Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

INGEGNERIA ELETTRONICA, TELECOMUNICAZIONI E TECNOLOGIE DELL'INFORMAZIONE

Ciclo 33

Settore Concorsuale: 09/F2 - TELECOMUNICAZIONI

Settore Scientifico Disciplinare: ING-INF/03 - TELECOMUNICAZIONI

DESIGN AND DEVELOPMENT OF NETWORK MONITORING STRATEGIES IN P4-ENABLED PROGRAMMABLE SWITCHES

Presentata da: Damu Ding

Coordinatore Dottorato

Alessandra Costanzo

Supervisore

Domenico Siracusa

Co-supervisore

Alessandra Costanzo

Marco Savi

Federico Pederzolli

Esame finale anno 2021

Abstract

Network monitoring is of paramount importance for effective network management: it allows to constantly observe the network's behavior to ensure it is working as intended, and can trigger both automated and manual remediation procedures in case of failures and anomalies. The concept of Software-Defined Networking (SDN) decouples the control logic from legacy network infrastructure to perform centralized control on multiple switches in the network, and in this context the responsibility of switches is only to forward packets according to the flow control instructions provided by controller. However, as most widely adopted SDN switches (e.g. OpenFlow-based) only expose simple per-port and per-flow counters, the controller has to do almost all the processing to determine the network state, which causes significant communication overhead and excessive latency for monitoring purposes. The absence of programmability in the data plane of SDN prompted the advent of programmable switches, which allow developers to customize the data-plane pipeline (e.g. match-action table) and implement novel programs operating at wire speed directly in the switches. This means that we can offload certain monitoring tasks to programmable data planes, to perform fine-grained monitoring even at very high packet processing speeds.

Given the central importance of network monitoring exploiting programmable data planes, the principal goal of this thesis is to enable a wide range of monitoring tasks in programmable switches, with a specific focus on the ones equipped with programmable ASICs (Application-Specific Integrated Circuits). Indeed, most of the network monitoring solutions available in literature do not take computational and memory constraints of programmable switches into due account, preventing, de facto, their successful implementation in real commodity switches.

This claims that network monitoring tasks can also be executed in programmable switches. To achieve this goal, this thesis makes three main contributions: *(i.)* We enhance P4-enabled data plane programmability for network monitoring; *(ii.)* We design and develop several network monitoring tasks in programmable data planes;

(iii.) We combine multiple tasks in a single commodity switch to collect various metrics for different monitoring purposes.

In terms of the first contribution, we propose new algorithms to approximate the arithmetic operations in the programmable switches, which enhances the data plane programmability for network monitoring. They can be used as additional building blocks to further implemented monitoring tasks described in the second contribution. When it comes to the second contribution, we studied and developed five different kinds of monitoring tasks for programmable data planes: heavy-hitter detection to detect heavy flows with large packet counts, flow cardinality estimation to estimate the number of distinct flows in millions of packets, network traffic entropy estimation to track the flow distribution, total traffic volume estimation to know how many packets are in the network, and volumetric DDoS detection to detect potential volumetric DDoS attacks according to the change of normalized entropy and per-source flow cardinality to the same destination host. The aim of this contribution is to offload as much as possible these network monitoring functionalities into the switch, that is, in the best case, the task can be executed entirely in programmable data planes thus performing in-network monitoring. Focusing on the third contribution, we revisited our designed tasks in the second contribution, and proposed a new way to combine them into a single real hardware switch: we only use the programmable switch to store flow and packet statistics, and the responsibility of the controller is to compute and track different monitoring metrics. In this way, the five tasks mentioned in the second contribution can work together in a single programmable switch to perform high speed monitoring, while the controller can guarantee high accuracy on the estimation of monitoring metrics to diagnose performance and security issues.

Our evaluations have shown that the contributions in this thesis could be used by network administrators, network operators, as well as network security engineers, to better understand the network status depending on different monitoring metrics, and thus prevent network infrastructure and service outages.

Contents

1 Introduction

1.1 Network monitoring in Software-Defined networks

Network monitoring is of primary importance: it is the main enabler of various network management and security tasks, ranging from accounting [56][58] to traffic engineering [62][38], anomaly detection [83], Distributed Denial-of-Service (DDoS) detection [121], Super-spreader detection [91], and scans detection [128][118], among others. With the advent of Software-Defined Networking (SDN), the significance of network monitoring has been certainly increased. This is because SDN, with the idea of a (logically) centralized control, allows an easier coupling of network management operations with the observed network status. As a result, SDN has been seen as the answer to many of the limitations of legacy network infrastructures [77][119][68]. However, such a noble intent has been limited by SDN's current predominant incarnation, the OpenFlow (OF) protocol. Indeed, current OpenFlow APIs are ill-suited for monitoring large network data streams and cannot provide accurate data-plane measurements: the main mechanism exposes simple per-port and per-flow counters available in the switches [97]. An application running on top of the controller can periodically poll each counter using the standard OF APIs and then react accordingly, instantiating the appropriate rule changes. As a consequence, OF suffers from two important limitations: *(i)* the controller has to monitor all flows in the network and *(ii)* as the data plane exposes just simple counters, the controller needs to do all the processing to determine the network state. This also causes that OF-enabled devices are only able to collect raw flow statistics to be sent to a monitoring collector, causing significant communication overhead for monitoring purposes. This limitation is well-know for legacy devices as well (e.g. SNMP- and sFlow-supporting equipment) [117]. Another limitation of legacy devices is large detection latency (the monitoring interval is mostly greater than 1 minute [9]), which imposes strict time limits on the analysis done by the collector.

Lately, the advent of the so-called *programmable switches* (e.g. P4-enabled switches [42]) has introduced the possibility to program data plane with advanced

functionality and enabled the possibility to implement more refined monitoring solutions directly in the switch hardware while performing line-speed packet processing (in the order of nanoseconds). Such an innovative technology has attracted a growing number of researchers and practitioners that in turn have proposed many different solutions to enhance SDN capabilities in the context of network monitoring [93][89][132][74][113]. As a result, the prospect of realizing fine-grained network-wide monitoring, by analyzing the exposed information from all the switches in a network, has attracted a lot of interests [132][36][69]. For instance, memory-efficient data structures, such as *sketches* [131][48], have been proven to be implementable in programmable switches to reduce redundant monitoring information.

1.2 Challenges in network monitoring via programmable data planes

There are many challenges when network monitoring comes to programmable data planes, including but not limited to: *(i.)* limited computational and memory resources on-board programmable switches; *(ii.)* how to design and implement the monitoring tasks that fit the programmable data plane switches; *(iii.)* how to arrange limited resource for a wide range of monitoring tasks in the programmable switch to maximize the monitoring visibility.

- **Challenge 1: overcoming limited programmability and resources of programmable switches.** To assure line-rate packet processing, the domain-specific languages, such as P4 [42] and POF [88], truncates many common arithmetic and logical instructions in well known programming language (e.g. C/C++). For instance, the loops (i.e. *For* and *While*) are not allowed to use. This is because if the switch uses iterations to process the same packet, the following incoming packets have to wait until the this packet has been processed. This will significantly delay the packet processing in the switch. For a similar reason, in the hardware switch, such as ASIC (Application-specific integrated circuit), the same metadata (i.e. the fields in the packet header) cannot be executed more than one time of computation. Moreover, the programmable switches have only few tens of MB, which means that the memory efficiency should be taken into consideration. In summary, to enable new functionalities in the programmable switches, a good design considering both computational and memory limitations is necessary.

- **Challenge 2: design and development of monitoring solutions in programmable data planes.** A wide range of network monitoring tasks can be executed in the monitoring servers, but not all of them can be offloaded to the programmable switches. To enjoy the benefits of programmable switches, the first step is to understand which tasks are possible and useful to be developed in the switch. Afterwards, we need to investigate which compact data

structures (e.g. sketches) or actions (e.g. sample the packets) are suitable for specific monitoring tasks. Finally, we should not ignore the resource limitations of programmable switch described in the last section and make the tasks executable in data plane.

- **Challenge 3: combination of monitoring tasks.** Due to high cost of programmable switches (e.g., a Tofino switch costs more than 10,000 dollars) and increasing throughput requirements of modern networks, if we can enable several tasks together in a switch and deploy this switch in an appropriate position of the network, this will significantly reduce the required hardware-upgrade budget. Thus, another challenge is how to combine multiple tasks in the programmable switch to perform high-speed monitoring. In this case, we need to consider not only the resource limitation in the switch but also the resource allocation for different tasks.

1.3 Goals, Research Questions and Approaches

On the basis of the challenges listed in Section 1.2, we define the first research goal of this thesis as follows:

> ***Goal 1**: To investigate network monitoring and programmable data plane in Software-Defined Networks, learn how to program switches and enhance their P4-enabled data plane programmability for monitoring.*

We also point out that to develop monitoring tasks in programmable switches, we need to know their different requirements. For this reason we define a second research goal of this thesis as follows:

> ***Goal 2**: To study network monitoring tasks and how to offload them in programmable data planes.*

Finally, measuring different metrics to perform various monitoring together in a single switch is an interesting topic to discuss. This leads us to the final research goal of this thesis:

> ***Goal 3**: To combine studied monitoring tasks in suitable network scenarios composed of programmable switches.*

1.3.1 Goal 1: Enhancement of data plane programmability

Research questions

In the first goal we expressed that we want to investigate the state of the art of network monitoring. This leads to our first research question:

> ***RQ1**: Why network monitoring is so important in modern telecommunication networks? In particular, what are the benefits to implement it in the data plane?*

We answer RQ 1 in all chapters of this thesis.

Upon we understand the importance of network monitoring in the data plane, the next thing to know is how to develop the interesting tasks into programmable switch. This leads to our second research question:

> ***RQ2**: How does a programmable data plane work? How is the data plane of a switch programmed? What are the limitations of this type of functionality?*

We address RQ 2 in Chapter 2.

To implement the tasks for monitoring requires specific operations, which are missing in programmable data plane switches. This brings us to the third research question:

> ***RQ3**: Is there any way to improve P4-enabled data plane programmablity? If yes, how?*

We address RQ 3 in Chapter 3.

Approach

To address the research questions above, we investigated several typical monitoring tasks, including heavy-hitter detection, flow cardinality estimation, network traffic entropy estimation, and volumetric DDoS detection. The outcome of those tasks can be further used to diagnose the network performance and security issues. We then studied the domain-specific language P4 for programmable switches and learned the constraints for implementation. To overcome those constraints, we propose some approximation methods (e.g. logarithm and expoential function estimation) as building blocks to support further implemented monitoring tasks.

1.3.2 Goal 2: Network monitoring in programmable data planes

The second goal of this thesis is to study diverse monitoring solutions in the context of executing them on a programmable switch, which includes their usages for network management and security as well as factors that drive their use. We define the first research question towards meeting this goal as:

Research questions

> ***RQ4**: What is heavy-hitter detection? Why do we need to track heavy flows? How to detect them by using the programmable switch? What is the performance in a network composed of partially deployed programmable switches?*

We address RQ 4 in Chapter 4.

After studying the heavy flow identification, our next research question focuses on how to know number of distinct flows in millions of packets:

> ***RQ5**: What is flow cardinality estimation? Why is it necessary for monitoring? How do we design new idea for flow cardinality estimation on a programmable switch? What is the performance with respect to the state-of-the-art?*

We address RQ 5 in Chapter 5.

We then switch to another monitoring task: network traffic entropy estimation, and the research question is:

> ***RQ6**: What is network traffic entropy estimation? Which metric of network does network traffic entropy indicate? Any problems while implementing it in the data plane of programmable switch?*

We address RQ 6 in Chapter 6.

Afterwards, we address another practical problem named *network-wide total traffic volume estimation*:

> ***RQ7**: Why do we need to know network-wide total traffic volume estimation? What is the key problem to coordinate multiple programmable switches for packet counting in the network? How do we solve it?*

We address RQ 7 in Chapter 7.

Finally, we study volumetric DDoS detection by using some common metrics that can be collected and analyzed from the packet header:

> ***RQ8**: What is volumetric DDoS attack? What are the detection methods? What is the difference between different methods? How to design and implement them in programmable switches?*

We address RQ 8 in Chapter 8 and Chapter 9.

Approach

As the first goal is to understand the importance of monitoring, our approach to answering the research questions for the second goal is to design and develop practical monitoring solutions in programmable data planes. We started with simulations to understand the behaviors of multiple network monitoring tasks. We then designed some interesting monitoring ideas and implemented them in P4. Afterwards, the P4 program is compiled and installed in the simulated switches. We then conducted experiments with simulated switches in an emulated environment (i.e., mininet [8]) to better understand the workflow of programmable switches. Finally, considering more strict hardware constraints, we implemented some feasible monitoring solutions in the programmable switches equipped with ASIC, proving that our proposed approaches can be deployed in real network scenarios.

1.3.3 Goal 3: Combination of tasks

The final goal of this thesis is to study potential problems with the combination of multiple monitoring tasks in a single programmable switch.

Research questions

> ***RQ9**: How to coordinate multiple monitoring tasks in a single programmable hardware switch while overcoming the resource limitations?*

We address RQ 9 in Chapter 10.

Approach

Our approach to answering the last research question is in line with the previous approaches. We first deeply investigated the resource usages of multiple monitoring tasks, and we then realized that executing all monitoring tasks entirely in programmable switches is not feasible. Thus, we only use compact data structures in programmable switches to store flow and packet statistics. More complicated operations, such as complex computations, are executed in SDN controller or in the control plane of the programmable switch. This allows the switches to report summarized flow and packet statistics instead of raw data to controller while processing the packets at line rate. On the other side, controller can perform very accurate estimations on different monitoring tasks, meanwhile it consumes a small amount of communication overhead with switches.

1.4 Organization and Key Contributions

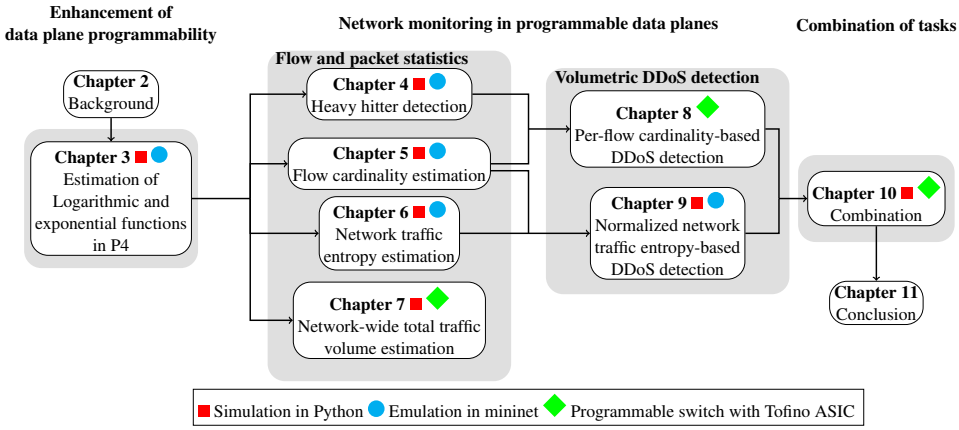


Figure 1.1: The scheme of thesis

Figure 1.1 shows a schematic structure of this thesis. The scheme shows the relation between chapters, as well as how chapters compose distinct parts of the thesis. Moreover, the scheme also specifies how the proposed approaches have been tested, including simulations in Python, emulations in mininet, and experiments in programmable commodity switches. In the following, we provide a summary for each chapter with corresponding publications.

Chapter 2: Background on network monitoring in programmable data planes

In this chapter we provide background information on network monitoring in programmable data planes. We will show a brief background on the SDN and explain the advantages of programmable data planes for today's network. Then we will outline how the packets are processed in programmable data planes together with P4. Afterwards, a sequence of network monitoring tasks are introduced.

Chapter 3: Estimation of logarithmic and exponential functions in P4

In this chapter we will take the first step to investigate P4 and report its limitations. We then show how we overcome those limitations to estimate logarithms and exponential functions with a given precision by only using P4-supported arithmetic operations. This enhancement helps several monitoring tasks to implement entirely in programmable data planes.

This chapter is based on the part of following peer-reviewed papers:

- **Damu Ding**, Marco Savi, and Domenico Siracusa. *Estimating logarithmic and exponential functions to track network traffic entropy in P4.* IEEE/IFIP Network Operations and Management Symposium (NOMS) 2020.
- **Damu Ding**, Marco Savi, and Domenico Siracusa. *Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4* submitted to IEEE Transactions on Dependable and Secure Computing (TDSC).

Chapter 4: Network-wide heavy-hitter detection robust to partial deployment

This chapter discusses why the detection of heavy hitter is necessary in the network. Furthermore, we present a network-wide heavy-hitter detection strategy for identifying global heavy flows in the network composed by fully deployed programmable switches. To optimize our network-wide heavy-hitter detection performance in case of partial deployment of prorgammable switches, we also propose an incremental deployment algorithm to maximize the flow visibility in the network.

This chapter is based on the following peer-reviewed publications:

- **Damu Ding**, Marco Savi, Gianni Antichi, and Domenico Siracusa. *Incremental deployment of programmable switches for network-wide heavy-hitter detection.* IEEE Conference on Network Softwarization (NetSoft) 2019.
- **Damu Ding**, Marco Savi, Gianni Antichi, and Domenico Siracusa. *An incrementally-deployable P4-enabled architecture for network-wide heavy-hitter detection.* IEEE Transactions on Network and Service Management 17.1 (2020): 75-88.

Chapter 5: Flow cardinality estimation

In Chapter 5, we design a new flow cardinality estimator of a large packet stream in programmable data planes. Such an estimation can be used to monitor the number of active connections in a link and distinct flows in the network. The monitored results can be further used to diagnose network security problems, such as DDoS attacks and port scans.

This chapter is based on part of the following paper under review:

- **Damu Ding**, Marco Savi, and Domenico Siracusa. *Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4* submitted to IEEE Transactions on Dependable and Secure Computing (TDSC).

Chapter 6: Network traffic entropy estimation

Chapter 6 focuses on *Entropy*: the most widely-used metric to evaluate traffic distribution, which is an important indicator to understand the network behavior. This

metric can be used to support a wide range of tasks including congestion control, load balancing, port-scan detection, distributed denial-of-service (DDoS) attacks detection, and worm detection.

This chapter is based on the following peer-reviewed publication:

- **Damu Ding**, Marco Savi, and Domenico Siracusa. *Estimating logarithmic and exponential functions to track network traffic entropy in P4*. IEEE/IFIP Network Operations and Management Symposium (NOMS) 2020.

Chapter 7: Network-wide total traffic volume estimation

In this chapter we aim to solve a fundamental problem arising when exploiting programmable data planes for network-wide monitoring: how to estimate the overall number of packets in the network (i.e., the traffic volume) while avoiding packet double counting. Its correct estimation is necessary to support a broad range of monitoring tasks, such as configuring the detection threshold of network-wide heavy hitters. If the traffic volume is computed by summing the packet counters as recorded by all the switches (i.e., the double counting problem is not considered), the result will be largely overestimated and so affect the network monitoring performance.

This chapter is based on the following peer-reviewed publication:

- **Damu Ding**, Marco Savi, Federico Pederzolli, and Domenico Siracusa. *INVEST: Flow-based Traffic Volume Estimation in Data-plane Programmable Networks* IFIP Networking Conference 2021.

Chapter 8: Per-flow cardinality-based DDoS detection

In this chapter, we first introduce the BACON Sketch, a memory-efficient data structure to estimate per-destination flow cardinality, and theoretically analyze its error bounds. Then we propose a simple in-network DDoS victim identification strategy relying on BACON Sketch to detect the destination hosts for which the number of incoming connections exceeds a pre-defined threshold. We successfully implement our DDoS detection idea in a programmable switch equipped with on a Tofino ASIC while overcoming the limitations imposed by real hardware.

This chapter is based on the following paper under review:

- **Damu Ding**, Marco Savi, Federico Pederzolli, and Domenico Siracusa. *In-Network Volumetric DDoS Victim Identification Using Programmable Commodity Switches* submitted to IEEE Transactions on Network and Service Management (TNSM).

Chapter 9: Normalized network traffic entropy-based DDoS detection

In this chapter, we combined the achievements in Chapter 5 for flow cardinality estimation and Chapter 6 for network traffic entropy estimation to track the normalized entropy of distinct destination IPs and then to address a more practical security issue in the network: volumetric DDoS detection. During a DDoS attack, the normalized entropy of distinct destination IPs observed in the network significantly decreases comparing to previous observations windows. Based on this observation, we propose a new normalized entropy-based DDoS detection approach and implement it entirely in programmable data plane.

This chapter is based on the following paper under review:

- **Damu Ding**, Marco Savi, and Domenico Siracusa. *Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4* submitted to IEEE Transactions on Dependable and Secure Computing (TDSC).

Chapter 10: Combination

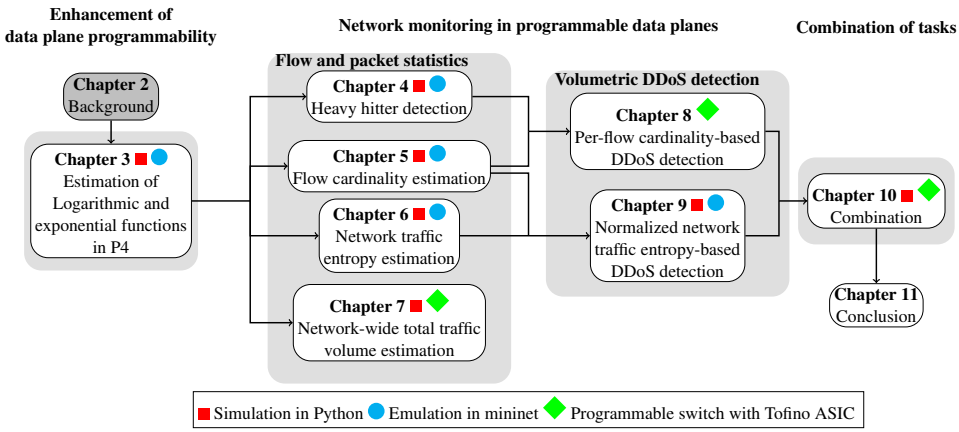
This chapter is entirely new. Unlike some of previous chapters the experiments are designed and conducted in emulated environment, we combined and implemented several monitoring tasks in ASIC switching. The motivation of this chapter is to demonstrate that our ideas are practical and can be deployed in real industrial network scenarios.

Chapter 11: Conclusion

After studying all previous chapters, we conclude this thesis in the final chapter. In addition, we outline some possible directions for future work.

2

Background on network monitoring



The purpose of this chapter is to give the reader basic background information on programmable data planes within the Software-Defined Networks landscape. We will start by providing a brief background on SDN and explain why the advent of programmable data planes is necessary for today's networks. Then we will outline how packets are processed in programmable data planes. Afterwards, we will address the high-level domain-specific language, named P4, for programmable switches and report its workflow. Specifically, we provide an introduction to different types of network monitoring tasks that can be executed in programmable switches. Finally, we report the experimental environment used for the evaluation of our developed monitoring tasks.

2.1 Software-Defined Networks

SDN architectures [66] are based on the fundamental idea of data and control plane separation. Particularly, the SDN architecture can be modeled as three layers: application layer, controller layer, and network hardware layer. The layers

are connected by two types of interfaces: Southbound interfaces allow the communications between the controller layer and the network hardware layer. In the meanwhile, Northbound interfaces permit the engineers to develop applications to control the high-policy and network.

At the top of the SDN stack lies the application layer, which includes all the applications that exploit the services provided by the controller in order to perform network-related tasks, like load balancing, network virtualization etc. One of the most important features of SDN is the portability, this provides to third-party developers via the abstractions and defines for the easy development and deployment of new applications in various networked environments from data centers and WANs to wireless and cellular networks.

Due to well-defined Northbound APIs, Intent framework in SDN allows an application to request a new service without knowing how the service is applied. This enables the network manager to program the network at a high level, they just need to specify their intent: a policy statement or connectivity requirement.

Moving to the next layer we can observe the controller layer. SDN controller provides services that can realize a distributed control plane, as well as abet the concepts of state management and centralization. A network operating system offers a more general abstraction of network state in switches, revealing a simplified interface for controlling the network. This abstraction assumes a logically centralized control model, in which the applications view the network as a single system. In other words, the network operating system acts as an intermediate layer responsible for maintaining a consistent view of network state, which is then exploited by control logic to provide various networking services for topological discovery, routing, management of mobility and statistics etc. Therefore, SDN controller plays a key role in SDN architecture. Nowadays, various types of SDN controllers have been researched in literature, such as ONOS [11], ODL [12], and RYU [27].

The southbound protocols is very important for the manipulation of the behaviour of network hardware by the controller. It is the way that SDN attempts to program the network. There are several southbound protocols which can be adopted at this moment, such as OpenFlow and Netconf

At the bottom of the SDN architecture is the network hardware. Prior to 2014, switches in the SDN are often represented as basic forwarding hardware accessible via an open interface, as the control logic and algorithms are offloaded to a controller. The recent advent of programmable data plane switches and P4 [42] allows developers to execute part of the network monitoring/security operations directly in their data plane pipeline and deliver to the centralized monitoring/control plane information that is partially or fully processed. The reason is that the programmable switches can achieve several orders of magnitude higher throughput and lower latency compared with highly optimized software solutions [79].

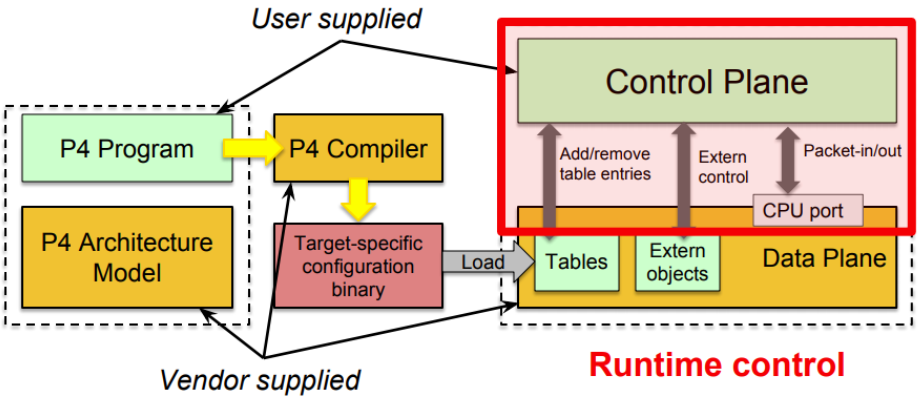


Figure 2.1: The scheme of P4Runtime

2.1.1 Programmable data plane switches

Current trends in SDN have extended the network programmability to the data plane through programmable switching ASICs (Application-Specific Integrated Circuits) and domain-specific languages (e.g., P4 [42]). In programmable switches, packets are processed sequentially in a pipeline, including registers for the store and count of packets, match-action table for data plane customization, and stateful ALUs for computation. Match-action tables match certain packet fields or metadata and apply actions on the packet. Each table modifies packet fields and generates metadata through which tables can share information. Using programmable switches, programmers can customize data plane logic with domain-specific languages like P4.

2.1.2 P4 language

P4 [42] is a high-level domain-specific language designed for programmable data planes. In a P4 program, developers can customize packet headers, build packet processing graphs, and specify entries in match-action tables. The compiler provided by switch vendors can compile the programs into binaries and generate interactive APIs. The compiled binaries specify data plane contexts, and then they are loaded into switches Finally, the generated APIs are used by control plane applications to communicate with the data plane.

2.1.3 P4Runtime

Control planes manage the runtime behavior of P4 targets via data plane APIs. Alternative terms are control plane APIs and runtime APIs. The data plane API is provided by a device driver or an equivalent software component. It exposes data plane features to the control plane in a well-defined way. Figure 2.1 shows

the main control plane operations: data plane APIs facilitate runtime control of P4 entities (Match-action tables and externs). They typically also comprise a packet I/O mechanism to stream packets to/from the control plane. They also include reconfiguration mechanisms to load P4 programs onto the P4 target. Control planes can control data planes only through data plane APIs, that is, if a data plane feature is not exposed via a corresponding API, it cannot be used by the control plane.

2.1.4 Controller use case patterns

Embedded/Local Controllers

Embedded/Local Controllers are implemented by the device driver and are executed on the local CPU of the device that applies the programmable data plane. The APIs are usually presented as a set of C function calls just like for other devices that operating system are accessing.

Remote Controllers

The controllers at remote add the ability to invoke API calls from a separate system. This increases system stability and modularity, and is essential for SDN and any system with centralized control. Remote control APIs follow the base methodology of remote procedure calls (RPCs) but rely on modern message-based frameworks that allow asynchronous communication and concurrent calls to the API, such as Thrift [1] or gRPC [5]. For instance, gRPC uses HTTP/2 for transport and includes many functionalities ranging from access authentication, streaming, and flow control. The data structures, services, and serialization schemes of protocols are described by using protocol buffers (protobuf) [4].

2.2 Network monitoring tasks in programmable data planes

In this section, we briefly present various types of monitoring tasks and report their practical usages.

2.2.1 Heavy hitter detection

Heavy hitters are identified as the flows that carry more than a fraction of the overall packets in the network. Alternatively, the heavy hitters can be the top k flows by size (the “top-k” problem). Many network management applications can benefit from finding the set of heavy flows contributing significant amounts of traffic to a link: for example, to relieve link congestion [39], to plan network capacity [61], or to detect network anomalies and attacks [82].

2.2.2 Flow cardinality estimation

Flow cardinality estimation is the task of determining the number of distinct flows in a stream of packets [137]. In the domain of online traffic monitoring of high-speed networks, the cardinality estimation can be used to detect traffic anomalies, such as network IP/port scan and distributed denial-of-service (DDoS) attacks. Moreover, such an estimation can also be used to monitoring the number of active connections in a link. However, as pointed out in [71], cardinality estimation over large data sets presents a challenge in terms of computational resources and memory. A non-negligible fraction of packets in the network may not be computed since they exceeded the available memory.

2.2.3 Network traffic entropy estimation

Network traffic entropy is a metric that gives an indication of the traffic distribution across the network [55]. Relying on the definition of *Shannon entropy* [110], the traffic entropy reaches 0 when all packets belong to the same flow, while it reaches its maximum value when each flow carries the same number of packets. The knowledge of the flow distribution helps diagnose performance and countermeasures the security issues of network, including congestion control [78], load balancing [125], port-scan detection [67][40], distributed denial-of-service (DDoS) attacks detection [85][94] and worm detection [120].

2.2.4 Total traffic volume estimation

The ability to precisely estimate the total traffic volume [36] (i.e., number of distinct packets flowing in the network), and the related number of distinct flows and average flow size (i.e., average number of packets per flow) is necessary to support a broad range of monitoring tasks. In general, whenever a metric requires to set the network-wide threshold for a monitoring task, then an accurate estimation of the total traffic volume is of paramount importance. For instance, the threshold of heavy hitter detection is usually set as a given fraction of total traffic volume.

2.2.5 Volumetric DDoS detection

A volumetric DDoS attack can be identified according to many different metrics, such as looking for a significant decrease of the normalized entropy in distinct destination IP addresses observed in the network [65][80][122], or a large number of distinct flows (sequences of packets with the same source IPs) contacting a specific destination host (i.e., per-destination flow cardinality) [134][93][74]. Note that entropy-based DDoS detection can only detect DDoS attacks, but flow cardinality-based DDoS detection is also able to identify the DDoS victims, which allows operators to mitigate the impact on targeted nodes as soon as an attack is detected.

2.3 Experimental environment

In this section, we describe the environments that we used in this thesis to evaluate the performance of our work.

2.3.1 Behavioral model and emulated environment

The P4 behavioral model, named *Bmv2* [13], is a simulator used to describe the behavior of P4 switches, including parsers, tables, actions, ingress and egress in the P4 pipeline. The P4 code in Bmv2 simulator can be easily deployed in a server, which acts as a software switch. In case of emulation, Mininet [8] is a good choice for the deployment of P4 switches implementing the network monitoring tasks. The P4 code is compiled by p4c compiler [19] into a JSON file that describes the behavior of P4 switch. The JSON file is then loaded by any P4 switch created according to the behavioral model. Finally, a topology in Mininet is created connecting such behavior-defined P4 switches.

2.3.2 ASIC switching

Recently, programmable ASICs (Application-Specific Integrated Circuits) have been introduced: apart from standard data plane features (i.e., high-speed switching and forwarding), they offer the possibility to customize functionalities, if properly programmed through domain-specific programming languages like P4 [42]. For instance, programmable switches equipped with the Tofino ASIC [2] can always forward packets at line-rate once the P4 program (including innovative features) is compiled and installed in the switches. For the other chips, like Network Interface Cards (NICs), Field Programmable Gate Arrays (FPGAs) and Network Processing Units (NPUs), they cannot perform high-throughput and low-latency at the same level as ASICs [136].

Tofino ASIC architecture

Figure 2.2 shows the architecture of programmable commodity switches with Tofino ASIC. There are multiple *pipes* (each composed by an *ingress* and *egress pipeline*) in the switch, and several ports are associated with one ingress or egress pipeline. Programs executed in different pipes are independent in terms of memory and computational resources. For instance, register values in pipe 1 cannot be read by programs running in pipe 2. When packets enter the switch, they are first processed by the input port's ingress pipeline, then, after crossing the switching matrix, are processed again by the output port's egress pipeline. Each pipeline contains a limited number of *stages*, each including one or more code *blocks* (i.e., sets of operations) that are applied to each packet in sequence. Boundaries on the number of operations executed within each stage exist to ensure that the ASIC can process packets at line rate, irrespective of the custom logic being implemented. Blocks can contain

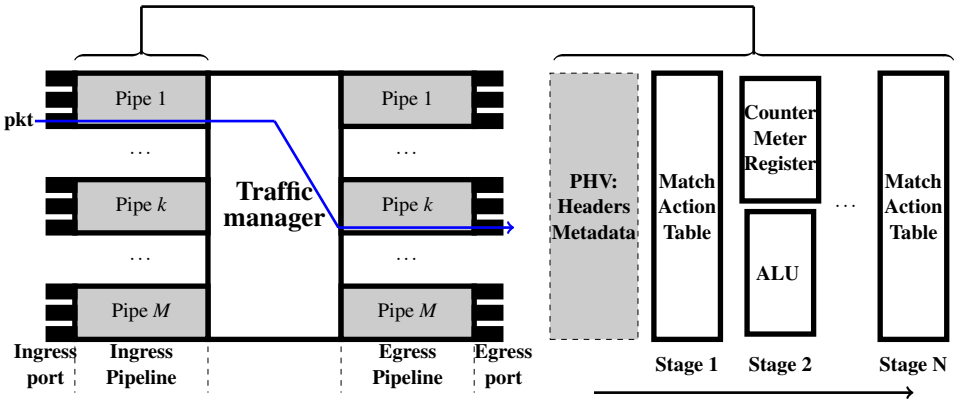


Figure 2.2: Tofino-based switch data plane architecture

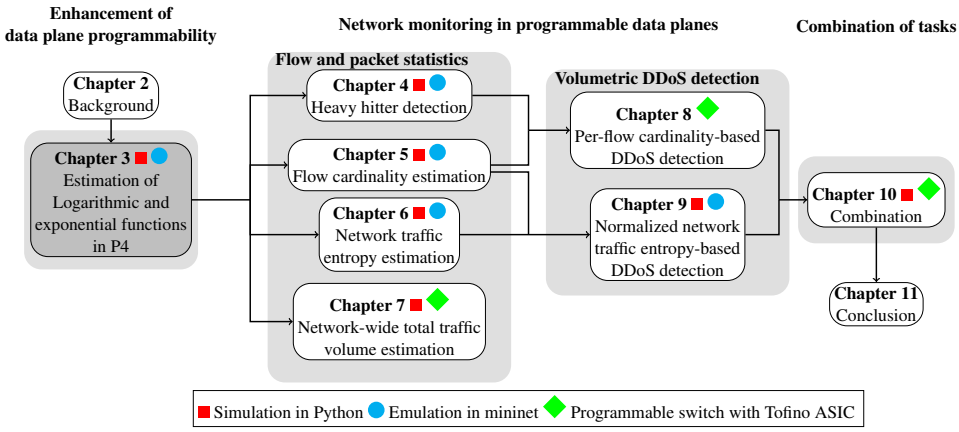
operations including *(i.)* applying match-action tables for customized packet processing, *(ii.)* reading or writing counters/meters/registers for counting or storing packets, and *(iii.)* calls to Arithmetic Logic Units (ALUs) for local computations. Note that each stage has limited hardware resources, such as memory size (Static and Ternary Random Access Memory, i.e., SRAM and TCAM) and number of ALUs.

2.3.3 Concluding remark

Upon understanding the background on network monitoring in programmable data planes, this thesis puts forth the claim that the barriers between the development of monitoring tasks and the constraints of programmable switches can be effectively overcome with the support of properly designed novel network monitoring strategies. To achieve this goal, we started with simulations to understand different monitoring behaviors and propose new ideas. Then we implemented our new ideas with a domain-specific language for programmable switches and tested them in an emulated environment (i.e. mininet). Finally, we migrated our work into a programmable switch equipped with Tofino ASIC in our physical testbed.

3

Estimation of Logarithmic and Exponential functions in P4



In this chapter we take the first steps towards the enhancement of P4. We address some limitations on P4 language and provide two new estimation algorithms for logarithm and exponential-function with a given precision by only using P4-supported arithmetic operations, which can be used to design various of new network monitoring tasks in programmable data planes.

This chapter is based on our previously published paper "Damu Ding, Marco Savi, and Domenico Siracusa. Estimating logarithmic and exponential functions to track network traffic entropy in P4. NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium. IEEE, 2020." [55] and paper "Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4" submitted to IEEE Transactions on Dependable and Secure Computing (Under review).

3.1 Introduction

In order to assure line rate packet processing, the domain-specific language for programmable data planes, namely P4, removes many arithmetic and logical operations that may delay the packet processing, including loops (e.g. *For* and *While*), division, logarithm, exponentiation and floating numbers. However, many network monitoring tasks rely on those operations, and without them, the switches have to report all raw data to the controller, so that it can perform such computations, which incurs large communication overhead and delay. A straightforward solution to overcome these limitations could be executing all arithmetic operations unsupported by the P4 language at SDN controller, but this would almost nullify the benefits of the proposed solution as discussed so far. If the necessary operations can be approximated in P4, the tasks are implementable in the switches. This means that they can summarize the network statistics directly in the switch, and then only report the filtered information to controller.

In this chapter we take the first steps toward this goal. We start by theoretical analysis to approximate logarithmic and exponential functions by only using P4-supported operations. We then successfully implemented them in P4 (namely P4Log and P4Exp), and they can be used to enable the execution of monitoring tasks requiring the computation of such functions.

In this chapter we focus on answering the following questions:

- How do we overcome the limitations to estimate logarithmic and exponential functions in P4?
- Which monitoring tasks need our estimations?

3.2 Basic knowledge

3.2.1 Hamming weight computation for logarithmic estimation

Hamming weight represents the number of non-zero values in a string. In a binary string, the Hamming weight indicates the overall number of ones. For example, given the binary string 01101, the Hamming weight is 3. Hamming weight can be computed by means of different algorithms: as part of P4Log, in this chapter we adopt the *Counting 1-Bits* algorithm presented in [123], as it only relies on bitwise operations that are completely supported by P4 language [17].

3.2.2 Binomial series expansion of exponential function 2^x

The proposed P4Exp algorithm relies on binomial series expansion [104] of 2^x, where x is a real positive number. In general, the binomial series expansion of

Table 3.1: Operations supported by P4 and their symbols [17]

Symbol	Operation
$+$	Addition
$-$	Subtraction
$\cdot$	Multiplication
$\gg$	Logical right shift (Division by power of two)
$\ll$	Logical left shift (Multiplication by power of two)
$\hat{}$	Bitwise XOR
$\vert$	Bitwise inclusive OR
&	Bitwise inclusive AND
$\sim$	Bitwise COMPLEMENT

$(1+\alpha)^x$ is defined in the following way:

$$(1+\alpha)^x = \sum_{k=0}^{+\infty} \binom{x}{k} \alpha^k$$

When $\alpha = 1$ we have the binomial series expansion of 2^x:

$$2^x = \sum_{k=0}^{+\infty} \binom{x}{k} = \underbrace{1 + x + \frac{x(x-1)}{2!} + \cdots}_{N_{terms}}$$

With $\alpha = 1$, the series converges absolutely iff $\mathrm{Re}(x) > 0$ or $x = 0$. In our case this always holds, since x is a real positive number. In P4Exp we will rely on a truncation of the binomial series to the first N_{terms} terms.

3.3 Estimation of Log and Exp Functions in P4

In this section, we propose *P4Log* and *P4Exp*, two new algorithms for the estimation of logarithm and exponential function that leverage only arithmetic and logical operations supported by the P4 language (see Table 3.1) and can be executed entirely in the data plane. The P4 code of two algorithms is open sourced in [23].

3.3.1 P4Log algorithm

Given an L-bit integer x and a logarithmic base d, the goal is to estimate $\log_d x$. Since operations on floating numbers are forbidden in P4, our algorithm computes $\log_d x$ amplified by 2^{10} times (i.e, $\log_d x \cdot 2^{10}$). This amplification (similar to what is done in [85]) is performed to deal with integer numbers without loosing accuracy on decimal parts, and can be done using the left shift operator ($\ll$). This is needed because P4, in the case of operations resulting in floating numbers, truncates the resulting value to its integer part: without any amplification our algorithm (as any other algorithm dealing with floating numbers) would result in a very bad estimation accuracy. Additionally, we will show that an amplification of 2^{10} times is enough to achieve very good accuracy on the estimation (see Section 3.4).

Algorithm 1: P4Log algorithm

Input: An L-bit integer x ($L \in \{16, 32, 64, 128\}$) and a given logarithmic base d
Output: An L-bit integer estimation of $\log_d x$ amplified 2^{10} times ($\log_d x \cdot 2^{10} \equiv \log_d x \ll 10$)

1 **Function** $log_2ES(x)$**:**
2 $w \leftarrow x|(x \gg 1)$
3 **for** int $i \in \{1, \ldots, \log_2 L - 1\}$ **do**
4 $w \leftarrow w|(w \gg 2^i)$
5 $b \leftarrow HammingWeight(w)$
6 $n \leftarrow b - 1$
7 $\log_2 x \ll 10 \leftarrow n \ll 10 + \underbrace{\log_2(1 + \frac{\bar{x} - 2^{N_{bits}}}{2^{N_{bits}}}) \ll 10}_{\text{Tree search } \rightarrow N_{digits}}$
8 **return** $\log_2 x \ll 10$
9 **Function** $log_dES(x, \log_d 2 \ll 10)$**:**
10 $\log_d x \ll 10 \leftarrow (log_2ES(x) \cdot \log_d 2 \ll 10) \gg 10$
11 **return** $\log_d x \ll 10$

To compute $\log_d x$, we can write it as $\log_d x = \log_2 x \cdot \log_d 2$. Since d is known, $\log_d 2$ is a constant value that can be pre-computed and loaded in the P4 program. Thus, logarithm estimation in P4 language reduces to the estimation of $\log_2 x$: if we can estimate $\log_2 x$, then it is always possible to estimate the logarithm of an input value x with any given base d, as far as the constant value $\log_d 2$ has been stored in the P4 program as a constant.

As shown in Algorithm 1, the algorithm first estimates $\log_2 x$ ($log_2ES(x)$ function). Initially, it computes the integer part of $\log_2 x$, which is equal to the index of leftmost 1 of x when expressed in binary notation. To get this information, all bits at the right of leftmost 1 of x are iteratively converted to 1 and the result is stored in a binary string named w (Lines 2 - 4). For instance, given a binary value 010010, the resulted w is 011111. This operation is needed because, in P4, numbers are always handled in decimal notation, while we need a binary string as input of the next step. Then, the Hamming weight of w (see Section 3.2.1) is retrieved, indicating the number of bits from the leftmost 1 (including itself) and denoted by b (Line 5). Hence, the index of the leftmost 1, called n and equal to $b - 1$, stores the integer part of $\log_2 x$ (Line 6).

The algorithm then estimates the decimal part. Note that $\log_2 x = n + \log_2(1 + \frac{x - 2^n}{2^n})$, meaning that the estimation of the decimal part reduces to the estimation of $\log_2(1 + \frac{x-2^n}{2^n})$. We adopt the first N_{bits} bits starting from the leftmost 1 to estimate it, using a set of pre-computed decimal values stored in the P4 program as constants and rounded to a float with N_{digits} digits of precision. If N_{bits} bits are used to estimate the decimal part, it means that $2^{N_{bits}}$ constants need to be pre-computed and stored in the program. We call $\bar{x}$ the binary sub-string used to estimate the decimal part. For example, considering $N_{bits} = 2$, there are four possible cases:

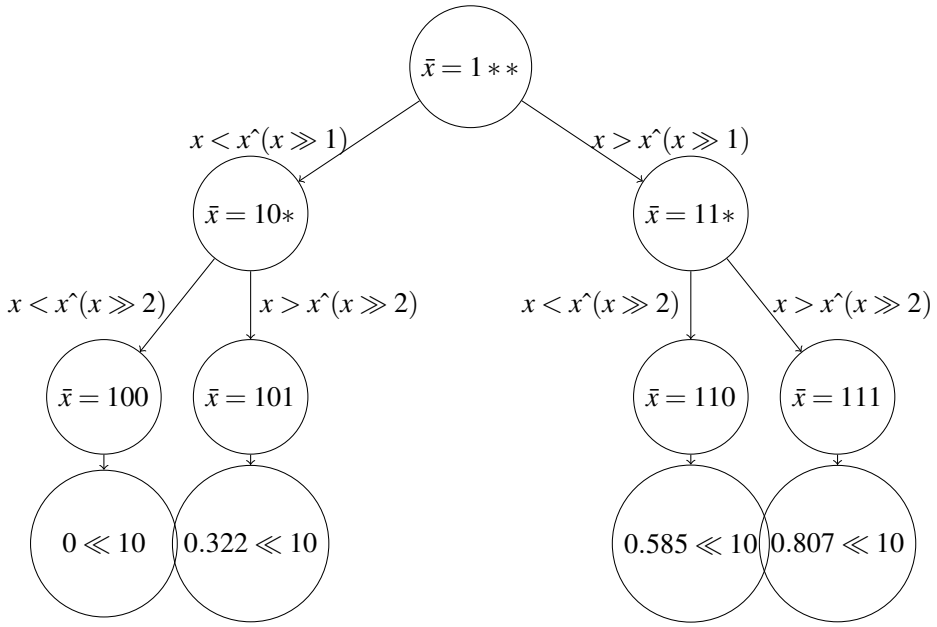


Figure 3.1: Binary-tree data structure to extract the first $N_{bits} = 2$ bits of $\bar{x}$ and retrieve the estimated decimal part ($N_{digits} = 3$)

$\bar{x} \in \{100, 101, 110, 111\}$ (the leftmost 1 is always included in the sub-string). With $N_{digits} = 3$, each of the four cases leads to the following different estimations of the decimal part, computed as $\log_2(1 + \frac{\bar{x} - 2^{N_{bits}}}{2^{N_{bits}}})$:

$$\bar{x} = 100 \implies \log_2(1 + \frac{(000)_2}{(100)_2}) = \log_2(1 + \frac{0}{4}) = 0$$
$$\bar{x} = 101 \implies \log_2(1 + \frac{(001)_2}{(100)_2}) = \log_2(1 + \frac{1}{4}) \approx 0.322$$
$$\bar{x} = 110 \implies \log_2(1 + \frac{(010)_2}{(100)_2}) = \log_2(1 + \frac{2}{4}) \approx 0.585$$
$$\bar{x} = 111 \implies \log_2(1 + \frac{(011)_2}{(100)_2}) = \log_2(1 + \frac{3}{4}) \approx 0.807$$

The greater N_{bits} and N_{digits} are, the more accurate the estimation of the decimal part is. However, the bigger N_{bits} is, the more pre-computed constants must be stored. Note also that, if $n < N_{bits}$, the algorithm performs zero padding.

Unfortunately, for the same limitation of P4 recalled above, retrieving the $\bar{x}$ binary sub-string is not straightforward. However, by iteratively comparing x with $x\hat{}(x \gg j)$, where $j \in \{1, \ldots, N_{bit}\}$ (integer), it is possible to obtain the string $\bar{x}$ and get the associated estimated decimal part. In fact, if $x < x\hat{}(x \gg j)$, it means that the $(j+1)$-th bit of $\bar{x}$ is 1, otherwise it is 0, being the first bit always 1 by definition. To this aim, we can define a binary tree in the P4 program by using $2^{N_{bits}} - 1$ if-else

statements. An example of such a binary tree, in the case $N_{bits} = 2$ and $N_{digits} = 3$, is shown in Fig. 3.1. As shown in the figure, the constant decimal values are amplified 2^{10} times to ensure that they are integer numbers. Once the decimal part is retrieved, it is added to n (also amplified 2^{10} times) to get an integer (amplified) estimation of $\log_2 x$. All these operations are summarized in Line 7 and 8 of Algorithm 1. Finally, the amplified estimated value of $\log_2 x$ (output of $log_2ES(x)$) is used to estimate $\log_d x$ ($log_dES(x, \log_d 2 \ll 10)$ function). The constant value $log_d 2$ is stored amplified 2^{10} times to prevent it being a floating number. For this reason, the result of $\log_2 x \cdot \log_d 2$, where both terms are amplified 2^{10} times, requires a division by 2^{10} to obtain an estimation of $\log_d x$ still amplified 2^{10} times. This can be done using the right-shift operator ($\gg$) (Lines 10-11).

