Regis Francisco Teles Martins

Packet Routing Analyses Using Probabilistic Data Structures in Multi-Tenant Networks based on Programmable Devices

Sorocaba, SP

September 2019

Regis Francisco Teles Martins

Packet Routing Analyses Using Probabilistic Data Structures in Multi-Tenant Networks based on Programmable Devices

Masters dissertation submitted to Postgraduate Program in Computer Science of Federal University of São Carlos in partial fulfillment of the requirements for the degree of Master of Computer Science. Research area: Software Engineering and Computer Networking.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Supervisor: Prof. Dr. Fábio Luciano Verdi

Co-supervisor: Prof. Dr. Rodolfo da Silva Villaça

Sorocaba, SP

September 2019

Teles Martins, Regis Francisco
Packet Routing Analyses Using Probabilistic Data Structures in Multi-Tenant Networks based on Programmable Devices/ Regis Francisco Teles Martins. – 2019.
147 f. : 30 cm.

Dissertation (Masters) – Universidade Federal de São Carlos – UFSCar
Centro de Ciências em Gestão e Tecnologia – CCGT
Programa de Pós-Graduação em Ciência da Computação – PPGCC-So, campus Sorocaba, Sorocaba
Supervisor: Prof. Dr. Fábio Luciano Verdi
Co-supervisor: Prof. Dr. Rodolfo da Silva Villaça
Examination board: Prof. Dr. Fábio Luciano Verdi, Prof. Dr. Leobino N. Sampaio, Profa. Dra. Yeda Regina Venturini
Bibliografia

1. Network Monitoring. 2. Sketches. 3. Multi-tenant. I. Prof. Dr. Fábio Luciano Verdi, Prof. Dr. Rodolfo da Silva Villaça. II. Universidade Federal de São Carlos. III. Packet Routing Analyses Using Probabilistic Data Structures in Multi-Tenant Networks based on Programmable Devices.

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Regis Francisco Teles Martins, realizada em 04/09/2019:

Prof. Dr. Fabio Luciano Verdi
UFSCar

Prof. Dr. Leobino Nascimento Sampaio
UFBA

Profa. Dra. Yeda Regina Venturini
UFSCar

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Leobino Nascimento Sampaio e, depois das arguições e deliberações realizadas, o(s) participante(s) à distância está(ao) de acordo com o conteúdo do parecer da banca examinadora redigido neste relatório de defesa.

Prof. Dr. Fabio Luciano Verdi

To my beloved wife Patrícia

To my lovely children Hadassa, Sarah and Samuel

Acknowledgements

I would like to thank,

God, who is the source of life, knowledge, and true wisdom.

My wife for your love and patience during this journey, providing me with unfailing support and continuous encouragement .

My children, who served as my inspiration to pursue this undertaking.

My parents for all motivational words and for the unceasing encouragement, support and attention.

Professor Fábio Verdi and Professor Rodolfo Villaça for the excellent cooperation and hard work reviewing and advising my work, and for the opportunity I was given to conduct my research and further my dissertation at UFSCar.

Professor Luciana Zaina, for advising and steering me in the right direction.

The Sandvine company, for the support.

UFSCar PPGCCS teachers, for the great classes and knowledge transferred.

UFSCar, for providing me with all the necessary facilities for the research.

My colleagues from the LERIS and the University, especially Johannes Von Lochter, for all the guidance.

One and all, who directly or indirectly, have lent their hand in this venture.

“For I am persuaded, that neither death, nor life, nor angels, nor principalities, nor powers, nor things present, nor things to come, nor height, nor depth, nor any other creature, shall be able to separate us from the love of God, which is in Christ Jesus our Lord.”

(Romans 8.38-39)

“Whether you think you can or think you can’t, you’re right.”

(Henry Ford)

Abstract

Given the network traffic growth, due to applications that heavily use computational cloud infrastructure, the need for improving the monitoring traffic techniques has increased. For traffic engineering, it is essential to gain total visibility into the traffic flowing across the network. The most used methods for traffic monitoring in the industry are those based on dedicated monitoring protocols as SNMP (*Simple Network Management Protocol*), NetFlow, SFlow, among others.

With processing capacity evolution of forwarding devices, new techniques have been proposed. The use of sketches has become widely popular for traffic monitoring tasks. Sketches are compact data structures capable of summarizing and store information about the state of packets. Using sketches, it is possible to monitor a network traffic, understanding the path travelled by each packet and which devices were responsible for the packet forwarding.

Analyzing traffic over the network is a challenge that changes the traditional monitoring approach. The current performance indicator metrics provided by network devices are not enough to analyze and create insights for the network traffic as a whole. We need a way to produce key performance indicators that can be correlated across different network devices on the same network. This new approach opens opportunities for researching and developing novel techniques to obtain a holistic network traffic visibility, to support decisions in traffic engineering, to detect traffic anomalies and other applications.

Using a single sketch named BitMatrix, proposed in this work, it is possible to monitor network traffic, understand the path travelled per packet and which devices forwarded this packet along its path. In this context, this probabilistic structure was adopted to identify the path used to forward a packet in a multi-tenant network in two different scenarios: a) in an emulated network, using P4 routers and, b) in a simulated network, processing real traffic traces, using a Python framework. As a result, overloaded routers, links and paths and heavy user tenants were identified.

Key-words: Network Monitoring, Sketches, Bitmaps, Multi-tenant, P4 Language, Programmable Network.

Resumo

Com o crescimento do tráfego de rede, devido às aplicações que utilizam serviços de infraestrutura computacional em nuvem, aumentou a necessidade de se otimizar a utilização dos dispositivos responsáveis pelo encaminhamento e monitoramento do tráfego. Para engenharia de tráfego, é essencial ter visibilidade do tráfego que está sendo encaminhado. Os métodos mais comuns para monitoramento de tráfego são baseados em protocolos dedicados a esse fim, como o SNMP (*Simple Network Management Protocol*), NetFlow, SFlow, entre outros.

Com a evolução da capacidade de processamento dos equipamentos de encaminhamento, novas técnicas têm sido propostas. Dentre estas, o uso de *sketches* tem se tornado cada vez mais comum. Os *sketches* são estruturas de dados compactas, capazes de armazenar de forma sumarizada informação sobre o estado de um pacote. Com o uso de *sketches* é possível monitorar o tráfego de uma rede, entendendo o caminho percorrido por cada pacote e quais dispositivos foram responsáveis pelo seu encaminhamento.

A análise do tráfego através da rede é um desafio científico que redireciona a forma tradicional de monitoramento, movendo o foco dos elementos de rede e seus contadores para o tráfego em si. Esse redirecionamento de foco cria oportunidades para pesquisa e desenvolvimento de novas formas de se obter visibilidade do tráfego para auxiliar nas tomadas de decisão em engenharia de tráfego, detecção de anomalias e outras aplicações.

Fazendo uso do *sketch* denominado BitMatrix, proposto neste trabalho, foi posssivel monitorar o trafico de rede, entendendo o caminho percorrido por cada pacote e o quais dispositivos de rede foram responsaveis pelo encaminhamento desse pacote atraves da rede. Nesse contexto, esta estrutura probabilistica foi adotada para identificar a rota usada para o encaminhamento do pacote em uma rede *multi-tenant* em dois diferentes scenarios: a) em uma rede emulada, utilizando roteadores P4 (*Programming Protocol-Independent Packet Processors*) e, b) Em uma rede simulada, processando capturas de trafico reais, utilizando Python. Como resultado, tornouu-se possível identificar quais rotas e elementos estão sobrecarregados e quais os usuários (*tenants*) com maiores demandas.

Palavras-chaves: Network Monitoring, Sketches, Bitmaps, Multi-tenant, Linguagem P4, Redes Programaveis.

List of Figures

Figure 1 – Example of a traditional distributed network model, showing network devices executing application function, control and data planes. 39
Figure 2 – Example of a SDN network. 40
Figure 3 – P4 abstraction model. 40
Figure 4 – Simple Switch target architecture. 43
Figure 5 – The BitMatrix is a group of n bitmap vectors. 47
Figure 6 – The proposed framework architecture for the project, showing the interaction between its components. 51
Figure 7 – The phases for collect, parse and store the BitMatrix information. . . . 56
Figure 8 – The network diagram proposed to illustrate the analytics possibilities. . 57
Figure 9 – An example of a traffic matrix. 59
Figure 10 – Proposed topology to exemplify the sketches deployment in a P4 programmable network device. 61
Figure 11 – Parse graph for the P4 deployment. 63
Figure 12 – Top-level control graph, generated for the ingress control block, of the P4 BitMatrix code. 74
Figure 13 – Top-level control graph, generated for the egress control block, of the P4 BitMatrix code. 74
Figure 14 – BitMatrix structure and its bitmaps. 78
Figure 15 – Mininet emulated network topology with P4-enabled forwarding. 79
Figure 16 – Relation between % of collision X % of occupation. 82
Figure 17 – BitMatrix Occupancy and Collisions for tenant A in P4 switch 1. . . . 83
Figure 18 – Amount of packets per tenant in P4 switch 1. 84
Figure 19 – Total amount of packets per P4 switch. 84
Figure 20 – Amount of bytes per tenant in P4 switch 1. 85
Figure 21 – Total amount of bytes per P4 switch. 85
Figure 22 – Amount of packets on path AB+BA. 86
Figure 23 – Amount of bytes on paths AB+BA. 86
Figure 24 – Time for BitMatrix collection via Thrift interface - using the command bm_register_read 88
Figure 25 – Time for BitMatrix collection via Thrift interface - using the command bm_register_read_all 88
Figure 26 – Average hash collisions per hash algorithm broken down by source of the trace and set of fields selection. 93
Figure 27 – Average hash collisions per hash algorithm 94
Figure 28 – Average hash collisions set of fields selection. 94

Figure 29 – New T3 Backbone Service for NSFNET 1992 95
Figure 30 – Topology detail for tenants, routers and links. 96
Figure 31 – Average throughput per link on every 10 seconds. 98
Figure 32 – Packets per second average rate, per router, on every 10 seconds. . . . 99
Figure 33 – Packets per second average rate on router 12, per tenant, on every 10 seconds. 99
Figure 34 – Packets per second average rate per tenant in the network, on every 10 seconds. 100
Figure 35 – Dashboard showing the traffic contribution for each tenant per router. . 100
Figure 36 – Number of processed packets measured by the BitMatrix counter and the packet counter for Router 2 and Tenant 11, on every 10 seconds. . . 102
Figure 37 – Number of processed packets measured by the BitMatrix counter and the packet counter for Router 1 and Tenant 1, on every 10 seconds.. . . 102
Figure 38 – Number of processed packets measured by the BitMatrix counter and the packet counter for Router 3 and Tenant 3, on every 10 seconds. . . 103
Figure 39 – Percentage of bitmap occupation versus the percentage of collisions, per bitmap. 103
Figure 40 – Number of packets processed measured by the BitMatrix counter, packet counter and the BitMatrix adjusted, for Router 2 and Tenant 11, on every 10 seconds. 106
Figure 41 – Number of packets processed measured by the BitMatrix counter, packet counter and the BitMatrix adjusted, for Router 1 and Tenant 1, on every 10 seconds. 107
Figure 42 – Number of packets processed measured by the BitMatrix counter, packet counter and the BitMatrix adjusted, for Router 3 and Tenant 3, on every 10 seconds. 107

List of Tables

Table 1 – Main works in the area 37
Table 2 – Data Dictionaty 53
Table 3 – Primitive Actions 68
Table 4 – Traffic path between tenants 79
Table 5 – Different hash algorithm and set of fields used to calculate the packet position in the corresponding bitmap, for each packet. 93
Table 6 – Tenants and theirs assigned group of network prefixes. 97
Table 7 – Database partitioning for k-fold cross validation 104
Table 8 – Average MSE (mean squared error) and Average STDDEV (standard deviation) for the test database, per method used as hypotheses. 104
Table 9 – Path used for traffic between tenants 147

List of Algorithms

1 Bitmap vector packet store algorithm 46
2 BitMatrix packet store algorithm 48

Listings

5.1 Partial P4_14 code defining headers 61
5.2 Partial P4_14 code defining parser 63
5.3 Partial P4_14 code defining field list definition 64
5.4 Partial P4_14 code defining metadata 65
5.5 Partial P4_14 code calculation process 66
5.6 Partial P4_14 code register definition 67
5.7 Partial P4_14 code actions 69
5.8 Partial P4_14 code tables definitions 70
5.9 Partial P4_14 code tables definitions 70
5.10 Partial P4_14 code tables definitions 71
5.11 Partial P4_14 code counter definition 72
5.12 Partial P4_14 code control 73
5.13 Commands for the run-time process of table population, in the network device 75
6.1 Partial P4_14 code defining Header and Parser 80
6.2 Partial P4 code defining input fields for hashing and the hash algorithm used. 81
6.3 `bm_register_read` commnad definition in standard.thrift code for the behavioral-model 87
6.4 `bm_register_read_all` command definition in standard.thrift code for the behavioral-model 87
6.5 Function definition in Python for Checksum 16 hash calculation 89
6.6 Using crcmod Python library to compute the CRC hash code. 90
6.7 Using MD5 - message-digest algorithm Python module to obtain the MD5 hash code. 91
A.1 P4_14 code implementing BitMatrix 119
A.2 P4_14 code for tables feed via Thrift runtime interface 125
A.3 Mininet enviromnet defined by topo.py 125
A.4 Collector and Controller component implemented in Python 129
A.5 Python framework for NSF 92 network simulation 139

List of abbreviations and acronyms

ADDR	Address
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
BMv2	Behavioral Model version 2
CAIDA	Center for Applied Internet Data Analysis
CLI	Command Line Interface
CPU	Central Processing Unit
CRC16	Cyclic Redundancy Check 16 bit
CRC32	Cyclic Redundancy Check 32 bit
DF	Degrees of Freedom
DSCP	Differentiated Services Code Point
DST	Destination
ECN	Explicit Congestion Notification
ETL	Extract Transform and Load
ForCES	Forwarding and Control Element Separation
FPGA	Field Programmable Gate Arrays
IDS	Intrusion Detection System
ICMP	Internet Control Message Protocol
IHL	Internet Header Length
IP	Internet Protocol
ISP	Internet Service Provider
JSON	JavaScript Object Notation

KPI	Key Performace Indicator
MAC	Media Access Control
MAPE	Mean Absolute Percentage Error
Mbps	Megabits per second
MD5	Message Digest algorithm 5
MSE	Mean Squared Error
NETRESEC	Network Forensics and Network Security Monitoring
NPU	Network Processor Unit
NSFNET	National Science Foundation Network
OLAP	Online Analytical Processing
OS	Operational System
OVS	Open Virtual Switch
P4	Programming Protocol-Independent Packet Processor
PISA	Protocol-Independent Switch Architecture
POC	Proof of Concept
POF	Protocol-oblivious Forwarding
PPS	Packets per second
RPC	Remote Procedure Call
RSA	Rivest Shamir and Adelman
SCREAM	Sketch Resource Allocation for Software-defined Measurement
SDN	Software Defined Network
sFlow	Sampled Flow
SNMP	Simple Network Management Protocol
SRC	Source
SSE	Sum Squared Error
TCP	Transmission Control Protocol

TTL	Time To Live
UDP	User Datagram Protocol
UnivMon	Universal Monitoring

List of symbols

Σ	Greek letter Sigma

Contents

1 INTRODUCTION **31**

2 TECHNICAL BACKGROUND **35**
2.1 Related Work **35**
2.2 Sketches **36**
2.3 Software Defined Networks **38**
2.4 P4 Language **40**
2.4.1 BMv2 - Behavioural Model version 2 41
2.5 Mininet **42**

3 BITMATRIX **45**
3.1 Architecture **45**
3.1.1 Muti-tenant Environment 46
3.2 Monitoring Architecture **47**
3.3 Caveats and Limitations **49**

4 MONITORING FRAMEWORK **51**
4.1 Framework Architecture **51**
4.2 Network Traffic Monitoring per Tenant **53**
4.2.1 Network Device Packet Processing 53
4.2.2 Collection and Control 54
4.2.3 Parse and Storage 55
4.2.4 Analyses 56
4.3 Network Traffic Matrix **59**

5 BITMATRIX IN P4 LANGUAGE **61**

6 TESTS AND RESULTS **77**
6.1 P4 implementation using Mininet **77**
6.1.1 BitMatrix Tests and Methodology 77
6.1.2 Parsing the Packet Header 80
6.1.3 Hashing and Hash Inputs 81
6.1.4 Collisions versus Occupation Tests 82
6.1.5 Retrieving and Processing BitMatrix 82
6.1.6 Results 83
6.1.7 Contribution - New Command bm_register_read_all added to Thrift interface 87
6.2 Python Implementation **88**

6.2.1 Hash Algorithms and Packet headers Selection 89
6.3 NSF92 Framework in Python 94
6.3.1 Results 98
6.4 Machine Learning for BitMatrix Measurement Adjustment 101

7 CONCLUDING REMARKS 109

Bibliography 113

APPENDIX A – APPENDIX 119
A.1 Deployment in P4 example 119
A.2 Mininet modifications for P4 network emulation 125
A.3 Collector and Controller component implemented in Python 129
A.4 NSF 92 Python framework 138
A.5 Routing table for NSF 92 Python framework 145

1 Introduction

The exponential growth of network traffic emphasizes the importance of network monitoring, management, planning and traffic engineering. In addition, cloud computing have been widely used, and the number of cloud-based services has increased rapidly in the last years, increasing the complexity of the infrastructure behind these services. In this way, data centers need to have an accurate and fine-grained monitoring to operate efficiently, considering the use of shared resources by multiple tenants. Consequently, cloud administrators could use monitoring information for essential processes such as accounting, audit tracking, debugging, fault detection, job scheduling and performance analysis.

Faced with the need for more effective ways to monitor and understand how network traffic is handled and what changes should be made on traffic routing, a valuable contribution is made with the utilization of compact data structures (sketches). These data structures can be implemented on P4 enabled devices, to monitor network traffic.

In this context, sketches are probabilistic data structures used to store summarized information about the network traffic. The two primary advantages of using sketches are; the low memory usage in the device and the adjustable accuracy achieved regarding the amount of memory used to store information. There are several applications in measurement tasks for sketches, such as heavy-hitter detection (MATHEW; KATKAR, 2011; YU; JOSE; MIAO, 2013; KRISHNAMURTHY et al., 2003), traffic pattern change detection (KRISHNAMURTHY et al., 2003; SCHWELLER et al., 2004), flow-size distribution estimation (DUFFIELD; LUND; THORUP, 2005), and others.

As an evolution of Software Defined Networking (SDN), P4 (Programming Protocol-Independent Packet Processor) was presented as a high-level programming language for packet forwarding devices (BOSSHART et al., 2014). The P4 language makes programming devices (targets) more flexible. To use a program written in P4, it needs to be compiled to the target device, which can be a hardware or software-based system and the program can be simple or very complex, depending on the behaviour planned for the device when forwarding packets. P4 allows the creation of several mechanisms for traffic measurement (KIM et al., 2015; SIVARAMAN et al., 2017), among them, the implementation of probabilistic data structures, or sketches.

So, the main goal of this work is to propose and implement a framework, based on the well-known bitmap and counter-array sketches, that goes further than the traditional monitoring process of counting packets and bytes. A new compact probabilistic structure, named BitMatrix, is presented. BitMatrix was created to support the multi-tenant monitoring using an array of bitmaps, combined with a storage method that enables the

information to be segmented per tenant, in a later phase. Besides the general statistics, it enables detail analyses on a packet level in the network, i.e. it is possible, using the BitMatrix, to determine how many packets travelled through a specific set of network devices. Thus, we demonstrate how to produce critical and frequent monitoring information such as the number of bytes and packets per network device per tenant, the number of packets per path per tenant and also the paths being used by each tenant.

Using bitstreams collected from bitmaps and counter arrays at pre-determined periods, sufficient information is obtained to understand the volume of traffic being processed in a given network. The data collecting from data structures must happen in the same time; thus, they will correspond to the same period in all elements. This time frame is called epoch. After collecting bitstreams, data structures are restarted to proceed with collecting information for the next epoch. Once the information from the data structures is collected and stored, it is possible to process this information from two or more elements, to obtain indicators about forwarding traffic in the network.

Performance indicators such as data volume and the number of packets processed by each network device and the path that a given packet has taken through the network can be obtained and used for decision making in the planning, expansion and traffic engineering process.

Objectives

The main objective of this project is to create a sketches-based framework capable of generating tenant segmented data streams for storing packets and bytes across network devices, enabling several traffic analyses. This framework consists of three elements that will digest packets and temporarily store the information in the memory of the data plane element, collect the information from memory, parse and store it in a database and present statistics and key performance indicators based on that information.

The main goals for this project are:

- Creation of a bitmap array using P4 language for traffic segmentation. In this project, segmentation will be by tenant and may be extended to other types of segmentation;

- Perform a near-to-synchronous collection of data streams to estimate the number of packets and bytes processed in network devices or, for a given path in the network;

- Determine the number of packets and bytes that traveled through a given route, allowing segmentation by source tenant.

Outline

The remaining of this dissertation is organized as follows:

- Chapter 2: presents related works and technical background that are the basis for this work;
- Chapter 3: details the BitMatrix architecture, features and operation.
- Chapter 4: explain the implementation of a framework using the BitMatrix to store the packets and other elements to collect, parse, store a and a analyze the information obtained. Also, an introduction about how to create a network traffic matrix, using BitMatrix information.
- Chapter 5: walks through a deployment example in P4 Language, explaining the P4 structure and primitives used to implement it.
- Chapter 6: describes the testing methodology and its respective results, detailing two frameworks, one for emulation, using Mininet and another for simulation using Python language, developed to validate the BitMatrix and all its elements. Also presents a discussion about the impact of using different hash algorithms with different inputs and finally, the result of using machine learning, to correct the error introduced by the process;
- Conclusion: contains our final remarks and future works that can be developed based on this work.

2 Technical Background

This chapter presents the related works for this project and gives an introduction about the enabling technologies used in this work.

2.1 Related Work

Monitoring network traffic is the method used to understand issues or problems within a network environment. In order to understand, prevent and resolve issues, there are numerous methods available for traffic monitoring. The study and research of monitoring network traffic evolved due to the increase in traffic over the years. By monitoring, we can extract relevant information for performing network management tasks such as attack and anomaly detection (MATHEW; KATKAR, 2011; ZHANG, 2013), forensic analysis (XIE et al., 2005) and engineering of traffic (BENSON et al., 2011; FELDMANN et al., 2001).

Metrics at different levels are required for each management task, such as flow size distribution (KUMAR et al., 2004), heavy hitters (BENSON et al., 2011), or changes in the pattern of traffic detection (SCHWELLER et al., 2004), according to what we want to achieve. At a high level, there are two classes of techniques for obtaining performance metrics.

The first is traffic measurement using counters. The most widely used monitoring protocol for this purpose is the Simple Network Management Protocol (SNMP) (CASE J. D.; DAVID, 1990). SNMP collects real-time performance indicator values and maintains an in-memory database with this information available for queries, furthermore, it can send alarms when an indicator threshold is reached.

The second class of techniques involves two approaches to estimate the traffic metrics. The first approach is based on generic flow monitoring, typically using some protocol to collect traffic samples to estimate these metrics. Among the most popular protocols are NetFlow (CLAISE, 2004) and SFlow [1]. Although generic flow monitoring is sufficient to monitor traffic in general, previous work has shown the low accuracy of this technique in generating more granular metrics (DUFFIELD; LUND; THORUP, 2005; ESTAN; VARGHESE, 2001; RAMACHANDRAN et al., 2008). The limitations of sampling methods have led to an alternative sketching approach, where real-time algorithms and compact data structures are designed to generate specific metrics (BRAVERMAN et al., 2015; ESTAN; VARGHESE, 2001; RAMACHANDRAN et al., 2008; KRISHNAMURTHY

[1] sFlow is a sampling technology for monitoring network traffic. - Available at: http://www.sflow.org/ - Accessed 08/06/2019.

et al., 2003; LALL et al., 2006; YU; JOSE; MIAO, 2013).

While research in sketching and data streaming areas make a significant contribution, in the long run, it is impossible to sustain the continued creation of dedicated purpose algorithms. The need to support new metrics demands the development of new algorithms, as well as hardware and languages that support them. Tools like OpenSketch (YU; JOSE; MIAO, 2013) and SCREAM (MOSHREF et al., 2015) provide libraries that reduce implementation effort and provide efficient resource allocation. However, they do not address the fundamental need to create new sketches for each task. Also, the fact that sketches are implemented on demand according to the set of metrics we want to monitor creates blind spots on the metrics not monitored. Recently, UnivMon (LIU et al., 2016) presented a proposal for a framework that reconciles generality and high fidelity for a broad spectrum of monitoring tasks, and the FlexSketchMon that introduces a novel data plane architecture for collecting traffic flow statistics and provide flow aggregations for monitoring applications (WELLEM et al., 2019). The work presented here focused on a lightweight sketches structure, named BitMatrix, able to cover a wide range of metrics generation, using the intelligence provided by the control plane.

From the environment perspective, monitoring sketches can be implemented in software defined networks (SDN), where the responsible processing and population are the controllers (HUANG et al., 2017; SHAHBAZ et al., 2016), as well as through the use of packet-oriented languages such as P4 (BOSSHART et al., 2014). There are enough implementations with complex algorithms (HUANG et al., 2017; SIVARAMAN et al., 2015; KIM et al., 2015; DANG et al., 2016), in addition to UnivMon, which is the first published work in monitoring using sketches and implemented in P4 language. Table 1 showns the main works related to this project.

2.2 Sketches

Sketch is the most common name existing in literature for describing data structure. By definition, sketches are a compact data structure used in streaming algorithms to store and summarize traffic statistics (DIMITROPOULOS; HURLEY; KIND, 2008). Different from the traditional traffic measurement techniques, as Netflow (CLAISE, 2004) and Sflow (SFLOW-RT, 2019), this data structure offers a fine-grained measurement not only sampling packets from traffic but processing every packet forwarded in the network, making very effective usage of the memory and with provable tradeoffs of memory and accuracy (YU; JOSE; MIAO, 2013; CORMODE; MUTHUKRISHNAN, 2005; ESTAN; VARGHESE, 2003; HUANG; LEE, 2015; MITZENMACHER; PAGH; PHAM, 2014; SCHWELLER et al., 2004).

There are restrictions when implementing sketches in traditional devices, as vendors

Work	Citation	Purpose	Methodology
MicroTE	(MATHEW; KATKAR, 2011)	Traffic enginnering ,Heavy Hitters	OpenFlow
OpenWatch	(ZHANG, 2013)	Anomaly detection	OpenFlow
Lossy Data Structure	(KUMAR et al., 2004)	Traffic engineering	Sketchs, trace analyses
Reversible Sketches	(SCHWELLER et al., 2004)	Pattern of traffic detection	Sketchs, trace analyses
Estimating Flow Distri.	(DUFFIELD; LUND; THORUP, 2005)	Flow monitoring	Statistical inference, trace ana.
FlexSample	(RAMACHANDRAN et al., 2008)	Anomaly detection, traffic mon.	Sketchs, trace analyses
Sketch-based Chg Detect.	(KRISHNAMURTHY et al., 2003)	Anomaly detection	Sketchs, trace analyses
OpenSketch	(YU; JOSE; MIAO, 2013)	Anomaly detection, traffic mon.	NetFPGA, sketchs,trace ana.
Scream	(MOSHREF et al., 2015)	Traffic engineering	Sketchs, trace analyses
UnivMon	(LIU et al., 2016)	Flow monitoring	P4, sketchs, trace analyses
FlexSketchMon	(WELLEM et al., 2019)	Flow monitoring	Sketchs, traffic mon., NetFPGA

Table 1 – Main works in the area

would need to re-engineer their programmable ASIC to deploy a specific sketch function, for this reason, sketches were not widely deployed in the past. With the advent of programmable devices and soft-switches, many sketch-based solutions began to emerge, especially given the low overhead in the packet processing pipeline and the low memory usage, when compared to flow-based counters (HUANG; LEE, 2015). However, contrary the common sense, analyses from HUANG et al., shows that a sketch-based solution in software can consume substantial computing resources, causing overload in the CPU, given that sketches are only primitives and often requires additional extensions and components incurring in a heavy load in CPU, especially during traffic bursts in the data plane.

Some examples of measurement solutions implemented using sketches are heavy hitter detection (BANDI et al., 2007), traffic change detection (SCHWELLER et al., 2004), flow size distribution estimation (KUMAR et al., 2004), global iceberg detection (Guanyao Huang et al., 2009), and fine-grained delay measurement (SANJUàS-CUXART et al., 2011). Furthermore, sketches have many applications in networking problems, specifically, in estimating the flow-size distribution of traffic streams (KUMAR et al., 2004; ZHANG et al., 2004), in identifying anomalies (KRISHNAMURTHY et al., 2003; SCHWELLER et al., 2004; LI et al., 2006).

Sampling techniques to reduce the necessary memory resources and the processing overhead were addressed by GIBBONS; MATIAS, DEMAINE; LóPEZ-ORTIZ; MUNRO, and KAMIYAMA; MORI, for identifying heavy hitters. As a trade, sampling techniques typically present lower accuracy. Moreover, several other studies for related problems, such as data streaming algorithms for finding sources and destinations, communicating with other distinct destinations or sources (ZHAO et al., 2005), and algorithms for the distributed version of the top-k problem, to minimize communication overhead between vantage points for finding the globally k most frequent elements (BABCOCK; OLSTON, 2003).

In this work, we used a simple data structure, named BitMatrix to store digested packets using a unique bitmap, and its bytes using a counter array, segmented by a tenant. This storing process results in several datastreams, that are processed in background, to produce several network traffic statistics.

2.3 Software Defined Networks

In a traditional network, devices operate independently, holding management and configuration functions on themselves. Each device is built to perform a specific function in the network such as packet switching, routing and deep packet inspection. They process packets according to instructions configured into the device or received through the network, via a dedicated protocol or distributed algorithm. Indeed, network devices process packets

based on information from many sources with different mechanisms, in an autonomous way. The control information and the hardware and software elements responsible for its exchange constitutes the control plane in the network. The data plane is the set of hardware and software elements in the path of data packets. In the distributed network model the control plane and the data plan are independent for each device and are composed of proprietary hardware and software elements, designed to a specific function in the network, as shown in Figure 1. As each vendor has different concepts and different operational systems, configuring, debugging and troubleshooting those devices can be a challenge for network operators. Moreover, as the devices are task-oriented, the ability to change their packet processing behaviour, for implementing new solutions, is limited.

