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

ÂNGELO CARDOSO LAPOLLI

Offloading Real-time DDoS Attack Detection to 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
December 2019

CIP — CATALOGING-IN-PUBLICATION

Cardoso Lapolli, Ângelo

Offloading Real-time DDoS Attack Detection to Programmable Data Planes / Ângelo Cardoso Lapolli. – Porto Alegre: PPGC da UFRGS, 2019.

65 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Luciano Paschoal Gaspary.

1. Network Security. 2. Programmable Data Planes. 3. DDoS Attacks. I. Paschoal Gaspary, Luciano. II. Título.

“Technology is a useful servant but a dangerous master.”

— CHRISTIAN LOUS LANGE

ACKNOWLEDGEMENTS

I would first like to express my sincere gratitude to my advisor, Luciano Gaspary, who provided me with remarkable guidance since the definition of the research topic, through design and experimentation methodology, to the exposition of our findings.

I am also grateful to Jonatas Marques for his availability and crucial contribution in sharing his research experience and technical expertise to enrich the discussion concerning this thesis.

I acknowledge the support from the professors and staff of UFRGS Institute of Informatics through its Postgraduate Program in Computing (PPGC). Especially, I recognize the collaboration of the Computer Networks research group for inspiring academic excellence through its distinguished members.

I thank the understanding of my life partner, work colleagues, friends, and family on my occasional absence. Above all, I thank my parents for their love, comfort, encouragement, and wise counseling.

ABSTRACT

In recent years, Distributed Denial-of-Service (DDoS) attacks have escalated both in frequency and traffic volume, with outbreaks reaching rates up to the order of terabits per second and compromising the availability of supposedly highly resilient infrastructure (e.g., DNS and cloud-based web hosting). The reality is that existing detection solutions resort to a combination of mechanisms, such as packet sampling and transmission of gathered data to external software, which makes it very difficult (if at all possible) to reach a good compromise for accuracy (higher is better), resource usage footprint, and latency (lower is better). Data plane programmability has emerged as a promising approach to help meeting these requirements as forwarding devices can be configured to execute algorithms and examine traffic at line rate. In this thesis, we explore P4 primitives to design a fine-grained, low-footprint, and low-latency traffic inspection mechanism for real-time DDoS attack detection. Our proposal – the first to be fully in-network – contributes to shed light on the challenges to implement sophisticated security logic on forwarding devices given that, to operate at high throughput, the inspection (and overall processing) of packets is subject to a small time budget (dozens of nanoseconds) and limited memory space (in the order of megabytes). We evaluate the proposed mechanism using packet traces from CAIDA. The results show that it can detect DDoS attacks entirely within the data plane with high accuracy (98.2%) and low latency ($\approx$250 ms) while keeping device resource usage low (dozens of kilobytes in SRAM per 1 Gbps link and a few hundred TCAM entries).

Keywords: Network Security. Programmable Data Planes. DDoS Attacks.

Delegando a Detecção de Ataques Distribuídos de Negação de Serviço a Planos de Dados Programáveis

RESUMO

Nos últimos anos, ataques distribuídos de negação de serviço vêm crescendo tanto em frequência quanto em volume de tráfego com surtos atingindo taxas da ordem de terabits por segundo e compremetendo a disponibilidade de infraestruturas supostamente resilientes (*e.g.*, DNS e hospedagem Web na nuvem). Na prática, as soluções de detecção existentes valem-se de uma combinação de mecanismos, como amostragem de pacotes e transmissão dos dados coletados a um software externo, que dificulta a obtenção de uma boa relaçao entre acurácia (maior é melhor), consumo de recursos e latência (menor é melhor). Planos de dados programáveis emergem como uma abordagem promissora para ajudar a cumprir esses requisitos, visto que dispositivos comutadores de pacotes podem ser configurados para executar algoritmos e examinar o tráfego em velocidade de linha. Neste trabalho, exploramos primitivas em P4 a fim de projetar um mecanismo de inspeção de tráfego com baixa granularidade, baixo consumo de recursos e baixa latência para a detecção de ataques distribuídos de negação de serviço em tempo real. A nossa proposta – a primeira a ser completamente implementada em plano de dados – contribui para lançar luz sobre os desafios da implementação de lógica de segurança sofisticada nesse contexto, dado que, para operar a altas taxas de transferência, a inspeção (e o processamento em geral) de pacotes está sujeita a um orçamento de tempo reduzido (dezenas de nanossegundos) e um espaço de memória limitado (da ordem de dezenas de megabytes). Nós avaliamos o mecanismo proposto usando capturas de pacotes da CAIDA. Os resultados mostram a detecção de ataques exclusivamente a partir do plano de dados com alta acurácia (98,2%) e baixa latência ($\approx$250 ms) mantendo o consumo de recursos reduzido (dezenas de kilobytes de SRAM por link de 1 Gbps e poucas centenas de entradas TCAM).

Palavras-chave: Segurança de Redes. Planos de Dados Programáveis. Ataques Distribuídos de Negação de Serviço.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
AS	Autonomous System
CAIDA	Center for Applied Internet Data Analysis
CPU	Central Processing Unit
DDoS	Distributed Denial-of-Service
DNS	Domain Name System
EWMA	Exponentially Weighted Moving Average
EWMMD	Exponentially Weighted Moving Mean Difference
FPGA	Field-Programmable Gate Array
FPR	False-Positive Rate
IEEE	Institute of Electrical and Electronics Engineers
IFIP	International Federation for Information Processing
IP	Internet Protocol
ISP	Internet Service Provider
JSON	JavaScript Object Notation
LPM	Longest-Prefix Match
NIDS	Network Intrusion Detection System
NOS	Network Operating System
SDN	Software-Defined Networking
SRAM	Static Random Access Memory
TCAM	Ternary Content-Addressable Memory
TPR	True-Positive Rate

CONTENTS

1 INTRODUCTION

Despite consistent efforts towards effective detection and mitigation mechanisms, Distributed Denial-of-Service (DDoS) attacks remain among the top networking security concerns as outbreaks escalate both in frequency and traffic volume (ALCOY et al., 2018). Reports of recent attacks targeting Dyn (HILTON, 2016) and GitHub (KOTTLER, 2018) reveal peak rates of up to the order of terabits per second compromising the availability of (supposedly) scalable and resilient infrastructure. Given this trend, we should expect events like these to get even worse in the future (ALCOY et al., 2018).

Existing defensive mechanisms typically rely on standardized monitoring primitives such as packet sampling - e.g., sFlow (PHAAL; PANCHEN; MCKEE, 2001)) - and flow-based accounting - e.g., NetFlow (CLAISE, 2004), OpenFlow (The Open Networking Foundation, 2015). However, these primitives present significant overhead regarding packet processing and resource utilization to provide fine-grained traffic visibility. While packet sampling conveys information from a reduced set of packets to keep a reasonable load in terms of CPU processing and network management traffic (PHAAL, 2009), flow-based accounting is limited to coarsely aggregated volume metrics due to elevated memory footprint (MOSHREF; YU; GOVINDAN, 2013). Resulting from these limitations, we advocate that the existing tooling for monitoring falls short in either *accuracy* or *resource usage* when it comes to DDoS attack detection. Furthermore, these approaches are subject to a long control loop, resulting in non-negligible detection *latency*.

As a promising alternative to these issues, the emerging concept of *data plane programmability* offers flexibility to readily implement novel in-switch packet processing algorithms (BOSSHART et al., 2014). These algorithms assume a packet stream as input and are modeled as a pipeline of elementary primitives, memory accesses, and table lookups. Human operators are thus able to define monitoring functions and delegate them to forwarding devices across the whole network. This still relatively unexplored concept has the potential of enabling all packets of a stream to be examined with reduced processing/communication overheads and achieving low-latency anomaly detection. Yet, to operate at line rate on high-speed links, this processing is constrained to a small time budget (dozens of nanoseconds) and a limited memory space (e.g., $\approx 50\,\mathrm{MB}$ SRAM and $\approx 5\,\mathrm{MB}$ TCAM) (BOSSHART et al., 2013).

Meeting the aforementioned constraints is a difficult challenge that limits the scope of the existing data plane monitoring solutions. For example, Sonata (GUPTA et al., 2016)

and Marple (NARAYANA et al., 2017) take advantage of data plane programmability to configure adaptive filters which determine packet streams to be forwarded to and examined by the control plane. This approach still implies a trade-off between communication overhead and detection latency, which is driven by the rate of data collection. StateSec (REBECCHI et al., 2019), in turn, is a DDoS attack detection mechanism fully implemented on Software-Defined Networking (SDN)/OpenFlow-based forwarding devices. It extends the *match/action* table structure and semantics to keep track of harmful packet exchange patterns. Nevertheless, it does not take into account the requirement of scaling to run at line rate on high-throughput hardware packet processors.

In this thesis, we give a consistent step towards *in-network*, programmable network security. We explore a promising data plane programming technology, namely P4 (BOSSHART et al., 2014), to design a mechanism to perform *low-latency, fine-grained traffic inspection* for *real-time DDoS attack detection* (LAPOLLI, 2019b). In contrast to existing solutions in the context of SDN, the proposed mechanism is fully implementable on forwarding devices. It comprises a processing pipeline to estimate the entropies of both source and destination IP addresses of incoming packets. The entropy measurements are used to both characterize the traffic and calculate anomaly detection thresholds (as functions of a parameterizable sensitivity coefficient). In order to meet the strict time and memory constraints of forwarding devices, we approximate the frequencies of distinct IP addresses through specially tailored count sketches (CHARIKAR; CHEN; FARACH-COLTON, 2002). Further, compute-intensive arithmetic functions are solved with the aid of a memory-optimized longest-prefix match (LPM) lookup table. Based on realistic datasets of legitimate traffic and DDoS attacks, we assess the entropy estimation error and evaluate the detection performance in terms of accuracy and resource consumption. We also compare the effectiveness and efficiency (i.e., latency) of the proposed mechanism with those of the "de facto" approaches.

The primary research contributions of this thesis are threefold. First, we stress data plane programmability primitives (in this work, of P4) – known for their limited functionality – to design a reasonably sophisticated in-network DDoS attack detection mechanism. Second, we demonstrate, through an extensive evaluation, the performance benefits that security mechanisms can reap from a data plane-based design. Third, we discuss challenges and present insights associated with the development of security mechanisms in the data plane that can be valuable for new research initiatives in the area.

The remainder of this work is structured as follows. We first provide a background

on SDN, P4 and anomaly-based DDoS attack detection, and discuss related work (Chapter 2). Then, we specify the attack scenario and threat model, introduce the detection strategy and describe the design of the proposed mechanism (Chapter 3). Further, we detail our implementation as a proof of concept in terms of data-plane operation feasibility (Chapter 4). Next, we present the evaluation methodology and results, discussing relevant findings (Chapter 5). Finally, we point out the major lessons learned in the design process and outline plans for future work (Chapter 6).

2 BACKGROUND AND RELATED WORK

In this chapter, we portray the research context surrounding our DDoS attack detection mechanism and discuss related work. We first introduce the motivations and the fundamental concepts of SDN/OpenFlow (Section 2.1). Next, we overview how the P4 language abstracts data plane devices and enables programmable protocol-independent packet processing (Section 2.2). Then, we review anomaly-based strategies for DDoS attack detection regarding traffic feature extraction and inspection approaches (Section 2.3). Finally, we discuss prominent related work (Secion 2.4).

2.1 Software-Defined Networking (SDN)

Network operators traditionally implement control policies through the individual configuration of packet forwarding devices offering a preset collection of standardized protocols and vendor-specific features. As networks grow with numerous distinct devices, decomposing high-level network-wide intents into low-level distributed settings becomes increasingly complex. Further, the typically lengthy process of protocol standardization hampers timely innovation, whereas proprietary solutions preclude vendor interoperability. Since operators cannot directly explore internal device structures (e.g., forwarding tables, flow-state data storage) to implement their algorithms, this network management model is highly dependent on vendor support. This scenario has lead to an intrinsic difficulty in changing networking control designs, characterizing what is known as “network ossification" (FEAMSTER; REXFORD; ZEGURA, 2014).

Prominent research efforts towards overcoming such innovation barriers date back to the 1990s. Back then, active networking advocated for the exposition of forwarding device resources (e.g., processing, storage) in the form of a network Application Programming Interface (API), through which network operators or end-users could conceive novel approaches for packet processing (TENNENHOUSE; WETHERALL, 2007). Despite considerable research effort in this direction, there have not been enough compelling applications to motivate the use of this technology. More recently, the hassle of managing increasing traffic volume and network sizes pushed for the embracement of the Software-Defined Networking (SDN) paradigm (FEAMSTER; REXFORD; ZEGURA, 2014).

SDN proposes a clear separation of concerns regarding network control and operation, which is given by three distinct planes as described below (KREUTZ et al., 2015).

- The *data plane* refers to the interconnection infrastructure composed by hardware- or software-based devices endowed with elementary forwarding operations. These operations are performed over incoming packets according to remotely preset rules through what is called the southbound interface.

- The *control plane* is where a logically centralized software, called Network Operating System (NOS) or controller, handles the installation of forwarding rules and monitors the status of data plane devices and links to consolidate a global network view. The main purpose is to abstract the complexity of managing low-level details regarding the distributed forwarding devices providing operators with a simpler API, i.e, the northbound interface.

- The *management plane* consists in the set of applications exploring the northbound interface to control the network operation. The independence of forwarding device implementations at this level encourages complementary innovation efforts: while device vendors can focus on designing efficient hardware for packet processing, network operators are free to promptly test and deploy custom control algorithms from a higher abstraction level.

The OpenFlow protocol, initially proposed to enable innovation in campus networks (MCKEOWN et al., 2008), has driven the decoupling of the control plane from the data plane. OpenFlow specifies a southbound interface for the controller to establish secure communication channels to manage the forwarding device rules and receive status updates (The Open Networking Foundation, 2015). These rules designate actions (e.g., forward, flood, drop) to packet flows determined from patterns of header field values (e.g., Ethernet/IP addresses, TCP/UDP ports). The status includes information from link layer discovery and counters of packet and bytes associated to flow table entries (supporting a global network view at the control plane).