3.3.2 P4Exp algorithm

Given an L-bit integer x and an exponent d, the goal is to estimate an integer approximation of x^d, with d being any real number ($exp_dES(x,d)$ function). Since $x^d = 2^{d \log_2 x}$, x^d first requires the estimation of $\log_2 x$ by means of P4Log, and then the computation of 2^y where $y = d \log_2 x$. Our initial idea was to calculate 2^y by executing $1 \ll d \log_2 x$ in P4 language [17]. Unfortunately, in our case it is not possible to do so. In fact, the output of $log_2ES(x)$ in Algorithm 1 is the estimation of $\log_2 x$ amplified 2^{10} times (to prevent accuracy losses) and 2^y cannot exceed L bits ($2^L - 1$ is the biggest possible value), otherwise the computed number is set to 0 by P4. To ensure that the estimated 2^y value does not exceed L bits, the exponent y cannot be bigger than $\log_2 L$. Considering the biggest possible value for L, i.e., $L = 128$, the inequality $d \log_2 x \cdot 2^{10} < 128$ holds only in the case that $d \log_2 x < 2^{-3}$, which is still a too small value to make this approach meaningful.

To work around such a limitation, the algorithm decomposes $d \log_2 x = e_{int} + e_{dec}$, where e_{int} is its integer part and e_{dec} is its decimal part, meaning that $x^d = 2^{e_{int}} \cdot 2^{e_{dec}}$. P4Exp initially stores the result of $log_2ES(x)$ (i.e., $\log_2 x \ll 10$) multiplied by d in an integer variable called $exp \ll 10$ (Line 2). Note that the decimal part of the product is neglected and that this is the amplified version of the exponent. e_{int} (not amplified) is then calculated by computing $exp \gg 10$, leveraging the limitation of P4 that a resulting floating number is always truncated to its integer part (Line 3). The algorithm then computes the amplified version of e_{dec} as difference between the amplified versions of exp and e_{int} (Line 4). The estimated amplified version of $2^{e_{dec}}$ is retrieved by truncating its binomial series expansion to the first N_{terms} terms (see Section 3.2.2).

All constants in the binomial series expansion need to be amplified by 2^{10} times. The inverse of the factorial number $v!$ in the binomial series can be estimated by $\lfloor \frac{2^{10}}{v!} \rfloor \gg 10$, where $\lfloor \frac{2^{10}}{v!} \rfloor$ is a pre-computed constant in the P4 program. For example, $\frac{1}{2!} = \frac{1}{2}$ can be computed by $\lfloor \frac{2^{10}}{2} \rfloor \gg 10 = 512 \gg 10 = \frac{1}{2}$. As N_{terms} increases, more and more multipliers are amplified 2^{10} times, with the risk of going out of the L-bit range. Thus, the algorithm right-shifts 10 bits after each multiplication in the polynomial: this ensure that the resulted $2^{e_{dec}}$ is only amplified 2^{10}

Algorithm 2: P4Exp algorithm

Input: An L-bit integer x (L $\in \{16, 32, 64, 128\}$) and a given exponent d
Output: An L-bit integer approximation of x^d

1 **Function** $exp_dES(x,d)$**:**
2 $\quad exp \ll 10 \leftarrow d \cdot log_2ES(x)$
3 $\quad e_{int} \leftarrow exp \gg 10$
4 $\quad e_{dec} \ll 10 \leftarrow exp \ll 10 - e_{int} \ll 10$
5 $\quad 2^{e_{dec}} \ll 10 \leftarrow (1 \ll 10) + e_{dec} \ll 10$
6 $\quad +(e_{dec} \ll 10 \cdot (e_{dec} \ll 10 - (1 \ll 10))$
7 $\quad \gg 10 \cdot \lfloor \frac{2^{10}}{2!} \rfloor) \gg 10 + \underbrace{\cdots}_{\text{until } N_{terms}}$ (Binomial series expansion)
8 $\quad$ **if** $e_{int} < 10$ **then**
9 $\quad\quad x^d \leftarrow ((1 \ll e_{int}) \cdot (2^{e_{dec}} \ll 10)) \gg 10$
10 $\quad$ **else**
11 $\quad\quad x^d \leftarrow (1 \ll (e_{int} - 10)) \cdot (2^{e_{dec}} \ll 10)$
12 $\quad$ **return** x^d

Table 3.2: Default parameters for P4Log and P4Exp

Alg	Parameter	Value
P4Log	Digits of precision for decimal part (N_{digits})	3
	Number of bits for estimation of decimal part (N_{bits})	4
P4Exp	Digits of precision in $log_2ES(x)$ (N_{digits})	3
	Number of bits in $log_2ES(x)$ (N_{bits})	7
	Number of terms in binomial series (N_{terms})	7

times. Lines 5-7 reports the estimation of $2^{e_{dec}} \ll 10$ with $N_{terms} = 3$.

Since computed values larger than $2^L - 1$ are set to 0, it must be ensured that x^d will not be out of range for reasonable values of d. Thus, in the case $e_{int} < 10$ (small integer part), x^d is estimated by calculating $2^{e_{int}}$ (not amplified and smaller than 2^{10}), multiplying it by $2^{e_{dec}}$ (amplified as computed above) and dividing it by 2^{10}, to get a non-amplified integer approximation (Lines 8-9).

Conversely, for $e_{int} \geq 10$ integer parts, the algorithm compensates the 2^{10}-times amplification of $2^{e_{dec}}$ by computing $2^{e_{int}-10}$ and multiplying it by the amplified version of $2^{e_{dec}}$ (Lines 10-11). Reducing the size of the exponent of e_{int} by a factor of 10 helps prevent x^d being out of range.

3.4 Evaluation of P4Log and P4Exp

We implemented P4Log and P4Exp in Python for evaluation and sensitivity analysis, reported in this section.

3.4.1 Evaluation metrics and settings

Metrics

We consider *relative error* as key metric. For P4Log, given an input value x and base d, relative error is defined as $\frac{|log_d ES(x,\log_d 2)-\log_d x|}{\log_d x} \cdot 100\%$, where $\log_d x$ is the exact value. For P4Exp, given input base x and exponent d, the relative error is defined as $\frac{|exp_d ES(x,d)-x^d|}{x^d} \cdot 100\%$. In both cases, we consider as acceptable target a relative error of 1%, as also done in previous work [111].

Default experimental settings

Unless otherwise specified, the default tuning parameters in all experiments are the ones reported in Table 3.2.

3.4.2 Evaluation of P4Log

Testing values for P4Log

When using N_{bits} to estimate the logarithm in our P4Log algorithm, all the bits after them are ignored and considered as 0. Intuitively, the algorithm leads to worst-case estimations when most significant bits after N_{bits} are 1s. To make it as general as possible, we choose five different l-bit-length (i.e., $l = 4, 8, 16, 32, 64$ bit) input values where all bits are 1s (and thus the respective decimal value is $2^l - 1$). Moreover, we always consider $d = 2$ as logarithmic base since, as shown in Section 3.3.1, different bases only require the multiplication of $\log_2 x$ with the constant value $\log_2 d$, and this operation does not affect the relative error to the exact value. We also randomly select $5 \cdot 10^6$ integer numbers such that $x \in \{1, 2^{64} - 1\}$ and average the relative error in logarithm estimation, named *AVG* in the following. With $5 \cdot 10^6$ randomly-selected number, 95% confidence-interval width of relative error is always smaller than 0.01%, and we do not plot it in shown graphs since it would overlap with the plotted markers.

Sensitivity to N_{bits} and N_{digits}

Figure 3.2 shows the sensitivity of P4Log with respect to a variation of N_{bits} and N_{digits}. As shown in Fig. 3.2(a), the relative error of $\log_2 x$ decreases as N_{bits} increases, with more significant improvement when the input value x is small. When $N_{bits} = 4$, the relative error is below 1% in all the considered cases, becoming almost 0 when $N_{bits} = 6$. Instead, Fig. 3.2(b) shows that (i) an increase of N_{digits} does not improve much the relative error, (ii) all relative errors are below 1% and (iii) when $N_{digits} = 4$ the relative error reaches its minimum. The *AVG* curve always shows an average relative error below 1%.

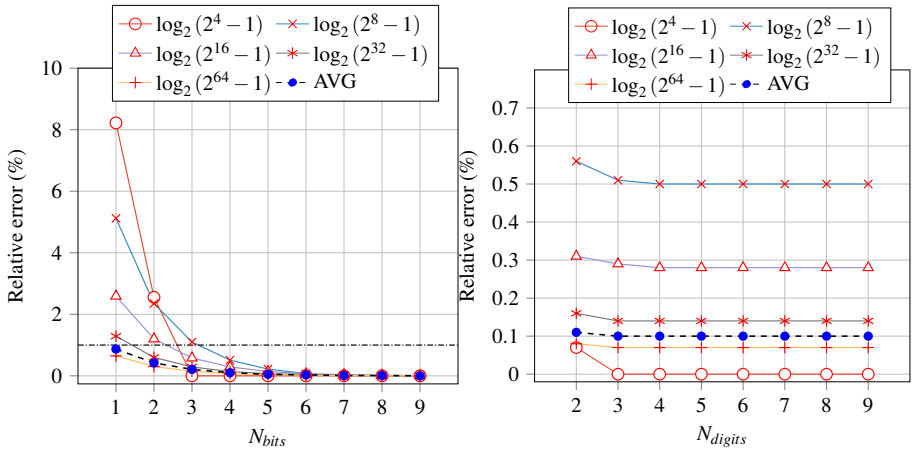


Figure 3.2: Sensitivity of P4Log to N_{bits} (a) and N_{digits} (b)

3.4.3 Evaluation of P4Exp

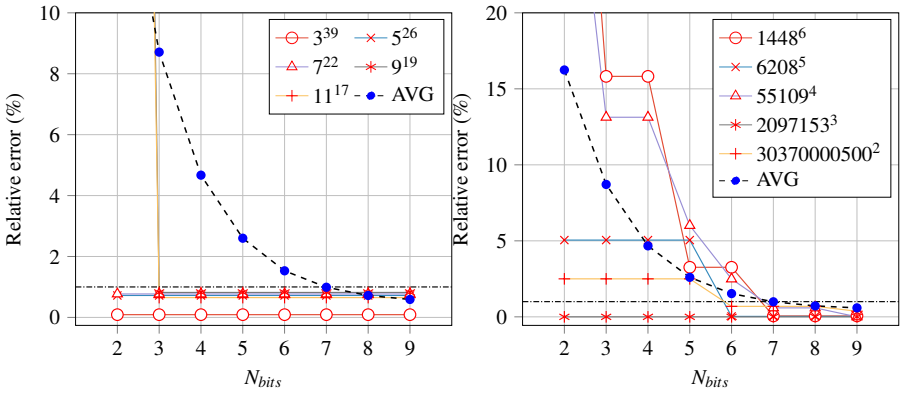
Testing values for P4Exp

In this case, we evaluate the performance of the algorithm when both base and exponent of x^d vary. We choose input values according to the following rules: (i) to evaluate the impact of base variation, we fix a 64-bit integer base x to a chosen value, then we find the integer exponent d that maximizes the output x^d within 64 bits (we call this test *chosen-base variation*); (ii) to evaluate the impact of exponent variation, we fix the integer exponent d to a chosen value, then we find the largest 64-bit integer base x that maximizes the output x^d within 64 bits (we call this test *chosen-exponent variation*); (iii) we select $5 \cdot 10^6$ integer numbers with base $x \in \{1, 2^{32} - 1\}$ and exponent $d \in \{2, 32\}$ both randomly chosen (these ranges ensure that x^d is always within 64 bit) and average the relative error in exponential function estimation, named *AVG* in the following. Also in this case, 95% confidence interval has a width smaller than 0.01% and is not plotted in the graphs.

We analyze the sensitivity of P4Exp with respect to a variation of N_{bits} and N_{digits} of $Log_2ES(x)$ used for exponential estimation (see Algorithm 2) and to a variation of N_{terms}.

Sensitivity to N_{bits}

As shown in Fig. 3.3(a), in the case of chosen-base variation, when $N_{bits} \geq 3$ all the relative errors of considered x^d are under 1%. When N_{bits} is too small (N_{bits} = 2) a very large relative error (around 80%) is experienced for bases $x \geq 9$. This is because a small N_{bits} causes a bad estimation of $d \cdot log_2ES(x)$ that is exponentially



(a) Chosen-base variation (b) Chosen-exponent variation

Figure 3.3: Sensitivity of P4Exp to N_{bits}

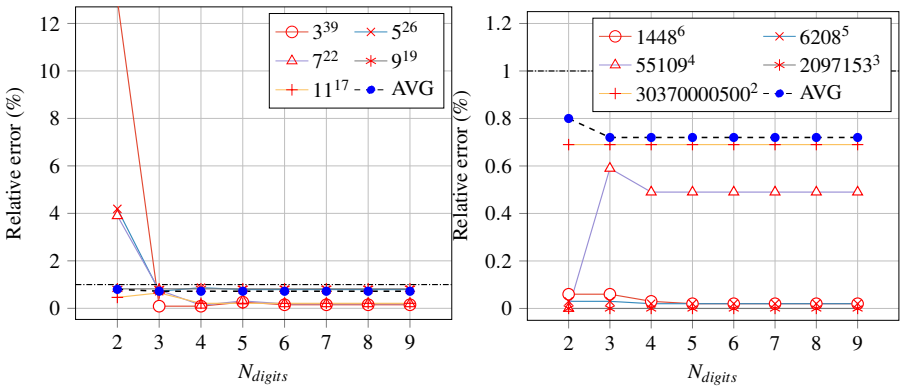
amplified when computing $2^{d \cdot \log_2 ES(x)}$. Such a bad estimation is especially evident when $d \cdot log_2 ES(x)$ is large. Figure 3.3(b) shows instead that exponential functions with small exponent d but large base x are more sensitive to N_{bits} and, as shown also above, the relative error decreases as N_{bits} increases. When $N_{bits} = 7$ relative errors of all estimations are below 1%. The *AVG* curve shows that, on average, a large relative error is experienced when N_{bits} is small and that $N_{bits} \geq 7$ ensures an average error below 1%.

Sensitivity to N_{digits}

As shown in Fig. 3.4(a), in the case of chosen-base variation (and thus bigger exponents), for $N_{digits} \geq 3$ the relative error reaches values below 1%. Instead, Fig. 3.4(b) shows that computations involving smaller exponents but bigger bases are not very sensitive to N_{digits}, being relative errors always under 1%, and that considering bigger N_{digits} may in some cases even prove counterproductive. The *AVG* curve shows a similar trend: on average, an increase in N_{digits} does not affect much performance, and the average relative error is always below 1%.

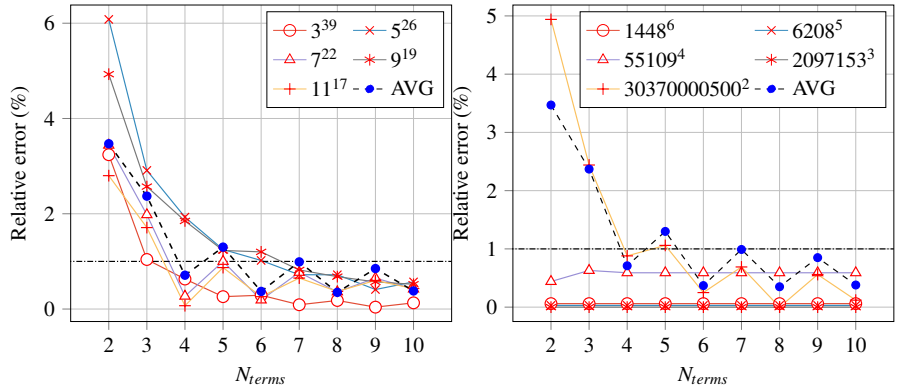
Sensitivity to N_{terms}

Figure 3.5(a) shows that N_{terms} strongly affects the estimation of x^d especially when the exponent is large and the base is small. Relative errors oscillate but, in a long term, decrease as N_{terms} increases. Oscillation is due to the way how binomial series converges. When $N_{terms} \geq 7$ relative error for all estimations is under 1%. Instead, as shown in Fig. 3.5(b), with small chosen exponents and large bases, relative errors are all below 1% when $N_{terms} \geq 6$. Oscillation is also well visible in the *AVG* curve that shows how $N_{terms} \geq 6$ leads to an average relative error below 1%.



(a) Chosen-base variation (b) Chosen-exponent variation

Figure 3.4: Sensitivity of P4Exp to N_{digits}



(a) Chosen-base variation (b) Chosen-exponent variation

Figure 3.5: Sensitivity of P4Exp to N_{terms}

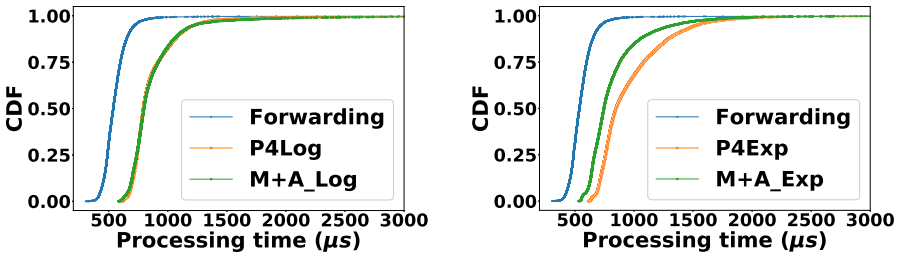
3.5 Evaluation of packet processing time in programmable data planes

In the previous section, we proposed and evaluated the accuracy of two algorithms for exponential function and logarithm estimation. We believe that a comparison of *packet processing time* between P4Exp, P4Log and the corresponding state of the art strategies is important to understand how the different approaches affect packet processing in the P4 pipeline, and to take a decision on what exponential function and logarithm estimation algorithms we should leverage for the design and implementation of network monitoring tasks as presented in the next Chapters.

We chose Mininet [8] as emulated network environment for a single P4 switch. The data plane pipeline is described by P4 code compiled using the *bmv2* behavioral model [13]. We then connected the P4 switch to two hosts, ensuring that

Table 3.3: P4 programs properties

Algorithm	Parameter [55]	Value [55]	Instructions	M+A entries
P4Log	N_{digits}	3	47	1
	N_{bits}	4		
P4Exp	N_{terms}	7	64	1
M+A_Log	-	-	0	1920
M+A_Exp	-	-	0	2049
Forwarding	-	-	0	1



(a) Log estimation

(b) Exp estimation

Figure 3.6: Cumulative distribution function of packet processing time

packets can be forwarded from one to the other host through the switch. The emulated environment is built on top of a virtual machine deployed by OpenStack on our local testbed with dedicated access to 4 $\times$ 2.7GHz CPU cores and to 4GB of RAM. We used WireShark [29] to capture a packet timestamp t_{in} at the ingress interface of the switch and the timestamp t_{out} at the egress interface when the same packet is forwarded to the destination host. The packet processing time is thus calculated by $t_{out} - t_{in}$.

In addition to P4Exp and P4Log implementations, we implemented the logarithmic and exponential function estimation strategies reported in [111], named here as *M+A_Log* and *M+A_Exp*, respectively. For 64-bit operands in P4, to ensure a relative error below 1% for the estimated values with respect to the real ones, M+A_Exp needs 2048 entries in an exact match table, while M+A_Log requires 1919 entries in a ternary match table (to be stored in TCAM). However, in behavioral model [13], any M+A table can include at most 1024 entries, so we had to assign two exact match tables for M+A_Exp and two ternary match tables for M+A_Log. A simpler benchmark strategy, named *Forwarding*, is also implemented: it only requires a M+A table for forwarding the packets from source to destination host according to pre-defined flow rules. All the other strategies implement the same forwarding logic in their pipeline. P4Log and P4Exp parameters are taken from [55] and shown in Table 3.3. The table also summarizes the number of required instructions (i.e., logical and arithmetical operations) and of M+A entries for all the considered strategies (including forwarding capabilities), showing the inherent differences of the approaches.

We then evaluate the packet processing time, considering base-2 logarithm and exponential function. We generated and forwarded 10000 packets: Fig. 3.6 shows the cumulative distribution function (CDF) of packet processing time. As shown in Fig. 3.6(a), both P4Log and M+A_Log cause a higher processing time than Forwarding since they need to carry out more complex operations. However, their CDF curves are almost overlapped: this means that P4Log does not cause any additional overhead on processing time with respect to M+A_Log, but it has the benefits of not requiring any M+A table. Likewise, Fig. 3.6(b) reveals that both exponential function estimation strategies slightly increase the processing time with respect to Forwarding. P4Exp has just a slightly higher packet processing time than M+A_Exp but, also in this case, it does not require any M+A table to work. Note also that packets, in real high-performance programmable switches, are expected to be processed in few hundreds of ns, thus such a difference in processing time would impact even less on performance (in absolute terms). We have shown that P4Log and P4Exp have comparable accuracy (see [55]) and efficiency as the state of the art, while preventing from the usage of expensive and power-hungry switch memory (e.g. TCAM) for their execution: we thus chose to leverage P4Log and P4Exp for all the logarithmic and exponential-function estimations needed in the following Sections.

3.6 Related work

3.6.1 Logarithmic and exponential function estimation in P4

Since P4 language does not support logarithm and exponential function computation, many advanced algorithms leveraging on those operations (e.g., HyperLogLog for linear counting [63]) are not directly implementable using such domain-specific language. However, these advanced algorithms are useful for executing many network functionalities, such as congestion control [78], flow-cardinality estimation [111] and DDoS detection [85][121], so finding a way to support them is of paramount importance. Naveen et al. [111] have already successfully implemented estimation of logarithm and exponential function in P4, but their strategy requires the storage of appropriate pre-computed values in TCAM. It is shown that, to ensure a relative error in the estimation below 1%, they require around 0.5KB of TCAM memory occupation. This is something that P4Exp and P4Log algorithms do not need.

3.7 Concluding remarks

In this chapter, we first propose a novel algorithm for the estimation of logarithm, called *P4Log*, that only uses arithmetic operations supported by the P4 language. Moreover, we propose another algorithm, called *P4Exp*, for the estimation of exponential functions with real-number exponent.

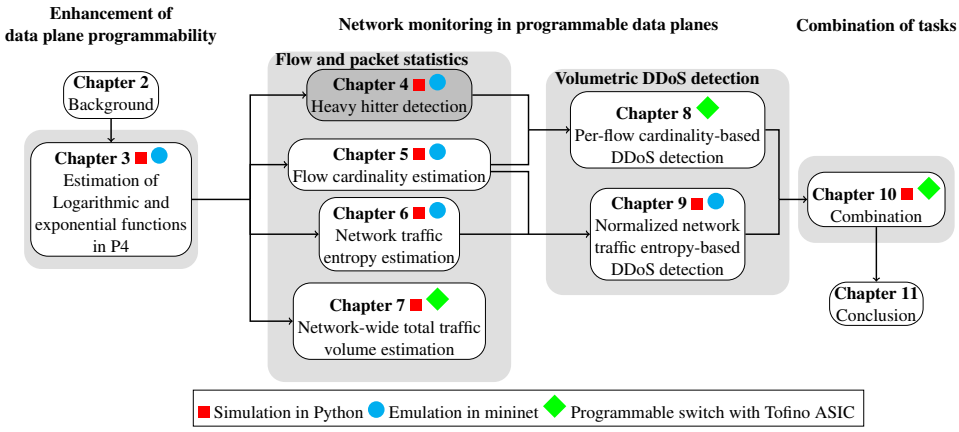
We then evaluate P4Log and P4Exp by means of simulations to show their effectiveness and their sensitivity to different tuning parameters. Results show that our algorithms can ensure similar relative error as state-of-the-art solutions that leverage on ternary Match+Action (M+A) tables to store some pre-computed values for estimation. The advantage of our strategies with respect to the state of the art is three-fold. First, our approach avoids the usage of any M+A table: this is especially beneficial to save memory consumption of TCAM (which is used to store ternary M+A tables but is limited, power-hungry and expensive). Additionally, our approach does not require any interaction with the control plane in executing the foreseen operations. Conversely, state-of-the-art solutions require that M+A tables are properly populated by a controller, generating some communication overhead: this is why we claim that our strategies work *entirely* in the data plane.

The results in the emulated network environment show that P4Log does not cause any additional overhead on processing time with respect to state-of-the-art solution, and P4Exp has just a slightly higher packet processing time but, also in this case, it does not require any M+A table to work. Note also that packets, in real high-performance programmable switches, are expected to be processed in few hundreds of ns, thus such a difference in processing time would impact even less on performance (in absolute terms).

We have shown that P4Log and P4Exp have comparable accuracy and efficiency as the state of the art, while avoiding from the usage of expensive and power-hungry switch memory (e.g. TCAM) for their execution: we thus chose to leverage P4Log and P4Exp for all the logarithmic and exponential-function estimations needed in the following chapters. The two algorithms will be used in Chapter 6 for network traffic entropy estimation, Chapter 5 for flow cardinality estimation and Chapter 9 for normalized network traffic entropy-based DDoS detection.

4

Heavy-hitter detection



In this chapter, we present our study on the heavy hitter phenomenon in programmable switches. We start by proposing a new network-wide heavy hitter detection incorporating programmable switches and SDN controller, aiming at minimizing communication overhead and maximizing the detection performance. We then consider a partial deployment as well with the goal to use the smallest number of switches to achieve the best performance in the ISP network.

This chapter is based on our previously published paper "Damu Ding, Marco Savi, and Domenico Siracusa. "Incremental deployment of programmable switches for network-wide heavy-hitter detection." 2019 IEEE Conference on Network Softwarization (NetSoft). IEEE, 2019." [53] and "Damu Ding, Marco Savi, and Domenico Siracusa "An incrementally-deployable P4-enabled architecture for network-wide heavy-hitter detection." IEEE Transactions on Network and Service Management (2020): 75-88." [54].

4.1 Introduction

In the last chapter we have investigated the limitations of P4 and developed the enhancements. Starting from this chapter, we present several monitoring tasks that have been implemented by us in programmable data planes.

Heavy hitters refer to all flows that are larger (in number of packets or bytes) than a fraction of the total packets seen on the link. Identifying the "heavy hitter" flows or flows with large traffic volumes in the data plane is of primary importance for several applications, such as flow-size aware routing, DoS detection, and traffic engineering. However, measurement in the data plane is constrained by the need for limited memory in switching hardware and large communication overhead with controller.

The goal of this chapter is twofold. First, we study network-wide heavy-hitter detection strategy by analyzing the exposed filtered information from all programmable switches in the network. And second, we propose a novel approach for an *incremental deployment* of programmable switches in Internet Service Provider (ISP) networks with the goal to have visibility over the largest number of distinct flows.

We propose a new algorithm that is capable of detecting *network-wide* heavy flows (i.e., *heavy hitters*) using as input only partial information from the data plane. We evaluate our incremental deployment strategy alongside the proposed heavy-hitter detection algorithm in simulation. We also implemented our heavy-hitter detection strategy in P4 [42] and tested it in an emulated environment. We then propose a novel approach for an *incremental deployment* of programmable switches in Internet Service Provider (ISP) networks. To optimize network-wide monitoring practices, it is important to have visibility over the largest number of distinct flows. To this end, we exploit the HyperLogLog algorithm [63] that is generally used for the count-distinct problem, approximating the number of distinct elements in a multi-set. By comparing our incremental deployment solution with state-of-the-art proposals, results show that we can achieve a better monitoring accuracy using less switches. Moreover, our heavy-hitter detection strategy outperforms existing ones in terms of F1 score while relying on less information from the data plane, while leading to only slightly higher packet processing times in executing the programmable switch pipeline.

In this chapter, we focus on answering the following questions:

- How to effectively detect heavy hitters in programmable data planes?
- How SDN controller coordinates programmable switches to perform network-wide heavy hitter detection?
- How to deploy limited number of programmable switches in the network?
- What is the performance of our network-wide heavy hitter detection in the network with partially deployed programmable switches?

4.2 Basic knowledge and used compact data structure

4.2.1 Estimation of flow packet count for heavy-hitter detection

Estimating the number of packets for a specific flow is fundamental for a proper detection of heavy hitters. Different algorithms exist to perform such an estimation: we choose to use Count-Min Sketch [48], which relies on a probabilistic data structure (i.e., sketch) based on pairwise-independent hash functions. Count-Min Sketch envisions two types of operations: *Update* and *Query*. Update operation is responsible for continuously updating the sketch with information on incoming packets in the switch, whereas Query operation is used to retrieve the estimated number of incoming packets for a specific flow. To formalize the problem, we consider a stream of packets $S = \{a_1, a_2, ..., a_m\}$. The Count-Min Sketch algorithm returns an estimator of packet count $\hat{f}_x$ of flow $x \sim$ (srcIP$_x$,dstIP$_x$) satisfying the following condition: $Pr[|\hat{f}_x - f_x| > \varepsilon|S|] \leq \delta$, where ε $(0 \leq \varepsilon \leq 1)$ is the relative biased value and δ $(0 \leq \delta \leq 1)$ is the error probability. In Count-Min Sketch, the space complexity is O($\varepsilon^{-1} log_2(\delta^{-1})$), and per-update time is O($log_2(\delta^{-1})$) [50]. Additionally, the estimation of packet count satisfies $f_x \leq \hat{f}_x \leq f_x + \varepsilon|S|$, where f_x is the real packet count value. The accuracy of Count-Min Sketch depends on ε and δ, which can be tuned by respectively defining *(i)* the output size N_s of each hash function and *(ii)* the number N_h of hash functions of the data structure.

In previous definitions, a heavy hitter is a flow whose packet count overcomes a threshold $\vartheta|S|$ $(0 < \vartheta < 1)$. If a Count-Min Sketch is adopted, the probability to erroneously detect a heavy hitter due to packet miscount is defined in the following way: $Pr[\exists x|\hat{f}_x \geq (\vartheta + \varepsilon)|S|] \leq \delta$. If N_h is large enough, error probability δ is negligible.

4.2.2 Estimated count of distinct flows

An efficient and effective method to count a number of distinct items from a set is HyperLogLog [63]. In our specific case, given a packet stream $S = \{a_1, a_2, ..., a_m\}$, where each packet is characterized by a specific (srcIP,dstIP) pair (generically called *flow key*), it returns an estimation of cardinality of flows, i.e., how many (srcIP,dstIP) distinct pairs exist in the stream[1]. In this chapter, we use $\hat{n} \leftarrow Hll(S)$ as notation to indicate input and output of the HyperLogLog algorithm: Hll indicates the algorithm, S the input packet stream and $\hat{n}$ the cardinality of flows (i.e., number of distinct flows). The relative error of HyperLogLog is only $\frac{1.04}{\sqrt{m}}$, where m is the size of HyperLogLog register. Apart from its high accuracy, HyperLogLog is also very fast since the query time complexity is O(1). Moreover, calculating the *union* (or merge) of two or more HyperLogLog data structures (also called *sketches*) is also very efficient, and can be used to count the number of distinct flows in the union of two (or more) data streams, e.g. S_a and S_b. In our notation,

[1] Note that in this chapter, without any loss of generality, we consider source/destination pairs as flow identifiers. However, other definitions could also be adopted (e.g. 5-tuple).

Table 4.1: Main symbols and meanings

S	Packet stream
$\|S\|$	Packet count of stream S
δ	Error probability of Count-Min Sketch
ε	Relative biased value of Count-Min Sketch
N_h	Number of hash functions in Count-Min Sketch
N_s	Output size of hash functions in Count-Min Sketch
$\hat{n}$	Estimated number of distinct flows
$\hat{f}_x$	Estimated packet count of flow x
f_x	Real packet count of flow x
θ_H	Heavy-hitter identification fraction (controller)
K	Sampling rate for heavy-hitter detection (switch)
W	Local ratio for heavy-hitter detection (switch)
R	Recall
Pr	Precision
$F1$	F1 score
T_{int}	Time interval
N	Number of P4 switches in the network
P	Number of legacy switches to be replaced by programmable switches
Hll	HyperLogLog algorithm
m	Register size in HyperLogLog

this can be written as $\hat{n}_{union} \leftarrow Hll(S_a \cup S_b)$, where $\hat{n}_{union}$ is the number of distinct flows of the packet streams union $S_a \cup S_b$.

4.3 A network-wide heavy-hitter detection strategy robust to partial deployment

In this section we propose a novel network-wide heavy-hitter detection strategy. While taking inspiration from an existing approach [69], it differs from the state of the art by introducing the concept of *local* and *global sample lists*, which make it robust to partial deployments aiming at maximizing the network flow visibility.

Figure 4.1 shows the interaction between switches and a centralized controller for network-wide heavy-hitter detection, in the case of partial deployment of programmable switches (i.e., when only some switches can perform monitoring operations in the data plane). Time is divided in intervals, and in every time interval each programmable switch dynamically stores in a *sample list* only those flows whose packet count is larger than a dynamic *sampling threshold*. At the end of each time interval, if any programmable switch stores in its sample list one or more flows with packet count larger than a dynamic *local threshold*, it reports a *true flag* to the centralized controller. The flows whose packet count overcomes the local threshold are called *potential heavy hitters*. The controller, if at least one true-flag report is received, then polls all the programmable switches in the network to gather the {flow key, packet count} pairs of all the flows stored in their sample lists. This

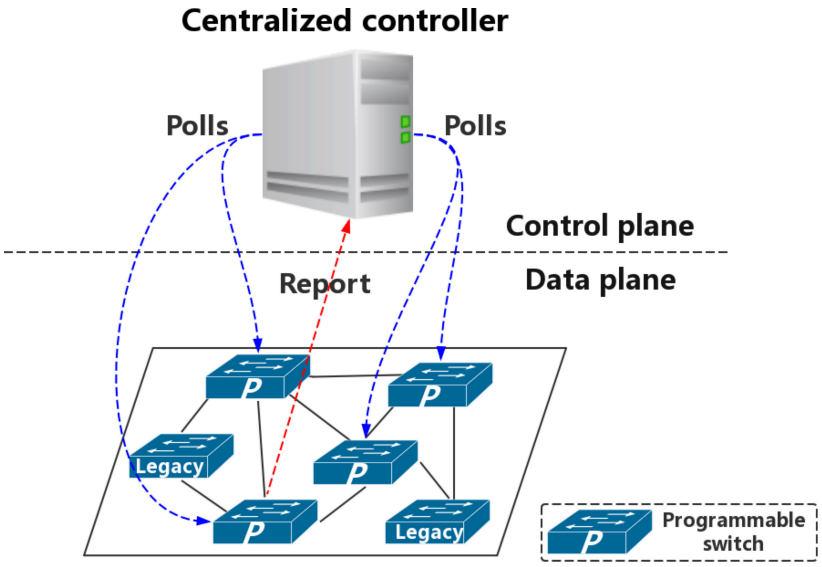


Figure 4.1: Interaction between controller and programmable switches for network-wide heavy-hitter detection in a partial deployment scenario

information is used to estimate the whole *network volume*, i.e., the number of all the unique packets transmitted in the network in the given time interval. Finally, the controller computes the so-called *global threshold* leveraging the estimated network volume and gets all the network-wide heavy hitters, i.e., the flows from sample lists whose packet count is larger than the global threshold. Finally, all the flow and packet statistics are reset and a new time interval is started.

In the following subsections, we formalize the problem of network-wide heavy-hitter detection and describe in detail the algorithms running both in the programmable switches and in the controller to implement the proposed high-level strategy.

4.3.1 Problem definition

We formulate the network-wide heavy-hitter detection problem as follows.
Given:

- A heavy-hitter identification fraction θ_H ($0 < \theta_H < 1$);
- A time interval T_{int};
- The set of unique packets S transmitted in the network;

Identify the set of flows which are network-wide heavy hitters (HH), i.e., carry in the time interval T_{int} a number of packets larger than the global threshold $\theta_H |S|_{tot}$, where $|S|_{tot}$ is the *network volume* (i.e., number of transmitted packets).

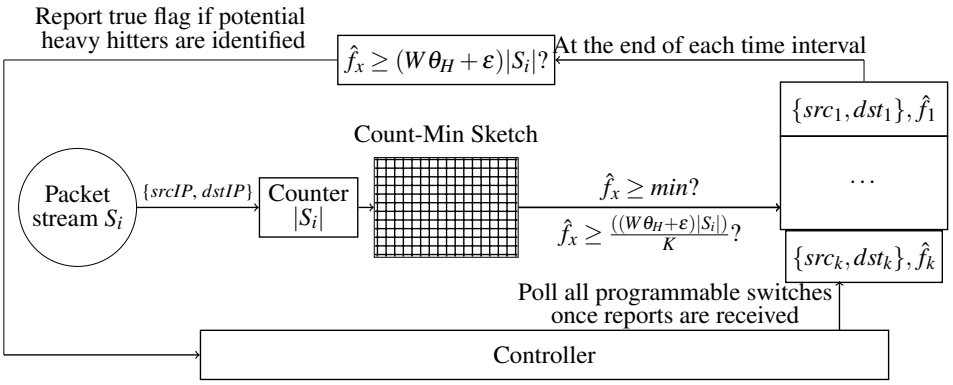


Figure 4.2: Scheme of the proposed network-wide heavy-hitter detection strategy

4.3.2 Algorithm in programmable switches (Data plane)

As shown in Fig. 4.2, when a packet comes into a programmable switch i, a packet counter named $|S_i|$ is updated to count all the incoming packets. A Count-Min Sketch data structure, which is used to store the estimated packet counts for all the flows, is updated to include the information from the current packet, and then it is queried to retrieve the estimated packet count $\hat{f}_x$ for the flow $x \sim$ (`srcIP`$_x$,`dstIP`$_x$) such packet belongs to. This information is used to understand whether the packet belongs to a flow that must be inserted in the sample list.

The flow x is inserted in the sample list if $\hat{f}_x \geq \frac{(W\theta_H+\varepsilon)|S_i|}{K}$, where $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$ is the sampling threshold. The parameters W ($W \geq 1$) and K ($K \geq 1$) affect the size of the sample list: the larger W and the smaller K are, the smaller the sample list size is, thereby consuming less memory in the switch. However, as we will report in the next subsection, this would reduce the accuracy on the estimation of the overall network volume and on the identification of heavy hitters. Therefore, K (called *sampling rate*) should be carefully set in each programmable switch to store only flows carrying a significant number of packets. ε is instead the biased value caused by Count-Min Sketch (see Section 4.2): we sum $\varepsilon = 1/N_s$ to $W\theta_H$ in order to compensate such bias. The sample list is thus used to dynamically store the packet counts for the most frequent flows crossing the switch.

Since at the beginning of each time interval T_{int} the sampling threshold is low, being $|S_i|$ a small value (that can even be < 1), flows with small packet counts would be stored in the sample list.

We thus introduce a parameter *min* operating in conjunction with the sampling threshold: only if packet count of the considered flow is larger than both *min* and sampling threshold $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$, the flow is inserted in the sample list. This is described in Lines 6-11 of Algorithm 3.

At the end of the time interval T_{int}, $|S_i|$ counts all the incoming packets in the considered time frame. Thus, as shown in Lines 13-14, the algorithm removes from the sample list all those flows with packet count lower than $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$, where

Algorithm 3: Network-wide heavy-hitter detection algorithm - Programmable switch $i \in \mathscr{P}$ (Data plane)

Input: Flow stream S, Local minimum min, Heavy-hitters identification fraction θ_H, Local ratio W, Sampling rate K, Count-Min Sketch size $N_h \times N_s$, Time interval T_{int}

Output: $flag$ (`true` if potential heavy hitters are identified in T_{int}, `false` otherwise)

1 $\varepsilon \leftarrow 1/N_s$
2 **Function** *StoreFlowsInSampleList*:
3 $|S_i| \leftarrow 0$
4 $flag \leftarrow$ `false`
5 **while** *currentTime* $\in T_{int}$ **do**
6 **for** *Each packet belonging to flow* $x \sim (srcIP_x, dstIP_x)$ *received* **do**
7 $|S_i| \leftarrow |S_i| + 1$
8 **if** $\hat{f}_x \geq min$ *and* $\hat{f}_x \geq \frac{(W\theta_H+\varepsilon)|S_i|}{K}$ **then**
9 $SampleList_i(x) \leftarrow \hat{f}_x$
10 **Function** *PotentialHHsDetection*:
11 **if** *currentTime* $=$ *End time of* T_{int} **then**
12 **for** *Each flow x in* $SampleList_i$ **do**
13 **if** $\hat{f}_x < \frac{(W\theta_H+\varepsilon)|S_i|}{K}$ **then**
14 $SampleList_i.remove(x)$
15 **if** $SampleList(x) \geq (W\theta_H + \varepsilon)|S_i|$ **then**
16 $flag \leftarrow$ `true`
17 **return** $flag$

$|S_i|$ is the final stored value. This means that the algorithm keeps in the sample list only those flows which have packet counts larger than the final sampling threshold, while discarding the flows with packet counts greater than the temporary threshold dynamically computed and updated within the time interval. Note that the sample list stores at most $\frac{K}{(W\theta_H+\varepsilon)}$ flows, and thus its memory occupation increases as K increase or W decrease, as already discussed.

Finally, the algorithm evaluates whether potential heavy hitters cross the switch. They are the flows whose packet counts are greater than the switch local threshold, set as $(W\theta_H + \varepsilon)|S_i|$. A true flag is sent to the controller if at least one potential heavy hitter is detected, otherwise no information is sent (Lines 15-17). Note that local threshold and sampling threshold are similar: with respect to sampling threshold, local threshold just misses in its definition K, which has been introduced to set the size of the sample list. The primary role of W is instead to set the proportion (or *ratio*) between the local threshold $(W\theta_H + \varepsilon)|S_i|$ and the global threshold

$\theta_H|S|_{tot}$, being S_i a local (and smaller) value than $|S|_{tot}$.

4.3.3 Algorithm in centralized controller (Control plane)

At the end of the time interval T_{int}, once the controller receives from programmable switches any report including a true flag, it polls all of them to obtain their sample lists. Note that different sample lists can include the estimated packet count for the same flows: this happens if a flow crosses multiple programmable switches. To avoid an overestimation of $|S|_{tot}$, Algorithm 4 makes sure that *(i)* only the minimum-estimated packet count is kept, i.e., the one less overestimated by Count-Min Sketch, and *(ii)* it is stored in a list (i.e., *GlobalSampleList*) including all distinct flows from sample lists (Lines 2-8). The algorithm then sums up the packet counts for all the identified distinct flows and estimates the whole network volume $|S|_{tot}$ (Lines 9-12). If packet counts of flows belonging to *GlobalSampleList* (which surely includes the potential heavy hitters) are larger than the global threshold $\theta_H|S|_{tot}$, we consider them as network-wide heavy hitters HH (Lines 13-17). At last, the controller triggers the reset of counters in all programmable switches.

4.3.4 Implementation in P4 language

Programmable switches (Data plane)

We have implemented our prototype of algorithm in P4, leveraging the P4 behavioral model [13] to describe the behavior of P4 switches (e.g., parsers, tables, actions, ingress and egress in the P4 pipeline). The source code is available at [16]. In P4-enabled switches, registers are stateful memories whose values can be read and written [18]. We first allocate a one-sized register S_i to count the overall number of packets in the given time interval T_{int}. When each packet arrives at the switch, the value in this register is incremented by one.

Moreover, we allocated several one-dimensional registers for Count-Min Sketch implementation. We chose *xxhash* [46] to implement the needed pairwise-independent hash functions by varying the *seed* of each hash function, which is associated to a different register. In our case, the seed is set to be the same value as the index of each register. In the Count-Min Sketch *Update* operation, each register relies on its corresponding hash function, which takes as input $x \sim$`(srcIP,dstIP)`, to obtain as output the index of the register cell whose value must be incremented by one.

To perform the *Query* operation on the Count-Min Sketch, we set a *count-min* variable to the queried value obtained from the first register, which is associated to the first hash function. Consequently, the queried value for the remaining registers is compared with *count-min*. If the queried value for a register is smaller than current *count-min* value, then *count-min* is updated accordingly. Its final value is thus the packet count estimation *count-min* $= \hat{f}_x$ for the queried flow x.

One of the drawbacks of P4 language is that it does not allow variable-sized lists, as *sample list* is. We thus used three same-sized registers, named *samplelist_src*, *samplelist_dst* and *samplelist_count* to implement the sample list and

Algorithm 4: Network-wide heavy-hitter detection algorithm - Centralized controller (Control plane)

Input: Heavy-hitter identification fraction θ_H, Time interval T_{int}, Sample lists $SampleList_i$ from all programmable switches $i \in \mathscr{P}$

Output: Set of network-wide heavy hitters HH in T_{int}

Function *RetrieveDistinctFlowsPacketCounts*:
  for *Each switch* $i \in \mathscr{P}$ **do**
    for *Each flow x in* $SampleList_i$ **do**
      if *flow x is in GlobalSampleList* **then**
        if $SampleList_i(x) < GlobalSampleList(x)$ **then**
          $GlobalSampleList(x) \leftarrow SampleList_i(x)$
      else
        $GlobalSampleList(x) \leftarrow SampleList_i(x)$
Function *EstimateVolume*:
  $|S|_{tot} \leftarrow 0$
  for *Each flow x in GlobalSampleList* **do**
    $|S|_{tot} \leftarrow |S|_{tot} + GlobalSampleList(x)$
Function *GetNetworkWideHH*:
  for *Each flow x in GlobalSampleList* **do**
    if $GlobalSampleList(x) \geq \theta_H |S|_{tot}$ **then**
      $HH.add(x)$
  return HH

to store *srcIP*, *dstIP* and *packet count* of flows, respectively. Information on flow keys `(srcIP,dstIP)` associated to significant packet counts is stored in these registers. Since P4 language does not support *for* loops to find the flows with smallest packet count in the sample list to be replaced, the hash function CRC32 is used to decide whether to replace list's entries.

When the stored *count-min* value is larger than both the pre-set minimum value *min* and sampling threshold $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$ (where the value of $\mathsf{S_i}$ is read from register $\mathsf{S_i}$), the algorithm hashes the flow key `(srcIP,dstIP)` using CRC32 hash function and checks whether the sample list register cells indexed by the output value of CRC32 are empty or not. If they are empty, *srcIP*, *dstIP* and *packet count* are added to sample list registers in the same indexed position. Otherwise, the switch compares the current estimated packet count $\hat{f}_x$ with stored packet count $\hat{f}_y$ (from the generic stored flow y) in *samplelist_count*. If the current packet count $\hat{f}_x$ is larger than existing packet count $\hat{f}_y$, the algorithm replaces *srcIP*, *dstIP* and *packet count* to new values from flow x. Additionally, if packet count $\hat{f}_x$ is larger than current largest value *max* stored in the register *count_max*, *max* is replaced by $\hat{f}_x$.

Table 4.2: Simulation parameters

Time interval T_{int}	5s [93]
Sampling rate K	10
Local ratio W	3
Heavy-hitter identification fraction θ_H	0.05% [93]
Local minimum *min*	1
Count-Min Sketch size ($N_h \times N_s$)	40×10000

Using the P4 behavioral model, *ingress_global_timestamp* allows to record the timestamp when the switch starts processing the incoming packet. Hence, when *ingress_global_timestamp* is larger than current time interval T_{int} end time, and *max* in the register *count_max* is larger than current local threshold $(W\theta_H + \varepsilon)|S_i|$, the switch clones the current packet and embeds a customized one-field header including the field *Flag* with binary value 1. This cloned packet is sent to the controller to report that a potential (and local) heavy hitter exists, while the original packet is forwarded to the expected destination.

Centralized controller (Control plane)

We implemented a preliminary version of the centralized controller in Python. The controller can read the registers in the switches by using the *simple_switch_CLI* offered by the P4 behavioral model. If the controller receives packets from P4 switches at the end of time interval T_{int}, it gathers the registers *samplelist_src*, *samplelist_dst* and *samplelist_count* from P4 switches and merges them into a global sample list. Finally, according to the heavy-hitter global threshold $\theta_H|S|_{tot}$, the controller is able to detect network-wide heavy hitters. Finally, the controller resets all registers in P4 switches and starts a new round of heavy-hitter detection.

4.4 An algorithm for the incremental deployment of programmable switches

In this section we propose a novel algorithm for the incremental upgrade of a legacy infrastructure with programmable switches, which aims at ensuring good network monitoring performance, as discussed in Section 4.4.1.

4.4.1 Hints for improved monitoring performance with limited flow visibility

When only a limited number of programmable switches can be deployed in an ISP network, the network operator must ensure that they are deployed in such a way it is made the best use of them in terms of monitoring performance, measured as we will explain later by F1 score. Fig. 4.3 shows the results of a simple test: we simulated an ISP topology of 100 nodes [70] with real traffic, and we evaluated the F1

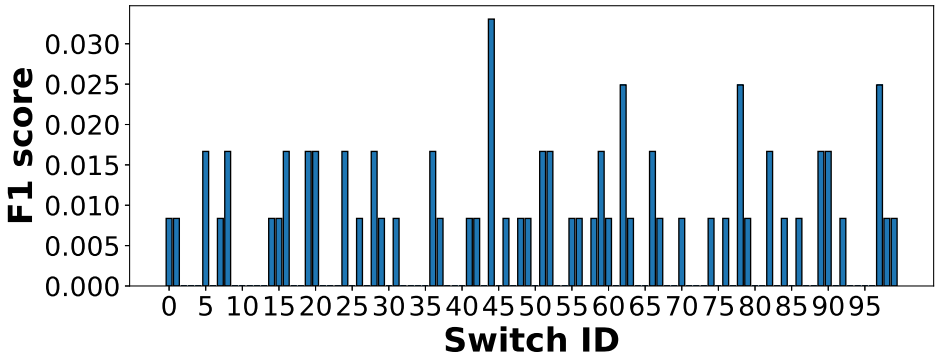


Figure 4.3: F1 score of Harrison's heavy-hitter detection strategy with single programmable-switch deployment

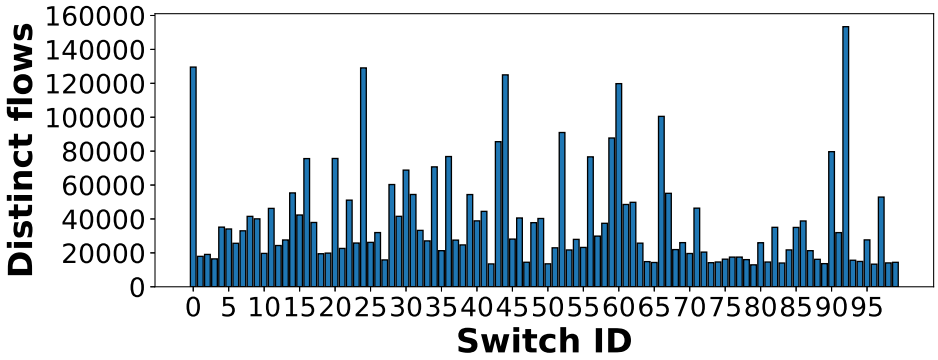


Figure 4.4: Number of distinct flows crossing each switch

score of an existing threshold-based network-wide heavy-hitter detection strategy, proposed by Harrison et al. [69] (see Section 4.5 for more details on simulation settings and evaluated metrics) when only one legacy switch/router[2] is replaced with a programmable switch. The graph shows all the 100 possible deployments. What we can see is that the F1 score *(i)* is in all cases low but *(ii)* substantially varies depending on the placement position of the programmable switch. Consideration *(i)* comes from the fact that by replacing only one switch the flow visibility is very low, since only heavy hitters crossing such switch can be detected, while *(ii)* proves that how we deploy programmable switches in the network is a fundamental aspect to ensure good monitoring performance with limited flow visibility.