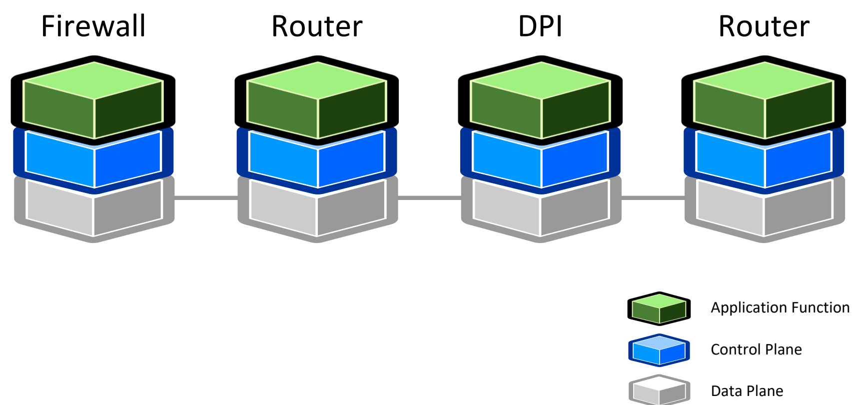

Figure 1 – Example of a traditional distributed network model, showing network devices executing application function, control and data planes.

The SDN paradigm creates a separation between the control plane and data plane functions in a network devices' software and hardware perspective. Figure 2 presents the SDN network model. In the data plane, resides only packet processing abstractions, which is used by the control plan to define the packet processing behaviour for the data plane devices. Data plane, in SDN terminology are often referred as forwarding devices and are enabled with SDN technology like OpenFlow (MCKEOWN et al., 2008), ForCES (Forwarding and Control Element Separation) (HALPERN et al., 2010), POF (Protocol-oblivious Forwarding) (SONG, 2013) and P4 (BOSSHART et al., 2014).

A centralized control plane offers control and management for all data plane devices in the network. The element that provides a logically centralized control plane is the SDN controller. It provides control APIs to connect to network devices and control their packet processing behaviour, using the configuration and management protocols mentioned above. Those protocols provide standard open interfaces to manage data plane network devices. This control and data plane separation model make the network data plane devices open to change their packet processing behaviour even after deployment.

P4 language extends the SDN concepts by enabling network devices not only to

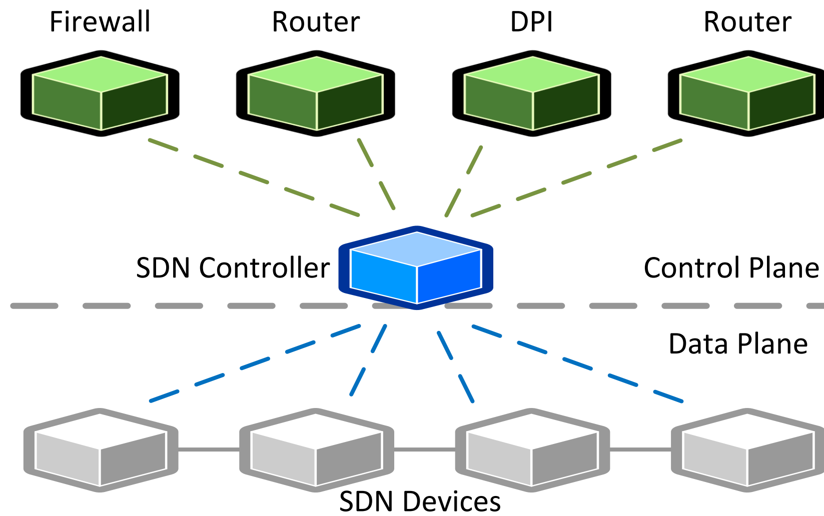

Figure 2 – Example of a SDN network.

have their packet processing behaviour controlled, but also to program them. It allows to parse the packet to obtain required fields, process the parsed fields and deparse the packet using the processed fields.

2.4 P4 Language

P4 is a high-level language used for expressing how a hardware or software network device, such as network interface card, switch or any other network function appliance, will process packets on its pipeline. It is based on a model divided into ingress and egress pipelines, which consists of a parser, a set of match+action tables and a deparser. The parser will create fields based on headers' definitions for each incoming packet. Tables perform a lookup using a set of header fields for a match and apply corresponding actions, expressed on each table. Figure 3 diagram shows the P4 model.

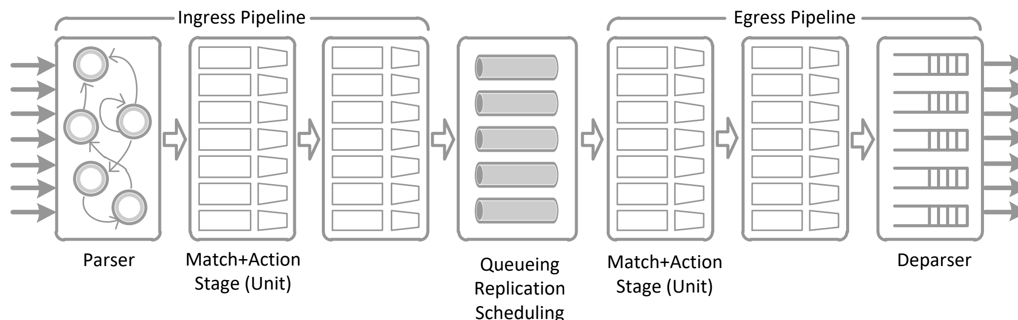

Figure 3 – P4 abstraction model.

P4 is a protocol-independent language and allows expressing any forwarding plane protocol, specifying the following for each element:

- **Header and Parser:** Headers will describe the layout of fields in a packet, providing names to be referenced later in the code. Fields in a header are defined in a structure describing fields belonging to the header and their length, in bits. The parser will use this definition structure to create a header instance parsing each header field in the packet sequentially, according to the number of bits in header definitions for each field. Headers can also define metadata instances, which are registers maintained during the packet processing mainly for controlling purposes, as holding the port number that the egress pipeline should use to forward the packet processed to the next hop.
- **Match and Action:** These tables contain rules defining the exact set of fields that should be examined to find a match on the header fields that were extracted from the parser and execute user-defined actions. In case of no matches are found, the default action for the table will be applied. Actions can create or modify header fields, select output ports or drop the packet. Tables are empty when the target starts and the population happens by using a target specific API at runtime. The P4 specification does not define any specific API to be used for this task, leaving it open to vendors to define.
- **Pipelines and Queues:** The match and action process will compose the ingress and egress pipelines and will generate an egress specification to determine the port that each packet will be sent, creating a queue. This egress queue may buffer packets when an output port is overloaded. The metadata `egress_spec` is used to specify the port destination of a packet. This metadata indicates the queue to be used with each destination and may represent a physical or logical port, or even a multicast group. After all processing in the egress pipeline, the packet instance's header is deparsed and the resulting packet is transmitted.

Devices that are able to compile P4 programs are called targets. The target will provide a framework for implementation, a P4 compiler and an API for managing the behaviour of data plane objects from the control plane. In this work, we used the Behavioral Model version 2 (BMv2) target, which uses the Thrift API for runtime interactions.

2.4.1 BMv2 - Behavioural Model version 2

The Behavioural Model version 2 (BMv2) is a framework, developed in C++, allowing P4 programming, and acting as a software switch. The project is committed to implementing full support for P4 specifications and being architectural independent. Three

different targets can be implemented using this framework: `simple_router`, `l2_switch` and `simple_switch`. In this work, we used the `simple_switch`, which is widely used to test and deploy different P4 features for most users. This target is equivalent to the abstract switch model described in the P4 specifications (The P4 Language Consortium, 2017).

The main components for the BMv2 framework are the following:

- **P4 Compiler:** There are two P4 compilers available for BMv2: the p4c-bm, which is the legacy compiler, supporting only sources written in P4_14 language version and the p4c. The p4c is the recommended compiler to be used, and it offers support for P4 version 14 and 16 programs.

- **Runtime Interface:** Each BMv2 instance runs a Thrift RPC server that the command line interface `runtime_CLI.py` can access to populate tables, read and write information on counters, meters and registers. The CLI uses Python's cmd command and supports auto-completion. This interface is used in this work to collect the BitMatrix information from the BMv2 switch.

- **Debbuger:** The BMv2 also has a debugger component that enables event logging, using the Python nanolog library. The debug feature needs to be enabled when compiling the BMv2 and can be used to display event of significance, such as table hits/misses and parse transitions, for each packet.

The BMv2 workflow is simplistic. The P4 program is compiled to a JSON representation by the p4c compiler. The BMv2 loads this JSON file and initialize its data structures resulting in the desired switching behaviour. Figure 4 shows the `simple_switch` architecture. The ingress_thread is responsible for receiving the packet from the switch ports and executing the specified ingress pipeline. Then, it forwards the packets to an egress thread, which is responsible for the egress pipeline. Finally, the packet is delivered to the transmit thread and transmitted to the output port.

2.5 Mininet

Mininet is a Python-based framework that allows creating a complete emulated network of switches, Openflow switches, controllers, hosts and links in a virtual environment. Mininet uses process abstraction to create virtualization to run network elements on a single machine, using its OS kernel. Using Linux virtualization features, it provides exclusive processes with isolated network interfaces, routing and ARP tables. Linux traffic control can limit the bandwidth for links to manage the traffic to a specific rate. Each emulated

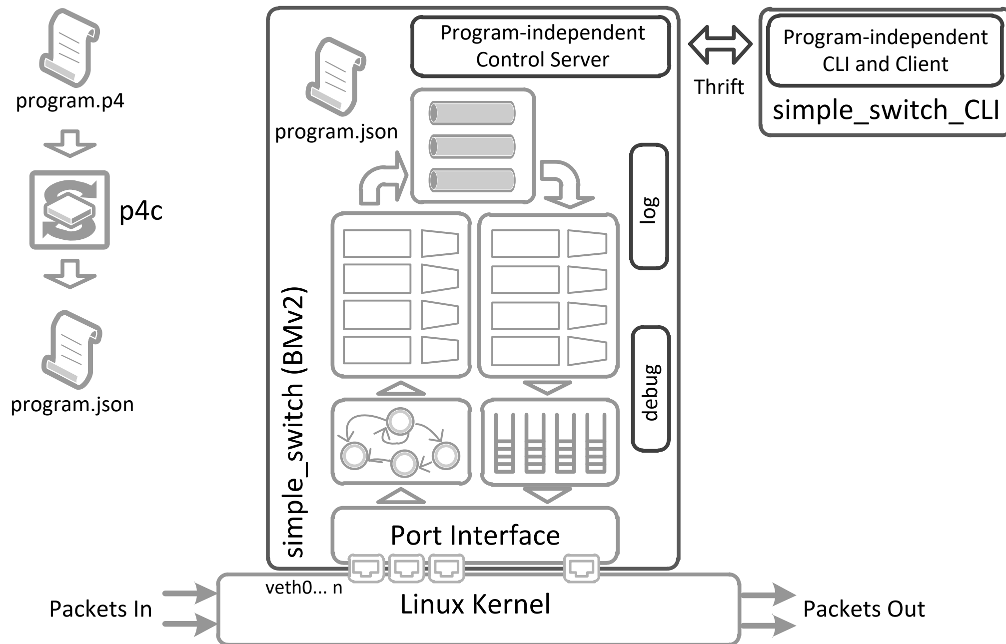

Figure 4 – Simple Switch target architecture.

host can have one or more exclusive network interfaces. The Linux Bridge or the Open vSwitch (PFAFF et al., 2015) is used to switch packets across host and switches interfaces.

Mininet is mostly used to develop experiments with OpenFlow and SDN systems. In this work, we used a modified version of Mininet to support the BMv2 soft switch for tests. This version was initially created to be used in P4 tutorials, and we adapted it to run the environment for our experiments. The Appendix A.2 Mininet modifications for P4 network emulation shows the modifications done in the original code.

3 BitMatrix

In this chapter, we describe in detail the BitMatrix paradigm, exposing its conception, implementation and statistics that can be generated from information provided from it.

3.1 Architecture

A bitmap vector, in this context, is a sequence of bits with a fixed length, as shown in Definition 3.1. Thus, each bit position will create an index for the bitmap, making it possible to refer to a specific position by using this index. The bitmap vector was used to store each packet processed by a network device. To determine which index position a packet should occupy, we use a hash function.

$$bitmap = \{bit_1, bit_2, bit_3, \ldots, bit_n\} \tag{3.1}$$

The packet's fields, used as input for the hash algorithm, must be able to represent the packet uniquely and may not change across hops during the forwarding process. The hash will be calculated using the invariant portion of the packet and the first 8 bytes of the payload - TCP, UDP or another subsequent layer - if present. IP fields modified during the packet forwarding across the network are not used as input as this would result in a different hash for the same packet along its path. According to Snoeren, the first 28 invariant bytes of a packet, are sufficient to differentiate almost all non-identical packets (SNOEREN et al., 2001).

With each packet represented by a hash, the bitmap vector was used to store packets, by setting the index bit corresponding to the hash to one. Initially, the bitmap has all positions set to zero, and then, for each packet processed, one position, defined by the hash algorithm, is set to one. In this way, every time a packet is processed, it will generate a hash number and will change the value of the bit in the position corresponding to the hash value to 1. In case of the bitmap vector has fewer positions than the hash value, a modulo function needs to be used to adjust the corresponding index to that packet.

Algorithm 1 demonstrates the procedure used to process a packet, generate a hash, and determine the corresponding position for that packet in the bitmap vector.

Algorithm 1 Bitmap vector packet store algorithm

1: **procedure** BITMAP VECTOR POPULATION
   Input: First 28 invariant bytes of the packet
   Output: Corresponding bitmap vector position set to 1
2: $hash_value = hash_function(i)$
3: **if** $hash_value > bitmap_length$ **then**
4: $bitmap_position = \mod(bitmap_length, hash_value)$
5: **else**
6: $bitmap_position = hash_value$
7: **end if**
8: $bitmap[bitmap_position] = 1$
9: **end procedure**

Furthermore, an counter array sketch is used to store the total length of the packet processed. It will use the same hash code calculated for the bitmap, to store the packet length in the counter array. In this way, it is possible to recover the total length of the packet by using the index from the bitmap sketch.

3.1.1 Muti-tenant Environment

The use of the bitmap vector, however, is limited to store packets indistinctly and does not allow any segmentation, as it is composed of a single vector. In a multi-tenant environment, it may be desired to use one bitmap vector to store packets for each tenant using the network. Instead of using several vectors individually, we proposed a bitmap matrix, that we called BitMatrix, and it is represented in Definition 3.2.

$$BitMatrix = \{bitmap_1, bitmap_2, bitmap_3, \ldots, bitmap_n\} \quad (3.2)$$

The BitMatrix is a group of bitmap vectors, used to store packets in a segmented manner, as shown in Figure 5. The main benefit of this method is that it is still possible to segment the packet storage using a single probabilistic structure. This segmentation can happen in different ways, such as per sub-net, per layer three protocol, or layer four port, although for a multi-tenant network, we are using a per tenant segmentation approach, where every packet originated by a specific tenant, identified by the packet source IP address, will be stored in the same bitmap vector in the BitMatrix.

The number of bitmaps in the BitMatrix will depend on the number of tenants monitored in the network. The number of tenants will determine how many bits the BitMatrix will use in each position, as it will use one bit per bitmap per position. From another perspective, a BitMatrix storing several bits on a position will create an integer number that represents the position bits. e.g., Consider that a BitMatrix is composed of four bitmaps. Consequently, it has four bits for every position. The first bit, for the

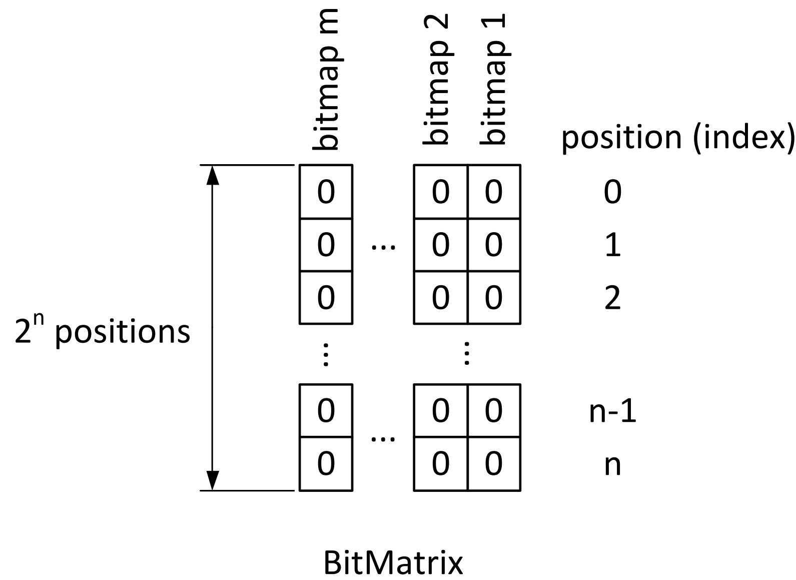

Figure 5 – The BitMatrix is a group of n bitmap vectors.

$bitmap_1$, second bit for the $bitmap_2$, third bit for the $bitmap_3$ and fourth bit for the $bitmap_4$. Each position of the BitMatrix has a single value that can vary from zero, if there is no packet stored in that position for any bitmap - [0000], to 15, if all bitmaps have packets stored in the same position - [1111].

Thus, for the storing procedure, it is not enough to determine the position that the packet will be stored in the BitMatrix but also, what bitmap to use. The $bitmap_1$, from the previous example, will store packets in the first bit of the position, so the value is 1 - [0001]. The $bitmap_2$, will store packets in the second bit of the position, so the value is 2 - [0010]. Hence, it is not possible only write this value in the BitMatrix position, as it would delete a previous value stored in the same position, and consequently the packet previously stored in another bitmap. Instead, the logical disjunction operator (OR) is used, avoiding any loss of packets previously stored from other bitmaps.

Algorithm 2 shows the procedure for storing packets in the BitMatrix on its different bitmaps.

Using the described technique is possible to segment the packets stored in a BitMatrix by creating a process to distinguish different tenants. In this work, we are identifying different tenants by using the source IP address of the packet. In this way, it is possible to identify which tenant originated the stored packet.

3.2 Monitoring Architecture

The monitoring process used in this work aims to determine:

Algorithm 2 BitMatrix packet store algorithm

1: **procedure** BitMatrix vector population
   Input: First 28 invariant bytes of the packet
   Output: Corresponding bit in the BitMatrix associated position set to 1
2: $hash_value = hash_function(i)$
3: **if** $hash_value > BitMatrix_length$ **then**
4: $BitMatrix_position = \mod(BitMatrix_length, hash_value)$
5: **else**
6: $BitMatrix_position = hash_value$
7: **end if**
8: $BitMatrix_Stored_Value = BitMatrix[BitMatrix_position]$
9: **if** $bitmap = bitmap_1$ **then**
10: $BitMatrix_value = 1$ [0001]
11: **else if** $bitmap = bitmap_2$ **then**
12: $BitMatrix_value = 2$ [0010]
13: **else if** $bitmap = bitmap_3$ **then**
14: $BitMatrix_value = 4$ [0100]
15: ...
16: **else**
17: $BitMatrix_value = m$
18: **end if**
19: $BitMatrix[BitMatrix_position] = BitMatrix_Stored_Value$ OR $BitMatrix_value$
20: **end procedure**

- How many unique packets crossed the network.
- What were their origin and destination.
- What were the network devices responsible for routing those packets.

The BitMatrix, allied with the network topology information, is sufficient to perform the proposed monitoring and further analysis.

We deployed an identical BitMatrix on every device in the network. In this way, we will have the same packet stored precisely in the same position in the BitMatrix, on each device responsible for forwarding it across the network. On a fixed period, we collect the BitMatrix with all packets stored on it. The number of packets processed in a specific device, per tenant and in total can be obtained by analyzing the information from the BitMatrix.

The number of packets processed in a specific device can be obtained by summing bits, which represent packets, stored in the BitMatrix. The number of packets counted in the BitMatrix will always be less than the number of packets processed by the device, given the probability of a hash collision to happen, as exposed in Section 3.3 Caveats and Limitations. To estimate the number of packets for a specific tenant, it is needed to select

the correspondent bitmap to that tenant inside the BitMatrix, and then, perform the sum. The total number of packets processed by a device, can be estimated by counting the number of bits stored in the whole BitMatrix. Assuming that all packets from tenant_{A} were stored in the bitmap_1, the Equation 3.3 represents the total number of packets sent from tenant_{A}.

$$\text{Total of packets from } tenant_A = \sum bitmap_1 \tag{3.3}$$

Equation 3.4 expresses the total amount of packets processed by the network device from where the BitMatrix was collected.

$$\text{Total processed packets} = \sum BitMatrix \tag{3.4}$$

To obtain the number of unique packets processed in the network, per tenant and in total, we used a logical disjunction in the tenant related bitmaps collected from BitMatrices across the network devices, in the same time frame. The logical operator OR will avoid counting the same packet more than once. The Expression 3.5 shows the sum of the bits resulting from the logical disjunction of all bitmaps correspondent to tenant_{A}, collect from all BitMatrices accross the network. The notation is: $\text{bitmap}_{\text{Tenant,BitMatrix}}$. The total number of unique packets processed in the network can be obtained by summing the total number of unique bits per tenant.

$$\text{Total of unique packets of tenant}_{\text{A}} = \sum \{bitmap_{\text{A},1} \vee bitmap_{\text{A},2} \vee \ldots bitmap_{\text{A,n}}\} \tag{3.5}$$

The packet's tenant source can be determined according to the bitmap in which the packet is stored, as each bitmap corresponds to one tenant. The destination of the packet is defined by analyzing which network devices stored the same packet. This is where the network topology knowledge becomes necessary. Thus, it is possible to create a traffic matrix based on that information.

3.3 Caveats and Limitations

Even using the invariant bytes of a packet for hash calculation, it is possible that two hashes with different parameters result in the same value. This occurrence is called hash collision and creates a gap between the number of packets actually forwarded by the network device and the total number of positions marked in the bitmap. There are three main factors that can cause variation in the number of hash collision in the proposed scenario: a) the size of the bitmap or, in other words, the number of positions available in

the vector; b) the occupation of the bitmap when marking a new position; c) the type of the hash function used to generate the hash value.

The size of the bitmap and the size of the hash value are related, in a sense that there is no point in setting a bitmap with more positions than a hash value. Positions beyond the maximum hash value will never be used. Nevertheless, setting a bitmap smaller than the maximum hash value will demand a modulo operation, using the bitmap size and the resulted hash value to determine the offset for the present packet. For example, in a given bitmap with 100 possible positions and a hash algorithm that returns a value between 1 and 1000, suppose that for the first packet the hash operation returned a value of 20. Thus, the marked position in the bitmap is position 20. For the next packet, the hash operation returns a value of 320. Once the returned value is bigger than the bitmap size, there is a need for performing the modulo operation to determine the bitmap position for this packet, which will result in position 20, causing a hash collision. And finally, but also important, we need to consider the bitmap occupancy. It is clear that the more occupied is a bitmap, bigger are the chances of a hash collision. Later in this work, in Chapter 6 Tests and Results, Section 6.1 P4 implementation using Mininet, Subsection 6.1.4 Collisions versus Occupation Tests, we discuss the relation between the bitmap occupancy and the number of hash collisions and the results of the usage of different hash algorithms.

4 Monitoring Framework

In this chapter, we describe the framework proposed for monitoring the traffic in a data center. We also discuss some example analyses that can be performed, and how to create a network traffic matrix, using the information provided by this framework.

4.1 Framework Architecture

The framework consists of four main components: data structures - sketches, collector and controller component, database and a query and presentation component. Figure 6 presents an overview of the complete architecture.

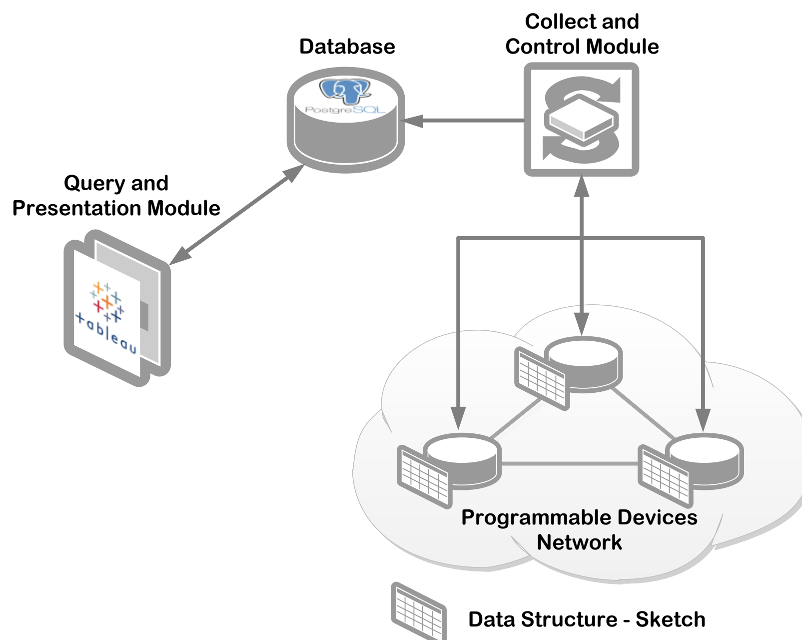

Figure 6 – The proposed framework architecture for the project, showing the interaction between its components.

The next paragraphs describe in detail each component of the proposed framework for this project.

Data Structures - Sketches: We implemented two types of data structures: an array of bitmaps, named BitMatrix in this context, intended to store each packet processed, segmented by a tenant which originated the packet, and a counter array, to store the number of bytes corresponding to each packet. The counter array will use the same hash

generated for BitMatrix, in order to determine the position in the structure that the device must add the bytes of the packet processed.

Due to the time constraint to collect the bitstreams generated by these structures, we implemented a redundant pair of sketches in an active-standby fashion. We explain in more detail its operation in the collector and controller component description.

Collector and Controller component: The primary function of this component is to collect the bitstream, which contains information from packets processed, and stores it out of the network device. Once had the bitstream collected, this component will reset the structure in the device, and the device will start to store new packets information on it. However, this process incurs in an issue, caused by the fact that network device process packets uninterruptedly and the collector component should take some time to perform the bitstream collection and reset the structure to make it ready to restart storing information of new packets. This lapse of time creates a gap where all packets forwarded by the device during that time, will not be stored in the data structure.

To overcome this obstacle, we created the previously mentioned active-standby redundant pair of sketches. The active sketch is the one in current use for storing information of the packet processing, and the standby sketch is an idle sketch ready to get into operation. The Collector and Controller mechanism, after a predefined interval of time, from now on referred to as the epoch, performs an active-standby switchover, making the hitherto active sketchs available to have its information collected, as the device will not store any more information into it. After the component performs the collection, it resets the sketches, now in a standby state, making it ready to get into operation when the next epoch starts.

After the Collector and Controller component has retrieved the bitstream from the sketches, it will parse and store this information in the database. The component will include meta-data for the bitstream, as the time and date for collection, network device information from where the component collected the bitstream and correspondent tenant.

Database: In this work, the relational database PostgreSQL was used to store the information produced by the sketches and parsed by the Collector and Controller component. Once stored, the information becomes available to be queried as needed. In Table 2 the data dictionary for the table used in database to store the bitstream information is detailed.

Query and Presentation Component: We used the commercial analytics platform Tableau to retrieve sketches information from the database and present relevant key performance indicators (KPIs) to support decision making in network traffic engineering. Many other tools could be used, such as MicroStrategy or Microsoft Excel. Tableau was selected due to our familiarity with the tool and the grant for a student licence from Tableau. We can use the indicators to create the following insights:

Table 2 – Data Dictionaty

Table	Column	Data type	PK	Nullable	Description
bitmatrix	bitstreamid	INT	Y	not null	bitstream identification
bitmatrix	deviceid	INT	Y	not null	device identification
bitmatrix	tenantid	INT	Y	not null	tenant identification
bitmatrix	collectdate	DATE	Y	not null	date of collection
bitmatrix	collecttime	TIME	Y	not null	time of collection
bitmatrix	bitstream	VARBIT	Y	not null	tenant bitstream
bitmatrix	countstream	ARRAY	Y	null	tenant bytes counter array

- Estimate the total number of packets and bytes processed in the network device in a specific time interval;
- Estimate the number of packets and bytes sent per a tenant, processed in the network device in a given time interval;
- Estimate how many packets and bytes used a specific path in the network, segmented per a tenant, in a given time interval; and
- Compare traffic among paths for load balancing.

4.2 Network Traffic Monitoring per Tenant

The network traffic monitoring approach, adopted for this work, uses compact probabilistic structures, also known as sketches, to store traffic information of packets forwarded across the network. Based on sketches, we created an array of bitmaps called BitMatrix to store packets processed by the network device, using a different bitmap to store packets originated by each tenant. From the BitMatrix, we can extract estimated measurements based on packets information. We also used a counter array sketch per tenant, to store the packet length information and then to be able to estimate the volume of the network traffic.

To better explain the network traffic monitoring process, we described it step by step, including packet processing and forwarding, sketches' information collection and the different analyses performed.

4.2.1 Network Device Packet Processing

Every time the network device receives a new packet, it parses the packet header to create a list of header fields that hash function will use as input to process and obtain the hash correspondent to the packet. The packet header fields used as input for the hash algorithm must uniquely represent the packet across all devices in the network, for that, the list will include only header fields that do not change across the hops in the forwarding

path. Thus, the input list for the hash algorithm excludes the following information from the packet headers: Time To Live (TTL), Differentiated Services Code Point (DSCP), Explicit Congestion Notification (ECN) and Checksum. The following fields from the packet header are used to compute the hash: Version, Internet Header Length (IHL), Total Length, Identification, Flags, Fragment Offset, Protocol, Source Address and Destination Address. Additionaly, as part of the input list, the first 8 bytes of the IP payload was also included.

