programmable data planes. In contrast to our work, their approach does not require a TCAM-backed pre-computed function (PCF). Their solution requires additional processing steps for each packet—which potentially implies allocating more pipeline stages. The analysis of the suitability of this solution as a substitute for our PCF-based approach (regarding the processor time and memory space trade-off) exceeds the scope of our current work.

After the entropy estimation phase finishes, the switch enters the subsequent functional stages—traffic characterization and anomaly detection.

3.2.1.3 Traffic Characterization

We summarize entropy measurements in terms of their EWMAs and EWMMDs (§ 3.1.2, Equations 3.4a, 3.4b, 3.5a, and 3.5b). As in the case of entropy estimation, we use fixed-point notation. We choose different representations to allow for sufficient numeric precision. For instance, we represent entropy measurements as 28 integer and 4 fractional bits, and we store moving averages and deviations as 14 integer and 18 fractional bits. For the smoothing coefficient α, eight fractional bits are sufficient. In order to preserve precision, we take special care to specify operation order and binary radix point alignment.

3.2.1.4 Anomaly Detection

As in the traffic characterization component, attack detection uses fixed-point arithmetic to calculate source and destination thresholds. The sensitivity coefficient k is represented with five integer and three fractional bits. We check whether the last entropy measurements exceed these thresholds according to Equations 3.6a and 3.6b. If, and only if, both entropy estimates fall within the dynamically-calculated thresholds, we update the traffic model. Conversely, if an anomaly is detected, the switch records such occurrence by setting a packet metadata field. This flag enables the generation of a signaling packet and triggers a state transition in the attack mitigation mechanism, which we will discuss next.

3.2.2 Attack Mitigation

The attack mitigation mechanism, formalized together with its detection counterpart in Algorithm 4 (p. 48), follows the principles we discussed in § 3.1.3. The anomaly detection component triggers state transitions in the defense-readiness finite-state machine (FSM). The FSM, in turn, directs the operation of all the attack mitigation logic. Next, we discuss the implementation of the state machine and the remaining security stages.

3.2.2.1 Defense Readiness

In this stage, executed once for each observation window (OW), EUCLID checks the attack alarm and the defense-readiness (DR) state to perform a conditional transition (Algorithm 4, Lines 29-34). If the attack alarm metadata field is set, the state machine transitions to `DEFENSE ACTIVE`, as explained in § 3.1.3. In contrast, if the attack alarm flag is inactive, there are two possibilities: (*i*) when in `DEFENSE ACTIVE`, DR moves to `DEFENSE COOLDOWN`, in which it remains for an additional predetermined number of observation windows (in our implementation, we set this number to one); (*ii*) when already in `DEFENSE COOLDOWN`, DR transitions back to `SAFE`. After DR executes, its resulting state lasts at least until the end of the next OW, when new checks and possibly new transitions will occur. In both `DEFENSE` states, EUCLID submits every incoming packet to the subsequent stages—frequency variation analysis, packet classification, and policy enforcement, which we discuss next.

3.2.2.2 Frequency Variation Analysis

In this component, we follow the observation that entropy anomalies are more likely due to excessive occurrences of the IP addresses of the attackers, whose frequencies have varied the most between a reference, baseline OW, and the OW in which we detected an attack (§ 3.2.1.4). By uncovering these highly-divergent addresses, we can identify attack sources. In this manner, we solve the source identification problem by finding excessive variations. Since EUCLID already performs frequency approximation to calculate entropies, we benefit from that functionality to pinpoint, accurately, malicious traffic sources.

This step also uses count-sketches to obtain approximate quantities, similar to the frequency approximation stage. However, in this component, we process historical data, i.e., counts obtained in W_{last} and W_{safe} (respectively, the previous observation window and the last OW during which the network was safe, as defined in § 3.1.3). We store this data in four additional sketches: two for source addresses, two for destination addresses, i.e., $C_{src,last}$, $C_{dst,last}$, $C_{src,safe}$, and $C_{dst,safe}$. We further extend the count-sketch by adding to it the COPY operation (Algorithm 3). COPY iterates on all depth levels of the target and origin sketches to copy counters associated with a given IP address x.

Algorithm 3 Count-sketch COPY Operation

Input: $CTARGET, CORIGIN, x$ ▷ $CTARGET, CORIGIN$: count-sketches; x: IP address.
1: **for** $i \in [1, 2, ..., d]$ **do** ▷ For each sketch row.
2: $CTARGET_{i,h_i(x)} \leftarrow CORIGIN_{i,h_i(x)}$
Output: $CTARGET$

Algorithm 4 Attack Detection and Mitigation

Input: P ▷ Packet headers and metadata

Frequency Approximation (§ 3.2.1.1).

1: **for** $i \in [1, 2, ..., d]$ **do** ▷ For each sketch row
2: $h_{src}(i) \leftarrow h_i(P.src)$
3: $g_{src}(i) \leftarrow g_i(P.src)$
4: **if** $C_{src}(i, h_{src}(i)).W_{ID} \neq W_n$ **then**
5: **if** $W_n > 1$ **and** $DR_{state} =$ `SAFE` **then**
6: $C_{src,safe}(i, h_{src}(i)) \leftarrow C_{src,last}(i, h_{src}(i))$
7: $C_{src,last}(i, h_{src}(i)) \leftarrow C_{src}(i, h_{src}(i))$
8: $C_{src}(i, h_{src}(i)) \leftarrow 0$
9: $C_{src}(i, h_{src}(i)).W_{ID} \leftarrow W_n$
10: $C_{src}(i, h_{src}(i)) \leftarrow C_{src}(i, h_{src}(i)) + g_{src}(i)$
11: $f_{src} \leftarrow median(\{g_{src}(i)C_{src}(i, h_{src}(i))|\forall i \in [1, 2, ..., d]\})$
12: $S_{src} \leftarrow S_{src} + \text{PCF}(f_{src})$

The same procedure presented in Lines 1-12 is similarly carried out for the destination address. Omitted for space.

13: $PC \leftarrow PC + 1$
14: **if** $PC = m$ **then**
15: $W_n \leftarrow W_n + 1$

Entropy Estimation (§ 3.2.1.2).

16: $H_{src} \leftarrow \log_2(m) - (S_{src} >> \log_2(m))$
17: $H_{dst} \leftarrow \log_2(m) - (S_{dst} >> \log_2(m))$

Traffic Characterization (§ 3.2.1.3) and Attack Detection (§ 3.2.1.4).

18: **if** $W_n = 1$ **then**
19: $M_{src} \leftarrow H_{src}$; $M_{dst} \leftarrow H_{dst}$
20: $D_{src} \leftarrow 1$; $D_{dst} \leftarrow 1$
21: **else**
22: $A \leftarrow (H_{src} > (M_{src} + kD_{src})$ **or** $H_{dst} < (M_{dst} - kD_{dst}))$
23: **if** A **is** *False* **then**
24: $M_{src} \leftarrow \alpha H_{src} + (1 - \alpha)M_{src}$
25: $M_{dst} \leftarrow \alpha H_{dst} + (1 - \alpha)M_{dst}$
26: $D_{src} \leftarrow \alpha|H_{src} - M_{src}| + (1 - \alpha)D_{src}$
27: $D_{dst} \leftarrow \alpha|H_{dst} - M_{dst}| + (1 - \alpha)D_{dst}$
28: $P_n \leftarrow 0$; $S_{src} \leftarrow 0$; $S_{dst} \leftarrow 0$

Defense Readiness (§ 3.2.2.1).

29: **if** A **is** *True* **then**
30: $DR_{state} \leftarrow$ `DEFENSE ACTIVE`
31: **else if** $DR_{state} =$ `DEFENSE ACTIVE` **then**
32: $DR_{state} \leftarrow$ `DEFENSE COOLDOWN`
33: **else if** $DR_{state} =$ `DEFENSE COOLDOWN` **then**
34: $DR_{state} \leftarrow$ `SAFE`

Frequency Variation Analysis (§ 3.2.2.2) and Packet Classification (§ 3.2.2.3).

35: $P.metadata.classification \leftarrow$ `LEGITIMATE`
36: **if** $DR_{state} \neq$ `SAFE` **then**
37: $f_{src,last} \leftarrow \text{ESTIMATE}(C_{src,last}, P.src)$
38: $f_{src,safe} \leftarrow \text{ESTIMATE}(C_{src,safe}, P.src)$
39: $f_{dst,last} \leftarrow \text{ESTIMATE}(C_{dst,last}, P.dst)$
40: $f_{dst,safe} \leftarrow \text{ESTIMATE}(C_{dst,safe}, P.dst)$
41: $V_{src} \leftarrow f_{src,last} - f_{src,safe}$
42: $V_{dst} \leftarrow f_{dst,last} - f_{dst,safe}$
43: $V \leftarrow V_{dst} - V_{src}$
44: **if** $V > t$ **then**
45: $P.metadata.classification \leftarrow$ `MALICIOUS`

Policy Enforcement (§ 3.2.2.4).

46: **if** $P.metadata.classification$ **is** `LEGITIMATE` **then**

Apply the normal forwarding table.

47: **else**

Apply the mitigation forwarding table.

Output: P

EUCLID uses count-sketches for this purpose instead of alternatives such as count-min-sketches (CMS), which would suffice for frequency variation analysis and could be smaller and faster (CORMODE, 2011). The reason for our choice is twofold. First, EUCLID's entropy estimation component requires unbiased frequency estimates for accurate detection, which the CMS does not provide. Second, at this point in execution, EUCLID has already calculated the values of the sixteen hash functions needed for the main count-sketches (which we store in arrays as exemplified in Algorithm 4, Lines 2-3) and can re-use these values to copy data to the historical count-sketches. Using another type of sketch (with different sizes) would require computing additional hash functions for each packet, increasing the computing resources footprint.

Right before executing the frequency approximation steps, we follow a procedure to ensure the updating of the counters related (*i*) to the current OW (stored in C_{src} and C_{dst}); (*ii*) to the last OW (stored in $C_{src,last}$ and $C_{dst,last}$); (*iii*) and to the baseline OW (stored in $C_{src,safe}$ and $C_{dst,safe}$). Since the counters from W_{last} and W_{safe} do not change more than once per OW, we only perform the corresponding operations at the *first* occurrence of x_{src} (resp., x_{dst}) in W_{curr}. Moreover, to avoid having to allocate memory for temporary data, we perform the operations in the order specified in Lines 5-7 of Algorithm 4. Furthermore, we only update the counters from W_{safe} if the DR state is `SAFE` (Line 5). Back at the frequency variation analysis component, we perform the calculations based in Equations 3.7, 3.8a, and 3.8b (§ 3.1.3), as indicated in Lines 37-43. At this point, EUCLID is ready to proceed to the packet classification component.

3.2.2.3 Packet Classification

For every packet that goes through the data plane when the defense state is `ACTIVE` or `COOLDOWN`, EUCLID calculates the frequency variation $\hat{V}$. As discussed in § 3.1.3, we expect that legitimate packets have smaller $\hat{V}$ values than the malicious packets have. By applying a mitigation threshold (t), our mechanism attempts to identify packets as legitimate or malicious. We seek optimal results both for the true-positive rate (TPR, the proportion of malicious packets correctly identified) and the false-positive rate (FPR, the proportion of legitimate packets mistaken as malicious). An ideal TPR is close to 100% so that our mechanism can correctly submit most malicious packets to the countermeasures they require. In contrast, the FPR must be close to zero percent so that legitimate traffic does not suffer undue disruption.

The mitigation threshold t is parameterizable by the network operator. We envision the dynamic adjustment of this threshold as future work. Once defined, t is used in a

test: if $\hat{V} > t$, our mechanism sets a metadata field to flag the packet as suspect (§§ 3.1.3, Equation 3.9b) (Lines 44-45). Thus, it sets up the packet to be processed by the next component—policy enforcement. Otherwise, i.e., if $\hat{V} \leq t$, the switch proceeds to perform its ordinary forwarding functions.

3.2.2.4 Policy Enforcement

The last stage of our security pipeline is policy enforcement (Lines 46-47). At this point, we apply a match-action table to determine what processing the packet must undergo. Our design allows the network operator to choose between three policies to be applied to suspect packets: `discard`, `throttle`, and `divert`. The `discard` policy is implemented by directly calling the P4 `drop` primitive. The `throttle` policy sends the packet to a rate-limited egress queue (although more elaborate implementations are viable). The `divert` policy changes the egress interface so that the packet can be processed off the main path by different devices (e.g., a deep packet inspector). The network operator selects policies by populating a match-action table with applicable rules.

3.3 Evaluation

To the best of our knowledge, EUCLID is the first work to explore data plane programmability, more specifically P4, to devise a fully in-network DDoS attack detection and mitigation mechanism. Due to the constraints related to the reduced set of $P4_{16}$ programming primitives, implementing our design requires numeric approximations and compact data representations (i.e., sketches). Hence, it is imperative to assess the *accuracy*, *resource utilization*, and *responsiveness* of our proposed mechanism thoroughly. In this section, we seek answers to the following research questions (RQs):

- *RQ1: How accurate is the entropy estimation pipeline as a function of memory space requirements?*
- *RQ2: Assuming reliable entropy estimates (RQ1), how accurate is our detection mechanism under different settings and attack intensities?*
- *RQ3: How accurate and responsive is our detection mechanism as compared to other monitoring strategies?*
- *RQ4: Assuming reliable detection (RQ2), how effective is our attack mitigation mechanism under different settings?*

Collectively, these RQs prompt us to investigate to what extent and under which conditions it is possible to rely on a fully in-network defense against DDoS attacks.

In the following subsection, we detail our evaluation methodology and experimental setup. Right after it, we discuss the results that support us in answering the RQs. Finally, we relate our findings to our mechanism's applicability to various attack scenarios.

3.3.1 Evaluation Methodology and Experimental Setup

This subsection describes the topology, target devices, datasets, and traffic generation methods used in the evaluation, followed by details about our experimental design.

Topology and Target Devices. Recall from § 3.1.1 that we expect our proposed mechanism to be deployed in an intermediate position within the autonomous systems (ASes) closest to the victim. Also, to prevent congestion of lower-capacity links, our mechanism should be installed on border routers (in each traffic ingress point), where it can process inter-AS traffic. Given these scenarios, without loss of generality, we use a single forwarding device (assuming traffic traverses one point only, which is the case of numerous setups). The single switch represents the point at which we deploy EUCLID.

We designed our solution for deployment on an RMT-based (BOSSHART et al., 2013) hardware device (e.g., a Barefoot Tofino switch (Barefoot Networks, 2020)). However, due to the relatively recent introduction of the P4 language and the small number of P4-programmable switch suppliers, this hardware has yet to become an off-the-shelf commodity. Despite this, software solutions facilitate the progress of the research on programmable networks. In this work, we conduct our experiments on a software-based $P4_{16}$ switch (The P4 Language Consortium, 2020). This setup does not affect our evaluation since both the accuracy and resource utilization are target-independent (i.e., hardware and software targets are functionally equivalent). Furthermore, by design, a hardware RMT-based pipeline forwards packets at line-rate, and within a fixed delay between ingress and egress (if there is no recirculation) (CHOLE et al., 2017). Since EUCLID does not use recirculation, timing is, therefore, not of concern.

The EUCLID source code is available at our Github repository[1] and can be used as a starting point for new developments in the area. Our repository also includes the data analysis notebooks, scripts, and tools we used for this work.

Datasets and Traffic Generation Strategy. We perform a packet trace-driven evaluation using representative datasets of legitimate and malicious traffic. As legitimate

[1] Available at <https://www.github.com/asilha/euclid>.

traffic, we use the *CAIDA Anonymized Internet Traces 2016* (CAIDA, 2016) dataset, recorded from high-speed Internet backbone links. As attack traffic, we use the *CAIDA DDoS Attack 2007* (CAIDA, 2007) dataset, which consists of an attempt to deplete the computing resources of a target server and to saturate its connection to the Internet. Despite not being recent, the DDoS dataset is renowned for its thoroughness and applicability to assess system performance under attack, as several high-impact publications on network security have attested. The volumetric DDoS attack captured in the dataset matches the attack scenario described in § 3.1.1.