Our intuition is that, when deploying a single programmable switch, *an effective strategy is to replace the one crossed by the highest number of distinct flows*.

[2]In the remainder of this chapter, we will use the generic term *legacy device* to generically refer to legacy switches or routers. In fact, programmable switches can support both Layer-2 and Layer-3 functionalities.

This is because the highest the number of monitored flows is, the highest (on average) the chance of monitoring some heavy hitters is. This consideration holds also for any other flow-based network-wide monitoring algorithm (e.g. heavy changes detection [89], network traffic entropy estimation [93], etc.), whose analysis is however out of the scope of this chapter. Fig. 4.4 shows the number of distinct flows crossing each one of the switches in the given time interval. Two observations can be made: *(i)* if a switch crossed by a few number of distinct flows is replaced, it is highly probable that it cannot detect any heavy hitters (i.e., F1 score is often zero); *(ii)* a (weak) correlation between F1 score and number of monitored distinct flows indeed exists. For example, replacing the switch with ID 44 leads to the highest F1 score, and the same switch is one of the switches crossed by the highest number of distinct flows. However, the network-wide heavy-hitter detection strategy proposed in [69] has not been explicitly designed to best exploit the available information on switches' monitored distinct flows, unlike our proposed strategy (see Section 4.3), so correlation between F1 score and number of distinct flows is small.

The same considerations can be made when more than one programmable switches have to be deployed in the network. In this case, it must be ensured that *the subset of programmable switches to be deployed monitors the highest number of distinct flows overall* (i.e., neglecting duplications): this guarantees satisfactory performance in the execution of monitoring tasks such as heavy-hitter detection. The considerations and intuitions discussed in this section have thus guided us in the design of our algorithm for incremental deployment of programmable switches, shown in Section 4.4.

4.4.2 Problem definition

Our problem of incremental deployment of programmable switches can be formalized in the following way.
Given:

- An ISP network topology of a legacy network infrastructure $\mathscr{G} = (\mathscr{N}, \mathscr{L})$, where $\mathscr{N}$ is the set of legacy devices and $\mathscr{L}$ the set of interconnection links;
- A long-term estimation of the transmitted packets in the network between different sources and destinations (i.e., traffic matrix T), including their routing paths and possible re-routing paths in case of failures. From this information it is possible to retrieve the estimated packet stream T_i for each switch $i \in \mathscr{N}$. Note that, unlike in intra data-center scenarios, traffic distribution in ISP networks is much less dynamic, being routes prevalence and persistence increasing over time [45]. For this reason, it is possible to estimate a reasonable traffic matrix T using historical data;
- A number $P \leq |\mathscr{N}|$ of legacy devices to be replaced with programmable switches;

Algorithm 5: Incremental deployment algorithm

Input: Long-term traffic statistics T, Network topology $\mathscr{G}$, Number of legacy devices P to be replaced
Output: Set of legacy devices $\mathscr{P}$ to be replaced

$max \leftarrow 0$, $\mathscr{P} \leftarrow \{\}$, $\hat{n} \leftarrow 0$;
$\hat{n}^{pre} \leftarrow 0$, $T^{pre} \leftarrow \{\}$, $key \leftarrow$ empty;
for *Each legacy device* $i \in \mathscr{N}$ *carrying traffic* T_i **do**
    $\hat{n} \leftarrow Hll\ (T_i)$;
    if $\hat{n} > max$ **then**
        $max \leftarrow \hat{n}$
        $key \leftarrow i$
$\mathscr{P}.add(key)$
if $P > 1$ **then**
    $\hat{n}^{pre} \leftarrow max$
    $T^{pre} \leftarrow T_{key}$
    while $\mathscr{P}.size() \leq P$ **do**
        for *Each switch* $i \in \mathscr{N} \setminus \mathscr{P}$ *carrying traffic* T_i **do**
            $\hat{n} \leftarrow Hll(T^{pre} \cup T_i)$
            if $\hat{n} > max$ **then**
                $max \leftarrow \hat{n}$
                $key \leftarrow i$
        $\mathscr{P}.add(key)$
        $\hat{n}^{pre} \leftarrow max$
        $T^{pre} \leftarrow T^{pre} \cup T_{key}$
return $\mathscr{P}$

Replace a subset of $\mathscr{P}$ (such that $P = |\mathscr{P}|$) of legacy devices with programmable switches with the goal of monitoring the highest number of distinct flows in the network and in an *incremental way*. This means that it must be assured that any subset of programmable switches $\mathscr{Z}$ (with $|\mathscr{Z}| \leq P$) that have been already deployed in the network as intermediate step, monitors the highest number of distinct flows as well.

4.4.3 Incremental deployment algorithm

As we mentioned above, HyperLogLog has good performance on estimating the distinct flows from an union of packet streams, so we use it to estimate the number of distinct flows passing through a set of legacy devices [26]. The pseudo code of our proposed algorithm is shown in Algorithm 5. To place the first programmable switch, we compute the estimated number of distinct flows $\hat{n}$ carried by each of the $i \in \mathscr{N}$ legacy devices using the HyperLogLog algorithm. The input

of HyperLogLog, for each device $i \in \mathcal{N}$, is T_i. For the replacement with a programmable switch, the algorithm selects the legacy device crossed by the highest number (*max*) of distinct flows (Lines 4-7). Such legacy device is added to $\mathcal{P}$. Once the first legacy device has been replaced, the principle to replace any other legacy device consists in progressively finding the one that, if replaced, allows to overall monitor the highest number of distinct flows in the network. The choice must be carefully taken, since different devices may be crossed by the same flows, and thus some flow information can be duplicated. To do so, we exploit the *union* property of HyperLogLog (Line 14), recalled in Section 4.2. As shown in Lines 10-21, the algorithm estimates the number of monitored distinct flows $\hat{n}^{pre}$ coming from the union of packet streams *(i)* of all the previously-upgraded programmable switches (T^{pre}) and *(ii)* of any legacy device $i \in \mathcal{N} \setminus \mathcal{P}$ still in the network (T_i). Then, the algorithm selects for replacement (and thus addition to set $\mathcal{P}$) the legacy device i leading to the largest number of monitored distinct flows overall (*max*). This operation is iterated until a number $P = |\mathcal{P}|$ of legacy switches has been replaced. Once the set $\mathcal{P}$ has been defined, the network operator can proceed with the physical replacement of the legacy devices with programmable switches, while ensuring interoperability in a hybrid environment [119]. Note that our incremental deployment algorithm focuses on the replacement of switches instead of new additions to the network. In fact, adding new switches may imply changes to routing and flow statistics alteration, making our algorithm ineffective.

4.5 Simulation results

Based on open source implementation of Count-Min Sketch [25], we implemented our incremental deployment algorithm and we simulated both our and Harrison's [69] network-wide heavy-hitter detection strategies in Python. In the following the simulation settings are reported.

4.5.1 Simulation settings and evaluation metrics

Traces and network topology

We divided 50 seconds 2018-passive CAIDA flow trace [3] into 10 time intervals. The programmable switches send reports to the controller when they detect potential heavy hitters at the end of any of those time intervals: in each time interval are transmitted around 2.3 million packets. As testing topology, we adopted a 100-nodes ISP backbone network [30]: adjacency matrix and plotted topology are available in [16]. A 32-bit cyclic redundancy check (CRC) [43] function was used to randomly assign each packet (characterized by a specific `(srcIP,dstIP)` pair) to a source/destination node couple in the network, and each packet is routed on the shortest path.

Tuning parameters

Unless otherwise specified, we set the simulation parameters as reported in Table 4.2. For HyperLogLog, we considered $m = 2^{12}$ because it leads to small-enough relative errors for our purposes (see Section 4.2). We then chose T_{int} and θ_H as per [93] and a Count-Min Sketch with size $N_h = 40 \times N_s = 10000$. Considering that each counter in Count-Min Sketch occupies 4 Bytes, the memory to be allocated for the sketch in each switch can be quantified as $10000 \cdot 40 \cdot 4B = 1.6MB$. Since the total memory of a real programmable switch chip (i.e., Barefoot Tofino [2]) is few tens of MB [109], it may be a reasonable size for a real large-scale ISP network scenario. However, in the following we will give a sensitivity analysis of monitoring performance with respect to N_s and N_h, analyzing what happens if more stringent memory requirements arise in the switch. Moreover, we chose $W = 3$ and $K = 10$ after a rigorous sensitivity analysis since, as will be shown later, these values lead, for the considered network topology, to the best trade-off between communication overhead, occupied memory and F1 score (i.e., they allow to overcome state-of-the-art performance for all the considered metrics).

Metrics

We set *recall* R and *precision* Pr as key metrics to evaluate our network-wide heavy-hitter detection strategy. They are defined in the following way:

$$R = \frac{Count_{HeavyHitters}^{detected/true}}{Count_{HeavyHitters}^{detected/true} + Count_{HeavyHitters}^{undetected/true}} \tag{4.1}$$

$$Pr = \frac{Count_{HeavyHitters}^{detected/true}}{Count_{HeavyHitters}^{detected/true} + Count_{HeavyHitters}^{detected/false}} \tag{4.2}$$

In our evaluations, we consider *F1 score* ($F1$) as compact metric taking into consideration both precision and recall, and measuring the accuracy of our strategy. It is defined as:

$$F1 = \frac{2 \cdot Pr \cdot R}{Pr + R} \tag{4.3}$$

Additionally, we consider each {flow key, packet count} pair as *unit* to evaluate both consumed *communication overhead* (when sent) and overall *occupied memory* (when stored in programmable switches). We consider as *unit* also flags sent by programmable switches to controller for local heavy-hitter notification[3]. All the reported results are the average value obtained in the considered 10 time intervals. We did not include the Count-Min Sketch size in the evaluated occupied memory, since it is constant: Table 4.5 gives an insight on Count-Min Sketch memory occupation for some sketch sizes, including $N_h = 40 \times N_s = 10000$.

[3]In real scenarios, each flag can be encoded by one bit, while {flow key, packet count} pairs require few bytes to be encoded. However, we consider both of them as a single *unit* for the purpose of easier quantification.

Table 4.3: Sensitivity to W in the case of full deployment ($K = 10$)

Evaluated metrics	SOTA	NWHHD+			
		W=1	W=3	W=5	W=20
F1 score	0.821	0.948	0.907	0.881	0.823
Communication overhead	71877	131707	60354	41076	13898
Occupied memory	760042	131608	60255	40977	13799

Table 4.4: Sensitivity to K in the case of full deployment ($W = 1$)

Evaluated metrics	SOTA	NWHHD+			
		K=1.2	K=10	K=20	K=100
F1 score	0.821	0.846	0.948	0.970	0.998
Communication overhead	71877	24760	131707	218370	570956
Occupied memory	760042	24661	131608	218264	570875

4.5.2 Evaluation of the network-wide heavy-hitter detection strategy in a full deployment scenario

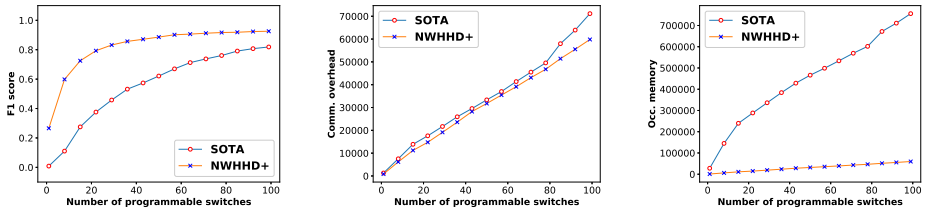
We evaluate NWHHD+ strategy against SOTA also in a *full deployment scenario*, i.e., when all the legacy devices have been replaced with programmable switches. In NWHHD+, we have introduced two parameters, i.e., ratio W and sampling rate K, which allow network operators to explore the trade-off between F1 score, communication overhead and memory occupation. Tables 4.3 and 4.4 show the sensitivity of NWHHD+ to W and K, and a performance comparison with SOTA. Note that, in the two tables, the columns related to $W = 1$ and $K = 10$ report results for the same settings we used in previous subsections on partial deployment, showing (as expected) a much higher F1 score and lower memory consumption than SOTA, but greater communication overhead. The tables also show that by properly tuning W and K it is possible to get a desired performance trade-off among F1 score, communication overhead and memory occupation.

As shown in Table 4.3, an increase of W leads to a decrease in the sample list size, which means that less {flow key, packet count} pairs are reported to the controller when the switches are polled (i.e., less communication overhead). Additionally, a decrease in the sample list size means that less memory is occupied in the switches. Intuitively, as side effect, the detection accuracy is affected (i.e., lower F1 score). Table 4.3 also shows that with $W = 20$, NWHHD+ and SOTA have comparable F1 score, but NWHHD+ leads to a significant reduction of both memory occupation and communication overhead.

Similar results can be obtained by properly tuning K (Table 4.4). An increase of K leads to a smaller sample list and, consequently, to less memory consumption.

Moreover, communication overhead is also reduced, because less information is sent when the switches are polled by the controller. Also in this case, with $K = 1.2$, NWHHD+ and SOTA have similar F1 score, but NWHHD+ considerably reduces communication overhead and memory occupation.

Note that such a tuning of W and K leads to analogous trends also in the case of partial deployment, but we omit a quantitative evaluation for the sake of conciseness.



(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Figure 4.5: Performance comparison of NWHHD+ with a state-of-the-art strategy [69]

4.5.3 Evaluation of the incremental deployment algorithm

We compare our *Incremental deployment algorithm*, where programmable switches are used for the detection of network-wide heavy hitters, with four existing algorithms: *Highest volume*, *Highest closeness*, *Highest betweenness* and *Random locations*. The Highest volume algorithm exploits long-term flow statistics to replace first the switches overall crossed by the largest number of packets. In the Highest closeness algorithm, the switches are ordered according to decreasing closeness, and in a partial deployment of P programmable switches only the top P switches in the list are replaced [101]. The Highest betweenness algorithm behaves in the same way, but betweenness of nodes [99] is evaluated instead of closeness. Both of these algorithms only depend on the network topology, and their underlying assumption is that nodes with highest centrality should be replaced first. Finally, the Random locations algorithm replaces P randomly-selected nodes: we average results over five randomized instances.

As shown in Fig. 4.5, which reports F1 score as a function of the number of deployed programmable switches, our Incremental deployment algorithm allows network operators to deploy a less number of programmable switches while ensuring the same F1 score of the other algorithms. It especially works well when a small number of programmable switches is deployed (i.e., for less than 50 switches), while it has comparable performance as the other algorithms when more than half programmable switches are deployed. This means that our strategy of first replacing switches that monitor the highest number of distinct flows effectively improves

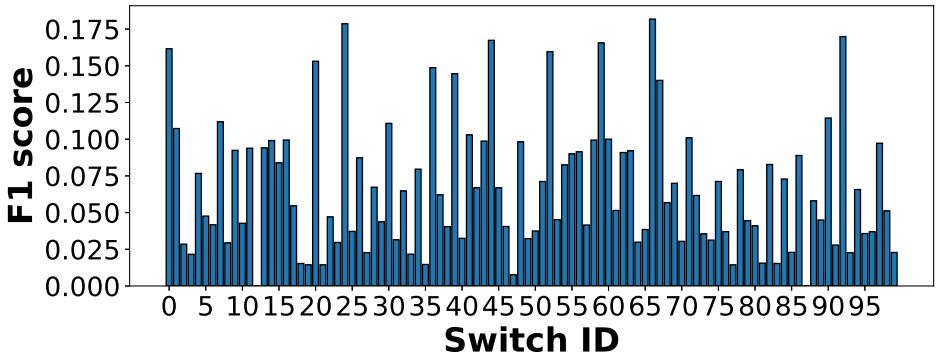
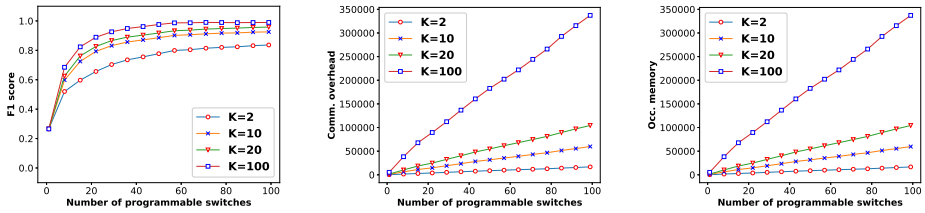


Figure 4.6: F1 score of NWHHD+ with single-programmable-switch deployment

flow visibility when it is inherently limited, also with respect to a strategy defined on the same long-term flow statistics (i.e., Highest volume algorithm). In fact, this latter strategy misses to consider that switches could carry many packets belonging to a multitude of small flows, and network-wide heavy hitters may remain undetected.



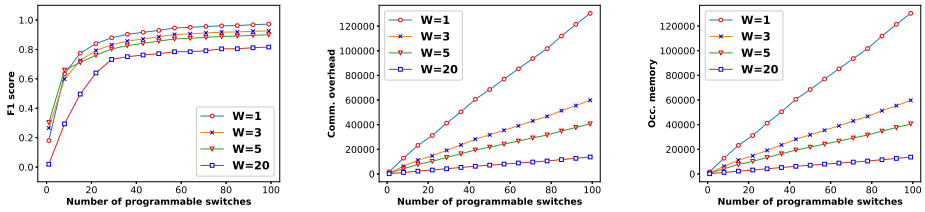
(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Figure 4.7: Sensitivity analysis of sampling rate K in NWHHD+ ($W = 3$)

4.5.4 Evaluation of the network-wide heavy-hitter detection strategy in an incremental deployment scenario

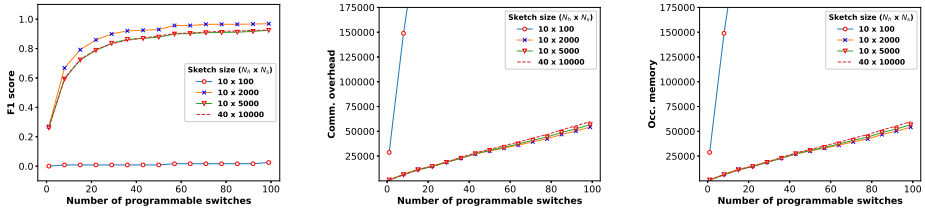
We compare the performance of our proposed network-wide heavy-hitter detection strategy, named *NWHHD+* for the sake of brevity, with the state-of-the-art strategy (called *SOTA* in the remainder of the section) proposed by Harrison et al. [69]. In order to fairly compare these two strategies, we set the global threshold for SOTA to $T_g = \overline{d}\theta_H|S|_{tot}$, where $\overline{d}$ is the average path length for the flows in all the time intervals, θ_H is fraction for heavy-hitter identification, and $|S|_{tot}$ is the whole network volume. Smoothing parameter is $\alpha = 0.8$ as per [69].

Figure 4.5(a) shows that NWHHD+, when deploying programmable switches using our incremental deployment algorithm, always leads to higher F1 score than



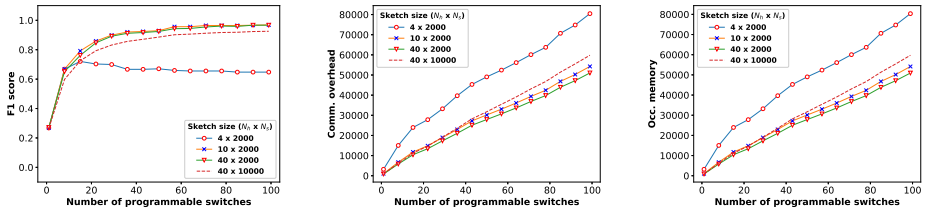
(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Figure 4.8: Sensitivity analysis of local ratio W in NWHHD+ ($K = 10$)



(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Figure 4.9: Sensitivity analysis of hash function output size N_s in NWHHD+ ($W = 3$ and $K = 10$)

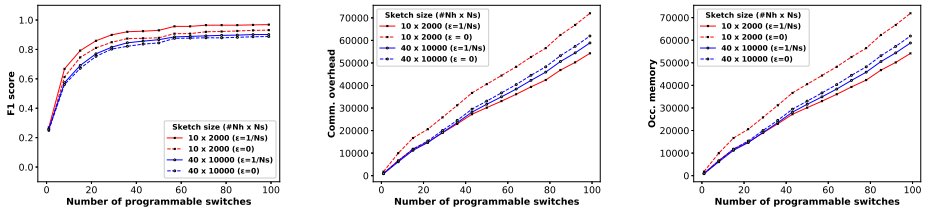


(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Figure 4.10: Sensitivity analysis of number of hash functions N_h in NWHHD+ ($W = 3$ and $K = 10$)

SOTA, especially when the number of programmable switches in the network is small. This means that NWHHD+ better exploits partial flow information provided by the programmable switches to detect the network-wide heavy hitters.

Figure 4.5(b) shows instead a comparison on the average-generated communication overhead. It clearly shows that NWHHD+ has a comparable communication overhead as SOTA, and NWHHD+ leads to less communication overhead than SOTA as the number of programmable switches increases. In NWHHD+, if at least one local heavy hitter is identified in a given time interval (as always hap-



(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Figure 4.11: Impact of ε in NWHHD+ ($W = 3$ and $K = 10$)

pens in our simulations), at the end of it the controller polls all the programmable switches to estimate the global network volume. Such an approach may cause high communication overhead, but by properly tuning W (as shown later) the local threshold can be increased, significantly reducing communication exchanges between switches and controller. Conversely, the SOTA strategy coarsely estimates the overall network volume at the controller and polls the programmable switches only if the estimated value is above the defined global threshold.

Figure 4.5(c) shows the average occupied memory by the two strategies. NWHHD+ outperforms SOTA, always occupying much less memory. This happens because NWHHD+ only stores *(i)* the sample list (and not all the {flow key, packet count} pairs, as SOTA does) and *(ii)* one local threshold for all the flows in each programmable switch (while SOTA stores per-flow local thresholds).

Note that unlike SOTA, as can be noticed from all the graphs of this section, NWHHD+ leads to very similar communication overhead and memory occupation results. In fact, communication overhead units in NWHHD+ are equal to {flow key, packet count} pair units stored in the switches sample lists, since such lists from all programmable switches are sent to the controller at the end of any time interval in which at least a local heavy hitter has been detected, plus the number of flags reported by programmable switches to controller to notify the existence of local heavy hitters. This means that the unit difference between communication overhead and memory occupation is at most the number of deployed programmable switches, and this small value cannot be noticed on the graphs.

Additionally, Fig. 4.6 recalls the simple test described in Section 4.4.1. What we report in the figure is F1 score for all the possible 100 deployments for the programmable switch when NWHHD+ is adopted. Compared to Fig. 4.3, we can see that the F1 score is generally higher in NWHHD+ (as already discussed), and that in most of the cases the peaks in F1 score correspond to IDs of switches that are crossed by a high number of distinct flows (see Fig. 4.4). This means that NWHHD+ better exploits the distinct flows information than SOTA. This can be even further proven by computing the normalized cross-correlation in $\tau = 0$ [115] between the number of distinct flows and F1 score, which is 13.11 for NWHHD+ and 6.57 for SOTA.

Table 4.5: Count-Min Sketch size and its memory occupation

$N_h \times N_s$	Memory occ.	$N_h \times N_s$	Memory occ.
40×10000	1.6MB	10×2000	80KB
10×100	4KB	10×5000	200KB
4×2000	32KB	40×2000	320KB

In order to show how the performance of NWHHD+ is sensitive to different tuning parameters, we ran simulations by varying the following parameters one at a time, while fixing the others to the values specified in Table 4.2.

Sensitivity to sampling rate K

Figure 4.7 shows the sensitivity analysis of NWHHD+ performance on sampling rate K. As shown in Figure 4.7(a), when K increases, F1 score increases as well, especially from $K = 10$ to $K = 100$. Figure 4.7(b) shows that also communication overhead significantly increases as K increases, as well as occupied memory (Figure 4.7(c)). The reason is that an increase of K leads to a smaller sampling threshold and thus to a bigger sample list to be stored and sent to the controller. In summary, choosing a larger K leads to a performance improvement on heavy-hitter detection, but implies an extra consumption of memory and communication resources. In our case, $K = 10$ is a good trade-off choice since it leads to comparable F1 score than choosing larger K while using much less memory and generating much less communication overhead.

Sensitivity to local ratio W

Figure 4.8 shows the sensitivity analysis of NWHHD+ performance on local ratio W. As shown in Figure 4.8(a), F1 score decreases as W increases, and the difference of F1 score is substantial between $W = 5$ and $W = 20$. Figure 4.8(b) and Figure 4.8(c) show that both communication overhead and occupied memory also significantly decrease when W increases. This happens because, opposite to K, an increase of W leads to a decrease in the sample list size, which means that less {flow key, packet count} pairs are reported to the controller when the switches are polled (i.e., less communication overhead) and less memory is also occupied. As side effect, the detection accuracy is also affected (i.e., lower F1 score). Thus, a smaller W enhances the performance on heavy-hitter detection, but more memory and communication overhead are required. According to the shown results, $W = 3$ leads to a good trade-off among F1 score, communication overhead and memory occupation.

Sensitivity to Count-Min Sketch hash function output size N_s

Figure 4.9 shows the sensitivity analysis of NWHHD+ performance on hash function output size N_s of Count-Min Sketch. For better understanding, we report the memory occupation of various Count-Min Sketches in Table 4.5. The blue cells recall the sketch size adopted in this chapter, unless otherwise specified. As shown in Figure 4.9(a), if the output size is too small (i.e., $N_s = 100$), Count-Min Sketch highly overestimates flows packet count and this causes very poor F1 score. By increasing the value of N_s, F1 score significantly increases until around $N_s = 2000$. However, after this value (i.e., for $N_s = 10000$) F1 score slightly decreases. The reason of this behavior is that $N_s = 2000$ leads to a slight overestimation of packet count for flows stored in sample lists that does not badly affect heavy-hitter detection. Conversely, such slight overestimation compensates missing packet counts of small flows that are not stored in those lists, leading to an estimated whole network volume S_{tot} very close to the real value. This effect makes the controller able to identify network-wide heavy hitters with a more accurate global threshold, thus leading to high F1 score.

Figures 4.9(b) and 4.9(c) show that the overestimated packet count due to too small output size also causes very large-sized and badly-estimated sample lists and consequently leads to high and unnecessary occupied memory and communication overhead. While increasing N_s, communication overhead and memory occupation decrease until a minimum is reached at around $N_s = 2000$ then, for larger N_s, they marginally increase. This slight increase is due to the value $\varepsilon = 1/N_s$ used in Algorithm 3 for bias compensation in sampling and local thresholds (see Section 4.3.2). The effectiveness of introducing ε in our strategy is shown in Figure 4.11: considering ε in the computation of sampling and local thresholds not only increases F1 score, but also decreases communication overhead and memory occupation. On the other hand, considering $\varepsilon = 1/N_s$ also means that N_s directly affects the value of sampling threshold $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$ and local threshold $(W\theta_H+\varepsilon)|S_i|$. If the output size N_s is larger, ε is smaller: this leads to smaller sampling and local thresholds. A smaller sampling threshold generates a larger sample list to be stored and sent to the controller (i.e., higher communication overhead and occupied memory). Conversely, having a smaller N_s leads to less communication overhead and occupied memory, and this explains why lower N_s (but not too small, where overestimation dominates) generates lower communication overhead and requires less memory than sketches with larger output size.

Sensitivity to number of Count-Min Sketch hash functions N_h

Figure 4.10 shows the sensitivity analysis of NWHHD+ performance on number of hash functions N_h of Count-Min Sketch. As shown in Figure 4.10(a), F1 score increases as the number of hash functions N_h increases. Nevertheless, when N_h is large enough to correctly estimating flow packet counts, further increasing it does not improve heavy-hitter detection performance anymore. Figures 4.10(b) and

4.10(c) show that communication overhead and memory occupation decrease as the number of hash function N_h increases. Especially, if N_h is too small, wrongly-estimated flow packet counts lead to high communication overhead and memory consumption. Finally, note that sketch sizes of 10×2000 and 40×2000 both lead to better results than sketch size of 40×10000. This happens because, as shown previously, a too large N_s has bad impact on NWHHD+ performance: results thus show that NWHDD+ is more sensitive to variations to N_s than to N_h.

4.6 Evaluation in emulated P4 environment

Based on an open source P4 implementation of Count-Min Sketch [14], we implemented both our network-wide heavy-hitter detection (i.e., NWHHD+) and Harrison's (i.e., SOTA) strategies in P4 language and tested them. In the following, we report some details on the emulated network environment.

4.6.1 Environment settings and evaluation metrics

Emulated network environment

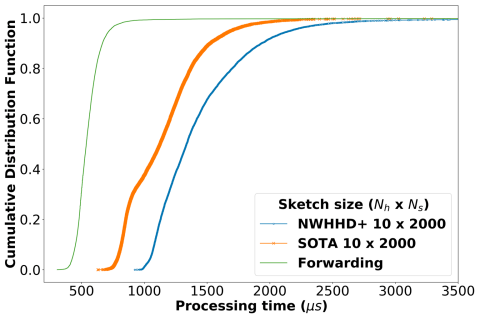
We chose Mininet [8] as our emulated network environment for the deployment of P4 switches implementing the network-wide heavy-hitter strategies. The P4 code is compiled by p4c compiler [19] into a JSON file that describes the behavior of P4 switch (i.e., parser, tables and actions in the P4 pipeline). Then, the JSON file is loaded by any P4 switch created according to the behavioral model [13]. Finally, a topology in Mininet is created connecting such behavior-defined P4 switches. The adopted topology is composed by three interconnected switches, and there is a host connected to each switch. All the packets are forwarded from source host to destination host on the shortest path. We did not consider a larger topology for scalability reasons, since all the P4 switches are emulated on a virtual machine deployed by OpenStack on our testbed with dedicated access to 4×2.7GHz CPU cores and to 4GB of RAM. However, given the nature of the performed tests, as we will show later, this does not represent a limitation. The controller is implemented in Python as we explained in Section 4.3.4.

Tuning parameters

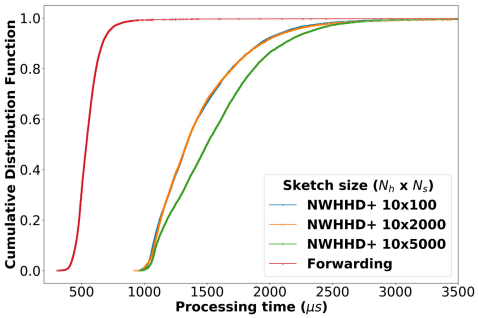
We used the same settings as our simulations in Python shown in Table 4.2, unless otherwise specified. Additionally, we set the size of register $\mathsf{S_i}$ to 1 and all the remaining registers but Count-Min Sketch (e.g., sample list) to 100. Count-Min Sketch registers size is set to N_s.

Metrics

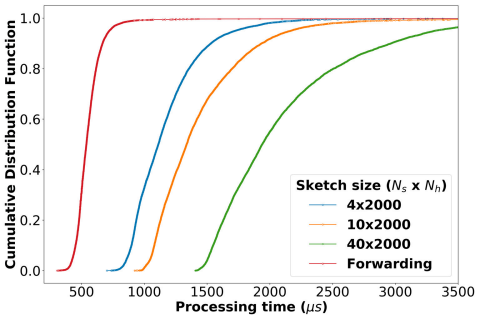
We evaluate NWHHD+ performance in terms of *packet processing time*. We believe that packet processing time is an important metric to evaluate, since it dis-



(a) Comparison with SOTA and simple forwarding



(b) Sensitivity to output size of hash functions N_s



(c) Sensitivity to number of hash functions N_h

Figure 4.12: Cumulative distribution function of packet processing time for NWHDD+ (10000 packets)

closes whether the algorithm implemented in the P4 pipeline performs well or not and whether it can be used for line-rate transmissions. We used Wireshark to capture the time when each packet arrives at an input interface of a P4 switch and the time when it is forwarded by an output interface of the same switch. The packet processing time is defined as the difference between such times and estimates the time spent by the packet in the P4 switch pipeline. We also implemented a simple *forwarding* strategy, where no heavy-hitter detection (neither SOTA nor NWHHD+) is performed and the packet is just forwarded to the right output inter-

face.

As additional metric, we also evaluate the *controller response time*. This metric represents the time overhead generated in the interaction between data plane and control plane for the identification of heavy hitters.To measure such response time, we captured the following timestamps at the controller: *(i)* the timestamp related to the first true-flag packet that arrives at the controller, meaning that at least one potential heavy-hitter exists in the network and *(ii)* the timestamp when any network-wide heavy hitter has been detected (if it exists). The response time is then defined as the difference between the latter and the former timestamps.

4.6.2 Evaluation of packet processing time

Figure 4.12 reports the cumulative distribution function of packet processing time measured for 10000 generated packets. As shown in Figure 4.12(a), both NWHHD+ and SOTA strategies lead to more packet processing time than simple forwarding, since more operations need to be performed on the packet. However, 50% of the packets can be processed within 1500 μs in the switch when the Count-Min Sketch size is set to $N_h \times N_s = 10 \times 2000$. Since our strategy has more read and write actions in the additional registers (e.g., sample list) than SOTA, SOTA leads to slightly lower processing times. Increasing the output size N_s (up to a certain threshold) and the number of hash functions N_h can improve F1 score for heavy-hitter detection as shown in Figures 4.9 and 4.10, but this also has some impact on packet processing time in P4 switches. Figure 4.12(b) shows how N_s affects packet processing time: it slightly increases as N_s increases significantly. This happens because a higher N_s requires a hash function performing more lookups to obtain the output value. Figure 4.12(c) shows instead the impact of N_h: results clearly show that increasing the number of hash functions N_h has a more impacting effect on packet processing time than increasing the output size of hash functions N_s.

These evaluations confirm that the size of Count-Min Sketches implemented in the data plane must be carefully defined. In fact, an increase in N_h improves monitoring performance (see Figure 4.10) but requires larger packet processing time. Moreover, by also referring to Figure 4.9, correctly dimensioning N_s is of paramount importance to avoid both large packet processing time and F1 score reduction. Finally, note also that packet processing times shown in this section (i.e., in the order of few ms) include the time needed to cross several virtualized layers in the single-node emulated environment. In real carrier-grade hardware (e.g. Barefoot Tofino, with throughput in the order of 6.5 Tb/s or more [2]), packet processing time is expected to be several orders of magnitude lower (i.e., in the order of ns or few μs).

4.6.3 Evaluation of the controller response time

For each different time interval size, we measured the response time in 10 intervals and computed the average response time. Table 4.6 shows the average response

Table 4.6: Average controller response time

Time interval T_{int}	5s	10s	15s	20s
Response time	1.4488s	1.4542s	1.4656s	1.5036s

time. It is around 1.5 s and slightly increases while increasing T_{int}, since more data needs to be processed for longer time intervals. These results mean that network-wide heavy hitters are detected in average 1.5s after that the controller receives the first flag from any switch identifying a potential heavy hitters. Note that this time depends on the computational capacity of the controller, so we expect it to be even smaller while adopting carrier-grade hardware for the controller in real deployments. Moreover, we do not include in the evaluated response time the retrieval time of the flag messages from the programmable switches, which strongly depends on where the controller is placed and is at most in the order of few tens of ms [112] (i.e., negligible with respect to the controller response time).

4.7 Related work

4.7.1 Network-wide heavy-hitter detection in programmable data planes

In the last years, many strategies have been proposed to monitor heavy hitters directly in the data plane by exploiting the flexibility of programmable switches. Some among them are OpenSketch [134], UnivMon [93], Elastic Skecth [132], FlowRadar [89] , SketchVisor [74], NitroSketch [92], SketchLearn [76] and HashPipe [113]. However, they only focused on heavy-hitter detection at a single SDN switch, but this is not enough for heavy-hitter detection in large networks, since some heavy hitters may be undetected or wrongly detected by relying on limited information at a single location.

Thus, the concept of *network-wide heavy hitter* has been introduced in literature [49][95][133][75]. A network-wide heavy hitter uses distributed information, which can be made available by programmable switches, to accurately and effectively monitor heavy hitters from a global perspective. Harrison et al. [69] and Basat et al. [36] have proposed two different strategies to monitor network-wide heavy hitters. In Harrison's strategy [69], at the end of each time interval, if any heavy hitter has been detected through a local threshold-based mechanism in P4-enabled switches, the controller polls the programmable switches and uses a different (global) threshold-based mechanism to decide whether local heavy hitters are network-wide heavy hitters or not. However, in their strategy, packets belonging to the same flow are counted multiple times by different switches, and this duplicated information is not discarded by the controller while estimating network-wide heavy hitters: for this reason, it is very difficult to correctly set the global threshold. Basat's work [36] provides a solid method for network-wide heavy-hitter detection by using a data streaming model, but the introduced communication overhead and

occupied memory are significant. Another key limitation is that the hash functions needed by their strategy do not exist in practice. Our network-wide heavy-hitter detection strategy is similar to the one proposed by Harrison et al., but we define new and more intuitive local and global thresholds and we exploit information on distinct flows to prevent duplicate counting of packets, thus reducing the communication overhead and occupied memory in programmable switches and improving monitoring performance.

4.7.2 Partial deployment of SDN solutions in ISP networks

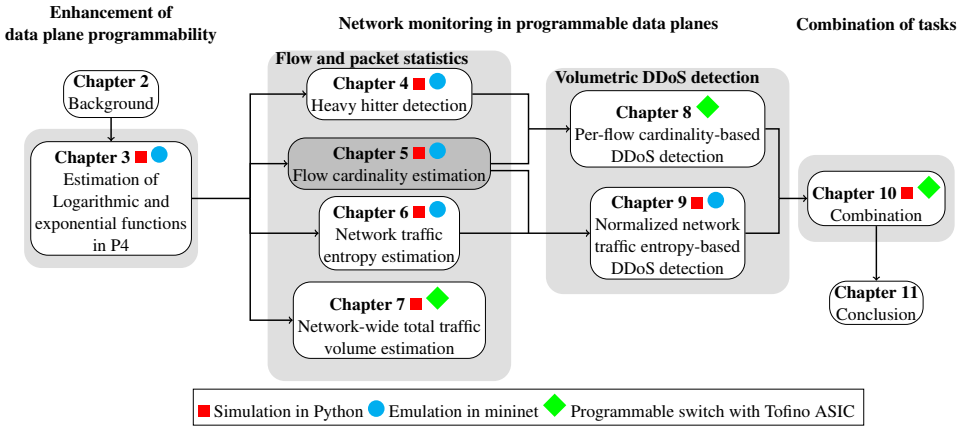
The appearance of SDN simplifies the network management and enhances the flexibility of the network. However, currently it is not feasible to upgrade all legacy switches to SDN switches due to the limitation of budgets and operational burdens, so the current trend for network operators is to deploy a limited number of SDN switches and make the network best work in a hybrid environment. A good strategy for partial SDN deployment is thus needed to cost-effectively bring benefits to ISPs. Unfortunately, obtaining the best partial deployment of SDN switches is a NP-hard problem [72]. In literature, most of the works focus on the problem of partial deployment of OpenFlow switches [97] in legacy infrastructures, and either Integer Linear Programming (ILP) [96] or incremental deployment heuristic algorithms [72][129][73][87] have been adopted to solve such problem, focusing on interoperability and routing issues in a hybrid environment while achieving the best load balancing or maximizing the throughput. Incremental deployment heuristic strategies are a good approach to solve the problem of partial deployment, since they aim at iteratively replacing legacy equipment by ensuring local optimal performance. However, the previous work neither takes into account the problem of incremental deployment of programmable switches in a legacy infrastructure to improve network monitoring performance nor proposes a solution for topological placement of programmable switches: with our chapter, we try to fill this gap. Moreover, our solution is inter-operable with other techniques: the raw data gathered from legacy devices could be used with filtered data collected from programmable switches for an improved network monitoring.

4.8 Concluding remarks

In this chapter, we proposed a novel network-wide heavy-hitter detection strategy incorporating programmable switches and controller in Software-Defined Networks. This strategy has been implemented in P4 language and tested in an emulated environment. We also presented a new greedy algorithm suitable for an effective incremental deployment of SDN programmable switches in legacy infrastructures that aims at monitoring as many distinct network flows as possible. This algorithm best supports monitoring tasks such as heavy-hitter detection when only a limited number of legacy devices can be replaced with programmable switches.

Both the network-wide heavy-hitter detection strategy and the incremental deployment algorithm were proven to outperform existing approaches. By adopting our incremental deployment algorithm, network operators can ensure very good monitoring performance by replacing less than half of the legacy devices in the network. Moreover, our network-wide heavy-hitter detection strategy outperforms an existing approach both when only a limited number of programmable switches is deployed and when the network is entirely upgraded, since it allows network operators to strike a balance between heavy-hitter detection accuracy, communication overhead and occupied memory. As marginal side effect, our strategy has been shown to lead to slightly more packet processing time in the execution of the P4 pipeline than the considered state-of-the-art approach. Our heavy-hitter detection can be built in conjunction with per-flow cardinality-based DDoS detection presented in Chapter 8 for detection DDoS victims under a large traffic volume attack.

5 Flow cardinality estimation



In this chapter we study the flow cardinality estimation in programmable data planes, which can be used to monitor the number of active hosts and distinct flows in the network. The monitored results can be further used to detect network anomalies, such as DDoS attacks and port scans. We put an emphasis on how number of distinct flows is estimated in case of a large packet stream in ISP networks.

This chapter is based on the paper "Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4" submitted to IEEE Transactions on Dependable and Secure Computing (Under review).

5.1 Introduction

Counting the number of different connections is basically identifying the number of distinct elements (connections) in a set (packets) which has been widely studied. This problem is also known as *flow cardinality* estimation and many algorithms have been proposed over the years. As the number of packets is usually very large

in the network, programmable switches need a compact data structure to count the incoming packets to efficiently estimate the number of flows.

To this aim, based on P4-based solutions for the estimation of logarithm (P4Log) and exponential function (P4Exp) in Chapter 3, we here propose *P4LogLog*, a novel memory-efficient strategy that takes inspiration from LogLog algorithm [57] for the estimation of flow cardinality in P4. P4LogLog can guarantee high accuracy while ensuring small memory usage.

The main contributions of this chapter include:

- We design a new cardinality estimation approach, named *P4LogLOg*, which is built on top of P4Log and P4Exp algorithms presented in Chapter 3
- We implement the prototype of P4LogLog in P4 behavioral model [13], and such a monitoring task can act as an in-network monitoring task in the programmable switches
- The results show that P4LogLog can achieve high accuracy of large packet streams in the network while ensuring small memory usage.

5.2 Basic knowledge and used compact data structure

5.2.1 LogLog algorithm for flow cardinality estimation

LogLog [57] is a sketch-based algorithm that can be adopted to estimate the number of distinct flows crossing a switch. In brief, it works as follows. Given an incoming packet with *flow key i*, LogLog applies to i a hash function with output size os: the resulted os-bit binary string s is denoted by $s = \{s_{os-1}s_{os-2}\cdots s_0\}$. LogLog then updates an m-sized LogLog register Reg. Let $bucket$ be the rightmost k bits of s (with $k = \log_2 m$) and x the remaining bits, i.e., $bucket = \{s_{k-1}\cdots s_0\}$ and $x = \{s_{os-1}\cdots s_k\}$. Reg is updated following this rule: $Reg[bucket] = max(Reg[bucket], value)$, where $value$ is the index of the rightmost 1 of x plus one. For instance, x is 1000, then $value$ is 4. Reg can then be queried to estimate the flow cardinality $\hat{n}$, which is computed as $\hat{n} = \alpha_m m 2^{\frac{1}{m}\sum_{bucket=0}^{m-1} Reg[bucket]}$, where α_m is a bias correction parameter. An interesting property of LogLog is that multiple LogLog sketches can be merged to a single sketch, which can be used to count the flow cardinality of the union of many packet streams.

5.2.2 Hamming weight computation for LogLog

Hamming weight represents the number of non-zero values in a string. In a binary string, the Hamming weight indicates the overall number of ones. For example, given the binary string 01101, the Hamming weight is 3. It can be computed by means of different algorithms: as part of P4LogLog, in this chapter we adopt the *Counting 1-Bits* algorithm presented in [123], as it only relies on bitwise operations that are completely supported by P4 language [17].

5.3 Flow cardinality estimation: P4LogLog

In this section, we propose P4LogLog for the estimation of flow cardinality. The problem is formulated as follows:

Problem definition: *Given* a stream S of incoming packets, each one belonging to a specific *flow* i, *returns* the estimated *flow cardinality* $\hat{n}$ of S, i.e., the estimated number of distinct flows in S.

For instance, if we identify as *flow key* i each packet *destination IP*, meaning that a flow includes all the packets towards a specific destination. Then the flow cardinality of destination IPs represents the number of destination IPs in the network. Same consideration would hold for any other flow definition (e.g. packets with the same 5-tuple, same source/destination IP pair, etc.) without any loss of generality. In the following, we report the details of Update and Query operations of P4LogLog, which both follow specifications from LogLog [57] (see Section 5.2.1) while only using P4-supported instructions. The P4 source codes are available in [24].

Update

As shown in Algorithm 6, *Update* function iteratively updates a readable and writable stateful register *Reg* for each incoming packet, which belongs to a flow with flow key i. The flow key i of the packet is hashed by a given hash function, and the output value is converted to a os-bit binary string s (Line 6). In this chapter, we consider $os = 32$ and an m-sized register *Reg*, where $m = 2^k$ and integer $k \in \{4, ..., 16\}$ (as per [57]). The index of the register's cell to be updated, named *bucket* ($0 \leq bucket \leq m-1$), is the binary number represented by the rightmost k bits of s, which can be obtained by $s\&(2^k-1)$, i.e., $s\&0\underbrace{11\cdots1}_{k}$ (Line 7). & is the bitwise inclusive AND operator and 2^k-1 (in binary) is pre-stored in the P4 program once k is chosen. The algorithm then right-shifts s to k bits to get a binary string x where the first k bits are 0s and the remaining $os-k$ bits are the first $os-k$ bits of s (Line 8). The index of rightmost 1 in x, called *value*, is then used to update the LogLog register's cell in *bucket* position. Unfortunately, retrieving such rightmost 1 is not trivial. As shown from Lines 9 to 12, the algorithm adopts the following strategy: all bits of x on the left of the rightmost 1 are iteratively converted to 1, and the result of this iterative operation is stored in w ($|$ is the bitwise inclusive OR operator). For example, an os-bit binary value $x = \underbrace{00\cdots01}_{os-1}0$ is converted to $w = \underbrace{11\cdots11}_{os-1}0$. The algorithm for *Hamming weight* recalled in Section 5.2.2 is then used to count b, i.e., the number of 1s in x (Line 12): *value* is equal to $os+1-b$ (Line 13). Finally, if *value* is larger than *bucket*-indexed value in the register, *value* replaces the stored value (Lines 14-15).

Algorithm 6: P4LogLog

Input: Packet stream S
Output: Flow cardinality estimation $\hat{n}$

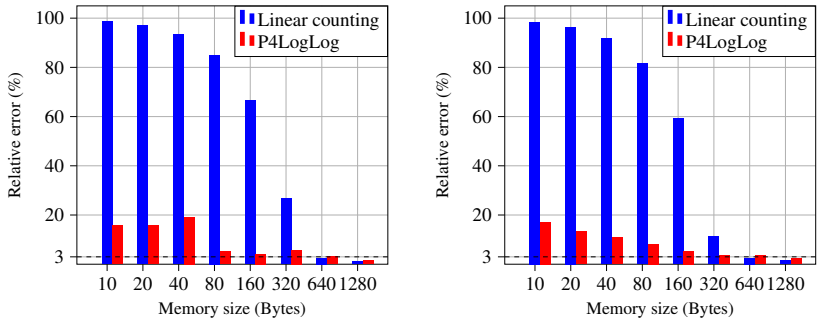
1 $m \leftarrow 2^k$ $(k \in \{4, ..., 16\})$
2 $os \leftarrow 32$
3 $Reg \leftarrow m$-sized empty LogLog register
4 **Function** $Update(Reg)$**:**
5 **for** *Each received packet belonging to flow* i **do**
6 $s \leftarrow (Hash(i) \rightarrow \{0,1\}^{os})$
7 $bucket \leftarrow s\&(2^k - 1)$
8 $x \leftarrow (s \gg k)$
9 $w \leftarrow x|(x \ll 1)$
10 **for** *int* $l \in \{1, \cdots, \log_2(os) - 1\}$ **do**
11 $w \leftarrow w|(w \ll 2^l)$
12 $b \leftarrow HammingWeight(w)$
13 $value \leftarrow os + 1 - b$
14 **if** $value > Reg[bucket]$ **then**
15 $Reg[bucket] \leftarrow value$
16 **return** Reg
17 $\alpha_m \leftarrow 0.39701 \ll 10$
18 **Function** $Query(Reg)$**:**
19 $exp \leftarrow P4Exp((\sum_{bucket=0}^{m-1} Reg[bucket]) \gg k)$
20 $\hat{n} \leftarrow (exp \cdot \alpha_m \cdot m) \gg 10$
21 **return** $\hat{n}$

Query

Query function in Algorithm 6 estimates the flow cardinality directly in the switch. The flow cardinality estimation $\hat{n}$ is computed as in [57] and Section 5.2.1 from all LogLog register's stored values by exploiting P4Exp. The k-bit right-shift operation carried out on the sum of values from Reg is equivalent to dividing such sum by $m = 2^k$ (Line 19). The floating parameter α_m, chosen as in [57], is amplified 2^{10} times through left shift operation, and the resulted value from the computation executed in Line 20 is right-shifted 10 bits to get the estimated flow cardinality $\hat{n}$.