There are numerous hashing algorithms that a network device can use to process these fields and output a hash number representing the packet. We tested different techniques and hashing algorithms, as discussed in Section 6.2.1 Hash Algorithms and Packet headers Selection. In summary, the main concern is to make sure that the hashing algorithm used can generate a number bigger or equal to the BitMatrix length. Otherwise, the BitMatrix will never use the positions indexed above the hash generated number. e.g. If the network device uses the Fletcher-16 checksum hash algorithm, the output will never be higher than 2^{16} or 65,536. If the BitMatrix length is bigger than this value, it will never use those positions above that value.

Taking this into consideration, the hashing algorithm will generate a number, that will be used by the device to set the correspondent position for the tenant in the BitMatrix. Also, the device will use the same hash number to determine a position in the tenant's counter array. Once the position is defined, the device will sum the value from the Total Length packet header to the value in that position of the counter array.

Next, the network device will forward the packet, based on its forwarding table. The network device will change the frame header to reflect the new layer 2 addresses (source and destination MAC addresses) and forward it using the appropriate network interface to the next hop in the network, which will repeat the process described here hop by hop, until the packet reaches its final destination, according to its IP destination address.

4.2.2 Collection and Control

The Collector and Controller component starts the BitMatrix and counter array collection process, for all network devices at the same time, aiming to achieve some degree of synchronism for the collection. Synchronism in the collecting process is desired to assure that all sketches have stored packets forwarded in the same epoch, in all network devices. Even though, packets in a forwarding process across the network, in the moment of the collection, will not be stored in all network devices because it is not delivered yet. This packet will be stored on the network devices that has already processed it, but not in those devices on the next hops, as the packet has not reached them at that moment.

In a network, real-time traffic processing, sketches collection and reset is a challenging task. This process would take a short time to execute, even so, during its execution, packets are continuing to be processed on the network device, and will not be stored in the BitMatrix. Depending on the time spent for the collection, those lost packets may introduce even more errors in the measurements. In order to avoid this, as previously stated, we use a pair of BitMatrix for every network device, in an active-standby fashion. This architecture changes the collection process introducing a control layer on it. The control will make the standby BitMatrix take over the active one. This action will make the BitMatrix available to be collected without any losses of packets, as the network device will be storing packets in the other BitMatrix instance. After collection, the standby BitMatrix is reset and made available to switchover to active again when the next collection occurs. The same mechanism is also used for the counter array sketch control and collection.

The period for collection will vary mainly to assure that the BitMatrix is not saturated, as discussed in Chapter 6 Tests and Results, Section 6.1 P4 implementation using Mininet, Subsection 6.1.4 Collisions versus Occupation Tests. Once it is a probabilistic structure, there is the possibility to occur a hash collision, making the device trying to store a packet in an already occupied position in the BitMatrix. The chances of a collision to happen increases as the BitMatrix gets more occupied along the time. Ideally, the collection frequency should happen often enough to avoid the saturation of the BitMatrix. Other factors that will directly impact the occupancy are the BitMatrix length and the number of packets processed per second on the device. The BitMatrix length and the period of the collection are the parameters that can be manipulated to keep the BitMatrix occupancy at an acceptable level.

4.2.3 Parse and Storage

Once collected, the BitMatrix data runs through a parsing process, to extract the bitstream corresponding to every tenant stored in there. In a raw state, the BitMatrix data will look like a counter array, as each position contains many bits, indeed, one for each tenant to store a packet. The parsing process will extract from BitMatrix data, the bitstream for every tenant individually. Figure 7 demonstrates the parsing process of a BitMatrix that was designed to store information for four tenants.

In Figure 7, the phases of the parse and store processes are exposed in letters A to D; phase A is the BitMatrix collection phase. For this process, the collector and controller component will connect to the network device and retrieve the bitstreams from the BitMatrix, storing it locally, in memory. Phase B is the conversion of the decimal values from BitMatrix to binary values. The next phase, C, is when the parsing happens dividing the BitMatrix into bitmaps, one for each tenant; T1, T2, T3 and T4. In this example, we are using four tenants to simplify, but a BitMatrix can scale to accommodate

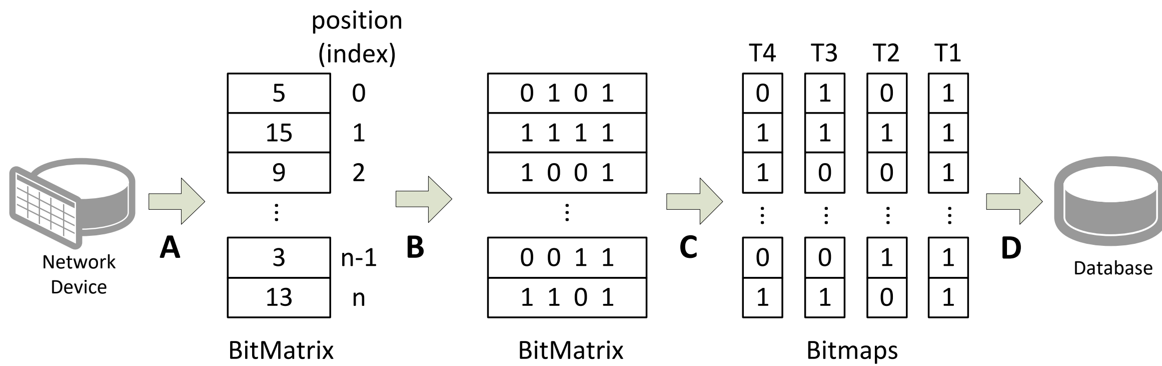

Figure 7 – The phases for collect, parse and store the BitMatrix information.

a higher number of tenants, depending on the memory available in the network device, and finally, in phase D is when the bitmap and the metadata are stored in the database.

4.2.4 Analyses

Bitmaps can be used to estimate the volume of traffic between two observed network devices in any period during the monitoring activity. For this, it is required to retrieve the set of bitmaps collected during this interval for both devices and then compare the two sets of bitmaps. The more common bits in the same position, the more common packets have passed through these devices. If the intersection of bits in the same position is small, it is possible to conclude that few common packets have passed by these elements.

Several analyses can be done by using the tenants' bitmaps and counter arrays stored in the database. This collection of bitmaps and counter arrays will enable us to perform network analyses by selecting and manipulating the bitmap information to create measurements, and using its metadata as dimensions. For more clarity, consider the network diagram from Figure 8.

Based on the collected bitmaps and counter arrays information, it is possible to answer the following questions, for a given period:

- How many packets were exchanged between tenant_A and tenant_B?
- What was the network device with the highest load?
- What was the link with higher throughput? Also, what was the average throughput on it?

How many packets exchanged between $\textbf{tenant}_\textbf{A}$ and $\textbf{tenant}_\textbf{B}$? To answer this question, we will need to do two separated analysis, one to identify the packets sent from tenant_A to tenant_B, and another to identify packets sent from tenant_B to tenant_A.

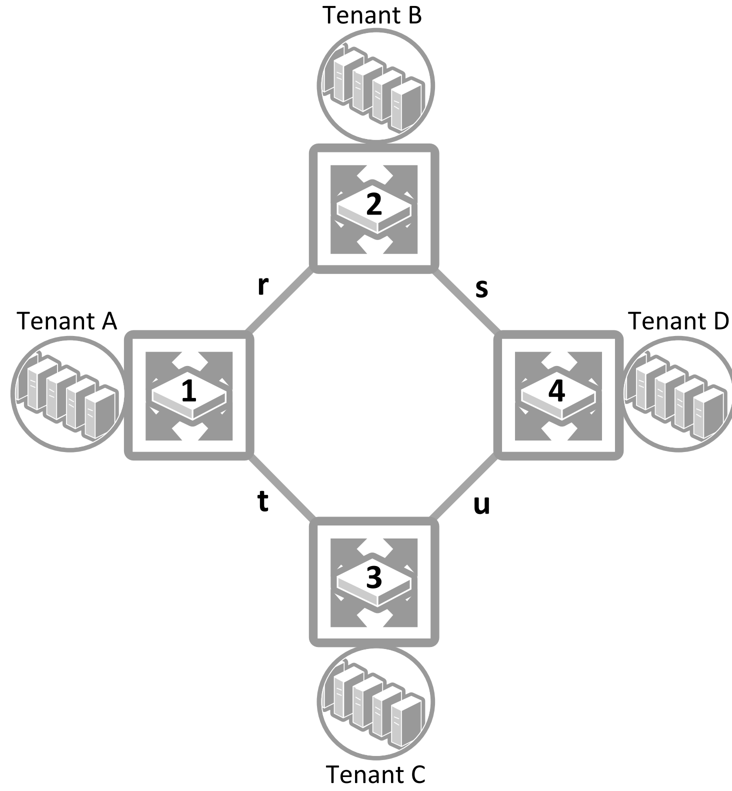

Figure 8 – The network diagram proposed to illustrate the analytics possibilities.

Firstly, for packets sent from tenant_{A} to tenant_{B}, we can infer, once knowing the network topology, that those packets are the ones present in devices number 1 and 2 but not present in devices 3 and 4. The Equation 4.1 shows the logical operations. The notation is $bitmap_{tenant,router}$.

$$\text{Total Pkts A} \rightarrow \text{B} = \sum\{bitmap_{\text{A},1} \wedge bitmap_{\text{A},2} \wedge \neg bitmap_{\text{A},3} \wedge \neg bitmap_{\text{A},4}\} \tag{4.1}$$

Secondly, we will use the same logic to calculate packets sent from tenant_{B} to tenant_{A}, now using the bitmaps associated with tenant_{B}. Equation 4.2 shows these logical operations.

$$\text{Total Pkts B} \rightarrow \text{A} = \sum\{bitmap_{\text{B},1} \wedge bitmap_{\text{B},2} \wedge \neg bitmap_{\text{B},3} \wedge \neg bitmap_{\text{B},4}\} \tag{4.2}$$

Finally, to answer the question, we need to determine the total number of the packets exchanged between tenant_{A} and tenant_{B}. For this, we need to sum the results

from Equation 4.1 and 4.2. The Equation 4.3 shows the final result and the answer to the question.

$$\text{Total Pkts A} \leftrightarrow \text{B} = \sum\{\text{Total Pkts A} \rightarrow \text{B}, \text{Total Pkts B} \rightarrow \text{A}\} \tag{4.3}$$

What was the network device with the highest load? It is possible to determine the element with the higher load by estimating the number of packets processed for each one. To do this calculation, we need to sum the number of bits from every tenant's bitstream. The Equation 4.4 demonstrates how to estimate the total number of packets for a device.

$$\text{Total Pkts Dev_1} \ = \sum\{bitmap_{\text{A},1}, bitmap_{\text{B},1}, bitmap_{\text{C},1}, bitmap_{\text{D},1}\} \tag{4.4}$$

A similar process will be used to calculate the number of total packets for all other devices. Once calculated, we can sort by the total number of packets, determining what was the network device with the highest load in the network.

What was the link with higher throughput? Also, what was the average throughput on it? To determine what was the link with higher throughput, we need to identify what packets are present in a pair of directly connected devices. e.g. all packets present in device 1 and device 2 may have used the link r to travel from one device to another. However, it is possible to go from device 1 to device 2 through links t, u and s. This is possible, but unlikely in a regular network, unless an abnormal situation has happened, like a link failure. We will not consider this likelihood for this scenario. Thus, to find the total number of packets that crossed the link, we need to sum the packets present in both devices. We can determine what packet are present in both devices by using a logical conjunction operation between both tenants corresponding bitstream. The Equation 4.5 shows the operation to find the number of packets that crossed the link `r`.

$$\begin{aligned}\text{Total in Dev 1 and Dev 2} = \sum\{&bitmap_{\text{A},1} \land bitmap_{\text{A},2},\\ &bitmap_{\text{B},1} \land bitmap_{\text{B},2},\\ &bitmap_{\text{C},1} \land bitmap_{\text{C},2},\\ &bitmap_{\text{D},1} \land bitmap_{\text{D},2}\}\end{aligned} \tag{4.5}$$

The next step is to find the average throughput for each link. For this we will use the counter array created for every tenant. As discussed before, once the network device stores the packet in the BitMatrix, it uses the same index to store the total length packet header value in the tenant correspondent counter array. The counter array to be used can

be either from device 1 or device 2, as both arrays will have the packet length value for the packet. To estimate the total amount of bytes from packets that crossed the link, we will use the result of the logical conjunction operation to find what position to read values from the counter array, as the packets stored in the BitMatrix and the packet length value stored in the counter array corresponds to the same packet.

Finally, reading the total packet length value, from positions in the counter array that correspond to the packets present in both devices, we can estimate the total amount of bytes that travelled through the link by summarizing values on the specified position. To calculate the average throughput, we will need to divide the total amount of the bytes by the period we are querying, in seconds.

4.3 Network Traffic Matrix

Traffic Matrix, in summary, is a representation of the traffic volume exchanged between source and destination pairs. These pairs can be composed of single routers or even of networks. The traffic volume measurements are in packets or bytes. Traffic matrices are used in several network engineering processes as capacity planning, network optimization, and anomaly detection among others (TUNE; ROUGHAN, 2013).

A traffic matrix is represented by a three-dimensional model, with *i,j*-th entries representing the incoming traffic from node *i* and outgoing traffic on the node *j*. Each entry represents the traffic volume, in packets or bytes, for the pair *i.j* in a time interval (XIAO, 2008). Figure 9 presents an example of a traffic matrix.

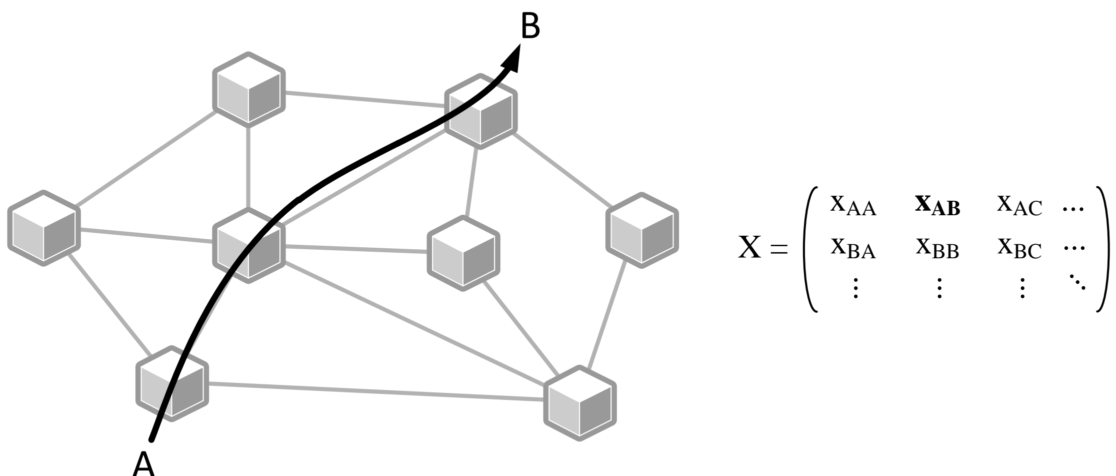

Figure 9 – An example of a traffic matrix.

Using the data set from BitMatrices, generated by the network devices, it is possible to build a traffic matrix with packets or bytes measurements, employing logical operators.

Using the network diagram from Figure 8, as an example, we can create a traffic matrix using the following steps.

- The ingress in the network is determined by the packet source IP address.
- The egress can be defined by observing the peers of a specific device. Packets present in the BitMatrix of the device in question and in one or any peer device, are egressing packets. Let us take the device 3 as an example. If one packet is present in device 3 and in device 1 OR in device 4, the packet is egressing the network by the device 3, if it was not ingressed the network by the device 3. A situation where the packet is not observed in any other peer device shows that the device in question, was responsible for the ingress end the egress of that packet.
- Once determining the packets to be counted as ingressing or egressing, it is possible to retrieve their length from counter arrays. This information can be used to create a traffic matrix based on throughput.

The proposed method for creating traffic matrices, using sketches, improves network visibility by generating information near to real-time, by analyzing a small volume of data. This improvement is possible because sketches are compact structures, and they are collected often, enabling traffic matrices creation for every set collected. According to TUNE; ROUGHAN, the generation of traffic matrix will depend on the application and available measurement and 5 minutes to an hour are common choices. In our tests, we collect sketches in periods of 10 seconds to 1 minute. This short period for collection allows detecting abnormalities in network traffic, in a near real-time manner.

5 BitMatrix in P4 Language

This Chapter describes the implementation of the sketches, described in the previous sections, in P4 language. We created a multi-tenant BitMatrix to store packets and counter arrays, to store packet's length (bytes). Also, for comparison purposes, we create counters to count the total number of packets and the sum of all packets length. In this way, it will be possible to compare the numbers from BitMatrix with those from counters.

The proposed topology is composed of two hosts, interconnected by a P4 network device. The network addresses corresponding to tenants and the IP and MAC addresses of each element can be found in Figure 10.

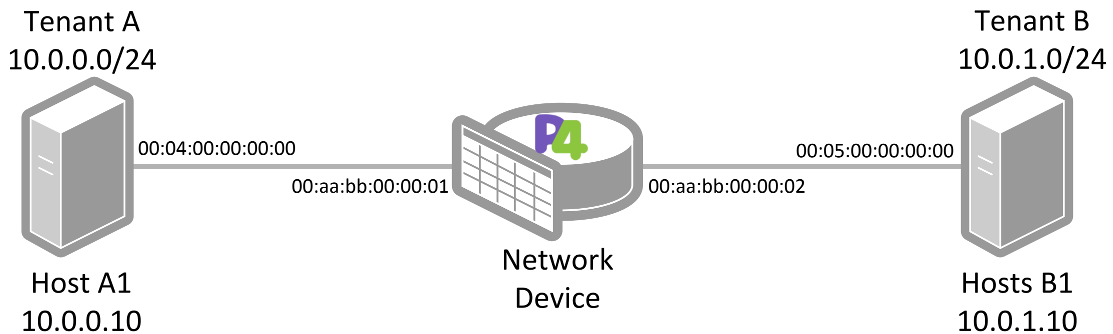

Figure 10 – Proposed topology to exemplify the sketches deployment in a P4 programmable network device.

P4 is a declarative language that expresses how a P4 enabled network device will process and forward a packet in the network. It is divided into an ingress and egress pipeline and consists of parsing and using match+action tables to manipulate the packet. The parsing process maps the headers present in the packet and the tables perform a lookup for a specific header field and applies the corresponding action for each table. Hereafter we describe the elements present in the code and how the packet processing happens. We will highlight parts of the code, explaining the processing instructions on it. The entire code can be found in Appendix A.1 Deployment in P4 example.

The first phase of the packet processing in P4 is the parsing. The parsing process occurs according to definitions made in the header, producing a parsed representation of the packet. The Header Type P4 abstraction in lines 3 and 13 in Listing 5.1, specifies the fields within the Ethernet layer header and the IP header. In lines 11 and 31, the header instance specifies the instance of the packet header.

Listing 5.1 – Partial P4_14 code defining headers

```
//header

header_type eth_t {
    fields {
        dstAddr     : 48;
        srcAddr     : 48;
        etherType   : 16;
    }
}

header eth_t eth;

header_type ipv4_t {
    fields {
        version     : 4;
        ihl         : 4;
        diffserv    : 8;
        totalLen    : 16;
        id          : 16;
        flags       : 3;
        fragOffset  : 13;
        ttl         : 8;
        protocol    : 8;
        hdrChecksum : 16;
        srcAddr     : 32;
        dstAddr     : 32;
        payload8B   : 64;
    }
}

header ipv4_t ipv4;
```

The parser in P4 works as a finite state machine. Figure 11 represent the parse graph for the Listing 5.2, with each state transition as an edge and each state as a node. This Figure shows a header for each state.

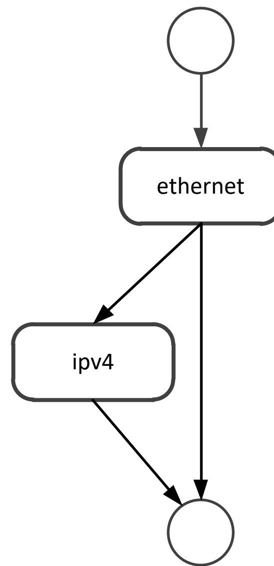

Figure 11 – Parse graph for the P4 deployment.

The parser creates a parsed representation of the packet, on which match+action stages will work. Match+action may update the parsed representation by modifying a field and by changing which header instances are valid, resulting in adding and removing headers. The parsed representation holds packet headers as the match+action process updates them.

Listing 5.2 – Partial P4_14 code defining parser

```
33 // parse
34
35 parser start {
36     return parse_eth;
37 }
38
39 #define ETHERTYPE_IPV4          0x0800
40
41 parser parse_eth {
42     extract(eth);
43     return select(latest.etherType) {
44         ETHERTYPE_IPV4      : parse_ipv4;
45         default: ingress;
46     }
47 }
48
49 parser parse_ipv4 {
50     extract(ipv4);
51     return ingress;
```

```
52      }
```

The field list declaration in Listing 5.3 specifies the header fields used for checksum calculation, from line 57 to line 67, and for the hash function, from line 71 to line 80. The fields used for hash calculation are the unvarying fields from the IP layer and the first 8 bytes of its payload.

Listing 5.3 – Partial P4_14 code defining field list definition

```
54  // field_list definitions
55
56  field_list ipv4_checksum_list {
57      ipv4.version;
58      ipv4.ihl;
59      ipv4.diffserv;
60      ipv4.totalLen;
61      ipv4.id;
62      ipv4.flags;
63      ipv4.fragOffset;
64      ipv4.ttl;
65      ipv4.protocol;
66      ipv4.srcAddr;
67      ipv4.dstAddr;
68  }
69
70  field_list hash_fields {
71      ipv4.version;
72      ipv4.ihl;
73      ipv4.totalLen;
74      ipv4.id;
75      ipv4.flags;
76      ipv4.fragOffset;
77      ipv4.protocol;
78      ipv4.srcAddr;
79      ipv4.dstAddr;
80      ipv4.payload8B;
81  }
```

The Listing 5.4 is defining a different kind of header type abstraction. This time, the header type has been used to declare metadata instances. The network device uses these metadata during the packet processing to store values for BitMatrix and counter array population and for packet forwarding.

In BitMatrix, there are four metadata fields used:

- bitmatrix_idx: is used to store the position corresponding to the packet, in the

BitMatrix. This position is calculated using a modulo function with the value returned from the hash function and the BitMatrix length.

- bitmatrix_flag: this field stores the value read from the position determined by the bitmatrix_idx. This value is used to perform the logical disjunction operation with the tenant corresponding value. Then, the result of the operation will be stored in the field again, to be written in the BitMatrix later.

- bitmatrix_tenant: the tenant information will be stored in this field during the BitMatrix population process and will be used later to determine what counter array will be used to store the packet length, as there is one counter array structure for each tenant.

- bitmatrix_value: this metadata is used to store the number of bytes already stored in the counter array position, determined by the bitmatrix_idx, and add the packet length value to it. After this operation, the new value is written in the same position of the counter array.

The metadata used in the routing process are the following two fields:

- nhop_ipv4: this field will store the next hop IP address returned from the match process in the table ipv4_lpm, then, it will be used to determine what is the destination MAC address.

- nhop_add: will store the destination MAC address to be used in the deparse process, when sending the packet.

Listing 5.4 – Partial P4_14 code defining metadata

```
83  // defining metadata
84
85  header_type custom_metadata_t {
86      fields {
87          bitmatrix_idx    : 16;
88          bitmatrix_flag   :  2;
89          bitmatrix_tenant :  2;
90          bitmatrix_value  : 20;
91      }
92  }
93
94  metadata custom_metadata_t custom_metadata;
95
96  header_type routing_metadata_t {
97      fields {
```

```
98          nhop_ipv4 : 32;
99          nhop_add  : 48;
100     }
101 }
102
103 metadata routing_metadata_t routing_metadata;
```

In P4 language it is possible to use a primitive to perform calculation, having as input a list of fields, parsed from the packet. This calculation process is defined in Listing 5.5. The texttfield_list_calculation declaration has as input, the texttfield_list defined in Listing 5.3. Using this list as input for the algorithm will result in an output that can be referenced as the texttfield_list_calculation name. On line 115, as an example, we can find the texttfield_list_calculation name hash. This hash will be used later in this code. The texttcalculated_field declaration, on line 123, uses the texttipv4_checksum field_list_calculation to update the IPv4 checksum value for the packet.

Listing 5.5 – Partial P4_14 code calculation process

```
105 // field_list_calculations
106
107 field_list_calculation ipv4_checksum {
108     input {
109         ipv4_checksum_list;
110     }
111     algorithm : csum16;
112     output_width : 16;
113 }
114
115 field_list_calculation hash {
116     input {
117         hash_fields;
118     }
119     algorithm : crc16;
120     output_width : 16;
121 }
122
123 calculated_field ipv4.hdrChecksum  {
124     update ipv4_checksum if (ipv4.ihl == 5);
125 }
```

Counters, meters and registers are called stateful memories and maintain state for longer than one packet. Its instantiation requires memory resources on the target. The stateful memory is organized in arrays of cells, and these cells can be read or update by an action, applied by a table. The action will reference a cell from its array name and index. For sketches used in this work, we used registers to create the structure in the network

device. On line 129 from Listing 5.6, we created a register that represents the BitMatrix structure, to store packets on each position. The `width: 2` creates space to accommodate two tenants, in this case, tenant_{A} and tenant_{B}. And the texttinstance_count represent the number of positions of the BitMatix. Register on lines 134 and 139 are used to store the bytes from packets stored in the BitMatrix. Notice that the textttinstance_count, which represents the length of the registers, are the same across all registers (8,192). This fact is because the index corresponding to a specific packet will be the same in BitMatrix and in tenant's counter_array.

Listing 5.6 – Partial P4_14 code register definition

```
127 // register definitions
128
129 register bitmatrix{
130     width : 2;
131     instance_count : 8192;
132 }
133
134 register counter_array_A{
135     width : 20;
136     instance_count : 8192;
137 }
138
139 register counter_array_B{
140     width : 20;
141     instance_count : 8192;
142 }
```

Action functions are called in tables. They receive parameters from tables to perform modifications in headers and metadata in parsed representation or in the stateful memory. The values passed to these parameters are programmed into the table entry by the run time API. P4 exposes a standard set of actions that may or may not be supported by the target. A brief summary for all primitive actions can be found in (The P4 Language Consortium, 2017). Here, in Table 3, are the actions used in this P4 code example.

API Name	**Summary**
drop	Drop a packet (in the egress pipeline).
modify_field_with_-hash_based_offset	Apply a field list calculation and use the result to generate an offset value.
register_read	Read from an indexed instance of a register and store the value into a field.
bit_or	Perform bitwise OR operation on two val- ues and store in a field.

register_write	Write a value into an indexed instance of a register.
modify_field	Set the value of a field in the packet's parsed representation.
add_to_field	Add a value to a field.

Table 3 – Primitive Actions

The action in Listing 5.7 is the one responsible for storing the packet in the BitMatrix, in the correct position for the correspondent tenant. The parameter `tenant_flag`, passed to the action function, indicates what tenant generated the packet. The table is a declarative structure specifying match and action operations. In this case, the table uses the packet's source IP address to match the tenant flag, and then invoke the action `set_bitmatrix`, passing the `tenant_flag` value to it.

In line 147, the action `modify_field_with_hash_based_offset` will determine what index position in the BitMatrix the packet will be stored, even in the case of the hash value returned from the hash function is bigger than the BitMatrix length. e.g., If we are using the hash function csum16, the value returned by this function can assume any value between 0 and 65,535. It will be a problem if the BitMatrix length is smaller than the hashed value. The action `modify_field_with_hash_based_offset` will use three parameters to calculate the hash and apply a modulo function to find the corresponding index for the packet. The first one, `custom_metadata.bitmatrix_idx`, will be used to save the result for the action modifying the field instance to the resulted value. The next parameter, `0`, represents the base value to add to the hash value. In our case, we do not need to add any value, as the BitMatrix can store a packet in positions starting from zero. The third paramenter specifies the field list calculation used to generate the hash value. In this case, the hash list name is `hash`. Please, refer to Listing 5.5, line 115, to verify the hash `field_list_calculation` declaration, and to Listing 5.3, to see the fields used in the hash calculation, in line 70. The fourth and last parameter will determine the size of the hash value range. It must be larger than 0, but can not be bigger than the BitMatrix length.

The packet store process in the BitMatrix needs to preserve the existing value for all positions, as one position will store, into a single value, packets from different tenants, one packet per bit. In order to achieve this, we perform a logical disjunction operation between the value stored in BitMatrix, in the position stored in the metadata by the `modify_field_with_hash_based_offset` action, and the value corresponding to the tenant in question, represented by the parameter `tenant_flag`. The code on line 148, is for the action `register_read`, which stores the value at an specific position (`custom_metadata.bitmatrix_idx`) from the `bitmatrix` register array , into another metadata field instance (`custom_metadata.bitmatrix_flag`) to be used in the next step, by the

bit_or operation.

The bit_or operation, will execute a logical disjunction operation using the value read from the BitMatrix and the tenant_flag. This operation will assure that the previously stored packet in the BitMatrix, will not be deleted. i.e., the tenant_flag is for $tenant_A$ uses the first bit of the BitMatrix position to store the packet. If there is a packet from $tenant_B$ already stored in the same position, using the second bit, the P4 action will read the value 2 (binary 10) and will perform an OR operation with the value corresponding to $tenant_A$, which is 1 (binary 01), resulting in a new value 3 (binary 11). Therefore, the new value will be written back into the BitMatrix, representing the packets stored in that position, one in each bit. The action register_write is the responsible for this write process. Finally, there is a primitive action used to store the tenant_flag parameter value into a metadata field instance for future use in the code. Line 151, shows how the modify_field primitive performing this action.