We use our research group's traffic generator TRAFG[2] to combine the aforementioned datasets, forming synthetic workloads. Each workload follows a common structure: a *training phase* and a *detection and mitigation phase*. We set the length of the detection and mitigation phase n to 2^{27} packets, and TRAFG automatically prepends a training phase with $n/2$ packets, i.e., 2^{26}. Thus, the total size of each workload is 1.5×2^{27} packets (approximately 192 million). During the training phase, EUCLID analyzes only legitimate traffic in order to initialize the characterization model. The detection and mitigation phase is subdivided into three segments: *pre-attack*, with $n/4$ legitimate packets, *attack*, with $n/2$ packets (both legitimate and malicious), and *post-attack*, with $n/4$ legitimate packets. The attack segment combines $pn/2$ malicious packets and $(1-p)n/2$ legitimate packets, randomly selected according to a Bernoulli distribution with probability p. By varying p (e.g., 3%, 3.5%, ..., 6%, 20%), we represent different attack intensities (i.e., the proportions of malicious packets to the total number of packets during the attack). The TRAFG generator takes as inputs the datasets and the parameters n and p, and it outputs a packet trace file that follows the structure we described. Once we have the workloads, each experiment consists of submitting the packets into a switch interface.

Experimental Design. Table 3.1 shows the system factor levels we use throughout the evaluation. To evaluate estimation accuracy and detection performance (RQ1-RQ3), we set the observation window length m to 2^{18} packets, which corresponds to approximately 250 ms of traffic at 1 Mpps (the mean packet rate of our workload). To evaluate mitigation performance (RQ4), we use three different observation window sizes ($m \in \{2^{14}, 2^{16}, 2^{18}\}$ packets). These sizes allow us to investigate the effect of the window size on the detection and mitigation delays, as well as on memory usage. While 2^{18} packets represent $\approx$ 250 ms of traffic, 2^{16} packets take $\approx$ 65 ms, and 2^{14} packets take $\approx$ 15 ms. We define varying value ranges for count-sketch dimensions (d and w), sensitivity coefficient (k), and observation window size (m). These variations enable a broad assessment of EUCLID under different configurations. As detection relies on hash functions (for the address

[2] Available at <https://www.github.com/aclapolli/ddosd-cpp>

Table 3.1 – System Factor Levels

	Levels Used in Each Subsection			
System Factors	**§ 3.3.2**	**§ 3.3.3**	**§ 3.3.4**	**§ 3.3.5**
Observation Window Size (m)	2^{18}	2^{18}	2^{18}	$\{2^{14}; 2^{16}; 2^{18}\}$
Hashing Coefficients (a_i, b_i)	pseudo-random and pairwise-independent			fixed
Count-Sketch Depth (d)	$\{4, 8, 16\}$	4	4	4
Count-Sketch Width (w)	$\{64, 368, 672, 976, 1280\}$		1280	1280
Sensitivity Coefficient (k)	NA	$\{0, 0.5, ..., 8\}$	4	$\{4.875, 4.875, 3.625\}$
Defense Threshold (t)	NA	NA	NA	$64k, k \in \{-16, ..., 16\}$

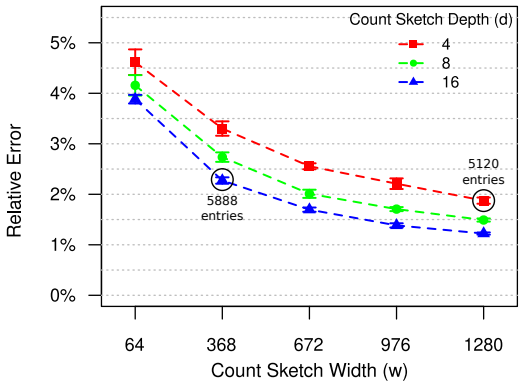
frequency approximation step), we must address the impact of their intrinsic bias on our proposed mechanism. Hence, to assess the detection performance (RQ1-RQ3), we conduct 15 repetitions for each configuration, using random hashing coefficients. Moreover, we present the results at a 95% confidence level. In contrast, the mitigation performance evaluation (RQ4) already assumes accurate detection. Thus, it is not necessary to evaluate the effects of the hashing coefficients over multiple repetitions. We expect variability *within* each experiment of the mitigation performance assessment. As an attack progresses, mitigation accuracy changes between different observation windows (OWs). Thus, we report the 95% confidence intervals for the mean of all the measurements in the detection phase. Across all experiments (RQ1-RQ4), we set the smoothing coefficient of the exponentially-weighted moving averages and deviations to $\alpha = 20 \cdot 2^{-8}$.

3.3.2 Entropy Estimation Error

Resource constraints of programmable data planes require space- and time-efficient designs. Thus, instead of attempting to calculate exact entropies, we propose estimating these values (§ 3.2.1.2). While estimation reduces the demands for memory space and processing time, it inexorably diminishes accuracy. Such a loss of accuracy can hide traffic anomalies, which would hinder detection performance. Consequently, we must assess the accuracy of our entropy estimates as a function of the count sketch dimensions (RQ1), which are the most crucial factors for determining the accuracy of the sketch-approximated frequencies (CHARIKAR; CHEN; FARACH-COLTON, 2002).

In our extended count-sketch implementation, we store each counter in a 32-bit register and its associated observation window (OW) identifier in an 8-bit register. We store the entropy and entropy norm estimates in 32-bit registers, using fixed-point notation with four fractional bits. We populate the longest-prefix match (LPM) table for our pre-computed function ensuring a maximum absolute error of 2^{-4} for each entry. The resulting table contains a total of 245 TCAM entries of 32 bits each, i.e., 980 bytes.

Figure 3.5 – Relative error of the entropy estimation as a function of count-sketch width and depth.



Source: Lapolli, Marques and Gaspary (2019).

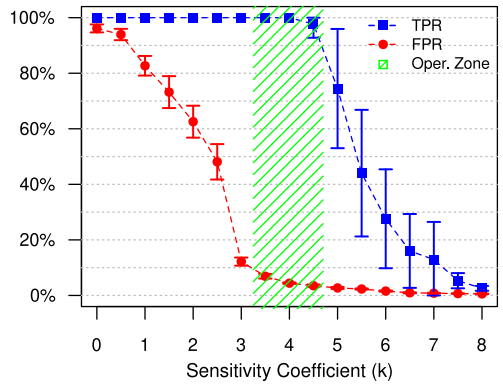
Figure 3.5 shows the relative estimation error for each count-sketch depth and width level listed in Table 3.1 (first column). By definition, the sketch width (w) is inversely related to the probability of hashing collisions (CHARIKAR; CHEN; FARACH-COLTON, 2002). By following the horizontal axis, we can observe how this factor affects the estimation error: larger widths reduce errors, although this reduction is attenuated until it stabilizes close to 1%. We highlight that the pre-computed function also slightly impacts the relative error, but the plot combines both influences.

Increases in the sketch depth d reduce the probability of obtaining estimates from counters affected by hashing collisions. By examining the error values for a single width, we can observe how the depth affects accuracy. However, larger sketch depths require (*i*) processing more hash functions for each packet and (*ii*) more execution steps to calculate the median (see § 3.2.1.1). We annotate Figure 3.5 with the total number of sketch entries (5 888 and 5 120) in two specific depths ($d = 16$ and $d = 4$, respectively). These values reveal that, for comparably-sized sketches, increasing depths does not significantly improve the accuracy of the estimates. Hence, we decide to set $d = 4$ in the subsequent experiments.

3.3.3 DDoS Attack Detection Performance

EUCLID allows network operators to configure the sensitivity coefficient (k) in order to obtain a suitable trade-off between the true-positive rate (TPR) and the false-positive rate (FPR) of attack detection. In the detection performance analysis, the TPR refers to the attack phase and indicates the number of OWs in which we detect attacks divided by the number of OWs in which there is an attack. The FPR refers to the pre- and post-attack phases and indicates the number of OWs in which we detect attacks divided by the total number of OWs in the pre- and post-attack phases. Seeking to answer RQ2, we

Figure 3.6 – Impact of the sensitivity coefficient k on the true-positive and false-negative attack detection rates. The area in green highlights the desired operating zone.



Source: Lapolli, Marques and Gaspary (2019).

first tune the factor k by observing its effects on the TPR and the FPR (§ 3.3.3.1). Then, we study the detection accuracy as related to attack proportion and memory usage (§ 3.3.3.2)

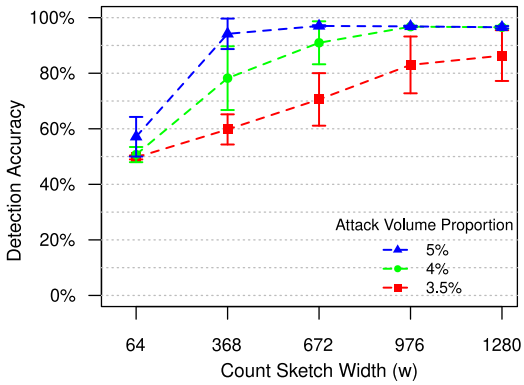
3.3.3.1 Sensitivity Coefficient Effect

In this experiment, we set the sketch dimensions to $d = 4$ and $w = 1\,280$ (§ 3.3.2). The proportion of malicious packets to the total number of packets during the attack is 5%. Figure 3.6 presents the TPR and the FPR for attack detection as a function of the sensitivity coefficient k. We can observe that for lower sensitivity coefficients, detection reaches excellent TPRs, i.e., close to 100%. However, the elevated FPRs indicate excessive proportions of false alarms. Following the horizontal axis, as we increase k, both TPR and FPR decrease until the mechanism ceases to generate attack alarms. The false-positive rate decreases from $k = 0$, reaching less than 10% for $k \geq 3.25$. From this point on, the true-positive rate remains close to 100% as long as $k \leq 4.75$. Thus, our mechanism is in its desired operating zone when the sensitivity coefficient k is within $[3.25, 4.75]$ (green hachure). Due to network traffic variability, it may be necessary to adjust k periodically. In our current design, it is the responsibility of the network operator to set and update k to an appropriate level. In future work, we will address this by proposing an automatic self-tuning of the sensitivity coefficient.

3.3.3.2 DDoS Attack Detection Accuracy

For this analysis, we set the sensitivity coefficient (k) set to 3.5 (which is within the operating range discussed in § 3.3.3.1). We now study the attack detection *accuracy* of our mechanism under various count-sketch widths (which correspond to memory utilization) and proportions of malicious traffic (see Table 3.1).

Figure 3.7 – DDoS attack detection accuracy in terms of memory utilization for different proportions of malicious traffic.



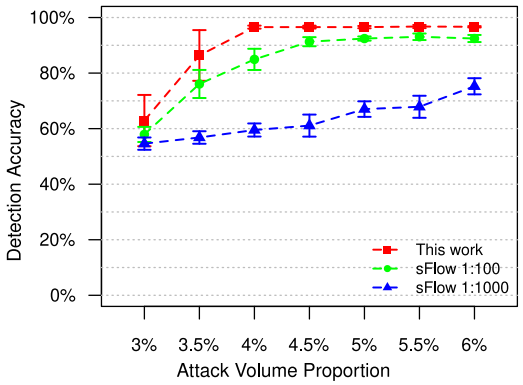
Source: Lapolli, Marques and Gaspary (2019).

Figure 3.7 shows a curve for each attack proportion we consider. As attacks become increasingly aggressive, detection accuracy reaches progressively higher rates (exceeding 90%). This outcome stems from the more significant entropy anomalies that stronger attacks cause. Along all curves, we observe that the sketch width profoundly influences the detection accuracy. This effect is noticeable even for less intense attacks (3.5%), which EUCLID detects with accuracy higher than 80% for $w \geq 976$. As we increase sketch sizes, entropy estimates become more accurate, which facilitates the detection of subtler attacks.

While enlarging sketches does improve detection accuracy, it also increases memory footprint. We illustrate how to calculate the static random-access memory (SRAM) requirements as follows. First, recall from § 3.2.1.3 that we track entropies separately for source and destination IP addresses. Thus, frequency approximation needs *two* sketches. Next, we use the sketch dimensions to obtain the total number of entries across both sketches. Then, we consider that each entry consists of a 32-bit counter and an 8-bit observation window (OW) identifier, totaling 40 bits per entry. Finally, we multiply the total number of entries by the entry size. For instance, assuming $d = 4$ and $w = 976$, we obtain $2 \cdot 4 \cdot 976 \cdot 40 \approx 312$ kilobits, i.e., 38.125 kB of SRAM. The footprint just described applies to each 1 Gbps link. Higher data rates demand larger OWs in order to enable a robust representation of the address frequency distributions. Given that the count-sketch estimation error is proportional both to $1/\sqrt{m}$ (where m is the OW length) and to the ℓ_2 norm of the address frequencies[3], faster data links require using proportionally larger sketches. Consequently, considering a 24-port 10 Gbps programmable switch (BOSSHART et al., 2013), we extrapolate the EUCLID memory footprint for detection to 8.93 MB, which amounts to 20% of the available SRAM (44.11 MB).

[3]We assume the ℓ_2 norm increases proportionally to the traffic rate.

Figure 3.8 – DDoS attack detection accuracy: comparison with packet sampling approaches.



Source: Lapolli, Marques and Gaspary (2019).

3.3.4 Comparison with Packet Sampling

Programmable switches can collect fine-grained data about all forwarded packets, which facilitates the deployment of highly-sensitive attack detection mechanisms. In contrast, detection strategies that rely on packet sampling must operate on significantly less data, which limits the detection accuracy. We explore the relation between attack intensity and detection accuracy by comparing EUCLID with an implementation of our detection strategy that receives samples from an sFlow collector (RQ3).

In this section, we perform our experiments using $d = 4$, $w = 1280$, and $k = 4$ (the center of the operating zone discussed in § 3.3.3.1). We assess the sFlow-based mechanism using two different sampling rates: (*i*) 1:1 000, which Phaal (2009) suggests for a 1 Gbps link, and (*ii*) 1:100, in an attempt to improve the detection results. Thus, we have three different scenarios for the assessment: our strategy and the two sFlow-based implementations. To ensure comparable baselines, we set different observation window sizes m for each scenario. For instance, during the time EUCLID processes m packets, the sFlow collector exports only approximately $m/1\,000$, or $m/100$ packets, depending on the sampling rate. Consequently, we scale m such that the time frames of the three scenarios become approximately equal.

Figure 3.8 depicts the attack detection accuracy for the three scenarios discussed above under different attack volume proportions. The lower curve indicates that the 1:1 000 sampling rate yields a severely degraded detection performance. At a 1:100 sampling rate, the sFlow-based implementation shows a significantly improved accuracy. Nevertheless, EUCLID outperforms the sFlow approach in every observed attack strength.

We also investigate the attack detection delay by analyzing the timestamps of the packets in our workloads. We use the timestamp of the first malicious packet to indicate

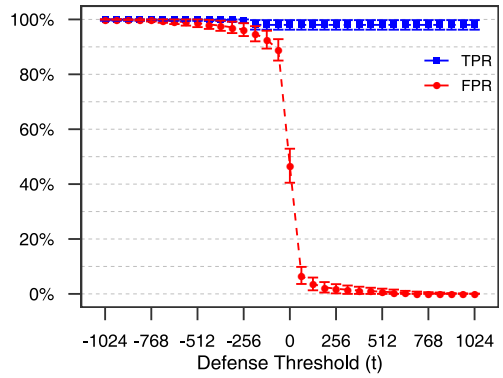
the beginning of the attack. We consider the time of detection the timestamp of the last packet of the OW that identifies the entropy anomaly. We observe that for lower attack proportions ($p \leq 4\%$), packet sampling takes several seconds to detect an attack. Under similar conditions, EUCLID detects an attack in a fraction of the time, i.e., at most a few hundred milliseconds (considering our workload has a mean packet rate of 1 Mpps). Such a lower delay may lead to earlier activation of attack mitigation mechanisms, thus potentially preventing service degradation or outage.