5.4 P4LogLog evaluation

We implemented P4LogLog in Python and simulated them for evaluation. The results are reported in this section.



(a) Flow key: source IP (b) Flow key: destination IP

Figure 5.1: Performance comparison of P4LogLog with an existing flow cardinality estimation approach [134]

5.4.1 Evaluation metrics and simulation settings

Testing flow trace and methodology

P4LogLog: We use 2018-passive CAIDA flow trace [3], including 50 seconds of network traffic, and divide it into 50 1-second time intervals (or observation windows). In each considered time interval there are around 460K packets.

Evaluated metrics

We consider *relative error* as an evaluation metric.
P4LogLog: Being n the exact number of distinct flows (either identified by *source IP* or *destination IP* as flow key) in a time interval and $\hat{n}$ its estimated value, the relative error is defined as the average value of $\frac{|n-\hat{n}|}{n} \cdot 100\%$ in all the consecutive 50 time intervals.

Tuning parameters

The default tuning parameters for P4Log and P4Exp, adopted for both P4LogLog, are set as in Tab. 3.3 of Chapter 3. The sketch (either Count-min or Count Sketch) used by P4NEntropy has size $N_h = 5 \times N_s = 2000$.

5.4.2 Evaluation of P4LogLog

As shown in Fig. 5.1, we compare our P4LogLog with another existing flow cardinality estimator (*Linear counting* [134]), implementable in a programmable data planes, in terms of relative error. 1 bit is used for each Linear counting register cell [124], while 5 bits are allocated for each P4LogLog register cell [57]. Given this,

we vary the *memory size* of each register for both the approaches (i.e., we vary the number of cells in the registers, which can be easily retrieved).

Figure 5.1(a) focuses on the estimation of distinct source IPs in the trace. The relative error on such flow cardinality estimation by adopting Linear counting is 50% higher than by adopting P4LogLog when the memory size is below 320 bytes, and its value for Linear counting is high for any memory size below 640 bytes. Conversely, our P4LogLog leads to acceptable relative errors with only 80 bytes. If we assign 1280-bytes registers to P4LogLog and Linear counting, the relative error of both is around 1%. Likewise, Fig. 5.1(b) shows the estimated number of distinct destination IPs in the trace. Our P4LogLog algorithm still outperforms Linear counting for small memory sizes. When the memory occupation reaches 640 bytes, the relative error of P4LogLog is below 3%, which is assumed as acceptable target.

Another solution for flow cardinality estimation is proposed in [93]. However such a solution always needs much more memory (i.e., at least 0.2MB) than Linear Counting and P4LogLog to get reasonable accuracy.

5.5 Related work

5.5.1 Flow cardinality estimation for network monitoring

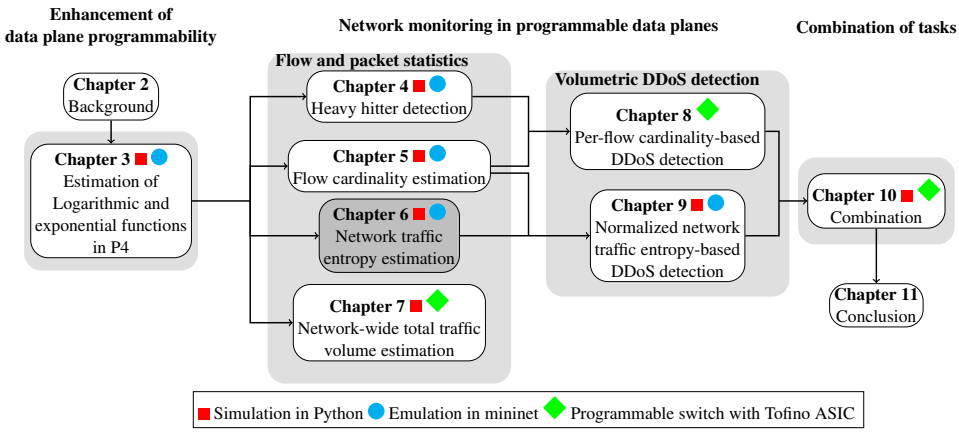
Many cardinality-estimation algorithms have been implemented to be executed in programmable data planes for the purpose of network monitoring [134][132][93], often based on linear counting [124]. However, all of them are able to only perform the update operation directly in the data plane, while the query operation has still to be executed by the controller. This is because programmable switches do not support arithmetic operations such as logarithm and exponential function computation, which are needed for flow cardinality estimation. Conversely, by leveraging our proposed strategies for logarithm and exponential-function estimation in the data plane, named P4Log and P4Exp [55], we developed P4LogLog, a flow cardinality estimation algorithm that takes inspiration from LogLog [57]. P4LogLog enables a flow cardinality estimation entirely in programmable switches, where both update and query operations can be executed in the data plane. Moreover, our P4LogLog can estimate cardinality with high accuracy while consuming less memory than existing approaches. Note that HyperLogLog [63] has higher theoretical accuracy than LogLog, but it is currently not implementable in P4 language due to the computation of harmonic mean.

5.6 Concluding remarks

In this chapter, relying on recently-proposed logarithmic and exponential function estimation solutions, we presented P4LogLog to estimate the number of distinct flows in the network by only using P4-supported operations. We also evaluated our proposed approach and compared it with state-of-the-art solutions. Results show

that P4LogLog has better accuracy than the state of the art especially when memory availability is small (i.e., smaller than 640 Bytes). The flow cardinality estimation is necessary for both Chapter 8 *per-cardinality-based DDoS detection* and Chapter 9 *normalized network traffic entropy-based DDoS detection*.

6 Network traffic entropy estimation



For the purpose of management, network operators need to constantly monitor the status of the network and ensure that it behaves as intended. Network traffic distribution *is an important indicator to understand the network behavior: the most widely-used metric to evaluate traffic distribution is* entropy. *In this chapter, we leverage the logarithmic and exponential function estimation proposed in Chapter 3 to estimate traffic entropy entirely in the switch data plane.*

This chapter is based on the previously published paper "Damu Ding, Marco Savi, and Domenico Siracusa. "Estimating logarithmic and exponential functions to track network traffic entropy in P4." NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium. IEEE, 2020." [55]

6.1 Introduction

The *entropy* of flow size indicates the network traffic distribution. A periodical tracking of this metric helps diagnose performance and security issues, by supporting the execution of tasks such as congestion control [78], load balancing [125],

port-scan detection [67][40], distributed denial-of-service (DDoS) attacks detection [85][94] and worm detection [120].

As our network traffic entropy estimation strategy relies on the calculation of logarithm and division, we first used *P4Log* proposed in Chapter 3 to estimate the logarithm. Moreover, since division operation $\frac{A}{B}$ can be expressed as $2^{\log_2 A - \log_2 B}$, we then adopted P4Exp in conjunction with P4Log for the computation of divisions. Based on these two algorithms, we then present a novel strategy, named *P4Entropy*, to disclose network traffic distribution by leveraging *Shannon entropy* [110] computation. A prototype of P4Entropy has been implemented in P4 behavioral model [13] and has been proven to be fully executable in a P4 emulated environment.

We then evaluate P4Entropy by means of simulations to show their effectiveness and their sensitivity to different tuning parameters. Our P4Entropy algorithm overcomes the limitation of the state-of-the-art benchmark strategy: while the existing strategy needs to set a fixed observation window on the number of processed packets to compute network traffic entropy, our approach allows to set as observation window any time interval, regardless of the number of processed packets. This is useful in the case estimated entropy values from multiple switches must be sent to the collector in a synchronized way.

The main contributions of this chapter are as follows:

- Based on proposed P4Log and P4Exp algorithms in Chapter 3, we present a new network traffic entropy estimation strategy, named *P4Entropy*, in P4-programmable data plane
- We implement the prototype of P4Entropy in P4 behavioral model [13], and such a monitoring task can be entirely executed in the programmable switches without constraining the number of packets in a given observation interval.
- The results show that the required memory of our algorithms in the switches is much smaller than that of the state-of-the-art solution while reaching the same accuracy.

6.2 Basic knowledge and used compact data structure

6.2.1 Network traffic entropy

Network traffic entropy [84] gives an indication on traffic distribution across the network. Each network switch can evaluate the traffic entropy related to the network flows that cross it in a given time interval T_{int}. Relying on the definition of *Shannon entropy* [110], network traffic entropy can be defined as $H = -\sum_{i=1}^{n} \frac{f_i}{|S|_{tot}} \log_d \frac{f_i}{|S|_{tot}}$, where f_i is the packet count of the incoming flow i, $|S|_{tot}$ is the total number of processed packets by the switch during T_{int}, n is the overall number of distinct flows

and d is the base of logarithm. Traffic entropy reaches $H = 0$ when in T_{int} all packets $|S|_{tot}$ belong to the same flow i, while it reaches its maximum value $H = \log_d n$ when each of the n flows i transports only one packet.

6.2.2 Sketch-based estimation of flow packet count

Estimating the number of packets for a specific flow crossing a programmable switch (f_i) is fundamental for network traffic entropy computation. Such an estimation can be performed by means of *sketches* [74], which are probabilistic data structures associated to a set of pairwise-independent hash functions. The *size* of each sketch data structure depends on the number of associated hash functions N_h and on the output size of each function N_s, and is $N_h \times N_s$. *Update* and *Query* operations are used to store and retrieve information from the sketch: Update operation is responsible for updating the sketch to keep track of flow packet counts, while Query operation retrieves the estimated number of packets for a specific flow. Two well-known algorithms to Update and Query sketches are *Count-min Sketch* [48] and *Count Sketch* [44]. A detailed theoretical analysis on the accuracy/memory occupation trade-off for these sketching algorithms is reported in [48][44]. From a high-level perspective, as any of N_h and N_s increase, memory consumption is larger but estimation is more accurate. Count Sketch leads to a better accuracy/memory consumption trade-off than Count-min Sketch, but its update time is twice slower [47].

6.3 Network traffic entropy estimation

Based on proposed P4Log and P4Exp algorithms in Chapter 3, we propose a new strategy, named *P4Entropy*, to estimate the network traffic entropy entirely in the programmable switches' data plane. The prototype of P4Entropy has been implemented in P4 behavioral model [13] and is executable in an emulated environment as Mininet [8]. The source code is available in [21]. Formally, the problem is defined as follows.

Problem definition: Given a stream of incoming packets S in a switch and a time interval T_{int}, returns *Shannon entropy* estimation (see Section 6.2.1) at the end of T_{int}.

6.3.1 Derivation of estimated entropy in P4

The goal of this section is to provide an estimation of network traffic entropy by only using P4-supported arithmetic operations and reducing as much as possible their number. The section also shows how relevant statistics, used for entropy estimation at the end of T_{int}, are iteratively updated every time a packet crosses the switch.

We first rewrite the Shannon entropy in the following way:

$$H(|S|_{tot}) = -\sum_{i=1}^{n} \frac{f_i(|S|_{tot})}{|S|_{tot}} \log_d \frac{f_i(|S|_{tot})}{|S|_{tot}}$$
$$= \log_d |S|_{tot} - \frac{1}{|S|_{tot}} \sum_{i=1}^{n} f_i(|S|_{tot}) \log_d f_i(|S|_{tot})$$

We consider $d = 2$ without any loss of generality. With respect to the definition given in Section 6.2.1, we use the notation $f_i(|S|_{tot})$ to make explicit that f_i refers to its value when $|S|_{tot}$ packets have been received (i.e., at the end of T_{int}). As packets arrive in the switch, the overall number of processed packets $|S|$ increases and must be stored in the switch to ensure that $H(|S|_{tot})$ can be computed at the end of T_{int}, when $|S| = |S|_{tot}$. We define $Sum(|S|) = \sum_{i=1}^{n} f_i(|S|) \log_d f_i(|S|)$, which must be updated as well. To understand how to update $Sum(|S|)$, let's assume that a new packet for a specific flow arrives and is the $|S|$-th packet. We call its packet count $\bar{f}_i(|S|)$. It holds that:

$$\begin{cases} f_i(|S|) = f_i(|S|-1) & (f_i(|S|) \neq \bar{f}_i(|S|)) \\ f_i(|S|) = f_i(|S|-1)+1 & (f_i(|S|) = \bar{f}_i(|S|)) \end{cases}$$

This allows us to re-write $Sum(|S|)$ in the following way:

$$Sum(|S|) = Sum(|S|-1) + \bar{f}_i(|S|)\log_2 \bar{f}_i(|S|) + $$
$$- (\bar{f}_i(|S|)-1)\log_2(\bar{f}_i(|S|)-1)$$

$Sum(|S|)$ thus needs two logarithmic computations for each incoming packet, and would require running P4Log twice with corresponding computational effort.

In the next step, we show how it is possible to estimate $Sum(|S|)$ with only (at most) one logarithmic computation. When $\bar{f}_i(|S|) = 1$, we estimate $Sum(|S|) = Sum(|S|-1)$, being $\bar{f}_i(|S|)\log_2 \bar{f}_i(|S|) = 1\log_2 1 = 0$ and defining $(\bar{f}_i(|S|)-1)\log_2(\bar{f}_i(|S|)-1) = 0\log_2 0 = 0$ [93]. Instead, when $\bar{f}_i(|S|) > 1$, we need to re-write once again $Sum(|S|)$ in the following way:

$$Sum(|S|) = Sum(|S|-1) + \log_2 \bar{f}_i(|S|) +$$
$$+ (\bar{f}_i(|S|)-1)\log_2(1+\frac{1}{\bar{f}_i(|S|)-1})$$

According to L'Hopital's rule [114]:

$$\lim_{\bar{f}_i(|S|)\to+\infty} (\bar{f}_i(|S|-1)\log_2(1+\frac{1}{(\bar{f}_i(|S|-1)}) = \frac{1}{\ln 2}$$

Thus, we set $1/\ln 2 \approx 1.44$ as the approximation of the third term of $Sum(|S|)$. This approximation best works when most of the flows in T_{int} carry a number of packets much greater than 1 (as usually happens in an ISP backbone network, which is

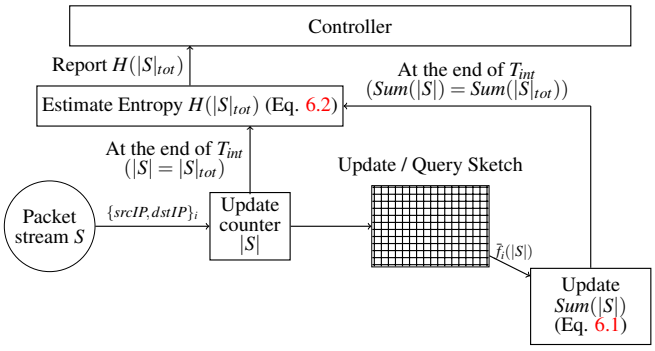


Figure 6.1: Scheme of P4Entropy

the most suitable scenario where to apply our strategy). Finally, $Sum(|S|)$ can be estimated as:

$$Sum(|S|) \approx \begin{cases} Sum(|S|-1) & (\bar{f}_i(|S|) = 1) \\ Sum(|S|-1) + \log_2 \bar{f}_i(|S|) + 1/\ln 2 & \\ & (\bar{f}_i(|S|) > 1) \end{cases} \tag{6.1}$$

This estimation requires at most one logarithm computation.

Since P4 language does not support division, we re-write $\frac{1}{|S|_{tot}} = 2^{-\log_2 |S|_{tot}}$. So, entropy can be written as:

$$H(|S|_{tot}) = \log_2 |S|_{tot} - 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})}$$

In this form, entropy can be estimated by only using P4-supported operations, leveraging P4Log and P4Exp algorithms. In the following, we show how it is possible to further slightly reduce complexity in entropy estimation.

When $|S|_{tot} = \sum_{i=1}^{n} f_i(|S|_{tot}) > Sum(f_i|S_{tot}|)$, it holds that $0 < 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})} < 1$. This is a corner case that happens only when flow distribution is almost uniform (i.e., when most of flows carry only one or very few packets). In this case, we neglect the computation of $2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})}$, meaning that we estimate entropy as flow distribution was perfectly uniform. Network traffic entropy can then finally be estimated as follows:

$$H(|S|_{tot}) \approx \begin{cases} \log_2(|S|_{tot}) & (|S|_{tot} > Sum(|S|_{tot})) \\ \log_2(|S|_{tot}) - 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})} & \\ & (|S|_{tot} \leq Sum(|S|_{tot})) \end{cases} \tag{6.2}$$

6.3.2 P4Entropy algorithm

Figure 6.1 and Algorithm 7 show the scheme and pseudocode of P4Entropy algorithm, leveraging outcomes from Section 6.3.1. First, the algorithm continuously updates $Sum(|S|)$ until the end of T_{int} (*UpdateSum* function) with flow information

Algorithm 7: P4Entropy algorithm

Input: Packet stream S, time interval T_{int}
Output: Entropy estimation $H(|S|_{tot})$ of S in T_{int}

1 $|S| \leftarrow 0$
2 $Sum(|S|) \leftarrow 0$
3 **Function** *UpdateSum*:
4 **while** *currentTime* $< T_{int}$ **do**
5 **for** *Each received packet belonging to flow i* **do**
6 $|S| \leftarrow |S| + 1$
7 $\bar{f}_i(|S|) \leftarrow Sketch(\{srcIP, dstIP\}_i)$
8 **if** $\bar{f}_i(|S|) > 1$ **then**
9 $Sum(|S|) \ll 10 \leftarrow Sum(|S|) \ll 10$
10 $+log_2ES(\bar{f}_i(|S|)) + 1.44 \ll 10$
11 $Sum(|S|_{tot}) \leftarrow (Sum(|S_{tot}|) \ll 10) \gg 10$
12 **return** $Sum(|S|_{tot})$, $|S|_{tot}$
13 **Function** *EstimateEntropy*($Sum(|S|_{tot})$, $|S|_{tot}$):
14 **if** *currentTime* = T_{int} **then**
15 **if** $|S|_{tot} > Sum(|S|_{tot})$ **then**
16 $H(|S|_{tot}) \ll 10 \leftarrow log_2ES(|S|_{tot})$
17 **else**
18 $\text{diff} \leftarrow log_2ES(Sum(|S|_{tot})) - log_2ES(|S|_{tot})$
19 $H(|S|_{tot}) \ll 10 \leftarrow log_2ES(|S|_{tot}) - exp_dES(2, \text{diff})$
20 **return** $H(|S|_{tot}) \ll 10$

from incoming packets. A counter $|S|$ is used to count all incoming packets in the switch. We consider as *flow key* the source IP-destination IP pair of the packet, with $i \sim \{srcIP, dstIP\}_i$. However, other flow definitions could be considered (e.g. 5-tuple) without any loss of generality. A sketch data structure (e.g., Count Sketch or Count-min Sketch, see Section 6.2.2) is used to store the estimated packet count for all the flows, being continuously updated to include information from new packets, and then it is queried to retrieve the estimated packet count $\bar{f}_i(|S|)$ for the flow i the current incoming packet belongs to. This value is then passed to a readable and writable stateful register named $Sum(|S|)$, which is updated as specified in Eq. 6.1. All the floating numbers in the equation must be amplified 2^{10} times, since P4Log outputs an amplified integer value. Only at the end of T_{int}, $Sum(|S|_{tot})$ is reduced by a factor of 2^{10} and its final value, together with $|S|_{tot}$, is returned (Lines 1-12 of the pseudocode).

Traffic entropy is then estimated as specified in Eq. 6.2. The resulted value of $H(|S|_{tot})$ is amplified 2^{10} times since output values of P4Log are amplified, while output values of P4Exp are not. Finally, the switch reports the amplified entropy estimation value to the controller, which can reset all the switch registers to start another estimation in the next T_{int}.

6.4 Evaluation of P4Entropy

We implemented P4Entropy in Python and simulated it for evaluation. We report results in this section.

6.4.1 Evaluation metrics and simulation settings

Testing flow trace

We use 2018-passive CAIDA flow trace [3] for evaluation into 10 observation windows with a fixed number of packets equal to 2^{21} each. Fixing the number of packets in an observation window is needed to compare our approach with a state-of-the-art solution [85], named *SOTA_entropy* for the remainder of the section, which can only be applied to observation windows where the number of packets is fixed to a power of two.

Metrics

We consider *relative error* as metric. We call $\hat{H}$ the estimated traffic entropy in an observation window and H its exact value. The relative error is defined as the average value of $\frac{|H-\hat{H}|}{H} \cdot 100\%$ in the 10 observation windows.

Tuning parameters

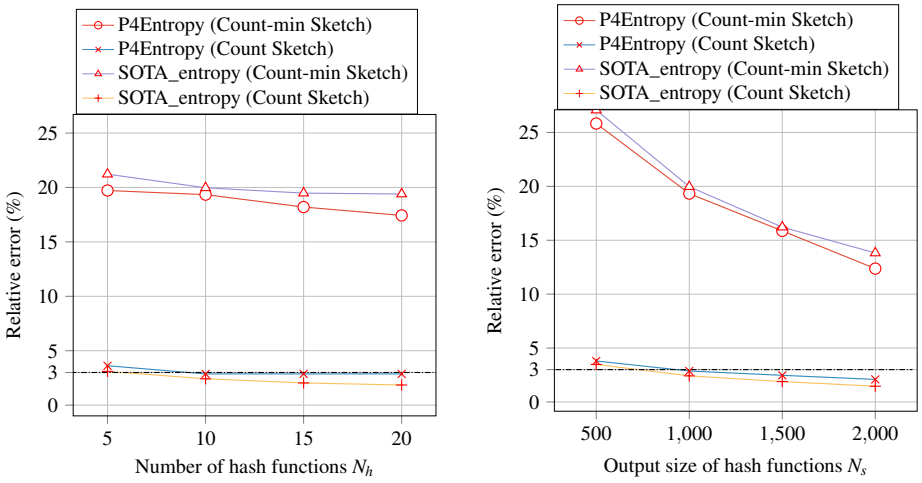
Unless otherwise specified, the default tuning parameters are set as per Table 6.1.

6.4.2 Simulation results

We simulate both our strategy and SOTA_entropy in the case that flow packet counts are estimated in the data plane by adopting either Count-min Sketch or Count Sketch (see Section 6.2.2). We show how entropy estimation is affected while changing the size $N_h \times N_s$ of the sketch (Fig. 6.2). Fig. 6.2(a) shows the relative error in network traffic entropy estimation for the two strategies when N_s is fixed and N_h varies. It shows that the relative error slightly decreases as N_h increases in all the cases. Moreover, P4Entropy and SOTA_entropy lead to similar relative error. It can be noted that, when adopting Count-min Sketch, both P4Entropy and SOTA_entropy have large relative error (around 20%) meaning that Count-min Sketch, with our settings, badly estimates flow packet counts $\bar{f}_i$ and both entropy estimation strategies result ineffective. Additionally, in this case, the relative error of SOTA_entropy is slightly higher than the one of P4Entropy, which is caused by the different ways how $Sum(f_i)$ is estimated. In SOTA_entropy, the Longest Prefix Match (LPM) lookup table for $F(f_i) = f_i \log_2 f_i - (f_i - 1)\log_2(f_i - 1)$ (see [85]) is sensitive to the large packet count (f_i) overestimation caused by Count-min Sketch. Conversely, P4Entropy needs to calculate $\log_2 f_i + \frac{1}{\ln 2}$ (see Eq.

Table 6.1: Default parameters for P4Entropy

	Alg	Parameter	Value
P4Entropy	P4Log	N_{digits}	3
		N_{bits}	4
	P4Exp	N_{terms}	7
Sketch size		$N_h \times N_s$	10×1000



(a) Sensitivity to N_h (b) Sensitivity to N_s

Figure 6.2: Performance comparison of P4Entropy with an existing approach [85]

6.1), which is less sensitive to large overestimations (i) due to the logarithm nature and (ii) because $\frac{1}{\ln 2}$ is a constant value. This effect does not happen while adopting Count-Sketch, since overestimations are much less frequent. In that case, P4Entropy leads to slightly worse results than SOTA_entropy because, unlike SOTA_entropy, it uses an approximation for the computation of the network traffic entropy (see Eq. 6.2).

Fig. 6.2(b) shows instead the impact of a variation of N_s on relative error in entropy estimation. Results are similar to what shown in Fig. 6.2(a), but it can be noted that both strategies are more sensitive to a variation of N_s than of N_h. In this case, when adopting Count Sketch, relative error is always close to 3%. Note that a relative error of 3% is the maximum possible value ensuring that accuracy of practical monitoring applications is not affected [84].

6.5 Related work

6.5.1 Network traffic entropy estimation in data plane

Many works can be found in literature dealing with network traffic entropy estimation partially performed in the switches' data plane. For example, SketchVisor [74], UnivMon [93] and Elastic Sketch [132] all envision some operations to be executed in the programmable data plane and send to the controller only summarized data. However, entropy estimation is executed at the controller due to the need of logarithm calculation. Our approach, instead, allows to compute the estimated entropy directly in the data plane, without any interaction with the controller. Additionally, Lapolli et al. [85] recently implemented network traffic entropy estimation in the data plane using the P4 language, with the aim of detecting DDoS attacks. Their approach is valid but they require the usage of TCAM, which is instead avoided by P4Entropy. Moreover, P4Entropy adopts a time-interval-based observation window, while [85] requires an observation window including a fixed power-of-two number of packets. Our approach is more beneficial since it allows a controller to synchronize the retrieval of estimated entropy among all deployed programmable switches to estimate traffic distribution on a *network-wide* scale [53], thus improving statistical relevance of monitored values.

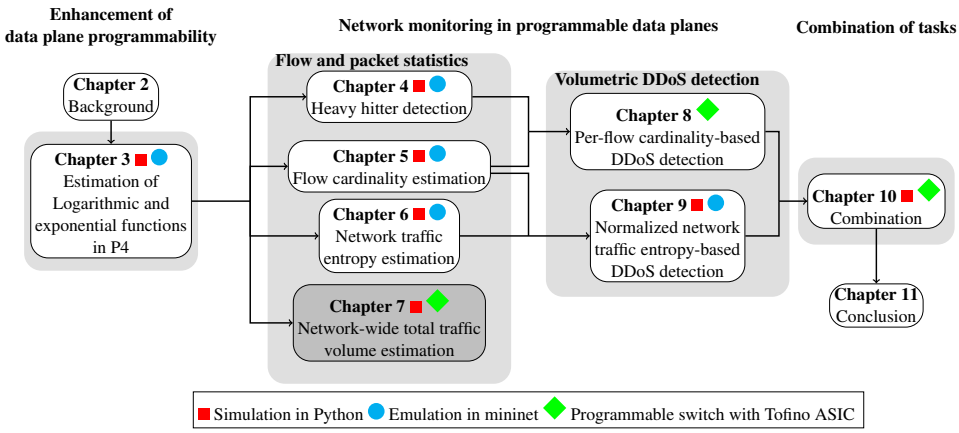
6.6 Concluding remarks

Based on P4Log and P4Exp reported in Chapter 3, we proposed P4Entropy, a novel strategy allowing the estimation of network traffic entropy entirely in the data plane, which has been implemented in P4 as well. We also proved that P4Entropy has comparable accuracy to an existing approach but, as P4Log and P4Exp, does not require the usage of TCAM. Moreover, unlike the state of the art, P4Entropy does not need a fixed-packet observation window, being then more suitable when entropy estimation from multiple switches has to be delivered to the controller in a synchronous fashion. This allows our P4entropy to be easily integrated with other network monitoring tasks in programmable data plane.

Combing this chapter with flow cardinality estimation presented in Chapter 6, it is possible to detect volumetric DDoS attacks entirely in programmable data planes. The details will be reported in Chapter 9.

7

Network-wide total traffic volume estimation



In this chapter, we aim to solve a fundamental problem arising when exploiting programmable data planes for network-wide monitoring: how to estimate the overall number of packets in the network (i.e., the traffic volume), and the related number and size of flows, while avoiding packet double counting. The ability to precisely estimate the traffic volume *(i.e., number of distinct packets flowing in the network), and the related* number of distinct flows *and average* flow size *(i.e., average number of packets per flow) is necessary to support a broad range of monitoring tasks that we mentioned in all chapters of this thesis. Most existing works solve this problem by ensuring that each packet is counted only once on its path, which limits routing or requires coordination among devices. We propose a different approach, INVEST, a flow-based traffic volume estimator for P4-based switches, that relies on and can reuse commonly employed data structures while naturally solving the double-counting problem.*

The study discussed in this chapter is based on the paper "INVEST: Flow-based Traffic Volume Estimation in Data-plane Programmable Networks" published in IFIP Networking Conference 2021.

7.1 Introduction

In the context of network monitoring, the ability to precisely estimate the *traffic volume* [36] (i.e., the number of distinct packets flowing in the network) in a time frame is fundamental. Its correct estimation is necessary to support a broad range of monitoring tasks, including heavy-hitter [36][69][54] and heavy-changer [74][93] detection, network traffic entropy estimation [132][55], DDoS attacks [85] and superspreaders [116] detection.

Obviously, traffic volume can be deterministically computed as the sum of packet counts of all ingress (or egress) interfaces of all border routers; however, this solution is cumbersome since it requires regularly polling possibly almost all network devices; in addition, it cannot be used to (easily) estimate flow count and size, which are also important for the tasks above.

Several network-wide monitoring solutions exploiting the potential of programmable data planes have already been proposed in literature [69][74][93][132]. However, as already pointed in [36], most of them assume that each packet is monitored and counted by only one programmable switch on its path. If this assumption does not hold, the proposed strategies are not accurate, due to *packet double counting*, which leads to degraded monitoring performance. Unfortunately, such an assumption has strong implications on how routing should be performed and/or necessitates a coordination between the programmable switches in the network, which makes such strategies either imprecise or impractical.

In this chapter, we propose INVEST (Improved Network traffic Volume ESTimaTion), an accurate and memory-efficient method for the estimation of traffic volume (as well as number of distinct flows and average flow size) at the SDN controller. It adopts generic (and commonly used) data structures that *(i.)* are allocated and updated in the data plane of a (potentially) reduced number of aggregation programmable switches and *(ii.)* is inherently robust with respect to packet double counting. More precisely, INVEST relies, in each programmable switch, on a local packet counter and on a distinct flow counter based on HyperLogLog [63]. Using these, our strategy is able to estimate, at the controller, the number of distinct flows in the network and the average flow size (i.e., number of packets per flow), which can then be used for total traffic volume estimation. The advantages of our method include that it does not require any control over border routers, exploits data structures that can be useful for a variety of flow-based monitoring tasks beyond traffic volume estimation, and can be feasibly deployed as a stand-alone module in existing network-wide measurement systems relying on programmable data planes [74].

We also theoretically analyze and experimentally evaluate INVEST, proving that it can estimate the traffic volume accurately by consuming low memory, once tuning parameters (e.g. HyperLogLog register size and flow type) are properly set. We implemented the INVEST data structures in a carrier-grade Tofino-based [2] P4-programmable switch and show that the method can process packets at line-rate with only small hardware resource usage overhead.

In summary, this chapter makes the following contributions:

- We design INVEST, a flow-based method that exploits programmable data planes to estimate the traffic volume while solving the double counting problem;
- We theoretically analyze INVEST, reporting bounds on the relative error of traffic volume estimation according to different tuning parameters;
- As a building block of INVEST, we implement the HyperLogLog Update procedure [63] for flow cardinality estimation in P4_16 [42];
- Based on our HyperLogLog Update implementation, we developed a prototype of INVEST, installed it in a carrier-grade programmable switch with Tofino Application Specific Integrated Circuit (ASIC) [2], and evaluated it.

7.2 Basic knowledge and used compact data structure

In this section we recall some background notions that will be exploited in the chapter.

7.2.1 HyperLogLog algorithm

HyperLogLog (HLL) [63] is a sketch-based algorithm for the estimation of the cardinality of a data stream. In our context, it can be adopted to estimate the number of distinct flows (i.e., flow cardinality) crossing a switch while requiring low memory occupation. HLL uses an m-sized register M, where m indicates the number of counters (each allocated to d bits) included in the register, which is *updated* to include data from new elements of the data stream and *queried* to get an estimation of the cardinality.

The *Query* operation works as follows. When an incoming packet with *flow key id* arrives at the switch, HLL applies a hash function with output size 2^{os} to *id* (with $os \geq \log_2 m + 2^d$): the resulting os-bit binary string H is denoted by $H = [0 : os-1]$, meaning that 0 is the index of the leftmost bit, while $os-1$ is the index of the rightmost one. HLL then updates register M. Let *bucket* be the leftmost $\log_2 m$ bits of H and x the remaining bits, i.e., $bucket = H[0 : \log_2 m - 1]$ and $x = H[\log_2 m : os-1]$; M is updated following the rule: $M[bucket] = max(M[bucket], value)$, where $value$ is the index of the rigthtmost 1 of x plus one.

The *Query* operation is used to estimate the flow cardinality $\hat{n}_M$: it is computed as a harmonic mean of the m counters: $\hat{n}_M = \alpha_m \cdot m^2 \cdot (\sum_{bucket=0}^{m-1} 2^{-M[bucket]})^{-1}$, where α_m is a bias correction parameter. The standard error of HLL has been proven to be $\frac{1.04}{\sqrt{m}}$ [63].

An interesting property of HLL, leveraged in this work, is that multiple HLL registers, e.g. M_m and M_n, can be merged into a single register $M_{mn} = M_m \cup M_n$ to count the flow cardinality of the packet streams that have independently updated

M_m and M_n, i.e., $\hat{n}_{M_m \cup M_n}$, avoiding any double counting (i.e., in general $\hat{n}_{M_m} + \hat{n}_{M_n} \geq \hat{n}_{M_m \cup M_n}$, and only where each flow traverses only one of the two switches $\hat{n}_{M_m} + \hat{n}_{M_n} = \hat{n}_{M_m \cup M_n}$).

7.2.2 Strong Law of Large Numbers

Given a sequence of i.i.d. (independent and identically distributed) random variables $X_1, X_2, X_3, \ldots, X_n$ with finite expectation μ, and its *sample mean* defined as $\bar{X_n} = \frac{\sum_{i=1}^{n} X_i}{n}$, the Strong Law of Large Numbers states that $\bar{X_n}$ converges *almost surely* to μ as $n \to \infty$, i.e., $\mathbb{P}(\lim_{n\to\infty} \bar{X_n} = \mu) = 1$.

7.2.3 Central Limit Theorem

Given a sequence of i.i.d. random variables $X_1, X_2, X_3, \ldots, X_n$ (i.e., whole population), n_i variables, each denoted by $X_{j,i}^{sample}$ (i.e., $X_{1,i}^{sample}, X_{2,i}^{sample}, \ldots, X_{i,n_i}^{sample}$), are randomly picked from the sequence: such a selection is called *random sample*, and the *sample mean* is defined as $\bar{X}_i^{sample} = \frac{\sum_{j=1}^{n_i} X_{j,i}^{sample}}{n_i}$. The Central Limit Theorem states that the sample mean $\bar{X}_i^{sample}$ follows a Gaussian distribution $\mathcal{N}(\mu, \frac{\sigma^2}{n_i})$ as $n_i \to \infty$, where μ and σ^2 are the expectation and variance of the random variables X_k belonging to the whole population, respectively.

7.3 Estimation of traffic volume

In this section, we present our traffic volume estimation method, named *INVEST* for the sake of brevity, and provide the theoretical foundations of its workings.

7.3.1 Problem definition

We start by formally defining the problem. **Given:**

- A time interval T_{int};
- A packet stream S in T_{int};
- The total number of packets $|S_i|$ that have traversed each switch i at the end of T_{int};
- The updated HLL register M_i (size m) for each switch i at the end of T_{int};
- The number q of programmable switches in the network;

Returns an estimation of the traffic volume (i.e., the overall number of packets from stream S), denoted by $|\hat{S}_{tot}|$, that have crossed the network in T_{int}.

Algorithm 8: INVEST Estimation Method

Input: A time interval T_{int}, a packet stream S in T_{int}, the number q of programmable switches in the network, an m-sized HLL register M_i for each programmable switch i, a packet counter $|S_i|$ for each programmable switch i

Output: An estimation of the overall number of packets in the network $|\hat{S}_{tot}|$ in T_{int}

Function *Update(S, M_i, S_i, T_{int}, q)*:
  $|S_i| \leftarrow 0 \; \forall i \in \{1,\dots,q\}$
  $M_i \leftarrow$ Empty m-sized HLL register $\forall i \in \{1,\dots,q\}$
  while *currentTime* $< T_{int}$ **do**
    for *Each packet in S identified by flow key id and traversing programmable switch i* **do**
      $|S_i| \leftarrow |S_i| + 1$
      $M_i.UpdateHLL(id)$;
  return $|S_i|$, $M_i \; \forall i \in \{1,\dots,q\}$

Function *Query(M_i, S_i, T_{int}, q)*:
  $M_{top-k} \leftarrow$ Empty m-sized HLL register
  $k \leftarrow 0$, $\hat{R}_{tot} \leftarrow 0$, $\hat{n}_{tot} \leftarrow 0$,
  $\hat{n}_i \leftarrow 0 \; \forall i \in \{1,\dots,q\}$
  if *currentTime* $= T_{int}$ **then**
    $\hat{n}_i \leftarrow QueryHLL(M_i) \; \forall i \in \{1,\dots,q\}$;
    $\hat{n}_{tot} \leftarrow QueryHLL(M_1 \cup M_2 \cup \cdots \cup M_q)$;
    $\mathcal{N} \leftarrow \{\hat{n}_1, \hat{n}_2, \cdots, \hat{n}_q\}$
    $\mathcal{N}_{aux} \leftarrow \{\hat{n}_1, \hat{n}_2, \cdots, \hat{n}_q\}$
    while $QueryHLL(M_{top-k}) < \hat{n}_{tot}$ **do**
      $\hat{n}_{max} \leftarrow Max(\mathcal{N}_{aux})$;
      $i \leftarrow$ index in $\mathcal{N}$ corresponding to $\hat{n}_{max}$
      $M_{top-k} \leftarrow M_{top-k} \cup M_i$
      $k \leftarrow k + 1$
      $\hat{R}_{tot} \leftarrow \hat{R}_{tot} + \frac{|S_i|}{\hat{n}_i}$
      $\mathcal{N}_{aux} \leftarrow \mathcal{N}_{aux} \setminus \hat{n}_{max}$
    $\hat{R}_{tot} \leftarrow \frac{1}{k}\hat{R}_{tot}$
    $|\hat{S}_{tot}| \leftarrow \hat{n}_{tot}\hat{R}_{tot}$
    return $|\hat{S}_{tot}|$

7.3.2 INVEST estimation method

The INVEST estimation method consists of two operations, *INVEST Update* and *INVEST Query* (see Fig. 7.1). INVEST Update is autonomously performed dur-

ing T_{int} by each programmable switch's data plane every time it is crossed by a packet, while INVEST Query is executed by the controller at the end of T_{int} using information made available by the switches. In the following, we describe those operations, whose pseudocode is reported in Alg. 8.

INVEST Update

Each time a packet crosses a switch, its data plane updates the counter $|S_i|$ and the HLL register M_i. $|S_i|$ is simply increased by one, while M_i is updated using the flow key *id* of the packet, as specified by the HLL Update operation [63].

INVEST Query

At the end of T_{int}, $|S_i|$ and $M_i \; \forall i \in \{1,\dots,q\}$ are retrieved by the controller. For each of the switches, the controller estimates the number of distinct flows $\hat{n}_i$[1] obtained by querying the HLL register M_i as specified by the HLL Query operation [63]:

$$\hat{n}_i = QueryHLL(M_i) \quad \forall i \in \{1,\dots,q\}$$

Due to the union property of HLL, the overall number of distinct flows $\hat{n}_{tot}$ in the network can then be estimated as:

$$\hat{n}_{tot} = QueryHLL(M_1 \cup M_2 \cup \cdots \cup M_q)$$

Once $\hat{n}_i \; \forall i \in \{1,\dots,q\}$ and $\hat{n}_{tot}$ have been estimated, the controller picks the *top-k* largest $\hat{n}_i \in \mathcal{N} = \{\hat{n}_1,\cdots,\hat{n}_q\}$ with k being the minimum value that satisfies $QueryHLL(M_1 \cup M_2 \cup \cdots \cup M_k) = \hat{n}_{tot}$[2]: the adopted procedure is described in Lines 18-24 of Alg. 8. Clearly, $k \leq q$ and the more the flows are concentrated on a few number of switches (i.e., the traffic load is strongly unbalanced), the lower k is.

Then, the average number of packets per flow, denoted by $\hat{R}_i$, is computed considering the switches belonging to the *top-k* set in the following way:

$$\hat{R}_i = \frac{|S_i|}{\hat{n}_i} \quad \forall i \in \{1,\dots,k\}$$

The average number of packets per flow $\hat{R}_{tot}$ in the network is estimated as an average of the average number of packets per flow per switch $\hat{R}_i$, by only considering the *top-k* switches:

$$\hat{R}_{tot} = \frac{1}{k}\sum_{i=1}^{k}\hat{R}_i = \frac{1}{k}\sum_{i=1}^{k}\frac{|S_i|}{\hat{n}_i} \tag{7.1}$$

[1] In this chapter, the *cap* symbol identifies *estimated* values, such as $\hat{n}_i$, $\hat{R}_i$, $|\hat{S}_{tot}|$. Their cap-less counterparts n_i, R_i, $|S_{tot}|$ indicate instead *exact* values.

[2] From now on the index i will refer, without any further ambiguity, to the switches belonging to the *top-k* set (also abbreviated in *top-k switches*).

Finally, the traffic volume $|S_{tot}|$ is estimated as:

$$|\hat{S}_{tot}| = \hat{n}_{tot}\hat{R}_{tot} = \frac{\hat{n}_{tot}}{k}\sum_{i=1}^{k}\frac{|S_i|}{\hat{n}_i}$$

Note that the accuracy of the estimation $|\hat{S}_{tot}|$ depends on two aspects: 1) how accurate HLL is on estimating the exact values of n_i and n_{tot} and 2) how accurate the estimation $\hat{R}_{tot}$ is with respect to its exact value $R_{tot} = \frac{|S_{tot}|}{n_{tot}}$. Taking this into account, in the next subsection we prove that INVEST does indeed converge to the desired values in realistic scenarios.

7.3.3 Theoretical analysis

Theorem 1. *The traffic volume estimator $|\hat{S}_{tot}|$ is an asymptotically unbiased estimator of $|S_{tot}|$ as $m \to \infty$, $n_i \to \infty\ \forall i \in \{1,\dots,k\}$ and $n_{tot} \to \infty$.*

Proof. In the considered T_{int} where $|S_{tot}|$ has to be estimated, the network is characterized by a flow stream $F_{tot} = \{f_1, f_2, \cdots, f_{n_{tot}}\}$, where f_j indicates the flow packet count of flow j and $n_{tot} = |F_{tot}|$ is the overall number of distinct flows. It thus hold that $|S_{tot}| = \sum_{j=1}^{n_{tot}} f_j$. If we define f_j^i as the packet count of f_j as recorded in switch i, with n_i being the number of distinct flows seen in switch i, we can also write $|S_i| = \sum_{j=1}^{n_i} f_j^i$.

With respect to the relative error of HLL, called ε_{HLL}, it holds that $\mathbb{P}(|\varepsilon_{HLL}| \leq 3\frac{1.04}{\sqrt{m}}) \geq 0.997$ [106] and thus HLL accuracy depends on the register size m. So, as $m \to \infty$, the estimations $\hat{n}_i$ and $\hat{n}_{tot}$ obtained by querying the HLL registers (and their union) converge to the real values (i.e., $\frac{\hat{n}_i}{n_i} = 1\ \forall i$ and $\frac{\hat{n}_{tot}}{n_{tot}} = 1$)[3] with arbitrary high probability.

We now assume that the $|F_{tot}|$ packet counts of distinct flows are independent and identically distributed (i.i.d.) random variables, meaning that the number of packets generated in a flow does not give any information on the number of packets generated in another flow, and that the packet count random variables have all the same probability distribution. This makes sense in practice even if we know that there are many distinct types of flow, e.g. short-lived HTTP requests, VoIP calls, file transfers, etc. yielding very different packet numbers per unit of time. Even if the distribution of an f_j clearly depends on the type of flow j, the distribution of flow types during T_{int} can be viewed as fixed (though not easy to characterize), hence the f_j random variables can be seen as drawn from the combined distribution mapping a flow to its type (or even instance) and then the number of packets (over T_{int}) of that type.

We then pick k samples obtained by randomly sampling with replacement (i.e., with the possibility that the same flow f_j is included in multiple random samples), each associated to index i and characterized by $F_i = \{f_1^i, f_2^i, \cdots, f_{n_i}^i\}$. It is here

[3] In this demonstration, the notations n_i and $\hat{n}_i$ can be used interchangeably. The same holds for n_{tot} and $\hat{n}_{tot}$ and for $R_i = \frac{|S_i|}{n_i}$ and $\hat{R}_i = \frac{|S_i|}{\hat{n}_i}$.

assumed that $F_1 \cup F_2 \cup \cdots \cup F_k = F_{tot}$, meaning that each flow belongs to at least one of the k random samples. In the practical scenario considered in this chapter, this means that any switch i among the selected *top-k* switches is randomly crossed by n_i flows and each flow j in F_{tot} is seen by at least one switch, which is ensured by design, based on how k is chosen in our strategy, when all q switches in the network are programmable switches and have installed the INVEST strategy. An additional requirement is that all packets are always routed on the same path or, alternatively, that a different flow key is specified for different routing paths (e.g. in the case of multicast traffic); if this latter assumption does not hold, two different switches may count a different number of packets for the same flow j while, to apply INVEST, $f_j^i = f_j^z$ must always hold if random samples F_i and F_z of i and z both include f_j[4].

As already introduced, $R_i = \frac{|S_i|}{n_i} = \frac{\sum_{j=1}^{n_i} f_j^i}{n_i}$ is the average flow packet count in random sample i. According to the Central Limit Theorem, when $n_i \to \infty$ $\forall i \in \{1,\ldots,k\}$, $R_i = \frac{|S_i|}{n_i} \sim \mathcal{N}(\mu, \frac{\sigma^2}{n_i})$, where μ is the expected number of packets per flow, σ^2 the expected variance of the Gaussian distribution, and where R_i are independent random variables, since they refer to different random samples obtained by sampling with replacement.

As stated by the Strong Law of Large Numbers, as $n_{tot} \to \infty$, μ approaches the expected number of packets per flow in F_{tot}, that is $\mu = \frac{\sum_{j=1}^{n_{tot}} f_j}{n_{tot}} = \frac{|S_{tot}|}{n_{tot}} = R_{tot}$.

In INVEST, the estimation $|\hat{S}_{tot}|$ requires the estimation of R_{tot}, which is computed as $\hat{R}_{tot} = \frac{1}{k}\sum_{i=1}^{k} R_i$. $\hat{R}_{tot}$ is a Gaussian random variable, being a linear combination of Gaussian random variables (i.e., R_i). Its expectation $\mathbb{E}[\hat{R}_{tot}]$ can be expressed, for $n_i \to \infty$ $\forall i \in \{1,\ldots,k\}$ and $n_{tot} \to \infty$, in the following way:

$$\mathbb{E}[\hat{R}_{tot}] = \mathbb{E}\left[\frac{1}{k}\sum_{i=1}^{k} R_i\right] = \frac{1}{k}\sum_{i=1}^{k}\mathbb{E}\left[\frac{|S_i|}{n_i}\right] = \frac{1}{k}\sum_{i=1}^{k}\mu = \frac{|S_{tot}|}{n_{tot}}$$

$\hat{R}_{tot}$ is thus an asymptotically unbiased estimator of R_{tot} as $n_{tot} \to \infty$ and $n_i \to \infty$ $\forall i \in \{1,\ldots,k\}$, since $\frac{\mathbb{E}[\hat{R}_{tot}]}{R_{tot}} = 1$.

$|\hat{S}_{tot}|$ is then estimated as $|\hat{S}_{tot}| = \hat{n}_{tot}\hat{R}_{tot}$. $|\hat{S}_{tot}|$ is a Gaussian variable, being so $\hat{R}_{tot}$. Since $\hat{R}_{tot}$ is an asymptotically unbiased estimator of R_{tot} as $n_{tot} \to \infty$ and $n_i \to \infty$ $\forall i \in \{1,\ldots,k\}$, $|\hat{S}_{tot}|$ is asymptotically unbiased as well if also $m \to \infty$ holds (i.e., $\frac{\hat{n}_{tot}}{n_{tot}} = 1$):

$$\frac{\mathbb{E}[|\hat{S}_{tot}|]}{|S_{tot}|} = \frac{\hat{n}_{tot}\mathbb{E}[\hat{R}_{tot}]}{|S_{tot}|} = \frac{\hat{n}_{tot}\frac{|S_{tot}|}{n_{tot}}}{|S_{tot}|} = 1$$

■

[4]Note that, in our case, double counting a flow j does not generate any issue in flow cardinality estimation, since the union property of HLL ensures that a packet counted twice (e.g. by switches i and z) is considered only once when estimating the traffic volume.

Remark. *In a practical network scenario as the one considered in this chapter, n_i is upper-bounded by $|S_i|$ (i.e., the number of distinct packets crossing the switch i): in this case, the Central Limit Theorem still holds if $n_i \to |S_i|$ $\forall i \in \{1,\dots,k\}$ and $|S_i|$ is big enough, a safe assumption for a large network such as that of an Internet Service Provider (ISP).*

Theorem 2. *As $m \to \infty$, $n_i \to \infty$ $\forall i \in \{1,\dots,k\}$ and $n_{tot} \to \infty$, it holds that the relative error $\left|\frac{|\hat{S}_{tot}|-|S_{tot}|}{|S_{tot}|}\right| = 0$, i.e., the estimation $|\hat{S}_{tot}|$ equals $|S_{tot}|$.*

Proof. As already defined in Theorem 1, the relative error of HLL estimation is ε_{HLL}. Additionally, we define the relative error of the estimation $\hat{R}_{tot}$ as $\varepsilon_{\hat{R}_{tot}}$. Being $\hat{n}_{tot}$ estimated by querying a union of HLL registers, it is then possible to write:

$$\begin{aligned}|\hat{S}_{tot}| = \hat{n}_{tot}\hat{R}_{tot} &= (1+\varepsilon_{HLL})n_{tot}(1+\varepsilon_{\hat{R}_{tot}})R_{tot} \\ &= (1+\varepsilon_{HLL})(1+\varepsilon_{\hat{R}_{tot}})|S_{tot}|\end{aligned}$$

We recall that $|\varepsilon_{HLL}|$ decreases as m increases, where m is the HLL register size.