The possibility of flexibly programming and disseminating new packet forwarding rules have been motivating the emergence of several open-source controllers – e.g., OpenDaylight (The OpenDaylight Foundation, 2019), ONOS (The Open Networking Foundation, 2018) – and applications in a wide range of topics, including traffic engineering, network virtualization, and network security (XIA et al., 2015). Within the industry, in general, OpenFlow implementation has only required software updates for its backward compatibility with the existing switching hardware. On the other hand, this characteristic has also limited OpenFlow to the support of standard protocols and predetermined packet

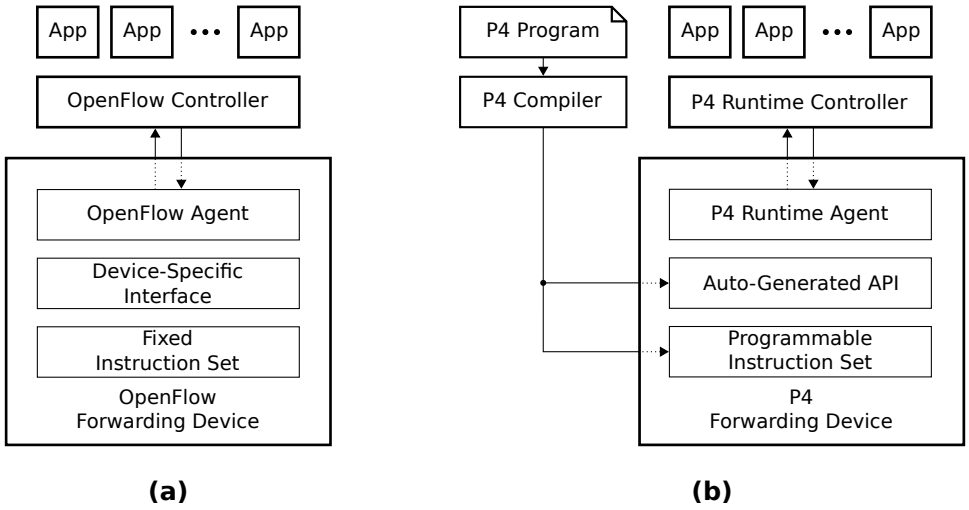
processing functions. The inability to effectively customize the data plane behavior hampers new designs for forwarding protocols and traffic measurement. These limitations have led to the emergence of novel technologies to enable data plane programmability, notably, P4 (BOSSHART et al., 2014).

2.2 P4: Programming Protocol-independent Packet Processors

P4 is a protocol-independent domain-specific language for describing packet processing within data plane devices. Figure 2.1 depicts the difference between the OpenFlow and the P4 control scheme. While OpenFlow abstracts device-specific interfaces built upon a fixed instruction set implementation, P4 is a mean for operators to program data plane instructions which become accessible through an auto-generated API. Thus, P4 enables a full top-down network management approach where it is possible to define packet processing primitives and explore them to develop control applications.

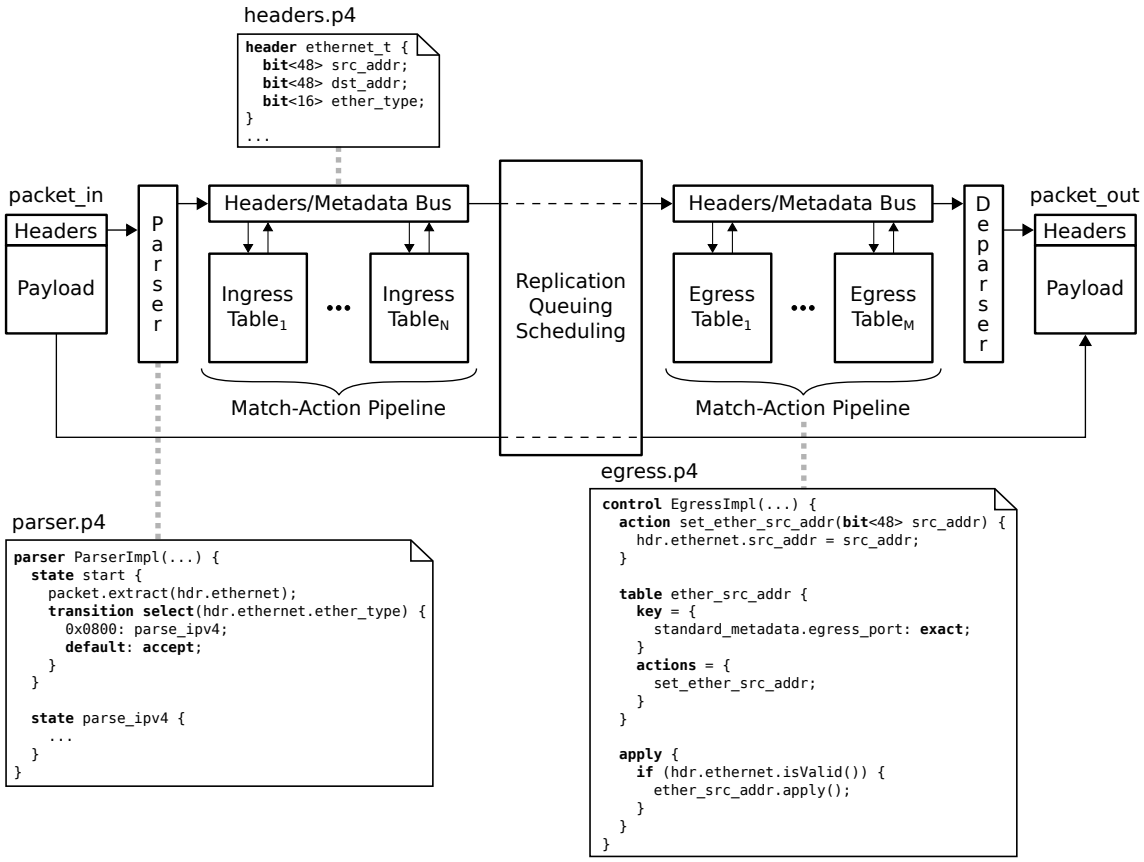
P4 includes sub-languages for expressing header parsers and match-action processing pipelines. Device vendors are expected to specify the architectural arrangement of such processing units and expose target-specific functionality (The P4 Language Consortium, 2017). Figure 2.2 illustrates a generic packet processing architecture and excerpts of P4 code. The headers of the incoming packet `packet_in` are input to the parser which decodes the bitstream into user-defined data types (see `headers.p4`). This data, along with ingress metadata (port, timestamp), is available to the match-action pipelines.

Figure 2.1: Comparison between (a) the OpenFlow and (b) the P4 control scheme.



Source: the authors (2019).

Figure 2.2: Generic Packet Processing Architecture and P4 Sample Code



Source: the authors (2019).

These pipelines consist in a user-defined table chaining which may alter header fields and propagate custom metadata (see `egress.p4`). The ingress pipeline is responsible for specifying the packet egress ports. An intermediate non-programmable block replicates packets in case of multicast or broadcast, and handles both queuing and scheduling. Next, the egress pipeline is executed independently on every packet replica. Finally, the deparser serializes updated headers for the output packet `packet_out`.

User-defined headers are ordered sets of P4 built-in base type fields including single bits, fixed- or variable-width bit-strings, and fixed-width signed integers (see `headers.p4`). The parser is modeled as a state machine which sequentially reads the packet bitstream to extract headers (see `parser.p4`). This processing unit is the only that may contain loops, as long as some header is extracted at each cycle. This restriction keeps the computational complexity linear to the headers size inhibiting packet processing pipeline stalls (The P4 Language Consortium, 2017).

The match-action pipelines include tables with arbitrary keys for matching packet header fields and metadata (see `egress.p4`). Actions may alter this data using primi-

tives such as elementary arithmetic (addition, subtraction, and multiplication) and bitwise operators (*not*, *and*, *or*, *xor*). Special-purpose metadata fields define the desired forwarding behavior. The table chaining control flow results from a sequential algorithm provided with conditional operators.

In addition to the P4 language core, device vendors may expose additional functionality to data plane programmers through *extern* objects and functions. Examples of such extensions include counters and meters associated with table entries, and registers to persist data across packet processing. These specific features are key elements to enable traffic measurements within the data plane. Registers, in particular, can help to decouple traffic statistics from the forwarding rules and consequently improve memory management.

Data plane programmability as proposed by P4 does not only enable quick innovation – since operators are able to readily implement custom forwarding logic – but also supports the collection of network information at the packet level. This possibility creates new design opportunities for network diagnosis (KIM et al., 2016) and measurements (GUPTA et al., 2016; NARAYANA et al., 2017). Upon the same technological foundation, we propose to perform fully in-network real-time anomaly-based DDoS attack detection.

2.3 Anomaly-Based DDoS Attack Detection

Anomaly-based Network Intrusion Detection Systems (NIDSs) model the legitimate traffic behavior to identify outliers which possibly represent a malicious activity. They are opposed to signature-based NIDSs that inspect the traffic for priorly known attack patterns. The anomaly-based approach, though typically more prone to issue false alarms, may detect unknown attacking strategies (GARCÍA-TEODORO et al., 2009). The inspection process to that end involves extracting meaningful traffic features and distinguishing deviations representing deceitful intents. In this section, we present a general view regarding these challenges and summarize relevant work over a comparison table (Table 2.1).

Volume metrics are basic building blocks for traffic feature extraction. They comprise any sort of counting (e.g., packets, bits, bytes, header field values) in terms of an aggregation criterion (e.g., flows, interfaces, destination, time interval, etc.). Given that exact measurements over high traffic volumes implies substantial memory utilization, se-

Table 2.1: Comparison of Anomaly-Based DDoS Attack Detection Mechanisms

Reference	Traffic Features	Detection Method
MULTOPS (GIL; POLETTO, 2001)	Ratio of the packet rates to/from aggregations of IP addresses.	Static Threshold
(LAKHINA; CROVELLA; DIOT, 2005)	Entropy of IP addresses and transport-layer ports.	k-means Clustering
(ÖKE; LOUKAS, 2007)	Bitrate (entropy, Hurst parameter) and round-trip time.	Random Neural Network
(MA; CHEN, 2014)	Entropy of IP addresses	Static Threshold to the Lyapunov Exponent
(XU; LIU, 2016)	Flow volume and flow rate asymmetry.	Self-Organizing Map
StateSec (REBECCHI et al., 2019)	Entropy of IP addresses and transport-layer ports	Dynamic threshold as a function of the mean and the standard deviation.
(HOQUE; KASHYAP; BHATTACHARYYA, 2017)	Correlation of entropy and variation index of source IP addresses, and packet rate.	Dynamic threshold as a function of the correlation mean and range.

Source: the authors (2019).

curity mechanisms often resort to coarser aggregation criteria (GIL; POLETTO, 2001; XU; LIU, 2016) or probabilistic approximations with reduced footprint (e.g, sketches) (YU; JOSE; MIAO, 2013; LIU et al., 2016; YANG et al., 2018). Without further processing, these measurements may reveal aggressive volumetric DDoS attack, but they are hardly distinguishable from legitimate events such as flash crowds.

The statistical analysis of the volume metrics in terms of ratios, correlation, and entropy contributes to discriminate the malicious behavior and adds sensitivity to semantic attacks. Ratios, as the relation of packets with different directions in a flow, reveal asymmetric resource allocation between communicating hosts (GIL; POLETTO, 2001; XU; LIU, 2016). Correlation measures indicate the degree of similarity between a normal traffic profile and the one being inspected (HOQUE; KASHYAP; BHATTACHARYYA, 2017). Entropy, on the other hand, characterizes the uncertainty of random variables helping to detect changes in the distribution patterns of packet header fields (LAKHINA; CROVELLA; DIOT, 2005; ÖKE; LOUKAS, 2007; MA; CHEN, 2014; REBECCHI et al., 2019; HOQUE; KASHYAP; BHATTACHARYYA, 2017).

The analysis of such traffic features for DDoS attack detection is typically modeled as a decision or a classification problem. A common approach in this sense is to define thresholds from the combination of averages and indices of dispersion (REBECCHI et al., 2019). In this case, updating the traffic characterization reference is essential to take into consideration the dynamics of legitimate traffic, but it can make the detection mechanism vulnerable to manipulation by the attacker. Another relevant detection method is the use of machine learning algorithms (e.g., k-means clustering (LAKHINA; CROVELLA; DIOT, 2005), Random Neural Network (ÖKE; LOUKAS, 2007), Self-Organizing Map (XU; LIU, 2016)). Regardless of the effectiveness of techniques of this sort, they usually imply increased computational effort representing a challenge for the inspection at high traffic rates.

2.4 Related Work

Despite the consistent efforts on selecting and processing traffic features to detect DDoS attacks, real-time security solutions are limited by the monitoring functionality currently implemented on most forwarding devices, in which accuracy necessarily translates into high overheads (MOSHREF; YU; GOVINDAN, 2013). It is in this context that programmable data plane-based approaches emerge as promising alternatives, being subject of consistent research work concerning, for example, scalable in-network packet processing models and fine-grained traffic measurement capability. Next, we review some of the most prominent investigations in the area.

Based on SDN/OpenFlow, Xu and Liu (XU; LIU, 2016) propose methods to detect DDoS attacks and identify their sources and victims. On top of the control plane, a software application classifies flows regarding their volume and rate asymmetry through an unsupervised learning algorithm. This traffic data is collected from the counters associated to flow table entries (in forwarding devices). Since the number of entries is limited, their aggregation granularity is adaptively changed to enable zooming into abnormal traffic patterns. This process is iterative and highly dependent on the management plane, introducing non-negligible latency (in the order of several seconds) to the detection of an ongoing attack.

To offload monitoring functions to the data plane, StateSec (REBECCHI et al., 2019) is a DDoS attack detection mechanism based on in-switch processing capabilities. StateSec is based on an OpenFlow extension in which flow tables can be used to specify

finite-state machines for packet processing (BIANCHI et al., 2014). The detection is performed through entropy analysis of both source/destination IP addresses and transport-layer ports. These metrics along with their mean and standard deviation are supposed to be calculated within the data plane. However, the mechanism requires a flow table entry for every distinct observed IP/port 4-tuple value, which implies unbounded table utilization. Furthermore, it does not elaborate on how to measure entropy while meeting the time budget to operate on high-throughput forwarding devices.

Advancing from more traditional SDN/OpenFlow-based measurement approaches to ones in which accounting is fully delegated to the data plane, OpenSketch (YU; JOSE; MIAO, 2013), UnivMon (LIU et al., 2016), and Elastic sketch (YANG et al., 2018) provide flexible hash table-based designs that enable network operators to implement a wide range of measurement tasks in the data plane. The forwarding devices are responsible for maintaining sets of hash tables (named *sketches*) with summarized up-to-date traffic counters. The control plane periodically collects this data for further processing. As a result, these solutions achieve high generality and accuracy in traffic measurement. However, they are subject to a trade-off between the data polling rate (which is directly related to anomaly detection latency) and network overhead due to additional management traffic.

In an attempt to offload additional monitoring logic to the data plane, Sonata (GUPTA et al., 2016) allows operators to define packet stream filtering queries. These queries are executed on programmable forwarding devices so that only the traffic of interest is sent to external stream processors. Packet headers are abstracted as tuples of field values, which – in addition to be filtered – can be sampled in the data plane. Based on these abstractions, network operators can optimize packet sampling to detect priorly known anomalous traffic patterns, but potentially missing novel attack strategies. Following a similar concept, Marple (NARAYANA et al., 2017) is a language for expressing monitoring queries that are compiled to target programmable forwarding devices. It enables in-network execution of functions over aggregation of packets backed by a new key-value store primitive. Despite providing forwading devices with the ability to measure traffic features, the inspection of such metrics is still delegated to external servers.These solutions significantly contribute to deploying generic traffic measurement tasks dynamically. Differently, our work assumes a reduced scope (DDoS attack anomaly detection), in which we seek to offload (as well) this inspection to forwarding devices.