Listing 5.7 – Partial P4_14 code actions

```
144 // actions
145
146 action set_bitmatrix(tenant_flag) {
147     modify_field_with_hash_based_offset(custom_metadata.
            bitmatrix_idx, 0, hash, 8191);
148     register_read(custom_metadata.bitmatrix_flag, bitmatrix,
            custom_metadata.bitmatrix_idx);
149     bit_or(custom_metadata.bitmatrix_flag, custom_metadata.
            bitmatrix_flag, tenant_flag);
150     register_write(bitmatrix, custom_metadata.bitmatrix_idx,
            custom_metadata.bitmatrix_flag);
151     modify_field(custom_metadata.bitmatrix_tenant, tenant_flag);
152 }
```

In Listing 5.8, the counter array used in the structure, accumulates the number of bytes processed by the network device. In P4 compilation time, the device creates one counter array per tenant. On the Listing, we can see two counter arrays, one for $tenant_A$ and another for $tenant_B$. The primitives in each action are the same, but the counter array used to store the bytes are different. As seem in Listening 5.6, registers used for the counter array have the same length as registers defined for the BitMatrix. However, there is a difference in terms of width; while the register used for BitMatrix has a length equivalent to the number of tenants to be monitored, the register used for counter array has a fixed width of 20 bits, allowing to store up to 1,048,576 bytes. Considering that a packet will carry up to 1,500 bytes, one position in the counter array is enough to store bytes for almost 700 packets. This width is way bigger than it needs to be, and we defined it with that size to not run into the risk of overflowing the register. The width was arbitrarily

defined and should be reviewed for production use. In line 155 of the Listing 5.8, the primitives actions are used to populate the counter array for tenant_{A}. The actions will read the value existing in the position, sum the total length value from the packet header to the retrieved value, and write the result to the same position in the counter array. The `set_counter_array` selection happens in the control flow.

Listing 5.8 – Partial P4_14 code tables definitions

```
154
155 action set_counter_array_A() {
156     register_read(custom_metadata.bitmatrix_value, counter_array_A
            , custom_metadata.bitmatrix_idx);
157     add_to_field(custom_metadata.bitmatrix_value, ipv4.totalLen);
158     register_write(counter_array_A, custom_metadata.bitmatrix_idx,
            custom_metadata.bitmatrix_value);
159 }
160
161
162 action set_counter_array_B() {
163     register_read(custom_metadata.bitmatrix_value, counter_array_B
            , custom_metadata.bitmatrix_idx);
164     add_to_field(custom_metadata.bitmatrix_value, ipv4.totalLen);
165     register_write(counter_array_B, custom_metadata.bitmatrix_idx,
            custom_metadata.bitmatrix_value);
166 }
```

The other actions in Listing 5.9 are for forwarding the packet, and they will set the correct source and destination MAC addresses, according to the egress port defined for the packet. Also, the Time To Live (TTL) in the IPv4 header is decreased in 1, as per the normal forwarding packet process in any network routing device. The `drop()` primitive will cause a packet drop in the device.

Listing 5.9 – Partial P4_14 code tables definitions

```
168
169 action set_nhop(nhop_ipv4, port) {
170     modify_field(routing_metadata.nhop_ipv4, nhop_ipv4);
171     modify_field(standard_metadata.egress_spec, port);
172     modify_field(ipv4.ttl, ipv4.ttl - 1);
173 }
174
175 action set_dmac(dmac) {
176     modify_field(eth.dstAddr, dmac);
177 }
178
179 action rewrite_mac(smac) {
```

```
180         modify_field(eth.srcAddr, smac);
181 }
182
183 action _drop() {
184         drop();
185 }
```

Tables Section specify match and action operations and other attributes. The action specification in a table designates which action functions are available to the table's entries. The table declaration specifies a list of field matches used for matching packets. A field match can be a reference to a header, the validity bit for a header, a reference to a field, or a masked reference to a field. The returned information from the matching step is used in an action defined in the table. The table population happens in run time, and the values can be changed at any time. The logic that selects which tables are applied to a packet when a pipeline processes it, is defined in the control flow. The Listing 5.10 shows all tables for the packet processing, and in Listing 5.13 are the values used in the tables' population.

Listing 5.10 – Partial P4_14 code tables definitions

```
186 // tables
187
188 table set_bitmatrix_table {
189         reads {
190             ipv4.srcAddr : lpm;
191         }
192         actions {
193             set_bitmatrix;
194         }
195         size: 32;
196 }
197
198
199 table set_counter_array_A_table {
200         actions {
201             set_counter_array_A;
202         }
203         size : 1;
204 }
205
206 table set_counter_array_B_table {
207         actions {
208             set_counter_array_B;
209         }
210         size : 1;
211 }
```

```
212
213 table ipv4_lpm {
214     reads {
215         ipv4.dstAddr : lpm;
216     }
217     actions {
218         set_nhop;
219         _drop;
220     }
221     size: 1024;
222 }
223
224 table forward {
225     reads {
226         routing_metadata.nhop_ipv4 : exact;
227     }
228     actions {
229         set_dmac;
230         _drop;
231     }
232     size: 512;
233 }
234
235 table send_frame {
236     reads {
237         standard_metadata.egress_port: exact;
238     }
239     actions {
240         rewrite_mac;
241         _drop;
242     }
243     size: 256;
244 }
```

Counters maintain state for longer than one packet, and they are called stateful memories. On Listing 5.11 we created counter to compare the result obtained through the sketches summarization and the byte and packet actually processed by the network device. The `pkt_counter` counter declares a set of counters attached to the table named `set_bitmatrix_table`. It allocates one counter cell for each entry in that table.

Listing 5.11 – Partial P4_14 code counter definition

```
245 // counter definition
246
247 counter pkt_counter {
248     type: packets_and_bytes;
249     direct : set_bitmatrix_table;
```

```
250 }
```

The control flow uses a sequence of tables for processing a packet. At configuration time, the control flow will express in what order the tables are to be applied. Control flow may apply tables, call other control flow functions or test conditions. The apply instruction indicates the execution of a table. The apply instruction may influence the control flow by specifying a set of control blocks from which one is selected to be executed. The control flow on Listing 5.12 will first test if the packet is valid and if the time to live is bigger than 0, in line 255. If so, it will apply the `set_bitmatrix_table`. Next, based on the `custom_metadata.bitmatrix_tenant` value, it will apply the tenant's corresponding counter array table. After that, all tables for the packet forwarding process are applied.

Listing 5.12 – Partial P4_14 code control

```
252 // control flow
253
254 control ingress {
255     if(valid(ipv4) and ipv4.ttl > 0) {
256         apply(set_bitmatrix_table);
257         if (custom_metadata.bitmatrix_tenant == 1) {
258             apply(set_counter_array_A_table);
259             }
260         else {
261                 apply(set_counter_array_B_table);
262             }
263         apply(ipv4_lpm);
264         apply(forward);
265     }
266 }
267
268 control egress {
269     apply(send_frame);
270 }
```

As part of p4c compiler, the graphs backend produces a visual representation of a P4 program. This representation helps to understand the control flow, showing which conditions and tables will be applied during the packet processing on ingress and egress pipelines. Figure 12 represents the control flow commands for the ingress pipeline and Figure 13 shows the command for egress pipeline.

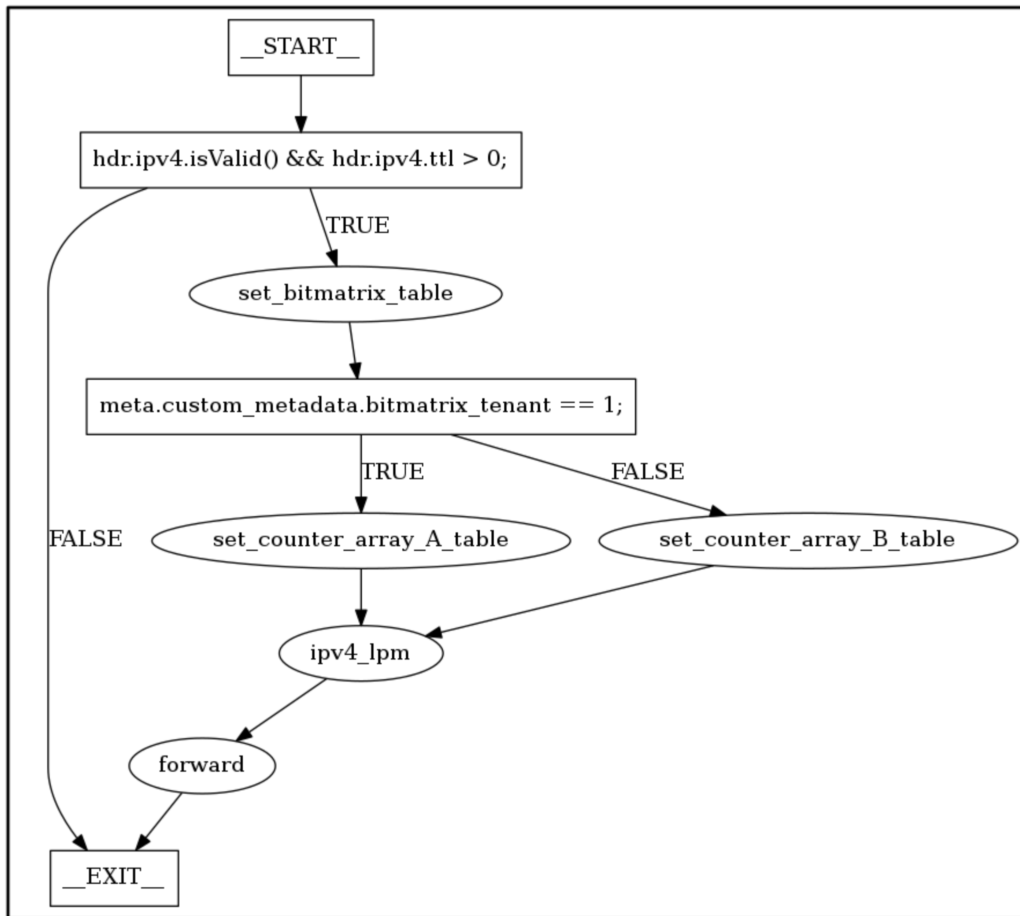

Figure 12 – Top-level control graph, generated for the ingress control block, of the P4 BitMatrix code.

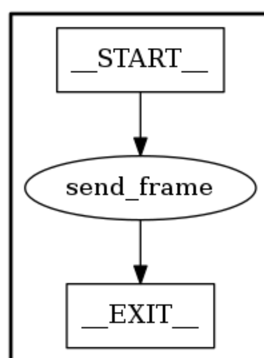

Figure 13 – Top-level control graph, generated for the egress control block, of the P4 BitMatrix code.

Tables population is a process that takes place during the run-time period. This procedure happens just after the network device has loaded the P4 code. Although, it is possible to add or delete entries anytime during the run-time period. Also, it is possible to set a default action to a table. The default action is performed when no table entry matches.

If no default action is designated, the table does not affect the packet and processing continues according to the control flow. The list of supported commands includes:

- `table_set_default <table name> <action name> <action parameters>`
- `table_add <table name> <action name> <match fields> => <action parameters> [priority]`
- `table_delete <table name> <entry handle>`

The Listing 5.13 shows the table commands used in this P4 code.

Listing 5.13 – Commands for the run-time process of table population, in the network device

```
table_set_default send_frame _drop
table_set_default forward _drop
table_set_default ipv4_lpm _drop
table_set_default set_bitmatrix_table set_bitmatrix
table_set_default set_counter_array_A_table set_counter_array_A
table_set_default set_counter_array_B_table set_counter_array_B

table_add set_bitmatrix_table set_bitmatrix_0 10.0.0.0/24 => 1
table_add set_bitmatrix_table set_bitmatrix_0 10.0.1.0/24 => 2
table_add ipv4_lpm set_nhop 10.0.0.10/32 => 10.0.0.10 1
table_add ipv4_lpm set_nhop 10.0.1.10/32 => 10.0.1.10 2
table_add forward set_dmac 10.0.0.10 => 00:04:00:00:00:00
table_add forward set_dmac 10.0.1.10 => 00:05:00:00:00:00
table_add send_frame rewrite_mac 1 => 00:aa:bb:00:00:01
table_add send_frame rewrite_mac 2 => 00:aa:bb:00:00:02
```

As demonstrated, the P4 language offers enough resources to deploy a model, using sketches for traffic analysis purposes. In this deployment, we could create, populate and retrieve information from the probabilistic structures and perform analysis regarding the network traffic in a multi-tenant environment.

It is possible to create several analyses, using the datastreams generated from the probabilistic structure, offering different insights about the network, devices and links behaviour. Those analyses are fundamental for decision making during the traffic engineering process for the network, also for capacity planning, performance and tenant behaviour analysis. Although, the method exposed focused more on explaining the process of information generation and lacks parameters' definition for deployment in a real-life network environment. Some of those parameters, and theirs factors will be discussed in Chapter 6 Tests and Results.

6 Tests and Results

This chapter is divided into two main sections. Section 6.1 outlines test and results for a P4 implementation in a adapted Mininet environment. In Section 6.3, we explore the usage of traces from real internet traffic, simulating a bigger network and running substancial more traffic on it. By the end of the chapter, in Section 6.4 we detail how we used machine learning to apply a adjusting factor to traffic statistics generated by the BitMatrix framework.

6.1 P4 implementation using Mininet

In this section, we describe the BitMatrix framework implementation using Mininet to emulate a simple network aiming to validate the results.

6.1.1 BitMatrix Tests and Methodology

The main goal for this test is to implement the BitMatrix using available commands and structures in the P4 language. To do that, the BitMatrix was implemented as a P4 register wide enough to host several bitmaps, each one used to measure traffic from a different tenant, identified by its Source IP subnetwork. The BitMatrix is used to aggregate bitmaps in order to register packets from different tenants. Tenant is defined as an external network, connected to the P4 network device. The goal is to use BitMatrix associated with counter arrays to estimate the amount of packets and bytes transmitted for each tenant and, in addiction, understand the path taken by those packets inside the network. Each packet received by the P4 device computes a hash value. This value is used to determine which position will be set in the BitMatrix, according to its origin (tenant). In this way, it is possible to determine which tenant is responsible for each packet in the network.

In this proof of concept, a BitMatrix composed by three bitmaps was used. Figure 14 shows its structure of this BitMatrix. This resulted in a P4 register with a width equal to 3. Thus, it is possible to segment traffic from up to three tenants, setting different bitmaps in the BitMatrix according to which tenant originated the packet. Using a P4 table, a value for each tenant was assigned, according to its source IP network: 1 to tenant A, 2 to tenant B and 4 to tenant C. Once the hash value of each packet is computed, a modulo operation is applied to the value to determine what position should be set in the BitMatrix. This is achieved by using the P4 primitive action `modify_field_with_hash_based_offset`. As each position of the BitMatrix has three bits, we used another P4 primitive action named `bit_or` to set the correct bit in that position of BitMatrix by performing a logical OR

operation using the current value for the selected position and the tenant value.

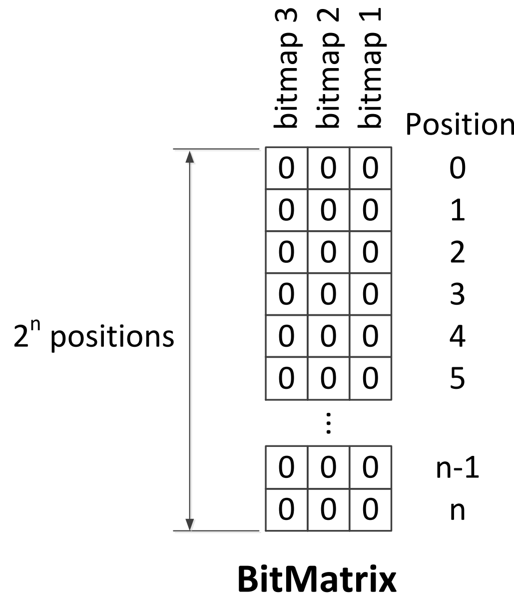

Figure 14 – BitMatrix structure and its bitmaps.

As an example, consider that the current value for a selected position is 4 (100 in binary). This indicates that a packet originated from tenant C has already set that position. Nevertheless, this has no impact for the same position in other bitmaps in the BitMatrix. If a packet from tenant A falls in the same position, the logical OR operation is applied using the current value in the BitMatrix (4 or 100 in binary) and the tenant A value (1 or 001 in binary), resulting in the new value 5 (101 in binary).

A P4 register was also used to create counter arrays. The usage of counter array targets to count the number of bytes transmitted for each packet. For this, the register used to create the counter array has the same length of BitMatrix, but a width large enough to avoid overflow. We used a width of 20 bits, which allows to store up to 1 Mbyte. Considering that each packet has a size of 1,500 bytes, it is enough to sum bytes from up to 699 packets, for each position. So, this can still counting bytes from packets until it reaches 700 hash collisions for that one specific position. Different from BitMatrix, the counter arrays can not be combined into an unique register. Thus, one counter array was defined for each tenant.

To continue the BitMatrix evaluation we used the results from Figure 16 and the percentage of hash collision was target to keep under 10%. The hash collision was calculated by dividing the total number of positions marked in the bit array by the number of processed packets. This approach resulted in an epoch of 60s, a bandwidth limited to 1Mbps and bitmap size of 16384 positions. The setup for this experiment was constructed using Mininet network emulator customized in order to enable P4 switch in the emulated

network.

The topology used was composed of three hosts and four P4 switches. Each host received an IP address from a different network, emulating different tenants. The topology is presented in Figure 15.

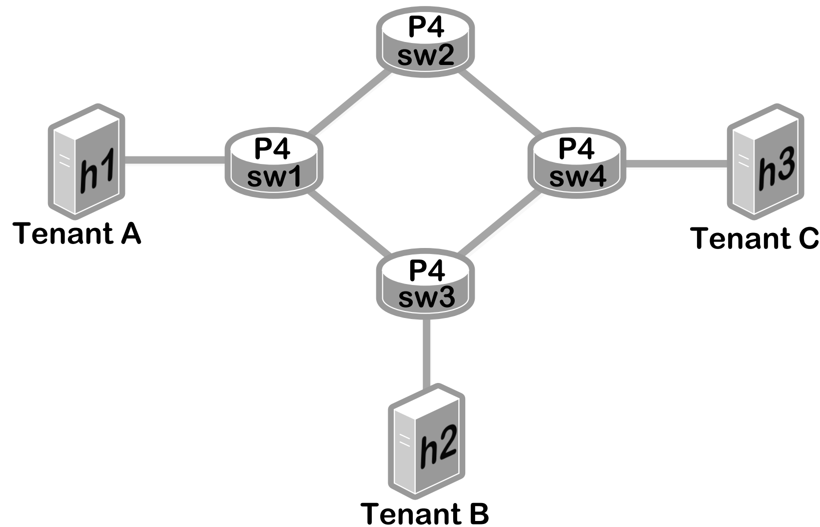

Figure 15 – Mininet emulated network topology with P4-enabled forwarding.

The paths between tenants were arbitrarily defined as shown in Table 4.

Table 4 – Traffic path between tenants

Tenant Pairs	Switches hop by hop in order
A to B	sw1 - sw3
B to A	sw3 - sw1
A to C	sw1 - sw2 - sw4
C to A	sw4 - sw2 - sw1
B to C	sw3 - sw4
C to B	sw4 - sw3

Every packet processed by the P4 switches generates entries in its corresponding BitMatrix (instantiated in each switch). Our P4 implementation consists of processing packets for the BitMatrix and can be described in the following general steps:

- Completely parse the packet headers of Layer 2, 3, 4 and the first 8 Bytes of the payload;
- Select the packet headers to be used in the hashing algorithm;
- Determine the position in the bitmap;

- Determine the position (using the same hashing value) in the counter array to sum the total bytes of the current packet with the previous ones;
- Forward the packet to the next hop.

6.1.2 Parsing the Packet Header

Parsing the packet header in a customized fashion is a flexibility provided by P4 (BOSSHART et al., 2014). The P4 implementation used in this work enabled the P4 switch to completely parse the packet headers from layer 2 to layer 4 and also the first 8 Bytes (64 bits) of the payload. We implemented the parser for TCP, UDP and ICMP protocols. As the TCP layer usually brings optional headers, we used a variable length header to accommodate it. Ignoring TCP optional header would mislead the parsing of the payload.

The parsing of payload was done by creating one header field to receive the 8 bytes (64 bits) subsequent to the Layer 4 header. The code in Listing 6.1 is part of the P4 source code for header definition and parsing instruction and shows how packet payload was parsed.

Listing 6.1 – Partial P4_14 code defining Header and Parser

```
...
parser parse_ipv4 {
    extract(ipv4);
    return select(latest.protocol) {
        IP_PROTOCOLS_ICMP   : parse_icmp;
        IP_PROTOCOLS_TCP    : parse_tcp;
        IP_PROTOCOLS_UDP    : parse_udp;
        default: ingress;
    }
}

...

parser parse_tcp {
    extract(tcp);
    return parse_payld;
}

...

header_type payld_t {
    fields {
        userdata8B  : 64;
    }
}
```

```
header payld_t payld;

parser parse_payld {
    extract(payld);
    return ingress;
}
```

6.1.3 Hashing and Hash Inputs

To identify a packet as unique across all hops in a network, the packet's headers used as input for the hash algorithm, must not vary during the forwarding process. Duffield and Gross-glauser (DUFFIELD; GROSSGLAUSER, 2001) define that the IPv4 fields with low entropy are those which do not vary along the forwarding path for a given packet. Then, to have a low entropy, in this work we used the invariant IPv4 header fields and the first 8 bytes of the payload as input for the hash algorithm. According to Snoeren (SNOEREN et al., 2001), those inputs are sufficient to differentiate unique packets.

The fields to be used in the hash operation for each packet are defined in the P4 `field_list` hash_fields as shown in the code of Listing 6.2.

Listing 6.2 – Partial P4 code defining input fields for hashing and the hash algorithm used.

```
field_list hash_fields {
    ipv4.version;        // 4
    ipv4.ihl;            // 4
    ipv4.totalLen;       //16
    ipv4.id;             //16
    ipv4.flags;          // 3
    ipv4.fragOffset;     //13
    ipv4.protocol;       // 8
    ipv4.srcAddr;        //32
    ipv4.dstAddr;        //32
    payld.userdata8B;    //64
}

field_list_calculation hash {
    input {
        hash_fields;
    }
    algorithm : crc16;
    output_width : 16;
}
```

6.1.4 Collisions versus Occupation Tests

Towards to determine what is the more efficient setup for the BitMatrix size and maximum occupancy to be target for it, we conducted tests using a fixed hash value size and varying the BitMatrix length and the percentage of occupancy. The hash algorithm used was the checksum 16 (csum16), which generates a value of 16 bits length. The BitMatrix tested were 2,048, 4,096, 8,192, 16,384, 32,768 and 65,536 bits length. We did not control the occupation itself, instead, we processed an amount of packets to be 5%, 10%, 25%, 50% and 100% of the BitMatrix length. The output was the occupation smaller than the amount of packets processed due the hash collision inherent to the process. The results can be observed in Figure 16.

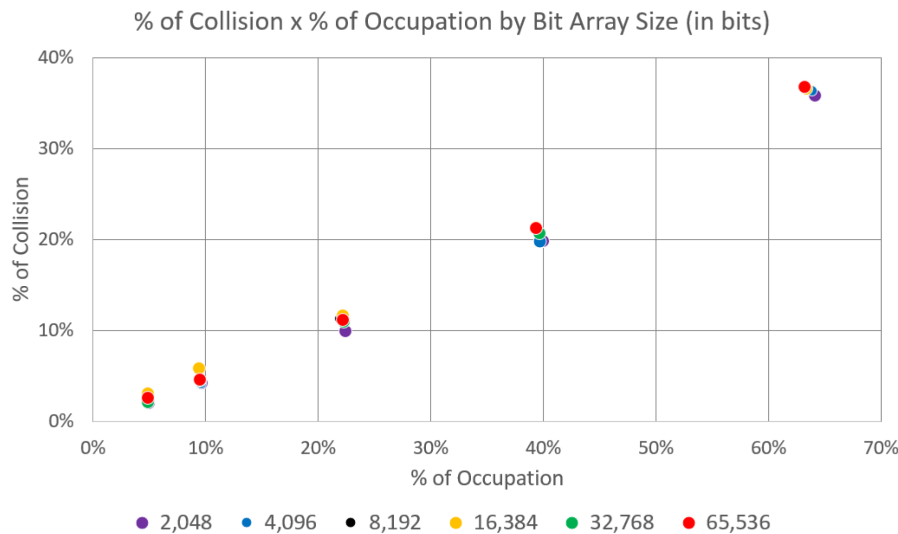

Figure 16 – Relation between % of collision X % of occupation.

There are pros and cons on every setup. While a setup with a longer BitMatrix can provide more positions for packets, it will also demand more available memory in the device which, sometimes, can be a limiting factor. Therefore, it will demand tests and a better assessment when implementing in a production environment.

6.1.5 Retrieving and Processing BitMatrix

To understand what was the path of a certain packet, it was necessary to compare the datastreams from different devices in the network. The data to be compared need to belong to the same epoch (monitoring interval). An Epoch is the time frame in which the P4 device stored information in the BitMatrix. From time to time, the data structures need to be collected and reset. This determines the beginning of a new epoch.

The P4 switch is not in charge of collecting and storing the data structures. This task was performed by the collector and controller component, who collected the values

from the P4 registers and reset them to start a new epoch. For this experiment, the time frame for each epoch was set to 60 seconds.

After collecting the data structures, they were stored in a database from where they can be retrieved and processed in order to provide the information requested. By processing the stored data from bitmaps and counter arrays, it is possible not only to obtain information regarding the amount of packets and bytes processed by each device per tenant, but also to identify how may packets and bytes per tenant went through a specific path in the network.

6.1.6 Results

An interesting metric directly related to the accuracy of the method is the occupancy of the bitmap. As demonstrated in Figure 16, the higher is the occupancy in a bitmap the lower is its accuracy. In Figure 17, the occupancy for the bitmap corresponding to tenant A was calculated for each epoch and plotted as a time series graphic. For this calculation, we took the number of bits set in the bitmap divided by the bitmap total length to find what was the bitmap occupancy for a given epoch, in percentage.

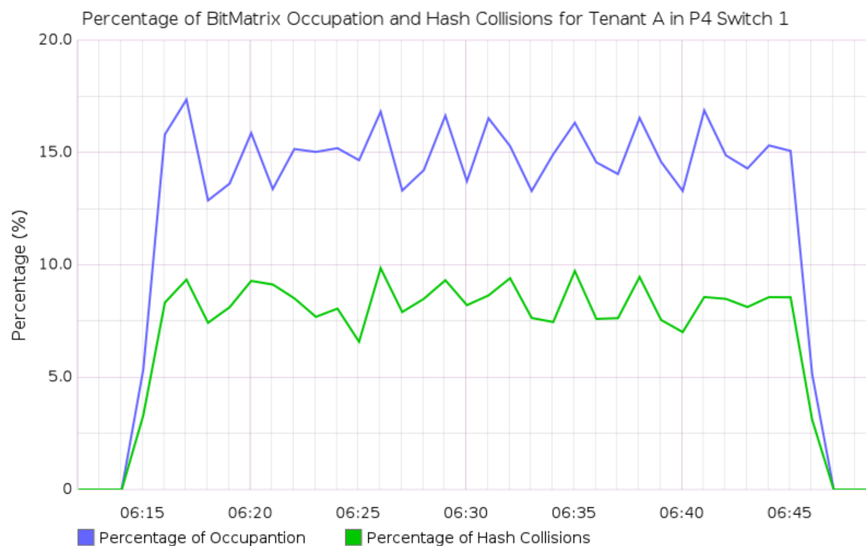

Figure 17 – BitMatrix Occupancy and Collisions for tenant A in P4 switch 1.

In Figure 18, the amount of packets per tenant can be visualized during a specific period. This metric was a result of the sum of all set positions in the bitmap corresponding to each tenant in the BitMatrix in each epoch.

By counting the positions with bits set to 1 from all bitmaps of the BitMatrix, we obtain the approximated total number of packets processed by each P4 device. Figure 19 shows these numbers. This value reflects less packets than each P4 switch actually processed due to the hash collisions.

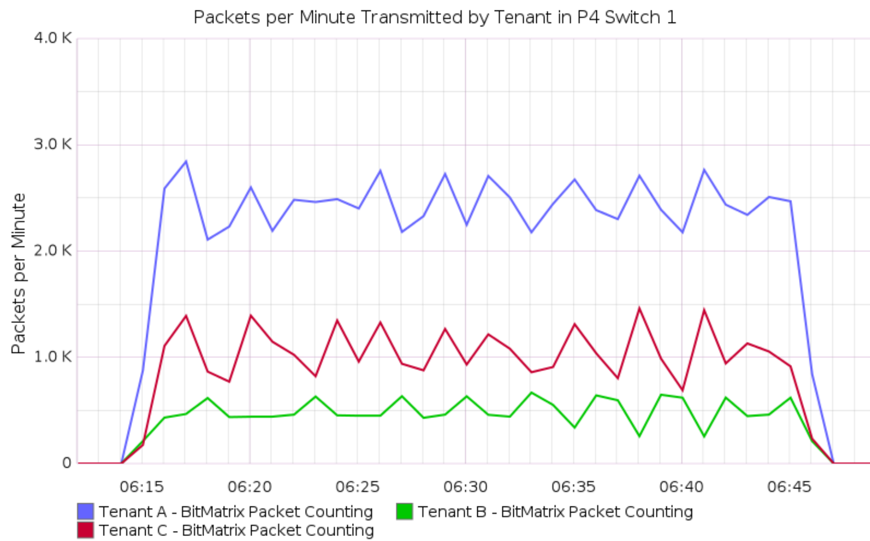

Figure 18 – Amount of packets per tenant in P4 switch 1.

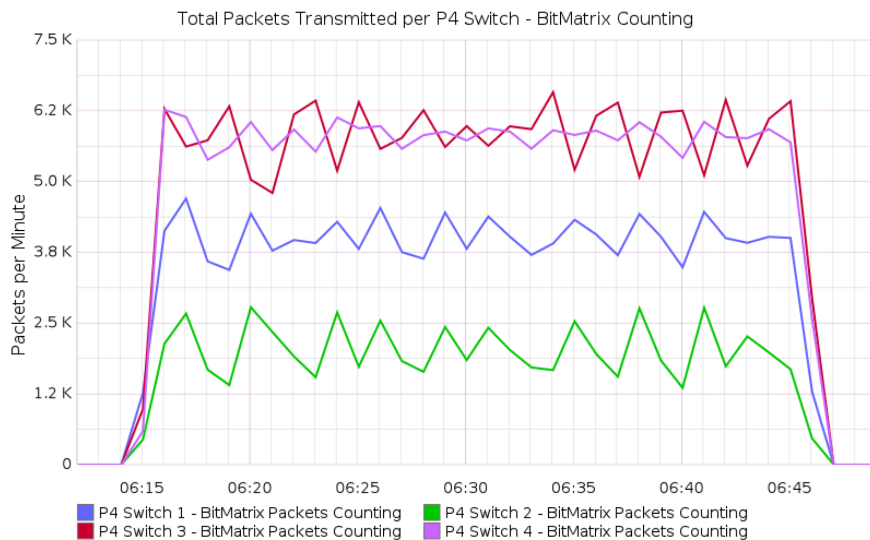

Figure 19 – Total amount of packets per P4 switch.