The relative error of the estimation $\hat{R}_{tot}$ of R_{tot} can be instead obtained by looking at the probability distribution of $\hat{R}_{tot}$. By Theorem 1, as $n_{tot} \to \infty$ and $n_i \to \infty$ $\forall i \in \{1,\dots,k\}$, $\hat{R}_{tot}$ is a Gaussian random variable with $\mathbb{E}[\hat{R}_{tot}] = \mu = \frac{|S_{tot}|}{n_{tot}}$. The absolute value of the relative error $|\varepsilon_{\hat{R}_{tot}}|$ is the coefficient of variation of $\hat{R}_{tot}$:

$$|\varepsilon_{\hat{R}_{tot}}| = \frac{\sqrt{Var[\hat{R}_{tot}]}}{\mathbb{E}[\hat{R}_{tot}]}$$

Being $\hat{R}_{tot}$ a linear combination of independent Gaussian random variables (i.e., R_i), $Var[\hat{R}_{tot}]$ is the following:

$$Var[\hat{R}_{tot}] = Var\left[\frac{1}{k}\sum_{i=1}^{k} R_i\right] = \frac{1}{k^2}\sum_{i=1}^{k} Var[R_i] = \frac{1}{k^2}\sum_{i=1}^{k}\frac{\sigma^2}{n_i}$$

As $n_i \to \infty$ $\forall i \in \{1,\dots,k\}$, $Var[\hat{R}_{tot}] = 0$ and $|\varepsilon_{\hat{R}_{tot}}| = 0$ as well. Thus, if this condition holds, the accuracy of the estimation $|\hat{S}_{tot}|$ is only affected by the HLL register size m and improves as m increases:

$$|\hat{S}_{tot}| = (1+\varepsilon_{HLL})|S_{tot}|$$

The above formula implies:

$$\left|\frac{|\hat{S}_{tot}| - |S_{tot}|}{|S_{tot}|}\right| = |\varepsilon_{HLL}| = \frac{1.04}{\sqrt{m}}$$

If also $m \to \infty$ holds, $|\varepsilon_{HLL}| = 0$ with arbitrary high probability and we can finally write:

$$\left|\frac{|\hat{S}_{tot}| - |S_{tot}|}{|S_{tot}|}\right| = 0$$

■

Remark. *Theorem 2 explains why INVEST considers only the top-k switches to estimate the traffic volume, and not all the q switches. The reason is that, to ensure good performance in a practical scenario, n_i must be large enough for all the switches involved in the estimation of $\hat{R}_{tot}$, so that $Var[\hat{R}_{tot}] \to 0$. In the case of networks characterized by unbalanced traffic matrices, it is not unusual that n_i is relatively small for some of the switches i, and including such n_i in the computation of $\hat{R}_{tot}$ would jeopardize the estimation. By selecting the top-k switches (in terms of n_i) that cover all the flows in the network, such negative effect is instead strongly mitigated. This will also be experimentally shown in Section 7.6.*

Remark. *In general, Theorems 1 and 2 tell us that the bigger m, n_i and n_{tot} are, the better the estimation of $|S_{tot}|$ is. The value of m must be chosen big enough to ensure good estimations for n_i and n_{tot}, whose value instead depends on how a "flow" is defined. In practice, the trivial best possible estimation of $|S_{tot}|$ can be obtained when $n_i = |S_i|$ and $n_{tot} = |S_{tot}|$, i.e., when each packet is considered as an independent flow. However this solution, similar to the one proposed in [36], requires a unique identifier/marker for each packet [35][138][90] (i.e., unique packet id), which does not scale well. Our Theorems show that considering sufficiently fine-grained flows (e.g., characterized by a $\{srcIP, dstIP\}$ pair as flow key rather than by simply srcIP or dstIP) can effectively enhance the estimation of $|S_{tot}|$, since such a choice increases n_i and n_{tot}.*

Theorem 3. *If $F_1 \cup F_2 \cup \cdots \cup F_k = F^k_{tot} \subset F_{tot}$ and thus $n^k_{tot} < n_{tot}$, as $m \to \infty$, $n_i \to \infty$ $\forall i = 1, \ldots, k$ and $n_{tot} \to \infty$, it holds that $\left|\frac{|\hat{S}^k_{tot}| - |S_{tot}|}{|S_{tot}|}\right| = \left|\frac{n^k_{tot}}{n_{tot}} - 1\right|$, where $|\hat{S}^k_{tot}|$ is the estimation of the traffic volume over the k random samples.*

Proof. By definition, $\hat{R}_{tot}$ is the estimation of R_{tot} considering the k random samples, while by Theorem 2 it holds that, when $m \to \infty$, $n_i \to \infty$ $\forall i = 1, \ldots, k$ and $n_{tot} \to \infty$, $|\varepsilon_{\hat{R}_{tot}}| = 0$ and $|\varepsilon_{HLL}| = 0$. We can then write:

$$\begin{aligned}
\left|\frac{|\hat{S}^k_{tot}| - |S_{tot}|}{|S_{tot}|}\right| &= \left|\frac{|\hat{S}^k_{tot}|}{|S_{tot}|} - 1\right| = \left|\frac{\hat{n}^k_{tot}\hat{R}_{tot}}{n_{tot}R_{tot}} - 1\right| \\
&= \left|\frac{(1+\varepsilon_{HLL})n^k_{tot}(1+\varepsilon_{\hat{R}_{tot}})R_{tot}}{n_{tot}R_{tot}} - 1\right| \\
&= \left|\frac{n^k_{tot}}{n_{tot}} - 1\right| \quad (As\ |\varepsilon_{\hat{R}_{tot}}| = |\varepsilon_{HLL}| = 0)
\end{aligned}$$

■

Remark. *Theorem 3 is relevant when it cannot be ensured that all flows are visible to INVEST, e.g. in a partial or incremental deployment scenario [54], where a number of programmable switches, q, coexists with legacy non-programmable devices. Theorem 3 shows that, in this case, it is better to first replace those non-programmable switches that are crossed by the largest number of distinct flows overall, so that n^k_{tot} is maximized and approaches n_{tot}. Such a strategy has already*

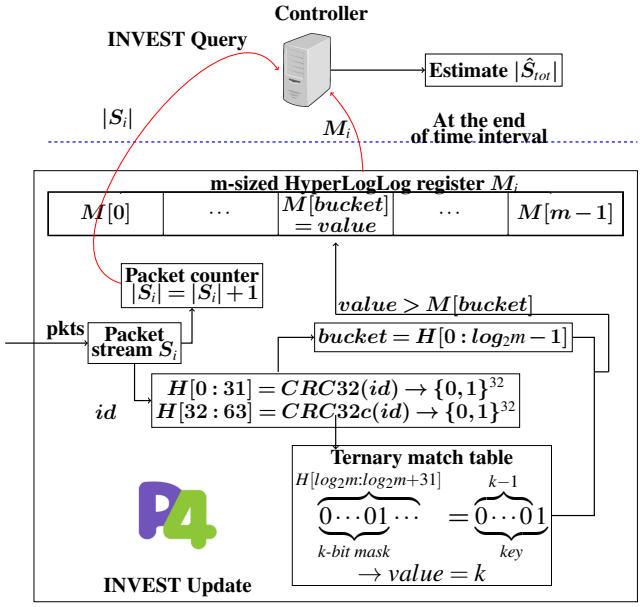


Figure 7.1: Scheme of the proposed INVEST strategy

been proposed in [54] and, as a consequence of Theorem 3, yields the best possible estimate of $|S_{tot}|$ in a hybrid scenario.

7.4 Implementation of INVEST in P4

We have successfully implemented our INVEST strategy, depicted in Fig. 7.1, in a small network including a P4 programmable commodity switch with a Tofino ASIC and a simple controller. An open-source version of the implemented P4_16 code has been released in [33]. In this Section, we report the details of our implementation.

7.4.1 INVEST Update (P4-enabled Switch)

Counting the number of packets $|S_i|$

We use a register (a counter could also be used) to count all the incoming packets in the switch.

Updating the HLL Register

We consider an m-sized HLL register with $m = 2048$, where each cell is assigned $d = 5$ bits. A hash function with at least $\log_2 m + 2^d = 43$ bit of output size is thus needed by the HLL Update operation (see Section 7.2.1). However, Tofino ASIC only supports hash functions with output size up to 32 bits. To address

Table 7.1: Properties of hash functions

Hash function name	poly	Reversed	init	xor
CRC32	0x104C11DB7	True	0	0xFFFFFFFF
CRC32c	0x11EDC6F41	True	0	0xFFFFFFFF

Table 7.2: Ternary match table used by INVEST Update

Mask (32 bits)	Key (32 bits)	Action (l value)
$1000\cdots0$	$0000\cdots0$	No Action
$1000\cdots0$	$1000\cdots0$	1
$1100\cdots0$	$0100\cdots0$	2
$1110\cdots0$	$0010\cdots0$	3
$\vdots$	$\vdots$	$\vdots$
$11\cdots110$	$00\cdots010$	31

this limitation, we concatenate the 32-bit outputs obtained by hashing the flow key *id* with two different supported hash functions, thus generating a 64-bit hash H. Specifically, *id* is first hashed by the CRC32 hash function, setting the bits $H[0:31]$. Bits $H[32:63]$ are set by hashing *id* with the CRC32c hash function. The properties of the adopted hash functions are reported in Table 7.1.

The *bucket* to be considered in the HLL register M for update is equal to the last $\log_2 m$ bits of H, which are easily obtained by truncating H. The *value*, which is the index of the rightmost 1 of the bits $H[\log_2 m : \log_2 m + 31]$ plus one (in brief l), is obtained with the help of a ternary match table where some pre-computed values are stored, as shown in Table 7.2. The table includes 31 entries: for each entry a *mask*, a *key* and an *action* (reporting the l value) are specified. The table is scrolled from top to bottom: the considered mask is used, by applying the logical AND operator with $H[\log_2 m : \log_2 m + 31]$, to retrieve the first l bits of $H[\log_2 m : \log_2 m + 31]$. Once the masked $H[\log_2 m : \log_2 m + 31]$ binary string is obtained, it is compared with the corresponding key in the table. If the masked binary string is equal to the key string, l, which is the *value* we are looking for, is retrieved from the *action* column and the search is stopped, otherwise the next row is considered.

Finally, once *value* is retrieved, if it is larger than the bucket-indexed value $M[bucket]$ in the HLL register, the new value replaces the old value.

7.4.2 INVEST Query (Controller)

The controller is implemented on top of the application program interface (API) provided by the switch vendor. In a large-scale scenario, it can pull $|S_i|$ and M_i from each switch i in the network and can estimate the traffic volume by executing the INVEST Query operation described in Alg. 8. We implemented the INVEST Query logic in Python.

7.5 INVEST adoption for network-wide monitoring

A good estimation of $|S_{tot}|$ is important for the execution of different *network-wide monitoring tasks* [93][132], where (partially processed) information exposed by multiple switches is collected by a centralized controller and used to take decisions at a global scale. In this section, we discuss what are the network-wide monitoring tasks that would benefit from a proper estimation of $|S_{tot}|$ or, more in general, from exploiting the data collected by INVEST.

- **Heavy-hitter detection**: Network-wide heavy hitters are flows that carry more than a small fraction of the overall packets in the network. P4-based strategies for network-wide heavy-hitter detection have already been proposed in literature. The one proposed in [69] does not prevent double packet counting and overestimates the traffic volume $|S_{tot}|$ at the controller, making it extremely hard to correctly set the global threshold for heavy-hitter detection. Conversely, in [54] the controller collects from each switch a *sample list* that includes the local packet count of each flow overcoming a sampling threshold, and uses this information to estimate $|S_{tot}|$. In this case, double counting is prevented, but a good $|S_{tot}|$ estimation is ensured only with low sampling thresholds, making such a solution cumbersome, since huge sample lists would be needed to be stored and forwarded to the controller. For both strategies, the adoption of INVEST would greatly simplify heavy-hitter detection.

- **Heavy traffic volume change detection**: Estimating and comparing $|S_{tot}|$ in consecutive time intervals makes it possible to detect heavy changes in the number of packets flowing in the network. A significant decrease on the traffic volume may help disclose malfunctioning in some areas of the network, while a consistent increase may unveil flash-crowd events. With INVEST it is possible to easily detect traffic volume changes at the controller.

- **Traffic entropy estimation**: Network traffic entropy is a metric that gives an indication on the traffic distribution across the network. In [55] a method to locally estimate network traffic entropy directly in each P4-based programmable switch's data plane is proposed: since the entropy is computed independently in each switch, it cannot be considered as a network-wide strategy. The network-wide traffic entropy can instead be computed by the controller as $H_{tot} = \log_2(|S_{tot}|) - \frac{1}{|S_{tot}|}\sum_{x=0}^{n_{tot}} f_x \log_2 f_x$ (see [55]), where f_x is the packet count of flow identified by key x. If a method to compute the different $f_x \log_2 f_x$ in the programmable switches and deliver them to the controller without double packet counts would be available (future work), $|S_{tot}|$ in the formula could instead be estimated by INVEST.

- **DDoS attack detection**: When a DDoS attack is occurring, a destination node (i.e., the target) is usually contacted by an abnormal number of sources.

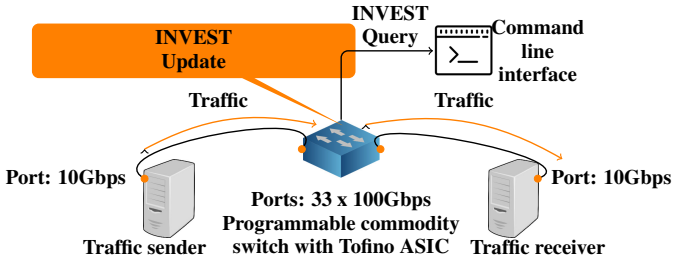


Figure 7.2: Configuration of the physical testbed

Some strategies for DDoS attack detection in P4-programmable switches have been already proposed in literature [51][85]. These strategies, however, focus on DDoS detection on a single switch, which may give false positives or negatives compared to a network-wide approach. The data collected by INVEST may help for the design of an effective network-wide DDoS detection strategy. In fact, the HLL registers M_i gathered by the controller, if updated using *srcIP* as flow key, can be merged to estimate the overall number n_{tot} of traffic sources in the network. If a method to estimate the number of sources routing traffic towards a specific destination was properly implemented in each programmable switch (future work), a threshold-based network-wide DDoS detection strategy could be easily implemented by the controller.

- **Superspreader detection**: A superspreader [74] is a source node disseminating data to an abnormal number of destinations (i.e., targets). In some aspects, superspreading is the opposite phenomenon of performing a DDoS attack. A network-wide superspreader detection has been implemented in P4 [116]. If we wanted to use the data collected by INVEST for network-wide superspreader detection, same considerations hold as for network-wide DDoS detection: the information stored in M_i (updated using *dstIP* as flow key) can be used for this purpose.

- **Flow cardinality and average flow size estimation**: Flow cardinality estimation is the problem of estimating the number of distinct flows from a packet stream [137], while average flow size [130] estimation is the problem of estimating the average number of packets per flow. As already shown in Section 7.3.2 and as a side effect of INVEST, both the network-wide flow cardinality n_{tot} and the network-wide average flow size R_{tot} can be estimated by the controller, and can be useful metrics to collect beyond the scope of estimating the traffic volume.

7.6 Performance evaluation

We implemented our INVEST strategy in Python to simulate the traffic volume estimation accuracy in large networks.

Additionally, we implemented INVEST Update in P4 (see Section 7.4 for details) in a commodity Edgecore Wedge-100BF-32X switch equipped with Barefoot Tofino 3.3 Tbps ASIC [2]; the switch supports up to 32 100 Gbps ports. Due to the prohibitive cost of 100 Gbps interfaces, we connected the switch to two servers (Intel(R) Xeon(R), CPU E3-1220 V2 @ 3.10GHz, 16 GB RAM) using 10 Gbps Ethernet interfaces, in the configuration shown in Figure 7.2. The information collected by INVEST in the switch (i.e., packet counter $|S_i|$ and HLL register M_i) is queried using a Command Line Interface that leverages the APIs provided by the switch's vendor.

We also implemented, in P4, a *simple forwarding* strategy for performance comparison and benchmarking purposes. It works as follows: if the destination IP of an incoming packet does not match any entry of an exact match-action table, the packet is forwarded to a specific egress port by applying Least Prefix Match on the entries of another match-action table.

In the following, we will report the results obtained by *(i.)* simulating the INVEST estimation accuracy in large networks and *(ii.)* evaluating the performance of the INVEST P4 implementation in the Tofino-based switch.

7.6.1 Evaluation metrics and simulation settings

Testing flow trace and topology

In our simulations, we used a 2018-passive CAIDA flow trace [3] collected from a 10 Gbps backbone link, lasting 50 seconds. We divided the trace into different time intervals, considering two different widths: $T_{int} = 1$s and $T_{int} = 5$s. In the former case, each of the 50 resulting time intervals includes around 450 thousand packets, while, in the latter, each of the 10 resulting time intervals includes around 2.3 million packets.

We considered two different ISP network topologies: the 45-nodes GÉANT ISP backbone topology [31] and the 100-nodes DEFO synth100 topology [30]. The traffic matrix is generated by adopting a CRC32 hash function to randomly assign each packet to a source/destination node couple in the network. Unless otherwise specified, the output size of the CRC32 hash function is set to the number of nodes in the considered topology (45 for GÉANT and 100 for DEFO) and the source (destination) node is obtained by hashing the source (destination) IP of the packet; then, each packet is forwarded from source to destination on the shortest path. We call this traffic matrix *balanced*, since any source/destination node couple has almost the same probability to be chosen for a source IP/destination IP couple.

Evaluated metric

We consider *relative error* as the evaluation metric. Being $|S_{tot}|$ the exact number of packets in a time interval and $|\hat{S}_{tot}|$ its estimated value, the relative error is defined as the average value of $\frac{||\hat{S}_{tot}|-|S_{tot}||}{|S_{tot}|} \cdot 100\%$ in all the consecutive time intervals (which are 50 for $T_{int} = 1$s and 10 for $T_{int} = 5$s).

Tuning parameters

Unless otherwise specified, HLL register M_i size is set to $m = 2^{11} = 2048$, with *bucket* size of 5 bits. The packet counters $|S_i|$ occupy 32 bit each. The considered flow key is $\{srcIP, dstIP\}$ pair.

7.6.2 Evaluation and comparison with existing strategies

We compare INVEST with two estimation methods:

1. *Sum*: the traffic volume is estimated by summing the packet counters $|S_i|$ as recorded by all the switches, thus neglecting the double counting problem. This trivial strategy is expected to lead to large overestimations.

2. *Sample*: the work [54] prevented double counting by using a sampling-based mechanism, where the packet count of the heaviest flows is kept in the switches in a sample list and delivered to the controller for the estimation of the traffic volume; we described such strategy in Section 7.5. However, a long tail of light flows or the existence of many heavy flows might lead to serious traffic volume underestimations.

According to the parameters specified in the previous subsection, the INVEST's data structure occupies 10272 bits of memory in each switch's data plane (32 bits for $|S_i|$ and $2048 \cdot 5$ bits for M_i). As specified in [54], each entry in a sample list of the Sample strategy requires 96 bits (64 bits to store the flow key, which is $\{srcIP, dstIP\}$ pair, and 32 bits to store the respective packet count). To make a fair comparison between INVEST and Sample, we thus consider a memory occupation for each sample list equal to the memory occupied by INVEST, which means that the number of entries of the sample list is set to $\frac{10272 \text{ bits}}{96 \text{ bits}} = 107$.

Table 7.3 shows the comparison results. In all the cases, INVEST strongly outperforms both Sum and Sample, being its relative error always below 3%. Note that a relative error of 3% is the maximum possible value ensuring that accuracy of practical monitoring applications is not affected [84].

As expected, results show that Sum is an inadequate estimation method (the relative error is higher than 300%) as it does not handle the double counting problem. Moreover, sample has quite bad performance too. In fact, a sampling list size with 107 entries is not large enough to ensure a good estimation, and the relative error is always above 25%. This reveals that INVEST improves the estimation

Table 7.3: Comparison of INVEST with existing strategies

Estimation method	Relative error			
	GÉANT		DEFO	
	$T_{int} = 1$s	$T_{int} = 5$s	$T_{int} = 1$s	$T_{int} = 5$s
INVEST	2.33%	2.05%	2.44%	2.19%
Sum	333.01%	323.00%	412.79%	412.89%
Sample [54]	33.69%	38.78%	25.78%	30.71%

Table 7.4: Estimation accuracy of INVEST parameters

Estimated parameter	Relative error			
	GÉANT		DEFO	
	$T_{int} = 1$s	$T_{int} = 5$s	$T_{int} = 1$s	$T_{int} = 5$s
$\lvert S_{tot} \rvert$	2.33%	2.05%	2.44%	2.19%
R_{tot}	1.92%	1.91%	2.39%	2.06%
n_{tot}	1.57%	1.91%	1.57%	1.91%

Table 7.5: Accuracy of INVEST with different flow key types

Flow key	# Distinct flows n_{tot} in T_{int}		Variance σ^2 of F_{tot} in T_{int}		Relative error			
					GÉANT		DEFO	
	1s	5s	$T_{int} = 1$s	$T_{int} = 5$s	$T_{int} = 1$s	$T_{int} = 5$s	$T_{int} = 1$s	$T_{int} = 5$s
srcIP	∼ 27K	∼ 67K	∼ 26K	∼ 111.7K	20.15%	26.43%	24.80%	32.35%
dstIP	∼ 22K	∼ 58K	∼ 46.5K	∼ 288K	23.94%	28.64%	29.13%	34.80%
{*srcIP*, *dstIP*}	∼ 47K	∼ 147K	∼ 13.8K	∼ 40K	2.33%	2.05%	2.44%	2.19%
{*srcIP*, *dstIP*, *prot*}	∼ 47.1K	∼ 147.6K	∼ 13.5K	∼ 39.8K	2.36%	2.73%	2.56%	2.37%
Unique packet id [36]	∼ 450K	∼ 2300K	0	0	0.48%	3.10%	0.48%	3.10%

accuracy of more than 20% while occupying the same amount of memory in the switch.

Table 7.4 reports the estimation accuracy of the INVEST parameters, namely $|S_{tot}|$ (shown also in Table 7.3), R_{tot} and n_{tot}. It can be seen that, in most of the cases, the relative error in the estimation of R_{tot} and n_{tot} is below 2%, meaning that the adoption of Eq. 7.1 to estimate R_{tot} and the usage of HLL to estimate n_{tot} are both effective means to keep the relative error of $|S_{tot}|$ low.

7.6.3 Sensitivity analysis

We now evaluate how the accuracy of INVEST is sensitive to different tuning parameters.

Sensitivity to flow key types

Table 7.5 shows how INVEST behaves when the HLL registers are updated considering different flow key types. As remarked in Section 7.3.3, as a consequence of Theorems 1 and 2 when the flows are not sufficiently fine-grained (e.g. *srcIP* or *dstIP*), high relative errors in the estimation occur (between 20% and 35%), as the number of distinct flows traversing each switch n_i is not large enough.

Instead, when a flow is identified by the $\{srcIP, dstIP\}$ pair key, which implies finer-grained flows, the relative error significantly drops and is under 3%. We also considered as flow key $\{srcIP, dstIP, protocol\}$, where *protocol* (in brief *prot*) is, for instance, UDP, TCP, ICMP; in this case, the relative error is almost the same as for $\{srcIP, dstIP\}$, without any notable improvement. We thus decided to adopt $\{srcIP, dstIP\}$ pair as default flow key for our evaluations, since it reduces the operations to be performed by INVEST in the data plane with respect to $\{srcIP, dstIP, prot\}$. For completeness, Table 7.5 also reports the relative error if INVEST was updated considering *unique packet ids*, similarly to [36]. In this case, the relative error caused by the estimation $\hat{R}_{tot}$ is zero. Results show that the relative error in both T_{int} widths is inline with the standard error of HyperLogLog, which is $\frac{1.04}{\sqrt{2048}} = 2.3\%$. However, as already said, ensuring unique packet ids in real scenarios is challenging and this case was not implemented in hardware.

Sensitivity to time interval width

Table 7.5 also shows the relative error of INVEST when different time interval widths are considered. A larger width is naturally accompanied by a larger number of monitored flows n_{tot}, which should have a good effect on traffic volume estimation according to Theorems 1 and 2 of Section 7.3.3; however, a higher variance σ^2 of flow packet counts in F_{tot} is also expected (see the table), and this negatively impacts on the estimation. Increasing the time interval width has thus a counteracting effect on traffic volume estimation but, unfortunately, the negative effect due to higher flow packet count variance tends to dominate. Adopting fine-gained flow keys can effectively mitigate such effect, since considering more flows in the network leads on average to less packets per flow and reduces the flow packet count variance.

Sensitivity to network topology

If the same flow trace and flow key type are considered, obviously the variance of flow packet count in F_{tot} does not change when considering GÉANT and DEFO topologies. Since DEFO has a larger number of nodes, each switch i is assigned a smaller number of flows n_i. As we explained in Theorem 2 of Section 7.3.3, smaller number of flows n_i leads to a higher variance on the estimation of R_{tot}: this is why the relative error of traffic volume estimation $|\hat{S}_{tot}|$ is generally slightly higher in DEFO.

Sensitivity to HLL register size m

Figure 7.3 shows how INVEST performs by varying the HLL register size m. Intuitively, the relative error decreases as m increases. When the HLL register size m is small (e.g., 2^4 and 2^5), the relative error is very significant. While increasing

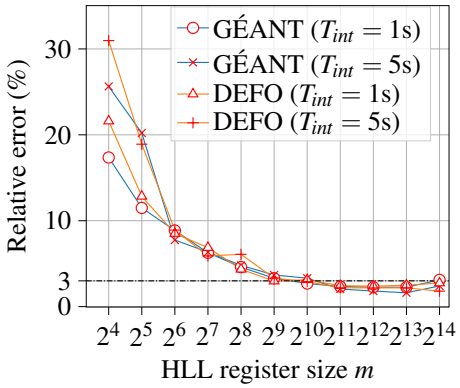
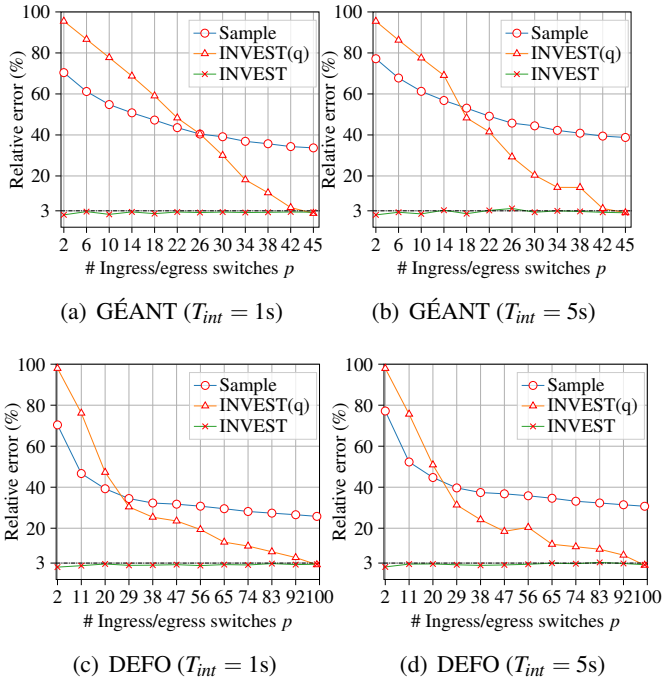


Figure 7.3: Sensitivity to HLL register size m

m, at a certain point (i.e., when $m = 2^{11}$) the curve flattens to a relative error that is always lower than 3%.

Sensitivity to flow distribution



(a) GÉANT (T_{int} = 1s) (b) GÉANT (T_{int} = 5s)

(c) DEFO (T_{int} = 1s) (d) DEFO (T_{int} = 5s)

Figure 7.4: Sensitivity to flow distribution

As described in Section 7.6.1, in previous evaluations we have considered a *balanced* traffic matrix, where any node in the network has the same probability

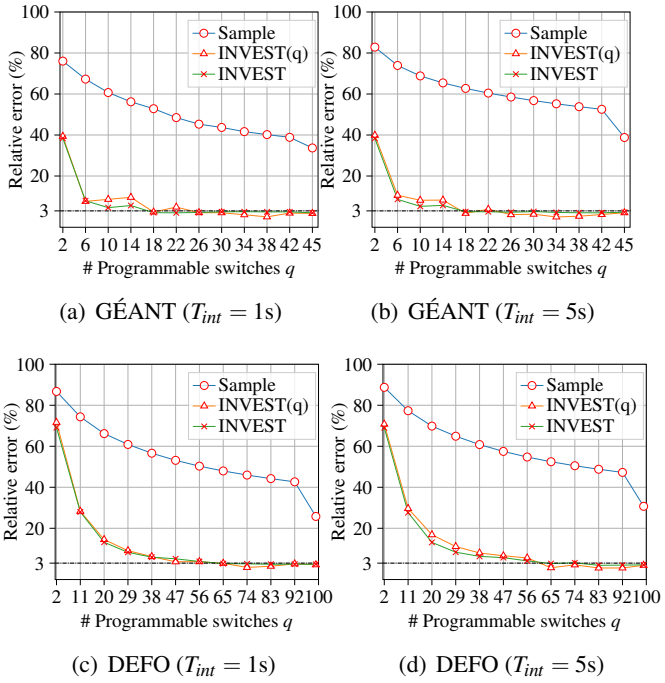
to be picked as source (ingress) or destination (egress) of a traffic flow. Here, we want to see how INVEST performs in the case of *unbalanced* traffic matrices. To do so, instead of considering any node in the ISP topology as possible ingress or egress node for the flows, we select a subset of nodes with cardinality p as possible sources/destinations and we use the CRC32 hash function (with output size p) to assign the flows' source and destination IPs to the nodes in that subset. Once the flows' source and destination IPs have been assigned to the p switches, they are routed as before on the shortest path. Thus, with this procedure, it is possible to tune the skewness of the flow distribution: the smaller p is, the more skewed the distribution is, leading to large variances on flow packet counts in the different switches.

As remarked in Section 7.3.3, as a consequence of Theorem 2, selecting the *top-k* switches for traffic volume estimation is beneficial in the case of unbalanced traffic matrices, since it ensures to keep $Var[\hat{R}_{tot}]$ small. To evaluate the impact of the *top-k* selection on the overall INVEST strategy, we introduce *INVEST(q)*: INVEST(q), instead of only considering the *top-k* switches in the estimation of $\hat{R}_{tot}$, considers all of them, that is, Eq. 7.1 in Section 7.3.2 is replaced with $\hat{R}_{tot} = \frac{1}{q}\sum_{i=1}^{q}\hat{R}_i = \frac{1}{q}\sum_{i=1}^{q}\frac{|S_i|}{\hat{n}_i}$. In the case that $|S_i| = \hat{n}_i = 0$, we define $\hat{R}_i = 0$.

Figure 7.4 shows the sensitivity of INVEST to different flow distributions. For completeness, we include in the evaluation also the Sample strategy, while we do not show results for Sum, which always leads to very large relative errors. The relative error of Sample decreases as p increases. However, when the traffic matrix is balanced, i.e., when p is equal to the overall number of nodes in the network, the relative error is still high (above 25%) as already shown in the previous subsection. INVEST(q) has even worse performance than Sample when p is small in both ISP topologies, since the zero value of $\hat{R}_i$ related to the nodes that are not traversed by any flow compromises the estimated $\hat{R}_{tot}$. In contrast, INVEST always leads to good estimation performance: no matter what network topology, time interval width or flow distribution is considered, the relative error of INVEST is always around 3%. Especially, using the *top-k* switches for estimating the traffic volume is shown to be very effective, especially when the traffic matrix is strongly unbalanced.

Sensitivity to number of programmable switches

Unlike previous evaluations, in this subsection we consider a hybrid network composed by both programmable switches and legacy devices (e.g., Openflow-based or SNMP-based switches) when the traffic is *balanced*. Only some nodes are programmable switches able to implement the INVEST Update strategy, while the remaining legacy devices are assumed not to provide any information to the SDN controller for the estimation of $|S_{tot}|$. The incremental deployment of programmable switches in the hybrid network is performed using the algorithm designed in [54], where the nodes leading to the largest union of distinct flows with previously-deployed programmable switches are iteratively replaced to ensure the



(a) GÉANT ($T_{int} = 1$s) (b) GÉANT ($T_{int} = 5$s)

(c) DEFO ($T_{int} = 1$s) (d) DEFO ($T_{int} = 5$s)

Figure 7.5: Sensitivity to number of programmable switches q in a hybrid network

highest possible flow visibility, until a number q of switches have been substituted with programmable equipment.

Figure 7.5 reports the sensitivity of INVEST performance to a variation on the number of programmable switches q deployed in such a hybrid network. We include in the figure also Sample and INVEST(q) strategies. When more programmable switches q are deployed, the relative error of Sample decreases smoothly, since more sampled flows are reported to the controller. However, as before, the estimation is jeopardized by the missing statistics on the long tail of small flows. Conversely, INVEST and INVEST(q) have good and comparable performance: the relative error of INVEST(q) is slightly lower than of INVEST when q is large, since INVEST(q) considers in the estimation more programmable switches, which may be beneficial when the traffic is balanced (but leads to much worse performance when the traffic is unbalanced, as shown in the previous evaluation). Another observation is that, being programmable only around 40% of the switches in a hybrid network and being them deployed following the strategy proposed in [54], INVEST is able to estimate $|S_{tot}|$ accurately (i.e., relative error $< 5\%$).

7.6.4 Evaluation of impact on network performance

We used the testbed shown in Fig. 7.2 to evaluate the performance of our P4 implementation of INVEST Update as installed in a carrier-grade programmable switch.

Table 7.6: Network performance of INVEST in a commodity switch

Type	iPerf size	Throughput	Jitter	Packet loss	Average additional processing time w.r.t. simple forwarding
TCP	64 KB	9.44 Gbps	/	/	41 ns
TCP	128 KB	9.44 Gbps	/	/	36 ns
UDP	500 B	0.96 Gbps	0.004 ms	0.016%	30 ns
UDP	1000 B	1.79 Gbps	0.004 ms	0.04%	35 ns
UDP	1470 B	3.02 Gbps	0.004 ms	0.04%	34 ns
UDP	3000 B	4.91 Gbps	0.004 ms	0.06%	35 ns
UDP	6000 B	8.64 Gbps	0.005 ms	0.044%	39 ns
UDP	9500 B	9.46 Gbps	0.008 ms	0.0092%	29 ns

We used iPerf [32] to generate packets of different kind and size and indirectly measure the network performance of the proposed strategy. The results are reported in Table 7.6. We first generated 10 Gbps of TCP traffic with iPerf buffer size of 64 KB and 128 KB respectively. In both cases, the throughput reached more than 9.4 Gbps, meaning that the TCP traffic could be processed at line-rate (remind that the testbed includes 10 Gbps interfaces). We then generated 10 Gbps of UDP traffic with different iPerf sizes ranging from 500 Bytes to 9500 Bytes. Using canonical (e.g., 1470 Bytes) datagram sizes, the throughput could not approach 10 Gbps. The issue is not related to inefficiencies of the INVEST strategy, but to the inability of our sender server to generate the required number of packets at the right speed, being thus its CPU processing capability a bottleneck. With larger datagram sizes, results show that the throughput signficantly increases and jitter marginally increases, while packet loss is kept below 0.05% in all the experiments. When the datagram size was set to 9500 Bytes, the throughput was 9.46 Gbps, indicating that the CPU of the sender server was not a bottleneck anymore, and INVEST could still process UDP datagrams at line-rate.

Additionally, we embedded two registers in our P4 program to monitor the *packet processing time*. One was placed in the ingress pipeline and stored the timestamp t_{in}, to record when a packet enters the switch; the other was instead placed in the egress pipeline to record the timestamp t_{out} once the packet had been processed. Per-packet processing time can thus be calculated as $t_{out} - t_{in}$. Results show that INVEST only requires around 40 ns more than simple forwarding, which is an acceptable time overhead.

7.6.5 Evaluation of resource usage

Table 7.7 shows the switch's data plane resources required by INVEST Update and simple forwarding implementations. To ensure a fair comparison, the INVEST Update implementation also includes the simple forwarding logic for packet forwarding. INVEST plus simple forwarding requires around 25% stages (see Section 7.4)

Table 7.7: Normalized switch resource usage of INVEST

Strategy	No. stages	SRAM	TCAM	No. ALUs	PHV size
Simple forwarding	16.67%	2.5%	8.33%	4.17%	7.30%
INVEST Update + Simple forwarding	41.67%	3.23%	9.03%	8.33%	7.68%

more than simple forwarding.

Simple forwarding needs 2.5% of the total available SRAM and 8.33% of TCAM. Instead, INVEST Update plus simple forwarding uses only 3.23% of total SRAM, which means that the occupied memory by both the packet counter $|S_i|$ and HLL register M_i is quite low. Additionally, INVEST needs 31 entries in a ternary match table (see 7.2) to compute the values in the HLL register, occupying additional 0.7% of TCAM with respect to what is needed by simple forwarding.

The number of needed ALUs gives an indication on the computational resource usage. Only 8.33% of total ALUs are used by INVEST plus simple forwarding to process the packets, and the number of ALUs is almost doubled with respect to simple forwarding.

The packet header vector (PHV) size indicates the amount of packet header information that can be passed across the pipeline stages. INVEST plus simple forwarding uses only 7.68% of PHV and, compared to the 7.30% of simple forwarding, means that INVEST requires to pass across stages only few additional customized metadata.

7.7 Related Work

7.7.1 Traffic volume estimation

The problem of estimating the number of packets in the network in a given time window is not new and is fundamental to support network-wide monitoring, as explained in Section 7.5. However, a common nomenclature for this metric has never been defined. For instance, Lawniczak *et al.* [86] call it *number of packets in transit (NPT)* and focus on the number of packets that, in a given instant, are on their routes to their destinations. Salah *et al.* [107], focusing on the incoming traffic to a host, refer to the term *packet rate*, which is the number of packets delivered to the host in a one-second time window: clearly, the same term can be used, in our context, to specify the number of unique packets traversing the network in a second. More recently, Basat *et al.* [36] have introduced the term *network traffic volume*, or simply *traffic volume*, to indicate the number of unique packets flowing in the network in a given time interval: in this chapter, we use this latter name.

In their works, Basat *et al.* [36][37] have proposed a distributed traffic volume estimation strategy that, as ours, rely on HLL-like cardinality estimation, is explicitly designed to avoid the double counting problem and can be executed in programmable switches' data plane. In both papers they require that, to estimate the

traffic volume exploiting the merge property of HLL, a unique identifier for each packet is needed. However, non straightforward mechanisms should be adopted for assigning a unique key to each packet [138][90] and, for this reason, it may be infeasible to adopt such a solution on high-speed carrier-grade networks. Conversely our solution, which exploits the merge property of HLL as well, uses aggregated per-flow information for an accurate estimation of the traffic volume, being a more practical generalization of the strategies proposed in [36][37].

Ding *et al.* proposed a strategy to estimate the traffic volume without double counting packets. It consists on locally storing the heaviest flows in a *sample list* maintained in the switches' data plane. The controller can prevent, when collecting the sample lists from multiple switches, from double counting the packets belonging to the same flow. The limitation of this solution is that the estimation accuracy significantly depends on the size of the sample lists. Considering currently available memory in programmable switches, the long tail of small flows is neglected in the traffic volume estimation. Contrariwise, INVEST can estimate the traffic volume of large data streams, also including the small flows long tail, with good accuracy and low memory occupation.

7.7.2 Network flow cardinality estimation

Many sketch-based algorithms for estimating the cardinality of large data streams have been proposed in literature, including Linear Counting [124], Multiresolution Bitmap [59], PCSA [64], LogLog [57] and HyperLogLog [63]; it has been proven that HyperLogLog is able to achieve the same accuracy of the other methods by requiring much less memory.

With the advent of programmable data planes, estimating the cardinality of flows directly in the data plane pipeline has become an appealing solution to enhance network monitoring. Recently, AROMA [37] proposed and implemented a customized HLL algorithm in P4, where each HLL register cell requires 32 bits instead of the 5 bits required by standard HLL [63], breaking the best trade-off between estimation accuracy and memory occupation as evaluated in [63]. Instead, INVEST implements the standard Update operation of HLL in P4, without any need for additional memory occupation.

7.8 Concluding remarks

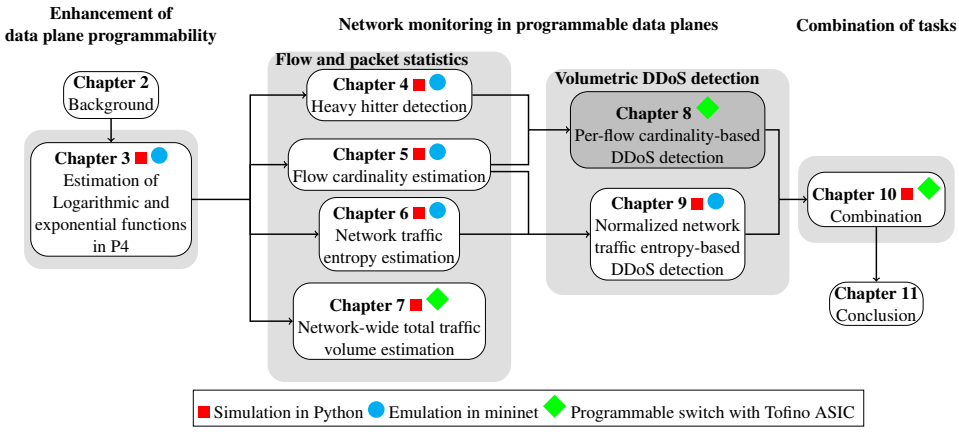
In this chapter we presented INVEST, a novel traffic volume estimation method that exploits modern data-plane programmable switches to jointly estimate number of flows, average flow size and total packet count in the network, using a limited number of sampling locations and robust against packet double counting. We provided the theoretical justification of why the method is an unbiased estimator and can work with a relatively small number of measurement locations. In addition, we overcame the strict resource constraints of real P4-programmable hardware

switches, to successfully implement our logic in an Edgecore commodity switch equipped with Tofino ASIC.

Our experimental evaluation showed that INVEST can estimate the traffic volume with high accuracy (relative error lower than 3%) and low memory occupation (around few KB per switch), outperforming existing strategies. INVEST works well also *(i.)* in the case of strongly unbalanced traffic matrices and *(ii.)* when only 40% of the network switches implement it. It also ensures line-rate packet processing, with only marginal time overhead with respect to a naïve forwarding strategy, and does not require any priori knowledge on the network topology, on routing and on flow distribution.

8

Flow cardinality-based DDoS detection



Chapter 5 presents a possible approach for the flow cardinality estimation in programmable data planes. A sensible curiosity that follows is: what is the potential application by using flow cardinality estimation? In this chapter, we first introduce the BACON data structure based on sketches, to estimate per-destination flow cardinality, and theoretically analyze it. Then we employ it in a simple in-network DDoS victim identification strategy to detect the destination IPs for which the number of incoming connections exceeds a pre-defined threshold. We describe its hardware implementation on a Tofino-based programmable switch using the domain-specific P4 language, proving that some limitations imposed by real hardware to safeguard processing speed can be overcome to implement relatively complex packet manipulations. Finally, we present some experimental performance measurements, showing that our programmable switch is able to keep processing packets at line-rate while performing volumetric DDoS detection, and also achieves a high F1 score on DDoS victim identification.

This chapter is based on the paper "In-Network Volumetric DDoS Victim Identification Using Programmable Commodity Switches" submitted to IEEE Transac-

tions on Network and Service Management (Under review).

8.1 Introduction

Distributed Denial-of-Service (DDoS) attacks are a critical security threat to modern telecommunication networks; not only do they cripple live services for legitimate users, but also cause large operational burdens on operators, which must dedicate significant resources to detecting and mitigating them. As the number and size of botnets and DDoS attacks persistently increases [108], so does this workload. In particular, *volumetric* DDoS attacks, designed to overwhelm network and server capacity, are among the most common and dangerous DDoS attacks [105].

Many techniques to both perform and detect volumetric DDoS attacks are documented in the scientific literature. Among the latter, a rather common feature of such attacks is the exploitation of a (large) number of (capacious) source hosts to direct a considerable amount of packets to a specific victim destination [134][93][74]. Attackers use seemingly legitimate TCP, UDP, or ICMP packets in volumes large enough to overwhelm network devices and servers, or deliberately incomplete packets designed to rapidly consume all available computing, storage, and transmission resources in servers. A majority of such attacks also make use of spoofed source IP addresses (e.g. DNS and NTP amplification DDoS attacks), that is, they forge the address that supposedly generated the requests to prevent source identification; this implies that tracking attack source IPs is usually ineffective. On the other hand, DDoS victim addresses cannot be spoofed, and identifying the victims is a useful step for network operators to mitigate such attacks.

This chapter describes the implementation and validation of volumetric DDoS victim detection and identification directly in Tofino-based [2] P4-enabled commodity switches. The proposed logic copes well with the aforementioned restrictions of data plane programmability. That is, unlike other recent works on this subject, we actually implemented and tested our technique in a real switch instead of limiting the work to a P4 simulator, such as the Behavioral model [13]. To that end, after describing some background mathematical results and data structures required by our technique, and we then propose two main contributions: *BACON Sketch* and *INDDoS*. BACON is a new sketch (a probabilistic data structure) combining a Direct Bitmap [59] and a Count-min Sketch [50], which allows switches to estimate the number of distinct flows (i.e., packets with the same flow key) contacting the same destination host. INDDoS is a simple volumetric DDoS victim identification strategy built on top of BACON Sketch to identify the destination IPs contacted by a number of source IPs greater than a threshold in a given time interval. We include extensive theoretical analysis and detail the modifications required to implement our approach in physical resource-constrained P4 Tofino switches and provide some insights on the integration of INDDoS in a full DDoS defense system, including attack detection and consequent mitigation steps. The results show that, using optimal parameters derived from our theoretical analysis,

our implementation can reach an F1 score higher than 0.95 on a real flow trace [3] captured on a 10 Gbps backbone link, without performance degradation in the switch's packet-processing capabilities.

To sum up, we make the following contributions in this chapter:

- We propose BACON Sketch, a simple but effective data structure to estimate the number of distinct flows to each destination.
- Based on BACON Sketch, we design a strategy, named INDDoS, to identify DDoS victims that is contacted by a large number of hosts in the network.
- We theoretically analyze the estimation accuracy of BACON Sketch and victim detection accuracy of INDDoS.
- We implement the prototype of INDDoS in a programmable switch equipped with Tofino ASIC while respecting the rigorous hardware constraints.

The work has been carried out within the GÉANT [6] GN4-3 project.

8.2 Basic knowledge and used compact data structure

8.2.1 Markov's inequality

Given a non-negative random variable X and a positive value a, Markov's inequality defines a constant upper bound for the probability that satisfies $\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$, where $\mathbb{E}[X]$ represents the expected value of X.

8.2.2 Direct Bitmap

Direct Bitmap [59] is a simple data structure that can be used to estimate the number of distinct flows occurring in a packet stream (also called *flow cardinality*): it is based on a bit array called *Bitmap register* and relies on one or more *hash functions*. Initially, all m cells in the Bitmap register are set to 0. When a packet arrives, its flow key is hashed: the hashed key indicates the index of the register cell to consider. The value of the indexed cell is set to 1 if it was previously 0, otherwise it is not updated. Note that packets sharing the same flow key are always hashed into the same cell, while different flows are hashed to different cells unless a collision occurs. The number of distinct flows can then be (under-)estimated by counting the number of bits with value 1 in the register.

8.2.3 Count-min Sketch

The estimation of *per-flow packet count* (i.e., number of packets carried by any flow in the network during an interval of observation) can be performed by using a *Count-min Sketch* [50], a probabilistic and memory-efficient data structure which

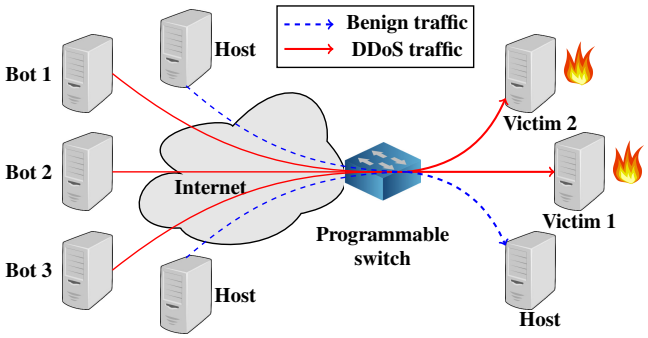


Figure 8.1: Deployment scenario

implements *Update* and *Query* operations: the former is responsible for continuously updating the sketch to count incoming packets in the switch, while the latter retrieves the estimated number of incoming packets for a specific flow. Count-min Sketch relies on d pairwise-independent hash functions, each with an output of size w. The data structure is composed by a matrix of $d \cdot w$ counters: the accuracy of packet count estimation in Count-min Sketch increases as d or w increase, and vice versa.