Hoque et al. (HOQUE; KASHYAP; BHATTACHARYYA, 2017) propose an hybrid real-time DDoS attack detection mechanism implemented within a FGPA (Field-

Programmable Gate Array) in cooperation with software modules. The mechanism consists in (*i*) a software pre-processor, which performs feature extraction over 1 s-long observation windows; (*ii*) an FPGA, responsible for the detection decision based on the computation of a novel correlation metric; and (*iii*) a software named security manager, which analyzes the traffic historical data to generate a normal profile and dynamically define a detection threshold. The design goal is to trigger DDoS attack alarms from the victim's end, not taking into consideration the possibility of network resource saturation prior to this point. Beyond that, the authors do not demonstrate the feasibility of the pre-processor to operate in real-time, despite it being in the mechanism's critical path.

Table 2.2 summarizes the above discussion in terms of the feature extraction and the inspection characteristics. OpenFlow presents an intrinsic limitation in memory management when it comes to fine-grained traffic measurement as the accounting criteria are coupled to the packet forwarding rules. While StateSec does not approach this issue, Xu and Liu compensate that with at the cost of a long control loop to dynamically change the aggregation criteria. OpenSketch, UnivMon, Elastic sketch, Sonata, and Marple are com-

Table 2.2: Related Work Summary

	Feature Extraction				**Inspection**	
Reference	**Technology**	**Granularity**	**Memory**	**Throughput**	**Location**	**Ctrl. Loop**
(XU; LIU, 2016)	OpenFlow	Dynamic	Low	High	Mgmt. Plane	Long
StateSec (REBECCHI et al., 2019)	OpenFlow + Extension	Fine	High	NA	Data Plane	Short
OpenSketch (YU; JOSE; MIAO, 2013)	Own Proposal	Fine	Low	High	Mgmt. Plane	Medium
UnivMon (LIU et al., 2016)	P4	Fine	Low	High	Mgmt. Plane	Medium
Elastic sketch (YANG et al., 2018)	P4 et al.	Fine	Low	High	Mgmt. Plane	Medium
Sonata (GUPTA et al., 2016)	Software Switch	Coarse	Low	Medium	Mgmt. Plane	Medium
Marple (NARAYANA et al., 2017)	P4	Fine	Low	High	Mgmt. Plane	Medium
(HOQUE; BHATTACHARYYA; KALITA, 2015)	Software Module	Fine (if feasible)	NA	NA	Out-of-Band FPGA	Medium
This Work	*P4*	*Fine*	*Low*	*High*	*Data Plane*	*Short*

Source: the authors (2019).

pelling solutions which explore the data plane to enable accurate and low-footprint feature extraction, but they do not include low-latency in-network inspection of such data. In contrast, Hoque et al. explore programmable hardware to perform low-latency inspection, but they count on software modules for feature extraction which (although not evaluated) likely represent a performance bottleneck.

The area of in-network security management is flourishing, with the potential to allow operators to devise novel attack detection mechanisms within much shorter design and deployment cycles. The aforementioned proposals represent consistent steps towards devising mechanisms to be executed in the data plane, but (*i*) resort to considerable communication with external controllers (delaying the detection of attacks and leading to a high network utilization) and (*ii*) use coarse-grained measurements to cope with the massive amount of data traversing high-speed links (degrading accuracy). Our proposed mechanism goes a consistent step further to enable DDoS attack detection entirely within the data plane. Our design explores data plane programmability functionality to its limit and achieves *accuracy*, *low intrusiveness*, and *timeliness*, as we describe next.

3 IN-NETWORK DDOS ATTACK DETECTION DESIGN

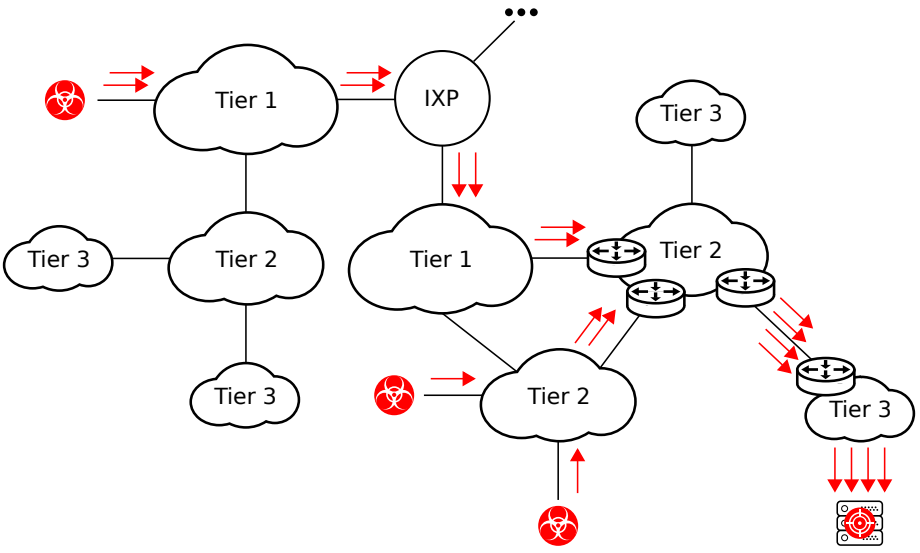
In this chapter, we introduce the proposed in-network DDoS attack detection mechanism. We first describe both the attack scenario and the threat model (Section 3.1). Next, we overview the foundations of the anomaly detection strategy (Section 3.2). Then, we detail the packet processing pipeline devised to run at line rate on programmable forwarding devices (Section 3.3).

3.1 Attack Scenario and Threat Model

The term distributed denial-of-service comprises a multitude of attack strategies to degrade or disrupt external facing services. In this work, we assume an attacker capable of coordinating globally distributed hosts to send illegitimate service requests to a single victim. These requests may either saturate the victim's network with high traffic volume or exploit a specific protocol semantic vulnerability to consume the computing resources of the target server. The attacker uses spoofed IP addresses to hinder the characterization and detection of the attack packets. Further, we assume that forwarding devices are not compromised.

Figure 3.1 illustrates this attack scenario within an hypothetical Autonomous System (AS) interconnection architecture. The spread of attacking hosts among independent

Figure 3.1: DDoS Attack Scenario: the red arrows indicate the malicious traffic flow, the explicit forwarding devices are the candidates to host the proposed detection mechanism.



Source: the authors (2019).

administrative domains imposes challenges to source-based detection mechanisms, because it is hard to coordinate security efforts across administrative borders. At the other extremity of the attack, i.e., the victim's infrastructure, the malicious traffic is aggregated and prominent for detection, but it may have already saturated both in-path and local resources.

Thus, our proposed mechanism is expected to be deployed at the ASs closest to the victim as they benefit from a privileged traffic view and high-throughput links to timely uncover and deter even voluminous outbreaks without exhausting their resources. Our mechanism should run on nodes peering with other ASs to detect attacks and enable mitigation before reaching lower-capacity links. At this location, typical software-based IDSs will hardly comply with the high traffic throughput. Dedicated appliances may suit this scenario, but they represent significant capital expenditure for a comprehensive network defense. P4 allows us to flexibly deploy and customize our mechanism using programmable hardware to keep up with high packet rates.

3.2 Detection Strategy Foundations

Given that the strict time and memory budgets for high-rate in-network packet processing translate to limited programming primitives, it is paramount to resort to a simple, yet powerful detection strategy. We assume DDoS attacks characterized by a large number of hosts (or spoofed sources) converging traffic to one or few victims (HOQUE; BHATTACHARYYA; KALITA, 2015). So, the source and destination IP addresses distributions tend to deviate from the legitimate pattern in the presence of malicious activity. On top of this observation, we design our mechanism based on the calculation of the Shannon entropy (SHANNON, 1948), which is recognized as a reliable method to identify such deviations accurately (LAKHINA; CROVELLA; DIOT, 2005; BHUYAN; BHATTACHARYYA; KALITA, 2015).

Considering X the set of IP addresses within a total of m packets, and $f_0, f_1, ..., f_N$ the frequencies of each distinct address, the entropy of X is given by:

$$H(X) = \log_2(m) - \frac{1}{m}\sum_{x=0}^{N} f_x \log_2(f_x), \tag{3.1}$$

where the summation $S = \sum_{x=0}^{N} f_x \log_2(f_x)$ is the entropy norm. Note that the entropy

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

In the course of a DDoS attack, we expect the entropy of source IP addresses to increase as the malicious packets introduce new values to the distribution. Conversely, we expect the entropy of destination IP addresses to decrease with the victim becoming more frequent as a destination. This effect is only observable when the number of packets m encompasses an adequate, robust representation of the current distributions. One must keep in mind that increasing m comes at the cost of higher attack detection latency, as more packets must be received for each measurement. On the other hand, when calculating the entropy over few packets, malicious traffic-related changes to the distributions may be indistinguishable from short-term fluctuations of legitimate traffic.

To address the mentioned trade-off, we propose setting dynamic thresholds to the entropies of the source and the destination IP addresses considering a preset value of m. We flag the observations in which any of these m packets are malicious, aiming to distinguish them in future work. In the following section, we present the packet processing pipeline devised to implement this approach.

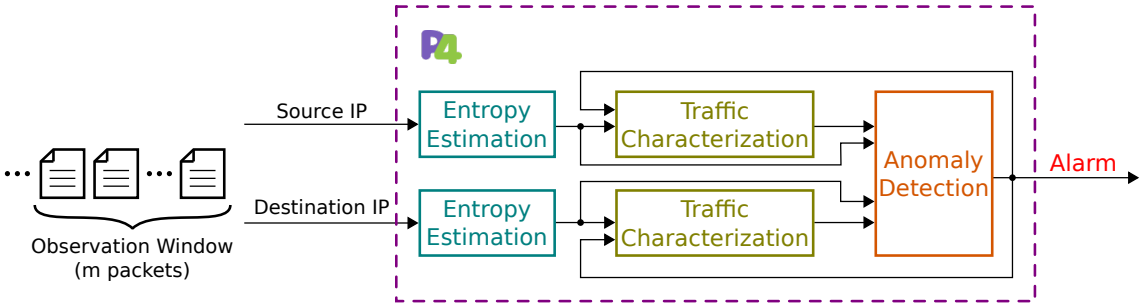
3.3 Packet Processing Pipeline

We build the detection mechanism on top of the P4 behavioral model reference implementation (BMv2) (The P4 Language Consortium, 2019a), which has a constrained set of processing primitives reflecting the limitations of the current programmable hardware devices. In this section, we describe how we overcome these restrictions to perform real-time DDoS attack detection.

Figure 3.2 depicts the top-level scheme of our proposed mechanism. The entropies of IP addresses are estimated for consecutive partitions of the incoming packet stream, named *observation windows* (Subsection 3.3.1). At the end of each observation window, the *traffic characterization* units read the calculated entropy values to generate a legitimate traffic model (Subsection 3.3.2). In turn, the *anomaly detection* unit calculates *detection thresholds* as functions of this model issuing an *attack alarm* when they are exceeded by the last entropy estimates (Subsection 3.3.3).

Our mechanism only issues alarms at the end of observation windows (i.e., every

Figure 3.2: DDoS Attack Detection Top-Level Scheme



Source: the authors (2019).

m packets) as it is when it obtains the entropy estimates. Instead, the use of a sliding window could grant lower detection latency as we would be able to verify attacks at every packet. However, this approach would require significant memory space for storing the last m IP address pairs. In the upcoming subsections, we describe how we achieve reduced memory footprint using disjoint windows. Note that, as per our threat model, the attacker cannot predict the order in which the malicious packets will arrive at the monitoring device related to the overall traffic. Thus, he cannot intentionally distribute the attack packets among windows to bypass the detection.

3.3.1 Entropy Estimation

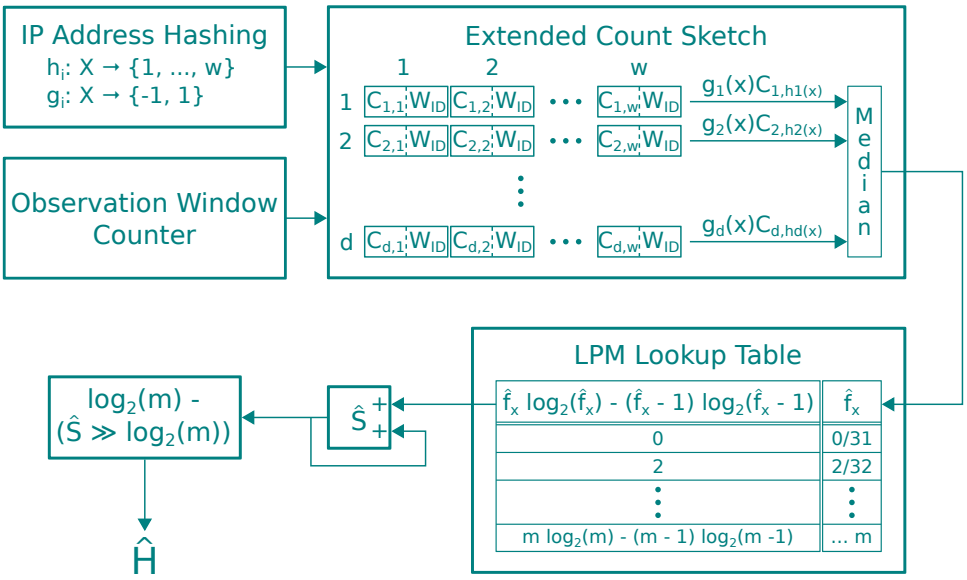
As the P4 behavioral model does not support the binary logarithm function, we assume a fixed value (yet parameterizable) for the observation window size m so that the first term in Equation 3.1 becomes a constant. Consequently, the real-time entropy estimation processing is reduced to the calculation of the second term, which is a function of the frequencies of each distinct observed address.

Next, we detail the approximation of these frequencies from the packet stream while meeting processing constraints. Then, we elaborate on the computation of the entropy norm without resorting to an offline processing stage. Finally, we show how to obtain the entropy estimate. Figure 3.3 illustrates the processing pipeline in its entirety.

Frequency Approximation

We base the approximation of address frequencies on a *count sketch* data structure (CHARIKAR; CHEN; FARACH-COLTON, 2002), which uses sub-linear space to represent a frequency table of events in a data stream. It requires the computation of

Figure 3.3: Entropy Estimation Pipeline



Source: the authors (2019).

hash functions and a two-dimensional array of counters to obtain unbiased probabilistic frequency approximations.