3.3.5 DDoS Attack Mitigation Performance

Whereas in the previous subsections, we investigate detection accuracy, in this subsection, we analyze the performance of our mitigation strategy. In the mitigation performance analysis, the true-positive rate (TPR) and the false-positive rate (FPR) indicate, respectively, the proportions of *packets* correctly and incorrectly identified as malicious. We underscore that EUCLID was originally designed to handle high-rate volumetric attacks (i.e., the packet rate is expected not to be "low"). In evaluating the detection and mitigation components of EUCLID, we look for the worst-case/tipping point, which is different in each case. For the detection component, we put it under stress with low proportions of malicious traffic (e.g., 3% to 6%). This is when detection accuracy may degrade (see the lowest curve in Figure 3.7), which is in line with the literature (XIANG; LI; ZHOU, 2011). Conversely, the mitigation component may reach its limits when the amount of traffic subjected to further inspection/filtering is high (e.g., 20% of malicious traffic). Hence, we set the attack intensity to 20% (i.e., $p = 0.2$). As for the other factor levels, we take into consideration the results from Subsection 3.3.3 regarding count-sketch dimensions (w and d) and sensitivity coefficient (k).

By design (Section 3.2), EUCLID raises attack alarms only at the end of the observation windows (OWs) in which malicious traffic causes excessive changes in entropy. Consequently, mitigation starts operating only at the beginning of the next OW. For instance, if a detectable incident emerges during OW_i, our countermeasures become active as OW_{i+1} begins. Therefore, we perform the experiments of this section under different observation window lengths in order to investigate whether it is possible to (*i*) shorten the mitigation delay and (*ii*) reduce the amount of memory required.

Figure 3.9 – Effects of the defense threshold t on the true-positive and false-positive packet classification rates.



Source: the author (2020).

3.3.5.1 Mitigation Threshold Effect

For an OW size of $m = 2^{18}$ packets, we measure the effect of different defense thresholds (t) in the interval $[-1\,024; 1\,024]$, with 64-packet increments. We consider observation windows from OW_{i+1} to OW_{i+k}, where $i + 1$ identifies the first window after attack detection, and $i + k$ indicates the last window in the attack segment. Finally, we calculate 95% confidence intervals for the mean TPR and FPR during the attack.

Figure 3.9 shows consistently high true-positive rates, between 98.7% and 100%, for thresholds $t \leq -256$ packets. After this point, for all tested values of t, the TPRs remain elevated, ranging from 96.3% to 99.7%. This result indicates that EUCLID correctly identifies most of the malicious packets in the attack. As a result, by enforcing a security policy (e.g., discarding), our mechanism can prevent over 96% of the spurious traffic from disrupting services at their target. We can also observe that the FPR quickly decreases as t increases. Notably, for $t \geq 768$ packets (approximately 0.25% of the window size), the FPR becomes less than 0.7%. Such a low FPR indicates that the mitigation threshold has near-perfect specificity, i.e., addresses whose observed frequency variations fall *below* the mitigation threshold can be safely assumed legitimate. Since the proportion of legitimate packets incorrectly classified as suspects is negligible, applying a mitigation policy to such packets is unlikely to cause noticeable problems to legitimate users.

3.3.5.2 Observation Window Size Effect

We now investigate the performance of attack mitigation under different observation window sizes. Measurement fluctuations related to these different OW sizes require adjusting the sensitivity coefficient (k) within the operating range (Table 3.1, last column).

Table 3.2 – Effects of the observation window size m.

OW Size (m)	2^{14}	2^{16}	2^{18}
Threshold (t)	96	384	768
TPR (%)	[97.45, 98.34]	[97.04, 98.81]	[96.26, 99.74]
FPR (%)	[0.00, 0.08]	[0.00, 0.05]	[0.00, 0.63]
Defense Delay	≈ 16 ms	≈ 64 ms	≈ 256 ms

For all three factor levels of m, we observe similar outcomes, i.e., the TPR remains over 90%, and the FPR quickly decreases as the mitigation threshold t grows. Moreover, the t value that yields near-zero FPRs varies with the OW size. We summarize these results in Table 3.2. These findings indicate that reducing the observation window length does not hinder the accuracy of our mitigation mechanism.

Considering the 1 Mpps average packet rate of our workload, an observation window with 2^{18} packets corresponds to approximately 250 ms of traffic. By reducing the OW length to 2^{14} packets, each OW will take close to 15 ms. Such a reduction can further reduce the attack mitigation delay. Since the memory space needed for frequency approximation is proportional to the size of the OW (§ 3.3.3.2), we can also reduce the amount of SRAM required for detection and mitigation.

3.3.5.3 Effects on Traffic Latency

Accurate measurements with hardware-based experiments of the latency introduced by EUCLID (actually, by any P4 design) will lead to results that vastly vary depending on the specific devices employed. Aiming to provide the reader with a more general analysis, we determine a theoretical upper bound for latency. To do this, we consider that our $P4_{16}$ prototype targets the RMT model implementation by Bosshart et al. (BOSSHART et al., 2013), whose switch chip: (*i*) has a 1 GHz operating frequency; (*ii*) parses packet headers in a single cycle; (*iii*) has 32 match-action stages in each pipeline (ingress and egress); and (*iv*) performs each match-action in a single cycle[4]. EUCLID uses only the *ingress* parser, pipeline, and deparser processing blocks[5] and does not require packet recirculation. Therefore, a worst-case scenario incurs the forwarding delay resulting from the sequential processing of 34 stages (32 match-action + 2 [de]parser), at 1 ns per stage, i.e., 34 ns. This analysis can be readily performed for alternative targets, but it is reasonable to expect average latencies in the order of tens of nanoseconds (including in experimental studies).

[4]Parallelism allows multiple match-actions in a single cycle.
[5]We only use the egress processing blocks to output diagnostic data.

3.3.6 Applicability and Limitations

Different DDoS attacks require distinct defense strategies. Our evaluation has so far indicated that EUCLID is effective against volumetric flooding attacks coming, e.g., from botnets (i.e., high source address entropy and low destination address entropy). Nevertheless, we can still reflect on our proposed solution's behavior under scenarios for which it was not designed originally, such as semantic, increasing-rate, and amplification-based attacks.

Semantic Attacks. We discuss this type of attack in Section 2.1. Defending against such attacks requires analyzing transport- or application-layer traffic. EUCLID, by design, observes network-layer traffic patterns; thus, it does not cover semantic attacks. Defenses against this attack category require a type of solution beyond the scope of this chapter.

Increasing-rate Attacks. In these brute-force campaigns, the attack rate starts low and gradually increases in an attempt to manipulate the baseline traffic characterization model, thus evading detection (MIRKOVIC; REIHER, 2004). To improve the robustness of the traffic model under such "slow-start" attacks, the network operator can adjust the sensitivity coefficient (§ 3.2.1.3) and the cooldown period (§ 3.2.2.1). More specifically, he/she is expected to tune these parameters more conservatively, making EUCLID spend more time in the DEFENSE states (in which we do not update the traffic model). Since the packet classification FPR is remarkably low, we do not anticipate overhead issues due to the more frequent activation of the mitigation mechanism.

Amplification-based Attacks. Attacks that rely on amplification strategies would also cause entropy anomalies. Differently from botnet-originated campaigns, however, amplification causes sudden, pronounced decreases in both source and destination entropy measurements. As we discussed in § 3.1.2, attack detection occurs whenever at least one of the entropy measurements deviates from the model. Consequently, a quick drop in destination entropy would cause the anomaly detector to engage the mitigation mechanisms. Regarding mitigation, given that the IP address frequency variations caused by reflection attacks are still anomalous, we can readily classify and treat malicious packets accordingly.

Fluctuations in the Number of Flows. A trace with large fluctuations in the number of flows would induce relevant entropy variations. The scale of the entropy variations influences how strict the traffic model is. If we train the model with traces that exhibit large fluctuations, detection will also require significant anomalies. It is possible that flow accounting could add details that might be useful to improve our mechanism accuracy. However, our experiments show that the attacker's behavior is sufficiently disruptive to cause detectable entropy anomalies within our attack model. Thus, we

advocate that using IP addresses is good enough for detection. Our tuning parameters (e.g., the sensitivity coefficient) can be adjusted to what is expected in a given scenario, allowing an operator to obtain adequate attack detection FPR and TPR values. Moreover, implementing flow accounting would require adding extra memory for bookkeeping. Exploring this compelling research avenue and assessing whether such a change would improve accuracy is left as future work.

3.4 Lessons Learned and Insights

While data plane programmability brings flexibility to packet forwarding, implementing this paradigm on hardware is a challenge. Obtaining adequate trade-offs between high performance and reasonable production costs is paramount. Modern programmable data planes reach this goal by carefully delimiting reconfigurability to a core repertoire of primitives related to packet forwarding (e.g., header parsing and match-action tables). Moreover, these data planes do not implement several popular programming constructs (e.g., repetition statements, stacks, and non-trivial mathematical operations). Additionally, since the amount of memory directly influences chip area and power consumption, packet switches provide relatively small sizes of SRAM and TCAM. On the one hand, these hardware design choices help prevent stalls in the packet processing pipeline as well as prohibitively complex hardware layouts. On the other hand, algorithms for programmable switches require particularly careful design and implementation. We detail the most important lessons learned from the design of EUCLID in the remainder of this section.

a) Limited syntactic expressiveness requires careful programming practices: P4-programmable switches have limited support for procedure definition and invocation (i.e., match-action tables are not as flexible as the function call and return instructions). Consequently, code reuse becomes challenging, which makes the implementation process more intricate. Intuitively, a way to work around this limitation is to write tools to generate P4 code automatically. However, perhaps it would be better to enrich the P4 language with standardized higher-level constructs that the compiler or preprocessor could turn into native P4 code. This strategy could also promote the widespread adoption and distribution of libraries with stable implementations of well-established building blocks (e.g., Bloom filters, count-sketches, and algorithms to approximate non-trivial mathematical functions).

b) Event-driven processing can compensate for the absence of iterative procedures: In a general way, all real-time systems have stringent time budgets. This characteristic profoundly influences the architecture of programmable data planes, which ultimately

prompts algorithm redesign. In this work, this observation emerges when addressing three design challenges: (*i*) the summation of the individual address frequency terms required by Equation 3.1, (*ii*) the need to reset sketch counters between observation windows in order to avoid outdated values, and (*iii*) the necessity to copy data between sketches. It is unfeasible to iterate over entire sketches for every single incoming packet since the resulting overhead would exceed the time budget that line-rate packet processing requires. Thus, we need to redesign iterative procedures as event-driven strategies in which every packet arrival triggers the execution of smaller, tractable steps. EUCLID handles the first challenge (*i*) by gradually accumulating the entropy norm variation as each packet arrives at the switch. We tackle the second challenge (*ii*) by augmenting the sketch counters with an observation window (OW) identifier W_{ID} and modifying the count sketch operations such that accesses to outdated counters produce an automatic reset followed by an update to the current window ID. Thus, we defer the resetting of each cell until they are necessary. Similarly, we approach the third challenge (*iii*) by placing copy operations close to counter updates so that, right before resetting counters, we can perform the required copies we discussed in § 3.2.2.2.

c) The number of pipeline stages in the data plane limits the size of multi-step procedures: The time budget of programmable data planes also constrains the maximum length of the code to deploy on the switch. As a result, attempts to manage the absence of iteration (e.g., by resorting to loop unrolling) and subroutines (e.g., by automatic generation of repeated code segments) might not always be a viable strategy. Thus, algorithm design needs to take into consideration the space requirements for the resulting code.

d) Pre-computed functions can address the lack of non-elementary mathematical functions: Current programmable switches do not directly implement operations such as divisions, exponentiations, and logarithms. Thus, we need to write algorithms to approximate these functions in terms of the primitives already available on them. We can handle this challenge by numerically analyzing the function signature concerning its domain and image bounds. We do this to identify opportunities for compact representations tailored to the use case at hand. In our work, there is the need to calculate updates to the entropy norm terms, which depend on the binary logarithm (Equation 3.10). The entropy norm update function has strictly-bounded intervals for both domain (frequencies ranging from one to the OW size) and image (Figure 3.4). As a result, it is possible to build a memory-efficient longest prefix match (LPM) table by aggregating domain entries with close values while ensuring enough accuracy for our purposes. Alternatively, for the general case, one can use numeric algorithms explicitly developed for in-switch execution (DING; SAVI; SIRACUSA, 2020).

e) Floating-point arithmetic may not be essential to operate numbers whose precision and range are strictly constrained: As traditional packet forwarding requires only integer arithmetic, switches typically lack floating-point instructions and registers. Nevertheless, integer arithmetic can operate on fractional numbers given a fixed-point representation. Throughout this work, we express non-integer quantities in fixed-point notation using scaling factors from 2^3 to 2^{18}. These numbers are the smoothing and sensitivity coefficients, the entropy norms, and the indices of central tendency and dispersion. Our evaluation (Section 3.3) shows that fixed-point representation is sufficiently accurate both to detect and to mitigate DDoS attacks.

f) The absence of dynamic memory allocation in the data plane limits the flexibility of the self-tuning of the mechanism: EUCLID has several tuning parameters (*i.e.*, α, k, m, and t) that the network operator can modify at runtime by updating register values. However, changes in parameters that dictate the memory size of data structure (i.e., the sketch dimensions d and w) are not straightforward. Since modern programmable data planes do not provide dynamic memory management, the compiler is responsible for allocating memory statically. Thus, changes in memory layout require reinstalling the P4 program, which is a disruptive operation. In this scenario, enhancing the flexibility of self-tuning beyond simple changes in register values demands the investigation of novel P4 constructs.

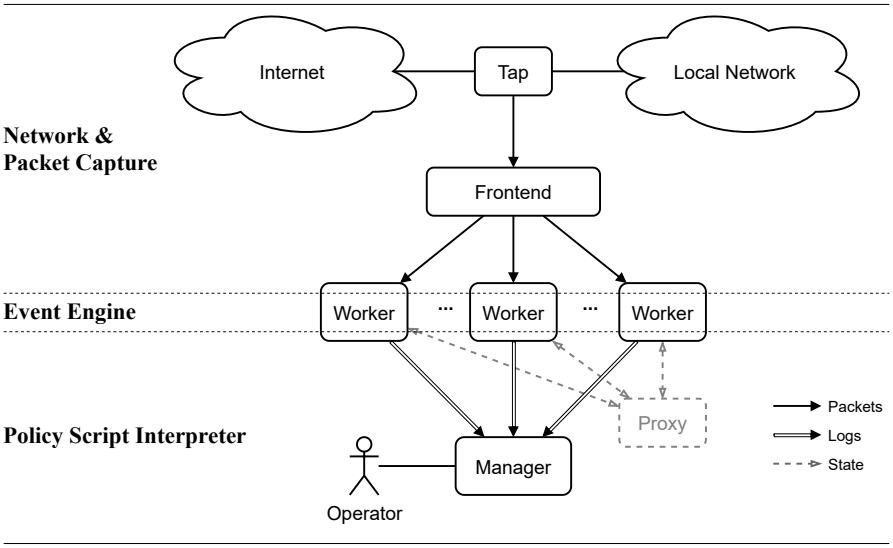
4 TOWARDS A GENERAL APPROACH FOR CYBERATTACK DETECTION USING PROGRAMMABLE DATA PLANES

This chapter introduces a general approach for data plane-based cyberattack detection. In contrast to our proposal in Chapter 3, which focused on a specific attack category (as EUCLID dealt with volumetric DDoS), this proposal seeks a comprehensive countermeasure for security threats. Our primary goal is to pave the way for scalable attack detection in present and future high-speed networks. To meet this goal, we propose a solution that (*i*) considers a broad range of network analysis and monitoring operations, such as those required by an Intrusion Detection System (IDS), (*ii*) translates these operations into constructs deployable to a Programmable Data Plane (PDP), and (*iii*) provides a runtime environment for IDS-to-PDP operation offloading. More specifically, we analyze a collection of event-driven network monitoring scripts, enumerate the related events of interest, and then devise a strategy to enable a Programmable Forwarding Device (PFD) to notify an IDS about the occurrence of such events. This three-phase analyze-enumerate-offload cycle can be carried out either *ad-hoc* or automatically. In this work, we take the *ad-hoc* path to gain a deeper understanding of the challenges this cycle entails. We design RNA - Reconfigurable Network Analytics, a framework to offload bulk traffic processing from general-purpose computers to high-performance programmable forwarding devices. Through a series of case studies, we explore the challenges of integrating cyberattack detection into a softwarized network. We present a proof-of-concept implementation of our design, targeting the Zeek Network Security Monitor (The Zeek Project, 2022) and P4-programmable V1 Model forwarding devices.