8.3 In-network DDoS victim identification

8.3.1 Threat model and deployment scenario

Threat model

In this chapter, we focus on volumetric DDoS attacks against victim destinations in the network. For the purpose of overwhelming the available resources of the victims, an attacker exploits a large number of distributed hosts (e.g. bots in a botnet) to frequently send traffic to the target host(s) (e.g., a web server). The attacker sources are usually spoofed to evade detection, which, coupled with the fact that each source may send only a small amount of traffic to the victim (in a stealthy volumetric DDoS attack), makes identifying attack sources (also called *superspreaders*) less efficient than focusing on destinations for DDoS mitigation. Unfortunately, there exists legitimate network events, such as flash crowds, that exhibit similar characteristics to volumetric DDoS attacks, making a suspiciously high number of sources contacting a destination a necessary but not sufficient condition to determine whether a destination is under attack. To further discriminate DDoS attacks from flash crowds in those destinations, a possible solution can be found in [135], where the correlation of flows is used to determine whether they are malicious or not.

Deployment scenario for the proposed DDoS detection

We target an ISP network, for which the best placement of our DDoS detection functionality involves deploying programmable switches at the edge of the network, so that, as shown in Figure 8.1, at least one switch has visibility on all flows towards each IP destination. Therefore, at least one switch is in the best place to estimate the number of source hosts contacting any destination host. Once a DDoS victim is identified, ideally after a more thorough analysis step at a centralized controller, any border programmable switch can also be used to limit the traffic rate towards it.

8.3.2 BACON Sketch

For the purpose of measuring flow cardinality directly inside switches, we combined Direct Bitmap registers [59] and Count-min Sketch [50] in a new sketch, which we named BACON (BitmAp COuNt-min) Sketch for the sake of brevity. As shown in Figure 8.2, in BACON Sketch the counters of Count-min Sketch are replaced with a m-sized Bitmap register, hence the size of BACON is $d \times w \times m$, where $d \times w$ is the size of Count-min Sketch. BACON Sketch enables the estimation of per-destination flow cardinality, with flows identified by different flow keys key_{src}, using very little memory in the switch. Two different flow keys are considered for each packet: the key key_{src} must include the source IP of the flow together with any subset of {source port, destination IP and port, protocol} without loss of generality, and the choice of key_{src} depends on the requirements of operators. Likewise, the flow key of the destination host, denoted by key_{dst}, can be either the destination IP or the {destination IP, destination port} pair.

Formally, BACON Sketch solves the following problem. Given:

- a packet stream S
- a Bitmap register size m
- a number of hash functions in Count-min Sketch d
- an output size of hash functions in Count-min Sketch w
- a time interval T_{int}
- a flow key key_{dst}

compute $\hat{E}_{dst}$, the estimated value of E_{dst} in T_{int}, where E_{dst} is the number of flows (identified by different key_{src} keys) that contact the destination host identified by key_{dst}.

Algorithm 9 shows the pseudo code of BACON Sketch. For each incoming packet, the switch hashes the flow key key_{src} with hash function h_{bm} and converts the hashed value to be within range $[0, m-1]$: this value, named *bucket*, is the index in the Bitmap register (Line 7). Then each of the d pairwise-independent

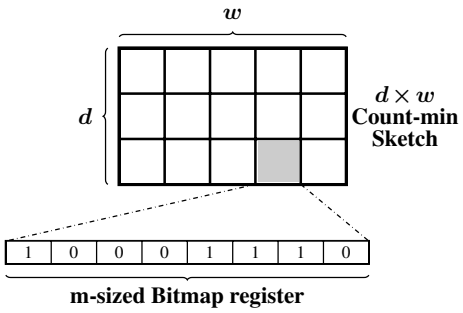


Figure 8.2: Data structure of BACON Sketch

Algorithm 9: BACON Sketch

Input: Packet stream S, where each packet is characterized by key_{src} and key_{dst}

Output: Estimated number of distinct flows contacting the packet's destination key_{dst} (i.e., $\hat{E}_{dst}$)

$d \leftarrow$ Number of hash functions h^i_{cm} in Count-min
$w \leftarrow$ Output size of hash functions h^i_{cm} in Count-min
$m \leftarrow$ Bitmap-register size with hash function h_{bm}
for *Each packet in S (with* key_{src} *and* key_{dst}*)* **do**
    Update(key_{src}, key_{dst})
Function *Update(*key_{src}, key_{dst}*)*:
    $bucket \leftarrow h_{bm}(key_{src})\%m$
    for *Each hash function* h^i_{cm} **do**
        $index \leftarrow (h^i_{cm}(key_{dst})\%w) \cdot m + bucket$
        if $BACON_i[index] == 0$ **then**
            $BACON_i[index] \leftarrow 1$ `// row i`
Function *Query(*key_{dst}*)*:
    $\hat{E}_{dst} \leftarrow 0$
    for *Each hash function* h^i_{cm} **do**
        $id \leftarrow (h^i_{cm}(key_{dst})\%w) \cdot m$
        $E_i \leftarrow \sum_{j=id}^{id+m-1} BACON_i[j]$
        if $min == 0$ **then**
            $\hat{E}_{dst} \leftarrow E_i$
        else if $\hat{E}_{dst} > E_i$ **then**
            $\hat{E}_{dst} \leftarrow E_i$
    return $\hat{E}_{dst}$

hash function in BACON, denoted by h^i_{cm}, hashes the packet's key_{dst} to the slot $(h^i_{cm}(key_{dst})\%w)\cdot m + bucket$ in each row, setting the related register cell to 1 (Lines 8-11). Note that this way to compute the cell's index to be updated assumes that the Bitmap registers' cells in each row are progressively numbered from 0 to $m\cdot w-1$, much like in an array. Concerning the *Query* operation for any key_{dst}, the involved Bitmap register in row i (i.e., the register indexes from $(h^i_{cm}(key_{dst})\%w)\cdot m$ to $h^i_{cm}(key_{dst})\%w)\cdot m + m - 1$) estimates the cardinality of key_{dst} by computing the sum of values in the register cells (i.e., the number of 1s). This is done for each row and the minimum estimated cardinality among all d rows is returned (Lines 14-20).

8.3.3 In-network cardinality-based DDoS victim identification

Using BACON Sketch, we propose a simple in-network volumetric DDoS detection mechanism, which we named INDDoS, aiming to identify likely DDoS victims, i.e., those hosts that are contacted by an abnormally high number of source IPs, and hence the associated network flows.

Formally, the problem is formulated as follows.
Given:

- a number of source IPs n
- a BACON Sketch with size $d\times w\times m$ $(m<n)$
- a DDoS threshold fraction θ (threshold: θn)
- a time interval T_{int}

return all destinations key_{dst}, named *DDoS victims*, that satisfy $\hat{E}_{dst} > \theta n$ within time interval T_{int}, where $\hat{E}_{dst}$ is the estimated number of sources contacting key_{dst} and obtained by querying the BACON sketch.

From this point onward, without any loss of generality, we will use *src* (the source IP) as key_{src}, and *dst* (the destination IP) as key_{dst}, to estimate how many flows from different source IPs are trying to contact a destination IP in a given time interval T_{int}. Observe that the proposed strategy, which requires updating and querying the BACON sketch for each incoming packet towards any destination *dst* during the time interval, interacts with a remote controller only when at least one attack is detected in the same interval, otherwise no communication between controller and switches takes place.

8.3.4 Theoretical analysis

In this section we present a theoretical analysis of the accuracy of our proposed DDoS detection strategy.

Error bounds for estimated flow cardinality

Theorem 4. *With probability at least* $1-(\frac{1}{2})^d$*, the cardinality estimation* $\hat{E}_{dst}$ *from BACON Sketch satisfies* $E_{dst} - 2(n - m(1-e^{-\frac{n}{m}})) < \hat{E}_{dst}$.

Proof. Let us start with the Bitmap register. Since a register cell is touched (i.e., set to 1) as soon as it is selected for one incoming element, the probability of any cell X_i to be touched after processing one element (packet) can be expressed as:

$$\mathbb{P}(X_i = 1) = \frac{1}{m}$$

so, the probability of i-th register cell to be untouched is:

$$\mathbb{P}(X_i = 0) = 1 - \mathbb{P}(X_i = 1) = 1 - \frac{1}{m}.$$

After processing E_{dst} different elements (where E_{dst} is the true number of *src* contacting *dst*) we then have:

$$\mathbb{P}(X_i = 0) = (1 - \frac{1}{m})^{E_{dst}} = ((1 - \frac{1}{m})^m)^{\frac{E_{dst}}{m}}.$$

Since $\lim_{m\to\infty}(1-\frac{1}{m})^m = e^{-1}$ and in our case m is large (i.e., ≥ 128), $(1-\frac{1}{m})^m \approx e^{-1}$ and we have:

$$\mathbb{P}(X_i = 0) \approx e^{-\frac{E_{dst}}{m}}.$$

Hence, the probability of a cell to be touched after E_{dst} elements is:

$$\mathbb{P}(X_i = 1) = 1 - \mathbb{P}(X_i = 0) = 1 - e^{-\frac{E_{dst}}{m}}.$$

Since there are m cells in each Bitmap register, the expectation of our Bitmap cardinality-estimation (i.e., the expected number of 1s in the register), for destination IP *dst*, is:

$$\begin{aligned}
\mathbb{E}[\hat{E}_{dst}^{Bitmap}] &= \sum_{i=1}^{m} X_i \cdot \mathbb{P}(X_i = 1) + X_i \cdot \mathbb{P}(X_i = 0) \\
&= \sum_{i=1}^{m} (1 \cdot \mathbb{P}(X_i = 1)) + 0 \\
&= \sum_{i=1}^{m} 1 - e^{-\frac{E_{dst}}{m}} = m(1 - e^{-\frac{E_{dst}}{m}}).
\end{aligned}$$

Note that, when there are no collisions in Count-min Sketch, $\mathbb{E}[\hat{E}_{dst}] = \mathbb{E}[\hat{E}_{dst}^{Bitmap}]$. Since $\hat{E}_{dst}^{Bitmap} \leq E_{dst}$, the difference between $\mathbb{E}[\hat{E}_{dst}]$ and $\mathbb{E}[E_{dst}]$[1] is:

$$f(E_{dst}) = \mathbb{E}[E_{dst}] - \mathbb{E}[\hat{E}_{dst}] = E_{dst} - m(1 - e^{-\frac{E_{dst}}{m}}).$$

[1]The expectation of E_{dst} is still E_{dst} since E_{dst} is a constant.

The derivative of $f(E_{dst})$, denoted by $f'(E_{dst})$, is:

$$f'(E_{dst}) = 1 - e^{-\frac{E_{dst}}{m}}.$$

Since $e^{-\frac{E_{dst}}{m}} < 1$, $f'(E_{dst}) > 0$. For this reason, $f(E_{dst})$ is monotonically increasing and $f(E_{dst}) \leq f(n) = n - m(1 - e^{-\frac{n}{m}})$. Then, applying Markov's inequality yields:

$$\begin{aligned}\mathbb{P}(E_{dst} - \hat{E}_{dst} \geq 2(n - m(1 - e^{-\frac{n}{m}}))) \\ &\leq \frac{E_{dst} - \mathbb{E}[\hat{E}_{dst}]}{2(n - m(1 - e^{-\frac{n}{m}}))} \\ &\leq \frac{n - m(1 - e^{-\frac{n}{m}})}{2(n - m(1 - e^{\frac{-n}{m}}))} = \frac{1}{2}.\end{aligned}$$

Considering that there are d hash functions in the Count-min Sketch part of BACON Sketch:

$$\mathbb{P}((E_{dst} - \hat{E}_{dst}) \geq 2(n - m(1 - e^{-\frac{n}{m}}))) \leq (\frac{1}{2})^d$$

Thus, with probability at least $1 - (\frac{1}{2})^d$,

$$\hat{E}_{dst} > E_{dst} - 2(n - m(1 - e^{-\frac{n}{m}})).$$

■

Theorem 5. *With probability at least* $1 - (\frac{1}{2})^d$*, the cardinality estimation* $\hat{E}_{dst}$ *from BACON Sketch satisfies* $\hat{E}_{dst} \leq E_{dst} + 2m(1 - e^{-\frac{n}{mw}})$.

Proof. Count-min Sketch occasionally writes two different inputs to the same cell; the expected number of such collisions is $\frac{n - E_{dst}}{w}$ [50]. Therefore, in BACON Sketch, the actual average number of different elements being written to the same Bitmap register is $E_{dst} + \frac{n - E_{dst}}{w} < E_{dst} + \frac{n}{w}$, so the expectation of $\hat{E}_{dst}$ becomes:

$$\begin{aligned}\mathbb{E}[\hat{E}_{dst}] = \sum_{i=1}^{m} \mathbb{E}[X_i] = \sum_{i=1}^{m} 1 - e^{-\frac{E_{dst} + \frac{n - E_{dst}}{w}}{m}} \\ &= m(1 - e^{-\frac{E_{dst} + \frac{n - E_{dst}}{w}}{m}}) \\ &< m(1 - e^{-\frac{E_{dst} + \frac{n}{w}}{m}}).\end{aligned}$$

Then, the expectation of $\hat{E}_{dst} - \hat{E}_{dst}^{Bitmap}$ is:

$$\begin{aligned}\mathbb{E}[\hat{E}_{dst} - \hat{E}_{dst}^{Bitmap}] &= \mathbb{E}[\hat{E}_{dst}] - \mathbb{E}[\hat{E}_{dst}^{Bitmap}] \\ &< m(1 - e^{-\frac{E_{dst} + \frac{n}{w}}{m}}) - m(1 - e^{-\frac{E_{dst}}{m}}) \\ &= me^{-\frac{E_{dst}}{m}}(1 - e^{-\frac{n}{mw}}) < m(1 - e^{-\frac{n}{mw}}).\end{aligned}$$

According to Markov's inequality,

$$\begin{aligned}\mathbb{P}((\hat{E}_{dst} - \hat{E}_{dst}^{Bitmap}) \geq 2m(1-e^{-\frac{n}{mw}})) \\ &\leq \frac{\mathbb{E}[\hat{E}_{dst}] - \mathbb{E}[\hat{E}_{dst}^{Bitmap}]}{2m(1-e^{-\frac{n}{mw}})} \\ &= \frac{m(1-e^{-\frac{n}{mw}})}{2m(1-e^{-\frac{n}{mw}})} = \frac{1}{2}.\end{aligned}$$

Since there are d hash functions in Count-min Sketch:

$$\mathbb{P}((\hat{E}_{dst} - \hat{E}_{dst}^{Bitmap}) \geq 2m(1-e^{-\frac{n}{mw}})) \leq (\frac{1}{2})^d.$$

Hence, with probability at least $1-(\frac{1}{2})^d$,

$$\hat{E}_{dst} - \hat{E}_{dst}^{Bitmap} < 2m(1-e^{-\frac{n}{mw}}).$$

Since $\hat{E}_{dst}^{Bitmap} \leq E_{dst}$, $\hat{E}_{dst}$ satisfies:

$$\hat{E}_{dst} < 2m(1-e^{-\frac{n}{mw}}) + E_{dst}.$$

■

Remark. *The lower error bound for the cardinality estimation is* $E_{dst} - 2(n - m(1-e^{-\frac{n}{m}}))$*, while the upper error bound is* $E_{dst} + 2m(1-e^{-\frac{n}{mw}})$*. This implies that we should carefully choose m and w instead of setting the sketch width (i.e., number of columns) as large as possible, due to their counteracting effect on estimation quality.*

Error bounds for DDoS detection

Lemma 1. *With probability at least* $1-(\frac{1}{2})^d$, $R \leq \frac{2n}{w}$*, where R is the overestimation of* E_{dst} *caused by Count-min Sketch.*

Proof. The expectation of R in each Bitmap register is $\mathbb{E}[R] = \frac{n-E_{dst}}{w} \leq \frac{n}{w}$, applying Markov's inequality yields:

$$\mathbb{P}(R \geq \frac{2n}{w}) \leq (\frac{1}{2})^d.$$

■

Given Lemma 1, Theorem 6 reports the *false negative bound* and Theorem 7 the *false positive bound* for INDDoS.

Theorem 6. *When* $E_{dst} \geq \theta n + 2(n - m(1-e^{-\frac{n}{m}}))$*, INDDoS reports dst as a DDoS victim with a probability of at least* $1-(\frac{1}{2})^d$.

Table 8.1: Properties of hash functions

Hash function name	poly	Reversed	init	xor
CRC32	0x104C11DB7	True	0	0xFFFFFFFF
CRC32c	0x11EDC6F41	True	0	0xFFFFFFFF
CRC32d	0x1A833982B	True	0	0xFFFFFFFF
CRC32q	0x1814141AB	False	0	0
CRC32mpeg	0x104C11DB7	False	0xFFFFFFFF	0

Proof. A victim should be reported if $E_{dst} > \theta n$. By Theorem 4, $\hat{E}_{dst} \geq E_{dst} - 2(n - m(1 - e^{-\frac{n}{m}})) \geq \theta n$ with probability at least $1 - (\frac{1}{2})^d$. Therefore, *dst* is reported as a DDoS victim with the same probability if and only if $\hat{E}_{dst} \geq E_{dst} - 2(n - m(1 - e^{-\frac{n}{m}}))$. ■

Theorem 7. *When $E_{dst} \leq \frac{2}{w}n$ and $\theta \geq \frac{4}{w}$, BACON Sketch reports dst as a victim with a probability of at most $(\frac{1}{2})^d$.*

Proof.

$$\begin{aligned}
\mathbb{P}(\hat{E}_{dst} \geq \theta n) &\leq \mathbb{P}(E_{dst} + R) \geq \theta n) \\
&\leq \mathbb{P}(R + \frac{2n}{w} \geq \theta n) \qquad (\textit{due to } E_{dst} \leq \frac{2}{w}n) \\
&= \mathbb{P}(R \geq (\theta - \frac{2}{w})n) \\
&\leq \mathbb{P}(R \geq (\frac{4}{w} - \frac{2}{w})n) \qquad (\textit{due to } \theta \geq \frac{4}{w}) \\
&= \mathbb{P}(R \geq \frac{2}{w}n) \leq (\frac{1}{2})^d \qquad (\textit{by Lemma 1}).
\end{aligned}$$

■

Remark. *The false negative and positive bounds for INDDoS show that increasing the BACON sketch height (i.e., number of rows) improves the victim identification performance (with equal column parameters).*

8.4 Implementation in commodity switches

Figure 8.3 shows a schematic representation of our in-network DDoS detection implementation within P4-enabled commodity switches. We implemented INDDoS with 680 lines of P4_16 code, released as open source at [15]. Our implementation of INDDoS fits entirely in the ingress pipeline, leaving the egress pipeline free for other uses as discussed in Section 8.5.3. Here we report the details of the implementation of BACON Sketch and INDDoS in the Tofino-based hardware switch.

8.4.1 Implementation of pairwise independent hash functions

Since our P4-enabled commodity switch does not natively provide support for pairwise independent hash functions, we varied the usual *poly*, *reverse*, *initial value*,

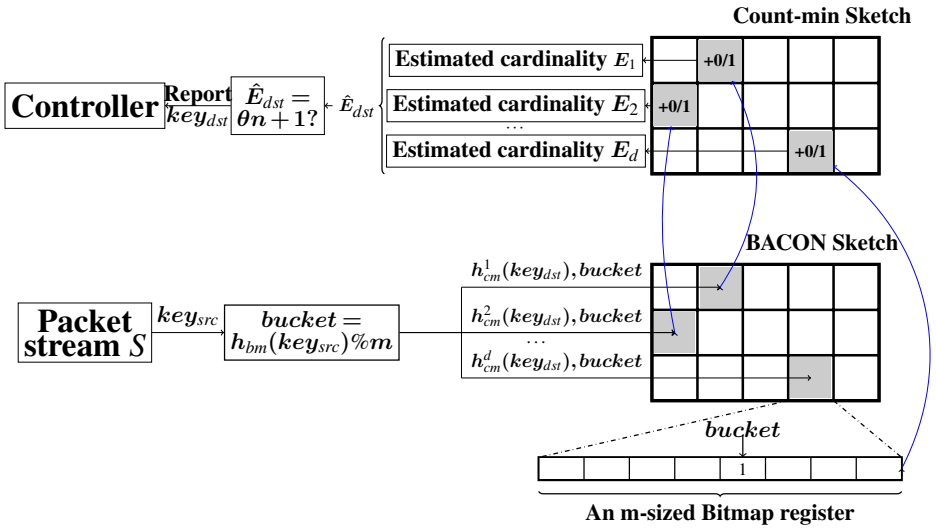


Figure 8.3: Scheme of INDDoS including Update (bottom) and Query (top) operations on BACON sketch as implemented in the commodity switch

and *xor* parameters in the available embedded CRC32 function to produce a set of suitable pairwise independent hash functions, such as CRC32c, CRC32d, CRC32q and CRC32mpeg (see Table 8.1). These hash functions guarantee that the hashed values of the same flow key are independent.

8.4.2 BACON Sketch - Implementation of updates

We implemented BACON Sketch with d rows of $w \times m$-sized registers. Considering that we cannot use more than two arithmetic operations to compute the input of a register (due to the atomic action rule [103]), we cannot calculate the *index* of the cell to be updated as $(h^i_{cm}(dst)\%w) \cdot m + bucket$ (see Alg. 9). Thus, we express *index* by concatenating *bucket* and h^i_{cm} as a binary number (see Figure 8.4), where the least significant bits $index[0 : \log_2(m) - 1]$ represent *bucket*, while the rest (i.e., $index[\log_2(m) : \log_2(m) + \log_2(w) - 1]$) represent $h^i_{cm}(dst)$. In order to assure that $\log_2(w)$ and $\log_2(m)$ are integers, w and m must be powers of two.

The maximum size of registers in the switch is 2^{17}, which means that if $\log_2(m) + \log_2(w)$ is larger than 17, the concatenated *index* is truncated to 17 bits. In order to overcome this limitation, it is possible, as shown in Figure 8.4, to initialize up to $\log_2(m) + \log_2(w) - 17$ registers in the switch, labeled from 0 to $\log_2(m) + \log_2(w) - 18$. The most significant bits of *index* are the label of the registers that should be updated. For instance, if $index[17 :] = k$, the index $index[0 : 16]$ at the register labeled k will be updated to 1. Unfortunately, this approach requires few additional pipeline stages in the switch due to using more registers.

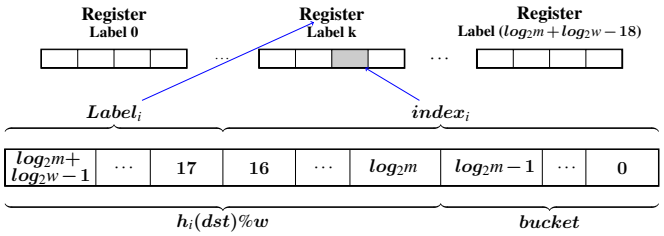


Figure 8.4: BACON Sketch updates to overcome the switch's hardware limitations

8.4.3 BACON Sketch - Implementation of queries

Calculating the sum of values in a Bitmap register is a costly operation inside a switch, since it needs to iteratively load and compute the sum of many individual values. For this reason, we used an *auxiliary* Count-min Sketch to directly store the updated sum of each Bitmap register (see Fig. 8.3). This Count-min Sketch is composed by $d \times w$ counters, each associated with the number of 1s in the corresponding Bitmap register of BACON Sketch. This sketch is updated as follows: when a packet arrives, the $h^i_{cm}(dst)$-th counter may *(i.)* stay the same if $index_i$ in row i of BACON Sketch is already 1 or *(ii.)* increase by 1 if the same index of BACON Sketch is 0. The updated value E_i in the counter $index_i$ is the same as the queried value in row i of BACON Sketch. Finally, the flow cardinality of dst is obtained by using *if-else statements* to take the minimum of estimated cardinalities E_i in each row of the auxiliary sketch.

8.4.4 Implementation of INDDoS

If a queried flow cardinality $\hat{E}_{dst}$ is equal to $\theta n + 1$ for one or more destinations, the switch's data plane prepares a short *digest* packing the destination IPs for which the flow cardinality estimation exceeds the threshold, and sends it to the controller. The equality prevents generating multiple digests for the same dst in one time interval. At the end of a time interval the switch resets all registers and starts a new round of DDoS victim identification.

8.4.5 Limitations hindering the implementation of other solutions

In this subsection, we briefly explain why competing state-of-the-art sketch-based per-flow cardinality estimation algorithms cannot currently be implemented in P4-enabled programmable hardware switches:

Virtual HyperLogLog (vHLL) [126]

we were not able to implement this strategy for three reasons: *(i.)* we could not find a way to count tailing 0s for the Update operation in HyperLogLog; *(ii.)* the required Harmonic mean for querying cardinality is currently not implementable

since P4 does not support the calculation of inverse numbers; *(iii.)* vHLL needs to access several register cells to query the cardinality of a single flow but, in our hardware, the same register cannot be accessed more than once per packet.

SpreadSketch [116] (combination of Multiresolution Bitmap [59] and Count-min Sketch [50])

we were not able to implement this strategy for two reasons: *(i.)* P4 does not support the computation of logarithms, which is necessary for Multiresolution Bitmap to estimate the cardinality (we proposed a logarithm estimation algorithm in P4 in [55], but currently it can only be implemented in Behavioral model and not in hardware); *(ii.)* arithmetic operations on the same metadata, in our hardware, cannot occur more than twice, while more are required here.

Thus, unfortunately, we cannot perform a direct comparison of our strategy with the state of the art since the solutions proposed in literature do not meet the stringent hardware constraints of, and therefore they cannot be entirely implemented in, current Tofino-based programmable switches. We are aware that, in general terms, BACON Sketch is a rather simple solution for per-flow cardinality estimation and more refined strategies have already been proposed. However, as pointed out earlier, BACON is the only solution that can be implemented in existing commodity programmable switches.

8.5 Integrating INDDoS in a full DDoS Defense Mechanism

In this section we explain how a DDoS defense system can benefit from the early detection of potential DDoS traffic in the data plane, as performed by INDDoS, as well as how the latter can benefit from the supervision of the former.

8.5.1 Setting the DDoS detection threshold

The performance of INDDoS depends on selecting an appropriate threshold fraction θ, which determines the minimum number of sources contacting the same destination that trigger an alarm, discriminating between legitimate and abnormal traffic. In principle, to strike a good balance between false positives and false negatives, θ should be chosen so that θn *(i.)* is larger than the largest flow cardinality of any host in normal conditions (i.e., not under attack), and *(ii.)* is not so large that it fails to detect some attacks. Additionally, the threshold should be dynamically adapted to traffic changes.

Even though the definition of a detailed strategy for dynamic threshold setting is beyond the scope of this chapter, we provide some indications on how it could be designed. In the most simple case, it could be set by mirroring and analyzing a large trace of legitimate traffic sampled in each switch. Alternatively, considering

a more complex DDoS detection function at the controller, it could be set via a slow negative feedback control loop: start with a relatively high threshold, and slowly decrease it over time; at some point, the switch will start flagging some destinations as potentially under attack, but the controller could determine whether those are false positives, and if so raise the threshold back up. Finally, when θ changes, the BACON sketch size should be tweaked according to our theoretical analysis, in order to maximize the detection performance of INDDoS.

8.5.2 DDoS attack mitigation inside programmable switches

INDDoS enables the early detection of DDoS attack victims in the data plane pipeline that can be exploited to immediately mitigate them. Once INDDoS has identified a possible DDoS victim and sent a digest to the controller, two simple DDoS victim-based mitigation operations can be triggered in the data plane, i.e., *Drop* and *Rate limit*:

- **Drop**: the controller automatically configures, in all programmable switches in the network, the relevant egress Access Control List (ACL) with identified attack traffic features (e.g. source, destination IPs, ports) in order to drop the packets belonging to likely malicious flows.
- **Rate limit**: a meter is used to aggregate traffic towards the identified victim. If the rate is larger than a pre-defined threshold, the traffic rate is limited.

Although we already implemented both operations in P4, we are keenly aware that neither can actually implement proper DDoS mitigation alone. For that, external aid from the controller or an additional, more precise and computationally expensive detection mechanism (operating on the limited subset of suspicious flows identified by INDDoS) is needed, as outlined in the next subsection.

8.5.3 DDoS attack mitigation outside programmable switches

More refined and complex strategies, making use of external servers, have been proposed in literature for DDoS attack mitigation. Here we describe a few and explain how they can benefit from INDDoS.

Bohatei [60] is a strategy that spins up virtual machines (VMs) in servers for the identification, analysis and mitigation of suspicious traffic as needed: once suspicious traffic is detected, a resource manager determines the type, number, and location of VMs to be instantiated, and such traffic is steered to them for further analysis (and possibly mitigation) with minimal impact on users' perceived latency. Another approach, named Poseidon [136], performs DDoS mitigation by combining the capabilities of programmable switches and of sets of external servers. The additional mitigation functionalities provided by the servers, with respect to those provided by programmable switches, significantly improves the performance with respect to DDoS defense: Poseidon can mitigate sixteen different types of attacks exploiting different protocols (i.e., ICMP, TCP, UDP, HTTP).

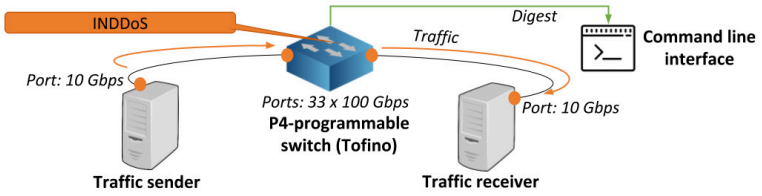


Figure 8.5: The physical testbed for experiments on DDoS victim identification

However, both Bohatei and Poseidon consider a “DDoS-defense-as-a-service scenario", that is, they assume that the DDoS victim is known a-priori. INDDoS is complementary to these approaches since it can effectively detect and identify likely DDoS victims, and hence route only a manageable subset of network traffic towards these detection services.

8.6 Experimental evaluation

We implemented INDDoS in a commodity Edgecore Wedge-100BF-32X switch equipped with Barefoot Tofino 3.3 Tbps ASIC [2], which supports up to 32 100 Gbps ports. Due to the high cost of 100 Gbps interfaces, we connected the switch to two servers (Intel(R) Xeon(R) CPU E3-1220 V2 @ 3.10GHz, 16 GB RAM) using 10 Gbps Ethernet interfaces, as shown in Figure 8.5. We also implemented a P4-based *simple forwarding* strategy for comparison purposes. Packets are sent across the switch via Tcpreplay [28].

8.6.1 Evaluation metrics and settings

Testing flow traces

In our first experiment we used, as done in previous works on DDoS detection [74], a 50 s passive CAIDA flow trace [3] collected from a 10 Gbps backbone link, which we divided into 10 time intervals. Each 5 s time interval contains around 2.3 million packets and 60 thousand distinct source IPs. Note that, although the trace did not contain actual DDoS traffic, we were still able (by using a low, sensitive threshold) to successfully detect outliers, despite the number of incoming connections for such cases being far smaller than in actual volumetric DDoS scenarios. In other words, our experiment is a pressure test: if INDDoS can detect such outliers accurately, by properly setting a threshold between legitimate and DDoS traffic, it should have little trouble in properly distinguishing actual attack traffic where such difference is more marked.

In a second experiment we then considered real DDoS attacks, namely from Booter [108]. Booter is an on-demand service that provides illegal support to launch DDoS attacks targeting websites and networks. We considered four 50 s Booter DDoS attack traces with different number of attack source IPs; Table 8.2

Table 8.2: Properties of DDoS flow traces [108]

DDoS trace name	Packets per second	Attack source IPs
Booter 6	~ 90000	7379
Booter 7	~ 41000	6075
Booter 1	~ 96000	4486
Booter 4	~ 80000	2970

reports their salient properties. For each trace, we split it into 10 time intervals as for the CAIDA trace, and appended each 5s attack trace at the end of its respective legitimate trace. In this way we generated four traces containing a single attack, as well as a fifth *Mixed* trace containing all attacks simultaneously along with the legitimate CAIDA traffic.

Metrics

We chose *recall R* and *precision Pr* as the key metrics to evaluate INDDoS, defined as follows:

$$R = \frac{Count_{DDoSvictim}^{detected/true}}{Count_{DDoSvictim}^{detected/true} + Count_{DDoSvictim}^{undetected/true}}$$

$$Pr = \frac{Count_{DDoSvictim}^{detected/true}}{Count_{DDoSvictim}^{detected/true} + Count_{DDoSvictim}^{detected/false}}$$

The count of DDoS victims is obtained via the command line interface provided by our Tofino-based switch.

In our evaluations, we considered the well known *F1 score* ($F1$) as a compact metric incorporating both precision and recall, and thus measuring the accuracy of our strategy. It is defined as:

$$F1 = \frac{2 \cdot Pr \cdot R}{Pr + R}$$

All results are mean values computed across 10 time intervals.

Tuning parameters

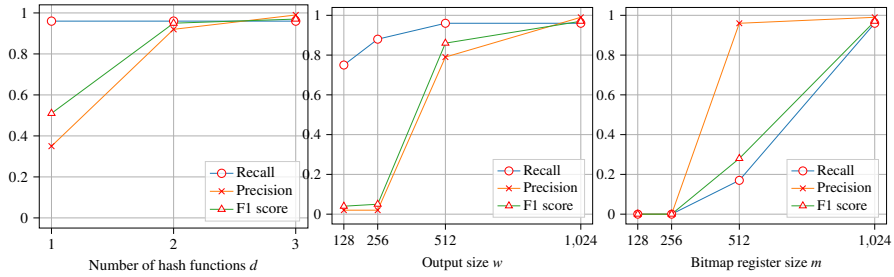
The default BACON Sketch size $d \times w \times m$ is $3 \times 1024 \times 1024$, and the DDoS threshold fraction θ is set to 0.5% as per [74] in the first experiment, and to 1% in the second, to better separate legitimate and DDoS traffic.

8.6.2 Exp 1: evaluation of DDoS victim identification accuracy

According to Theorems 1 and 2, the Bitmap register size m impacts both upper and lower bounds on the quality of BACON Sketch's estimation: increasing m increases the distance between the upper bound of the estimation and the true flow cardinality, but also reduces that between the lower bound and the true value. This

Table 8.3: Comparison of INDDoS performance as a function of BACON Sketch parameters

BACON Sketch size ($d \times w \times m$)	Recall	Precision	F1 score
$3 \times 1024 \times 1024$	0.96	0.99	0.97
$1 \times 2048 \times 1024$	0.98	0.54	0.70
$1 \times 1024 \times 2048$	0.94	0.38	0.54
$5 \times 1024 \times 512$	0.12	1.0	0.22
$5 \times 512 \times 1024$	0.96	0.89	0.92



(a) Sensitivity analysis to d (b) Sensitivity analysis to w (c) Sensitivity analysis to m

Figure 8.6: Sensitivity analysis of INDDoS to the parameters of BACON Sketch

implies that m should be neither too large nor too small. In our flow trace, the largest flow cardinality in each time interval is within $[2^9, 2^{10}]$, so we set m to 1024. Conversely, larger output sizes w generate lower false negative bounds and do not affect the false positive bound, but in order to apply the false positive bound proved in Theorem 7, w should satisfy $w \geq \frac{4}{\theta}$. Therefore, given $\theta = 0.5\%$ in our case, w should be larger than $\frac{4}{0.5\%} = 800$. However, due to hardware limitations, we cannot use arbitrarily large values for w; in fact, the only feasible parameter combinations that satisfy all conditions on m and w, while not exceeding the available pipeline stages, are: $3 \times 1024 \times 1024$ and $1 \times 2048 \times 1024$. Considering that $1 \times 2048 \times 1024$ only uses 1 hash function, the estimated flow cardinality has a high chance (i.e., $\frac{1}{2}$) to be out of both upper bound and lower bound. Note that the number of hash functions $d = 2$ was not considered since 2 hash functions support the same largest register size (i.e., $w \times m = 1024 \times 1024$) as 3 functions within the available pipeline stages. Likewise, 4 hash functions support the same largest register size (i.e., $w \times m = 512 \times 1024$) as 5 functions, but neither can satisfy Theorem 4. Thus, according to the theoretical analysis, the best choice of the sketch size should be $3 \times 1024 \times 1024$ (hence our default choice).

Table 8.3 shows the performance of INDDoS for different BACON Sketch sizes. In all cases listed in the table, all available pipeline stages are required to implement the strategy. Results show good performance on both recall and precision for $3 \times 1024 \times 1024$, which translates to a high F1 score. With only one hash function in BACON Sketch, INDDoS shows high recall, i.e., it is able to identify most DDoS victims, but low precision, meaning that a lot of false identifications

take place. When instead $m = 2048$, the recall of $1 \times 1024 \times 2048$ is as high as for $1 \times 2048 \times 1024$, but its precision decreases by a third. In contrast, when $m = 512$ and the sketch size is $5 \times 1024 \times 512$, the recall significantly decreases leading to a much lower F1 score than for $5 \times 512 \times 1024$. These two results indicate that a larger m causes higher recall but lower precision, and vice versa. The recall for sketch size $3 \times 1024 \times 1024$ is comparable to the one for size $5 \times 512 \times 1024$, but the precision is 10 percentage points higher. Note that in higher-speed networks, where the largest flow cardinality during a time interval may exceed 1024, INDDoS can still work properly even in currently available switches by shrinking the time interval.

8.6.3 Exp. 1: sensitivity analysis of DDoS victim identification

In order to show how the performance of INDDoS is sensitive to different tuning parameters (number of hash functions d, output size of hash functions w and Bitmap register size m) in BACON, we conducted some experiments by varying tuning parameters individually, while respecting the resource constraints of our switch. The results are reported in Figure 8.6.

Sensitivity to number of hash functions d

Figure 8.6(a) shows how INDDoS behaves with d ranging from 1 to 3. Recall is high in all three cases, but precision increases as d increases. This is because more hash functions lead to smaller number of collisions in the Count-min Sketch side of BACON Sketch, and hence to a lower overestimation of the number of source IPs contacting a specific destination.

Sensitivity to output size of hash functions w

Figure 8.6(b) shows how INDDoS performs by varying w from 128 to 1024. Clearly, precision is very sensitive to w, and it significantly decreases as w decreases. Conversely, even for small values of w (e.g., $w = 128$), recall remains above 0.75.

Sensitivity to Bitmap register size m

Figure 8.6(c) shows how INDDoS performance changes along with the value of m. Since the threshold in each time interval is θn, which is around $0.5\% \cdot 60000 = 300$ in our evaluation, when m is below 300, i.e., $m \in \{128, 256\}$, no victim can be detected and thus the F1 score is 0. Precision of $m = 512$ and of $m = 1024$ is comparable, but recall is much lower for $m = 512$. As proven theoretically, a smaller m causes a smaller lower bound on the flow cardinality estimation $\hat{E}_{dst}$, so it is more difficult for $\hat{E}_{dst}$ to exceed the threshold.

Table 8.4: INDDoS performance for Booter DDoS attacks

DDoS attack flow trace	Recall	Precision	F1 score
Booter 6	1.0 (1/1)	1.0 (1/1)	1.0 (1/1)
Booter 7	1.0 (1/1)	1.0 (1/1)	1.0 (1/1)
Booter 1	1.0 (1/1)	1.0 (1/1)	1.0 (1/1)
Booter 4	1.0 (1/1)	1.0 (1/1)	1.0 (1/1)
Mixed	1.0 (4/4)	1.0 (4/4)	1.0 (4/4)

Table 8.5: Network performance of INDDoS in the commodity switch

Type	iPerf size	Throughput	Jitter	Packet loss	Average additional processing time w.r.t. simple forwarding
TCP	64 KB	9.02 Gbps	/	/	106 ns
TCP	128 KB	9.41 Gbps	/	/	101 ns
UDP	500 B	1.03 Gbps	0.003 ms	0%	102 ns
UDP	1000 B	1.95 Gbps	0.003 ms	0%	102 ns
UDP	1470 B	2.87 Gbps	0.004 ms	0%	102 ns
UDP	3000 B	5.45 Gbps	0.004 ms	0%	107 ns
UDP	6000 B	9.62 Gbps	0.004 ms	0%	104 ns
UDP	9000 B	9.67 Gbps	0.006 ms	0%	101 ns

8.6.4 Exp. 2: evaluation of DDoS victim identification accuracy under Booter DDoS attacks

Table 8.4 shows the performance of INDDoS on victim identification under actual DDoS attacks with different number of attack source IPs. The threshold is approximately $60000 \cdot 1\% = 600$, which is larger than the largest flow cardinality for legitimate traffic in each 5 s interval. Using these parameters, INDDoS can always correctly identify the victim even though the number of attack source IPs in all cases is greater than the Bitmap size (1024), since our threshold is lower than that and a saturated count (i.e., all 1s in the Bitmap register) still results in flagging a destination as potential DDoS victim. Even though there are four distinct DDoS attacks in the Mixed case, INDDoS correctly identifies all four different victims. This suggests that, when a suitable threshold is used (insights on how to set it are given in Section 8.5), INDDoS can identify victims almost perfectly.

Furthermore, we observe that in our CAIDA flow trace the maximum flow cardinality in 5 s time intervals is around 500, whereas in 1 s intervals it is about 260. Therefore, shrinking the time interval not only allows INDDoS to detect attacks on faster networks, but it also increases the difference between legitimate and DDoS traffic, simplifying the selection of a threshold and increasing detection accuracy.

8.6.5 Evaluation of impact on network performance

We used iPerf3 [7] to indirectly measure the performance of INDDoS in the switch. The results are reported in Table 8.5. We first generated 10 Gbps of TCP traffic with iPerf buffer size 64 KB and 128 KB from one server to the other across the switch. In both cases, the throughput reached more than 9 Gbps, i.e., the TCP traffic could be processed at line-rate (with 10 Gbps interfaces). We then generated 10 Gbps of UDP traffic with different iPerf sizes ranging from 500 B to 9000 B. Using small (e.g., 1470 B) datagrams our server could not reach 10 Gbps (the server's CPU processing capacity is a bottleneck in this settings), therefore we increased the *maximum transmission unit* (MTU) to allow UDP datagram sizes larger than 1470 B. The results show that the throughput and jitter increased as packet size increased, but without packet loss. When the datagram size reached 6000 B, the throughput was 9.62 Gbps, indicating that INDDoS could also process UDP datagrams at line-rate.

Additionally, we embedded two registers in our P4 program to monitor processing time. One was placed at the ingress pipeline, and stored the timestamp t_{in} of when a packet entered the switch. The timestamp t_{out} recorded in another register at the egress pipeline when the packet had been processed. Per-packet processing time could thus be calculated as $t_{out} - t_{in}$. This averaged around 200 ns for simple forwarding, with INDDoS adding an additional 100 ns or so on top of that, which we deem an acceptable time overhead.

8.6.6 Evaluation of resource usage

Table 8.6 shows the switch resources required by our INDDoS and simple forwarding implementation. Considering that each stage can only apply a single atomic action, several stages must be used to carry out INDDoS calculations including simple forwarding, meaning that we require all available stages in the switch to achieve the best detection performance, whereas simple forwarding only requires a small fraction of them (16.67%). Moreover, our strategy uses only 8.33% of the total available SRAM. This is because each register cell in BACON Sketch only occupies 1 bit, and the size of Count-min Sketch is small (i.e., only 3×1024) though each of its register cells occupies 32 bits. INDDoS needs 56.25% of total ALUs for processing the packets. This is high since INDDoS not only needs to apply the match-action table for simple packet forwarding (4.2% of all ALUs), but also maintains two sketches (BACON and auxiliary), which require more arithmetic operations. The packet header vector (PHV) size indicates the amounts of packet header information passed across the pipeline stages. In our case, only 9.90% of PHV is required for storing this information, compared to 7.30% for simple forwarding, meaning that INDDoS does not embed much additional temporary metadata in the packet.

Table 8.6: Switch resource usage of INDDoS

Strategy	No. stages	SRAM	No. ALUs	PHV size
Simple forwarding	16.67%	2.5%	4.2%	7.30%
INDDoS + Simple forwarding	100%	8.33%	56.25%	9.90%

8.7 Related works

In this section we describe recent related works and solutions on *(i.)* DDoS detection in Software-Defined Networks, *(ii.)* programmable data plane capabilities with ASICs and *(iii.)* in-network monitoring using programmable switches.

8.7.1 DDoS detection in the context of SDN

Many techniques have already been proposed to detect various kinds of DDoS attacks in SDN networks. A DDoS attack can be identified according to many different metrics, such as looking for a significant decrease of the normalized entropy in distinct destination IP addresses observed in the network [65][80][122], or a large number of distinct flows (sequences of packets with the same source IPs) contacting a specific destination host (i.e., per-destination flow cardinality) [134][93][74]. Note that entropy-based DDoS detection can only detect DDoS attacks, but flow cardinality-based DDoS detection is also able to identify the DDoS victims, which allows operators to mitigate the impact on targeted nodes as soon as an attack is detected. However, the state-of-the-art approaches [134][93][74] in SDN still need the controller to periodically retrieve the information from the switches for further processing. With our flow-cardinality-based INDDoS approach we make a step further: we exploit data-plane programmable switches to just forward the information on DDoS victims to the controller, offloading the DDoS detection task to the switch.

8.7.2 Data plane programmable switches exploiting ASICs

Most of the existing SDN switches come with very limited (or no) programmability with respect to the data plane functions that can be executed. To enable new kinds of functionalities (e.g. support of additional protocols) it is necessary to upgrade the hardware, which requires significant capital expenditure.

Recently, programmable ASICs have been introduced: they ensure standard data plane features (i.e., high-speed switching and forwarding) while offering the possibility of customizing functionalities, if properly programmed through domain-specific programming languages like P4 [42]. For instance, programmable switches equipped with Tofino ASIC [2] can always forward packets at line-rate once the P4 program (including innovative features) is compiled and installed in the switches. Other programmable chips, like Network Interface Cards (NICs), Field Programmable Gate Arrays (FPGAs) and Network Processing Units (NPUs), cannot currently en-

sure high throughput and low latency on par with ASICs. Additionally, in the context of network security, compared to highly-optimized software solutions, such as inline Intrusion Detection Systems (IDSs) [81], the throughput ensured by ASICs is orders of magnitude higher and introduces much lower latency ($\sim 50\mu s - 1ms$) [136]. This makes programmable ASICs well suited for the implementation of some network monitoring/security tasks, such as the DDoS detection strategy proposed in this chapter.

8.7.3 In-network monitoring tasks using programmable switches

Network monitoring has been widely studied, including in the context dealing with the capability of programmable switches. Recently, researchers have started embedding network monitoring tasks directly into programmable switches, such as heavy hitter detection [113], network traffic entropy estimation [55] and entropy-based DDoS detection [85]. Most of these solutions are based on sketches, probabilistic data structures to track summarized information pertaining large numbers of packets using fixed size memory. It has been proven that sketch-based monitoring solutions have a better accuracy/memory trade-off than sampling-based solution, at least over short time scales [132]. A common feature of these approaches is that the monitoring outcomes gathered from sketches are reported to the controller only when an anomalous event is detected, therefore overcoming the limitations of large communication overhead and latency caused by the interaction between data plane and control plane. Unfortunately, another common theme among these works is that, unlike ours, their P4 code was only tested in the (largely resource-unconstrained) P4 Behavior model [13] simulator. Exceptions to this include Tang *et al.* [116], who proposed a new sketch for DDoS detection implementable in Tofino-based switches. Howerver, their solution relies on the controller querying the sketch for identifying the attacks, while in our proposal queries occur inside the switch, thus reducing switch-controller interaction needs. Dimolianis *et al.* [52] presented another in-network DDoS detection scheme working in actual Netronome SmartNICs [10]. Their approach measures three different features: total number of incoming traffic flows, subnet significance and packet symmetry. However, it is only able to identify the subnet under attack, which may limit the accuracy of deployed mitigation measures.

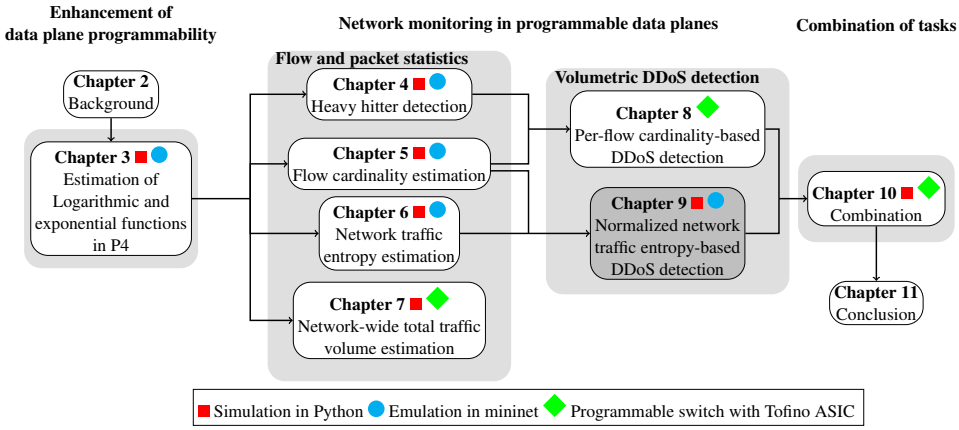
To the best of our knowledge, INDDoS is the first attempt to perform DDoS detection with host victim identification entirely in commodity switches equipped with programmable ASICs, while dealing with and overcoming all the constraints set by the hardware.

8.8 Concluding remarks

In this chapter, we proposed a novel in-network DDoS victim identification strategy, INDDoS, based on a new probabilistic data structure we named BACON

Sketch, which combines a Direct Bitmap and a Count-min Sketch to estimate the number of distinct flows contacting the same destination. INDDoS uses a threshold-based rule to identify victims directly in the programmable data plane of switches. We proved some parametric bounds on the quality of estimations produced by BACON and INDDoS, and implemented them using the P4 language and toolchain in a Edgecore commodity switch with Tofino ASIC. The analysis of the performance of our solution proves that it can precisely and accurately identify DDoS victims without adversely affecting the packet processing capabilities of the switch. Moreover, this approach only reports to the controller when a new victim is detected, greatly reducing the communication strain on the monitoring infrastructure. This work contributes to the ongoing DDoS attack detection and mitigation activities carried on in the GN4-3 project for upgrading the pan-European GÉANT network.

9 Normalized network traffic entropy-based DDoS detection



In Chapter 5 and 6 we studied two basic network monitoring tasks: (i) flow cardinality estimation (Chapter 5); and (ii) network traffic entropy estimation (Chapter 6). The combination of them can address a more practical security issue in the network: volumetric DDoS detection.