Let X be the set of all possible IP address values and C be a matrix of counters with depth d and width w (i.e., $C \in \mathbb{Z}^{d \times w}$), where $C_{i,j}$ indicates the counter at row i and column j (see Figure 3.3). We define two sets of independent hash functions $\{h_1, ..., h_d\}$ and $\{g_1, ..., g_d\}$, where each pair (h_i, g_i) is associated with a sketch row $i \in \{1, ..., d\}$. All hash functions have as input an IP address $x \in X$. Hash function h_i maps IP addresses to columns in row i (i.e., $h_i : X \mapsto \{1, ..., w\}$). Hash function g_i decides, for each IP address, if the counter $C_{i,h_i(x)}$ should be incremented or decremented (i.e., $g_i : X \mapsto \{-1, 1\}$). The count sketch algorithm defines two operations:

$$
\begin{aligned}
&Update(C, x)\text{:} \\
&\quad \textit{for } i = 1, ..., d : \\
&\quad\quad C_{i,h_i(x)} \leftarrow C_{i,h_i(x)} + g_i(x) \\
&Estimate(C, x)\text{:} \\
&\quad \textit{return } median(g_1(x)C_{1,h_1(x)}, ..., g_d(x)C_{d,h_d(x)})
\end{aligned}
$$

$Update(C, x)$ updates all depth levels of the sketch C to count the occurrence of x. $Estimate(C, x)$ returns an estimate of the frequency count of x, which we denote as $\hat{f}_x$ (see Figure 3.3). The mechanism uses the set of hash functions g_i to deal with the event of h_i colliding for multiple distinct IP addresses. In that event, it is expected that some of the addresses will increase the counter value and others will decrease it, making the

counter assume a clearly inconsistent value. When compared with the other counters of the same address stored on all depth levels, counters with collisions become outliers. The sketch avoids getting tainted by such outliers by using the median (instead of the mean, which is very sensitive to outliers) of the values stored in all depth levels as the frequency estimate.

In P4, this data structure can be implemented using registers, which persist general purpose data across packets. IP address hashing is possible through the definition of custom hash functions. We use functions of the type $(a_i x + b_i) \bmod p$, where a_i and b_i are co-prime coefficients, and p is a prime number. These functions have been successfully used before in programmable data planes (SIVARAMAN et al., 2017).

Obtaining independent consecutive entropy estimates would require to reset all sketch counters at the transition of observation windows. To avoid such a bursty processing overhead, we associate an additional register to each sketch counter to store the identifier of the observation window in which it was last updated (W_{ID}). We employ an observation window counter to generate these identifiers. Hence, whenever a counter is read, its value is only taken into account if the associated register holds the current window identifier; otherwise, it is presumed zero and updated accordingly.

Finally, we take the median value comparing the results of each sketch row. Note that the sketch depth is equal to the number of inputs to the median operator. Thus, such parameter is intrinsically related to the processing complexity of this step. In Chapter 5, we show that the use of few sketch rows does not impact the entropy estimation error significantly if we compensate with a greater sketch width.

Entropy Norm Estimation

Right after an IP address is read, and its current frequency on the observation window is retrieved, we compute its respective term in the summation composing S. We use this result to update the entropy norm estimate $\hat{S}$ (stored in a register). As IP addresses are expected to appear numerous times in a single observation window, we update $\hat{S}$ by incrementing the difference between the newly computed term and its previous value (if $\hat{f}_x > 1$), as follows:

$$\hat{S} \leftarrow \hat{S} + \underbrace{\hat{f}_x \log_2(\hat{f}_x)}_{\text{newly computed term}} - \underbrace{(\hat{f}_x - 1)\log_2(\hat{f}_x - 1)}_{\text{previous term value}}. \tag{3.2}$$

Since the P4 behavioral model does not support floating-point numbers, we rep-

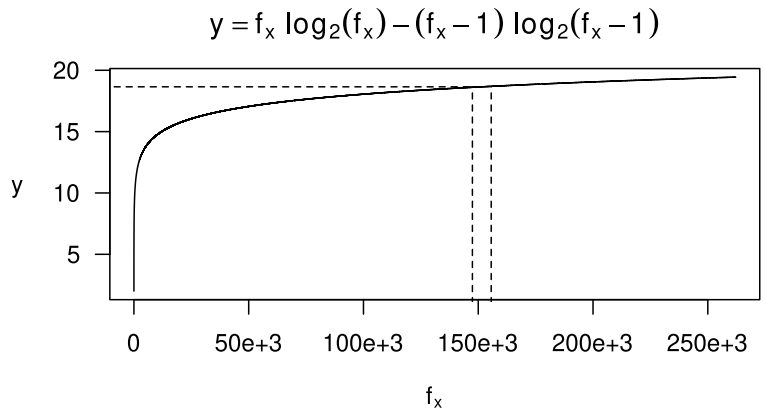
resent $\hat{S}$ in a fixed-point format to allow fractional precision. The required arithmetic operations can be derived from integers.

As an important building block for implementing Equation 3.2 in the data plane, we must compute the binary logarithm, which is not as straightforward. To overcome this challenge, we build an *LPM lookup table* with pre-calculated values for: $\hat{f}_x \log_2(\hat{f}_x) - (\hat{f}_x - 1)\log_2(\hat{f}_x - 1)$. Unlike a typical lookup table requiring an entry for each domain value, longest-prefix matching allows the aggregation of domain values to a single entry. This data structure is typically supported in a forwarding device by a Ternary Content-Addressable Memory (TCAM). Thus, we replace real-time compute-intensive operations with efficient TCAM table lookups.

Our pre-computed function is plotted in Figure 3.4. The dashed lines illustrate the aggregation of the domain values $[147\,456, 155\,647]$ to a single entry with the result set to $y = 18.65214$. In this case, the maximum approximation error is ≈ 0.04 when $f_x = 147\,456$. In general, the magnitude of the error depends on the function curve within the aggregation interval. One should wisely populate the table to meet an adequate trade-off between table entry count and error. In Algorithm 1, we describe our procedure to this end where f is any monotonic function defined within an integer domain. For demonstration, we assume this domain to contain 32-bit integers.

Throughout processing, every incoming packet x triggers $Update(C, x)$ and $\hat{f}_x \leftarrow Estimate(C, x)$. Then, the values of $\hat{f}_x$ are used as keys to obtain the increment to the entropy norm (Equation 3.2). Finally, by the end of each observation window, we calculate the entropy estimates from $\hat{S}$ as we describe next.

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: the authors (2019).

Algorithm 1 LPM Lookup Table Population

```
1:  procedure POPULATE(table, f, domain, max_error)
2:      x ← domain.start
3:      while x ≤ domain.end do
4:          for key.prefix_len ← 1 to 31 do
5:              key.base ← x ∧ ¬(0xffffffff ≫ key.prefix_len)
6:              if key.base ≥ x then
7:                  last ← key.base + 2^(32−key.prefix_len) − 1
8:                  if |f(key.base) − f(last)| ≤ max_error then
9:                      table[key] = (f(key.base) + f(last))/2
10:                     x = last + 1
11:                     break
12:             if key.prefix_len = 31 then
13:                 key.base ← x
14:                 key.prefix_len ← 32
15:                 table[key] ← f(x)
16:                 x ← x + 1
17:                 break
```

Entropy Measurement

In the interest of reducing the processing requirements, we constrain the observation window size to a fixed power of two so that $\log_2(m)$ results in an integer constant and S/m can be implemented as a simple arithmetic shift. Therefore, the entropy estimate is given by:

$$\hat{H} \leftarrow \log_2(m) - (\hat{S} \gg \log_2(m)), \tag{3.3}$$

where $\gg$ denotes an arithmetic shift. We store $\log_2(m)$ within a register so the network operator can change m at runtime.

3.3.2 Traffic Characterization

Anomaly-based intrusion detection has the advantage of dealing with attacks of unknown anatomy and different strengths, but usually requires a bootstrapping, training phase with legitimate traffic. In our proposed mechanism, we also build a model of the legitimate traffic, through the processing of successive entropy estimates.

The entropy time series of the source and the destination IP addresses are summarized independently in terms of an index of central tendency and an index of dispersion. Since the address distributions are legitimately subject to changes throughout time, the proposed mechanism updates this model in real-time. The entropy measurements identi-

fied as malicious by the anomaly detection unit are discarded from the characterization.

Index of Central Tendency

We represent the central tendency of the recent entropy estimates by their *Exponentially Weighted Moving Average* (EWMA) (ROBERTS, 1959), as follows:

$$M_n \leftarrow \alpha \hat{H}_n + (1-\alpha) M_{n-1} \quad \text{with} \quad M_1 = \hat{H}_1, \tag{3.4}$$

where $\alpha \in (0, 1)$ is known as the smoothing coefficient and n is an index representing the observation window. This metric allows parameterizable filtering of short-term fluctuations while giving prominence to the most recent values. In our P4-based design, we make use of a fixed-point representation for the smoothing coefficient. Again, we use a register so the operator may change it at runtime.

Index of Dispersion

Likewise, we measure the dispersion of entropy values through an *Exponentially Weighted Moving Mean Difference* (EWMMD), as follows:

$$D_n \leftarrow \alpha |M_n - \hat{H}_n| + (1-\alpha) D_{n-1} \quad \text{with} \quad D_1 = 0. \tag{3.5}$$

This index denotes the typical spread of entropy measurements relative to the EWMA, being a fundamental feature to the definition of anomaly detection thresholds.

3.3.3 Anomaly Detection

The anomaly detection is performed according to the following conditions:

$$\textit{source IP addresses:} \quad \hat{H}_n > M_{n-1} + k D_{n-1} \tag{3.6}$$

$$\textit{destination IP addresses:} \quad \hat{H}_n < M_{n-1} - k D_{n-1} \tag{3.7}$$

A DDoS attack alarm is triggered whenever any of these two conditions hold. *k* is a configurable parameter proposed as a *sensitivity coefficient*, which scales the detection thresholds. Since it is multiplied by the index of dispersion, this effect is proportional to the traffic characteristics. Increasing *k* results in more rigorous detection conditions, i.e.,

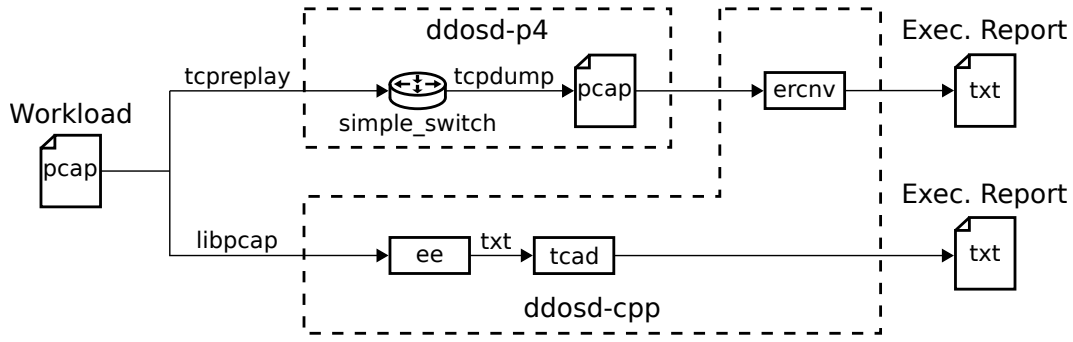
higher assertiveness. “Stealthier” attacks may go unnoticed, nevertheless. A lower value for k, in turn, may expand anomaly detection coverage with the cost of escalating false alarms. It is up to the network operators to determine and adaptively change a value for k to reach an adequate balance between true-positive and false-positive rates.

4 IMPLEMENTATION

In this chapter, we describe our implementation for the proposed DDoS attack detection mechanism. The main goals are to (*i*) verify data-plane execution feasibility conforming to the P4 language primitives and (*ii*) enable a thorough evaluation considering different parameterization and attack conditions. We approach (*i*) with *ddosd-p4* (LAPOLLI, 2019b), a proof-of-concept P4 program based on a simple target behavioral model (Section 4.1). Regarding goal (*ii*), we propose *ddosd-cpp* (LAPOLLI, 2019a), a suite of tools developed in C++ for emulating the mechanism packet processing units with adequate performance for evaluation (Section 4.2). The latter was necessary since performance has not been a design goal of our P4 target. We verify the logical equivalence of these implementations from both a syntactic and a functional analysis (Section 4.3).

Figure 4.1 illustrates the execution flow of our P4 and C++ approaches. We replay a *pcap* file workload to the P4 target running our detection mechanism. Our data plane program annotates the values of internal variables within packets, forwarding them to a specific network interface so we may dump them into a new *pcap* file. In turn, our C++ implementation reads the input file using *libpcap* (The Tcpdump Group, 2019) and emulates our detection mechanism units – entropy estimation (*ee*), traffic characterization and anomaly detection (*tcad*) – to generate an execution report text file. We use a conversion tool (*ercnv*) to translate the P4 *pcap* file output to the same format as the C++ report. These reports serve to functionally validate the equivalence of both implementations and to assess the detection mechanism performance as we explain further in Section 4.3.

Figure 4.1: P4 and C++ Implementation Execution Flow



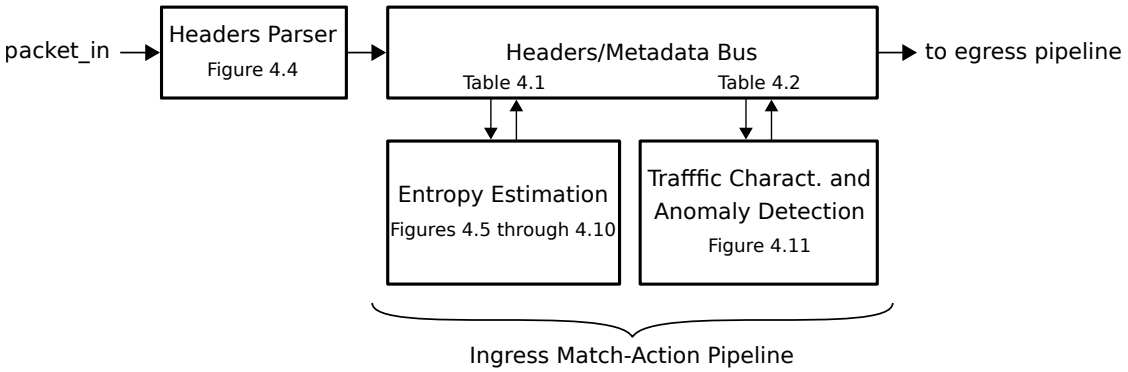
Source: the authors (2019).

4.1 Proof-of-Concept P4 Program

ddosd-p4 is the description of our detection mechanism to run on the *simple_switch* target from BMv2, i.e., the P4 behavioral model software reference implementation (The P4 Language Consortium, 2019a). With this choice, we intend to encourage the adaptation to other devices, which becomes straightforward as this target only contains basic processing functionality.

Figure 4.2 depicts the top-level scheme of our P4 implementation, serving as a reference for the following subsections. Next, we guide through the most relevant excerpts of our code starting by headers definition and parsing (Subsection 4.1.1), and then addressing both the ingress and the egress match-action pipelines (Subsection 4.1.2). For a comprehensive view of our program, one may refer to our publicly available repository (LAPOLLI, 2019b).

Figure 4.2: *ddosd-p4* Top-Level Scheme



Source: the authors (2019).