4.1 Identifying Candidate Operations

We need to identify IDS-related operations and traffic analysis tasks that are good candidates for offloading. To reach this goal, we investigate what activities are (*i*) frequently performed, (*ii*) critical for scalability and timeliness, and (*iii*) implementable under the constraints of the PDP. In preparation for this investigation, we performed an in-depth study of the Zeek architecture (see Section 2.2, p. 24) to understand the data and control flows within the IDS and to enumerate Zeek-related operations. We now discuss these operations as they relate to each Zeek architectural layer. Figure 4.1 shows the elements in a typical Zeek cluster deployment and how these elements relate to the Network, Packet Capture, Event Engine, and Policy Script Interpreter layers. The Packet Capture layer is

Figure 4.1 – Zeek Architectural Layers and Cluster Deployment



Source: adapted from The Zeek Project (2022).

implemented by the *Tap* and *Frontend* nodes. The Event Engine is bound to the *Worker* nodes. The Policy Script Interpreter (PSI) is tied to *Worker*, *Manager* (which consolidates logs), and *Proxy* nodes (which can execute arbitrary tasks). Layer by layer, the following paragraphs discuss these operations and their potential offloading to P4.

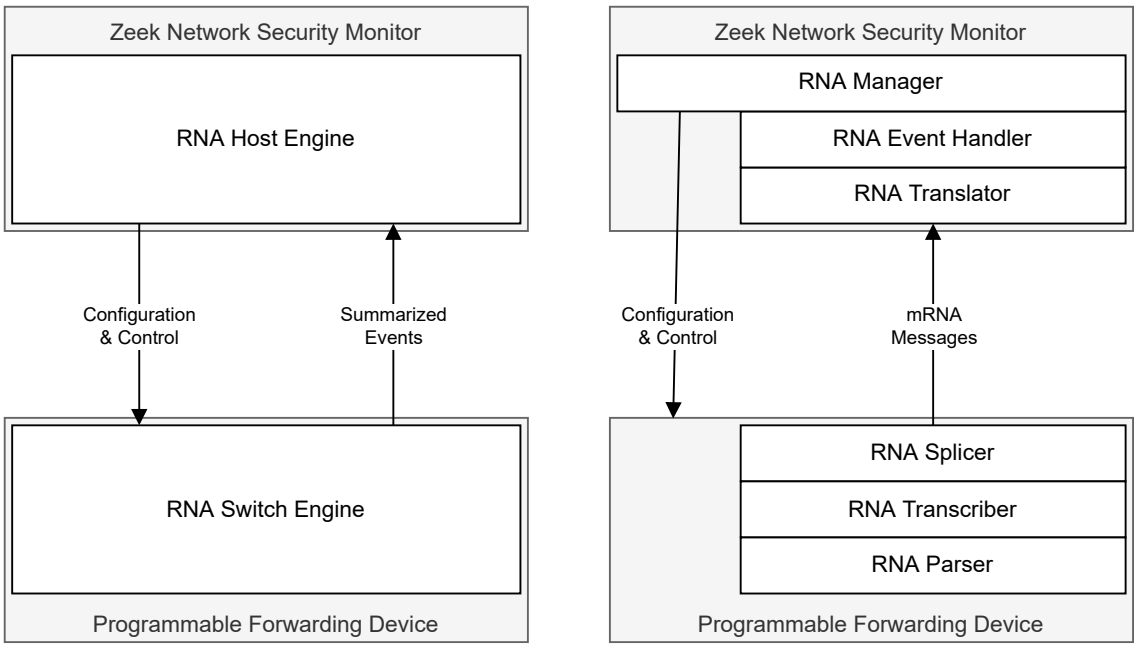
Packet Capture Operations. First, in the Packet Capture layer, we have (*i*) monitoring setup, (*ii*) header parsing, (*iii*) capture filtering, and (*iv*) load balancing. Monitoring setup consists in configuring the Tap (Figure 4.1) by setting up a "monitor session" in an in-path forwarding device. Ordinarily, a network operator would configure the Tap manually, but we can automate these operations by invoking the control interface of a P4 switch and adding code to clone packets. After monitoring has been set up, we can use the Programmable Forwarding Device (PFD) to perform additional checks. We begin by parsing and identifying common link-layer, network-layer, and transport-layer headers, followed by integrity validation. A PFD can efficiently perform these operations and discard invalid packets. After this step, we have enough information about L2-L4 header fields. At this point, packet capture filtering compares the data in these fields with the type of traffic Zeek is interested in (e.g., network prefixes and ports) to decide whether a given packet should be considered for further inspection. We can use P4 match-action tables to drive conditional cloning to implement filtering. Load balancing distributes packet analysis across a group of Worker nodes. Back in Figure 4.1, we see that a typical deployment has a Frontend, which ensures all packets within a given flow are forwarded to the same Worker. This property is necessary for session state tracking and byte stream reassembly. We can offload the Frontend operations to the PDP by using match-action tables to assign a Worker for each packet flow (according to arbitrary criteria, policies, and methods).

Event Engine Operations. Next, in the Event Engine layer, we have (*i*) lightweight packet inspection (LPI), (*ii*) session state tracking, (*iii*) timer management, (*iv*) network-layer fragment reassembly, (*v*) transport-layer byte stream reassembly, (*vi*) application protocol dissection, (*vii*) deep packet inspection (DPI), and (*viii*) event generation. LPI works for protocols that we can analyze statelessly and without reassembly (e.g., NTP and UDP-based DNS). We can implement LPI on P4 using its parsers and match-action tables to recognize and operate on protocol messages. Operations (*ii*) to (*v*) enable Zeek to translate sequences of related packets into flows and sessions, thus making more complex analyses feasible. A programmable switch can help perform these operations. For instance, we could generate flow identifiers for Zeek. Next, to perform operations (*vi*) and (*vii*), Zeek consumes reassembled streams. Dissection and DPI are complex operations whose execution requires sophisticated code and large amounts of memory, making their implementation in P4 unfeasible (mainly because of the lack of architectural support and memory space required for PDU reassembly). After the EE has finished processing, operation (*viii*) instantiates events and sends them to the PSI for handling. We can use a programmable switch to generate summarized event-signaling packets and send them to the Worker node, where we can use software to decode these packets and forward the results to the PSI.

Policy Script Interpreter Operations. Finally, the Policy Script Interpreter layer is responsible for (*i*) event handling, (*ii*) accounting, (*iii*) correlation, (*iv*) synthesis, and (*v*) notification. The PSI is event-driven: upon receiving an event, the PSI looks for registered event handlers. Each handler can perform arbitrary operator-defined tasks to gather and update traffic statistics and correlate events. A handler can also synthesize events to be forwarded to other handlers. Likewise, a handler can invoke external programs to perform arbitrary actions. Typically, event handling results in writing logs, storing data files, and sending alerts to operators. Most PSI tasks are very diverse and flexible, making them better suited to general-purpose computers. Moreover, these high-level operations require significant amounts of primary memory, making them unsuitable for resource-constrained equipment such as P4 forwarding devices.

In selecting candidate operations, we consider that P4 was designed for line-rate packet *header* parsing and manipulation instead of general-purpose *payload* processing (BOSSHART et al., 2013). Hence, the remainder of this work will tackle a subset of the Packet Capture and Event Engine functionalities: (*i*) monitoring setup, (*ii*) parsing link-layer, network-layer, and transport-layer protocols (Ethernet, IPv4, TCP, UDP, and ICMPv4), (*iii*) packet capture filtering, and (*iv*) lightweight packet inspection for application protocols (e.g., NTP and DNS).

Figure 4.2 – The RNA Framework



(a) Overview (b) Components

Source: the author (2022).

4.2 RNA - Reconfigurable Network Analytics

We propose the RNA Framework to offload the PDP-deployable intrusion detection operations we discussed in the previous section. Figure 4.2a shows an overview of our proposed solution. It has two main components: the RNA HOST ENGINE (executed in a Zeek Worker node) and the RNA SWITCH ENGINE (running in a P4 switch). Conceptually, the Host Engine enables Zeek to offload packet capture and analysis to the Switch Engine. The Switch Engine parses packets and transforms them into summarized messages to be submitted to Zeek. Upon receiving these messages, the Host Engine translates them into events the IDS can process. Figure 4.2b presents more details about our proposed solution and its subcomponents, which we will discuss in the following paragraphs.

RNA Host Engine. This component unfolds into three interrelated subcomponents: MANAGER, EVENT HANDLER, and TRANSLATOR. The Manager is a ZeekScript module which starts by setting up a monitoring session in the switch. Next, the Manager loads the Event Handler, which, in turn, activates the Translator submodules and subscribes to their events. Upon receiving these events, the Handler generates logfile entries and, optionally, displays console messages for debugging. At the core of the Host Engine, the Translator takes summarized events encoded as MRNA messages and synthesizes the corresponding Zeek-native events. The Translator fits into the Zeek Packet Analysis Framework (see

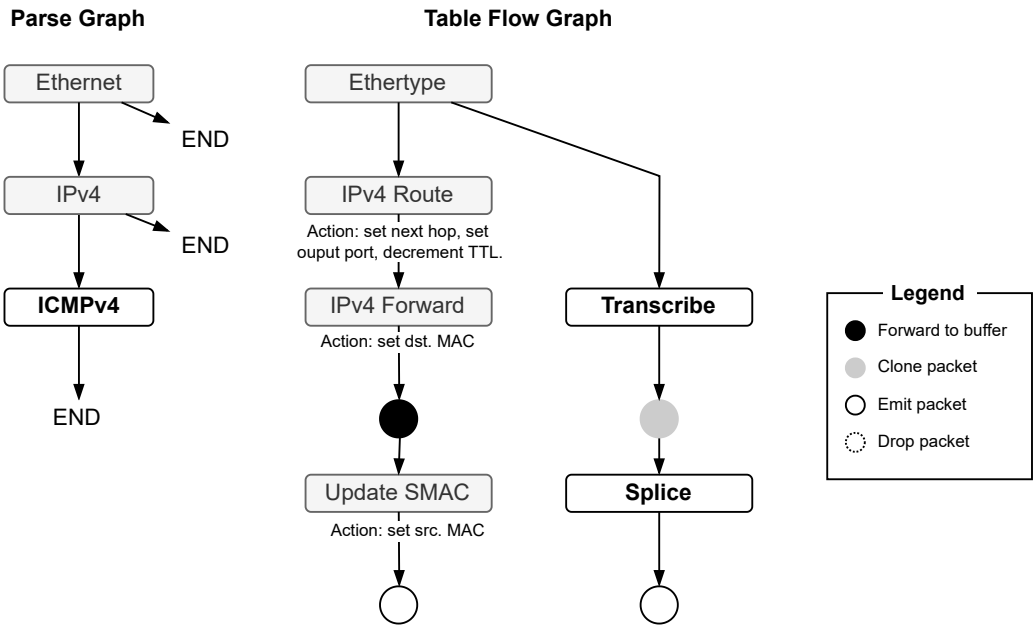
Section 2.2). We do not overwrite Zeek-native analyzers; instead, we insert the Translator right after the Zeek-native Ethernet analyzer. For instance, for ICMP, whereas the original Zeek data flow is Root-Ethernet-IP-ICMP (resulting in ICMP events), RNA changes it to Root-Ethernet-RNA (also resulting in ICMP events). Translator-generated events reach the Zeek PSI, where they are dispatched to subscribing handlers.

RNA Switch Engine. The Switch Engine has three main components: PARSER, TRANSCRIBER (data collector), and SPLICER (mRNA message synthesizer). As each packet enters the switch, the device parses its headers from the link layer to the application layer. Next, in the ingress pipeline, the Transcriber gathers metadata to be used later by the Splicer. The Transcriber also applies packet capture filtering, as it discards packets considered irrelevant for intrusion detection. The Splicer synthesizes an mRNA message and sends it to the Host Engine for translation into Zeek Events. The Switch Engine is implemented as a P4 program deployed into a Programmable Forwarding Device (PFD).

The MRNA Format. Communication from Splicer (in the switch) to Translator (in the host) uses mRNA messages (whose structure we will discuss shortly). The mRNA message summarizes the information obtained by the data plane as it parsed and processed the packet. By summarizing, we do not need to send a complete set of headers (from L2 to L7) for Zeek to dissect. Instead, we send preprocessed data to the Translator, which converts mRNA messages into Zeek-native events. The more we can implement within the RNA Switch Engine, the less Zeek will have to do. As a result, we are effectively offloading packet analysis to the programmable switch. For protocols we have not yet offloaded, our Switch Engine appends the original headers to the mRNA message. Later, the RNA Translator strips the mRNA header and forwards the original PDU to the next Zeek-native analyzer in the chain (typically, the IP analyzer, as introduced in Section 2.2).

The RNA Framework in Action. To show a minimal working example of how RNA processes IPv4 traffic in general, we will detail the processing of ICMPv4 echo request/reply packets within our solution. When a packet `p` arrives at a switch `s`, it undergoes all the RMT processing stages: programmable parser, ingress pipeline (which executes the RNA Transcriber), buffer, egress pipeline (which executes the RNA Splicer), and deparser (which does not play any major role in our solution, besides dispatching packets; henceforth, it is not mentioned). In this example, the programmable parser (Parse Graph in Figure 4.3) extracts Ethernet, IP, and ICMP headers. Next, the pipelines execute four main tasks, following the Table Flow Graph (Figure 4.3). In the ingress pipeline, the switch checks if the Ethernet frame contains an IP datagram, then applies routing and forwarding match-action tables to `p` and buffers it (black circle). Also in the ingress pipeline, the RNA Transcriber collects essential information about `p` (i.e., that it is an

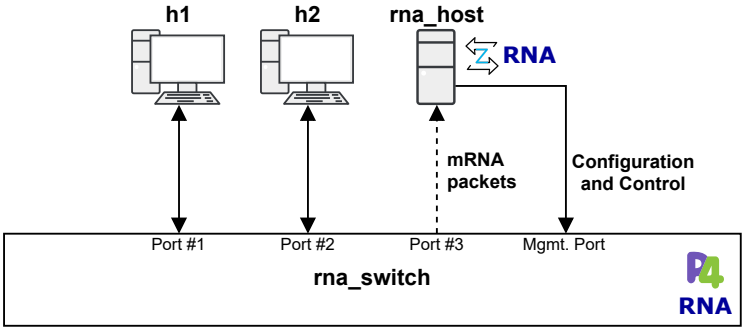
Figure 4.3 – The RNA Framework in Action



Source: the author (2022).

ICMPv4 echo request with a given `id` and `sequence`), copies this information into switch metadata registers, and clones `p` into `p'` (gray circle). Both packets go into the egress pipeline, where they follow different paths. The original packet `p` has its source MAC address updated and is submitted to its destination (white circle on the left). The RNA Splicer manipulates headers to turn the cloned packet `p'` into an mRNA message `m`, containing only essential information (i.e., the mRNA Ethertype, the original IP addresses, and the collected metadata). Finally, the switch submits (white circle on the right) `m` to the Zeek node running the RNA Host Engine. Zeek captures `m` and forwards it to the RNA Translator, which uses `m` to build a Zeek-native event `e` and insert it into the Event Engine queue. Finally, the PSI handles `e`, resulting in the generation of log entries. This entire process mimics what would occur had Zeek itself captured and processed `p`.

Figure 4.4 – Proof-of-Concept Topology



Source: the author (2022).