Unlike the previous chapter DDoS detection relying on the flow cardinality, in this chapter we track normalized network traffic entropy in programmable data planes and identify the DDoS attacks according to its variations

This chapter is based on the paper "Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4" submitted to IEEE Transactions on Dependable and Secure Computing (Under review).

9.1 Introduction

Distributed Denial-of-Service (DDoS) attacks are becoming one of the most significant threats for network operators and their customers as such attacks, carried out

by many different compromised hosts, are able to flood a victim with a huge load of superfluous traffic and exhaust its network and computational resources, causing service disruptions. In this context, detecting DDoS attacks in a smooth yet effective way plays a key role in today's network security. Periodical monitoring of specific network metrics has been widely adopted as a strategy to detect DDoS attacks. For instance, network traffic entropy is a statistical measure to describe the flow distribution, and the entropy of distinct destination IPs observed in the network significantly decreases during a DDoS attack [65][80][122]. Moreover, a significant increase in the number of source IPs contacting a specific destination IP [134][93][74] may also indicate that a DDoS attack is taking place.

Spotting variations of entropy over time, as done in previous works, may not be the most effective way to detect DDoS attacks. In fact, the number of distinct flows in the network (i.e., *flow cardinality*) changes dynamically, affecting, in turn, the value of traffic entropy. A more suitable metric is therefore the *normalized entropy*, which is normalized against flow cardinality and is more robust to legitimate changes on the number of distinct flows.

The goal of this chapter is thus to propose novel strategies to track normalized entropy with the final aim to accurately and timely detect DDoS attacks. To this aim, combing with in-network flow cardinality estimation *P4LogLog* presented in Chapter 5, we present *P4NEntropy*, a new strategy for normalized Shannon entropy [110] estimation in P4, and *P4DDoS*, our approach for DDoS detection based on P4NEntropy. The prototypes of P4DDoS have been implemented with the P4 behavioral model [13] and proven to be fully executable in a P4 emulated environment.

We then evaluate P4DDoS by means of simulations to show the effectiveness and sensitivity to different tuning parameters, with critical improvements (to the best of our knowledge) with respect to literature: P4DDoS ensures slightly better performance than the existing P4-enabled entropy-based DDoS detection solution [85]. In case of some stealthy DDoS attacks, such as an internal botnet DDoS attack or a DDoS attack with spoofed source IPs, our P4DDoS outperforms the state-of-the-art solution in terms of detection accuracy. Moreover, our P4DDoS does not need any interaction with the control plane in executing the foreseen operations, whereas [85] requires that a controller properly populates the TCAM of the switch with any pre-computed value. This is why we can claim that *our strategies work entirely in the data plane*.

9.2 Basic knowledge and used compact data structure

In this section we recall background concepts needed to understand the strategies proposed in the following sections.

9.2.1 Normalized network traffic entropy

Network traffic entropy [84] gives an indication on traffic distribution across the network. Each network switch can evaluate the traffic entropy related to the network flows that cross it in a given time interval T_{int}. Relying on the definition of *Shannon entropy* [110], network traffic entropy can be defined as $H = -\sum_{i=1}^{n} \frac{f_i}{|S|_{tot}} \log_d \frac{f_i}{|S|_{tot}}$, where f_i is the packet count of the incoming flow with *flow key* i (e.g. 5 tuple, source IP-destination IP pair, etc.), $|S|_{tot}$ is the total number of processed packets by the switch during T_{int}, n is the overall number of distinct flows and d is the base of logarithm. Traffic entropy is $H = 0$ when in T_{int} all packets $|S|_{tot}$ belong to the same flow i, while it assumes its maximum value $H = \log_d n$ when packets are uniformly distributed among the n flows. The *normalized entropy* is defined as $H_{norm} = \frac{H}{\log_d(n)}$ $(0 \leq H_{norm} \leq 1)$.

9.2.2 Sketch-based estimation of flow packet count

Estimating the number of packets for a specific flow crossing a programmable switch (f_i) is fundamental for network traffic entropy computation. Such an estimation can be performed by means of *sketches* [74], which are probabilistic data structures associated to a set of pairwise-independent hash functions. The *size* of each sketch data structure depends on the number of associated hash functions N_h and on the output size of each function N_s, and is $N_h \times N_s$. *Update* and *Query* operations are used to store and retrieve information from the sketch: Update operation is responsible for updating the sketch to keep track of flow packet counts, while Query operation retrieves the estimated number of packets for a specific flow. Two well-known algorithms to Update and Query sketches are *Count-min Sketch* [48] and *Count Sketch* [44]. A detailed theoretical analysis on the accuracy/memory occupation trade-off for these sketching algorithms is reported in [48][44]. From a high-level perspective, as any of N_h and N_s increase, memory consumption is larger but estimation is more accurate. Count Sketch leads to a better accuracy/memory consumption trade-off than Count-min Sketch, but its update time is twice slower [47].

9.3 Estimation of normalized traffic entropy

Based on Chapter 5, we use *P4LogLog* to estimate flow cardinality. P4LogLog is then used by *P4NEntropy*, which estimates the normalized network traffic entropy. The prototype of P4NEntropy has been implemented in P4 behavioral model [13] and are executable in an emulated environment as Mininet [8]. The P4 source codes are available in [22].

9.3.1 Normalized traffic entropy estimation: P4NEntropy

In this section we present a new strategy, named *P4NEntropy*, to estimate the normalized network traffic entropy using the P4 language in a given time interval. Formally, the problem is defined as follows.

Problem definition: *Given* a stream S of incoming packets, each one belonging to a specific *flow* i, and a time interval T_{int}, *returns* the *normalized Shannon entropy* estimation H_{norm} (see Section 9.2.1) at the end of T_{int}.

Derivation of estimated normalized entropy in P4

The goal of this section is to provide an estimation of network traffic normalized entropy by only using P4-supported operations and reducing as much as possible their number. The section also shows how relevant statistics, used for normalized entropy estimation at the end of T_{int}, are iteratively updated every time a packet crosses the switch.

We first rewrite the Shannon entropy as follows:

$$\begin{aligned} H(|S|_{tot}) &= -\sum_{i=1}^{n} \frac{f_i(|S|_{tot})}{|S|_{tot}} \log_d \frac{f_i(|S|_{tot})}{|S|_{tot}} \\ &= \log_d |S|_{tot} - \frac{1}{|S|_{tot}} \sum_{i=1}^{n} f_i(|S|_{tot}) \log_d f_i(|S|_{tot}) \end{aligned}$$

We consider $d = 2$ without any loss of generality. With respect to the definition given in Section 9.2.1, we use the notation $f_i(|S|_{tot})$ to make explicit that f_i refers to its value when $|S|_{tot}$ packets have been received (i.e., at the end of T_{int}). As packets arrive in the switch, the overall number of processed packets $|S|$ increases and must be stored in the switch to ensure that $H(|S|_{tot})$ can be computed at the end of T_{int}, when $|S| = |S|_{tot}$. We define $Sum(|S|) = \sum_{i=1}^{n} f_i(|S|) \log_d f_i(|S|)$, which must be updated as well. To understand how to update $Sum(|S|)$, let's assume that a new packet for a specific flow arrives and it is the $|S|$-th packet. We call its packet count $\bar{f}_i(|S|)$. It holds that:

$$\begin{cases} f_i(|S|) = f_i(|S|-1) & (f_i(|S|) \neq \bar{f}_i(|S|)) \\ f_i(|S|) = f_i(|S|-1)+1 & (f_i(|S|) = \bar{f}_i(|S|)) \end{cases}$$

This allows us to re-write $Sum(|S|)$ as follows:

$$\begin{aligned} Sum(|S|) = \; & Sum(|S|-1) + \bar{f}_i(|S|) \log_2 \bar{f}_i(|S|) + \\ & - (\bar{f}_i(|S|)-1) \log_2 (\bar{f}_i(|S|)-1) \end{aligned}$$

$Sum(|S|)$ thus needs two logarithmic computations for each incoming packet, and would require running P4Log twice with corresponding computational effort.

In the next step, we show how it is possible to estimate $Sum(|S|)$ with only (at most) one logarithmic computation. When $\bar{f}_i(|S|) = 1$, we estimate $Sum(|S|) =$

$Sum(|S|-1)$, being $\bar{f}_i(|S|)\log_2 \bar{f}_i(|S|) = 1\log_2 1 = 0$ and defining $(\bar{f}_i(|S|)-1)\log_2(\bar{f}_i(|S|)-1) = 0\log_2 0 = 0$ [93]. Instead, when $\bar{f}_i(|S|) > 1$, we need to re-write once again $Sum(|S|)$ in the following way:

$$Sum(|S|) = Sum(|S|-1) + \log_2 \bar{f}_i(|S|) + \\ + (\bar{f}_i(|S|)-1)\log_2(1+\frac{1}{\bar{f}_i(|S|)-1})$$

According to L'Hopital's rule [114]:

$$\lim_{\bar{f}_i(|S|)\to+\infty}(\bar{f}_i(|S|-1)\log_2(1+\frac{1}{(\bar{f}_i(|S|-1)}) = \frac{1}{\ln 2}$$

Thus, we set $1/\ln 2 \approx 1.44$ as the approximation of the third term of $Sum(|S|)$. This approximation best works when most of the flows in T_{int} carry a number of packets much greater than 1 (as usually happens in an ISP backbone network, which is the most suitable scenario where to apply our strategy). Finally, $Sum(|S|)$ can be estimated as:

$$Sum(|S|) \approx \begin{cases} Sum(|S|-1) & (\bar{f}_i(|S|) = 1) \\ Sum(|S|-1) + \log_2 \bar{f}_i(|S|) + 1/\ln 2 & \\ \qquad\qquad (\bar{f}_i(|S|) > 1) \end{cases} \tag{9.1}$$

This estimation requires at most one logarithm computation. Since P4 language does not support division, we re-write $\frac{1}{|S|_{tot}} = 2^{-\log_2 |S|_{tot}}$. So, entropy can be written as:

$$H(|S|_{tot}) = \log_2 |S|_{tot} - 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})}$$

In this form, entropy can be estimated by only using P4-supported operations, leveraging P4Log and P4Exp algorithms. In the following, we show how it is possible to further slightly reduce complexity in entropy estimation.

When $|S|_{tot} = \sum_{i=1}^{n} f_i(|S|_{tot}) > Sum(f_i|S_{tot}|)$, it holds that $0 < 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})} < 1$. This is a corner case that happens only when flow distribution is almost uniform (i.e., when most of flows carry only one or very few packets). In this case, we neglect the computation of $2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})}$, meaning that we estimate entropy as flow distribution was perfectly uniform. Network traffic entropy can then be estimated as follows:

$$H(|S|_{tot}) \approx \begin{cases} \log_2(|S|_{tot}) & (|S|_{tot} > Sum(|S|_{tot})) \\ \log_2(|S|_{tot}) - 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})} & \\ \qquad\qquad (|S|_{tot} \leq Sum(|S|_{tot})) \end{cases} \tag{9.2}$$

Finally, normalized entropy $H_{norm}(|S|_{tot})$ can be estimated as:

$$H_{norm}(|S|_{tot}) = 2^{\log_2(H(|S|_{tot})) - \log_2(\log_2 \hat{n})} \tag{9.3}$$

The number of estimated distinct flows $\hat{n}$ can be obtained using P4LogLog, that is, by updating a LogLog register for each incoming packet and by querying it at the end of T_{int}.

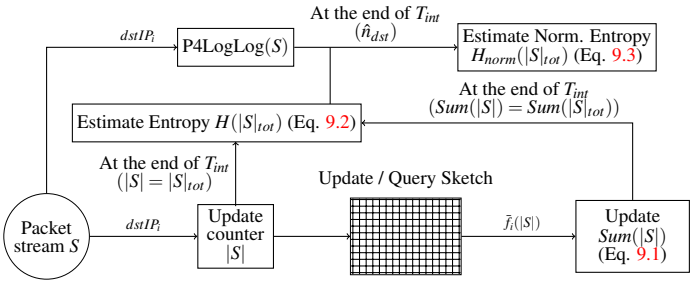


Figure 9.1: Scheme of P4NEntropy

Algorithm 10: P4NEntropy

Input: Packet stream S, time interval T_{int}
Output: Normalized entropy estimation $H_{norm}(|S|_{tot})$ of S in T_{int}

$|S| \leftarrow 0$
$Sum(|S|) \leftarrow 0$
Function *UpdateSum*$(Sum(|S|))$**:**
  while *currentTime* $< T_{int}$ **do**
    for *Each received packet belonging to flow i* **do**
      $|S| \leftarrow |S| + 1$
      $\bar{f}_i(|S|) \leftarrow Sketch(dstIP_i)$
      if $\bar{f}_i(|S|) > 1$ **then**
        $Sum(|S|) \ll 10 \leftarrow Sum(|S|) \ll 10$
        $+P4Log(\bar{f}_i(|S|)) + 1.44 \ll 10$
  $Sum(|S|_{tot}) \leftarrow (Sum(|S_{tot}|) \ll 10) \gg 10$
  return $Sum(|S|_{tot})$, $|S|_{tot}$
Function *EstimateNormEntropy*$(Sum(|S|_{tot}),\ |S|_{tot})$**:**
  if *currentTime* $= T_{int}$ **then**
    if $|S|_{tot} > Sum(|S|_{tot})$ **then**
      $H(|S|_{tot}) \ll 10 \leftarrow P4Log(|S|_{tot})$
    else
      $\text{diff} \leftarrow P4Log(Sum(|S|_{tot})) - P4Log(|S|_{tot})$
      $H(|S|_{tot}) \ll 10 \leftarrow P4Log(|S|_{tot}) - P4Exp(2, \text{diff})$
    $\hat{n}_{dst} \leftarrow P4LogLog(S, T_{int})$
    $\text{diff}_n \leftarrow P4Log(H(|S|_{tot}) \ll 10) +$
    $-(P4Log(P4Log(\hat{n}_{dst})) - 10 \ll 10)$
    if $diff_n > 0$ **then**
      $H_{norm}(|S|_{tot}) \ll 10 \leftarrow P4Exp(2, \text{diff}_n)$
    else
      $H_{norm}(|S|_{tot}) \ll 10 \leftarrow 0$
  return $H_{norm}(|S|_{tot}) \ll 10$

Description of P4NEntropy strategy

Figure 9.1 and Algorithm 10 show the scheme and pseudocode of P4NEntropy algorithm, leveraging outcomes from Sections 5.3 and 9.3.1. First, the algorithm continuously updates $Sum(|S|)$ until the end of T_{int} (*UpdateSum* function) with flow information from incoming packets. A counter $|S|$ is used to count all incoming packets in the switch. Note that we consider as *flow key* the destination IP of the packet, with $i \sim dstIP_i$. A sketch data structure (e.g., Count Sketch or Count-min Sketch, see Section 9.2.2) is used to store the estimated packet count for all the flows, being continuously updated to include information from new packets, and then it is queried to retrieve the estimated packet count $\bar{f}_i(|S|)$ for the flow i the current incoming packet belongs to. This value is then passed to a register named $Sum(|S|)$, which is updated as specified in Eq. 9.1. All the floating numbers in the equation must be amplified 2^{10} times, since P4Log outputs an amplified integer value. Only at the end of T_{int}, $Sum(|S|_{tot})$ is reduced by a factor of 2^{10} and its final value, together with $|S|_{tot}$, is returned (Lines 1-12 of the pseudocode). Traffic entropy is then estimated as specified in Eq. 9.2 (Lines 15-19). The resulted value of $H(|S|_{tot})$ is amplified 2^{10} times since output values of P4Log are amplified, while output values of P4Exp are not. Such an amplification makes it possible to use P4Exp in Eq. 9.3 to estimate $H_{norm}(|S|_{tot})$ amplified 2^{10} times. Note that $H(|S|_{tot}) \ll 10$ may be smaller than $\log_2(\hat{n}_{dst})$ but, in this case, the normalized network traffic entropy can be approximated to 0 (Lines 25-26). Since the result of P4Log is left-shifted 10 bits, the computation of $\log_2(\log_2(\hat{n}_{dst}))$ must be carefully handled. Considering that the result of P4Log$(\hat{n}_{dst})$ is $\log_2(\hat{n}_{dst}) \ll 10$, the output of P4Log$(\log_2(\hat{n}_{dst}) \ll 10)$ can be expressed as $\log_2(\log_2(\hat{n}_{dst}) \ll 10) \ll 10 = \log_2(\log_2(\hat{n}_{dst}) \cdot 2^{10}) \ll 10 = \log_2(\log_2(\hat{n}_{dst})) \ll 10 + 10 \ll 10$. Hence, $\log_2(\log_2(\hat{n}_{dst})) \ll 10$ is equivalent to P4Log(P4Log$(\hat{n}_{dst})) - 10 \ll 10$ (Line 22). The resulting value is used to compute the normalized network traffic entropy amplified 2^{10} times (Line 24).

9.4 Normalized entropy-based DDoS detection

Based on P4NEntropy, we present a simple yet effective entropy-based DDoS detection strategy in P4, named *P4DDoS*. The P4 code of P4DDoS is available in [20]. Formally, the problem is defined as follows.

Problem definition: *Given* a k-th time interval T^k_{int}, a stream S_k of incoming packets during T^k_{int}, the estimated normalized network traffic entropy of destination IPs H^k_{norm} at the end of T^k_{int} and an adaptive threshold λ^k_{norm}, *returns* an *alarm* to the controller, at the end of T^k_{int}, if a *potential DDoS attack* is identified.

Our proposed strategy triggers an alarm if $H^k_{norm} < \lambda^k_{norm}$. In fact, as empirically evaluated in previous works (e.g. [100][34]), when a DDoS attack occurs, the normalized network traffic entropy of destination IPs significantly decreases, since traffic is concentrated around few destination nodes. The most critical aspect for such an entropy-based strategy is how to set the threshold λ^k_{norm}. This will be

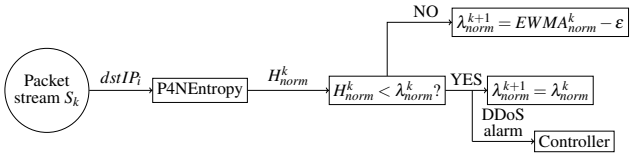


Figure 9.2: Scheme of P4DDoS

discussed in the next subsection.

9.4.1 Adaptive threshold setting

Since network traffic fluctuates over time, we define an *adaptive threshold* to protect our strategy from false positives that may be generated if using a fixed-value threshold in such a dynamic environment. Our proposed adaptive threshold leverages the computation of an *Exponentially Weighted Moving Average* (EWMA) of H^k_{norm} across different time intervals. The moving average $EWMA^k_{norm}$ in time interval T^k_{int} is expressed as:

$$EWMA^k_{norm} = \begin{cases} H^k_{norm} & (k=1) \\ \alpha H^k_{norm} + (1-\alpha) EWMA^{k-1}_{norm} & (k>1) \end{cases}$$

where $\alpha \quad (0 < \alpha < 1)$ is the smoothing factor for $EWMA^k_{norm}$. We define a threshold parameter ε $(0 \leq \varepsilon \leq 1)$, used to compute the threshold λ^{k+1}_{norm} in the next time interval T^{k+1}_{int} if no alarm is generated in T^k_{int}:

$$\lambda^{k+1}_{norm} = \begin{cases} EWMA^k_{norm} - \varepsilon & (\text{no alarm in } T^k_{int}) \\ \lambda^k_{norm} & (\text{alarm in } T^k_{int}) \end{cases}$$

As shown above, the threshold λ^{k+1}_{norm} is not updated if an alarm is generated in the time interval: this ensures that the threshold is updated when only legitimate traffic crosses the switch and its value is not biased by DDoS traffic. Note that setting the parameter ε in a proper way is also fundamental to get good DDoS detection performance. This aspect will be evaluated in Section 9.5.

9.4.2 Implementation in P4 language

Figure 9.2 and Algorithm 11 report the scheme and pseudocode of the P4DDoS strategy, with focus on a given time interval T^k_{int}. At the end of time interval T^k_{int}, the *DDoSDetection* function is executed. $Alarm^k_{ddos}$ is set to 0 and the normalized network traffic entropy H^k_{norm} is estimated by P4NEntropy, amplified 2^{10} times (Lines 2-3). It is then compared to the threshold $\lambda^k_{norm} \ll 10$ (Line 4). If smaller, the alarm $Alarm^k_{ddos}$ is set to 1 and the *UpdateThreshold* function is called (Lines 5-7). Otherwise, the *UpdateThreshold* function is called without changing $Alarm^k_{ddos}$ (Lines 9-10). If $Alarm^k_{ddos} = 1$, the switch clones the current packet and embeds

Algorithm 11: P4DDoS

Input: Packet stream S_k, time interval T_{int}^k, threshold parameter ε, smoothing factor α, threshold $\lambda_{norm}^k \ll 10$ and average $EWMA_{norm}^{k-1} \ll 10$ computed in T_{int}^{k-1}

Output: DDoS alarm $Alarm_{ddos}^k = 1$ if a DDoS attack is detected in T_{int}^k

1 **Function** $DDoSDetection(\lambda_{norm}^k \ll 10)$**:**
2 $\quad H_{norm}^k \ll 10 \leftarrow$ P4NEntropy(S_k, T_{int}^k)
3 $\quad Alarm_{ddos}^k \leftarrow 0$
4 $\quad$ **if** $H_{norm}^k \ll 10 < \lambda_{norm}^k \ll 10$ **then**
5 $\quad\quad Alarm_{ddos}^k \leftarrow 1$
6 $\quad\quad UpdateThreshold(k, \alpha, H_{norm}^k \ll 10, \varepsilon, Alarm_{ddos}^k,$
7 $\quad\quad EWMA_{norm}^{k-1} \ll 10, \lambda_{norm}^k \ll 10)$
8 $\quad$ **else**
9 $\quad\quad UpdateThreshold(k, \alpha, H_{norm}^k \ll 10, \varepsilon, Alarm_{ddos}^k,$
10 $\quad\quad EWMA_{norm}^{k-1} \ll 10, \lambda_{norm}^k \ll 10)$
11 $\quad$ **return** $Alarm_{ddos}^k$
12 **Function** $UpdateThreshold(k, \alpha, H_{norm}^k \ll 10, \varepsilon,$
13 $Alarm_{ddos}^k, EWMA_{norm}^{k-1} \ll 10, \lambda_{norm}^k \ll 10)$**:**
14 $\quad$ **if** $Alarm_{ddos}^k = 0$ **then**
15 $\quad\quad$ **if** $k = 1$ **then**
16 $\quad\quad\quad EWMA_{norm}^k \ll 10 \leftarrow H_{norm}^k \ll 10$
17 $\quad\quad$ **else**
18 $\quad\quad\quad EWMA_{norm}^k \ll 10 \leftarrow ((\alpha \ll 10) \cdot H_{norm}^k \ll 10$
19 $\quad\quad\quad +((1-\alpha) \ll 10) \cdot EWMA_{norm}^{k-1} \ll 10) \gg 10$
20 $\quad\quad \lambda_{norm}^{k+1} \ll 10 \leftarrow EWMA_{norm}^k \ll 10 - \varepsilon \ll 10$
21 $\quad$ **else**
22 $\quad\quad \lambda_{norm}^{k+1} \ll 10 \leftarrow \lambda_{norm}^k \ll 10$
23 $\quad$ **return** $\lambda_{norm}^{k+1} \ll 10, EWMA_{norm}^k \ll 10$

the value 1 in a customized header field. The cloned packet is then sent to the controller to report that a potential DDoS attack has been detected.

The *UpdateThreshold* function updates EMWA and the adaptive threshold as specified in Section 9.4.1. (Lines 12-23). Note that, since both EMWA and the threshold λ_{norm} are usually decimal numbers, all the operations are executed to ensure that their value is amplified 2^{10} times.

9.4.3 Insights on network-wide coordination

So far, we have focused on entropy-based DDoS detection in a single programmable switch. The switch can generate alarms if, according to the traffic flowing through its interfaces, a DDoS attack may be occurring. However, given the reduced network visibility of a single switch, a final decision on whether a DDoS attack is

actually carried out should be taken by the centralized controller from a *network-wide* perspective, that is, by cross-checking collected information from multiple switches and taking a global decision. For instance, UnivMon [93] and Elastic Sketch [132] present a way to estimate *network-wide traffic entropy*: the idea behind those works is to sample a set of flows with large packet count in any programmable switch, and send such statistics to the controller at the end of any time interval. The controller estimates the entropy of reported sampled "heavy" flows and considers it as a *network-wide entropy estimation*. As reported in [36], these two approaches assume that packets for a specific flow are counted only once in the network. By making the same strong assumption, in our case network-wide traffic entropy H^{nw} can be expressed as:

$$H^{nw} = \log_2(\sum_{j=1}^{w} |S_j|_{tot}) - \frac{1}{\sum_{j=1}^{w} |S_j|_{tot}} (\sum_{j=1}^{w} Sum_j)$$

where w is the number of switches in the network and $Sum_j = \sum_{i=1}^{n_j} f_i(|S_j|_{tot}) \log_d f_i(|S_j|_{tot})$ (see Section 9.3.1). Additionally, according to the union property of LogLog (see section 5.2.1), the normalized network-wide traffic entropy H^{nw}_{norm} can be expressed as:

$$H^{nw}_{norm} = \frac{H^{nw}}{\log_2(LogLog(S_1 \cup S_2 \cup \cdots S_w))}$$

In this case, the strong assumption considered above can be neglected, since the union property of LogLog makes it possible to estimate the network-wide number of distinct flows also if a packet is counted in different locations.

Given the above considerations, a network-wide strategy could be designed to forward to the controller all the needed information from the switches (i.e., $|S_j|_{tot}$, Sum_j and the j-th LogLog register) for the computation of network-wide normalized entropy in support to a centralized *network-wide DDoS detection*. However, since in real scenarios the packet may traverse multiple switches and generate double packet counts, the accuracy of the computed network-wide entropy N^{nw} would be compromised. How to overcome this issue is still open: we will work on refined strategies for network-wide entropy-based DDoS detection in the future.

9.5 P4DDoS evaluation

We implemented P4DDoS in Python and simulated it for evaluation. Additionally, we also implemented a state-of-the-art entropy-based DDoS detection approach [85] executable in programmable switches, named *SOTA_DDoS* for the sake of brevity, and compared them. To make a fair comparison, both DDoS detection strategies have been implemented leveraging our proposed P4NEntropy strategy and using a sketch, for packet count estimation, of the same size. Note, however, that the original version of SOTA_DDoS uses SOTA_Entropy for entropy estimation (see the previous subsection). Unlike P4DDoS, which triggers a DDoS alarm

only when the *normalized* entropy of destination IPs decreses below a threshold, SOTA_DDoS triggers a DDoS alarm when any of two conditions holds: *(i)* entropy (not normalized) of source IPs increases above an adaptive threshold and *(ii)* entropy (not normalized) of destination IPs decreases below an adaptive threshold.

9.5.1 Evaluation metrics and simulation settings

Testing flow trace and methodology

We consider three kinds of flow traces.
***Trace1*. Legitimate flow trace:** The same CAIDA flow trace [3] that we used for the evaluation of P4LogLog. The 50-seconds flow trace is divided into 50 1-second time intervals.
***Trace2*. Legitimate flow trace mixed with Booter DDoS attack traffic [108]:** 50-seconds traces are taken from a set of Booter DDoS attack traces, and split into 50 time intervals. Each 1-second attack trace is injected into the legitimate 50-seconds flow trace according to its sequential 1-second time intervals. We took four different packet-rate Booter DDoS attack traces into consideration: Tab. 9.1 reports their properties. This allows us to analyze DDoS attacks with different packet rate and number of attack source IPs. Moreover, we also injected all four Booter DDoS attack traces together into the legitimate flow trace: we name this trace as *Mixed*. Such a mixed DDoS attack flow trace can help us evaluate the performance of DDoS detection when multiple DDoS attacks occur simultaneously in the network.
***Trace3*. Legitimate flow trace mixed with internal Botnet DDoS attack traffic:** In this case, we assume that some internal hosts of the network (e.g., a datacenter network) are exploited by an attacker to reverse malicious traffic towards a DDoS victim within the same network. We varied the *attack traffic proportion* (i.e., the percentage of generated malicious traffic over the total traffic in the network) from 5% to 30%. This flow trace is generated by crafting Trace 1 in such a way that part of the traffic is forwarded to one specific DDoS victim (by changing the destination IP of a given proportion of the packets).

Evaluation metrics

We consider *true-positive rate* D_{tp}, *false-positive rate* D_{fp} and *detection accuracy* D_{acc} as evaluation metrics. Considering that *(i)* True Positive (TP) is the number of time intervals with a triggered DDoS alarm while a DDoS attack is occurring in those intervals, *(ii)* True Negative (TN) is the number of time intervals without any triggered DDoS alarm while no DDoS attack is occurring, *(iii)* False Positive (FP) is the number of time intervals with a triggered DDoS alarm while no DDoS attack is occurring, and *(iv)* False Negative (FN) is the number of time intervals without any triggered DDoS alarm while a DDoS attack is instead occurring, the metrics

Table 9.1: Properties of DDoS flow traces [108]

DDoS trace name	Packet per second	Attack source IPs
Booter 6	$\sim$ 90000	7379
Booter 7	$\sim$ 41000	6075
Booter 1	$\sim$ 96000	4486
Booter 4	$\sim$ 80000	2970

introduced above are defined as:

$$D_{tp} = \frac{TP}{TP+FN} \times 100\%$$
$$D_{fp} = \frac{FP}{TN+FP} \times 100\%$$
$$D_{acc} = \frac{TP+TN}{TP+TN+FP+FN} \times 100\%$$

Tuning parameters

The smoothing factor in $EWMA_{norm}$ and for the thresholds defined in SOTA_DDoS is set to $\alpha = 0.13$: with this value, all the previous computed averages (up to all the 50 time intervals) have some impact on EWMA. All the parameters for P4Log and P4Exp are the ones reported in Tab. 3.3 of Chapter 3. We choose Count Sketch as sketch for P4NEntropy, with $N_h = 5 \times N_s = 2000$. The register size in P4LogLog is set to $m = 2048$, which corresponds to 1280 Bytes of memory. The considered time intervals T_{int}, as already said, are 1-second wide. Finally, the normalized entropy parameter is set to $\varepsilon = 0.01$ unless otherwise specified.

9.5.2 Detection performance (Booter DDoS attacks)

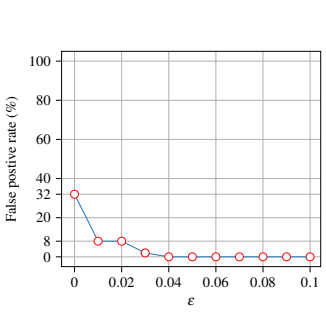
In this subsection, we evaluate our P4DDoS strategy against the state-of-the-art approach SOTA_DDoS in terms of D_{tp}, D_{fp} and D_{acc} in the case of Booter DDoS attacks. We also perform a sensitivity analysis of P4DDoS against the parameter ε, showing how the detection performance is affected by changing its value. The testing flow trace is composed by the concatenation of *Trace1* and *Trace2*: we first run 50-seconds legitimate flow trace (*Trace 1*) so that adaptive thresholds on entropy, for both strategies, are properly set in a legitimate traffic scenario: this trace allows us to evaluate D_{fp}. Then, *Trace 2* including different packet-rate DDoS attacks (also mixed), is used to evaluate D_{tp} and, together with results obtained in *Trace 1*, D_{acc}.

Comparison with the state of the art

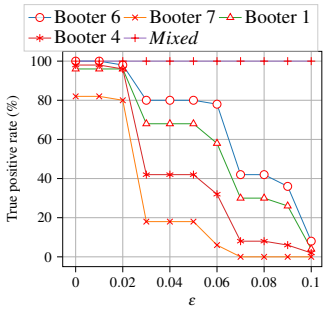
To fairly compare P4DDoS with SOTA_DDoS, we tuned the sensitivity coefficient k of SOTA_DDoS (see [85]) to different values: lower k leads to higher true-positive rate but also higher false-positive rate. Evaluation results are reported in Tab. 9.2. In the first 50 time intervals, four false alarms are detected

Table 9.2: Comparison of P4DDoS detection performance with a state-of-the-art approach [85] (Booter DDoS attacks)

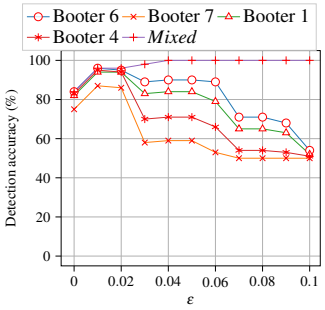
Algorithm	**False-positive rate D_{fp}**	**True-positive rate D_{tp} / Detection accuracy D_{acc}**				
		Booter 6	**Booter 7**	**Booter 1**	**Booter 4**	**Mixed**
P4DDoS	8%	100% / 96%	82% / 87%	96% / 94%	98% / 95%	100% / 96%
SOTA_DDoS (k=5.5)	6%	96% / 95%	32% / 63%	62% / 78%	70% / 82%	100% / 97%
SOTA_DDoS (k=4.5)	8%	100% / 96%	38% / 65%	82% / 87%	78% / 85%	100% / 96%
SOTA_DDoS (k=3.5)	10%	100% / 95%	74% / 82%	100% / 95%	94% / 92%	100% / 95%
SOTA_DDoS (k=2.5)	20%	100% / 90%	94% / 87%	100% / 90%	100% / 90%	100% / 90%
SOTA_DDoS (k=1.5)	38%	100% / 81%	100% / 81%	100% / 81%	100% / 81%	100% / 81%
SOTA_DDoS (k=0.5)	60%	100% / 70%	100% / 70%	100% / 70%	100% / 70%	100% / 70%



(a) Sensitivity to false-positive rate



(b) Sensitivity to true-positive rate



(c) Sensitivity to detection accuracy

Figure 9.3: Sensitivity analysis of P4DDoS to parameter ε

by P4DDoS, being thus the false-positive rate 8%. As said, the false-positive rate of SOTA_DDoS increases as k decreases. False-positive rate of P4DDoS is slightly higher than of SOTA_DDoS only when $k = 5.5$ but, in that case, P4DDoS outperforms SOTA_DDoS on both true-positive rate and detection accuracy for all the considered Booter attacks. The best trade-off between all the metrics for SOTA_DDoS is obtained with $k = 3.5$. In this case, P4DDoS and SOTA_DDoS have comparable performance (with slightly better performance for P4DDoS). This means that, in this scenario, comparing the normalized entropy of destination IPs

Table 9.3: Comparison of P4DDoS detection performance with a state-of-the-art approach [85] for different Botnet DDoS attack traffic proportions (ATPs)

Algorithm	False positive rate D_{fp}	True-positive rate D_{tp} / Detection accuracy D_{acc}					
		ATP: 5%	ATP: 10%	ATP: 15%	ATP: 20%	ATP: 25%	ATP: 30%
P4DDoS	8%	36% / 64%	92% / 92%	100% / 96%	100% / 96%	100% / 96%	100% / 96%
SOTA_DDoS (k=5.5)	6%	0% / 47%	12% / 53%	68% / 81%	100% / 97%	100% / 97%	100% / 97%
SOTA_DDoS (k=4.5)	8%	0% / 46%	40% / 66%	96% / 94%	100% / 96%	100% / 96%	100% / 96%
SOTA_DDoS (k=3.5)	10%	10% / 50%	50% / 70%	100% / 95%	100% / 95%	100% / 95%	100% / 95%
SOTA_DDoS (k=2.5)	20%	20% / 50%	88% / 84%	100% / 90%	100% / 90%	100% / 90%	100% / 90%
SOTA_DDoS (k=1.5)	38%	82% / 72%	94% / 78%	100% / 81%	100% / 81%	100% / 81%	100% / 81%
SOTA_DDoS (k=0.5)	60%	96% / 68%	100% / 70%	100% / 70%	100% / 70%	100% / 70%	100% / 70%

against a well-defined threshold is enough to get good performance on DDoS detection and that an evaluation of entropy of source IPs can be avoided (that is, same performance can be obtained with a simpler strategy).

Sensitivity analysis

Figure 9.3 reports the sensitivity of P4DDoS to normalized network traffic entropy parameter ε. Figure 9.3(a) shows that false-positive rate decreases as ε is smaller and stabilizes to zero once ε is larger than 0.04. This is because larger ε results in a smaller threshold, being more DDoS alarms triggered also when DDoS attacks are not occurring. Figure 9.3(b) reveals the behavior of true-positive rate with respect to an ε variation, showing that in general true-positive rate decreases as ε increases. Figure 9.3(c) shows the impact of ε on detection accuracy. The shown curves, apart from the Mixed case, have a maximum at around $\varepsilon = 0.01$: we then decided to set ε to this value, since it leads to the best trade-off considering all the three metrics.

9.5.3 Detection performance (Botnet DDoS attacks)

Table 9.3 shows a comparison on DDoS detection performance in case of internal Botnet DDoS attacks. The same methodology as described in Section 9.5.2 is adopted to prepare the testing flow trace but, in this case, *Trace1* and *Trace3* are concatenated. In this attack scenario, the cardinality of source IPs in the network does not change and the attack traffic proportion in Trace 3 is varied from 5% to 30%. Intuitively, the detection accuracy of P4DDoS increases as the attack traffic proportion increases. When the attack traffic rate is low, i.e., 5%, the true-positive rate of P4DDoS is 36%. This is the drawback of most normalized entropy-based DDoS detection strategies: these strategies struggle to detect low-packet-rate DDoS attacks since the normalized entropy may not significantly decrease. Nevertheless, our P4DDoS still has higher (or at least comparable) detection accuracy than SOTA_DDoS for any coefficient k. This is due to the fact that the entropy of destination IPs (not normalized) may decrease because of either a decrease in the cardinality of destination IPs in consecutive time intervals (see Section 9.2.1) or because a DDoS attack is occurring. Instead, the normalized entropy (used by

P4DDoS) decreases only when a DDoS attack is occurring, since it is normalized to the cardinality of destination IPs. Thus, considering non-normalized entropy as the metric to detect DDoS attacks as done by SOTA_DDoS, there is a higher chance of false positives due to legitimate traffic oscillations in consecutive time intervals. It is also important to note that the entropy of source IPs may not significantly increase when a Botnet DDoS attack occurs (as proven in [41]), so a simpler entropy-based DDoS detection system considering only normalized entropy of destination IPs may suffice for the detection of a wide range of attacks.

9.6 Related work

9.6.1 Entropy-based DDoS detection

Entropy-based DDoS detection has been widely studied in the context of SDN: a significant decrease in the (normalized) network entropy of destination IPs in a given time interval can be an indication of occurrence of a DDoS attack [98][65][80][122]. However, in most of previous works, entropy estimation is executed by the controller due to the complex way it is computed.

Some works can be found in literature dealing with network traffic entropy estimation performed partially in the switches' data plane. For example, papers [74][93][132] all envision some operations to be executed by the programmable data plane, so that only summarized data must be sent to the controller. However, since the controller needs to frequently retrieve information from all the switches, the generated communication overhead is significant. Recently, Lapolli et al. [85] have demonstrated the feasibility of performing network traffic entropy estimation in the data plane using the P4 language, with the aim of detecting DDoS attacks. Their approach is valuable but it requires the usage of TCAM, which is instead avoided by our proposed P4DDoS. Moreover, P4DDoS and P4NEntropy adopt a time-based observation window, while [85] requires an observation window that includes a fixed power-of-two number of packets, making their solution less flexible. In fact, our approach may allow a controller to synchronize the retrieval of the estimated entropy from many programmable switches, paving the way towards the estimation of network traffic entropy on a *network-wide* scale [53] to improve the statistical relevance of monitored values.

9.7 Concluding remarks

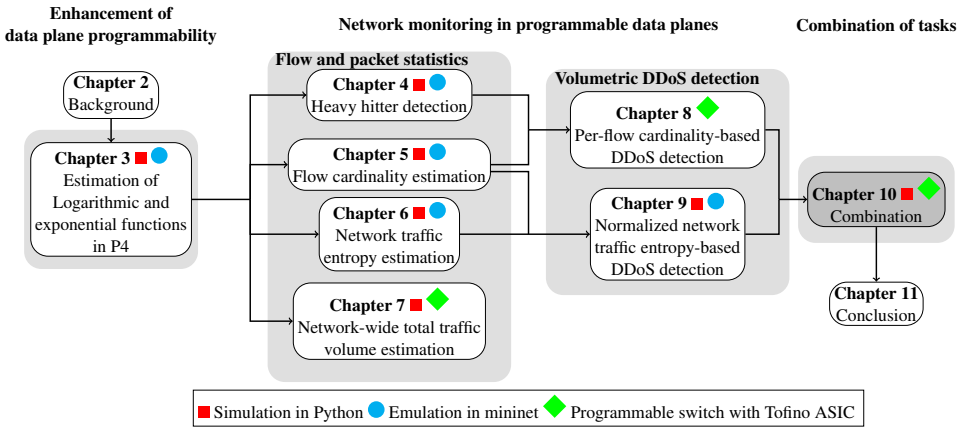
In this chapter, P4DDoS has been designed on top of P4NEntropy, with the goal of detecting DDoS attacks by means of an entropy-based system. We also evaluated all of our proposed approach and compared them with a state-of-the-art solution. P4DDoS outperforms existing DDoS detection solutions implemented in P4 in terms of detection accuracy, especially in the case of internal Botnet DDoS attacks, while implementing a simpler logic. Moreover, unlike existing approaches in liter-

ature, all of our strategies avoid any communication overhead between controller and programmable switches, since they work entirely in the data plane. Specifically, P4DDoS only reports an alarm to the controller when an attack is detected.

As future work, we plan to find a proper solution to detect low-packet-rate DDoS attacks (i.e., with attack traffic proportion $\leq 5\%$) with high accuracy. Furthermore, we also intend to work on an algorithm for the entropy-based detection of DDoS attacks on a network-wide scale, by collecting and combining the distributed entropy information from multiple programmable switches.

10

Combination of monitoring tasks



The previous chapters present single monitoring tasks relying on sketches in programmable data planes. However, the sketches are designed for specific measurement tasks, which poses a new challenge: is it possible to combine several tasks working together in programmable hardware switches? In this chapter, we combine and reuse two common sketches exploited in previous chapters, HyperLogLog and Count Sketch, to track overall traffic distribution statistics, including variance and entropy of flow packet count. The statistics can then be further used to diagnose performance and security problems, such as heavy-hitter detection and volumetric DDoS detection. We show that combination of monitoring tasks in programmable data planes is feasible while the packets can be processed at line rate.

This chapter has not been published before and is entirely new. The goal of this chapter is to enhance the connections between previous chapters and to show that the previously proposed monitoring solutions are practical in real network scenarios.

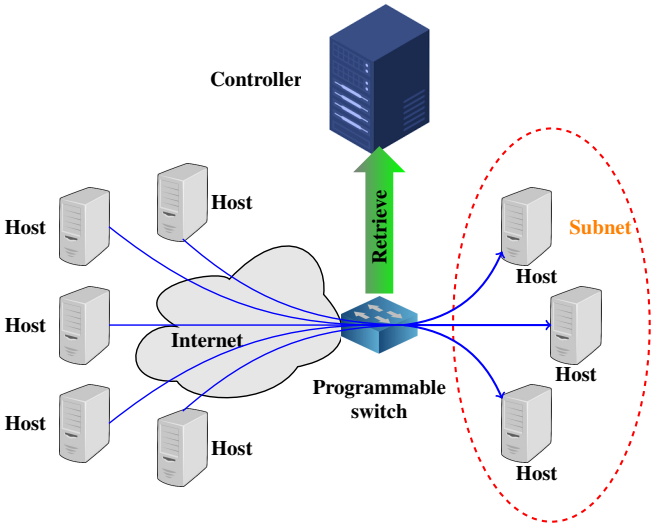


Figure 10.1: Deployment scenario

10.1 Introduction

Network operators often need to collect a variety of different network statistics and measurements, including per-flow sizes, heavy hitters [54], and aggregated information about flow distributions (called moments such as entropy [55] and variance [127]), which may be used to detect anomalies and understand the flow distribution in overall traffic patterns. However, in the previous chapters, we only focus on the specific individual tasks in programmable data planes. Then a question comes: is it possible to deploy several monitoring tasks in programmable hardware switches while considering their strict resource constraints?

In this chapter, we propose SEAMEN (SkEtch-bAsed flow Moments EstimatioN), a sketch-based solution to estimate the flow moments as the building blocks to support a wide-range of network monitoring tasks, including heavy-hitter detection and volumetric DDoS detection. The flow moments, such as flow cardinality, packet count, and the sum of the square of packet count, indicate the overall condition of the network traffic. Thus, the flow moment can be measured in time intervals to monitor the status of a group of hosts and servers. A possible deployment scenario is shown in Figure 10.1: the programmable switch is deployed at the border of a subnet. By tracking the flow moments, the controller can understand the overall condition of the traffic in the subnet relying on the flow statistics in the programmable switch, and so the anomalies or performance issues can be easily detected.

We implemented the SEAMEN data structures in a P4-programmable switch equipped with Tofino ASIC chip and show that the method can process packets at line-rate and is able to estimate the flow moments with high accuracy.

In summary, we make the following contributions in this chapter:

- We propose a sketch-based method, SEAMEN, to estimate the flow moments traversing the programmable switch.
- We analyze the possible usages of flow moments to execute a wide range of monitoring tasks
- We developed a prototype of SEAMEN, installed it in a programmable commodity switch with Tofino Application Specific Integrated Circuit (ASIC) [2], and conduct evaluations on a physical testbed.

10.2 Basic knowledge and used compact data structure

In this section, we present the compact data structures that we used to implement monitoring tasks in programmable data plane. Furthermore, we also present some background information that will used in our approach.

10.2.1 HyperLogLog

HyperLogLog [63] is a sketch-based algorithm that can be adopted to estimate large number of distinct flows (also named *flow cardinality*) crossing a network monitoring point. It envisions two types of operations: *Update* and *Query*. The Update operation updates the (Hyper)LogLog sketch with flow information from the incoming packet, whereas the Query operation is adopted to retrieve from the sketch the estimated flow cardinality.

The Update operation works as follows: given an incoming packet with *flow key id* (e.g., any subset of 5 tuple) and an m-sized ($m \in \{2^4, 2^5, \cdots, 2^{16}\}$) HyperLogLog register M with l bits each cell [1], HyperLogLog applies to id a uniform distributed hash function h with output size os ($os \geq 2^l + \log_2 m$): the resulted os-bit binary string $h(id)$ is denoted by $h(id) = [os-1:0]$. HyperLogLog then updates an m-sized register M. Let j be the rightmost $\log_2 m$ bits of $h(id)$ and x the 2^l bits of remaining, i.e., $j = h(id)[\log_2 m - 1 : 0]$ and $h(id) = H[\log_2 m + 2^l - 1 : \log_2 m]$. M is updated following this rule: $M_j = max(M_j, v(x))$, where $v(x)$ is the index of the rightmost 1 of x plus one. For instance, x is 0100, then $v(x)$ is 3.

HyperLogLog estimates flow cardinality $\hat{n}$ with Harmonic mean of power of 2, that is, $\hat{n} = \alpha_m^{HLL} m^2 (\sum_0^{m-1} 2^{-M_j})^{-1}$, in which α_m^{HLL} is a bias correction parameter of HyperLogLog. The theoretical standard error of HyperLogLog is $1.04/\sqrt{m}$, where m is the size of register M.

10.2.2 Count Sketch

Count Sketch is a compact data structure associated to a set of pairwise-independent hash functions, which is usually used to estimate the packet counts of flows in the network. The *size* of this data structure depends on the number of associated hash

[1] l-bits register cells are able to estimate cardinality up to 2^{2^l}

functions N_h and on the output size of each function N_s, and is composed by $N_h \times N_s$ counters. *Update* and *Query* operations are used to store and retrieve information from the sketch: Update operation is responsible for updating the sketch to keep track of flow packet counts, while Query operation retrieves the estimated number of packets for a specific flow. For a packet stream in the programmable switches, it works as follows: considering an $N_h \times N_s$-sized Count Sketch CS, the flow key of the packets id is hashed by two hash functions $h_j(id)$ and $g_j(id)$ in j-th row of the sketch ($0 \leq j \leq N_h - 1$). The output of $h_j(id)$ is the index q ($0 \leq q \leq N_s - 1$) of the row j to store the packet count of flow id, and that of $g_j(id)$ is either -1 or 1. The result of $g_j(id)$ is then added to current counter $CS_{i,j}$. Finally, the packet count of the flow is queried with the median value of all counters in $g_j(id) \cdot CS_{j,h_j(id)} \forall j$.

10.3 Sketch-based flow moments estimation for network monitoring

In this section, we present SEAMEN, a flow moments-based approach to track the overall condition of the network traffic. SEAMEN consists of two components, data plane switch part and controller part. In data plane switch, SEAMEN performs the updates in the registers (i.e. HyperLogLog and Count Sketch) to record the packet and flow statistics during time interval, while in controller, SEAMEN retrieves the available information from data plane switch to estimate flow moments at the end of the time interval, and then monitors the network status. In the following, we report these two operations in detail.

10.3.1 Moments of data stream in programmable data planes

In this section, we present the kth-order ($0 \leq k \leq 2$) moments of data stream $\sum_{i=1}^{n_{tot}} (f_i)^k$ and logarithmic order $\sum_{i=1}^{n_{tot}} f_i \log_d f_i$. Each of them can quantify the overall condition of the network traffic. We can measure the flow moment at regular time intervals and obtain a time series about this metric. Then, by testing whether its short-term change exceeds a threshold, we may detect the anomalies in the network.

The zeroth-order moment

The zeroth-order moment $\sum_{i=1}^{n_{tot}} (f_i)^0$ is equal to the number of flows n_{tot}, which can be used to understand how many distinct flows there are in the switch.

The first-order moment

The first-order moment $\sum_{i=1}^{n_{tot}} (f_i)$ is equal to the number of packets S_{tot}, which indicates the number of packets crossing the switch.