4.1.1 Headers Definition and Parsing

We take a minimalist approach regarding standard protocols, supporting only Ethernet/IPv4 packets as it is required by our workload (described further in Chapter 5). Yet, extending our program in this respect is an uncomplicated task for being a fundamental design goal of P4. We explore this language feature to define a custom header type `ddosd_t` (Figure 4.3), in which we annotate the values of the detection mechanism variables at runtime to compose the execution report of Figure 4.1. Note that we only implement such header for evaluation purposes as it is not a requirement of our detection mechanism.

Figure 4.3: Custom Header Type Definition

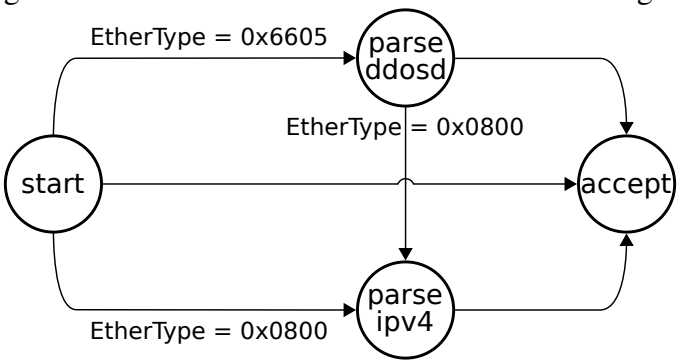
```
1  // EtherType 0x6605
2  header ddosd_t {
3      bit<32> packet_num;
4      bit<32> src_entropy;
5      bit<32> src_ewma;
6      bit<32> src_ewmmd;
7      bit<32> dst_entropy;
8      bit<32> dst_ewma;
9      bit<32> dst_ewmmd;
10     bit<8> alarm;
11     bit<16> ether_type;
12 }
```

Source: the authors (2019).

We use an *EtherType* of `0x6605` to identify our custom data structure and place it after the Ethernet header. The `packet_num` field indicates the packet sequence number within its observation window. The succeeding fields represent the output of the entropy estimation (`src_entropy`, `dst_entropy`), the traffic characterization (`src_ewma`, `dst_ewma`, `src_ewmmd`, `dst_ewmmd`), and the anomaly detection (`alarm`) units. The `ether_type` field serves to identify the subsequent protocol header type.

Figure 4.4 summarizes our header parser with a finite-state machine diagram. We assume all incoming packets to begin with the Ethernet header, which we extract in the `start` state. We proceed by checking the value of the *EtherType* field to extract the `ddosd_t` and the IPv4 headers (if any). Our workload packets follow the `start` → `parse_ipv4` → `accept` path, while the packets carrying information from our detection mechanism variables follow `start` → `parse_ddosd` → `parse_ipv4` → `accept`.

Figure 4.4: Header Parser Finite-State Machine Diagram



Source: the authors (2019).

4.1.2 Match-Action Pipelines

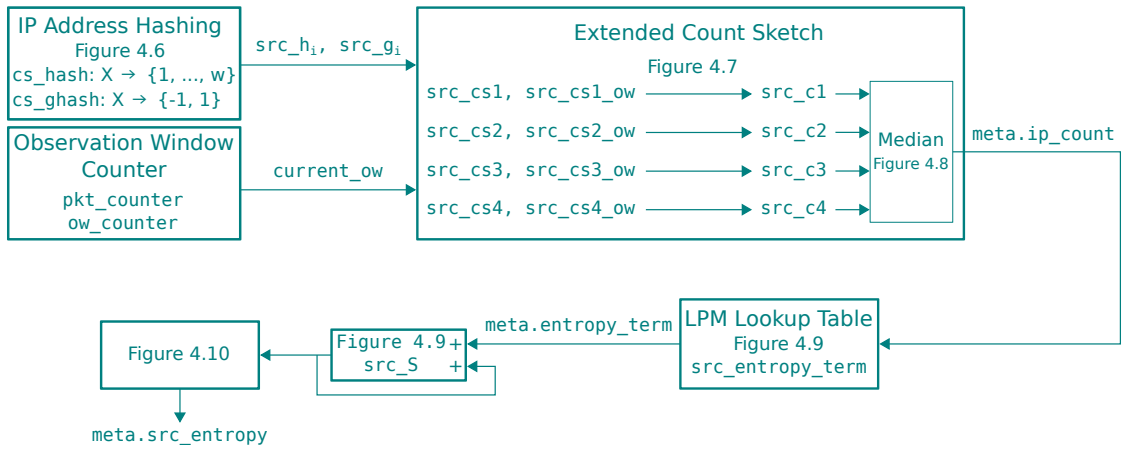
After extracting the packet headers to structured data types, our ingress match-action pipeline is responsible for determining the egress ports, and performing entropy estimation, traffic characterization, and anomaly detection. In turn, the egress pipeline inserts our custom header type to enable an execution report. We set up an LPM IPv4 routing table for packet forwarding. Next, we present our implementation approach concerning the detection mechanism, i.e., the entropy estimation, the traffic characterization, and the anomaly detection units.

Entropy Estimation

Figure 4.5 depicts an overview of our entropy estimation unit implementation, building upon our proposal as presented in Subsection 3.3.1. We annotate references to the code excerpts we explain next. Further, we specify the key variables involved in each processing step.

Table 4.1 summarizes the registers we use to compute the entropy estimates. Their values persist across packet processing, and we may manipulate them both from the data plane program and through the southbound interface; so we use them for holding the mechanism parameters and its stateful variables. We must declare these registers at compile-time which, in our case, precludes the possibility of adaptively changing the count sketch dimensions. Still, such change is possible by reprogramming the P4 target.

Figure 4.5: Entropy Estimation Implementation Top-Level Scheme



Source: the authors (2019).

Table 4.1: Entropy Estimation P4 Registers Summary

Name	Type	Description
`log2_m`	1× `bit<5>`	$\log_2(m)$ parameter allowing the operator to set the observation window size m constrained to a power of two. A bit width of 5 is sufficient to hold the logarithm of up to 2^{32}, which is the maximum possible value for the sketch counters (see `src_csi`, `dst_csi` below).
`pkt_counter`	1× `bit<32>`	Packet counter to identify the end of the observation windows.
`ow_counter`	1× `bit<32>`	Observation window counter to assist on resetting the count sketches.
`src_csi`, `dst_csi`	w× `int<32>`	Counters of the source and destination sketch rows $i = 1, 2, ..., d$.
`src_csi_ow`, `dst_csi_ow`	w× `bit<8>`	Observation window annotation (W_{ID}) of the source and destination sketch rows $i = 1, 2, ..., d$.
`src_S`, `dst_S`	1× `bit<32>`	Source and destination entropy norms represented with 28 integer bits and 4 fractional bits.

Source: the authors (2019).

In order to implement the count sketch, we extend both the *simple_switch* target[1] and the P4 language reference compiler[2] (The P4 Language Consortium, 2019b) to support the hash functions h_i and g_i for $i = 1, 2, ..., d$. Then, in our P4 program, we encapsulate hashing into actions as shown in Figure 4.6. The `cs_hash` action implements h_i functions while `cs_ghash` implements g_i. Lines 2–5 and 9–12 represent calls to our target device hashing operating unit. Since this unit is unable to generate negative numbers, we use a simple transformation to the results from `HashAlgorithm.gi`, i.e. $f(x) = 2x - 1$, so we obtain $\{-1, 1\}$ from $\{0, 1\}$ in lines 15–18.

For each incoming IP packet, we update the sketch counters and obtain the addresses frequency estimate from every sketch row. We demonstrate this process in Figure 4.7. In lines 3–4, we read the W_{ID} annotation (stored in `src_csi_ow`) and then compare it to the current observation window identifier in line 7. If they do not match, we consider the counter value to be outdated and reset it (lines 8–9); otherwise, we read its value (line 11). Next, in lines 13–14, we calculate the new counter value and update it within its register. Finally, we calculate our row estimate in line 16. Note that, in line 7, we only take the least significant byte of the observation window counter (read into

[1]Extension available at <https://github.com/aclapolli/behavioral-model>
[2]Extension available at <https://github.com/aclapolli/p4c>

Figure 4.6: P4 Actions for Count Sketch Hashing

```
action cs_hash(in bit<32> ipv4_addr, out bit<32> h1, out bit<32> h2
    ↪ , out bit<32> h3, out bit<32> h4) {
    hash(h1, HashAlgorithm.h1, 32w0, {ipv4_addr}, 32w0xffffffff);
    hash(h2, HashAlgorithm.h2, 32w0, {ipv4_addr}, 32w0xffffffff);
    hash(h3, HashAlgorithm.h3, 32w0, {ipv4_addr}, 32w0xffffffff);
    hash(h4, HashAlgorithm.h4, 32w0, {ipv4_addr}, 32w0xffffffff);
}

action cs_ghash(in bit<32> ipv4_addr, out int<32> g1, out int<32>
    ↪ g2, out int<32> g3, out int<32> g4) {
    hash(g1, HashAlgorithm.g1, 32w0, {ipv4_addr}, 32w0xffffffff);
    hash(g2, HashAlgorithm.g2, 32w0, {ipv4_addr}, 32w0xffffffff);
    hash(g3, HashAlgorithm.g3, 32w0, {ipv4_addr}, 32w0xffffffff);
    hash(g4, HashAlgorithm.g4, 32w0, {ipv4_addr}, 32w0xffffffff);

    // As the g hashes outputs either 0 or 1, we must map 0 to -1.
    g1 = 2*g1 - 1;
    g2 = 2*g2 - 1;
    g3 = 2*g3 - 1;
    g4 = 2*g4 - 1;
}
```

Source: the authors (2019).

Figure 4.7: Count Sketch Row Frequency Estimation

```
// Row 1 Estimation

bit<8> src_cs1_ow_aux;
src_cs1_ow.read(src_cs1_ow_aux, src_h1);

int<32> src_c1;
if (src_cs1_ow_aux != current_ow[7:0]) {
    src_c1 = 0;
    src_cs1_ow.write(src_h1, current_ow[7:0]);
} else {
    src_cs1.read(src_c1, src_h1);
}
src_c1 = src_c1 + src_g1;
src_cs1.write(src_h1, src_c1);

src_c1 = src_g1*src_c1;
```

Source: the authors (2019).

`current_ow`) to identify a window within the sketch. This approach allows us to reduce the memory space required by the count sketch annotation while maintaining our ability to identify outdated counters with sufficient accuracy.

Having the IP address frequency estimate from every count sketch row, we calculate their median, which we define as an action hardcoding the conditions for every possible combination of elements composing the output (see Figure 4.8). Given an even number of sketch rows d (in our case, $d = 4$), we must check for $\binom{d}{2}$ conditions. Thus, it

Figure 4.8: P4 Action for Median Calculation

```
1  action median(in int<32> x1, in int<32> x2, in int<32> x3, in int
       ↪ <32> x4, out int<32> y) {
2      // This is why we should minimize the sketch depth: the median
           ↪ operator is hardcoded.
3      if (...)
4          y = (x3 + x4) >> 1;
5      else if (...)
6          y = (x2 + x4) >> 1;
7      else if (...)
8          y = (x2 + x3) >> 1;
9      ...
10 }
```

Source: the authors (2019).

is crucial set a low sketch depth so that this operation is feasible within the data plane.

We use the median operator for updating the entropy norm as shown in Figure 4.9. We set its result into a metadata field, i.e., `meta.ip_count` (line 2), which serves as a key to the LPM lookup table. This table issues into `meta.entropy_term` the value which we must accumulate to the entropy norm (line 6). We read the last entropy norm values into `src_S_aux` and update it accordingly (see lines 11–14). We omit this process for the destination address for being analogous. Note that our target device requires that we use different tables (`src_entropy_term` and `dst_entropy_term`) for each address direction as it does not support multiple lookups for the same packet. Since we populate these tables with the exact same entries, their memory space may be shared depending on the device implementation.

Figure 4.9: Entropy Norm Update

```
1  // Count Sketch Source IP Frequency Estimate
2  median(src_c1, src_c2, src_c3, src_c4, meta.ip_count);
3
4  // LPM Table Lookup
5  if (meta.ip_count > 0)
6      src_entropy_term.apply();
7  else
8      meta.entropy_term = 0;
9
10 // Source Entropy Norm Update
11 bit<32> src_S_aux;
12 src_S.read(src_S_aux, 0);
13 src_S_aux = src_S_aux + meta.entropy_term;
14 src_S.write(0, src_S_aux);
```

Source: the authors (2019).

Figure 4.10 illustrates the last step for entropy estimation. Again, we omit the process for the destination addresses. In lines 2–5, we read the binary logarithm of the observation window size into `log2_m` and, from this value, we calculate `m`. While it is not the end of the window, we simply update the packet counter (line 8). Otherwise, we increment the observation window counter (lines 10–11), calculate the entropy considering the fixed point representation (line 13), and reset the variables for the entropy estimation of the next window (lines 18–20).

Figure 4.10: Entropy Estimation Final Step

```
1  // Observation Window Size
2  bit<32> m;
3  bit<5> log2_m_aux;
4  log2_m.read(log2_m_aux, 0);
5  m = 32w1 << log2_m_aux;
6
7  if (meta.pkt_num != m) {
8      pkt_counter.write(0, meta.pkt_num);
9  } else { // End of Observation Window
10     current_ow = current_ow + 1;
11     ow_counter.write(0, current_ow);
12
13     meta.src_entropy = ((bit<32>)log2_m_aux << 4) - (src_S_aux >>
           ↪ log2_m_aux);
14
15     ... // Traffic Characterization and Anomaly Detection
16
17     // Reset
18     pkt_counter.write(0, 0);
19     src_S.write(0, 0);
20 }
```

Source: the authors (2019).

Traffic Characterization and Anomaly Detection

At the end of each observation window, once we calculate the entropy estimates, we perform traffic characterization and anomaly detection as indicated in line 15 of Figure 4.10. Table 4.2 describes the P4 registers used for that purpose. We use 18 fractional bits for the traffic characterization indices. This representation is necessary as the exponentially weighted moving operators depend on the summation of many low-magnitude numbers, so we must minimize error accumulation.

We take into account three conditions for performing traffic characterization and anomaly detection. One is checking for the first observation window reading `ow_counter` to set up the characterization indices. The second is identifying the training phase with respect to `training_len` to suppress the anomaly detection alarm.

Table 4.2: Traffic Characterization and Anomaly Detection P4 Registers Summary

Name	Type	Description
`training_len`	1× bit<32>	Number of observation windows which consist the training phase.
`src_ewma`, `dst_ewma`	1× bit<32>	Source and destination EWMA represented with 14 integer bits and 18 fractional bits.
`src_ewmmd`, `dst_ewmmd`	1× bit<32>	Source and destination EWMMD represented with 14 integer bits and 18 fractional bits.
`alpha`	1× bit<8>	Smoothing coefficient represented with 8 fractional bits.
`k`	1× bit<8>	Sensitivity coefficient represented with 5 integer bits and 3 fractional bits.

Source: the authors (2019).

Third, it is to only update the characterization with entropy values that do not exceed the detection thresholds as we only want to model the legitimate traffic.