To demonstrate the RNA operation, we use our proof-of-concept prototype implementation. We instantiate a topology with three hosts connected to a single switch, as shown in Figure 4.4. Host `h1` has IP address 192.0.2.10/24, with default gateway 192.0.2.1 (switch port #1). Host `h2` has IP address 198.51.100.10/24, with default gateway 198.51.100.1 (switch port #2). The `rna_switch` is a P4 device running the RNA Switch Engine. In the Switch Engine, we populate tables to route and forward traffic between `h1` and `h2`. Data collected by the Transcriber is stored temporarily in an internal metadata structure. Figure 4.5 shows our mRNA message format, defined as an `rna_t` header.

Figure 4.5 – mRNA Header (defined in `rna_headers.p4`).

```
1  header rna_t {
2    bit<32> pkt_num;       // In-switch packet counter value.
3    bit<32> src_addr;      // Original source IP address.
4    bit<32> dst_addr;      // Original destination IP address.
5    bit<16> src_port;      // TCP/UDP: original src. port. ICMP: "type" field.
6    bit<16> dst_port;      // TCP/UDP: original dst. port. ICMP: "code" field.
7    bit<16> protocol_l3;   // Copied from L2 "ethertype" field; e.g., 0x0800 = IPv4.
8    bit<8>  protocol_l4;   // Copied from IP "proto" field; e.g., 0x06 = TCP.
9    bit<16> rnatype;       // mRNA message type, defined in rna_constants.p4.
10 }
```

Host `rna_host` runs Zeek and the RNA Host Engine. The Host Engine manages the switch through an out-of-band network interface. Figure 4.6 shows parts of the RNA Manager startup script. The Manager begins by setting up a monitoring session in the switch, so that mRNA packets are sent out through port #3 (lines 5-8). Next, the Manager binds the RNA Translator packet analyzer module to the mRNA (lines 10-13) and IPv4 (lines 14-17) Ethertypes (define in lines 1-2). As a result, when Zeek receives an mRNA packet, it forwards it to the RNA Translator. Similarly, if an mRNA packet encapsulates an IP datagram, the RNA Translator forwards such PDU to the IP analyzer. In the Host Engine, the RNA Translator parses the mRNA message and selects (based on its `protocol_l3`, `protocol_l4`, and `type` fields) the applicable action.

Figure 4.6 – RNA Manager Startup

```
1  const ETHERTYPE_RNA = 0x6606;
2  const ETHERTYPE_IPV4 = 0x0800;
3  event zeek_init() &priority=20
4  {
5    local sscli = "/usr/local/bin/simple_switch_CLI";
6    local sscmd = "mirroring_add 1 3";
7    local command = Exec::Command($cmd = sscli, $stdin = sscmd);
8    when (local result = Exec::run(command)) {}
9    # Error handling omitted.
10   PacketAnalyzer::register_packet_analyzer(
11       PacketAnalyzer::ANALYZER_ETHERNET,
12       ETHERTYPE_RNA,
13       PacketAnalyzer::ANALYZER_RNA);
14   PacketAnalyzer::register_packet_analyzer(
15       PacketAnalyzer::ANALYZER_RNA,
16       ETHERTYPE_IPV4,
17       PacketAnalyzer::ANALYZER_IP);
18 }
```

To test this example, hosts `h1` and `h2` exchange ICMP echo request/reply messages. In the Zeek Host, we observe the output in Figure 4.7. The "`[RNA Handler]`" messages show that the Translator has instantiated RNA Events and correctly identified the ICMPv4 packets. The "`[ICMP Handler]`" messages indicate events generated natively by Zeek.

Figure 4.7 – ICMP Example

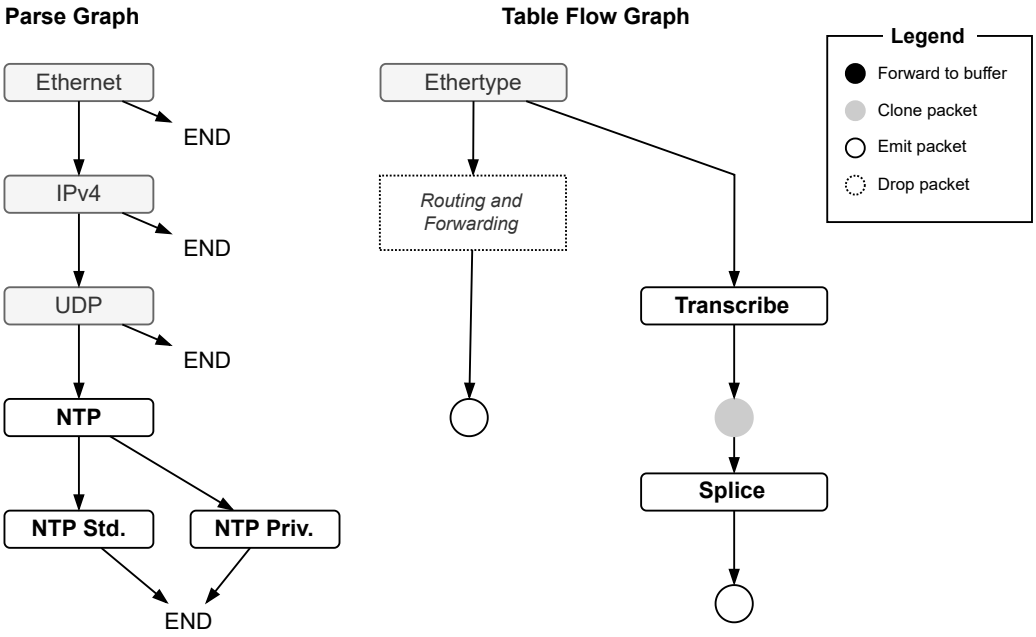
```
[RNA Handler]
    Packet #: 439
    Protocol: IPv4/ICMPv4
    Src: 192.0.2.10. Dst: 198.51.100.10. Type: 8/icmp. Code: 0/icmp.
[ICMP Handler] Echo Request, 192.0.2.10, 198.51.100.10, 121, 204
[RNA Handler]
    Packet #: 440
    Protocol: IPv4/ICMPv4
    Src: 198.51.100.10. Dst: 192.0.2.10. Type: 0/icmp. Code: 0/icmp.
[ICMP Handler] Echo Reply, 192.0.2.10, 198.51.100.10, 121, 204
[RNA Handler]
    Packet #: 441
    Protocol: IPv4/ICMPv4
    Src: 192.0.2.10. Dst: 198.51.100.10. Type: 8/icmp. Code: 0/icmp.
[ICMP Handler] Echo Request, 192.0.2.10, 198.51.100.10, 121, 205
[RNA Handler]
    Packet #: 442
    Protocol: IPv4/ICMPv4
    Src: 198.51.100.10. Dst: 192.0.2.10. Type: 0/icmp. Code: 0/icmp.
[ICMP Handler] Echo Reply, 192.0.2.10, 198.51.100.10, 121, 205
```

4.3 Case Studies

In this section, we demonstrate the feasibility of the RNA Framework. We present three case studies (CSs) as guiding examples. The first two studies relate to cyberattack detection scenarios. CS1 demonstrates how RNA performs lightweight packet inspection to expedite the generation of notifications about a type of semantic attack. CS2 shows how our framework can process DDoS attack notifications coming from EUCLID. Finally, CS3 discusses packet capture filtering. Throughout these studies, we follow a similar outline: (*i*) we describe the focus of the case study, explaining which aspect of RNA we intend to highlight through the study and what approach we will take; (*ii*) we give a detailed account about the interaction between the components of our solution, clarifying how data and control flow through these components, adding specifics about scripts, definitions, and P4 constructs used; (*iii*) we introduce the arrangements required for the demonstration that will follow, enumerating the interconnected nodes (hosts and switches), the necessary configurations, and the strategy we will use in the demonstration (i.e., live traffic or packet traces); and (*iv*) we perform the demonstration, illustrate (via console captures) its operation, and interpret the results we obtain.

Case Study 1: Lightweight Packet Inspection. *Objective:* In this study, we highlight how we can use the RNA Switch Engine to perform lightweight inspection (LPI)

Figure 4.8 – Case Study 1 – Lightweight Packet Inspection



Source: the author (2022).

of application-layer headers. LPI is useful to detect many sorts of attacks. In particular, in this case study, we use LPI to detect a specific type of DRDoS attack vector: NTP Monlist queries, which can exploit misconfigured or vulnerable NTP servers to obtain amplification. We investigate how the Switch Engine can detect these queries and notify the Host Engine about them using mRNA messages. Next, we elaborate on how the Host Engine translates these mRNA messages into events to be handled by the Zeek PSI. Finally, we illustrate how these events are handled.

Operation: This scenario requires preparing the PDP to detect NTP queries and generate the corresponding mRNA messages. Figure 4.8 shows the parse graph and table flow graph for this study. We also need to prepare Zeek to "understand" these messages. For NTP, mRNA messages contain source and destination IP addresses and ports (so we can identify the potential attack target), *l3_protocol* set to *IPv4*, *l4_protocol* set to *UDP*, and *rnatype* set to the `RNATYPE_NTP_MONLIST` constant.

Figure 4.9 shows the relevant parts of our P4 implementation. Within the PDP, we must be able to parse NTP queries. To make this possible, we begin by adding NTP support into our P4 parser, which requires adding header definitions (lines 11-49) and the corresponding states and transitions (lines 54-80). Considering NTP has two header formats (standard and private), the `parse_ntp` state uses its lookahead mechanism to check the `mode` field in the `ntp_flags_t` pseudoheader to determine which type of header it should parse next. In our ingress control, we check whether the packet contains a valid NTP header of the private type (line 89) and then we check if the NTP request code is one of the

Figure 4.9 – Case Study 1 – LPI for NTP on P4 – Code Excerpts

```
1  // Other constants omitted.
2
3  const bit<16> SERVICE_NTP = 123;
4  const bit<8> PRIV_RC_MON_GETLIST   = 20;
5  const bit<8> PRIV_RC_MON_GETLIST_1 = 42;
6  const bit<16> RNATYPE_NTP = 0xA0;
7  const bit<16> RNATYPE_NTP_MONLIST = 0xA1;
8
9  // Other headers omitted.
10
11 // NTP pseudoheader for lookahead.
12 header ntp_flags_t {
13   bit<2> leap;
14   bit<3> version;
15   bit<3> mode;
16 }
17
18 // NTP Header - Modes 1 to 5.
19 header ntp_std_t {
20   bit<2> leap;
21   bit<3> version;
22   bit<3> mode;
23   bit<8> stratum;
24   bit<8> poll;
25   bit<8> precision;
26   bit<32> root_delay;
27   bit<32> root_dispersion;
28   bit<32> reference_id;
29   bit<64> reference_ts;
30   bit<64> origin_ts;
31   bit<64> receive_ts;
32   bit<64> transmit_ts;
33 }
34
35 // NTP Header - Mode 7.
36 header ntp_priv_t {
37   bit<1>  response;
38   bit<1>  more;
39   bit<3>  version;
40   bit<3>  mode;
41   bit<1>  auth;
42   bit<7>  seq;
43   bit<8>  implementation;
44   bit<8>  request_code;
45   bit<4>  err;
46   bit<12> nb_items;
47   bit<4>  mbz;
48   bit<12> data_item_size;
49 }
50
51
52 parser ParserImpl(packet_in p, out headers
       hdr, /* ... */ ) {
53   // Other parser states omitted.
54   state parse_udp {
55     p.extract(hdr.udp);
56     transition select(hdr.udp.s_port, hdr.
         udp.d_port) {
57       (SERVICE_NTP,_): parse_ntp;
58       (_,SERVICE_NTP): parse_ntp;
59       (_,_): accept;
60     }
61   }
62   state parse_ntp {
63     transition select(p.lookahead<ntp_t>().
         mode) {
64       1: parse_ntp_std;
65       2: parse_ntp_std;
66       3: parse_ntp_std;
67       4: parse_ntp_std;
68       5: parse_ntp_std;
69       7: parse_ntp_priv;
70       default: accept;
71     }
72   }
73   state parse_ntp_std {
74     p.extract(hdr.ntp_std);
75     transition accept;
76   }
77   state parse_ntp_priv {
78     p.extract(hdr.ntp_priv);
79     transition accept;
80   }
81 }
82
83 control ingress(inout headers hdr, inout
       metadata meta, /* ... */) {
84   // Action and table definitions omitted.
85   // Other statements omitted.
86   if (hdr.ntp_std.isValid()) {
87     rna_transcribe_ntp();
88   }
89   else if (hdr.ntp_priv.isValid()) {
90     if (hdr.ntp_priv.request_code ==
91           PRIV_RC_MON_GETLIST ||
92         hdr.ntp_priv.request_code ==
93           PRIV_RC_MON_GETLIST_1) {
94       rna_transcribe_ntp_monlist();
95     }
96   }
97 // Other statements omitted.
98 }
```

ones used for Monlist queries (lines 90-93). If that is the case, we set `meta.rnatype` to `RNATYPE_NTP_MONLIST` (line 94, action `rna_transcribe_ntp_monlist()`). Later, in the `egress` control, we will use `meta.rnatype` to set the type of the outgoing mRNA message.

To prepare Zeek for NTP-indicating mRNA messages, we add an event definition `rna_ntp_monlist` and extend our RNA Handler code to generate this type of event when it observes an `rna_message` whose `rnatype` is `RNATYPE_NTP_MONLIST`. As the main handler generates an `rna_ntp_monlist` event, the handler for this specific type of event logs a message/notice indicating that a Monlist query has taken place.

Figure 4.10 – Case Study 1 – NTP Monlist Zeek Output

```
[RNA Handler]
    Packet #:   1
    Protocol: IPv4/UDP/NTP (Monlist!!!)
    Src: 198.51.100.47. Dst: 203.0.113.156. SrcPort: 58031/udp. DstPort: 123/udp.
[UDP Handler] UDP Request, 198.51.100.47, 203.0.113.156, 58031/udp, 123/udp
[RNA Handler]
    Packet #:   2
    Protocol: IPv4/UDP/NTP (Monlist!!!)
    Src: 203.0.113.156. Dst: 198.51.100.47. SrcPort: 123/udp. DstPort: 58031/udp.
[UDP Handler] UDP Reply, 198.51.100.47, 203.0.113.156, 58031/udp, 123/udp
[RNA Handler]
    Packet #:   3
    Protocol: IPv4/UDP/NTP (Monlist!!!)
    Src: 203.0.113.156. Dst: 198.51.100.47. SrcPort: 123/udp. DstPort: 58031/udp.
[UDP Handler] UDP Reply, 198.51.100.47, 203.0.113.156, 58031/udp, 123/udp
[RNA Handler]
    Packet #:   4
    Protocol: IPv4/UDP/NTP (Monlist!!!)
    Src: 198.51.100.47. Dst: 203.0.113.156. SrcPort: 53396/udp. DstPort: 123/udp.
[UDP Handler] UDP Request, 198.51.100.47, 203.0.113.156, 53396/udp, 123/udp
```

Instantiation/Illustration/Discussion: To demonstrate the operation in this case study, we use the same topology shown in Figure 4.4. We instantiate the virtual hosts and generate the traffic we intend to observe. To generate traffic, we replay trace files containing NTP Monlist queries and responses. These packets are injected into one of the `rna_switch` interfaces. The switch parses these packets and generates the corresponding mRNA messages, sending them to Zeek. As Figure 4.10 illustrates, Zeek shows console messages indicating that it has recognized the Monlist trafic represented by the mRNA packets. This example demonstrates that, in general, it is possible to parse application-layer headers directly in the P4 data plane and generate PSI-level notifications without depending on Zeek native parsing mechanisms (which would fully dissect every header from L2 to L7). We can use a similar strategy as the one we used for NTP in order to detect potentially abusive DNSSEC queries. As we expand our detection capabilities, the P4 data plane can help expedite the detection of other types of reflective attacks. The main challenge we anticipate is the handling of application-layer headers and payloads whose syntax exceeds the parsing capabilities of current P4 devices.

Case Study 2: Integration with EUCLID. *Objective:* In this case study, we show how we adapt RNA to enable Zeek to process EUCLID DDoS attack detection and mitigation signaling packets as events. *Operation:* We configure the RNA Switch Engine to recognize EUCLID packets, generate the corresponding mRNA messages, and send these messages to Zeek. We add support for EUCLID mRNA messages to our Host Engine so that each of these messages triggers the generation and handling of a Zeek event. Upon handling, we require that Zeek notifies the operator by showing a console message.