The amount of bytes sent by tenants was computed by counting the total values of each position from every tenant-correspondent counter array. In Figure 20, we present the amount of bytes transmitted for each tenant computed from P4 switch 1.

The total amount of bytes processed by each P4 switch is presented in Figure 21. The total number of Bytes related to packets forwarded by a particular P4 switch is computed by counting the values from each position in every tenant's counter array. As counter arrays indeed count the number of bytes, there is no hash collision and this result will reflect exactly the volume of bytes processed by the P4 switches.

By computing bitmaps from different network devices, it is possible to determine what was the path a packet went through in the network. In Figure 22, it is possible to see the amount of packets per minute originated by tenant A with destination to tenant B

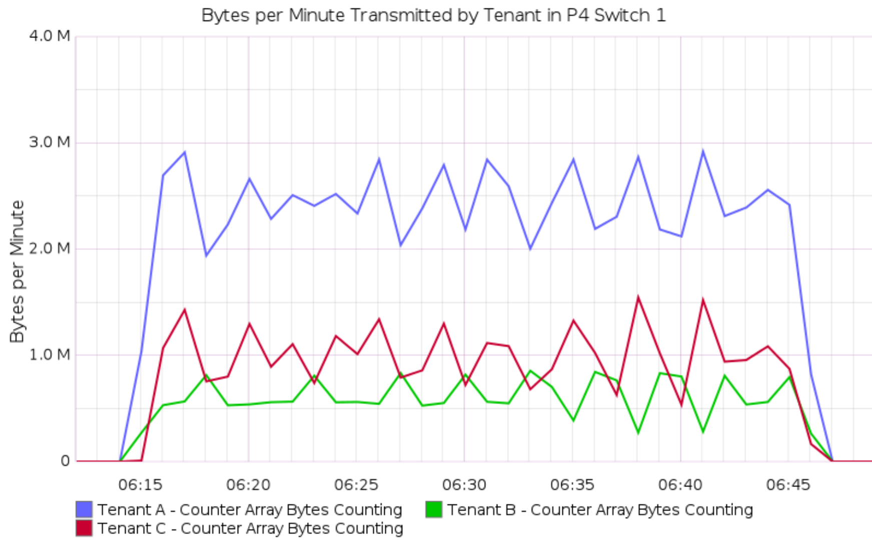

Figure 20 – Amount of bytes per tenant in P4 switch 1.

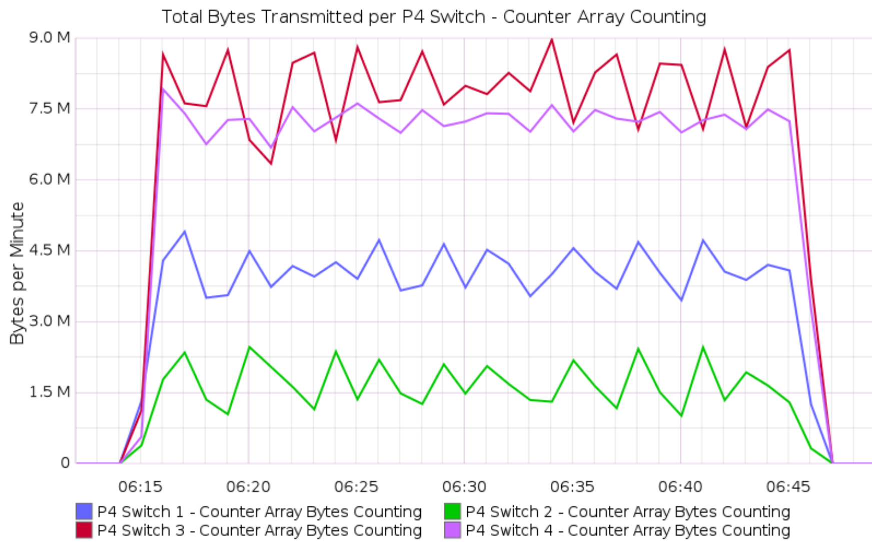

Figure 21 – Total amount of bytes per P4 switch.

and packets originated from tenant B with destination to tenant A. Those metrics were calculated using logical operations with the bitmaps for tenants A and B from every P4 switch responsible for forwarding packets between those two tenants. Considering the information in Table 4, it is possible to state that packets going from tenant A to tenant B will take the path passing through P4 switch 1 and then switch 3, and packets going from tenant B to tenant A will take the reverse route, passing firstly through P4 switch 3 and finally through P4 switch 1. With this information, it was possible to estimate which packets flowed from tenant A to tenant B. To do that, we used the logical expression ((sw1_A & sw3_A) & !sw4_A) where sw1_A is the bitmap corresponding to tenant A from P4 switch 1, sw3_A is the bitmap corresponding to tenant A from P4 switch 3, sw4_A is the bitmap corresponding to tenant A from P4 switch 4. Similar logic was used

to determine which packets sent from tenant B went through P4 switch 3 and P4 switch 1, towards tenant A. This logic indicates what position was set by packets exchanged between tenants A and B.

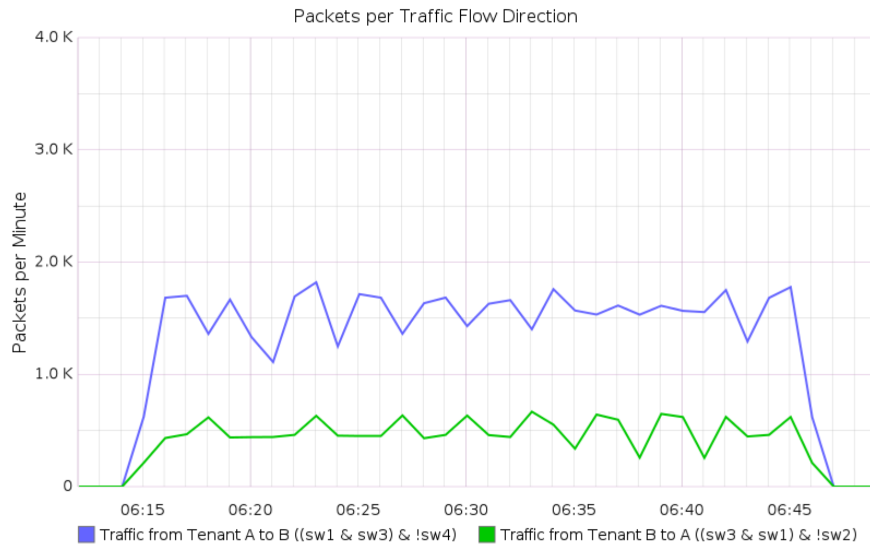

Figure 22 – Amount of packets on path AB+BA.

Once these positions are known, we were able to count how many bytes were involved in the data transfer between tenants A and B, as shown in Figure 23.

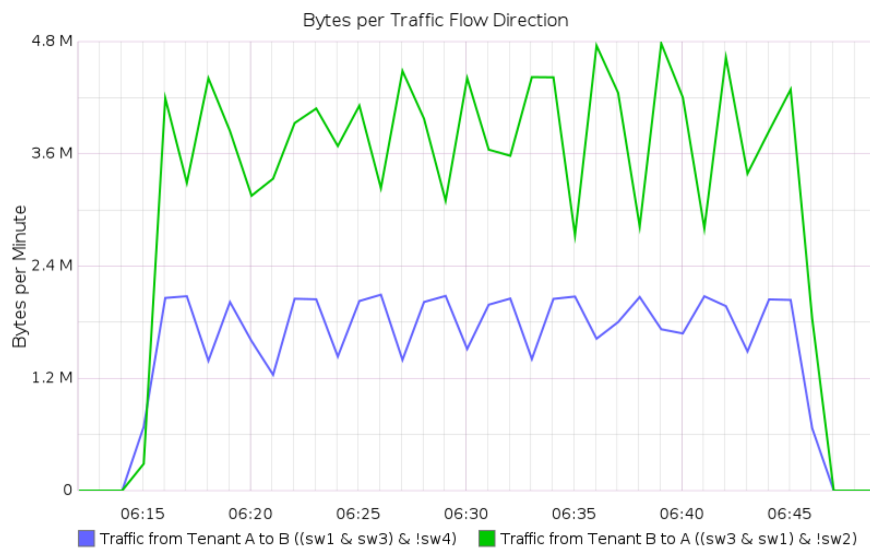

Figure 23 – Amount of bytes on paths AB+BA.

The amount of possibilities in terms of processing and mining the collected information is immense. Once the BitMatrix and counter array data are obtained, it is just a matter of making logic operations, counting and crossing the bitstreams to generate network information for each tenant and as a whole.

6.1.7 Contribution - New Command bm_register_read_all added to Thrift interface

The Thrift interface was used to retrieve the BitMatrix bitstreams periodically from the Behavioral Model v2 (BMv2) framework. However, when we were working in this project, the only command supported on Thrift interface, by the BMv2 was the `bm_register_read`. The runtime CLI `bm_register_read` command will read the information in the register, returning the value for one position per reading. This happens because the parameters needed for the command are the register name and the index position. Thus, if the register has 8,192 positions, to retrieve all values, we need to perform the command 8,192 times. The Listing 6.3 shows how the `bm_register_read` command is defined in the BMv2 context.

Listing 6.3 – `bm_register_read` commnad definition in standard.thrift code for the behavioral-model

```
BmRegisterValue bm_register_read(
  1:i32 cxt_id,
  2:string register_array_name,
  3:i32 idx
) throws (1:InvalidRegisterOperation ouch)
```

Reading a register several times creates an overload in the BMv2 framework, besides to introduce a considerable delay until to retrieve all the values from a register. These issues had a negative impact when scaling the solution. By increasing the traffic, the sketches need to be increased in length, and the time for reading needs to be shorter to avoid a high level of hash collisions. At a certain point, the time for collection will be longer than the sketch epoch, making the process unfeasible.

To solve this problem, after some discussions in the P4 community group (p4-dev-request@lists.p4.org - P4-dev Digest, Vol 24, Issue 1), the support for a new command was included in the commit #419 of the Behavioral Model v2. The command `bm_register_read_all` was created to read all register cells in one row, using the Thrift interface. Also, a modification in the `bm_register_read` runtime CLI command was done, making the index optional. If it is not declared, the entire register array will be read. Listing 6.4 shows the new command introduced in the BMv2 context.

Listing 6.4 – `bm_register_read_all` command definition in standard.thrift code for the behavioral-model

```
 list<BmRegisterValue> bm_register_read_all(
  1:i32 cxt_id,
  2:string register_array_name
) throws (1:InvalidRegisterOperation ouch)
```

Tests using the new command in the scenario described in section 6.1 P4 implementation using Mininet resulted in a reduction from tenths of seconds to less than 2 seconds. Figure 24 shows time enlapsed to collect the bitmaps from each elemente (sw1, sw2, sw3 and sw4) using the `bm_register_read` command, and Figure 25 shows time of collection when using the `bm_register_read_all` command.

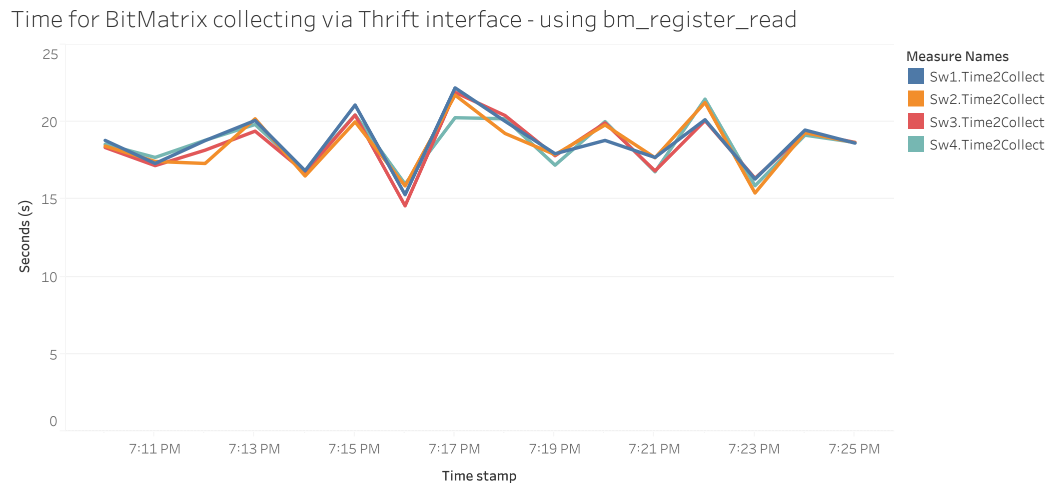

Figure 24 – Time for BitMatrix collection via Thrift interface - using the command bm_register_read

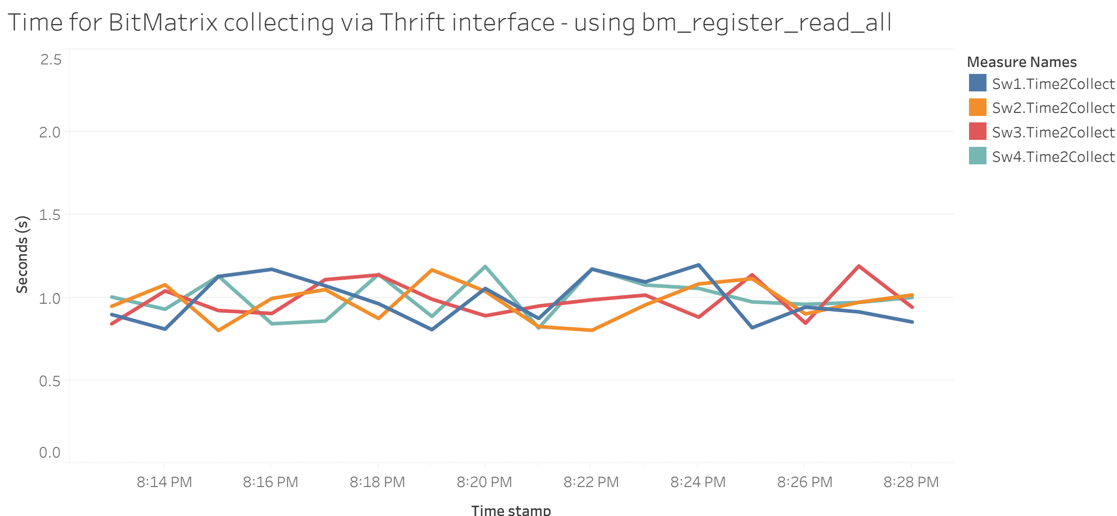

Figure 25 – Time for BitMatrix collection via Thrift interface - using the command bm_register_read_all

6.2 Python Implementation

The python implementation aims to simulate network devices, executing the BitMatrix algorithm on a set of packets from real-world traffic traces. Tests described in Section 6.1 P4 implementation using Mininet were performed in a emulated environment,

using Mininet, BMv2 and iperf3 to generate traffic. The results may suffer some bias from the traffic generation tool used. In python implementation, we used real traffic traces from CAIDA, captured from one of the routers in Equinix datacenter in San Jose, CA, connected to a backbone link of a Tier1 ISP between San Jose, CA and Los Angeles, CA (http://www.caida.org/data/passive/passive_2012_dataset.xml). The dataset contains anonymized passive traffic traces from CAIDA's equinix-sanjose monitors on high-speed Internet backbone links. Traffic traces were anonymized using prefix-preserving anonymization, and the payload was removed from all packets. Traces can be read with any software that reads the pcap (tcpdump) format. Depending on tests to be performed, we used a different Python framework to simulate packets processing and collect information generated from this process. The following subsections expose details about each framework used for tests and the results.

6.2.1 Hash Algorithms and Packet headers Selection

Ideally, the optimal result will happen when we don't have any hash collision storing packets in the sketch. However, this is not how it works in practice. Hash collisions happen very often and it is related to several factors, such as sketch elevate level of occupation, fragmented packets, and even for casuality. In order to verify other factors that could contribute in raising the collision number of incidences, we performed tests using different hash algorithms, different hash inputs and processing traffic traces from different sources.

In the sense of hash code and checksum are similar things - a numeric value, computed for a block of data, that is relatively unique, we used four algorithms for the tests, as follows:

Checksum 16
CRC 16
CRC 32
MD5

For checksum 16, the algorithm was programmed in Python as a function, using the logic for calculating the checksum for a string of bits. Listing 6.5 shows the code for the checksum calculation.

Listing 6.5 – Function definition in Python for Checksum 16 hash calculation

```
def checksum(str_):
    str_ = bytearray(str_)
    csum = 0
    countTo = (len(str_) // 2) * 2

    for count in range(0, countTo, 2):
        thisVal = str_[count+1] * 256 + str_[count]
```

```
            csum = csum + thisVal
            csum = csum & 0xffffffff

        if countTo < len(str_):
            csum = csum + str_[-1]
            csum = csum & 0xffffffff

        csum = (csum >> 16) + (csum & 0xffff)
        csum = csum + (csum >> 16)
        answer = ~csum
        answer = answer & 0xffff
        answer = answer >> 8 | (answer << 8 & 0xff00)
        return answer
```

To generate the hash code using the cyclic redundancy check (CRC), we used the crcmod 1.7 package, which is a Python module for generating objects that compute the CRC. It includes a optional C extension for fast calculation, although we used its pure Python implementation. This package allows the use of any 8, 16, 24, 32, or 64 bit CRC. There is no attempt to explain how the CRC works. For the tests, we generated a Python function for the CRC 16 and 32 bits, as shown in Listing 6.6.

Listing 6.6 – Using crcmod Python library to compute the CRC hash code.

```
    def crc16_comp(str_):
        str_ = bytearray(str_)
        crc16 = crcmod.mkCrcFun(0x18005, rev=False, initCrc=0xFFFF,
            xorOut=0x0000)
        answer = crc16(str(str_))
        return answer

    def crc32_comp(str_):
        str_ = bytearray(str_)
        crc32 = crcmod.mkCrcFun(0x104C11DB7, rev=False, initCrc=0
            xFFFFFFFF, xorOut=0xFFFFFFFF)
        answer = crc32(str(str_))
        return answer
```

The last hash algorithm used was the MD5 - message-digest algorithm. This module implements the interface to RSA's MD5 message digest algorithm. First, we need to use new() method to create an md5 object and then, feed this object with arbitrary strings using the update() method, in our case the string of bits representing the selected packet headers as input for the hash function. The output will be obtained using the digest() method. As the MD5 algorithm generates an output of 128 bits, we decided to use only the first 4 octets of the output as the hash code. Listing 6.7 presents the implementation of the MD5 algorithm.

Listing 6.7 – Using MD5 - message-digest algorithm Python module to obtain the MD5 hash code.

```
def md5_comp(str_):
    str_ = bytearray(str_)
    m = md5.new()
    m.update(str(str_))
    answer = int(("0x" + m.hexdigest()[:4]),16)
    return answer
```

From the perspective of packet header fields used as input for hash functions, we created three sets of fields: Set 1, Set 2 and Set 3.

The Set 1 of fields includes the unvarying header fields from the IP layer, and the first eight bytes of the layer 4 payload, counted after the layer 4 protocol header (TCP or UDP). The following fields were used as input for the hash function:

ipv4.version;
ipv4.ihl;
ipv4.totalLen;
ipv4.id;
ipv4.flags;
ipv4.fragOffset;
ipv4.protocol;
ipv4.srcAddr;
ipv4.dstAddr;
first 8 bytes after TCP or UDP layer.

Another field selection was Set 2. This selection uses the same unvarying header fields from the IP layer and the first eight bytes of its payload. The payload, in this case, does not depend on the layer 4 protocol used in the packet. It will include the next 8 bytes just after the IP protocol header. The following list shows the fields used for the Set 2 :

ipv4.version;
ipv4.ihl;
ipv4.totalLen;
ipv4.id;
ipv4.flags;
ipv4.fragOffset;
ipv4.protocol;
ipv4.srcAddr;
ipv4.dstAddr;
next 8 bytes after IP layer.

The last variation, Set 3, was a combination of Set 1 and Set 2 payloads. In this

list of fields, we used the same IP layer header fields, but now combining the payload from Set 1 with the payload from Set 2. This results in the fields below:

ipv4.version;
ipv4.ihl;
ipv4.totalLen;
ipv4.id;
ipv4.flags;
ipv4.fragOffset;
ipv4.protocol;
ipv4.srcAddr;
ipv4.dstAddr;
first 8 bytes after TCP or UDP layer;
next 8 bytes after IP layer.

Moreover, we used traffic traces from 3 different sources to perform the analyses for all possible combinations of hash algorithms and input field selection: From CAIDA, NETRESEC and iperf3 traffic.

CAIDA: These traffic traces are from CAIDA's monitors and includes anonymized data from Internet backbone links. The payload has been removed from all packets. More information can be found at http://www.caida.org/data/passive/passive_2012_-dataset.xml.

NETRESEC: This is a list of public packet capture repositories, which are freely available on the Internet. The trace used in tests is available in the repository Bro IDS trace files (no application layer data) at ftp://ftp.bro-ids.org/enterprise-traces/hdr-traces05/.

iperf3: This traffic trace was from traffic generated in a local network between two hosts running iperf3.

The methodology used for this test consisted of calculate the hash code, using different algorithms and sets of fields, for every packet from captures. Then, analyze the number of hash collisions produced while setting the bit in the bitmap corresponding to the [hash algorithm - set of fields] combination, as shown in Table 5.

We created 12 bitmaps, with 65,536 positions each, and processed 10,000 packets from each capture. Each packet was processed 12 times, one per hash algorithm - set of fields combination, and stored in the corresponding bitmap, as per Table 5. This procedure was performed five times, with a different set of 10,000 packets, from each capture. The number of collisions were measured, and an average from five results was used in the analysis.

Figure 26 shows the average of hash collisions after processing five sets of ten thousand packets and store in a bitmap with 65,536 positions, broken down by source of

Table 5 – Different hash algorithm and set of fields used to calculate the packet position in the corresponding bitmap, for each packet.

Hash Algorithm	Set of fields	Bitmap
Checksum 16	Set 1	bitmap_1
Checksum 16	Set 2	bitmap_2
Checksum 16	Set 3	bitmap_3
CRC 16	Set 1	bitmap_4
CRC 16	Set 2	bitmap_5
CRC 16	Set 3	bitmap_6
CRC 32	Set 1	bitmap_7
CRC 32	Set 2	bitmap_8
CRC 32	Set 3	bitmap_9
MD5	Set 1	bitmap_10
MD5	Set 2	bitmap_11
MD5	Set 3	bitmap_12

the trace and the different set of fields selection.

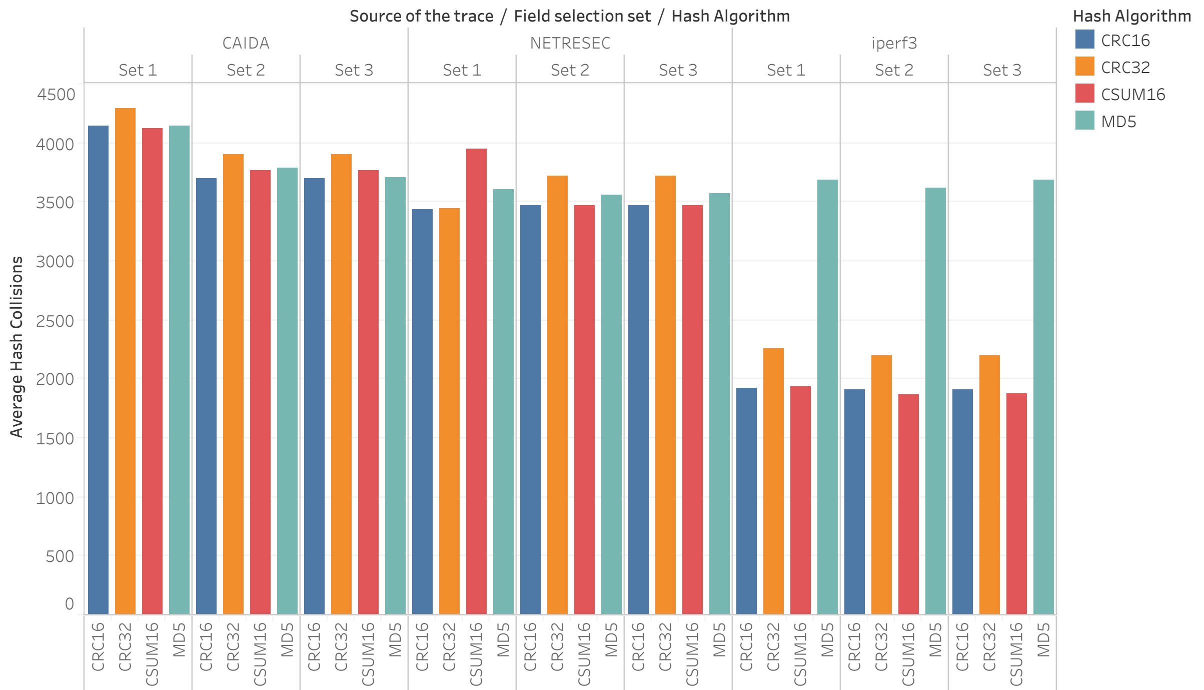

Figure 26 – Average hash collisions per hash algorithm broken down by source of the trace and set of fields selection.

The hash collisions average per hash algorithm, across all traffic traces and field selections is shown in Figure 27, in ascending order. Furthermore, Figure 28 presents the hash collisions average per fields selection, across all traffic traces and hash algorithms.

Observing the results, it is not possible to conclude that any hash algorithm has a better performance on avoiding hash collisions, despite the MD5 algorithm has a higher

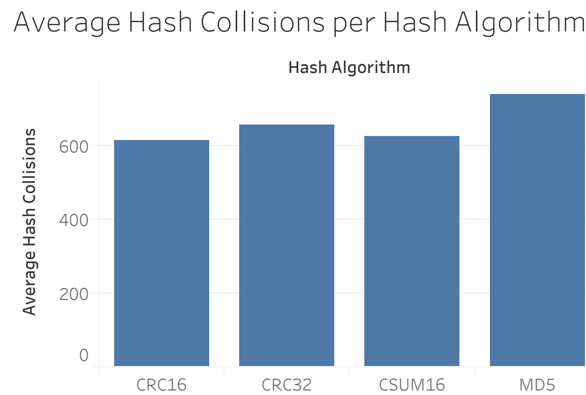

Figure 27 – Average hash collisions per hash algorithm

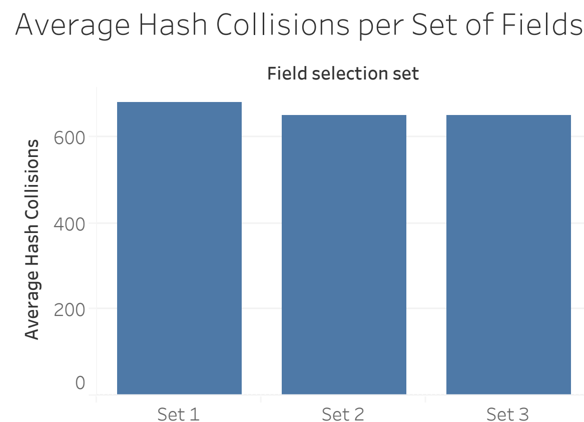

Figure 28 – Average hash collisions set of fields selection.

rate of collisions, when compared to other hash algorithms, for iperf3 traffic trace - as observed in Figure 26 -, the hash collisions average was not different from other traffic traces (CAIDA and NETRESEC). Also, none set of fields selection demonstrate a significant decrease in the average hash collisions rate. Thus, based on tests, the hash algorithm and the field selection do not seem to have a material impact on hash collisions rate. When implementing the BitMatrix on a network device, other factors should be taken into consideration, such as the capabilities for a specific hash algorithm implementation.

6.3 NSF92 Framework in Python

The NSFNET represents one of the more important pieces of Internet history. Staring in 1985, the communication infrastructure initiative, from the National Science Foundation, the NSFNET was the foundation of the United States Internet and the main precursor for the computer network around the world. The NSFNET backbone connected researchers located on university campuses to each other and their counterparts, in universities and research centers around the world.

The partnership that built the NSFNET backbone service also founded a model of technology transfer. From 217 networks connected in 1988 to more than 50,000 in 1995 when the NSFNET backbone service was retired, the NSFNET's growth stimulated the expansion of the Internet and provided an environment for the development of communications technologies. The NSFNET notable history made it one of the most knowledge and studied network model in academia.

To create a more realistic simulated model for the tests, we create a network environment based on the NSFNet from 1992, just after the sixteen T3 sites start operating, in the fall of 1991, and production traffic was phased in. It linked sixteen sites and over 3,500 networks. The new and improved NSFNET backbone service provided the community with networking and connectivity to the fastest production network in the world. Figure 29 shows the locations and T3 links interconnecting them, forming the NSF Network of 1992.

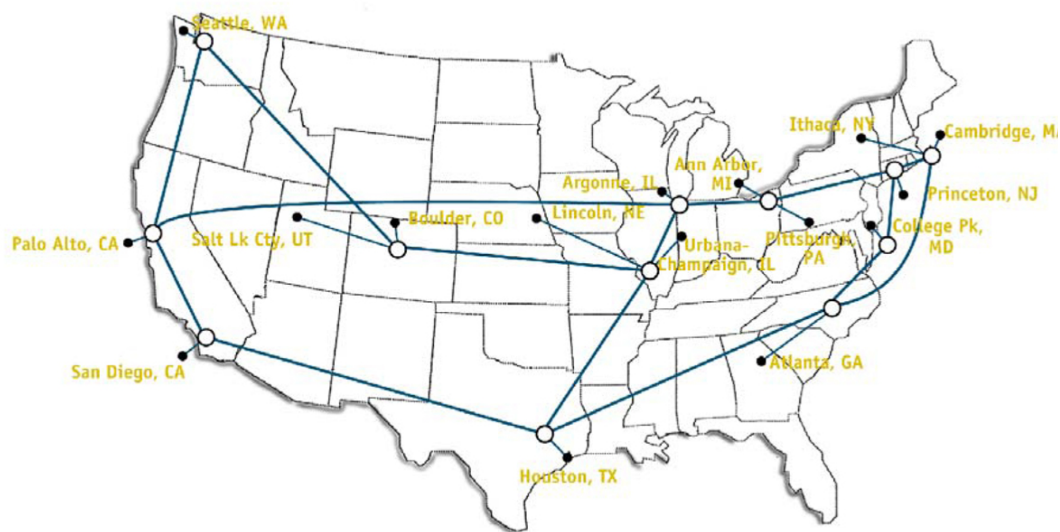

Figure 29 – New T3 Backbone Service for NSFNET 1992

The network diagram in Figure 30 has the same topology as the NSFNET 1992. The location names become tenants, from 1 to 16, representing the 16 sites in the NSF network, and the routers and link are identified with numbers, with a total of 12 routers and 31 links interconnecting the tenants and routers. This model was the network design used in the tests.