Figure 4.11 depicts the P4 code excerpt for traffic characterization and anomaly detection. In lines 1–4, we read the last characterization index values. If we are at the first observation window, we initialize the entropy EWMA and EWMMD (lines 7–10). Otherwise, if we are past the training phase, we calculate the detection thresholds (lines 17–24), and verify if the current entropy estimate exceeds them to issue an alarm (lines 26-27). We only update the characterization indices in absence of alarm (line 30). We read the smoothing coefficient (lines 31–32), and update the indices of central tendency (lines 34–36) and dispersion (lines 38–45).

4.2 C++ Emulation Tools

Since BMv2 does not provide adequate performance for a thorough evaluation of our detection mechanism, we developed tools in C++ to emulate its processing units, i.e., *ee* (Subsection 4.2.1) for entropy estimation, and *tcad* (Subsection 4.2.2) for traffic characterization and anomaly detection. The separation into two stand-alone tools allows to first evaluate the entropy estimation, and then to independently assess the effects of different parameter levels in further processing. In the following subsections, we overview the implementation of each emulation tool. The full source code is publicly available for those interested in an in-depth analysis (LAPOLLI, 2019a).

Figure 4.11: Traffic Characterization and Anomaly Detection

```
1  src_ewma.read(meta.src_ewma, 0);
2  src_ewmmd.read(meta.src_ewmmd, 0);
3  dst_ewma.read(meta.dst_ewma, 0);
4  dst_ewmmd.read(meta.dst_ewmmd, 0);
5
6  if (current_ow == 1) {
7      meta.src_ewma = meta.src_entropy << 14;
8      meta.src_ewmmd = 0;
9      meta.dst_ewma = meta.dst_entropy << 14;
10     meta.dst_ewmmd = 0;
11 } else {
12     meta.alarm = 0;
13
14     bit<32> training_len_aux;
15     training_len.read(training_len_aux, 0);
16     if (current_ow > training_len_aux) {
17         bit<8> k_aux;
18         k.read(k_aux, 0);
19
20         bit<32> src_thresh;
21         src_thresh = meta.src_ewma + ((bit<32>)k_aux*meta.src_ewmmd
               ↪  >> 3);
22
23         bit<32> dst_thresh;
24         dst_thresh = meta.dst_ewma - ((bit<32>)k_aux*meta.dst_ewmmd
               ↪  >> 3);
25
26         if ((meta.src_entropy << 14) > src_thresh || (meta.
               ↪ dst_entropy << 14) < dst_thresh)
27             meta.alarm = 1;
28     }
29
30     if (meta.alarm == 0) {
31         bit<8> alpha_aux;
32         alpha.read(alpha_aux, 0);
33
34         meta.src_ewma =
35             (((bit<32>)alpha_aux*meta.src_entropy) << 6) +
36             (((32w256 - (bit<32>)alpha_aux)*meta.src_ewma) >> 8);
37
38         bit<32> abs_diff;
39         if ((meta.src_entropy << 14) >= meta.src_ewma)
40             abs_diff = (meta.src_entropy << 14) - meta.src_ewma;
41         else
42             abs_diff = meta.src_ewma - (meta.src_entropy << 14);
43         meta.src_ewmmd =
44             (((bit<32>)alpha_aux*abs_diff) >> 8) +
45             (((32w256 - (bit<32>)alpha_aux)*meta.src_ewmmd) >> 8);
46
47         ...
48     }
49 }
```

Source: the authors (2019).

4.2.1 Entropy Estimation through *ee*

We designed the *ee* tool to emulate several independent entropy estimation units set with different sketch dimensions. In addition, for each sketch configuration, *ee* may perform a parameterizable number of repetitions using different random coefficients for the hashing functions. As output, beyond the entropy estimates for every observation window, the tool issues their timestamp (in view of the packet capture metadata), and optionally the exact entropy values to serve as a baseline to assess the estimation error.

In order to model this tool, we followed an object-oriented design composed of four main classes. The `PcapReader` class is responsible for parsing the workload *pcap* file extracting the sequence of source and destination IP addresses. The `ExtendedCount-Sketch` class abstracts the sketch operations $Update$ and $Estimate$ as presented in Subsection 3.3.1. The `LpmLookupTable` class represents the lookup table issuing the values to increment the entropy norm for the given address frequencies. Finally, the `EntropyEstimator` top-level class orchestrates `ExtendedCountSketch` and `LpmLookupTable` objects to compute entropy estimates according to user-defined parameters.

Unlike our P4 implementation, *ee* does not inject the values of its variables into the packet header; it writes them in plain-text to the standard output stream. Therefore, when reading the workload *pcap* file, we only implement the extraction of IP addresses on

Figure 4.12: *pcap* file Reader C++ Class Structure

```
1  struct PcapPacket {
2      ...
3  };
4
5  class PcapReader {
6      public:
7          PcapReader(const std::string& pcap_filename);
8
9          ...
10
11         int nextPacket(PcapPacket& pcap_packet);
12         uint32_t srcIpv4(const PcapPacket& pcap_packet) const;
13         uint32_t dstIpv4(const PcapPacket& pcap_packet) const;
14
15     private:
16         ...
17 };
```

Source: the authors (2019).

top of *libpcap*. We encapsulate such processing within the `PcapReader` class as shown in Figure 4.12. This class processes the *pcap* file given in its constructor (line 7). The `nextPacket` method (line 11) serves to retrieve packets from the traffic stream. The `srcIpv4` and `dstIpv4` methods (lines 12–13) act as a header parser extracting the IP addresses from the given packet.

Once the addresses are read from the workload traffic, the `ExtendedCount-Sketch` class plays a fundamental role estimating their frequency. We depict relevant code excerpts of its implementation in Figure 4.13. We provide two constructors for building an object of this class (lines 1 and 4), both taking the sketch depth and

Figure 4.13: Extended Count Sketch C++ Code Excerpts

```
1  ExtendedCountSketch::ExtendedCountSketch(uint32_t depth, uint32_t
       ↪ width)
2      ...
3
4  ExtendedCountSketch::ExtendedCountSketch(uint32_t depth, uint32_t
       ↪ width, const HashingCoefficients& h_coefficients, const
       ↪ HashingCoefficients& g_coefficients)
5      ...
6
7  int32_t ExtendedCountSketch::update(uint32_t key) {
8      std::vector<int32_t> counts;
9      for (uint32_t i = 0; i < mDepth; ++i) {
10         const uint32_t h = hash(i, key);
11         const int32_t g = ghash(i, key);
12
13         if (mStates[i][h] != mCurrentState) {
14             mCounters[i][h] = g;
15             mStates[i][h] = mCurrentState;
16         } else {
17             mCounters[i][h] += g;
18         }
19
20         counts.push_back(g*mCounters[i][h]);
21     }
22
23     // Median
24     sort(counts.begin(), counts.end());
25     const size_t size = counts.size();
26     if (size % 2 == 0)
27         return (counts[size/2 - 1] + counts[size/2])/2;
28     else
29         return counts[size/2];
30 }
31
32 void ExtendedCountSketch::reset() {
33     ++mCurrentState;
34 }
```

Source: the authors (2019).

width as parameters. When using the first constructor, we auto-generate random coefficients for the hashing functions, while the second allows their assignment from the `HashingCoefficient` input variables. The latter approach allows us to enforce the same parameters as used in the P4 implementation. Within the `update` method, in lines 8–21, we change the sketch to count the insertion of a given `key` (i.e., an IP address). Note that, in lines 13–15, we deal with the event of setting the counter for the first time in the current window. In lines 23–29, we calculate the median to return the current frequency estimate of this `key`. The `reset` method (lines 32–34) increments the current observation window so we may further identify outdated counter values.

Another element for the entropy estimation is the `LpmLookupTable` class, illustrated in Figure 4.14. The constructor (lines 1–2) populates the table from a function with a 32-bit unsigned integer domain `f`, the maximum domain value `max`, and the maximum approximation error allowed `max_error`; it follows the Algorithm 1 from Subsection 3.3.1. The `get` method (lines 5–17) looks for the longest-prefix key matching the input parameter `x`.

The `ExtendedCountSketch` and the `LpmLookupTable` classes compose the `EntropyEstimator` class as shown in Figure 4.15. The `f` method (lines 1–9) is the function which we map to the LPM lookup table. The `update` method uses sketch objects to obtain the frequency estimates of incoming IP addresses (line 12), and

Figure 4.14: LPM Lookup Table C++ Code Excerpts

```
1  template<typename T>
2  LpmLookupTable<T>::LpmLookupTable(std::function<T(uint32_t)> f,
       ↪ uint32_t max, T max_error)
3      ...
4
5  template<typename T>
6  T LpmLookupTable<T>::get(uint32_t x) const {
7      for (int8_t prefix_len = 32; prefix_len >= 0; --prefix_len) {
8          LpmLookupKey key;
9          key.base = prefix_len == 32? x : x & ~(0xffffffff >>
               ↪ prefix_len);
10         key.prefix_len = static_cast<uint8_t>(prefix_len);
11         if (mLookupTable.find(key) != mLookupTable.end()) {
12             return mLookupTable.at(key);
13         }
14     }
15
16     throw std::runtime_error(...);
17 }
```

Source: the authors (2019).

Figure 4.15: Entropy Estimator C++ Code Excerpts

```
1  uint32_t EntropyEstimator::f(uint32_t x) {
2      if (x < 2)
3          return 0;
4      return static_cast<uint32_t>(
5                  round(pow(2, 4)*(
6                            x*log2(static_cast<double>(x)) -
7                            (x - 1)*log2(static_cast<double>(x - 1))
8                       )));
9  }
10
11 void EntropyEstimator::update(uint32_t src_ipv4, uint32_t dst_ipv4)
       ↪  {
12     const int32_t src_fx = mSrcSketch.update(src_ipv4);
13     if (src_fx > 0)
14         mSrcS += mLookupTable.get(static_cast<uint32_t>(src_fx));
15
16     ...
17 }
18
19 void EntropyEstimator::reset() {
20     mSrcSketch.reset();
21     mSrcS = 0;
22     ...
23 }
24
25 uint32_t EntropyEstimator::srcEntropy() const {
26     return (mLog2M << 4) - (mSrcS >> mLog2M);
27 }
```

Source: the authors (2019).

then increment the entropy norms with their respective term taken from the lookup table (lines 13–14). The `reset` method (lines 19–23) clears the sketches and the entropy norms preparing for the entropy estimation of the next observation window. Finally, the `srcEntropy` method (lines 25–27) illustrates the final step for entropy calculation.

The *ee* tool takes a JSON configuration file to parameterize the `EntropyEstimator` objects. It reads the input *pcap* file using an instance of the `PcapReader` class keeping a packet counter to identify the end of the observation windows and reset the estimators. At that instant, it outputs space-delimited values of the current packet timestamp and the entropy estimates.

4.2.2 Traffic Characterization and Anomaly Detection through *tcad*

The *tcad* tool reads the output of the entropy estimation to perform traffic characterization and anomaly detection. It computes and appends to the each entropy estimate

pair (source and destination) the traffic characterization indices and the alarm status.

The `TrafficCharacterizer` class depicted in Figure 4.16 is the main building block of the *tcad* tool. Its constructor only takes the smoothing coefficient `alpha` (lines 1–2). The `update` method processes the incoming source and destination entropy estimates (lines 4–23). In lines 6–10, if it is the first `update` call, we initialize the characterization indices. Otherwise, we calculate the new values for the EWMA and EWMMD in lines 12–21. Similarly to our P4 implementation, we only operate with integer number arithmetic and bit shifting.

Figure 4.16: Traffic Characterizer C++ Code Excerpts

```
1  TrafficCharacterizer::TrafficCharacterizer(uint8_t alpha)
2      : mAlpha(alpha), mSetup(false) { }
3
4  void TrafficCharacterizer::update(uint32_t src_entropy, uint32_t
       ↪ dst_entropy) {
5      if (!mSetup) {
6          mSrcEwma = src_entropy << 14;
7          mSrcEwmmd = 0;
8          mDstEwma = dst_entropy << 14;
9          mDstEwmmd = 0;
10         mSetup = true;
11     } else {
12         mSrcEwma = ((mAlpha*src_entropy) << 6) +
13                    (((256 - mAlpha)*mSrcEwma) >> 8);
14
15         uint32_t abs_diff = mSrcEwma > (src_entropy << 14)?
16                                 mSrcEwma - (src_entropy << 14) :
17                                 (src_entropy << 14) - mSrcEwma;
18         mSrcEwmmd = ((mAlpha*abs_diff) >> 8) +
19                     (((256 - mAlpha)*mSrcEwmmd) >> 8);
20
21         ...
22     }
23 }
```

Source: the authors (2019).

Further, we perform anomaly detection within the main function block. We show this process in Figure 4.17. In lines 1–4, we calculate the detection thresholds from the characterization indices and the sensitivity coefficient `k`. In line 6, we check if the current entropy estimates exceeds these thresholds and define the alarm status. Finally, we proceed with the update of the traffic characterization only if an alarm has not been issued (lines 7–8).

Figure 4.17: Anomaly Detection C++ Code

```
1 const uint32_t src_thresh = characterizer->srcEwma() +
2                             ((k*characterizer->srcEwmmd()) >> 3);
3 const uint32_t dst_thresh = characterizer->dstEwma() -
4                             ((k*characterizer->dstEwmmd()) >> 3);
5
6 const bool alarm = ow_counter > training_length && ((src_entropy <<
    ↪  14) > src_thresh || (dst_entropy << 14) < dst_thresh);
7 if (!alarm)
8     characterizer->update(src_entropy, dst_entropy);
```

Source: the authors (2019).

4.3 P4 and C++ Implementation Equivalence Analysis

We approach the equivalence analysis of our P4 and C++ implementations in a twofold manner. The first is a syntactic validation from the comparison of our descriptions in each language in terms of their control flow, number representation, and arithmetical operations. The second is a functional validation from the inspection of variable values at runtime.

We enforce the syntactical equivalence of our implementations by design. As evidence of this claim, we may contrast the code excerpts in Section 4.1 and Section 4.2. Note the number representation mapping, in which, for instance, `bit<32>` translates to `uint32_t` whereas `int<32>` maps to `int32_t`. Further, observe the equivalence of conditional executions and the absence of floating point arithmetic within our C++ code.

Even beyond this validation during design, we provide a method for checking the equivalence post-compilation. We set our P4 program to clone the last packet of each observation window, insert the custom header type `ddosd_t`, and forward it to a specific network interface. We dump these packets to a *pcap* file so we may compare the execution information with our emulation results. We use the *ercnv* tool to extract the information from the packet capture and write it to a text file using the same format as the *tcad* output.

We implement our custom header type parsing within the `PcapReader` class. We define a C struct `ddosd_t` composed by unsigned integers (`uint8_t`, `uint16_t`, `uint32_t`) representing the header fields presented in Figure 4.3. Figure 4.18 illustrates the extraction process. We check for the EtherType `0x6605` within the L2 header (lines 2–3), extract the header bytes (line 4), and convert the fields encoding to the host byte order (line 5).