Instantiation: To integrate support to EUCLID notifications into the Switch Engine, we add (*i*) a header definition (Figure 4.11, lines 7-18), (*ii*) a selection case for

Figure 4.11 – Case Study 2 – EUCLID Support – P4 Code Excerpts

```
1  // Other constants omitted.
2  const bit<16> ETHERTYPE_IPV4    = 0x0800;
3  const bit<16> ETHERTYPE_EUCLID = 0x6605;
4  const bit<16> RNATYPE_EUCLID = 0x6605;
5
6  // Other headers omitted.
7  header euclid_t {
8    bit<32> pkt_num;
9    bit<32> src_entropy;
10   bit<32> src_ewma;
11   bit<32> src_ewmmd;
12   bit<32> dst_entropy;
13   bit<32> dst_ewma;
14   bit<32> dst_ewmmd;
15   bit<8> alarm;
16   bit<8> dr_state;
17   bit<16> ethertype;
18 }
19
20 parser ParserImpl(packet_in p, out headers
       hdr, /* ... */ ) {
21
22   state start {
23     transition parse_ethernet;
24   }
25   state parse_ethernet {
26     p.extract(hdr.ethernet);
27     transition select(hdr.ethernet.
           ethertype) {
28       ETHERTYPE_IPV4:   parse_ipv4;
29       ETHERTYPE_IPV6:   parse_ipv6;
30       ETHERTYPE_EUCLID: parse_euclid;
31       default:   accept;
32     }
33   }
34   state parse_euclid {
35     p.extract(hdr.euclid);
36     transition accept;
37   }
38   // Other parser states omitted.
39 }
40
41 control ingress(inout headers hdr, inout
       metadata meta, /* ... */) {
42   // Other statements omitted.
43   action transcribe_euclid() {
44     meta.rnatype = RNATYPE_EUCLID;
45     meta.protocol_l3 = ETHERTYPE_EUCLID;
46   }
47 }
```

`ETHERTYPE_EUCLID` in the `parse_ethernet` state of our P4 parser (line 30), (*iii*) a `parse_euclid` state (lines 34-37), (*iv*) and a `transcribe_euclid()` action (lines 43-46), which the switch triggers for packets with a valid EUCLID header. In the RNA Translator, mRNA messages with type `RNATYPE_EUCLID` are forwarded to the EUCLID packet analyzer, which parses the header and enqueues an `rna_euclid` event for handling by the PSI. In the PSI, we add a handler for EUCLID mRNA messages, which outputs the information contained in the original EUCLID packet.

Illustration/Discussion: To test this use case, we replay a trace containing EUCLID notification packets. Figure 4.12 shows the Zeek PSI output. As we can see, while the RNA Handler indicates that an EUCLID mRNA message has arrived, the EUCLID Handler displays the statistics collected in the last observation window. This case study shows that it is straightforward to extend RNA to support EUCLID notifications. It is worth noting that these notifications can carry arbitrary data coming from their source switches, such as the device address and the notification timestamp.

Case Study 3: Packet Capture Filtering. *Objective:* Recall from Section 2.2 that the Zeek architecture includes a Packet Capture layer that can turn a raw packet stream into a filtered packet stream. This case study aims to demonstrate how we can offload packet capture filtering to a P4 switch. *Operation:* For this case study, we assume that the `libpcap` packet *filter expression* only contains these kinds of primitives: (*i*) `type id` relating to IPv4 network prefixes (e.g., `net 192.0.2.0/24`), (*ii*) `proto port id` relating to TCP and UDP ports (e.g., `tcp port 53` or `udp port 123`), and (*iii*)

Figure 4.12 – Case Study 2 – EUCLID – Zeek PSI Output

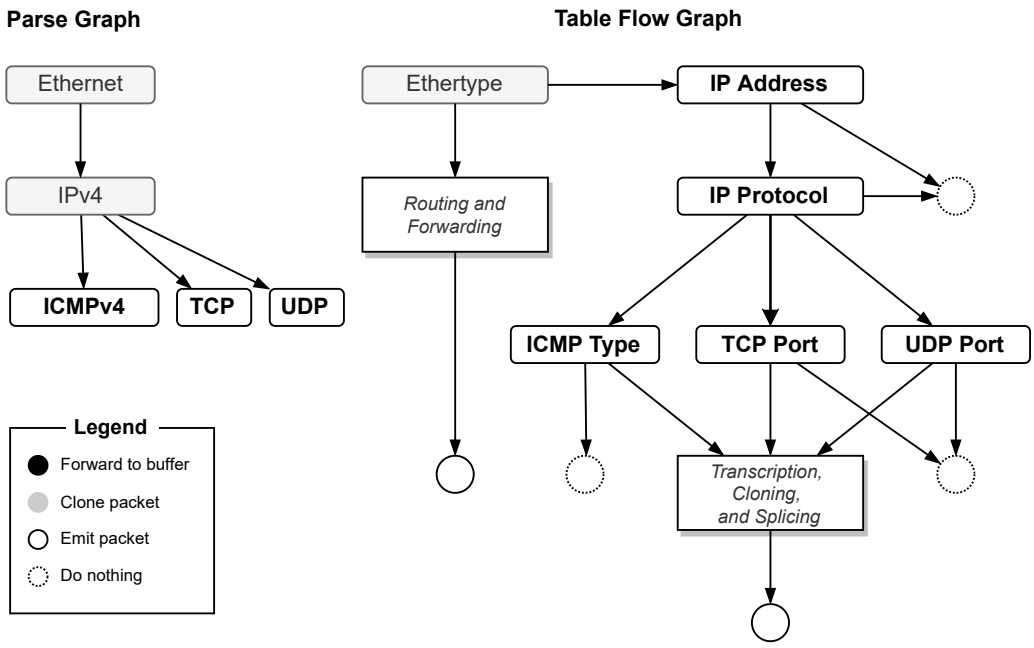
```
[RNA Handler]
    Packet #: 1073
    Protocol: Euclid
[Euclid Handler]
    Src: Entropy 11.62, EWMA 11.65, EWMMD 0.0423
    Dst: Entropy 11.44, EWMA 11.41, EWMMD 0.0630
    Alarm: OFF. Defense Readiness State: Safe.
[RNA Handler]
    Packet #: 1074
    Protocol: Euclid
[Euclid Handler]
    Src: Entropy 11.62, EWMA 11.65, EWMMD 0.0407
    Dst: Entropy 11.38, EWMA 11.41, EWMMD 0.0608
    Alarm: OFF. Defense Readiness State: Safe.
[RNA Handler]
    Packet #: 1075
    Protocol: Euclid
[Euclid Handler]
    Src: Entropy 10.88, EWMA 11.65, EWMMD 0.0407
    Dst: Entropy 10.31, EWMA 11.41, EWMMD 0.0608
    Alarm: ON. Defense Readiness State: Active.
[RNA Handler]
    Packet #: 1076
    Protocol: Euclid
[Euclid Handler]
    Src: Entropy 11.38, EWMA 11.65, EWMMD 0.0407
    Dst: Entropy 10.19, EWMA 11.41, EWMMD 0.0608
    Alarm: ON. Defense Readiness State: Active.
```

ICMP type checks (e.g., `icmptype==icmp-echo`)[1]. Figure 4.13 shows the packet capture filtering parse graph and table flow graph for this study. The P4 switch parses headers from layers 2 to 4. In the Parse Graph, we highlight the states relevant to this case study: ICMPv4, TCP, and UDP. Next, a chain of match-action tables determines whether or not to consider the packet for inspection. In the Table Flow Graph, we highlight the tables IP Address, IP Protocol, ICMP Type, TCP Port, and UDP Port. In P4, these tables are implemented as `filter_net_src`, `filter_net_dst`, `filter_icmp_type`, `filter_tcp_src_port`, `filter_tcp_dst_port`, `filter_udp_src_port`, and `filter_udp_dst_port`. For packets matching the IP Address table, we apply the action `set_inspect_l3()`, which sets an internal metadata flag indicating that the L3 filtering condition has been met. Similarly, for packets matching any of the L4 tables, we apply the action `set_inspect_l4()`. Finally, we check whether both L3 and L4 conditions have been met, in which case the packet is considered relevant for inspection. Relevant packets undergo the *Transcription, Cloning, and Splicing* chain (shown as a rectangle in the graph), which results in mRNA packet generation. Irrelevant packets are not cloned (as indicated by the dotted circles).

Instantiation: For instance, suppose we are interested in IPv4 traffic involving one of our local networks (i.e., 192.0.2.0/24 and 198.51.100.0/24). Furthermore, consider we are only interested in TCP port 80, UDP port 123, and ICMP types *Echo* and *Echo Reply*.

[1]These are `libpcap` constants referring, respectively, to the *ICMP Type* header field and its value.

Figure 4.13 – Case Study 3 – Packet Capture Filtering



Source: the author (2022).

The resulting filter expression would be:

```
(net 192.0.2.0/24 or net 198.51.100.0/24) and
(tcp port 80 or udp port 123 or icmptype==icmp-echo or icmptype==icmp-echoreply).
```

Figure 4.14 – Case Study 3 – Populating filtering tables through the switch CLI.

```
1  # Syntax: table_add table action key => parameters
2  table_add filter_net_src set_inspect_l3 192.0.2.0/24  =>
3  table_add filter_net_dst set_inspect_l3 192.0.2.0/24  =>
4  table_add filter_net_src set_inspect_l3 198.51.100.0/24 =>
5  table_add filter_net_dst set_inspect_l3 198.51.100.0/24 =>
6  table_add filter_tcp_src_port set_inspect_l4 80 =>
7  table_add filter_tcp_dst_port set_inspect_l4 80 =>
8  table_add filter_udp_src_port set_inspect_l4 123 =>
9  table_add filter_udp_dst_port set_inspect_l4 123 =>
10 table_add filter_icmp_type set_inspect_l4 0 =>
11 table_add filter_icmp_type set_inspect_l4 8 =>
```

Illustration/Discussion: To deploy these primitives to the Switch Engine, we use the `simple_switch_cli` commands shown in Figure 4.14 to insert entries in the filtering tables. Lines 2-5 assign the action `set_inspect_l3` to the network prefixes we are interested in. Lines 6-11 assign the action `set_inspect_l4` to the L4 ports and ICMP types we intend to inspect. We applied the commands above to our testbed and observed that the rules are enforced by the switch, i.e., only the traffic of interest undergoes the Transcriber and Splicer stages. This case study integrates into the RNA Switch Engine the conditional replication mechanism we proposed in a previous work (ILHA, 2019).

5 CONCLUSION AND FUTURE WORK

In this thesis, we investigated the potential of Programmable Data Planes (PDPs) as a foundation for cybersecurity solutions. Our work has two iterations. In the first iteration, we proposed EUCLID, a novel real-time DDoS attack detection and mitigation mechanism that can be executed entirely in a P4 forwarding device. This work showed that our P4-based design has the potential to meet increasingly strict performance requirements in high-volume networks. As another significant contribution, we shared lessons learned during the design, implementation, and evaluation of EUCLID, hoping these insights may be helpful for future research on programmable networks. We also shared with the community the source code of our prototype implementation and of our analysis toolkit, which can be found at <https://www.github.com/asilha/euclid>. The datasets used in the evaluation can be obtained from CAIDA (CAIDA, 2007; CAIDA, 2016).

Our experimental evaluation indicated that EUCLID can detect and mitigate the effects of DDoS outbreaks quickly and accurately. For instance, in a link with a traffic rate of one million packets per second, EUCLID detects attacks and launches its mitigation mechanism within 250 ms. Attack detection is over 90% accurate, correctly signaling most DDoS incidents while keeping the proportion of false alarms below 10%. We observed that the detection accuracy of our mechanism is superior to that of an approach based on packet sampling. We estimate that deploying detection on all interfaces of a 10 Gbps switch requires a total of 9 MB of SRAM, well within the capacity of existing programmable data plane targets. Attack mitigation correctly identifies more than 96% of malicious packets as suspects, with a false-positive rate (FPR) smaller than 1%. EUCLID can effectively steer the attack traffic away from its intended target, thus preventing service outage. The low FPR ensures that legitimate users can still access the protected service. Our mitigation components require an additional 375 kB of SRAM per 10 Gbps link.

In the second iteration, we leveraged the experience accumulated and the lessons learned during the work on EUCLID to research a general approach for cyberattack detection using PDPs. Considering that defense against advanced persistent threats (APTs) typically requires a Network Intrusion Detection System (NIDS), we observed that NIDSes face relevant scalability problems. These issues stem from (*i*) packet copies from the forwarding plane to the main memory of a general-purpose CPU-based computer and (*ii*) stateful inspection and payload analysis upon copied packets. We glanced at an opportunity to tackle this challenge by putting PDPs to work on the broad range of operations required by a NIDS. As a response, we introduced Reconfigurable Network Analytics – RNA, an innovative framework to offload NIDS-related operations from general-purpose CPUs

to high-performance PDPs. RNA uses the mechanisms of a programmable switch to analyze traffic, summarize information about it, and send these summaries to a host-based component, which, in turn, translates these summaries into events the NIDS can handle.

Using the BMv2 P4 switch and the Zeek Network Security Monitor as platforms, we built a proof-of-concept implementation of our framework. We published our prototype source code, which can be obtained at <https://www.github.com/asilha/rna>. Through a series of examples and case studies, we demonstrated the feasibility of our design and its integration with Zeek. Specifically, we showed that: (*i*) we can automate monitoring session setup, (*ii*) it is possible to offload lightweight packet inspection to the PDP, (*iii*) RNA can forward EUCLID alarms to Zeek, and (*iv*) we can filter traffic for Zeek in the PDP. From these examples and studies, we also concluded that we can gradually add data plane support for more protocols and adapt our framework to identify higher-level network events. Furthermore, we can do this without modifying Zeek (other than loading into it our plugin package). As RNA capabilities grow, we reduce the need for Zeek to do all the CPU-intensive packet analysis by itself. In this work, we used an *ad-hoc* approach to develop our first offloading strategies. Our goal is not "to improve Zeek" but "to improve intrusion detection performance in general." Offloading complex traffic analysis tasks to a programmable data plane can significantly improve IDS scalability and timeliness. We envision a "smart" network that can perform arbitrary IDS tasks as fast as possible. Our results suggest that there is an exciting research avenue towards automatic offloading. We expect our findings will help lay the ground for automated translation of high-level IDS policies into PDP-deployable code.

Opportunities for future work. We plan on expanding the RNA framework capabilities, such as (*i*) adding LPI to other protocols, (*ii*) implementing load balancing to distribute traffic across worker nodes in a Zeek cluster, and (*iii*) devising methods to detect internal reconnaissance activities (such as host- and port-scanning). Data plane-based application protocol dissection is a core issue. Operations that require reassembling packet fragments or byte streams are exceedingly demanding for a P4 device. An automated approach for a systematic translation strategy requires compiling IDS policy scripts to generate code and control instructions for Zeek and PDPs. Our research group is already working on an automated solution called Zeek P4 Optimizer (ZPO). ZPO fetches events of interest and generates the code needed to offload the detection of these events to the PDP.

REFERENCES

AFEK, Y.; BREMLER-BARR, A.; SHAFIR, L. Network anti-spoofing with SDN data plane. In: **IEEE INFOCOM 2017 - IEEE Conference on Computer Communications**. New York, NY, USA: IEEE, 2017. p. 1–9.

AHMED, M.; MAHMOOD, A. N.; HU, J. A survey of network anomaly detection techniques. **Journal of Network and Computer Applications**, Elsevier, v. 60, p. 19–31, 2016.

AL-MOHANNADI, H. et al. Cyber-attack modeling analysis techniques: An overview. In: IEEE. **2016 IEEE 4th international conference on future internet of things and cloud workshops (FiCloudW)**. New York, NY, USA: IEEE, 2016. p. 69–76.

ALCOZ, A. G. et al. Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense. In: **Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2022. (SIGCOMM '18). Available from Internet: <https://nsg.ee.ethz.ch/fileadmin/user_upload/sigcomm22-final615.pdf>.