Also, we created a routing table to specify the path that a packet needs to travel across the network to go from a source tenant to a destination tenant. Table 9 in Appendix A.5 Routing table for NSF 92 Python framework shows the path composed by routers for every pair of tenants. The same path is used for upstream and downstream traffic. Then, we did not include all the entries for the routing table here.

Once defined the network topology and the routing table, the next step is to

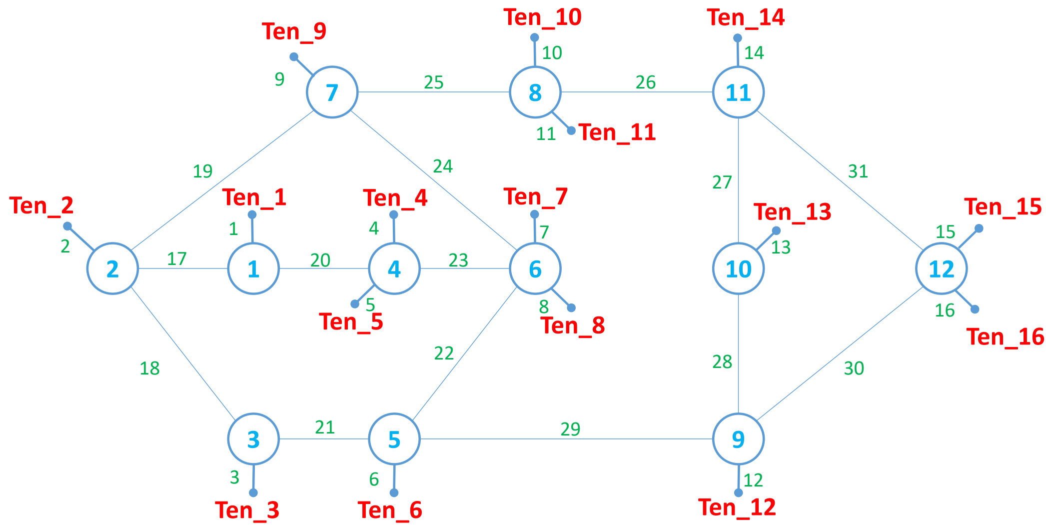

Figure 30 – Topology detail for tenants, routers and links.

determine what traffic to use for the simulation. The CAIDA traffic traces offer a sufficient volume of packets to create relevant traffic for all tenants in the simulated network. Ideally, every tenant should generate a similar volume of traffic, so the traffic load in the network tends to be more balanced across the devices. Therefore, we distributed the packets' source IP address network among the tenants. By analyzing a sample of one million packets from a traffic capture, we created 16 groups of networks, based on packets' source IP addresses in a way that each group of networks has a similar number of packets. Then, we assigned those groups of networks to tenants, one group per tenant. In this manner, each tenant receives networks prefixes that summarize approximately the same number of packets. Table 6 shows the network prefixes assigned to each tenant. Packets with IP address not belonging to any of the listed network prefixes from tenants 1 to 15 were assigned to tenant 16, which is a sort of catch-all tenant.

Tenant	**Network prefixes assigned to the tenant**
Tenant 1	180.0.0.0/8, 128.0.0.0/8
Tenant 2	223.0.0.0/8, 208.0.0.0/8, 55.0.0.0/8, 108.0.0.0/8
Tenant 3	48.1.159.0/24, 151.0.0.0/8, 48.0.0.0/8
Tenant 4	145.0.0.0/8, 158.0.0.0/8, 54.0.0.0/8
Tenant 5	61.0.0.0/8, 184.0.0.0/8, 181.0.0.0/8, 203.0.0.0/8
Tenant 6	48.1.136.0/24, 132.0.0.0/8, 177.0.0.0/8, 186.0.0.0/8, 197.0.0.0/8
Tenant 7	48.1.137.0/24, 83.0.0.0/8, 34.0.0.0/8, 141.0.0.0/8, 116.0.0.0/8
Tenant 8	48.2.0.0/8, 155.0.0.0/8, 49.0.0.0/8, 187.0.0.0/8, 37.0.0.0/8
Tenant 9	135.0.0.0/8, 144.0.0.0/8, 60.0.0.0/8, 220.0.0.0/8, 236.0.0.0/8, 118.0.0.0/8, 113.0.0.0/8

Tenant	Network prefixes assigned to the tenant
Tenant 10	41.0.0.0/8, 142.0.0.0/8, 147.0.0.0/8, 247.0.0.0/8, 50.0.0.0/8, 32.0.0.0/8, 125.0.0.0/8
Tenant 11	178.0.0.0/8, 70.0.0.0/8, 221.0.0.0/8, 148.0.0.0/8, 248.0.0.0/8, 219.0.0.0/8, 152.0.0.0/8, 138.0.0.0/8, 115.0.0.0/8
Tenant 12	53.0.0.0/8, 150.0.0.0/8, 48.1.156.0/24, 201.0.0.0/8, 42.0.0.0/8, 228.0.0.0/8, 68.0.0.0/8, 104.0.0.0/8, 35.0.0.0/8, 85.0.0.0/8
Tenant 13	39.0.0.0/8, 159.0.0.0/8, 183.0.0.0/8, 36.0.0.0/8, 33.0.0.0/8, 112.0.0.0/8, 182.0.0.0/8, 242.0.0.0/8
Tenant 14	143.0.0.0/8, 218.0.0.0/8, 79.0.0.0/8, 78.0.0.0/8, 77.0.0.0/8, 253.0.0.0/8, 254.0.0.0/8, 163.0.0.0/8, 98.0.0.0/8, 109.0.0.0/8, 105.0.0.0/8
Tenant 15	176.0.0.0/8, 40.0.0.0/8, 140.0.0.0/8, 190.0.0.0/8, 149.0.0.0/8, 43.0.0.0/8, 146.0.0.0/8, 231.0.0.0/8, 174.0.0.0/8, 48.1.226.0/24, 80.0.0.0/8, 84.0.0.0/8, 134.0.0.0/8, 131.0.0.0/8, 210.0.0.0/8
Tenant 16	0.0.0.0/0

Table 6 – Tenants and theirs assigned group of network prefixes.

In order to determine how many packets should be processed before collecting statistics and reset the counters, we needed to set some variables. Once the links capacity have being arbitrarily defined in 100Mbps, the question to be responded was: How many packets do the framework need to process to generate satisfactory traffic load in the simulated network, respecting the imposed limit for the links? To answer this question, we created measurements per link, to calculate the average throughput, in Mbps, after processing the number of packets that we considered ten seconds of traffic. So, the only variable now is the number of packets to be processed. By varying the number of packets processed, we can increase or decrease the throughput in the links. Figure 31 shows the final result for determining the number of packets to be processed to create a time dimension in the structure. The framework processed a batch of 430,000 packets, summarizing statistics on every 10 seconds, to generate the graph. With this bound created, we can translate one second in time on every 43,000 packets processed. With this number of packets processed, we can maintain the simulated traffic under the pre-determined link capacity of 100 Mbps.

The next step was to determine the parameters for the sketches' structure creation and collection. The main parameters were the sketches' length and the period for collection (epoch). Both parameters are related, so we start using a period for collection equal to 10 seconds or, on every 430,000 packets. Based on that, we used a length of 65,536 positions for the BitMatrices sketches creation. The main goal here was to maintain a level of BitMatrix occupancy that avoids a high level of hash collisions and, at the same time,

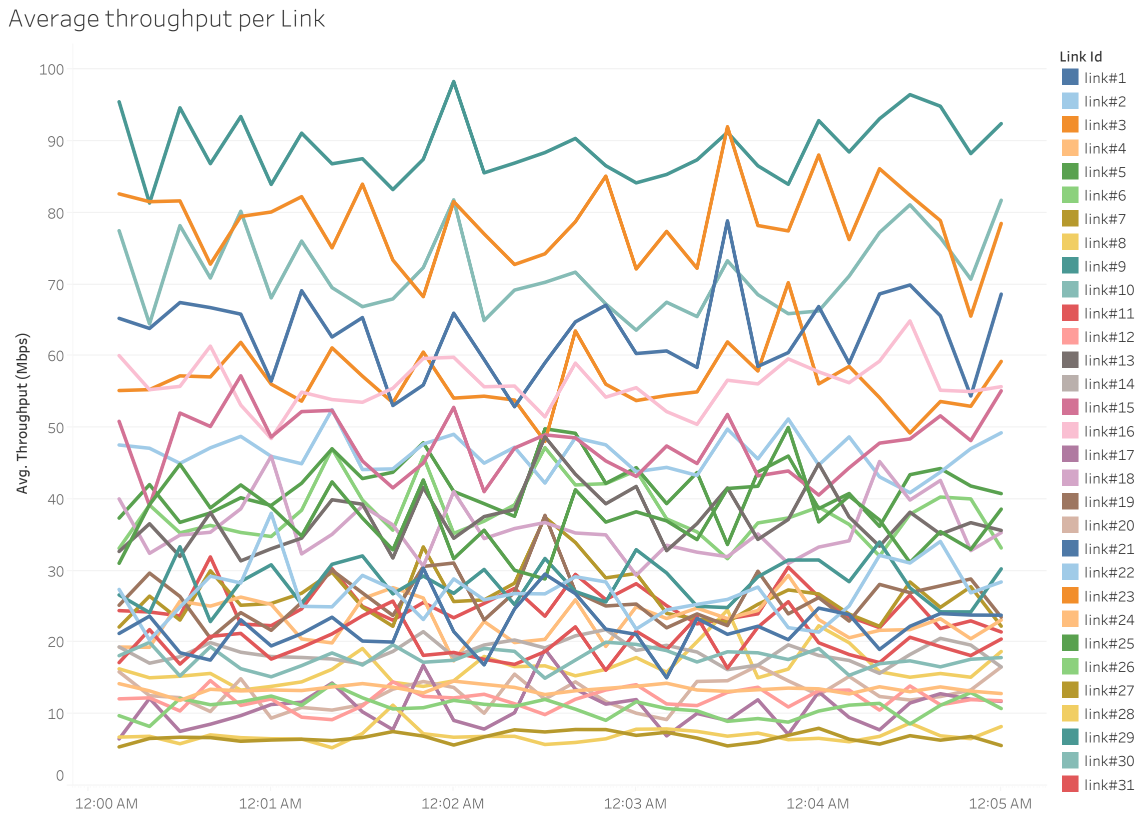

Figure 31 – Average throughput per link on every 10 seconds.

was coherent in terms of memory occupancy and time for collection. Using the described parameter, we created statistics presented in Section 6.3.1 Results.

6.3.1 Results

To analyze the statistics generated by the Python framework, we used the Tableau software. This methodology allows us to create different insights and to validate the information extracted from the BitMatrix with information from counters. We create a sample of 30 minutes of traffic, for traffic analyses. The simulator processed 77,400,000 packets from CAIDA traces, giving a good idea about the BitMatrix solution operation environment.

Figure 32 shows the packets per second processing rate, per router. To calculate this key performance indicator (KPI), we summarized the total number of packets per router from its BitMatrix and divided the number by 10, as the BitMatrix collection occurs on every ten simulated seconds.

Analyzing the graph in Figure 32, the router with higher packets traffic is router 12. We may want to understand what is the traffic on router 12. So, in Figure 33 we can see a graph for that specific router, breaking down the traffic by tenant.

We can identify the high traffic of packets in the router 12, from tenant 3, 15 and

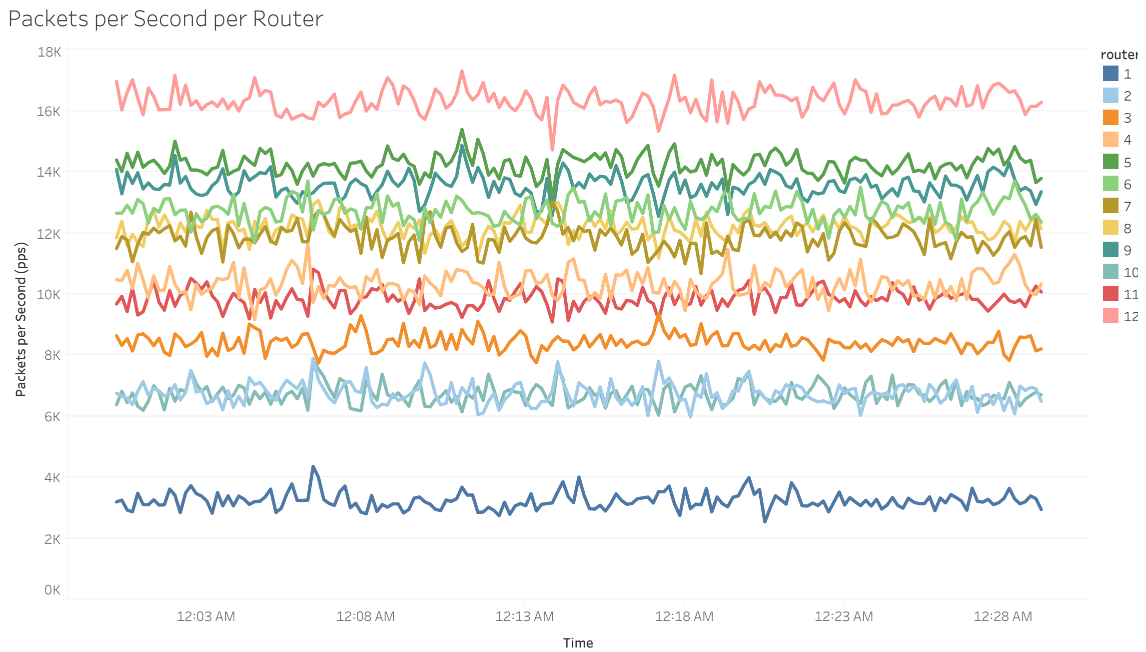

Figure 32 – Packets per second average rate, per router, on every 10 seconds.

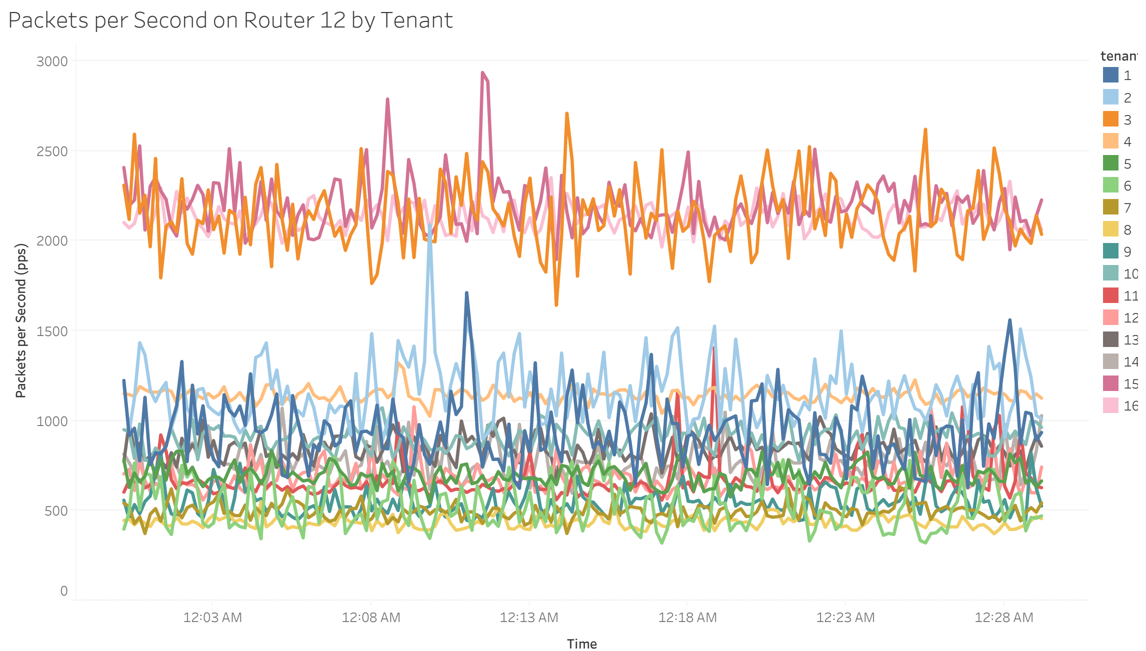

Figure 33 – Packets per second average rate on router 12, per tenant, on every 10 seconds.

16, when compared to other tenants. It is expected that every router receives higher traffic from tenants directly connected to it, thus the unexpected traffic is from tenant 3. Let us analyze the traffic per tenant in the network to better understant each tenant traffic profile. Figure 34 shows traffic per tenant.

The graph in Figure 34 demonstrates the higher traffic from tenant 3, which can be identified as the biggest offender in the perspective of traffic generation in the network. Figure 35 presents a dashboard with a complete view representing tenant traffic

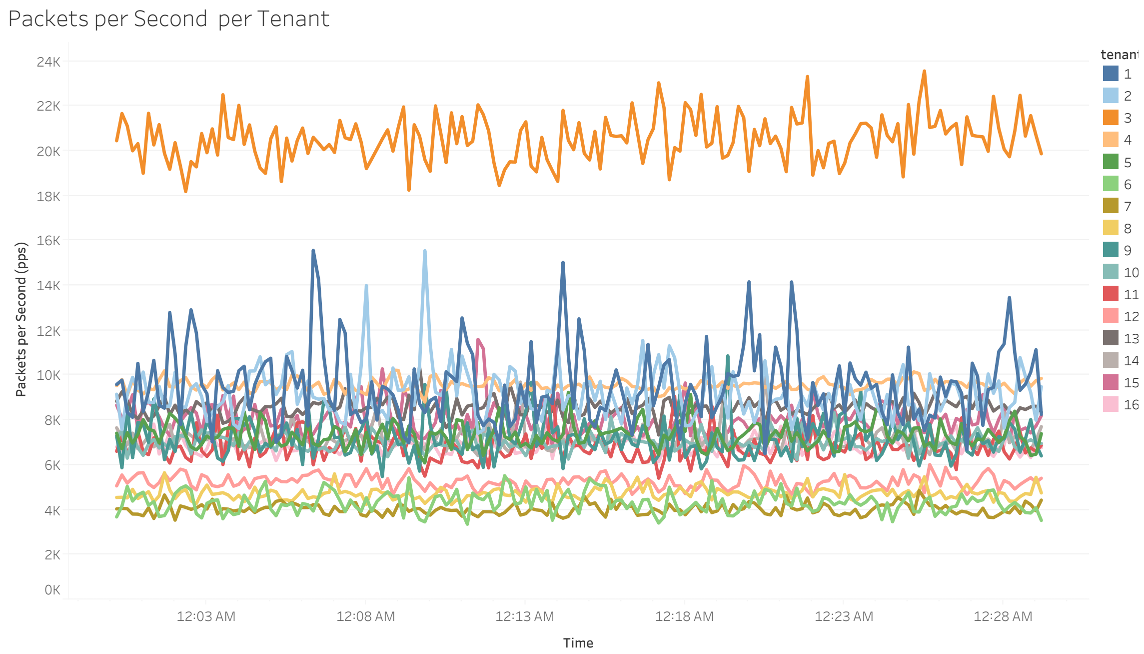

Figure 34 – Packets per second average rate per tenant in the network, on every 10 seconds.

contribution on every router in the period. The percentage is only presented to the tenant with the highest contribution.

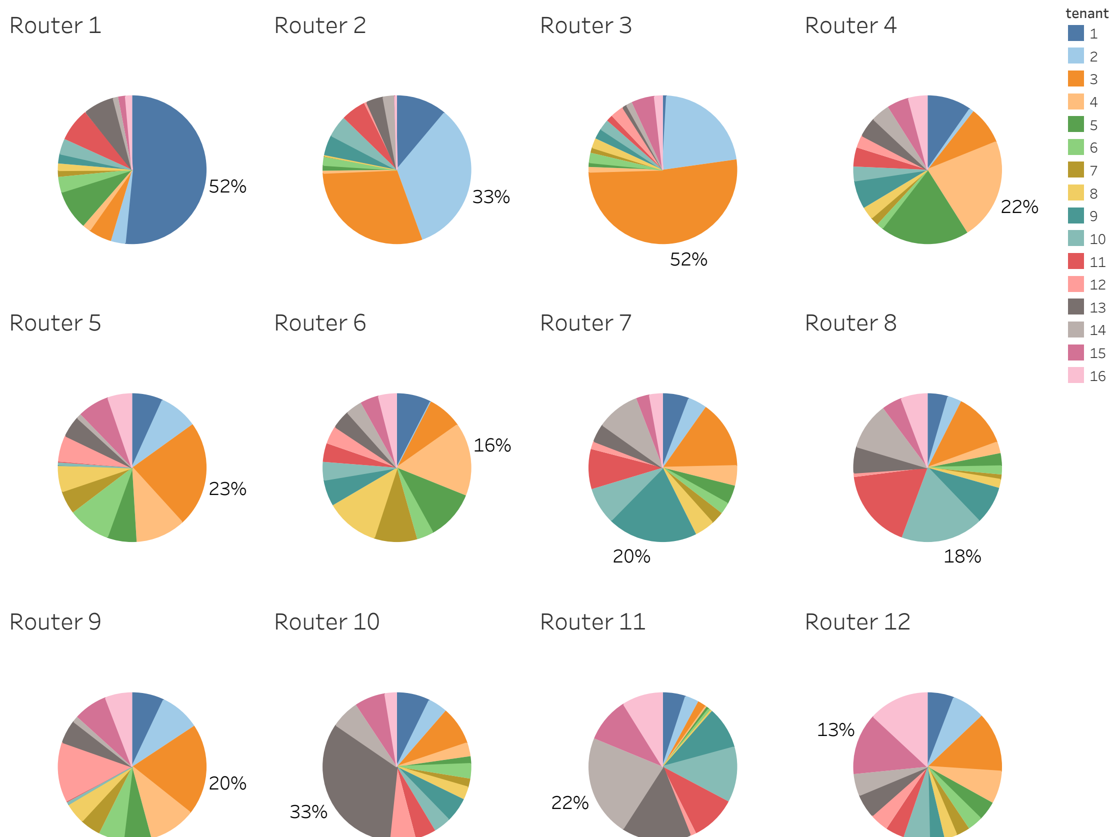

Figure 35 – Dashboard showing the traffic contribution for each tenant per router.

Several other analyses can be performed with the bitstreams collected, as previously described in this work. In this framework implementation, we could comprehend a few insights and better understand the possible analyses that can be done using the packet digested information from BitMatrix. Another crucial point is that, as the information collection occurs every 10 seconds, we can have a near real-time view about network behaviour and performance analysis.

The BitMatrix concept, however, is based on a probabilistic structure of sketches, and hash collisions are part of the process. The hash collisions introduce errors in the measurements performed using the data collected from sketches. In Section 6.4 Machine Learning for BitMatrix Measurement Adjustment, we explore the statistics about collisions, bitmap occupation and propose a model, based on polynomial regression to adjust the results, compensating the losses due to the hash collision.

6.4 Machine Learning for BitMatrix Measurement Adjustment

The Python framework generates one BitMatrix containing 16 bitmaps (one per tenant), per router on every 10 seconds (or, on every 430.000 packets processed). In total, after processing packets to simulate 30 minutes of traffic, it creates about 34,500 bitmaps. The Python framework, besides generating BitMatrices, also was used to create counters to evaluate the quality of the information provided by the BitMatrix structure. When comparing measurements from bitmaps and counters, we notice that there is a gap between these two values.

In Figures 36, 37 and 38 , we can observe the gap between the packets measurement based on BitMatrix and the Packet Counter. The Packet Counter reports the actual number of packets processed and the BitMatrix Counter is the sum of the bits for the corresponding router/tenant bitmap. This difference is due to hash collisions during the process of storing digested packets in some index of the BitMatrix. If the position is already in use, then the interference happens.

We observed that the difference between BitMatrix Counter and Packet Counter increases as more packets are processed. Note that the gap between trend lines in Figure 37 is more significant than in Figure 36, and in Figure 38 is more prominent than in Figure 37. That gap amplification happens as a result of the hash collision probability increasing as more packets are processed. Chances of collision rise as the bitmap get more occupied.

Aiming to find a machine learning algorithm based on the historical data to apply an adjustment to the BitMatrix counter approximating it to the real value, which is the Packet Counter, we used the relation between the percentage of bitmap occupation and the percentage of hash collisions. Figure 39 shows the relation between these two indirect measurements.

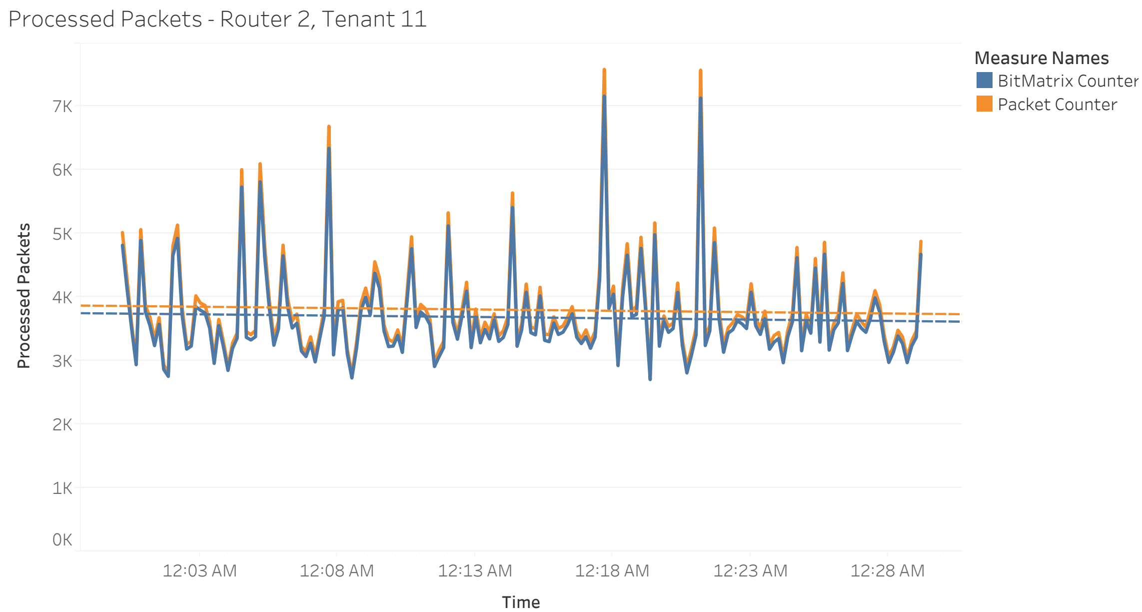

Figure 36 – Number of processed packets measured by the BitMatrix counter and the packet counter for Router 2 and Tenant 11, on every 10 seconds.

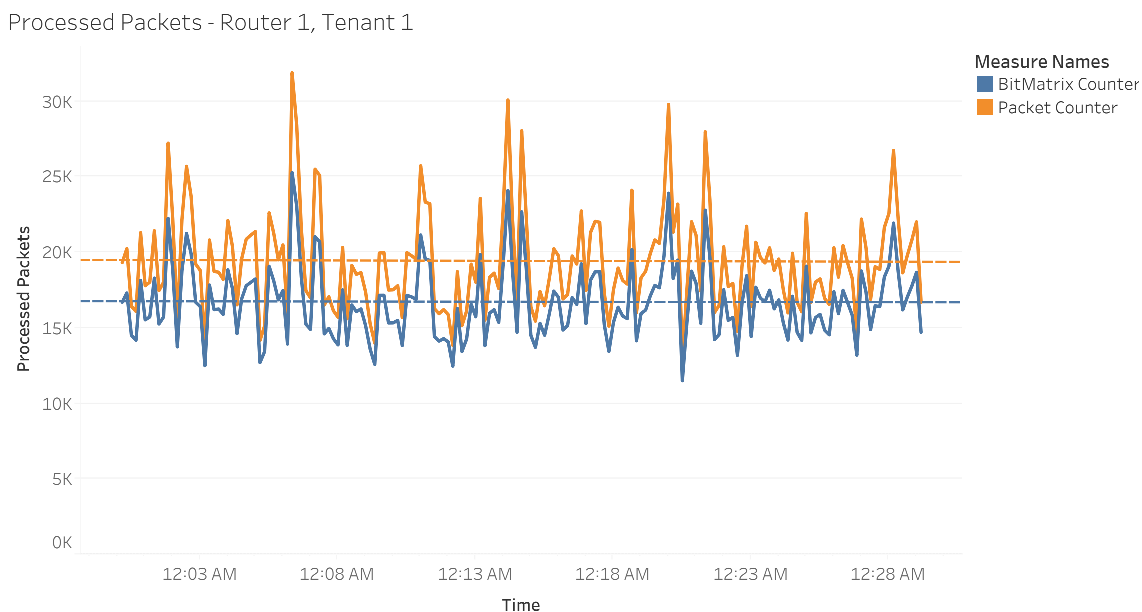

Figure 37 – Number of processed packets measured by the BitMatrix counter and the packet counter for Router 1 and Tenant 1, on every 10 seconds..