The second-order moment

The second-order moment of all flow sizes $SumSquare = \sum_{i=1}^{n_{tot}} f_i^2$, can be used to calculate the variance of the size distribution of all flows.

$$SumSquare+ = F(f_i) = f_i^2 - (f_i - 1)^2 = 2f_i - 1$$

The logarithmic-order moment

The logarithmic-order moment of all flow sizes $SumLog = \sum_{i=1}^{n_{tot}} f_i \log_d f_i$. We use Euler's number e as the base d, and so in our case $SumLog = \sum_{i=1}^{n_{tot}} f_i \ln f_i$. This metric can be used to measure the diversity of the size distribution of all flows.

$$SumLog+ = G(f_i) = f_i \ln(f_i) - (f_i - 1)\ln(f_i - 1)$$

10.3.2 Estimate the variance of flow size and network traffic entropy in controller

Problem definition

Given:

- A time interval T_{int}
- A packet stream $S = \{f_1, f_2, \cdots, f_{n_{tot}}\}$
- Number of packets $S_{tot} = \sum_{i=1}^{n_{tot}} f_i$
- Estimated number of distinct flows by HyperLogLog $\hat{n}_{tot}$
- The second-order moment of all flow sizes $SumSquare = \sum_{i=1}^{n_{tot}} f_i^2$
- The logarithmic-order moment of all flow sizes $SumLog = \sum_{i=1}^{n_{tot}} f_i \log_{f_i}$

Return: an estimated variance of flow packet count $\hat{Var}[f_i]$ in packet stream S

Variance of flow size

The variance $Var[f_i]$ can be expanded as follows

$$Var[f_i] = \mathbb{E}[f_i^2] - (\mathbb{E}[f_i])^2$$

Since $\mathbb{E}[f_i^2] = \frac{SumSquare}{n_{tot}}$ and $\mathbb{E}[f_i] = \frac{S_{tot}}{n_{tot}}$, we estimate $Var[f_i]$ as:

$$\hat{Var}[f_i] = \frac{SumSquare}{\hat{n}_{tot}} - (\frac{S_{tot}}{\hat{n}_{tot}})^2$$

Application study: heavy hitter detection By Central Limit Theorem, it is deserved to note that the mean value of any k flow packet counts $\frac{\sum_{i=1}^{k} f_i}{k}$ follows a a normal distribution $\frac{\sum_{i=1}^{k} f_i}{k} \sim \mathcal{N}(\mu, \frac{\sigma^2}{k})$, where μ is $\frac{S_{tot}}{n_{tot}}$ and $\sigma^2 = \frac{SumSquare}{n_{tot}} - (\frac{S_{tot}}{n_{tot}})^2$.

Then it is possible to use standard normal distribution table to analyze the desired false positive rate (FPR), that is, $FPR = 1 - \Phi(x) = 1 - \mathbb{P}(\frac{\sum_{i=1}^{k} f_i}{k} < x)$ where $\Phi(x)$ is the cumulative distribution function for the standard Normal distribution $\mathcal{N}(0,1)$. For example, if x is set to 2, the probability of $|\frac{\sum_{i=1}^{k} f_i}{k} - \mu| \leq 2\frac{\sigma}{\sqrt{k}}$ is 95%, and FPR should be only 5%. While $x = 3$ is chosen, FPR is just 0.3%. Relying on such an analysis, the variance can be used to detect heavy hitters. If we track k flows with packet count f_i (e.g. a group of flows towards the same subnet) that satisfy $\frac{\sum_{i=1}^{k} f_i}{k} > \mu + 3\frac{\sigma}{\sqrt{k}}$, this means that there exist heavy hitters among k flows, and they can be detected with only 0.3% false positive rate.

Normalized entropy of flow size

The Shannon entropy of flow size H can be rewritten as:

$$\begin{aligned} H &= -\sum_{i=1}^{n_{tot}} \frac{f_i}{S_{tot}} \ln \frac{f_i(S_{tot})}{S_{tot}} \\ &= \ln S_{tot} - \frac{1}{S_{tot}} \sum_{i=1}^{n_{tot}} f_i \ln f_i \end{aligned}$$

The entropy is estimated as:

$$H = \ln S_{tot} - \frac{1}{S_{tot}} SumLog$$

and its corresponding normalized entropy is estimated:

$$H_{norm} = \frac{\ln S_{tot} - \frac{1}{S_{tot}} SumLog}{\ln \hat{n}_{tot}}$$

Application study: Volumetric DDoS detection As we explained in Chapter 9, the variation of normalized entropy is a good metric to detect DDoS attacks. Once the normalized entropy is retrieved from the programmable switch, there should be a broad range of approaches to track volumetric DDoS attacks, such as P4DDoS that we proposed in the last chapter.

10.4 Implementation in P4

We have successfully implemented our SEAMEN method, described in Fig. 10.2, in a network including a P4 programmable commodity switch with a Tofino ASIC and a simple controller. In this Section, we report the details of our implementation.

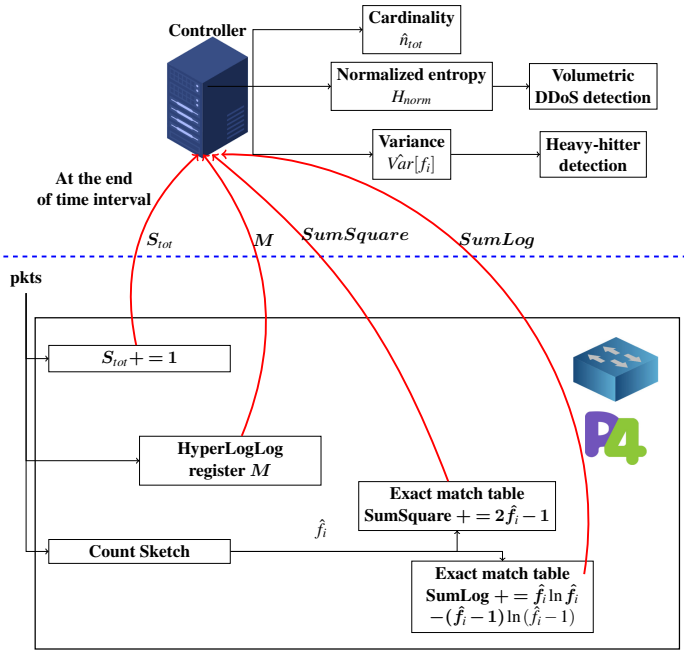


Figure 10.2: Scheme in programmable data planes

10.4.1 SEAMEN Update (Programmable switch)

Algorithm 12 reveals the implementation details in programmable data plane.

Counting the number of packets S_{tot}

We use a register (a counter could also be used) to count all the incoming packets in the switch (Line 7).

Updating the HLL Register

As shown in Line 8 to 12, we first concatenate CRC32 and CRC32c as the 64-bit hash function h to hash the flow id (e.g. any subset of 5 tuple) of the packet. The value $v(x)$ in HyperLogLog register (see Section 10.2.1) can be retrieved by using a ternary match table. For instance, given a mask $(0111)_2$ and a key 4, if the hashed value $x = h(id)[\log_2 m + 31 : \log_2 m]$ satisfying x & (AND) $mask = key$, that is, the left most three bits of x is $(100)_2$, the output of ternary match table returns 3. We then compare $v(x)$ to the value at the j-th indexed register cell M_j, where j is the rightmost $\log_2 m$ bits of $h(id)$. In case that $v(x)$ is still greater than M_j, $v(x)$ replaces M_j.

Algorithm 12: Update (Programmable data plane)

Input: Packet stream S

Output: HyperLogLog register M, packet counter S_{tot}, sum of logarithmic-order moment *SumLog*, sum of second-order moment *SumSquare*

1 $m \leftarrow 2^l$ $(l \in \{4,...,16\})$, $M \leftarrow$ m-sized empty HyperLogLog register
2 $os \leftarrow 32$, $S_{tot} \leftarrow 0$, SumLog $\leftarrow 0$, SumSquare $\leftarrow 0$
3 $CS \leftarrow N_h \cdot N_s$-sized empty Count Sketch
4 **Function** *Update*(M)**:**
5 **for** *Each received packet belonging to flow id* **do**
6 $\mathscr{C} \leftarrow \{\}$
7 $S_{tot} \leftarrow S_{tot} + 1$
8 $s \leftarrow (Hash(id) \rightarrow \{0,1\}^{os})$
9 $x \leftarrow s[l+31:l]$
10 $v(x) \leftarrow$ (Index of rightmost one of x) $+ 1$
11 **if** $v(x) > M_j$ **then**
12 $M_j \leftarrow v(x)$
13 **for** *each row* j *in* N_h *hash functions* **do**
14 $q \leftarrow h_j(id)$
15 $CS[j][q] += g_j(id)$
16 $\mathscr{C}.add(g_j(id) \cdot CS[j][q])$
17 $\hat{f}_i \leftarrow median(\mathscr{C})$
18 $SumLog += \hat{f}_i \ln \hat{f}_i - (\hat{f}_i - 1)\ln(\hat{f}_i - 1)$
19 $SumSquare += 2\hat{f}_i - 1$
20 **return** M, S_{tot}, *SumLog*, *SumSquare*

Updating Count Sketch

Lines 13-16 report how we implement Count Sketch in programmable data planes following the procedure described in Section 10.2.2. The Count Sketch CS is composed by N_h registers and each with N_s outputs. Each register indexed j is associated with two different hash functions h_j and g_j. The hashed flow key id $q = h(id)$ indicates which counter in register j to update. Moreover, id is hashed by $g(id)$ with the output either -1 or 1, and then the counter q in register j is incremented $g(id)$. The estimated packet count of the incoming flow $\hat{f}_i$ is queried as the median of $g_j(id) \cdot CS[j][q]$ of all registers in Count Sketch, which can be computed by using several if-else conditions in the switch.

Updating *SumLog*

Once the flow packet count $\hat{f}_i$ is retrieved from the Count Sketch, we used an exact match table to compute $\hat{f}_i \ln \hat{f}_i - (\hat{f}_i - 1)\ln(\hat{f}_i - 1)$. This is because Tofino

Algorithm 13: Query (Controller)

$\alpha_m^{HLL} \leftarrow 0.7213/(1+1.079/m)$
Function *Query(M, S_{tot}, SumLog, SumSquare)*:
  if *At the end of time interval* **then**
    $\hat{n}_{tot} \leftarrow (\alpha_m^{HLL} \cdot m^2 \cdot \sum_{j=0}^{m-1} 2^{-M_j})$
    $H_{norm} \leftarrow \frac{\ln S_{tot} - \frac{1}{S_{tot}} SumLog}{\ln \hat{n}_{tot}}$
    $\hat{Var}[f_i] \leftarrow \frac{SumSquare}{\hat{n}_{tot}} - (\frac{S_{tot}}{\hat{n}_{tot}})^2$
    return $\hat{n}_{tot}$, H_{norm}, $\hat{Var}[f_i]$

ASIC switching does not support complicated computations on the metadata but we can pre-load the computational result in the match-action table. If an input $\hat{f}_i$ is matched in the table, the corresponding result of $\hat{f}_i \ln \hat{f}_i - (\hat{f}_i - 1)\ln(\hat{f}_i - 1)$ as a fixed point value is retrieved. Then the computed value is added to current counter in *SumLog*.

Updating *SumSquare*

Similar to what we did in *SumLog*, $2\hat{f}_i - 1$ is calculated from an exact match table and summed into the counter in *SumSquare*.

10.4.2 SEAMEN Query (Controller)

At the end of time interval, the controller retrieves the register values, including packet counter S_{tot}, HyperLogLog register M, sum of logarithmic-order moment *SumLog*, sum of second-order moment *SumSquare*. After that, the controller resets all counters in the switches. Then the estimated normalized entropy, variance, flow cardinality traversing the switch can be easily computed as what we analyzed in Section 10.3.2. This is because the controller is not limited by hardware constraint as programmable switches, and so it has more computational resources. The two applications, volumetric DDoS detection and heavy-hitter detection can be built on top of estimated normalized entropy and variance, and we consider them as our future work.

10.5 Evaluation

10.5.1 Testing flow trace and default settings

Testing flow trace

We use a 50-seconds 2018-passive CAIDA flow trace [3] for evaluation. The 50-seconds flow trace is divided into 50 1-second time intervals. Each time interval contains around 450 thousand packets.

Metrics

We consider *relative error* as metric to evaluate the estimation performance. We call $\hat{H}_{norm}$ the estimated normalized traffic entropy in a time interval and H_{norm} its exact value. The relative error is defined as the average of $\frac{|H_{norm}-\hat{H}_{norm}|}{H_{norm}} \cdot 100\%$ in 50 time intervals. Similarly, being $\hat{Var}$ the estimated variance of flow packet count crossing the switch and Var its exact value, the relative error is the mean value of $\frac{|Var-\hat{Var}|}{Var} \cdot 100\%$ in 50 time intervals.

Tuning parameters

Unless otherwise specified, the default HyperLogLog register size is $2^{10} = 1024$, and the register cell size is 5 bits each. The Count Sketch is composed by 3 registers with output size $2^{11} = 2048$ each. The default flow key is $\{srcIP, dstIP\}$ pair.

10.5.2 Evaluation of accuracy on normalized entropy and variance estimation varying tuning parameters

We now evaluate how the accuracy of SEAMEN is sensitive to different tuning parameters. The goal of this work is to assure the relative error of estimated variance and normalized entropy is below 3%, which is the largest value that does not affect practical network monitoring performance [84].

Sensitivity to number of hash functions N_h

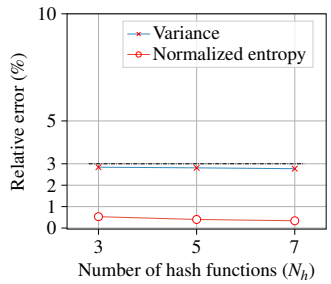
As shown in Figure10.3(a), we varied the number of hash functions N_h from 3 to 7. Choosing N_h as an odd number is recommended since there is only a unique median. The results show that the relative error of variance and normalized entropy does not vary much. This is because when the sketch size is large enough, the bias caused by the sketch is small, and so the accuracy of the estimations remains stable.

Sensitivity to output size N_s

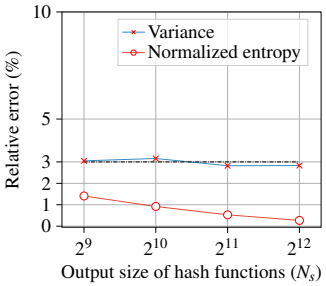
Figure10.3(b) reveals that the accuracy of normalized entropy estimation is more sensitive to the output size of hash functions: increasing the output size from 2^9 to 2^{12} would decrease more than 1% relative error, but all of them have the relative error below 3%. Thus, the choice of N_s depends on the accuracy of variance. It can be noted that the relative error of variance is below 3% once N_s reaches 2^{11}.

Sensitivity to HyperLogLog register size m

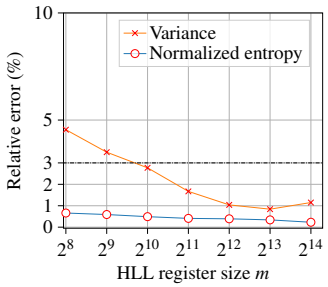
Figure10.3(c) shows the performance of SEAMEN by varying the HyperLogLog register size m. Intuitively, the relative error of variance and normalized entropy decreases as m increases. To guarantee the relative error of both of them is below



(a) Sensitivity to N_h ($N_s = 2048$ and $m = 1024$)



(b) Sensitivity to N_s ($N_h = 3$ and $m = 1024$)



(c) Sensitivity to m ($N_h = 3$ and $N_s = 2048$)

Figure 10.3: Sensitivity analysis

3% while minimizing the memory usage in the switch, $m = 2^{10}$ is a preferable solution.

10.5.3 Evaluation of resource usage in physical testbed

Table 10.1 shows the normalized switch's data plane resources usage of simple forwarding and SEAMEN. To assure a fair comparison, the implementation of SEAMEN also contain the simple forwarding for packet processing.

While simple forwarding occupies 16.67% stages in the programmable hardware switch, implementing SEAMEN requires additional 41.66% of overall stages. Simple forwarding needs 2.5% of the total available SRAM since we assigned the exact match-action table for packet forwarding the maximum size. The number of needed ALUs indicates the computational resource usage. 12.5% of total ALUs are used by SEAMEN plus simple forwarding to process the packets, and all ALUs are performed in registers to update the count.

The packet header vector (PHV) size shows the amount of customized metadata in packet header that is used for packet processing. SEAMEN plus simple forwarding uses 11.98% of PHV and, compared to the 7.30% of simple forwarding, meaning that they only require few additional customized metadata to pass across stages.

Table 10.1: Switch resource usage of SEAMEN

Strategy	No. stages	SRAM	No. ALUs	PHV size	Processing time w.r.t. simple forwarding
Simple forwarding	16.67%	2.5%	4.2%	7.30%	-
SEAMEN + Simple forwarding	58.33%	6.04%	12.5%	11.98%	71ns

Finally, we perform the additional processing time with respect to simple forwarding. To implement SEAMEN in ASIC, only 71ns are necessary.

10.6 Related work

10.6.1 Sampling-based and sketch-based monitoring

sFlow [102] and NetFlow [9] are two well-known approaches that provide generic support for different measurement tasks: they collect flow-level counts for sampled packets in the data plane to diagnose the network performance and security problem. However, A high sampling rate would lead to a large number of counters, while a lower sampling rate may miss flows. This means that to assure a good performance on network monitoring, both approaches need to too large resources, including CPU, memory, and bandwidth.

As an alternative solution, many sketch-based streaming algorithms have been proposed in research community [93][132], which provide efficient measurement support for several management tasks. With respect to Sampling-based solutions, sketch-based solutions can guarantee higher accuracy while using the same memory in the switch.

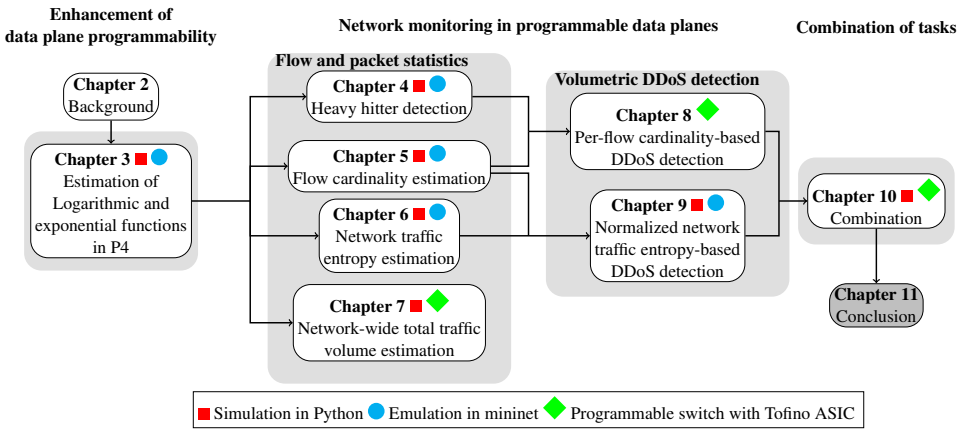
10.6.2 Network monitoring tasks using ASIC switching

Most of legacy SDN switches (e.g. Openflow-supported switches) come with very limited programmability with respect to the data plane functions that can be executed. To enable new kinds of network monitoring functionalities in the switch it is necessary to upgrade the hardware, which incurs significant additional cost. Recently, programmable ASICs have been introduced: they ensure standard data plane features (i.e., high-speed switching and forwarding) while offering the possibility of customizing new functionalities, if properly programmed through domain-specific programming languages like P4 [42]. For example, programmable switches equipped with Tofino ASIC [2] can always forward packets at line-rate once the P4 monitoring program is compiled and installed in the switches. Even though network monitoring can be executed in the server, but they cannot currently ensure high throughput and low latency as ASICs [136].

10.7 Concluding remarks

In this chapter, we successfully combined the monitoring tasks that we proposed in the thesis. While implementing all tasks entirely in the programmable switches is challenging due to the limited hardware resource, we only summarized the flow statistics in the data plane and but left the more complicated computations for controller. In this way, the switches can keep processing the packets at line-rate, while the controller can understand the network status by retrieving the information from the switches.

11 Conclusion



In this final chapter we summarize and conclude our research presented in earlier chapters. We first draw our main contributions in the thesis, with respect to the three goals that we set out with at the start of this thesis. We then revisit and discuss in detail each research question defined in the first chapter. Finally, we provide prospects for future research that can build on the research presented in this thesis.

11.1 Main conclusion

The absence of high-speed monitoring in modern computer networks constraints the performance to detect network anomalies and failures. Network monitoring in programmable data planes therefore becomes an appealing idea due to its fast packet processing. As a result, the main contribution of this thesis is to leverage network monitoring and opportunities arising from the data plane programmability to track current network status and detect possible threats in the network.

At the start of this thesis we identified three challenges surrounding this topic: First, the design of network monitoring solutions needs to consider limited computational and memory resources in programmable switches. Second, how to offload monitoring tasks into programmable switches without overwhelming resources; Third, how to allocate resources for multiple monitoring tasks in the programmable switch. Our goals, which we will revisit them later, are related to these three challenges. We believe that the efforts to offload network monitoring into programmable switches are indispensable contributions for modern telecommunication networks.

We now draw our conclusions for each of the three goals that we presented at the start of this thesis.

11.1.1 Goal 1

***Goal 1**: to investigate network monitoring and programmable data plane in Software-Defined Networks, learn how to program switches and enhance their P4-enabled data plane programmability for monitoring*

The first goal focuses on understanding the state of the art on network monitoring and programmable data planes in Software-Defined Networks The background information has been described in Chapter 2. While we knew that the advantages of programmable switch are line-speed packet processing and flexible program customization, we planned to enable more new monitoring functionalities directly in the switch to accelerate the network. We chose P4 as the programming language to develop new programs in the switches. During the learning process, we discovered some inherent limitations of P4 for the implementation of monitoring tasks. To overcome those limitations, in Chapter 3 we propose new methods to approximate some arithmetic operations that P4 does not support, including approximation logarithmic and exponential function estimation. They can be used as building blocks to support further implemented monitoring tasks.

11.1.2 Goal 2

In the introduction of this thesis we pointed out the importance of network monitoring in programmable data planes. Our second goal was defined with this in mind:

***Goal 2**: to study network monitoring tasks and how to offload them in programmable data planes*

From Chapter 4 to Chapter 9, we studied five different kinds of monitoring tasks: heavy-hitter detection, flow cardinality estimation, network traffic entropy estimation, total traffic volume estimation, and volumetric DDoS detection. We

would like to offload as much as possible the network monitoring functionalities into the switch, that is, in the best case, the task can be executed entirely in programmable data planes to perform in-network monitoring. Given this goal, we started with simulations to understand the behaviors of various network monitoring tasks. We then designed and implemented new monitoring ideas in P4, and tested the performance in an emulated environment. Finally, we set up the programmable switches in our physical testbed and tried to migrate available monitoring solutions from emulated environment to real hardware switch. In Chapter 4, we first designed a network-wide heavy hitter detection robust to partial deployment of programmable switches in ISP networks. The switches only report flows with large packet counts to controller for further network-wide detection according to a global threshold. The cardinality estimation (Chapter 5), entropy estimation (Chapter 6), and normalized entropy-based DDoS detection (Chapter 9) have been fully implemented in P4 and can be executed in an emulated environment. This means that they can act as a software switch in the server. However, we failed to migrate them into programmable switches with ASIC due to the hardware resource constraints. Hence, with this in mind, after this work, we started designing monitoring ideas taking the hardware limitations into consideration. We therefore overcame the limitations and implemented total traffic volume estimation in Chapter 7 and flow cardinality-based volumetric DDoS detection in Chapter 8.

11.1.3 Goal 3

The final technical challenge that we pointed out relates to the combination of multiple monitoring tasks. The different tasks have different requirements, how to allocate the limited switch resources to each task is an open topic to discuss. With this in mind, we defined Goal 3 as:

***Goal 3**: to combine studied monitoring tasks in suitable network scenarios composed of programmable switches*

We first revisited our designed tasks in previous chapters, and sought the possibilities to migrate them into hardware switch With this in mind, we discovered that implementing all tasks entirely in the switch is not practical due to both the limitations in hardware and programming language P4. We therefore designed an alternative solution in programmable data planes: we only track the flow moments in programmable data plane. The tracked flow moments can provide SDN controller support to understand overall condition of the network, including variance, flow cardinality, and entropy estimation These tasks can then further be used in controller to detect heavy hitters and volumetric DDoS attacks.

11.2 Research questions revisited

In this section, we provide the answers for our RQs defined in Chapter 1 (Introduction). Each section contains the set of research questions that relate to a particular goal.

11.2.1 Research Questions for Goal 1

In this section we discuss each of the three research questions that relate to the first goal of this thesis.

> ***RQ1****: Why network monitoring is so important in modern telecommunication networks? In particular, what are the benefits to implement it in the data plane?*

We studied this research question in nearly every chapter of this thesis. Network monitoring is the main enabler of various network management tasks, ranging from accounting, traffic engineering, anomaly detection, Distributed Denial-of-Service (DDoS) detection, Super-spreader detection, and scans detection. With the advent of programmable data planes in Software-Defined Networking (SDN), the monitoring functionalities can be offloaded into switches to diagnose performance and security issues while processing packets at line rate. For instance, in a programmable switch equipped with ASIC, the latency is only in the order of nanoseconds.

Our second research question, RQ 2, built on the advantages of programmable data planes answered in RQ1:

> ***RQ2****: How does a programmable data plane work? How is the data plane of a switch programmed? What are the limitations of this type of functionality?*

The answer is covered by Chapter 2. In the data plane of programmable switches, packets are processed sequentially in a pipeline, and the pipeline can be customized by a high-level domain-specific language named P4. In P4 programs,the developers can indicate which packet headers to modify and customize the match-action tables for the purpose of forwarding and computations. In order to assure fast packet forwarding, P4 removes many arithmetic and logical operations that may delay the packet processing, including loops (e.g. *For* and *While*), division, logarithm, exponentiation and floating numbers. Then the developed program can be compiled and installed into programmable switches, and meanwhile the corresponding interactive APIs are generated. Finally, the generated APIs are used by control plane applications to communicate with the data plane.

Our third research question, RQ 3, relates to the enhancement of data plane programmability:

> ***RQ3**: Is there any way to improve P4-enabled data plane programmablity? If yes, how?*

In Chapter 3, we revealed that we used some mathematical operations in P4 to approximate logarithm and exponential function, as well as the division. With respect to state-of-the-art solutions, our approximations only incur slightly higher packet processing time but does not require any match-action table to work. Those approximations are non-trivial building blocks for some monitoring tasks reported in the latter chapters.

11.2.2 Research Questions for Goal 2

In this section we discuss the five research questions for the second goal of this thesis, which concerns specific monitoring task in programmable data planes. The first research question in this goal relates to heavy-hitter detection:

> ***RQ4**: What is heavy-hitter detection? Why do we need to track heavy flows? How to detect them by using the programmable switch? What is the performance in a network composed of partially deployed programmable switches?*

We tackled RQ 4 in Chapter 4 of this thesis, in which we identify heavy hitters as the flows that carry more than a given fraction of total traffic volume in the network. We first used Count-min Sketch to filter the flows with relatively large packet counts during a given time interval. At the end of time interval, the controller retrieved the filtered heavy flows from all programmable switches in the network. Finally, if the packet counts of heavy flows are greater than the global threshold, the network-wide heavy hitters are detected. In order to show that our network-wide heavy-hitter detection is robust to partial deployment scenario, we then propose an incremental deployment of programmable switches approach in Internet Service Provider (ISP) networks with the goal to have visibility over the largest number of distinct flows. The results show that when only a limited number of programmable switches is deployed, our network-wide heavy-hitter detection strategy outperforms an existing approach in terms of detection accuracy, memory occupation and communication overhead.

Our fifth research question, RQ 5, relates to flow cardinality:

***RQ5**: What is flow cardinality estimation? Why is it necessary for monitoring? How do we design new idea for flow cardinality estimation on a programmable switch? What is the performance with respect to the state-of-the-art?*

Counting distinct flows or flow cardinality estimations are widely used in network monitoring for security. They can be used, for example, to detect the malware spread, network scans, or a denial of service attack. Many network anomalies and cyber-attacks often involve sudden changes in the cardinality of the traffic data that are related to them In Chapter 5, we studied LogLog data structure to estimate the flow cardinality of large data streams composed by millions of packets. With respect to state-of-the-art solution, our new flow cardinality estimator in P4 program can guarantee high accuracy while ensuring small memory usage.

The sixth research question RQ 6 is

***RQ6**: What is network traffic entropy estimation? Which metric of network does network traffic entropy indicate? Any problems while implementing it in the data plane of programmable switch?*

The *entropy* of flow size indicates the network traffic distribution. Traffic entropy reaches 0 when all packets belong to the same flow, which means that there is only one flow in the network. On the other hand, it reaches its maximum value when the flows are uniform distributed, i.e., each flow carries the same number of packets. As network traffic entropy relies on the calculation of logarithm and division, in Chapter 6, we designed a time interval-based entropy estimation strategy relying on the estimations proposed in Chapter 3. A prototype has been implemented in P4 behavioral model and has been proven to be fully executable in a P4 emulated environment.

In RQ 7, we pose a new challenge: how SDN controller coordinates multiple programmable hardware switches to perform network-wide monitoring?

***RQ7**: Why do we need to know network-wide total traffic volume estimation? What is the key problem to coordinate multiple programmable switches for packet counting in the network? How do we solve it?*

In Chapter 7, we presented a novel traffic volume estimation method that exploits data-plane programmable switches to estimate number of flows, average flow size and total packet count in the network, which are necessary to support a broad range of monitoring tasks that we mentioned in the thesis. Most network-wide monitoring systems assume that each packet is monitored and counted by a single programmable switch on its path through the network, which limits the routing or requires coordination among switches. In the network that packets traverse mul-

tiple switches, the proposed strategies are not accurate any more. We named this problem *packet double counting*. We solved the double counting problem relying on Central limit theorem and theoretically demonstrate that our method an unbiased estimator of total traffic network volume. Finally, we overcame the hardware resource constraints and successfully implemented the prototype on top of a HyperLogLog data structure in an Edgecore commodity switch equipped with Tofino ASIC.

Our final research question RQ8 in this goal is about problems with volumetric DDoS detection:

***RQ8**: What is volumetric DDoS attack? What are the detection methods? What is the difference between different methods? How to design and implement them in programmable switches?*

Volumetric DDoS attacks are a critical security threat that aims to overwhelm the available resources of victims. The goal of volumetric DDoS detection in programmable data planes is to use some common metrics to detect potential DDoS attacks. In Chapter 8, we first investigated DDoS detection based on per-flow cardinality. This is because when an attacker launches a number of bots to flood a large amount of attack traffic to the victim, the per-source flow cardinality to the destination victims will significantly increase. This prompted us to track the per-source flow cardinality to all the destinations. An unexpected large increment of per-source flow cardinality may indicate that a volumtric DDoS attack is taking place. Another metric that we discovered to detect volumetric DDoS attacks is the change of normalized entropy. As we mentioned in Chapter 9, the entropy indicates the flow distribution in the network, when there are a number of distinct flows with large packet counts towards the same destination, the entropy will drastically decrease. Taking the flow fluctuation into consideration, normalized entropy is a more suitable metric to detect volumetric DDoS attacks. The differences between them are twofold: First, entropy-based DDoS detection can only detect DDoS attacks, but per-flow cardinality-based DDoS detection is also able to identify the DDoS victims. Second, per-flow cardinality-based DDoS detection requires more switch resource usages than entropy-based solution.

11.2.3 Research Questions for Goal 3

In this section we discuss the research questions for the final goal of this thesis, which concerns problems with combination of tasks.

***RQ9**: How to coordinate multiple monitoring tasks in a single programmable hardware switch while overcoming the resource limitations?*

In Chapter 10, we combined several monitoring tasks in a programmable switches equipped with Tofino ASIC. While we realized that implementing a number of tasks into programmable switches is not feasible, we used two sketches, Count Sketch and HyperLogLog, only for storing the flow and packet statistics. The SDN controller is responsible for retrieving the summarized statistics and performing estimations on the metrics required by a wide range of monitoring tasks. In this way, the packets can be processed at line rate in the programmable switches while the operator can get the necessary information for monitoring.

11.3 Prospects for future research

In this final section of our final chapter we discuss prospects for future research. We imagine three directions to advance our understanding of network monitoring problem and to increase situational awareness about the security and robustness of modern telecommunications networks.

Machine-learning assisted monitoring: Machine learning is recognized as primary anomaly detection methodology due to its ability to automatically learn hidden patterns in traffic flows, which can help to solve some inherent problems for monitoring. For instance, most monitoring tasks, such as DDoS detection and heavy-hitter detection, rely on a threshold to detect network anomalies, how to effectively set a suitable threshold to maximize the detection performance (e.g. F1 score) poses a new challenge. In this thesis, the threshold is configured based on statistical methods, and we believe that building new strategies on machine learning in SDN controller to provide adaptive thresholds is another interesting topic.

The protection of compact data structures for monitoring: The monitoring tasks in programmable data planes usually accompanies one or several compact data structures to filter and summarize the flow and packet statistics. However, the security and robustness of those compact data structures have not been taken into consideration. For instance, an attacker can attempt to modify the counter values in Count-min Sketch, which may lead to a number of undetected heavy hitters. Similarly, a DDoS attack can evade the increments in HyperLogLog to avoid the detection. We therefore consider the compact data structure protection as potential future works.

Integrating programmable switches into a large-scale monitoring system: This thesis focuses on offloading some application-layer tasks to programmable switches for line-speed monitoring. We realize that the programmable switches can only process the packet headers and perform very limited number of functionalities with respect to the servers. Hence, we believe that combing SDN controller, high-speed forwarding programmable switches, and high-performance servers into a large-scale monitoring system is a more preferable solution. While the programmable switches perform line-rate packet processing and update the raw data into compact data structures, the servers analyze the statistics from the switches and provide the support for data storage (e.g., key-value stores) and network protection (e.g. DDoS mitigation). Finally, SDN controller is responsible to synchronize the states of switches and to carry out network-wide monitoring. We set this work as our next goal in the future.

Bibliography

[1] Apache thrift. https://thrift.apache.org/.

[2] Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[3] CAIDA UCSD anonymized internet traces dataset - [passive-2018]. http://www.caida.org/data/passive/passive_dataset.xml.

[4] Google protocol buffers. https://developers.google.com/protocol-buffers/.

[5] grpc. https://grpc.io/.

[6] GÉANT network. http://https://www.geant.org/.

[7] iPerf3. https://iperf.fr/iperf-doc.php.

[8] Mininet. http://mininet.org/.

[9] Netflow. http://www.cisco.com/warp/public/732/Tech/netflow.

[10] Netronome SmartNICs. https://www.netronome.com/products/agilio-cx/.

[11] Open network operating system. https://opennetworking.org/onos/.

[12] Opendaylight. https://www.opendaylight.org/.

[13] P4 behavioral model. https://github.com/p4lang/behavioral-model.

[14] P4 implementation of Count-Min sketch. https://github.com/open-nfpsw/M-Sketch/tree/master/count_min_Vanilla_P4.

[15] P4 implementation of INDDoS. https://github.com/DINGDAMU/INDDoS.

[16] P4 implementation of NWHHD+. https://github.com/DINGDAMU/Network-wide-heavy-hitter-detection.

[17] The P4 language specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[18] P4 register definition. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[19] P4c. https://github.com/p4lang/p4c.

[20] P4DDoS source code. https://github.com/DINGDAMU/P4DDoS.

[21] P4Entropy source code. https://github.com/DINGDAMU/P4Entropy.

[22] P4Entropy source code. https://github.com/DINGDAMU/P4Entropy_normalized.

[23] P4Log and P4Exp source code. https://github.com/DINGDAMU/P4Log_and_P4Exp.

[24] P4LogLog source code. https://github.com/DINGDAMU/P4LogLog.

[25] Python implementation of Count-Min sketch. https://github.com/rafacarrascosa/countminsketch.

[26] Python implementation of HyperLogLog. https://github.com/ekzhu/datasketch.

[27] Ryu. https://ryu-sdn.org/.

[28] Tcpreplay. https://tcpreplay.appneta.com/.

[29] Wireshark. https://www.wireshark.org/.

[30] DEFO ISP topology, 2020. https://sites.uclouvain.be/defo.

[31] GÉANT ISP topology, 2020. https://www.geant.org/Networks/Pan-European_network/Pages/GEANT_topology_map.aspx.

[32] iPerf, 2020. https://iperf.fr/iperf-doc.php.

[33] P4 implementation of INVEST, 2020. https://anonymous.4open.science/r/166fa1a4-c56f-4d6e-ac23-a6d7a5167433/.

[34] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. Detecting heavy flows in the sdn match and action model. *Computer Networks*, 136:1–12, 2018.

[35] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. Detecting heavy flows in the sdn match and action model. *Computer Networks*, 136:1–12, 2018.

[36] Ran Ben Basat, Gil Einziger, Shir Landau Feibish, Jalil Moraney, and Danny Raz. Network-wide routing-oblivious heavy hitters. In *IEEE/ACM Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2018.

[37] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, Shir L. Feibish, Danny Raz, and Minlan Yu. Routing oblivious measurement analytics. *arXiv*, pages arXiv–2005, 2020.

[38] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *ACM COnference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2011.

[39] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, pages 1–12, 2011.

[40] Przemysław Bereziński, Marcin Szpyrka, Bartosz Jasiul, and Michał Mazur. Network anomaly detection using parameterized entropy. In *IFIP International Conference on Computer Information Systems and Industrial Management*, 2015.

[41] Abhinav Bhandari, AL Sangal, and Krishan Kumar. Destination address entropy based detection and traceback approach against distributed denial of service attacks. *International Journal of Computer Network and Information Security*, 7(8):9, 2015.

[42] Pat Bosshart, Dan Daly, Glen Gibb, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[43] Guy Castagnoli, Stefan Brauer, and Martin Herrmann. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. *IEEE Transactions on Communications*, 41(6):883–892, 1993.

[44] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Springer International Colloquium on Automata, Languages, and Programming (ICALP)*, 2002.

[45] Mosharaf Chowdhury, Rachit Agarwal, Vyas Sekar, and Ion Stoica. A longitudinal and cross-dataset study of internet latency and path stability. *Dept.*

EECS, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-172, 2014.

[46] Yann Collet. xxhash–extremely fast hash algorithm. https://github.com/Cyan4973/xxHash.

[47] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases*, 2011.

[48] Graham Cormode. Count-min sketch. In *Springer Encyclopedia of Database Systems*. pp. 511-516, 2009.

[49] Graham Cormode and Minos Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *International conference on Very large data bases*, pages 13–24, 2005.

[50] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the Count-min Sketch and its applications. *Elsevier Journal of Algorithms*, 55(1):58–75, 2005.

[51] M. Dimolianis, A. Pavlidis, and V. Maglaris. A multi-feature DDoS detection schema on P4 network hardware. In *Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020.

[52] Marinos Dimolianis, Adam Pavlidis, and Vasilis Maglaris. A multi-feature DDoS detection schema on P4 network hardware. In *Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020.

[53] D Ding, M Savi, G Antichi, and D Siracusa. Incremental deployment of programmable switches for network-wide heavy-hitter detection. In *IEEE Conference on Network Softwarization (NetSoft)*, 2019.

[54] D. Ding, M. Savi, G. Antichi, and D. Siracusa. An incrementally-deployable P4-enabled architecture for network-wide heavy-hitter detection. *IEEE Transactions on Network and Service Management*, 17(1):75–88, 2020.

[55] Damu Ding, Marco Savi, and Domenico Siracusa. Estimating logarithmic and exponential functions to track network traffic entropy in P4. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2020.

[56] Nick Duffield, Carsten Lund, and Mikkel Thorup. Charging from Sampled Network Usage. In *ACM Internet Measurement Workshop (IMW)*, 2001.

[57] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003.

[58] Cristian Estan and George Varghese. New Directions in Traffic Measurement and Accounting. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2002.

[59] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *ACM SIGCOMM conference on Internet measurement*, 2003.

[60] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic DDoS defense. In *USENIX Security Symposium*, 2015.

[61] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions On Networking*, 9(3):265–279, 2001.

[62] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *IEEE/ACM Transactions on Networking*, vol. 9, issue 3, 2001.

[63] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pp. 137-156, 2007.

[64] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[65] Kostas Giotis, Christos Argyropoulos, Georgios Androulidakis, Dimitrios Kalogeras, and Vasilis Maglaris. Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments. *Elsevier Computer Networks*, 62:122–136, 2014.

[66] Paul Goransson, Chuck Black, and Timothy Culver. *Software defined networks: a comprehensive approach*. Morgan Kaufmann, 2016.

[67] Yu Gu, Andrew McCallum, and Don Towsley. Detecting anomalies in network traffic using maximum entropy estimation. In *ACM SIGCOMM conference on Internet Measurement*, 2005.

[68] Arpit Gupta, Robert MacDavid, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *USENIX Networked Systems Design and Implementation (NSDI)*, 2016.

[69] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *ACM Symposium on SDN Research (SOSR)*, 2018.

[70] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. *ACM SIGCOMM computer communication review*, 45(4):15–28, 2015.

[71] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692, 2013.

[72] David Ke Hong, Yadi Ma, Sujata Banerjee, and Z Morley Mao. Incremental deployment of SDN in hybrid enterprise and ISP networks. In *ACM Symposium on SDN Research (SOSR)*, 2016.

[73] Meitian Huang and Weifa Liang. Incremental SDN-enabled switch deployment for hybrid software-defined networks. In *IEEE Computer Communication and Networks (ICCCN)*, 2017.

[74] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust network measurement for software packet processing. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[75] Qun Huang and Patrick PC Lee. A hybrid local and distributed sketching design for accurate and scalable heavy key detection in network data streams. *Computer Networks*, 91:298–315, 2015.

[76] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590. ACM, 2018.

[77] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined Wan. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[78] Libin Jiang, Devavrat Shah, Jinwoo Shin, and Jean Walrand. Distributed random access algorithm: scheduling and congestion control. *IEEE Transactions on Information Theory*, 56(12):6182–6207, 2010.

[79] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 35–49, 2018.

[80] Kübra Kalkan, Levent Altay, Gürkan Gür, and Fatih Alagöz. JESS: Joint entropy-based DDoS defense scheme in SDN. *IEEE Journal on Selected Areas in Communications*, 36(10):2358–2372, 2018.

[81] Tammo Krueger, Christian Gehl, Konrad Rieck, and Pavel Laskov. An architecture for inline anomaly detection. In *European Conference on Computer Network Defense*, 2008.

[82] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 201–206, 2004.

[83] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. In *ACM SIGCOMM Computer Communication Review*, volume 35, 2005.

[84] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pp. 145-156, 2006.

[85] Angelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. Offloading real-time DDoS attack detection to programmable data planes. In *IEEE/IFIP Symposium on Integrated Network and Service Management (IM)*, 2019.

[86] Anna T. Lawniczak and Shengkun Xie. Number of packets in transit as a function of source load and routing. *Procedia Computer Science*, 1(1):2363–2370, 2010.

[87] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, and Anja Feldmann. Panopticon: Reaping the benefits of incremental SDN deployment in enterprise networks. In *USENIX Annual Technical Conference*, 2014.

[88] Shengru Li, Daoyun Hu, Wenjian Fang, Shoujiang Ma, Cen Chen, Huibai Huang, and Zuqing Zhu. Protocol oblivious forwarding (pof): Software-defined networking with enhanced programmability. *IEEE Network*, 31(2):58–66, 2017.

[89] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[90] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2016.

[91] Yang Liu, Wenji Chen, and Yong Guan. Identifying high-cardinality hosts from network-wide traffic measurements. *IEEE Transactions on Dependable and Secure Computing*, 13(5):547–558, 2016.

[92] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. ACM, 2019.

[93] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[94] Xinlei Ma and Yonghong Chen. DDoS detection method based on chaos analysis of network traffic entropy. *IEEE Communications Letters*, 18(1):114–117, 2013.

[95] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (recently) frequent items in distributed data streams. In *IEEE International Conference on Data Engineering (ICDE)*, pages 767–778, 2005.

[96] M. Markovitch and S. Schmid. Shear: A highly available and flexible network architecture marrying distributed and logically centralized control planes. In *IEEE International Conference on Network Protocols (ICNP)*, 2015.

[97] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communication Review*, 38(2), 2008.

[98] Seyed Mohammad Mousavi and Marc St-Hilaire. Early detection of ddos attacks against sdn controllers. In *2015 International Conference on Computing, Networking and Communications (ICNC)*, pages 77–81. IEEE, 2015.

[99] Mark EJ Newman. Scientific collaboration networks. II. shortest paths, weighted networks, and centrality. *APS Physical review E*, 64(1), 2001.

[100] George Nychis, Vyas Sekar, David G. Andersen, Hyong Kim, and Hui Zhang. An Empirical Evaluation of Entropy-Based Traffic Anomaly Detection. In *ACM SIGCOMM Conference on Internet Measurement*, 2008.

[101] Kazuya Okamoto, Wei Chen, and Xiang-Yang Li. Ranking of closeness centrality for large-scale social networks. In *Springer International Workshop on Frontiers in Algorithmics*, 2008.

[102] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. 2001.

[103] Kun Qian, Sai Ma, Mao Miao, Jianyuan Lu, Tong Zhang, Peilong Wang, Chenghao Sun, and Fengyuan Ren. FlexGate: High-performance heterogeneous gateway in data centers. In *Asia-Pacific Workshop on Networking (APNet)*, 2019.

[104] Maurice H Quenouille. A relation between the logarithmic, Poisson, and negative binomial series. *Biometrics*, 5(2):162–164, 1949.

[105] Sivaramakrishnan Ramanathan, Jelena Mirkovic, Minlan Yu, and Ying Zhang. SENSS against volumetric DDoS attacks. In *Annual Computer Security Applications Conference*, 2018.

[106] Pedro Reviriego and Daniel Ting. Security of HyperLogLog (HLL) Cardinality Estimation: Vulnerabilities and Protection. *IEEE Communications Letters*, 24(5):976–980, 2020.

[107] K. Salah and F. Haidari. Performance evaluation and comparison of four network packet rate estimators. *AEU - International Journal of Electronics and Communications*, 64(11):1015–1023, 2010.

[108] J.J. Santanna, R. van Rijswijk-Deij, R. Hofstede, A. Sperotto, M. Wierbosch, L. Zambenedetti Granville, and A. Pras. Booters - An analysis of DDoS-as-a-service attacks. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015.

[109] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[110] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[111] Naveen Kr Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[112] Ashutosh Kumar Singh and Shashank Srivastava. A survey and classification of controller placement problem in sdn. *International Journal of Network Management*, 28(3):e2018, 2018.

[113] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM Symposium on SDN Research (SOSR)*, 2017.

[114] Dirk J Struik. The origin of L'Hopital's rule. *The Mathematics Teacher*, 56(4):257–260, 1963.

[115] D. Sundararajan. Convolution and correlation. In *Wiley Discretewavelet Transform*, pp. 21-36, 2015.

[116] Lu Tang, Qun Huang, and Patrick PC Lee. SpreadSketch: Toward invertible and network-wide detection of superspreaders. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2020.

[117] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. OpenNetMon: Network monitoring in OpenFlow software-defined networks. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2014.

[118] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[119] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central control over distributed routing. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[120] Arno Wagner and Bernhard Plattner. Entropy based worm and anomaly detection in fast IP networks. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE)*, 2005.

[121] Chenxu Wang, Tony TN Miu, Xiapu Luo, and Jinhe Wang. SkyShield: a sketch-based defense system against application layer DDoS attacks. *IEEE Transactions on Information Forensics and Security*, 13(3):559–573, 2018.

[122] Rui Wang, Zhiping Jia, and Lei Ju. An entropy-based distributed DDoS detection mechanism in software-defined networking. In *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[123] Henry S Warren. Hacker's delight. In *Pearson Education*, 2013.

[124] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[125] Kehe Wu, Long Chen, Shichao Ye, and Yi Li. A load balancing algorithm based on the variation trend of entropy in homogeneous cluster. *International journal of grid and distributed computing*, 7(2):11–20, 2014.

[126] Qingjun Xiao, Shigang Chen, You Zhou, Min Chen, Junzhou Luo, Tengli Li, and Yibei Ling. Cardinality estimation for elephant flows: a compact solution based on virtual register sharing. *IEEE/ACM Transactions on Networking*, 25(6):3738–3752, 2017.

[127] Qingjun Xiao, Zhiying Tang, and Shigang Chen. Universal online sketch for tracking heavy hitters and estimating moments of data streams. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 974–983. IEEE, 2020.

[128] Yinglian Xie, Vyas Sekar, David A. Maltz, Michael K. Reiter, and Hui Zhang. Worm Origin Identification Using Random Moonwalks. In *IEEE Security and Privacy (SP)*, 2005.

[129] Hongli Xu, Xiang-Yang Li, Liusheng Huang, Hou Deng, He Huang, and Haibo Wang. Incremental deployment and throughput maximization routing for a hybrid SDN. *IEEE/ACM Transactions on Networking*, 25(3):1861–1875, 2017.

[130] L. Yang and G. Michailidis. Sampled based estimation of network traffic flow characteristics. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2007.

[131] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2584–2593. ACM, 2018.

[132] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[133] Ke Yi and Qin Zhang. Optimal tracking of distributed heavy hitters and quantiles. *Algorithmica*, 65(1):206–223, 2013.

[134] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[135] Shui Yu, Wanlei Zhou, Weijia Jia, Song Guo, Yong Xiang, and Feilong Tang. Discriminating DDoS attacks from flash crowds using flow correlation coefficient. *IEEE Transactions on Parallel and Distributed Systems*, 23(6):1073–1080, 2011.

[136] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Po-

seidon: Mitigating volumetric DDoS attacks with programmable switches. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[137] Z. Zhou and B. Hajek. Per-flow cardinality estimation based on virtual loglog sketching. In *Annual Conference on Information Sciences and Systems (CISS)*, 2019.

[138] Yibo Zhu, Nanxi Kang, Jiaxin Cao, et al. Packet-level telemetry in large datacenter networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2015.