ALSABEH, A. et al. A survey on security applications of P4 programmable switches and a STRIDE-based vulnerability assessment. **Computer Networks**, Elsevier BV, v. 207, p. 108800, abr. 2022. ISSN 1389-1286. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S1389128622000287>.

ALSHAMRANI, A. et al. A Survey on Advanced Persistent Threats: Techniques, Solutions, Challenges, and Research Opportunities. **IEEE Communications Surveys & Tutorials**, IEEE, v. 21, n. 2, p. 1851–1877, 2019.

Barefoot Networks. **Tofino: World's Fastest P4-Programmable Ethernet Switch ASICs**. 2020. Available from Internet: <https://barefootnetworks.com/products/brief-tofino/>.

BHATT, P.; YANO, E. T.; GUSTAVSSON, P. Towards a framework to detect multi-stage advanced persistent threats attacks. In: IEEE. **2014 IEEE 8th international symposium on service oriented system engineering**. New York, NY, USA: IEEE, 2014. p. 390–395.

BHUYAN, M. H.; BHATTACHARYYA, D. K.; KALITA, J. K. An empirical evaluation of information metrics for low-rate and high-rate DDoS attack detection. **Pattern Recognition Letters**, v. 51, p. 1–7, 2015. ISSN 0167-8655. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S016786551400244X>.

BIANCHI, G. et al. OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 44, n. 2, p. 44–51, abr. 2014. ISSN 0146-4833.

BOITE, J. et al. Statesec: Stateful monitoring for DDoS protection in software defined networks. In: **2017 IEEE Conference on Network Softwarization (NetSoft)**. New York, NY, USA: IEEE, 2017. p. 1–9.

BONFIM, M. et al. A real-time attack defense framework for 5G network slicing. **Software, Practice & Experience**, Wiley, v. 50, n. 7, p. 1228–1257, feb. 2020.

BOSSHART, P. et al. P4: Programming Protocol-independent Packet Processors. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833.

BOSSHART, P. et al. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In: **Proceedings of the ACM SIGCOMM 2013 Conference**. New York, NY, USA: ACM, 2013. (SIGCOMM '13), p. 99–110. ISBN 978-1-4503-2056-6.

BROUER, J. D. **eBPF - extended Berkeley Packet Filter**. 2016. Retrieved on 2018-12-01. Available from Internet: <https://prototype-kernel.readthedocs.io/en/latest/bpf/>.

BURAGLIO, N. **Overview of the Bro Intrusion Detection System (webinar)**. 2015. Energy Sciences Network. Retrieved on 2018-11-15. Available from Internet: <http://fasterdata.es.net/science-dmz/more-references/esnet-helpful-talks-and-tutorials/overview-of-the-bro-intrusion-detection-system/>.

CAIDA. **The CAIDA UCSD DDoS Attack 2007 Dataset**. 2007. Available from Internet: <http://www.caida.org/data/passive/ddos-20070804_dataset.xml>.

CAIDA. **The CAIDA UCSD Anonymized Internet Traces 2016**. 2016. Available from Internet: <http://www.caida.org/data/passive/passive_2016_dataset.xml>.

CHARIKAR, M.; CHEN, K.; FARACH-COLTON, M. Finding Frequent Items in Data Streams. In: WIDMAYER, P. et al. (Ed.). **Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. (Lecture Notes in Computer Science, v. 2380), p. 693–703. ISBN 978-3-540-45465-6. Available from Internet: <https://doi.org/10.1007/3-540-45465-9_59>.

CHEN, P.; DESMET, L.; HUYGENS, C. A Study on Advanced Persistent Threats. In: DECKER, B. D.; ZÚQUETE, A. (Ed.). **Communications and Multimedia Security**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 63–72. ISBN 978-3-662-44885-4.

CHOLE, S. et al. DRMT: Disaggregated Programmable Switching. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 1–14. ISBN 9781450346535.

Cisco Systems. **Snort FAQ – What is Snort?** 2022. Available from Internet: <https://www.snort.org/faq/what-is-snort>.

Cisco Systems, Inc. **What Is an Advanced Persistent Threat (APT)?** 2022. Available from Internet: <https://www.cisco.com/c/en/us/products/security/advanced-persistent-threat.html>.

CLAISE, B. RFC, **Cisco Systems NetFlow Services Export Version 9**. RFC Editor, 2004. 1–33 p. Internet Requests for Comments. Available from Internet: <http://www.rfc-editor.org/rfc/rfc3954.txt>.

CORDEIRO, W. L. da C.; MARQUES, J. A.; GASPARY, L. P. Data Plane Programmability Beyond OpenFlow: Opportunities and Challenges for Network and Service Operations and

Management. **Journal of Network and Systems Management**, v. 25, n. 4, p. 784–818, oct. 2017. ISSN 1573-7705.

CORMODE, G. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. **Foundations and Trends in Databases**, v. 4, n. 1–3, p. 1–294, 2011. ISSN 1931-7883. Available from Internet: <https://www.nowpublishers.com/article/Details/DBS-004>.

CORMODE, G.; MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. **Journal of Algorithms**, v. 55, n. 1, p. 58–75, 2005. ISSN 0196-6774. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0196677403001913>.

DACIER, M. C. et al. Security Challenges and Opportunities of Software-Defined Networking. **IEEE Security & Privacy**, v. 15, n. 2, p. 96–100, mar. 2017. ISSN 1540-7993.

DALMAZO, B. L. et al. A systematic review on distributed denial of service attack defense mechanisms in programmable networks. **International Journal of Network Management**, v. 31, n. 6, p. e2163, 2021.

DING, D.; SAVI, M.; SIRACUSA, D. Estimating Logarithmic and Exponential Functions to Track Network Traffic Entropy in P4. In: **2020 IEEE/IFIP Network Operations and Management Symposium (NOMS)**. New York, NY, USA: IEEE, 2020.

DREGER, H. et al. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In: KEROMYTIS, A. D. (Ed.). **15th USENIX Security Symposium (USENIX Security 06)**. Vancouver, BC, Canada: USENIX Association, 2006. p. 257–272. Available from Internet: <https://www.usenix.org/conference/15th-usenix-security-symposium/dynamic-application-layer-protocol-analysis-network>.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The Road to SDN: An Intellectual History of Programmable Networks. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 44, n. 2, p. 87–98, abr. 2014. ISSN 0146-4833.

FRIEDBERG, I. et al. Combating advanced persistent threats: From network event correlation to incident detection. **Computers & Security**, Elsevier, v. 48, p. 35–57, 2015.

GUPTA, A. et al. Network Monitoring As a Streaming Analytics Problem. In: **Proceedings of the 15th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: ACM, 2016. (HotNets '16), p. 106–112. ISBN 978-1-4503-4661-0.

GUPTA, A. et al. Sonata: Query-driven streaming network telemetry: query-driven streaming network telemetry. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2018. p. 357–371.

HOQUE, N.; BHATTACHARYYA, D. K.; KALITA, J. K. Botnet in DDoS Attacks: Trends and Challenges. **IEEE Communications Surveys & Tutorials**, v. 17, n. 4, p. 2242–2270, 2015. ISSN 1553-877X.

HU, Q.; YU, S.-Y.; ASGHAR, M. R. Analysing performance issues of open-source intrusion detection systems in high-speed networks. **Journal of Information Security and Applications**, v. 51, p. 102426, 2020. ISSN 2214-2126. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S2214212619306003>.

HUMMEL, R.; HILDEBRAND, C. **NETSCOUT Threat Intelligence Report - Issue 7: Findings From 1H 2021 - The Long Tail of Attacker Innovation**. Westford, MA, USA, 2021. Retrieved on 2022-02-23. Available from Internet: <https://www.netscout.com/threatreport>. Accessed in: 2022-02-23.

HUTCHINS, E. M.; CLOPPERT, M. J.; AMIN, R. M. Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. In: ARMISTEAD, L. (Ed.). **Proceedings of the 6th International Conference on Information Warfare and Security**. Reading, UK: Academic Publishing International Limited, 2011. p. 113–125. Available from Internet: <https://www.lockheedmartin.com/content/dam/lockheed-martin/rms/documents/cyber/LM-White-Paper-Intel-Driven-Defense.pdf>.

ILHA, A. S. **Accelerating Real-Time Intrusion Detection by Offloading Capture Filters to P4 Programmable Switches**. 2019. Institute of Informatics - UFRGS. Monograph (Specialization). Advisor: Luciano Paschoal Gaspary. Available from Internet: <https://www.inf.ufrgs.br/~asilha/artidp4.pdf>.

ILHA, A. S. et al. Euclid: A Fully In-Network, P4-Based Approach for Real-Time DDoS Attack Detection and Mitigation. **IEEE Transactions on Network and Service Management**, Institute of Electrical and Electronics Engineers (IEEE), v. 18, n. 3, p. 3121–3139, sep. 2021.

JACOBSON, V.; LERES, C.; MCCANNE, S. **The tcpdump manual page**. Berkeley, CA, USA, 1989.

JACOBSON, V.; LERES, C.; MCCANNE, S. **libpcap**. 1994. Lawrence Berkeley Laboratory, Berkeley, CA. Retrieved on 2018-12-01. Available from Internet: <https://www.tcpdump.org/>.

KICINSKI, J.; VILJOEN, N. Comprehensive XDP offload - handling the edge cases. In: **NetDev 2.2**. NetDev Society, 2017. Retrieved on 2018-12-01. Available from Internet: <https://netdevconf.org/1.2/session.html?jakub-kicinski>.

KIM, C.; LEE, J. **Programming the network dataplane. ACM SIGCOMM Tutorial.** 2016. Available from Internet: <https://conferences.sigcomm.org/sigcomm/2016/files/program/netpl/netpl16-kim.pdf>.

KREUTZ, D. et al. Software-Defined Networking: A Comprehensive Survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, jan. 2015. ISSN 0018-9219.

KRISHNAMURTHY, B. et al. Sketch-Based Change Detection: Methods, Evaluation, and Applications. In: **Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement**. New York, NY, USA: Association for Computing Machinery, 2003. (IMC '03), p. 234–247. ISBN 1581137737.

KUROSE, J. F.; ROSS, K. W. **Computer Networking: A Top-Down Approach**. Seventh edition. Harlow, Essex, England: Pearson Education, 2017. ISBN 978-0-13-359414-0.

LAKHINA, A.; CROVELLA, M.; DIOT, C. Mining Anomalies Using Traffic Feature Distributions. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 35, n. 4, p. 217–228, aug. 2005. ISSN 0146-4833.

LAPOLLI, A. C. **Offloading Real-time DDoS Attack Detection to Programmable Data Planes**. Dissertation (Master) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, RS, Brazil, 2019. Available from Internet: <https://lume.ufrgs.br/handle/10183/204658>. Accessed in: 2021-11-15.

LAPOLLI, A. C.; MARQUES, J. A.; GASPARY, L. P. Offloading Real-time DDoS Attack Detection to Programmable Data Planes. In: **2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. IEEE, 2019. p. 19–27. ISBN 978-3-903176-15-7. ISSN 1573-0077. Available from Internet: <https://ieeexplore.ieee.org/document/8717869>.

LI, G. et al. Enabling Performant, Flexible and Cost-Efficient DDoS Defense With Programmable Switches. **IEEE/ACM Transactions on Networking**, v. 29, n. 4, p. 1509–1526, aug. 2021. ISSN 1558-2566.

LI, H. et al. vNIDS: Towards Elastic Security with Safe and Efficient Virtualization of Network Intrusion Detection Systems. In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2018. (CCS '18), p. 17–34. ISBN 978-1-4503-5693-0.

LIU, Z. et al. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 101–114. ISBN 978-1-4503-4193-6.

MARCHETTI, M. et al. Analysis of high volumes of network traffic for advanced persistent threat detection. **Computer Networks**, Elsevier, v. 109, p. 127–141, 2016.

MARROW, A.; STOLYAROV, G. **Russia's Yandex says it repelled biggest DDoS attack in history**. 2021. Reuters. Available from Internet: <https://www.reuters.com/technology/russias-yandex-says-it-repelled-biggest-ddos-attack-history-2021-09-09/>.

MCCANNE, S.; JACOBSON, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: **Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings**. Berkeley, CA, USA: USENIX Association, 1993. (USENIX'93), p. 2–2. Available from Internet: <http://dl.acm.org/citation.cfm?id=1267303.1267305>.

MCKEOWN, N. **Creating an end-to-end programming model for packet forwarding**. 2020. Netdev 0x14. Available from Internet: <https://tinyurl.com/tenfx2r8>.

MCKEOWN, N. et al. OpenFlow: Enabling Innovation in Campus Networks. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833.

MCWHORTER, D. **Mandiant Exposes APT1–One of China's Cyber Espionage Units & Releases 3,000 Indicators**. Alexandria, VA, USA, 2013. Available from Internet: <https://www.mandiant.com/resources/apt1-exposing-one-of-chinas-cyber-espionage-units>.

MIRKOVIC, J.; REIHER, P. A taxonomy of DDoS attack and DDoS defense mechanisms. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 34, n. 2, p. 39–53, 2004.

MOSHREF, M.; YU, M.; GOVINDAN, R. Resource/Accuracy Tradeoffs in Software-defined Measurement. In: **Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: ACM, 2013. (HotSDN '13), p. 73–78. ISBN 978-1-4503-2178-5.

MURALEEDHARAN, N.; JANET, B. Behaviour analysis of HTTP based slow denial of service attack. In: **2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)**. New York, NY, USA: IEEE, 2017. p. 1851–1856.

NARAYANA, S. et al. Language-Directed Hardware Design for Network Performance Monitoring. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 85–98. ISBN 978-1-4503-4653-5.

NICHOLSON, P. **AWS hit by Largest Reported DDoS Attack of 2.3 Tbps**. 2020. A10 Networks. Available from Internet: <https://www.a10networks.com/blog/aws-hit-by-largest-reported-ddos-attack-of-2-3-tbps/>.

PAOLUCCI, F. et al. P4 Edge Node Enabling Stateful Traffic Engineering and Cyber Security. **IEEE/OSA Journal of Optical Communications and Networking**, OSA, v. 11, n. 1, p. A84–A95, jan. 2019. Available from Internet: <http://jocn.osa.org/abstract.cfm?URI=jocn-11-1-A84>.

PAXSON, V. Bro: A System for Detecting Network Intruders in Real-time. **Computer Networks**, Elsevier North-Holland, Inc., New York, NY, USA, v. 31, n. 23-24, p. 2435–2463, dec. 1999. ISSN 1389-1286.

PENG, T.; LECKIE, C.; RAMAMOHANARAO, K. Survey of network-based defense mechanisms countering the DoS and DDoS problems. **ACM Computing Surveys**, ACM New York, NY, USA, v. 39, n. 1, 2007.

PHAAL, P. **sFlow: Sampling Rates**. 2009. Available from Internet: <https://blog.sflow.com/2009/06/sampling-rates.html>.

PHAAL, P.; PANCHEN, S.; MCKEE, N. RFC, **InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks**. RFC Editor, 2001. 1–31 p. Internet Requests for Comments. Available from Internet: <http://www.rfc-editor.org/rfc/rfc3176.txt>.

RADACK, S. M. et al. **Managing Information Security Risk: Organization, Mission, and Information System View**. Gaithersburg, MD, USA, 2011.

ROBERTS, S. W. Control Chart Tests Based on Geometric Moving Averages. **Technometrics**, Taylor & Francis, v. 1, n. 3, p. 239–250, 1959.

ROSSOW, C. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In: **Proceedings 2014 Network and Distributed System Security Symposium**. Reston, VA, USA: Internet Society, 2014.

SHANNON, C. E. A Mathematical Theory of Communication. **Bell Systems Technical Journal**, v. 27, p. 623–656, 1948.

SHIN, S. et al. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks. In: **Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security**. New York, NY, USA: ACM, 2013. (CCS '13), p. 413–424. ISBN 978-1-4503-2477-9.

SIVARAMAN, V. et al. Heavy-Hitter Detection Entirely in the Data Plane. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 164–176. ISBN 978-1-4503-4947-5.