For the percentage of bitmap occupation and hash collisions calculation, we used the total length of the bitmap as a reference, as in Equation 6.1 for occupation rate and in Equation 6.2 for collision rate. In this framework, we used a length of 65,536 bits for the BitMatrix; consequently, the bitmap derivative from the BitMatrix will have the same length. We computed the percentage values in Figure 39 using this length value. e.g. If during the process, the framework counted 2,000 hash collisions when storing packets for the same tenant in the BitMatrix, the percentage of collisions will be 2,000 divided

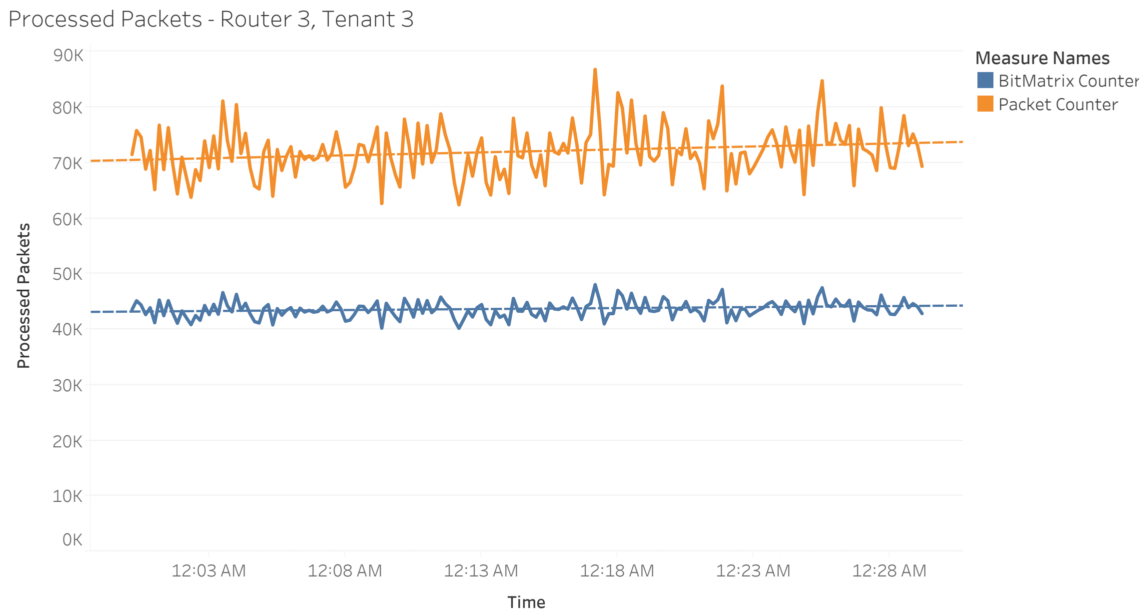

Figure 38 – Number of processed packets measured by the BitMatrix counter and the packet counter for Router 3 and Tenant 3, on every 10 seconds.

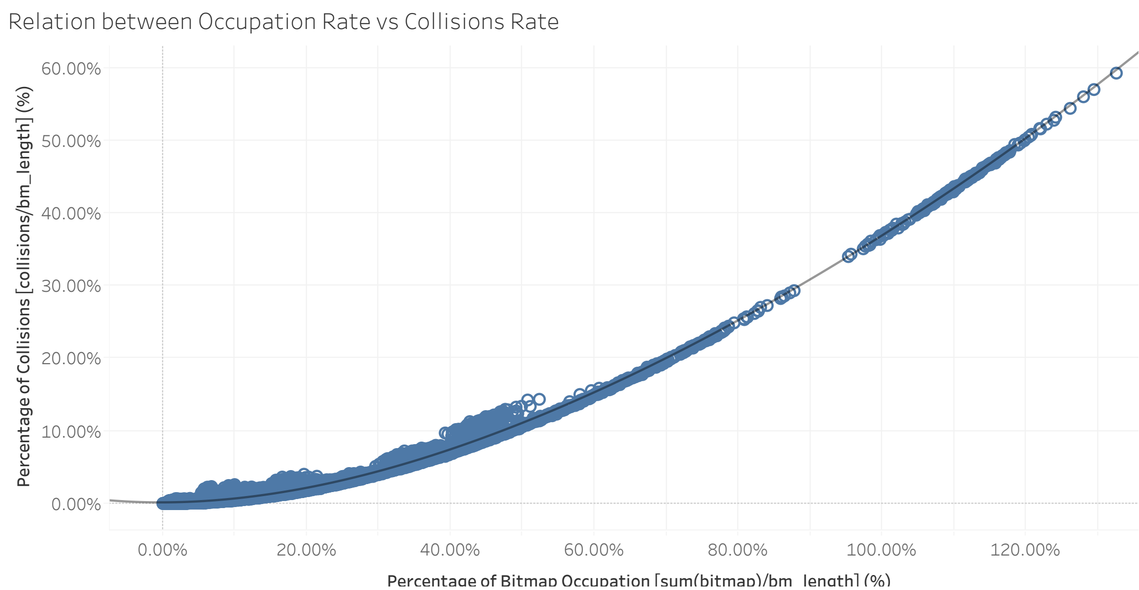

Figure 39 – Percentage of bitmap occupation versus the percentage of collisions, per bitmap.

by 65,536, which is equal to 0.03051758 or 3.051758%. In the same way, for a bitmap occupation of 32,768 positions, the percentage of occupation will be 32,768 divided by 65,536, which is equal to 0.5 or 50%.

$$\%\text{ of occupation} = \sum bitmap/len(bitmap) \tag{6.1}$$

$$\%\text{ of collisions} = \text{number of collision}/len(bitmap) \tag{6.2}$$

Thus, this relation was used to create a model. This model uses the bitmap occupation rate to predict the number of hash collisions that occurred during the BitMatrix population. Using this value to adjust the BitMatrix measurement by adding it to the bitmap sum of bits value, we got the BitMatrix adjusted measurement, which approximates to the real number of packets in Packet Counter measurement.

We explored a limited space of possible hypotheses for the problem, including linear and polynomial regression.The bitmap database used has 33,600 samples and was divided into five partitions of 6,720 samples each, for the k-fold validation. In k-fold validation, we used 4 partitions for training the algorithm and 1 partition for testing. Each field of the database has its index, from 1 to 33,600. The index field was used for field selection, as shown in Table 7.

Table 7 – Database partitioning for k-fold cross validation

	Training set	**Test Set**
Partition 1	[Idx] < 26881	[Idx] > 26880
Partition 2	[Idx] > 6720	[Idx] < 6721
Partition 3	[Idx] < 6721 OR [Idx] > 13440	[Idx] > 6720 AND [Idx] < 13441
Partition 4	[Idx] < 13441 OR [Idx] > 20160	[Idx] > 13440 AND [Idx] < 20161
Partition 5	[Idx] < 20161 OR [Idx] > 26880	[Idx] > 20160 AND [Idx] < 26881

Table 8 demonstrates the results for the k-fold cross-validation process for the used methods. The average mean squared error is the average for the results from the five partitions, and can indicate the best hypothesis to be applied to the problem. The lower the error, better is the result. The Table is sort in ascending order, showing the method with best results first.

Table 8 – Average MSE (mean squared error) and Average STDDEV (standard deviation) for the test database, per method used as hypotheses.

Method	Average Mean Squared Error	Average Std. Deviation
Polynomial Degree 4	1.76E-05	0.039634
Polynomial Degree 3	1.84E-05	0.039674
Polynomial Degree 2	3.66E-05	0.039381
Linear	4.95E-04	0.034805
Logarithmic	1.30E-03	0.025877
Exponential	5.96E-00	2.466112

The best result was the polynomial regression degree 4, and it is described as follow: Model formula: ($\%\text{occupation}^4$ + $\%\text{occupation}^3$ + $\%\text{occupation}^2$ + %occupation + intercept)

Number of modeled observations:	25891
Number of filtered observations:	0
Model degrees of freedom:	5
Residual degrees of freedom (DF):	25886
SSE (sum squared error):	0.330595
MSE (mean squared error):	1.277e-05
R-Squared:	0.99365
Standard error:	0.0035737
p-value (significance):	< 0.0001

Individual trend lines:

Panes		Lines	
Row	Column	P-value	DF
%collision	%occupation	< 0.0001	25886

Coefficients				
Term	Value	StdErr	t-value	p-value
$\%\text{occupation}^4$	0.0600287	0.0045164	13.2912	< 0.0001
$\%\text{occupation}^3$	-0.226063	0.0088914	-25.4249	< 0.0001
$\%\text{occupation}^2$	0.531893	0.0053133	100.105	< 0.0001
%occupation	0.0019715	0.0010979	1.79571	0.0725523
intercept	0.0007011	5.91e-05	11.8629	< 0.0001

The Equation 6.3 expresses the number of collisions as a function of the occupation rate (%occupation).

$$\text{number of collision} = bm_length * (0.0600287 * \%occupation^4 + -0.226063 * \%occupation^3 + 0.531893 * \%occupation^2 + 0.0019715 * \%occupation + 0.000701056) \quad (6.3)$$

Using the number of collisions, resulting from the algorithm, we calculated a BitMatrix adjusted value as a more accurate traffic indicator. Equation 6.4 expresses the

new adjusted value for the traffic.

$$\text{bitmap adjusted} = \text{number of collision} + \sum bitmap \tag{6.4}$$

Figures 40, 41 and 42, present the three values, for comparison:

- BitMatrix Counter: it is the value resulting from summarizing the number of bits in the bitmap.
- Packet Counter: it is the number of packets counted by a counter, created for reference. This is the real number of packet processed for the router, tenant, selected.
- BitMatrix Adjusted it is the value resulting from summarizing the number of bits in the bitmap and adding the calculated delta using the polynomial regression algorithm, based on the bitmap occupation.

The BitMatrix adjusted measurement has an mean absolute percentage error (MAPE) of ±6.14%. It is also possible to observe that even under low or high occupation, the adjusting performance of the algorithm does not degrade.

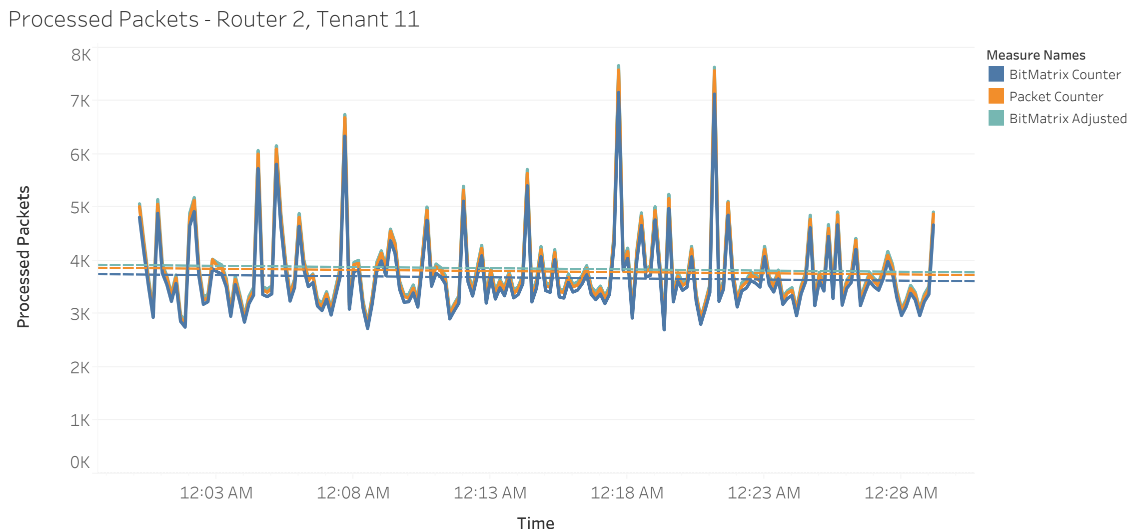

Figure 40 – Number of packets processed measured by the BitMatrix counter, packet counter and the BitMatrix adjusted, for Router 2 and Tenant 11, on every 10 seconds.

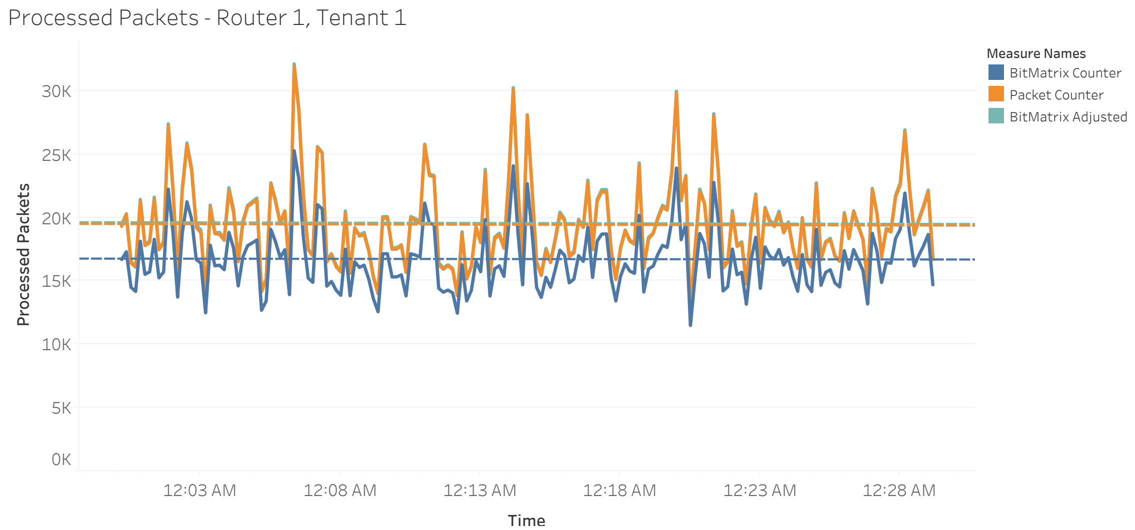

Figure 41 – Number of packets processed measured by the BitMatrix counter, packet counter and the BitMatrix adjusted, for Router 1 and Tenant 1, on every 10 seconds.

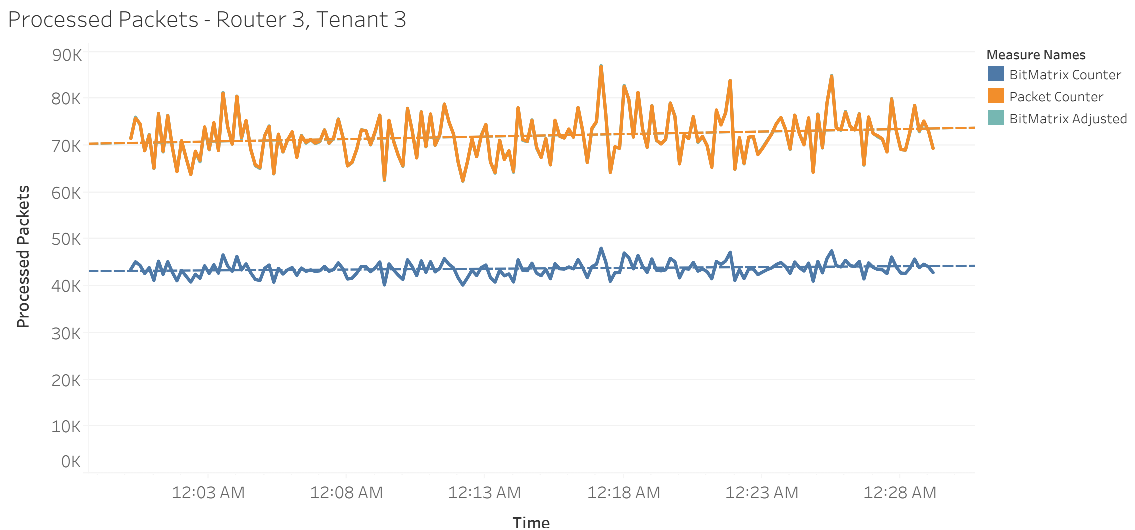

Figure 42 – Number of packets processed measured by the BitMatrix counter, packet counter and the BitMatrix adjusted, for Router 3 and Tenant 3, on every 10 seconds.

7 Concluding remarks

Networking monitoring is a crucial task for a network operator. Information provided by monitoring tools offers intelligence for the decision-making process for capacity planning and traffic engineering. The solution proposed in this work, the BitMatrix framework, goes further than the traditional monitoring process of counting packets and bytes. Besides the general statistics, it enables detail analyses on a packet level, in the network, i.e. it is possible, using the BitMatrix, to determine how many packets travelled through a specific set of network devices. This information cannot be obtained only using counters, making this method the key contribution from this work. Furthermore, using a single sketch, BitMatrix can store digested packet information, segmented by a tenant, enabling the solution to offer the same capabilities in a multi-tenancy network.

The BitMatrix framework, made up of three modules, one composed of sketches, deployed directly in the data plane, and other two, implemented in Python, responsible for the application control plane, collecting information from sketches, storing in a database and post-processing the information to create a broad set of statistics, from a single device packet count to a traffic matrix for the network.

In this work, we implemented BitMatrix sketches using the Behavioral Model v2 P4 soft switch in a Mininet emulated environment to generate traffic statistics, segmented by tenant, observing packets and bytes exchanged between them. For this deployment, the Mininet code required to be adapted to create the emulated network using the P4 target BMv2. The code modification creates a small contribution for the community willing to use the same topology in Mininet for P4 implementations.

For a larger scale statistics generation, we used a framework, written in Python, to process real traffic captures in a simulated network, based on the Network Science Foundation from 1992 producing a considerable number of traffic statistics for routers, broken down by tenant. Moreover, using the generated statistics from this simulation framework, we could create an algorithm, using supervised machine learning, to reduce the errors in statistics introduced by hash collisions, which is another contribution.

Future Work

We plan to keep working in BitMatrix development, improving it in many aspects. The Query and Presentation module would require a more user-friendly interface for creating rules to generate specific metrics from the bitstream information, stored in the database. Another task, aiming to speed access to the statistics, would be the creation

of OLAP multidimensional cubes, using an ETL tool to retrieve the information from bitstreams in the database, aggregating the metrics (number of packets and bytes) by network device and tenant dimensions, rolling up to a 5 minutes, hourly and daily to create a faster access to the statistics by a reporting tool, such as Tableau or MicroStrategy.

We hope to see the following topics being explored in future works:

- Networking Slicing: In this work, the BitMatrix is segmented by tenant. Another possibility is to create a different and/or additional segmentation for the network statistics. Network slicing segmentation would allow us to created different views of the traffic, analyzing different slices of the network, as an independent system.

- Virtual BMv2 instance and bare-metal implementation: In preliminary tests realized in this work, it was possible to create a virtual machine running the BMv2 P4 target. Creating and testing a network using these elements would create not an emulated or simulated as those used in this work, but a more real environment where we could study the overhead in terms of memory and CPU utilization for a BitMatrix deployment.

- There are other projects, besides BMv2, that attempts to enable P4 support for OVS. The popular one is the PISCES project (SHAHBAZ et al., 2016), which is a programmable, protocol-independent soft switch derived from Open vSwitch, whose behaviour can be customized using P4. They are also an excellent field to test BitMatrix implementation.

- Several other programmable network devices, supporting P4, could be used to deploy and test BitMatrix implementation. Network hardware vendors are investing in different architectures such as Protocol-Independent Switch Architecture (PISA), Network Processor Unit (NPU) and Field Programmable Gate Arrays (FPGA). Among others, we can cite NetFPGA-SUME Virtex-7 FPGA development board from Xilinx (INC., 2019), designed in a collaborative effort between Digilent, the University of Cambridge, and Stanford University, supporting P4 and widely used for the research community.

Publications

The following publications are the output from this work:

1. "Using Probabilistic Data Structures for Monitoring of Multi-tenant P4-based Networks", IEEE Symposium on Computers and Communications 2018 .
   Regis F. T. Martins, Fábio L. Verdi, Luis F. U Garcia, Rodolfo S Villaça.

2. "Minicurso - Introdução à Linguagem P4 - Teoria e Prática", Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos 2018.
   Luis F. U. Garcia, Rodolfo S. Villaça, Moises R. N. Ribeiro, Regis F. T. Martins, Fábio L. Verdi, Cesar A. Marcondes.

Bibliography

BABCOCK, B.; OLSTON, C. Distributed top-k monitoring. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 03 2003. Cited in page 38.

BANDI, N. et al. Fast data stream algorithms using associative memories. In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. Beijing, China: ACM, 2007. (SIGMOD '07), p. 247–256. ISBN 978-1-59593-686-8. Disponível em: <http://doi.acm.org/10.1145/1247480.1247510>. Cited in page 38.

BENSON, T. et al. Microte: Fine grained traffic engineering for data centers. In: *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*. Tokyo, Japan: ACM, 2011. (CoNEXT '11), p. 8:1–8:12. ISBN 978-1-4503-1041-3. Disponível em: <http://doi.acm.org/10.1145/2079296.2079304>. Cited in page 35.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2656877.2656890>. Cited 4 times in pages 31, 36, 39, and 80.

BRAVERMAN, V. et al. New bounds for the CLIQUE-GAP problem using graph decomposition theory. *Mathematical Foundations of Computer Science 2015: 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part II*, Springer Berlin Heidelberg, Berlin, Heidelberg, p. 151–162, Jan 2015. ISSN 1432-0541. Disponível em: <https://doi.org/10.1007/978-3-662-48054-0_13>. Cited 2 times in pages 35 and 36.

CASE J. D., F. M. S. M. L.; DAVID, J. R. *A Simple Network Management Protocol (SNMP)*. http://www.ietf.org/rfc/ rfc1157.txt, 1990. RFC 1157. Cited in page 35.

CLAISE, E. B. *Cisco Systems NetFlow Services Export Version 9*. https://www.ietf.org/rfc/rfc3954.txt, 2004. RFC 3954. Cited 2 times in pages 35 and 36.

CORMODE, G.; MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, v. 55, n. 1, p. 58 – 75, 2005. ISSN 0196-6774. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0196677403001913>. Cited in page 36.

DANG, H. T. et al. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 46, n. 2, p. 18–24, maio 2016. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2935634.2935638>. Cited in page 36.

DEMAINE, E. D.; LóPEZ-ORTIZ, A.; MUNRO, J. Frequency estimation of internet packet streams with limited space. In: . [S.l.: s.n.], 2002. v. 2461, p. 348–360. Cited in page 38.

DIMITROPOULOS, X.; HURLEY, P.; KIND, A. Probabilistic lossy counting: An efficient algorithm for finding heavy hitters. *Computer Communication Review*, v. 38, p. 5, 01 2008. Cited in page 36.

DUFFIELD, N.; LUND, C.; THORUP, M. Estimating flow distributions from sampled flow statistics. *IEEE/ACM Trans. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 13, n. 5, p. 933–946, out. 2005. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/TNET.2005.852874>. Cited 3 times in pages 31, 35, and 37.

DUFFIELD, N. G.; GROSSGLAUSER, M. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 9, n. 3, p. 280–292, jun. 2001. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/90.929851>. Cited in page 81.

ESTAN, C.; VARGHESE, G. New directions in traffic measurement and accounting. In: *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. San Francisco, California, USA: ACM, 2001. (IMW '01), p. 75–80. ISBN 1-58113-435-5. Disponível em: <http://doi.acm.org/10.1145/505202.505212>. Cited 2 times in pages 35 and 36.

ESTAN, C.; VARGHESE, G. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems*, v. 21, p. 270–313, 2003. Cited in page 36.

FELDMANN, A. et al. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Trans. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 9, n. 3, p. 265–280, jun. 2001. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/90.929850>. Cited in page 35.

GIBBONS, P. B.; MATIAS, Y. New sampling-based summary statistics for improving approximate query answers. *ACM SIGMOD Record*, v. 27, 10 1999. Cited in page 38.

Guanyao Huang et al. Uncovering global icebergs in distributed monitors. In: *2009 17th International Workshop on Quality of Service*. [S.l.: s.n.], 2009. p. 1–9. ISSN 1548-615X. Cited in page 38.

HALPERN, J. M. et al. *Forwarding and Control Element Separation (ForCES) Protocol Specification*. RFC Editor, 2010. RFC 5810. (Request for Comments, 5810). Disponível em: <https://rfc-editor.org/rfc/rfc5810.txt>. Cited in page 39.

HUANG, Q. et al. Sketchvisor: Robust network measurement for software packet processing. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Los Angeles, CA, USA: ACM, 2017. (SIGCOMM '17), p. 113–126. ISBN 978-1-4503-4653-5. Disponível em: <http://doi.acm.org/10.1145/3098822.3098831>. Cited 2 times in pages 36 and 38.

HUANG, Q.; LEE, P. P. A hybrid local and distributed sketching design for accurate and scalable heavy key detection in network data streams. *Comput. Netw.*, Elsevier North-Holland, Inc., USA, v. 91, n. C, p. 298–315, nov. 2015. ISSN 1389-1286. Disponível em: <https://doi.org/10.1016/j.comnet.2015.08.025>. Cited 2 times in pages 36 and 38.

INC., D. *NetFPGA-SUME Virtex-7 FPGA Development Board*. 2019. <https://store.digilentinc.com/netfpga-sume-virtex-7-fpga-development-board/>. [Online; accessed 15-Aug-2019]. Cited in page 110.

KAMIYAMA, N.; MORI, T. Simple and accurate identification of high-rate flows by packet sampling. In: . [S.l.: s.n.], 2006. Cited in page 38.

KIM, C. et al. In-band Network Telemetry via Programmable Dataplanes. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. Santa Clara, CA, USA: ACM, 2015. (SOSR '15). Cited 2 times in pages 31 and 36.

KRISHNAMURTHY, B. et al. Sketch-based change detection: Methods, evaluation, and applications. In: *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*. Miami Beach, FL, USA: ACM, 2003. (IMC '03), p. 234–247. ISBN 1-58113-773-7. Disponível em: <http://doi.acm.org/10.1145/948205.948236>. Cited 5 times in pages 31, 35, 36, 37, and 38.

KUMAR, A. et al. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *SIGMETRICS Perform. Eval. Rev.*, ACM, New York, NY, USA, v. 32, n. 1, p. 177–188, jun. 2004. ISSN 0163-5999. Disponível em: <http://doi.acm.org/10.1145/1012888.1005709>. Cited 3 times in pages 35, 37, and 38.

LALL, A. et al. Data streaming algorithms for estimating entropy of network traffic. *SIGMETRICS Perform. Eval. Rev.*, ACM, New York, NY, USA, v. 34, n. 1, p. 145–156, jun. 2006. ISSN 0163-5999. Disponível em: <http://doi.acm.org/10.1145/1140103.1140295>. Cited 2 times in pages 35 and 36.

LI, X. et al. Detection and identification of network anomalies using sketch subspaces. In: . [S.l.: s.n.], 2006. p. 147–152. Cited in page 38.

LIU, Z. et al. One sketch to rule them all: Rethinking network flow monitoring with univmon. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. Florianopolis, Brazil: ACM, 2016. (SIGCOMM '16), p. 101–114. ISBN 978-1-4503-4193-6. Disponível em: <http://doi.acm.org/10.1145/2934872.2934906>. Cited 2 times in pages 36 and 37.

MATHEW, R.; KATKAR, V. Survey of low rate dos attack detection mechanisms. In: *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*. Mumbai, Maharashtra, India: ACM, 2011. (ICWET '11), p. 955–958. ISBN 978-1-4503-0449-8. Disponível em: <http://doi.acm.org/10.1145/1980022.1980227>. Cited 3 times in pages 31, 35, and 37.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1355734.1355746>. Cited in page 39.

MITZENMACHER, M.; PAGH, R.; PHAM, N. Efficient estimation for high similarities using odd sketches. In: *Proceedings of the 23rd International Conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery, 2014. (WWW '14), p. 109–118. ISBN 9781450327442. Disponível em: <https://doi.org/10.1145/2566486.2568017>. Cited in page 36.

MOSHREF, M. et al. Scream: Sketch resource allocation for software-defined measurement. In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. Heidelberg, Germany: ACM, 2015. (CoNEXT '15), p. 14:1–14:13. ISBN 978-1-4503-3412-9. Disponível em: <http://doi.acm.org/10.1145/2716281.2836099>. Cited 2 times in pages 36 and 37.

PFAFF, B. et al. The design and implementation of open vswitch. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015. p. 117–130. ISBN 978-1-931971-218. Disponível em: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff>. Cited in page 43.

RAMACHANDRAN, A. et al. Fast monitoring of traffic subpopulations. In: *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*. Vouliagmeni, Greece: ACM, 2008. (IMC '08), p. 257–270. ISBN 978-1-60558-334-1. Disponível em: <http://doi.acm.org/10.1145/1452520.1452551>. Cited 3 times in pages 35, 36, and 37.

SANJUàS-CUXART, J. et al. Sketching the delay: Tracking temporally uncorrelated flow-level latencies. *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, 11 2011. Cited in page 38.

SCHWELLER, R. et al. Reversible sketches for efficient and accurate change detection over network data streams. In: *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*. Taormina, Sicily, Italy: ACM, 2004. (IMC '04), p. 207–212. ISBN 1-58113-821-0. Disponível em: <http://doi.acm.org/10.1145/1028788.1028814>. Cited 5 times in pages 31, 35, 36, 37, and 38.

SFLOW-RT. 2019. [Online; accessed 9-Aug-2019]. Disponível em: <https://sflow-rt.com/>. Cited in page 36.

SHAHBAZ, M. et al. Pisces: A programmable, protocol-independent software switch. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. Florianopolis, Brazil: ACM, 2016. (SIGCOMM '16), p. 525–538. ISBN 978-1-4503-4193-6. Disponível em: <http://doi.acm.org/10.1145/2934872.2934886>. Cited 2 times in pages 36 and 110.

SIVARAMAN, A. et al. Dc.p4: Programming the forwarding plane of a data-center switch. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. Santa Clara, California: ACM, 2015. (SOSR '15), p. 2:1–2:8. ISBN 978-1-4503-3451-8. Disponível em: <http://doi.acm.org/10.1145/2774993.2775007>. Cited in page 36.

SIVARAMAN, V. et al. Heavy-hitter detection entirely in the data plane. In: *Proceedings of the Symposium on SDN Research*. Santa Clara, CA, USA: ACM, 2017. (SOSR '17), p. 164–176. ISBN 978-1-4503-4947-5. Disponível em: <http://doi.acm.org/10.1145/3050220.3063772>. Cited in page 31.

SNOEREN, A. C. et al. Hash-based ip traceback. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 31, n. 4, p. 3–14, ago. 2001. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/964723.383060>. Cited 2 times in pages 45 and 81.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: *Proceedings of the Second ACM SIGCOMM*

Workshop on Hot Topics in Software Defined Networking. New York, NY, USA: ACM, 2013. (HotSDN '13), p. 127–132. ISBN 978-1-4503-2178-5. Disponível em: <http://doi.acm.org/10.1145/2491185.2491190>. Cited in page 39.

The P4 Language Consortium. *The P4 Language Specification - version 1.0.4*. https://p4.org, 2017. v. 2017. Cited 2 times in pages 42 and 67.