We write the values of our custom header fields to the standard output, which we

redirect to a text file. Then, we verify the equality of variable values throughout their P4 and C++ implementation execution by comparing their textual report files byte by byte. In such a way, we ensure the conformity of our C++ emulation to our P4 program regarding their runtime state so we may proceed with a reliable evaluation of our detection mechanism.

Figure 4.18: `ddosd_t` Header Parsing

```
1 uint32_t PcapReader::ddosdSrcEntropy(const PcapPacket& pcap_packet)
    ↪ const {
2     if (l2EtherType(pcap_packet) != ETHERTYPE_DDOSD)
3         throw std::runtime_error(...);
4     const struct ddosd_t* ddosd_header = reinterpret_cast<const
        ↪ struct ddosd_t*>(pcap_packet.data + l2HeaderLength());
5     return ntohl(ddosd_header->src_entropy);
6 }
```

Source: the authors (2019).

5 EVALUATION

As far as we are aware of, this work is the first to explore data plane programmability, more specifically P4, to devise a sophisticated anomaly detection mechanism. Given the limited primitive set made available by P4 and the consequent simplifications that were mandatory in our design, it is paramount to assess the *accuracy*, *resource utilization* and *timeliness* of our proposed mechanism, comparing it with state-of-the-art approaches. In this section, we evaluate it aiming to answer the following three research questions:

- *RQ1: How accurate is the entropy estimation processing pipeline as a function of resource utilization footprint?*
- *RQ2: Assuming decent entropy estimation (RQ1), how accurate is the DDoS attack detection mechanism under different tuning parameters and attack strengths?*
- *RQ3: How does our mechanism compare to existing monitoring approaches regarding detection accuracy and latency?*

First, in Section 5.1, we describe the experimental setup and the evaluation methodology. Then, in each of the remaining sections, we discuss one of the questions above.

5.1 Experimental Setup and Evaluation Methodology

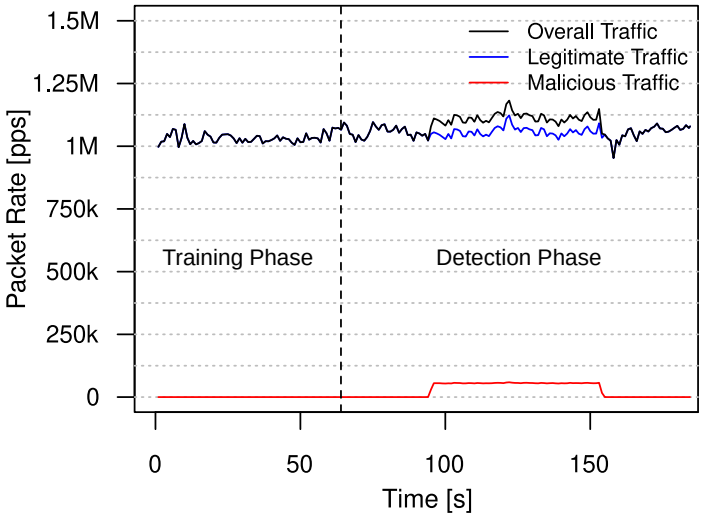
Given the novelty of P4 and the still scarce availability of equipment implementing it, we evaluate the proposed DDoS attack detection mechanism using a software-based P4 infrastructure. This setup does not limit our assessment, as both accuracy and resource utilization are expected to be equivalent regardless of the P4 target.

We use as legitimate traffic packet traces from the *CAIDA Anonymized Internet Traces 2016* (CAIDA, 2016) dataset, recorded from high-speed Internet backbone links. To represent DDoS attacks, we take packets from the *CAIDA DDoS Attack 2007* dataset (CAIDA, 2007), consisting of an attempt to consume the computing resources of a target server and to congest the network links connecting this server to the Internet. Despite not recent, this dataset was carefully built to only include attack-related traces and, for this reason, is consistently employed in high-impact publications in the area of network security. Furthermore, this choice is consistent with the scenario we introduced earlier in Subsection 3.1.

We set the observation window size m to 2^{18}, representing approximately $250\,\mathrm{ms}$ for the given workload average packet rate. We partition the workload into a training and a detection phase with respectively 250 and 500 observation windows. The training phase consists of only legitimate traffic serving the purpose of setting up the characterization model. For the detection phase, we take the subsequent legitimate traffic and superimpose it with attack packets from the 126th to the 375th observation window. We perform such superimposition at different malicious-to-legitimate packets proportions (3%, 3.5%, ..., 6%), generating a total of 7 workloads.

Figure 5.1 depicts the packet rate for the workload with the malicious traffic representing 5% of the overall traffic volume. The legitimate traffic ranges from $952\,\mathrm{kpps}$ ($\approx 1.43\,\mathrm{GB/s}$) to $1.12\,\mathrm{Mpps}$ ($\approx 1.68\,\mathrm{GB/s}$) while the attack approaches $59\,\mathrm{kpps}$ ($\approx 88.5\,\mathrm{MB/s}$). Such malicious-to-legitimate traffic proportion may either represent low-rate semantic DDoS attacks, the first stages of a volumetric attack, or a fraction of the malicious traffic passing through a link.

Figure 5.1: Workload packet rate with the malicious traffic representing 5% of the overall traffic volume.



Source: the authors (2019).

Table 5.1 summarizes the system factor levels set throughout the experiments. We select varying ranges for data structure size and sensitivity coefficient to allow a broad assessment of the proposed mechanism. We execute 15 repetitions for each configuration with random hashing coefficients and present the results using a 95% confidence level. In Section 5.2, we examine the relative error of the entropy estimates for different count sketch depth and width values. Next, in Section 5.3, we investigate the detection True-Positive Rate (TPR), False-Positive Rate (FPR), and accuracy with respect to the sensi-

Table 5.1: System Factor Levels

System Factors	Levels Used in Each Subsection		
	5.2	**5.3**	**5.4**
Hashing Coefficients (a_i, b_i)	random co-prime pairs		
Count Sketch Depth (d)	$\{4, 8, 16\}$	4	4
Count Sketch Width (w)	$\{64, 368, 672, 976, 1280\}$		1280
Sensitivity Coefficient (k)	NA	$\{0, 0.5, 1, ..., 8\}$	4

Source: the authors (2019).

tivity coefficient, the memory footprint, and the different proportions of malicious traffic volume. Finally, in Section 5.4, we compare our mechanism with approaches based on packet sampling regarding detection accuracy and latency.

5.2 Entropy Estimation Error

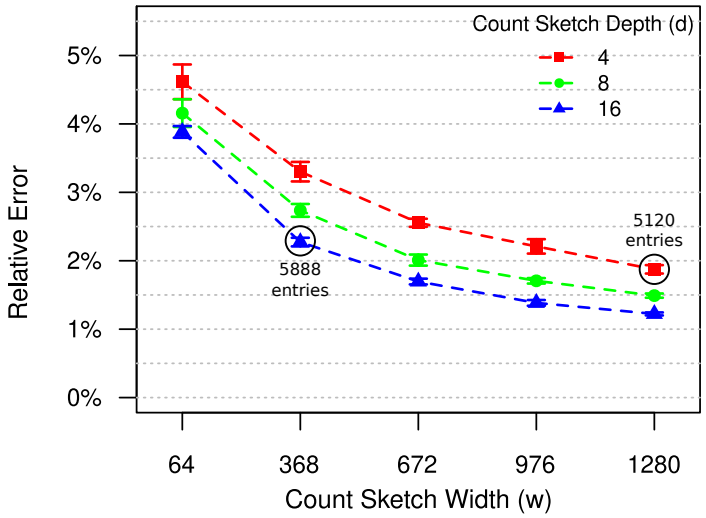
Instead of focusing on the calculation of exact entropy values, we propose an estimation processing pipeline that minimizes memory space and processing time. However, the loss of accuracy in this process has the potential to undermine the detection performance by hiding anomalous traffic patterns. On that account, we assess the relative error of the estimates as a function of the count sketch dimensions, which represent the dominant influential factors to correctness (RQ1).

We allocate a 32-bit register for each sketch counter and an 8-bit register for its associated observation window identifier. We store $\hat{S}$ and $\hat{H}$ in 32-bit registers considering a fixed-point representation having 4 fractional bits. We build the LPM lookup table ensuring a maximum error of 2^{-4} for each entry, resulting in a total of 245 TCAM entries.

Figure 5.2 presents the relative estimation error for the count sketch depth and width levels listed in Table 5.1 (first column). The sketch width is directly related to the probability of hashing collisions on each row. Along the horizontal axis, it is possible to observe how this parameter affects the estimation error. The errors reduce as we increase the width, but this reduction attenuates for larger widths and stabilizes close to 1%. Note that this error also results from the approximations present in the pre-computed LPM lookup table entries.

The increment of the sketch depth reduces the probability of getting the estimate

Figure 5.2: Relative error of the entropy estimation as a function of the count sketch width and depth.



Source: the authors (2019).

from a counter that has been affected by collisions. We observe this effect examining the different error values for a single sketch width. However, increasing the sketch depth implies (*i*) processing more hash functions for each IP address and (*ii*) increasing the complexity of the median operation. The annotations of Figure 5.2, indicating the total number of sketch entries (5 888 and 5 120) in two specific configurations ($d = 16$ and $d = 4$, respectively), reveal that, for comparably sized sketches, the use of more hashes (rows) does not result in significantly better estimates. Thus, we choose to set $d = 4$ in the subsequent experiments.

5.3 DDoS Attack Detection Performance

The proposed mechanism allows network operators to adjust the trade-off between the TPR and the FPR using the sensitivity coefficient k. To answer RQ2, we first take these metrics into account to tune this parameter (Subsection 5.3.1). Then, we investigate the detection accuracy regarding malicious traffic volume and memory utilization (Subsection 5.3.2).

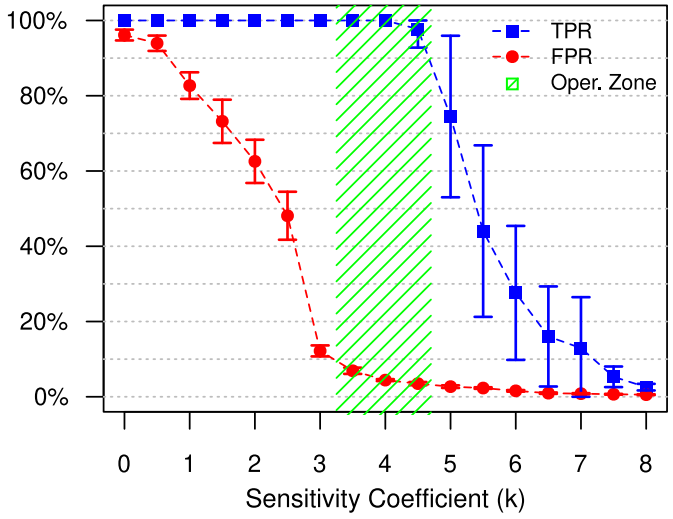
5.3.1 Sensitivity Coefficient Effect

Figure 5.3 presents the true-positive and false-negative detection rates in terms of the sensitivity coefficient. We consider a true positive whenever the mechanism triggers

the alarm for an observation window containing malicious packets; conversely, a false positive occurs if the alarm is set for a window with no malicious packets. The results are for the sketch dimensions $d = 4$, $w = 1\,280$, and the smoothing coefficient $\alpha = 20 \cdot 2^{-8}$. The proportion of the malicious traffic to the overall traffic during the attack is 5%.

Lower sensitivity coefficient values tighten the detection thresholds resulting in higher detection ratios at the cost of false positives. As we increase the coefficient, both the TPR and the FPR decrease to the point where the detection is utterly insensitive. The FPR starts to decrease from $k = 0$ and reaches less than 10% for k within $[3.25, 4.75]$, while the TPR is still close to 100%. This region (green hachure) represents the configuration in which the detection thresholds are expected to be set, i.e., between the entropy estimates of legitimate and malicious traffic. It characterizes the desired operating zone. Given the dynamic nature of traffic in production networks, the value of k may need to be adapted on a periodic basis. This will be addressed in future work.

Figure 5.3: Impact of the sensitivity coefficient k on the true-positive and false-negative rates. The area in green highlights the desired operating zone.

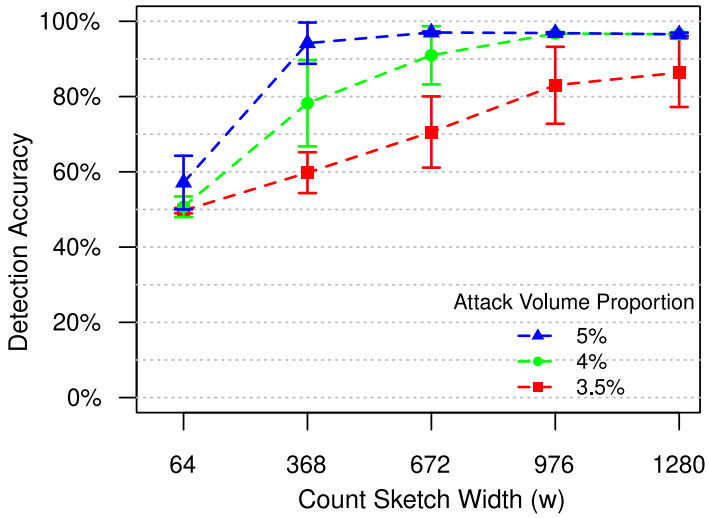


Source: the authors (2019).

5.3.2 DDoS Attack Detection Accuracy

With the sensitivity coefficient k set to 3.5, we now consider the resulting attack detection accuracy achieved with our proposed mechanism (see Figure 5.4). The analysis is carried out considering various attack proportions and count sketch widths (see Table 5.1).

Figure 5.4: DDoS Attack Detection Accuracy in terms of Memory Utilization for Different Proportions of Malicious Traffic



Source: the authors (2019).

As malicious traffic becomes more aggressive, i.e., assumes a higher proportion when compared to the legitimate traffic, the detection accuracy achieves increasingly higher rates (exceeding 90%). This accuracy is profoundly influenced by the count sketch width. Note that even lower magnitude attacks (3.5%) can be decently detected (with rates higher than 80%) as one parameterizes the mechanism with larger w (greater than 976). However, one must recognize that for the cases of lower volume attacks, the anomalous variation of entropy is attenuated and consequently harder to detect. This difficulty is intrinsic to anomaly-based attack detection and is exacerbated when considering lower count sketch widths, which result in less accurate entropy estimates.

If on the one hand, the use of larger sketches leads to higher attack detection accuracy, on the other, it implies a larger memory footprint. Considering 32 bits are allocated for each sketch counter and 8 bits for its associated observation window identifier, the cost for the source and destination IP addresses sketches is $38.125\,\mathrm{kB}$ when $d = 4$ and $w = 976$. This value is the memory space required for monitoring a single $1\,\mathrm{Gbps}$ link. For higher traffic rates, we would need to increase the observation window size to get a robust representation of the addresses distribution. Since the count sketch estimation error is proportional to $1/\sqrt{w}$ and to the square root of the observation ℓ_2 norm (CHARIKAR; CHEN; FARACH-COLTON, 2002), we would have to use proportionally larger sketches to obtain an equivalent entropy estimation accuracy. Hence, considering a 24x10 Gbps programmable forwarding device (BOSSHART et al., 2013), we extrapolate our mecha-

nism memory footprint to $9\,\mathrm{MB}$[1], which represents 18% of the available SRAM.