SOOD, A. K.; ENBODY, R. J. Targeted Cyberattacks: A Superset of Advanced Persistent Threats. **IEEE Security & Privacy**, IEEE, v. 11, n. 1, p. 54–61, jan. 2013. ISSN 1558-4046.

STOJANOVIĆ, B.; HOFER-SCHMITZ, K.; KLEB, U. APT datasets and attack modeling for automated detection methods: A review. **Computers & Security**, Elsevier, v. 92, p. 101734, 2020.

STROM, B. E. et al. **MITRE ATT&CK®: Design and Philosophy**. McLean, VA, USA, 2020. Available from Internet: <https://www.mitre.org/sites/default/files/publications/pr-19-01075-28-mitre-attack-design-and-philosophy.pdf>.

SWAMI, R.; DAVE, M.; RANGA, V. Software-Defined Networking-Based DDoS Defense Mechanisms. **ACM Computing Surveys**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 2, abr. 2019. ISSN 0360-0300.

TANKARD, C. Advanced persistent threats and how to monitor and deter them. **Network security**, Elsevier, v. 2011, n. 8, p. 16–19, 2011.

TAVARES, K.; FERRETO, T. DDoS on Sketch: Spoofed DDoS attack defense with programmable data planes using sketches in SDN. In: **Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)**. Porto Alegre, RS, Brazil: SBC, 2019. v. 37, p. 805–819. ISSN 2177-9384. Available from Internet: <https://sol.sbc.org.br/index.php/sbrc/article/view/7404>.

The MITRE Corporation. **The MITRE ATT&CK®Matrix for Enterprise - Version 11**. 2022. The MITRE Corporation. Version 11. Available from Internet: <https://attack.mitre.org/versions/v11/matrices/enterprise/>.

The Open Information Security Foundation. **Suricata User Guide**. 2022. Available from Internet: <https://suricata.readthedocs.io/>.

The Open Networking Foundation. **OpenFlow Switch Specification Version 1.5.1**. 2015.

The P4 Language Consortium. **BMv2**. 2020. Available from Internet: <https://github.com/p4lang/behavioral-model>.

The Zeek Project. **The Zeek Network Security Monitor**. 2022. Retrieved on 2022-01-03. Available from Internet: <https://docs.zeek.org/en/master/about.html>.

USSATH, M. et al. Advanced persistent threats: Behind the scenes. In: IEEE. **2016 Annual Conference on Information Science and Systems (CISS)**. New York, NY, USA, 2016. p. 181–186.

VAHDAT, A. **Coming of Age in the Fifth Epoch of Distributed Computing: The Power of Sustained Exponential Growth – ACM SIGCOMM 2020 Keynote**. 2020. Association for Computing Machinery. Available from Internet: <https://tinyurl.com/bdh8pjf3>.

VALDOVINOS, I. A. et al. Emerging DDoS attack detection and mitigation strategies in software-defined networks: Taxonomy, challenges and future directions. **Journal of Network and Computer Applications**, v. 187, p. 103093, 2021. ISSN 1084-8045. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S1084804521001156>.

WANG, C. et al. SkyShield: A Sketch-Based Defense System Against Application Layer DDoS Attacks. **IEEE Transactions on Information Forensics and Security**, Institute of Electrical and Electronics Engineers (IEEE), v. 13, n. 3, p. 559–573, mar. 2018. ISSN 1556-6021.

WARREN, T. **Microsoft says it mitigated one of the largest DDoS attacks ever recorded**. 2021. The Verge. Available from Internet: <https://www.theverge.com/2021/10/12/22722155/microsoft-azure-biggest-ddos-attack-ever-2-4-tbps>.

XIANG, Y.; LI, K.; ZHOU, W. Low-Rate DDoS Attacks Detection and Traceback by Using New Information Metrics. **IEEE Transactions on Information Forensics and Security**, v. 6, n. 2, p. 426–437, 2011.

XING, J.; WU, W.; CHEN, A. Architecting Programmable Data Plane Defenses into the Network with FastFlex. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2019. (HotNets '19), p. 161–169. ISBN 9781450370202.

XU, Y.; LIU, Y. DDoS attack detection under SDN context. In: **IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications**. New York, NY, USA: IEEE, 2016. p. 1–9.

YANG, T. et al. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 561–575. ISBN 978-1-4503-5567-4.

YOACHIMIK, O. **Cloudflare thwarts 17.2M rps DDoS attack—the largest ever reported**. 2021. The Cloudflare Blog. Available from Internet: <https://blog.cloudflare.com/cloudflare-thwarts-17-2m-rps-ddos-attack-the-largest-ever-reported/>.

YU, M.; JOSE, L.; MIAO, R. Software Defined Traffic Measurement with OpenSketch. In: FEAMSTER, N.; MOGUL, J. C. (Ed.). **10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)**. Lombard, IL: USENIX Association, 2013. p. 29–42. ISBN 978-1-931971-00-3. Available from Internet: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/yu>.

ZARGAR, S. T.; JOSHI, J.; TIPPER, D. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. **IEEE Communications Surveys & Tutorials**, v. 15, n. 4, p. 2046–2069, 2013. ISSN 1553-877X.

ZHANG, M. et al. Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches. In: **NDSS**. Internet Society, 2020. Available from Internet: <https://bit.ly/2vZviRE>.

ZHAO, Z. et al. Achieving 100Gbps Intrusion Prevention on a Single Server. In: **14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)**. USENIX Association, 2020. p. 1083–1100. ISBN 978-1-939133-19-9. Available from Internet: <https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng>.

APPENDIX A — RESUMO EXPANDIDO

Rumo a uma Solução Geral para Detecção de Ataques Cibernéticos Baseada em Planos de Dados Programáveis

Os sistemas conectados à Internet têm sido cada vez mais alvos de diversos tipos de ataques cibernéticos, que causam danos e prejuízos relevantes a sistemas corporativos e governamentais, incluindo infraestruturas críticas. Duas amplas categorias de ataque estão na vanguarda: *ataques distribuídos de negação de serviço* (DDoS) e *ameaças persistentes avançadas* (APTs). Os ataques DDoS continuam sendo a ameaça mais grave à segurança dos sistemas em rede (HUMMEL; HILDEBRAND, 2021). Campanhas cada vez mais frequentes e intensas constantemente ganham as manchetes por causar interrupções até mesmo em grandes provedores de serviços *online* (p.ex., Yandex (MARROW; STOLYAROV, 2021), Cloudflare (YOACHIMIK, 2021), Microsoft Azure (WARREN, 2021) e Amazon Web Services (NICHOLSON, 2020)). As taxas de transferência de dados geradas durante os ataques atingem vários terabits por segundo e inundam até mesmo enlaces de alta capacidade. Da mesma forma, ataques que atingem bilhões de pacotes ou milhões de solicitações por segundo podem inundar rapidamente dispositivos de encaminhamento e servidores de rede. Historicamente, produzir esses tsunamis digitais exigia o controle de numerosas fontes de ataque. No entanto, o surgimento de técnicas de amplificação dispensou esse requisito e originou os ataques do tipo *Distributed Reflective Denial-of-Service* (DRDoS) (ROSSOW, 2014).

A Ascensão das APTs. Assim como os ataques DDoS, as APTs tornaram-se cada vez mais proeminentes ao longo do tempo (ALSHAMRANI et al., 2019). Em contraste com os ataques DDoS, cujos efeitos geram alarde, as incursões de APTs são furtivas e podem passam despercebidas por longos períodos, até mesmo anos (MCWHORTER, 2013). Apesar de sorrateiras, as APTs expõem os alvos a danos duradouros ou mesmo permanentes—o que pode incluir a sabotagem de sistemas ciberfísicos. Em geral, atacantes buscam obter acesso a dados críticos, sensíveis ou estratégicos, contendo informações importantes, com o objetivo último de exfiltrar, corromper ou até mesmo destruir tais dados (CHEN; DESMET; HUYGENS, 2014). Uma campanha típica de APT encaixa-se em modelos como o *Intrusion Kill Chain* (HUTCHINS; CLOPPERT; AMIN, 2011) ou o *Attack Life Cycle* (MCWHORTER, 2013). Nesses modelos, os invasores inicialmente procuram implantar um *backdoor* em um ativo de rede vulnerável, estabelecendo, assim, um ponto de apoio dentro da infraestrutura de destino. Então, operando a partir desse ponto de entrada, de forma gradual e discreta, os invasores expandem seu controle para outros ativos da rede até obter privilégios suficientes para alcançar seus objetivos.

Definição do Problema. Os mecanismos de defesa devem atender às necessidades das redes atuais de alta velocidade, cujas taxas de dados também atingem a ordem de dezenas de terabits por segundo—-especialmente em Pontos de Troca de Tráfego (PTTs) e Provedores de Serviços de Internet (PSIs). É desafiador obter mecanismos que consigam defender a essas redes e aos seus clientes ao mesmo tempo em que atendem a requisitos cada vez mais rigorosos de acurácia, latência, taxa de transferência, custo e flexibilidade. Os mecanismos de defesa existentes buscam um equilíbrio satisfatório entre esses objetivos frequentemente conflitantes, geralmente recorrendo a hardware altamente especializado ou delegando funções a software executado em servidores remotos. O uso de hardware especializado, como *middleboxes* baseadas em circuitos integrados de propósito específico (ASICs) de função fixa, promove alta acurácia, baixa latência e alta vazão. No entanto, essa abordagem demanda altos custos de capital e operacionais (FEAMSTER; REXFORD; ZEGURA, 2014), além de potencialmente levar à dependência excessiva do fornecedor e a situações em que a atualização exigiria a substituição completa do parque de equipamentos. Em contraste, as soluções baseadas em software são mais flexíveis, mas exigem interação e coordenação contínuas entre servidores de rede e dispositivos de encaminhamento. Além disso, analisar cada pacote encaminhado em software geraria sobrecustos inaceitáveis em termos de tempo de processador, alocação de memória e tráfego de gerenciamento de rede. Para diminuir esse sobrecusto, costuma-se recorrer a abordagens como amostragem de pacotes (p.ex., sFlow (PHAAL; PANCHEN; MCKEE, 2001)) e agregação de estatísticas baseadas em fluxo (p.ex., NetFlow (CLAISE, 2004) e OpenFlow (The Open Networking Foundation, 2015)). Apesar de terem seus benefícios, essas abordagens ainda deixam a desejar em termos de precisão ou uso de recursos, dependendo da granularidade da análise (MOSHREF; YU; GOVINDAN, 2013). Além disso, a coordenação necessária entre os planos de dados e de controle implica longos ciclos de controle, o que leva a atrasos não desprezíveis na detecção e mitigação. O cenário já intrincado de soluções potenciais torna-se ainda mais complicado na defesa contra APTs, uma tarefa que requer Sistemas de Detecção de Intrusão de Rede (NIDSes). Embora os NIDSes já contem com mais de duas décadas de pesquisa e desenvolvimento, ainda existe um problema fundamental de escalabilidade com pelo menos duas facetas: primeiro, as redes modernas de alta velocidade tornam cada vez mais caro copiar pacotes dos dispositivos de encaminhamento para a memória principal (RAM) de computadores externos; segundo, os NIDSes exigem inspeção de estado e análise de carga útil de pacotes. Assim, realizar análises de tráfego em equipamentos de alto desempenho, a fim de descobrir pistas muito sutis de atividades maliciosas, continua sendo um desafio complexo.

Motivação. Recentes avanços tecnológicos apresentam uma oportunidade sem precedentes para enfrentar os desafios acima. A programabilidade do plano de dados possibilita executar algoritmos inovadores de processamento de pacotes diretamente dentro dos dispositivos de encaminhamento (FEAMSTER; REXFORD; ZEGURA, 2014). A programabilidade do plano de dados permite realizar a inspeção maciça de pacotes diretamente no plano de dados, facilitando assim a obtenção de defesa de rede com baixa latência e alto rendimento. Vários trabalhos aproveitam os Planos de Dados Programáveis (PDPs) para melhorar a escalabilidade das funções de gerenciamento e monitoramento de rede. Essas soluções introduzem blocos de construção essenciais, como algoritmos otimizados para execução em *line rate* (p.ex., Yu, Jose and Miao (2013), Liu et al. (2016), Yang et al. (2018)), linguagens de consulta baseada em processamento de *data streams* (p.ex., Gupta et al. (2016), Narayana et al. (2017)), e operações primitivas para defesa (p.ex., (ZHANG et al., 2020)). Alguns desses trabalhos apresentam estudos de caso interessantes relacionados à segurança. No entanto, a generalidade dessas soluções resulta em construtos de monitoramento que não atingem a funcionalidade exigida para instanciar mecanismos sofisticados de detecção e mitigação relacionados a segurança. Ainda assim, a imensa flexibilidade dos PDPs permite conceber soluções otimizadas para a defesa da rede.

Objetivos. Esta dissertação explora o potencial de planos de dados programáveis como base para novas soluções de defesa de rede. Esse trabalho rumo a um *framework* de segurança baseada em redes programáveis tem duas iterações. Na primeira iteração, com o objetivo de proteger as redes contra ataques DDoS volumétricos e ampliar os limites dos PDPs, propõe-se o EUCLID (ILHA et al., 2021), uma solução para análise de tráfego com baixa latência e granularidade fina para detectar e mitigar ataques DDoS. Esse trabalho, detalhado no Capítulo 3, baseia-se em mecanismo de detecção de anomalias (LAPOLLI; MARQUES; GASPARY, 2019) que utiliza a entropia de Shannon dos endereços IP para caracterizar padrões de tráfego legítimos e gerar alertas sobre condições anormais. O EUCLID introduz, ainda, um mecanismo de defesa que reage a esses avisos, integrando assim a detecção e a mitigação de ataques inteiramente no plano de dados. Até onde se sabe, este trabalho foi o primeiro a delegar esse tipo de mecanismo de detecção e mitigação de anomalias para dispositivos de encaminhamento programáveis. Para atender às restrições estritas de tempo e memória do plano de dados, EUCLID aproxima frequências usando *count sketches* personalizados (CHARIKAR; CHEN; FARACH-COLTON, 2002) e executa operações matemáticas complexas com o auxílio de uma tabela *longest prefix match* (LPM) otimizada para utilizar o mínimo possível de memória. O método proposto identifica pacotes suspeitos e impõe uma política de segurança arbitrária (como descarte, limitação ou desvio) para evitar que o tráfego suspeito prejudique os serviços de rede. Avalia-se a

eficácia do método por meio de experimentos baseados em um protótipo implementado em P4, ao qual se submetem cargas de trabalho realistas. Também compara-se o desempenho do mecanismo com o de uma solução bem estabelecida.

Na segunda iteração, partindo-se da experiência acumulada no trabalho anterior, investiga-se a possibilidade de avançar para uma abordagem geral para detecção de ataques cibernéticos. Busca-se uma solução que (*i*) considere as operações de análise e monitoramento de tráfego exigidas por um NIDS e que façam uso intenso de CPU, (*ii*) delegue essas operações para planos de dados programáveis e (*iii*) permita que o plano de dados notifique o NIDS sobre a ocorrência de eventos relevantes. Para atingir esses objetivos, propõe-se o RNA, um *framework* que empurra o processamento de tráfego para fora dos servidores NIDS e para dentro da própria rede. Utilizando-se a plataforma de monitoramento Zeek (que inclui recursos de NIDS) e a linguagem de programação P4, implementa-se uma prova de conceito da solução proposta, avaliada via estudos de caso representativos.

Contribuições. As principais contribuições deste trabalho são listadas a seguir.

1. Explora-se os limites das primitivas e construções de planos de dados programáveis (PDPs) para projetar um mecanismo *in-switch* contra ataques DDoS volumétricos.

2. Em contraste com abordagens existentes e trabalhos anteriores, projeta-se uma solução que integra diretamente no PDP a *detecção* e a *mitigação* de ataques.

3. Avalia-se minuciosamente as vantagens de desempenho de delegar para PDPs uma solução antiataques DDoS.

4. Apresenta-se um *framework* que delega rotinas de análise de tráfego de NIDSes baseados em CPU para um PDP.

5. Dá-se passos significativos em direção a uma solução geral aprimorada para PDP para detecção de ataques cibernéticos.