5.4 Comparison with Packet Sampling

By collecting information from every packet, programmable forwarding devices have the potential to detect very subtle traffic anomalies. In contrast, packet sampling approaches provide information at a coarser granularity, thus being less sensitive to such conditions. We investigate this difference by comparing our mechanism with an implementation of the same DDoS attack detection strategy fed by an sFlow collector (RQ3).

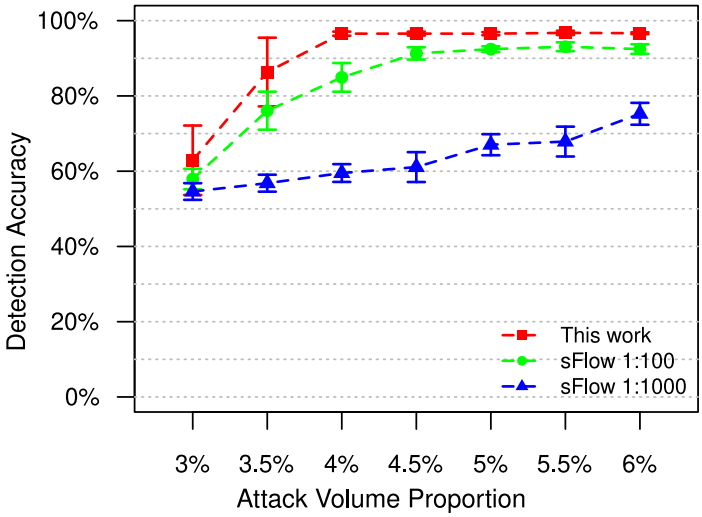
We evaluate the sFlow implementation with the sampling rate set to $1{:}1\,000$, as it is the suggested for a $1\,\mathrm{Gbps}$ link (PHAAL, 2009), and to 1:100 aiming to get even more optimized results. In order to analyze our approach and the two sFlow-based scenarios considering a comparable baseline, we set different values for m in each implementation seeking to represent approximately the same time duration. For instance, during the time our proposed mechanism reads m packets, the sFlow collector outputs only about $m/1\,000$ or $m/100$ depending on the chosen sampling rate. Hence, we scale the m so that all implementations consider the same time frame to trigger DDoS attack alarms.

Figure 5.5 depicts the DDoS attack detection accuracy for each approach considering different volume proportions for the malicious traffic. With the sampling rate at 1:1 000, the detection performance is severely degraded. At a 1:100 rate, the sFlow approach results are greatly improved, but our work still outperforms its accuracy in every observed condition.

We also consider the detection latency by measuring the time between the timestamp of the first malicious packet and the timestamp of the last packet of the observation window which set off the alarm. For the case of low-intensity attacks ($\leq$4%), we observe increased latency – in the order of seconds – to detect an attack when using packet sampling. Our proposed mechanism for a similar condition requires a fraction of the time (a few hundred miliseconds). The higher sensibility of our proposal may lead to earlier triggering of mitigation actions, possibly preventing service outages.

[1] We assume the ℓ_2 norm to increase at the same proportion as the traffic rate.

Figure 5.5: DDoS attack detection accuracy: comparison with packet sampling approaches.



Source: the authors (2019).

6 CONCLUSION

This thesis sheds light on the potential of exploring programmable data planes for network monitoring. We presented a real-time DDoS attack detection mechanism, implemented in P4, to be executed entirely in the data plane. The evaluation results showed that our mechanism could detect DDoS outbreaks quickly and accurately (Section 5.3), especially when compared with existing monitoring approaches (Section 5.4), while meeting strict memory space and processing time budgets associated with in-network packet processing (Section 5.2).

As a sign of the tangible impact of our research, this work was presented at the 16th IFIP/IEEE International Symposium on Integrated Networks (LAPOLLI; MARQUES; GASPARY, 2019) being granted the best student paper award. Those interested in further investigating our mechanism and building complementary solutions may refer to our P4 and C++ implementation publicly available repositories (LAPOLLI, 2019b; LAPOLLI, 2019a). As an additional contribution, we share next several insights which we took from our design and implementation process, expecting they are valuable for new developments in the area (Section 6.1). Finally, we present our plans for future work (Section 6.2).

6.1 Design and Implementation Insights

The instruction set available in P4 is very restricted. For example, there is no support for iteration/recursion (except for header parsing), floating-point arithmetic, and non-elementary mathematical functions. These language limitations are due to constraints in the current programmable hardware and help to prevent stalls in the processing pipeline. As a consequence, implementing sophisticated network functions (e.g., anomaly detection, load balancing) in P4 may be challenging. Next, we discuss the major lessons learned in the process of designing the proposed mechanism.

Iterative procedures need to be carefully decomposed into small tractable steps triggered by incoming packets. In our work, this observation came from two design challenges: (*i*) the summation of individual address frequencies for entropy estimation (Equation 3.1) and (*ii*) resetting the sketch counters between observation windows to avoid using outdated values. Iterating over the entire sketch during the processing of a single packet would violate line rate packet processing requirements. Our mechanism handles the challenge (*i*) by calculating the entropy gradually; it only accesses entries relative to

the addresses of each incoming packet. We deal with the challenge (*ii*) by augmenting entries with observation window identifiers. We check this annotation to reset a counter value only by the time it needs to be reused to support a calculation referring to a new window.

Non-elementary mathematical functions may be approximated using LPM lookup tables. Mathematical functions that build upon functions such as logarithm cannot be directly implemented in current programmable devices. When trying to adapt one of these functions to run at the data plane, it is important to analyze its image and argument bounds. Our mechanism uses LPM lookup tables with pre-calculated values to approximate the function for updating the entropy estimate (Equation 3.2). This function has well-defined argument bounds (i.e., each frequency cannot be higher than the observation window size) and a strict image set (Figure 3.4). Both of these properties enable having a memory-efficient LPM table by compressing entries with close values without significant loss in accuracy.

Floating-point support may not be essential to express numbers with a specific precision within a confined known range. Traditional packet forwarding does not require floating-point arithmetic. Thus, forwarding devices typically only provide instructions over integers. As an upside, integer arithmetic can be applied to handle fractional numbers assuming a fixed-point representation. Throughout our work, we used a fixed-point representation with a 2^4 scaling factor to express real numbers. These numbers in our mechanism are the entropy norm, the entropy itself, the smoothing coefficient, and both the indices of central tendency and of dispersion. Our experimentation showed that it is sufficiently accurate to detect DDoS attacks.

The absence of dynamic memory allocation functionality in the data plane can hamper mechanism self-tuning. The proposed mechanism has some parameters (i.e., m, k, and α) whose values could be modified at runtime through register updates. Other parameters (i.e., d and w) cannot be changed by the in-switch logic, demanding a new P4 program to be deployed on the forwarding devices by an external controller. The reason is that a P4 program cannot allocate dynamic memory. The implementation of more complex self-tuning capability, therefore, requires the investigation of novel data structures.

6.2 Future Work

We organize our future work into three axis. The first one is to extend our evaluation with a hardware P4 target and a more diverse workload set. Second, with the results of this process, we intend to further explore P4 to improve our detection mechanism. Third, we envisage approaching DDoS attack mitigation using the proposed detection mechanism as a building block.

In this work, due to the unavailability of P4 equipment, we resorted to a software-based infrastructure in which we cannot fully assess the implementation costs within high-rate forwarding devices. By using a hardware P4 target, we aim to precisely measure the overhead of our mechanism processing logic. In this new evaluation setup, we also plan to consider other DDoS attacks traces to validate our detection strategy against distinct malicious traffic patterns.

Our anomaly detection unit depends upon a sensitivity coefficient which we assume to be parameterized by the network operator. Provided with the new evaluation results, we expect to improve this processing unit with an artificial intelligence-based approach towards self-tuning capability. Hence, we seek a detection strategy which autonomously adapts to the traffic characteristics.

Finally, we identify an opportunity for performing DDoS attack mitigation from the in-network statistics computed by our mechanism. The first step is to convert our detection alarm to a continuous variable representing a degree of attack concern. Then, we may disseminate this information throughout the network by annotating it within packets. Thus, we are able to establish communication and coordination among attack sensors to obtain a more extensive view of malicious sources.

REFERENCES

ALCOY, P. et al. **NETSCOUT Arbor's 13^{th} Annual Worldwide Infrastructure Security Report**. 2018. Available from Internet: <https://pages.arbornetworks.com/rs/082-KNA-087/images/13th_Worldwide_Infrastructure_Security_Report.pdf>.

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, jan. 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. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 44, n. 2, p. 44–51, abr. 2014. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2602204.2602211>.

BOSSHART, P. et al. P4: Programming Protocol-Independent Packet Processors. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2656877.2656890>.

BOSSHART, P. et al. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In: **Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM**. New York, NY, USA: ACM, 2013. (SIGCOMM '13), p. 99–110. ISBN 978-1-4503-2056-6. Available from Internet: <http://doi.acm.org/10.1145/2486001.2486011>.

CAIDA. **The CAIDA UCSD "DDoS attack 2007" dataset**. 2007. Available from Internet: <https://www.caida.org/data/passive/ddos-20070804_dataset.xml>.

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

CHARIKAR, M.; CHEN, K.; FARACH-COLTON, M. Finding Frequent Items in Data Streams. In: **Proceedings of the 29^{th} International Colloquium on Automata, Languages and Programming**. London, UK, UK: Springer-Verlag, 2002. (ICALP '02), p. 693–703. ISBN 3-540-43864-5. Available from Internet: <http://dl.acm.org/citation.cfm?id=646255.684566>.

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>.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The Road to SDN: An Intellectual History of Programmable Networks. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 44, n. 2, p. 87–98, abr. 2014. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2602204.2602219>.

GARCÍA-TEODORO, P. et al. Anomaly-based network intrusion detection: Techniques, systems and challenges. **Computers & Security**, v. 28, n. 1, p. 18–28, 2009. ISSN 0167-4048. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167404808000692>.

GIL, T. M.; POLETTO, M. MULTOPS: a data-structure for bandwidth attack detection. In: **Proceedings of the 10th Conference on USENIX Security Symposium**. Berkeley, CA, USA: USENIX Association, 2001. (SSYM'01, v. 10). Available from Internet: <http://dl.acm.org/citation.cfm?id=1251327.1251330>.

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. Available from Internet: <http://doi.acm.org/10.1145/3005745.3005748>.

HILTON, S. **Dyn Analysis Summary Of Friday October 21 Attack**. 2016. Available from Internet: <https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/>.

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, jul. 2015. ISSN 1553-877X.

HOQUE, N.; KASHYAP, H.; BHATTACHARYYA, D. K. Real-time DDoS attack detection using FPGA. **Computer Communications**, Elsevier Science Publishers B. V., Amsterdam, Netherlands, v. 110, n. C, p. 48–58, set. 2017. ISSN 0140-3664. Available from Internet: <https://doi.org/10.1016/j.comcom.2017.05.015>.

KIM, C. et al. **In-band Network Telemetry (INT)**. 2016. Available from Internet: <https://p4.org/assets/INT-current-spec.pdf>.

KOTTLER, S. **February 28th DDoS Incident Report**. 2018. Available from Internet: <https://githubengineering.com/ddos-incident-report/>.

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.

LAKHINA, A.; CROVELLA, M.; DIOT, C. Mining Anomalies Using Traffic Feature Distributions. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 35, n. 4, p. 217–228, ago. 2005. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1090191.1080118>.

LAPOLLI, A. C. **ddosd-cpp**. 2019. Available from Internet: <https://github.com/aclapolli/ddosd-cpp>.

LAPOLLI, A. C. **ddosd-p4**. 2019. Available from Internet: <https://github.com/aclapolli/ddosd-p4>.

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)**. Arlington, VA, USA: IEEE, 2019. p. 19–27. ISSN 1573-0077.

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. Available from Internet: <http://doi.acm.org/10.1145/2934872.2934906>.

MA, X.; CHEN, Y. DDoS Detection Method Based on Chaos Analysis of Network Traffic Entropy. **IEEE Communications Letters**, v. 18, n. 1, p. 114–117, jan. 2014. ISSN 1089-7798.

MCKEOWN, N. et al. OpenFlow: Enabling Innovation in Campus Networks. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1355734.1355746>.

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. Available from Internet: <http://doi.acm.org/10.1145/2491185.2491196>.

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. Available from Internet: <http://doi.acm.org/10.1145/3098822.3098829>.

ÖKE, G.; LOUKAS, G. A Denial of Service Detector based on Maximum Likelihood Detection and the Random Neural Network. **The Computer Journal**, Oxford University Press, Oxford, UK, v. 50, n. 6, p. 717–727, nov. 2007. ISSN 0010-4620. Available from Internet: <http://dx.doi.org/10.1093/comjnl/bxm066>.

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>.

REBECCHI, F. et al. DDoS protection with stateful software-defined networking. **International Journal of Network Management**, v. 29, n. 1, p. e2042, 2019. E2042 nem.2042. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.2042>.

ROBERTS, S. W. Control Chart Tests Based on Geometric Moving Averages. **Technometrics**, Taylor & Francis, v. 1, n. 3, p. 239–250, ago. 1959. Available from Internet: <https://www.tandfonline.com/doi/abs/10.1080/00401706.1959.10489860>.

SHANNON, C. E. A mathematical theory of communication. **Bell Systems Technical Journal**, v. 27, p. 623–656, jul. 1948.

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. Available from Internet: <http://doi.acm.org/10.1145/3050220.3063772>.

TENNENHOUSE, D. L.; WETHERALL, D. J. Towards an active network architecture. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 37, n. 5, p. 81–94, out. 2007. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1290168.1290180>.

The Open Networking Foundation. **OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06)**. 2015. Available from Internet: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>.

The Open Networking Foundation. **ONOS**. 2018. Available from Internet: <https://onosproject.org>.

The OpenDaylight Foundation. **OpenDaylight**. 2019. Available from Internet: <https://www.opendaylight.org>.

The P4 Language Consortium. **P4$_{16}$ Language Specification version 1.0.0**. 2017. Available from Internet: <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf>.

The P4 Language Consortium. **BMv2**. 2019. Available from Internet: <https://github.com/p4lang/behavioral-model>.

The P4 Language Consortium. **p4c**. 2019. Available from Internet: <https://github.com/p4lang/p4c>.

The Tcpdump Group. **libpcap**. 2019. Available from Internet: <https://www.tcpdump.org/>.

XIA, W. et al. A Survey on Software-Defined Networking. **IEEE Communications Surveys Tutorials**, v. 17, n. 1, p. 27–51, jun. 2015. ISSN 1553-877X.

XU, Y.; LIU, Y. DDoS attack detection under SDN context. In: **IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications**. San Francisco, CA, 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. Available from Internet: <http://doi.acm.org/10.1145/3230543.3230544>.

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