TUNE, P.; ROUGHAN, M. Internet Traffic Matrices : A Primer. *Recent Advances in Networking*, p. 108–163, 2013. Disponível em: <http://sigcomm.org/education/ebook/SIGCOMMeBook2013v1{_}chapter3.> Cited 2 times in pages 59 and 60.

WELLEM, T. et al. A flexible sketch-based network traffic monitoring infrastructure. *IEEE Access.*, IEEE Press, Piscataway, NJ, USA, v. 7, p. 92476–92498, jul. 2019. ISSN 2169-3536. Disponível em: <https://ieeexplore.ieee.org/abstract/document/8758822>. Cited 2 times in pages 36 and 37.

XIAO, X. Chapter 11 - the new technical approach. In: XIAO, X. (Ed.). *Technical, Commercial and Regulatory Challenges of QoS*. Boston: Morgan Kaufmann, 2008, (The Morgan Kaufmann Series in Networking). p. 171 – 199. Disponível em: <http://www.sciencedirect.com/science/article/pii/B9780123736932000112>. Cited in page 59.

XIE, Y. et al. Worm origin identification using random moonwalks. In: *2005 IEEE Symposium on Security and Privacy (S P'05)*. Oakland, California, USA: IEEE, 2005. p. 242–256. ISSN 1081-6011. Cited in page 35.

YU, M.; JOSE, L.; MIAO, R. Software defined traffic measurement with opensketch. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. Lombard, IL: USENIX Association, 2013. (NSDI'13), p. 29–42. Disponível em: <http://dl.acm.org/citation.cfm?id=2482626.2482631>. Cited 4 times in pages 31, 35, 36, and 37.

ZHANG, Y. An adaptive flow counting method for anomaly detection in sdn. In: *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*. Santa Barbara, California, USA: ACM, 2013. (CoNEXT '13), p. 25–30. ISBN 978-1-4503-2101-3. Disponível em: <http://doi.acm.org/10.1145/2535372.2535411>. Cited 2 times in pages 35 and 37.

ZHANG, Y. et al. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In: . [S.l.: s.n.], 2004. p. 101–114. Cited in page 38.

ZHAO, Q. G. et al. Data streaming algorithms for accurate and efficient measurement of traffic and flow matrices. In: *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. Banff, Alberta, Canada: ACM, 2005. (SIGMETRICS '05), p. 350–361. ISBN 1-59593-022-1. Disponível em: <http://doi.acm.org/10.1145/1064212.1064258>. Cited in page 38.

APPENDIX A – Appendix

A.1 Deployment in P4 example

Code created as an example of BitMatrix implementation in P4 language, as detailed in Chapter 5.

Listing A.1 – P4_14 code implementing BitMatrix

```
1  //header
2
3  header_type eth_t {
4      fields {
5          dstAddr     : 48;
6          srcAddr     : 48;
7          etherType   : 16;
8      }
9  }
10
11 header eth_t eth;
12
13 header_type ipv4_t {
14     fields {
15         version     : 4;
16         ihl         : 4;
17         diffserv    : 8;
18         totalLen    : 16;
19         id          : 16;
20         flags       : 3;
21         fragOffset  : 13;
22         ttl         : 8;
23         protocol    : 8;
24         hdrChecksum : 16;
25         srcAddr     : 32;
26         dstAddr     : 32;
27         payload8B   : 64;
28     }
29 }
30
31 header ipv4_t ipv4;
32
33 // parser
34
35 parser start {
```

```
36      return parse_eth;
37  }
38
39  #define ETHERTYPE_IPV4          0x0800
40
41  parser parse_eth {
42      extract(eth);
43      return select(latest.etherType) {
44          ETHERTYPE_IPV4      : parse_ipv4;
45          default: ingress;
46      }
47  }
48
49  parser parse_ipv4 {
50      extract(ipv4);
51      return ingress;
52      }
53
54  // field_list definitions
55
56  field_list ipv4_checksum_list {
57      ipv4.version;
58      ipv4.ihl;
59      ipv4.diffserv;
60      ipv4.totalLen;
61      ipv4.id;
62      ipv4.flags;
63      ipv4.fragOffset;
64      ipv4.ttl;
65      ipv4.protocol;
66      ipv4.srcAddr;
67      ipv4.dstAddr;
68  }
69
70  field_list hash_fields {
71      ipv4.version;
72      ipv4.ihl;
73      ipv4.totalLen;
74      ipv4.id;
75      ipv4.flags;
76      ipv4.fragOffset;
77      ipv4.protocol;
78      ipv4.srcAddr;
79      ipv4.dstAddr;
80      ipv4.payload8B;
81  }
82
```

```
83  // defining metadata
84
85  header_type custom_metadata_t {
86      fields {
87          bitmatrix_idx    : 16;
88          bitmatrix_flag   :  2;
89          bitmatrix_tenant :  2;
90          bitmatrix_value  : 20;
91      }
92  }
93
94
95  metadata custom_metadata_t custom_metadata;
96
97
98  header_type routing_metadata_t {
99      fields {
100         nhop_ipv4 : 32;
101         nhop_add : 48;
102     }
103 }
104
105 metadata routing_metadata_t routing_metadata;
106
107 // field_list_calculations
108
109
110 field_list_calculation ipv4_checksum {
111     input {
112         ipv4_checksum_list;
113     }
114     algorithm : csum16;
115     output_width : 16;
116 }
117
118 field_list_calculation hash {
119     input {
120         hash_fields;
121     }
122     algorithm : crc16;
123     output_width : 16;
124 }
125
126 calculated_field ipv4.hdrChecksum  {
127     update ipv4_checksum if (ipv4.ihl == 5);
128 }
129
```

```
130 // register definitions
131
132 register bitmatrix{
133     width : 2;
134     instance_count : 8192;
135 }
136
137 register counter_array_A{
138     width : 20;
139     instance_count : 8192;
140 }
141
142 register counter_array_B{
143     width : 20;
144     instance_count : 8192;
145 }
146
147 // actions
148
149 action _drop() {
150     drop();
151 }
152
153 action set_bitmatrix(tenant_flag) {
154     modify_field_with_hash_based_offset(custom_metadata.
            bitmatrix_idx, 0, hash, 8191);
155     register_read(custom_metadata.bitmatrix_flag, bitmatrix,
            custom_metadata.bitmatrix_idx);
156     bit_or(custom_metadata.bitmatrix_flag, custom_metadata.
            bitmatrix_flag, tenant_flag);
157     register_write(bitmatrix, custom_metadata.bitmatrix_idx,
            custom_metadata.bitmatrix_flag);
158     modify_field(custom_metadata.bitmatrix_tenant, tenant_flag);
159 }
160
161 action set_counter_array_A() {
162     register_read(custom_metadata.bitmatrix_value, counter_array_A
            , custom_metadata.bitmatrix_idx);
163     add_to_field(custom_metadata.bitmatrix_value, ipv4.totalLen);
164     register_write(counter_array_A, custom_metadata.bitmatrix_idx,
             custom_metadata.bitmatrix_value);
165 }
166
167
168 action set_counter_array_B() {
169     register_read(custom_metadata.bitmatrix_value, counter_array_B
            , custom_metadata.bitmatrix_idx);
```

```
170     add_to_field(custom_metadata.bitmatrix_value, ipv4.totalLen);
171     register_write(counter_array_B, custom_metadata.bitmatrix_idx,
            custom_metadata.bitmatrix_value);
172 }
173
174
175 action set_nhop(nhop_ipv4, port) {
176     modify_field(routing_metadata.nhop_ipv4, nhop_ipv4);
177     modify_field(standard_metadata.egress_spec, port);
178     modify_field(ipv4.ttl, ipv4.ttl - 1);
179 }
180
181 action set_dmac(dmac) {
182     modify_field(eth.dstAddr, dmac);
183 }
184
185 action rewrite_mac(smac) {
186     modify_field(eth.srcAddr, smac);
187 }
188
189 // tables
190
191 table set_bitmatrix_table {
192     reads {
193         ipv4.srcAddr : lpm;
194     }
195     actions {
196         set_bitmatrix;
197         _drop;
198     }
199     size: 32;
200 }
201
202 table set_counter_array_A_table {
203     actions {
204         set_counter_array_A;
205     }
206     size : 1;
207 }
208
209 table set_counter_array_B_table {
210     actions {
211         set_counter_array_B;
212     }
213     size : 1;
214 }
215
```

```
216 table ipv4_lpm {
217     reads {
218         ipv4.dstAddr : lpm;
219     }
220     actions {
221         set_nhop;
222         _drop;
223     }
224     size: 1024;
225 }
226
227 table forward {
228     reads {
229         routing_metadata.nhop_ipv4 : exact;
230     }
231     actions {
232         set_dmac;
233         _drop;
234     }
235     size: 512;
236 }
237
238 table send_frame {
239     reads {
240         standard_metadata.egress_port: exact;
241     }
242     actions {
243         rewrite_mac;
244         _drop;
245     }
246     size: 256;
247 }
248
249 // counter definition
250
251 counter pckt_counter {
252     type: packets_and_bytes;
253     direct : set_bitmatrix_table;
254 }
255
256 // control
257
258 control ingress {
259     if(valid(ipv4) and ipv4.ttl > 0) {
260         apply(set_bitmatrix_table);
261         if (custom_metadata.bitmatrix_tenant == 1) {
262             apply(set_counter_array_A_table);
```

```
263             }
264         else {
265                 apply(set_counter_array_B_table);
266             }
267         apply(ipv4_lpm);
268         apply(forward);
269     }
270 }
271
272 control egress {
273     apply(send_frame);
274 }
```

Listing A.2 – P4_14 code for tables feed via Thrift runtime interface

```
table_set_default send_frame _drop
table_set_default forward _drop
table_set_default ipv4_lpm _drop
table_set_default set_bitmatrix_table set_bitmatrix
table_set_default set_counter_array_A_table set_counter_array_A
table_set_default set_counter_array_B_table set_counter_array_B

table_add set_bitmatrix_table set_bitmatrix_0 10.0.0.0/24 => 1
table_add set_bitmatrix_table set_bitmatrix_0 10.0.1.0/24 => 2
table_add ipv4_lpm set_nhop 10.0.0.10/32 => 10.0.0.10 1
table_add ipv4_lpm set_nhop 10.0.1.10/32 => 10.0.1.10 2
table_add forward set_dmac 10.0.0.10 => 00:04:00:00:00:00
table_add forward set_dmac 10.0.1.10 => 00:05:00:00:00:00
table_add send_frame rewrite_mac 1 => 00:aa:bb:00:00:01
table_add send_frame rewrite_mac 2 => 00:aa:bb:00:00:02
```

A.2 Mininet modifications for P4 network emulation

Code created to define the Mininet environment for the tests, as described in chapter 6 Tests and Results, section 6.1 P4 implementation using Mininet.

Listing A.3 – Mininet enviromnet defined by topo.py

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.topo import Topo
from mininet.log import setLogLevel, info
from mininet.cli import CLI
```

```
from mininet.link import TCLink
from mininet.link import TCIntf

from p4_mininet import P4Switch, P4Host

import argparse
from time import sleep
import os
import subprocess

_THIS_DIR = os.path.dirname(os.path.realpath(__file__))
_THRIFT_BASE_PORT = 22222

parser = argparse.ArgumentParser(description='Mininet demo')
parser.add_argument('--behavioral-exe', help='Path to behavioral
    executable',
                    type=str, action="store", required=True)
parser.add_argument('--json', help='Path to JSON config file',
                    type=str, action="store", required=True)
parser.add_argument('--cli', help='Path to BM CLI',
                    type=str, action="store", required=True)

args = parser.parse_args()

class MyTopo(Topo):
    def __init__(self, sw_path, json_path, nb_hosts, nb_switches,
        links, **opts):
        # Initialize topology and default options
        Topo.__init__(self, **opts)

        for i in xrange(nb_switches):
            switch = self.addSwitch('s%d' % (i + 1),
                                    sw_path = sw_path,
                                    json_path = json_path,
                                    thrift_port =
                                        _THRIFT_BASE_PORT + i,
                                    pcap_dump = True,
                                    device_id = i)
                                    #enable_debugger = True,
                                    #log_console = True)

        for h in xrange(nb_hosts):
            host = self.addHost('h%d' % (h + 1),
                                ip = "10.0.%d.10/24" % h,
                                mac = '00:04:00:00:00:%02x' % h)

        i = 0
```

```
        for a, b in links:
            self.addLink(a, b,
                         addr1 = '00:aa:bb:00:00:%02d' % i,
                         addr2 = '00:aa:bb:00:00:%02d' % (i + 1),
                         bw=1)
            i += 2

def read_topo():
    nb_hosts = 0
    nb_switches = 0
    links = []
    with open("topo.txt", "r") as f:
        line = f.readline()[:-1]
        w, nb_switches = line.split()
        assert(w == "switches")
        line = f.readline()[:-1]
        w, nb_hosts = line.split()
        assert(w == "hosts")
        for line in f:
            if not f: break
            a, b = line.split()
            links.append( (a, b) )
    return int(nb_hosts), int(nb_switches), links


def main():
    nb_hosts, nb_switches, links = read_topo()

    topo = MyTopo(args.behavioral_exe,
                  args.json,
                  nb_hosts, nb_switches, links)

    net = Mininet(topo = topo,
                  host = P4Host,
                  switch = P4Switch,
                  link = TCLink,
                  controller = None )
    net.start()

    sw_mac = ["00:aa:bb:00:00:%02x" % n for n in xrange(nb_hosts)]

    sw_addr = ["10.0.%d.1" % n for n in xrange(nb_hosts)]

    for n in xrange(nb_hosts):
        h = net.get('h%d' % (n + 1))
        print "**********"
        print "Hostname: %s" %(h.name)
```

```
    for off in ["rx", "tx", "sg"]:
        cmd = "/sbin/ethtool --offload eth0 %s off" % off
        print cmd
        h.cmd(cmd)
    print "disable ipv6"
    h.cmd("sysctl -w net.ipv6.conf.all.disable_ipv6=1")
    h.cmd("sysctl -w net.ipv6.conf.default.disable_ipv6=1")
    h.cmd("sysctl -w net.ipv6.conf.lo.disable_ipv6=1")
    h.cmd("sysctl -w net.ipv4.tcp_congestion_control=reno")
    h.cmd("iptables -I OUTPUT -p icmp --icmp-type destination-
       unreachable -j DROP")
    h.describe()
    print "**********"


sw_addr = ["10.0.%d.1" % n for n in xrange(nb_hosts)]

s = net.get('s1')
sw_mac_s1_eth1 = s.intf("s1-eth1").MAC()
h = net.get('h1')
h.setARP(sw_addr[0], sw_mac_s1_eth1)
h.setDefaultRoute("dev eth0 via %s" % sw_addr[0])

s = net.get('s3')
sw_mac_s3_eth1 = s.intf("s3-eth1").MAC()
h = net.get('h2')
h.setARP(sw_addr[1], sw_mac_s3_eth1)
h.setDefaultRoute("dev eth0 via %s" % sw_addr[1])

s = net.get('s4')
sw_mac_s4_eth1 = s.intf("s4-eth1").MAC()
h = net.get('h3')
h.setARP(sw_addr[2], sw_mac_s4_eth1)
h.setDefaultRoute("dev eth0 via %s" % sw_addr[2])


sleep(1)

for i in xrange(nb_switches):
    s = net.get('s%d' % (i + 1))
    print "**********"
    print "Switch Name: %s" %(s.name)
    print "Switch DPID: %s" %(s.dpid)
    for j in s.ports:
        print "port: %s - intf: %s - mac: %s" %(
            s.ports[s.intf(j)],
            j,
```

```
                    s.intf(str(j)).MAC()
                )
            print "Running command_s%d.txt" % (i + 1)
            cmd = [args.cli, "--json", args.json,
                   "--thrift-port", str(_THRIFT_BASE_PORT + i)]
            with open("command_s%d.txt" % (i + 1), "r") as f:
                print " ".join(cmd)
                try:
                    output = subprocess.check_output(cmd, stdin = f)
                    #print output
                except subprocess.CalledProcessError as e:
                    print e
                    print e.output
            print "**********"

    sleep(1)

    print "Ready !"

    CLI( net )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    main()
```

A.3 Collector and Controller component implemented in Python

Code created to collect information from the P4 network devices, as described in chapter 4 Monitoring Framework, section 4.2.2 Collection and Control. Also, this code perform the tenants' bitmap extraction from the BitMatrix, summarize the statistics and send all the information to Graphite platform (https://graphiteapp.org) that is a platform for store numeric time-series data and render graphs of this data on demand.

Listing A.4 – Collector and Controller component implemented in Python

```
#!/usr/bin/env python2.7

import time
import threading
import bmpy_utils as utils
from bm_runtime.standard import Standard
import socket
import schedule
```

```
sw_list = [
    ["localhost", 22222],
    ["localhost", 22223],
    ["localhost", 22224],
    ["localhost", 22225]
    ]

matrix_size = 32768
connections = [[] for _ in range(len(sw_list))]
results = [[] for _ in range(len(sw_list))]
services = [("standard", Standard.Client)] + [(None , None)]

def connections_setup():
    for i in xrange(0,len(sw_list)):
        globals()['std_client_s{}'.format(i + 1)], globals()['
            mc_client_s{}'.format(i + 1)] = utils.thrift_connect(
             sw_list[(i)][0],
             sw_list[(i)][1],
             services)
        connections[i] = globals()['std_client_s{}'.format(i + 1)]
        globals()['result_s{}'.format(i + 1)] = []
        results[i] = globals()['result_s{}'.format(i + 1)]

def collect_switch_info(self, result, bitmatrix, counter_arr):
    bitmatrix_pointer = self.bm_register_read(0, '
        bitmatrix_pointer', 0)
    if bitmatrix_pointer == 0:
        register_name = "bitmatrix_0"
        counter_array_A = "counter_array_0A"
        counter_array_B = "counter_array_0B"
        counter_array_C = "counter_array_0C"
        counter_array_D = "counter_array_0D"
        counter_table = "set_bitmatrix_0_table"
        self.bm_register_write(0, 'bitmatrix_pointer', 0, 1)
    else:
        register_name = "bitmatrix_1"
        counter_array_A = "counter_array_1A"
        counter_array_B = "counter_array_1B"
        counter_array_C = "counter_array_1C"
        counter_array_D = "counter_array_1D"
        counter_table= "set_bitmatrix_1_table"
        self.bm_register_write(0, 'bitmatrix_pointer', 0, 0)
    counter_name = "pckt_counter"
    packet_counter = int(self.bm_counter_read(0, counter_name, 0).
        packets)
    bytes_counter = int(self.bm_counter_read(0, counter_name, 0).
        bytes)
```

```
self.bm_counter_reset_all(0, counter_name)

bitmatrix_snapshot = []
counter_A_snap = []
counter_B_snap = []
counter_C_snap = []
counter_D_snap = []
counter_table_snap = []
start = time.time()

bitmatrix_snapshot.extend(self.bm_register_read_all(0,
   register_name))
counter_A_snap.extend(self.bm_register_read_all(0,
   counter_array_A))
counter_B_snap.extend(self.bm_register_read_all(0,
   counter_array_B))
counter_C_snap.extend(self.bm_register_read_all(0,
   counter_array_C))
#counter_D_snap.append(self.bm_register_read(0,
   counter_array_D, idx))
for idx in range(4):
    counter_table_snap.append(self.bm_mt_read_counter(0,
       counter_table, idx).packets)
    counter_table_snap.append(self.bm_mt_read_counter(0,
       counter_table, idx).bytes)
self.bm_register_reset(0, register_name)
self.bm_register_reset(0, counter_array_A)
self.bm_register_reset(0, counter_array_B)
self.bm_register_reset(0, counter_array_C)
self.bm_register_reset(0, counter_array_D)
self.bm_mt_reset_counters(0, counter_table)
end = time.time()
retrieve_time = (end - start)
occupation = 100*(float(packet_counter) / (matrix_size * 4))
info = self.bm_mgmt_get_info().device_id
result.extend([info, start, packet_counter, occupation,
   retrieve_time,
               counter_table_snap[0], counter_table_snap[1],
               counter_table_snap[2], counter_table_snap[3],
               counter_table_snap[4], counter_table_snap[5],
               counter_table_snap[6], counter_table_snap[7]])
bitmatrix.append(bitmatrix_snapshot)
counter_arr.extend([counter_A_snap,
                    counter_B_snap,
                    counter_C_snap,
                    counter_D_snap])
```

```
    print ("Finish collecting from sw%s at: %s" %(str(info), time.
        strftime("%H:%M:%S")))

def tenant_breakout(bitmatrix):
    tenant = [[] for _ in range(4)]
    for flag in bitmatrix[0]:
        flag = int(flag)
        flag = "{0:04b}".format(flag)
        tenant[3].append(int(flag[0]))
        tenant[2].append(int(flag[1]))
        tenant[1].append(int(flag[2]))
        tenant[0].append(int(flag[3]))
    bitmatrix.extend([tenant[0], tenant[1], tenant[2], tenant[3]])

def counters_sw_tenant(result, bitmatrix, counter_arr):
    pkt_counter_ten_a = sum(bitmatrix[1])
    pkt_counter_ten_b = sum(bitmatrix[2])
    pkt_counter_ten_c = sum(bitmatrix[3])
    pkt_counter_ten_d = sum(bitmatrix[4])
    total_pckt_counter = sum([pkt_counter_ten_a,
                              pkt_counter_ten_b,
                              pkt_counter_ten_c,
                              pkt_counter_ten_d])
    bytes_counter_ten_a = sum(counter_arr[0])
    bytes_counter_ten_b = sum(counter_arr[1])
    bytes_counter_ten_c = sum(counter_arr[2])
    bytes_counter_ten_d = sum(counter_arr[3])

    if result [2] == 0:
        collision = 0
    else:
        collision =  100*(1 - (float(total_pckt_counter) / result
            [2]))

    if result [5] == 0:
        collision_a = 0
    else:
        collision_a =  100*(1 - (float(pkt_counter_ten_a) / result
            [5]))

    if result [7] == 0:
        collision_b = 0
    else:
        collision_b =  100*(1 - (float(pkt_counter_ten_b) / result
            [7]))

    if result [9] == 0:
```

```
            collision_c= 0
        else:
            collision_c =  100*(1 - (float(pkt_counter_ten_c) / result
                [9]))

        if result [11] == 0:
            collision_d = 0
        else:
            collision_d =  100*(1 - (float(pkt_counter_ten_d) / result
                [11]))

        occupation_a = 100*(float(pkt_counter_ten_a) / matrix_size)
        occupation_b = 100*(float(pkt_counter_ten_b) / matrix_size)
        occupation_c = 100*(float(pkt_counter_ten_c) / matrix_size)
        occupation_d = 100*(float(pkt_counter_ten_d) / matrix_size)

        result.extend([total_pckt_counter,
                       collision,
                       pkt_counter_ten_a,
                       occupation_a,
                       collision_a,
                       bytes_counter_ten_a,
                       pkt_counter_ten_b,
                       occupation_b,
                       collision_b,
                       bytes_counter_ten_b,
                       pkt_counter_ten_c,
                       occupation_c,
                       collision_c,
                       bytes_counter_ten_c,
                       pkt_counter_ten_d,
                       occupation_d,
                       collision_d,
                       bytes_counter_ten_d])
        print result

    def count_routes(route_counter, bitmatrix, counter_arr):
        counter_AB_pkt = 0
        counter_AC_pkt = 0
        counter_BA_pkt = 0
        counter_BC_pkt = 0
        counter_CA_pkt = 0
        counter_CB_pkt = 0
        counter_AB_bytes = 0
        counter_AC_bytes = 0
        counter_BA_bytes = 0
        counter_BC_bytes = 0
```

```
counter_CA_bytes = 0
counter_CB_bytes = 0

for i in range(matrix_size):
    # Tenant A
    if bitmatrix[0][1][i] == 1:
        if bitmatrix[2][1][i] == 1:
            counter_AB_pkt += 1
            counter_AB_bytes += counter_arr[0][0][i]
        elif bitmatrix[3][1][i] == 1:
            if bitmatrix[1][1][i] == 1:
                counter_AC_pkt += 1
                counter_AC_bytes += counter_arr[0][0][i]
            else:
                pass
        else:
            pass
    else:
        pass
    # Tenant B
    if bitmatrix[2][2][i] == 1:
        if bitmatrix[0][2][i] == 1:
            counter_BA_pkt += 1
            counter_BA_bytes += counter_arr[2][1][i]
        elif bitmatrix[3][2][i] == 1:
            counter_BC_pkt += 1
            counter_BC_bytes += counter_arr[2][1][i]
        else:
            pass
    else:
        pass
    # Tenant C
    if bitmatrix[3][3][i] == 1:
        if bitmatrix[2][3][i] == 1:
            counter_CB_pkt += 1
            counter_BA_bytes += counter_arr[3][2][i]
        elif bitmatrix[0][3][i] == 1:
            if bitmatrix[1][3][i] == 1:
                counter_CA_pkt += 1
                counter_BA_bytes += counter_arr[3][2][i]
            else:
                pass
        else:
            pass
    else:
        pass
route_counter.extend([counter_AB_pkt, counter_AC_pkt,
```

```
                                counter_BA_pkt , counter_BC_pkt ,
                                counter_CA_pkt , counter_CB_pkt ,
                                counter_AB_bytes , counter_AC_bytes ,
                                counter_BA_bytes , counter_BC_bytes ,
                                counter_CA_bytes , counter_CB_bytes])


def post_graphite(content):
    s = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
    s.connect(('192.168.25.8', 2003))
    s.sendall(content)
    s.close()


def sched_job():
    schedule.every().second.do(collect_cycle)
    return schedule.CancelJob


def collect_cycle():

    bitmatrix = [[] for _ in range(len(sw_list))]
    counter_arr = [[] for _ in range(len(sw_list))]
    results = [[] for _ in range(len(sw_list))]
    route_counter = []

    print ("Start collecting at: %s" %(time.strftime("%H:%M:%S")))
    start_collect_time = time.time()

    threads = []

    for i in range(4):
        t = threading.Thread(target=collect_switch_info , args=(
            connections[i],
                                                                    results
                                                                       [
                                                                       i
                                                                       ],

                                                                    bitmatrix
                                                                       [
                                                                       i
                                                                       ],

                                                                    counter_arr
                                                                       [
                                                                       i
```

```
                                                                   ])
                                                                   )

    threads.append(t)
    t.start()

while threading.activeCount()>1:
    time.sleep(1)

for bitmatrix_sX in bitmatrix:
    tenant_breakout(bitmatrix_sX)

for i in range(4):
    counters_sw_tenant(results[i],
                       bitmatrix[i],
                       counter_arr[i])

count_routes(route_counter, bitmatrix, counter_arr)

print route_counter

print ("Finish collecting at: %s" %(time.strftime("%H:%M:%S"))
   )

for i in xrange (0,4):
    post_graphite("sw%d.counter %s %s\n" %(i+1, results[i][2],
        results[i][1]))
    post_graphite("sw%d.occup_pct %s %s\n" %(i+1, results[i
       ][3], results[i][1]))
    post_graphite("sw%d.time2colletc %s %s\n" %(i+1, results[i
       ][4], results[i][1]))
    post_graphite("sw%d.ten_A_counter_pckt %s %s\n" %(i+1,
       results[i][5], results[i][1]))
    post_graphite("sw%d.ten_A_counter_bytes %s %s\n" %(i+1,
       results[i][6], results[i][1]))
    post_graphite("sw%d.ten_B_counter_pckt %s %s\n" %(i+1,
       results[i][7], results[i][1]))
    post_graphite("sw%d.ten_B_counter_bytes %s %s\n" %(i+1,
       results[i][8], results[i][1]))
    post_graphite("sw%d.ten_C_counter_pckt %s %s\n" %(i+1,
       results[i][9], results[i][1]))
    post_graphite("sw%d.ten_C_counter_bytes %s %s\n" %(i+1,
       results[i][10], results[i][1]))
    post_graphite("sw%d.ten_D_counter_pckt %s %s\n" %(i+1,
       results[i][11], results[i][1]))
    post_graphite("sw%d.ten_D_counter_bytes %s %s\n" %(i+1,
       results[i][12], results[i][1]))
```

```
        post_graphite("sw%d.sum_bitmtx_pckts %s %s\n" %(i+1,
            results[i][13], results[i][1]))
        post_graphite("sw%d.collision_pct %s %s\n" %(i+1, results[
            i][14], results[i][1]))
        post_graphite("sw%d.pkt_counter_A %s %s\n" %(i+1, results[
            i][15], results[i][1]))
        post_graphite("sw%d.occup_pct_A %s %s\n" %(i+1, results[i
            ][16], results[i][1]))
        post_graphite("sw%d.collision_A %s %s\n" %(i+1, results[i
            ][17], results[i][1]))
        post_graphite("sw%d.bm_bytes_counter_A %s %s\n" %(i+1,
            results[i][18], results[i][1]))
        post_graphite("sw%d.pkt_counter_B %s %s\n" %(i+1, results[
            i][19], results[i][1]))
        post_graphite("sw%d.occup_pct_B %s %s\n" %(i+1, results[i
            ][20], results[i][1]))
        post_graphite("sw%d.collision_B %s %s\n" %(i+1, results[i
            ][21], results[i][1]))
        post_graphite("sw%d.bm_bytes_counter_B %s %s\n" %(i+1,
            results[i][22], results[i][1]))
        post_graphite("sw%d.pkt_counter_C %s %s\n" %(i+1, results[
            i][23], results[i][1]))
        post_graphite("sw%d.occup_pct_C %s %s\n" %(i+1, results[i
            ][24], results[i][1]))
        post_graphite("sw%d.collision_C %s %s\n" %(i+1, results[i
            ][25], results[i][1]))
        post_graphite("sw%d.bm_bytes_counter_C %s %s\n" %(i+1,
            results[i][26], results[i][1]))
        post_graphite("sw%d.pkt_counter_D %s %s\n" %(i+1, results[
            i][27], results[i][1]))
        post_graphite("sw%d.occup_pct_D %s %s\n" %(i+1, results[i
            ][28], results[i][1]))
        post_graphite("sw%d.collision_D %s %s\n" %(i+1, results[i
            ][29], results[i][1]))
        post_graphite("sw%d.bm_bytes_counter_D %s %s\n" %(i+1,
            results[i][30], results[i][1]))

    post_graphite("traffic.A->B_pkts %s %s\n" %(route_counter[0],
        results[0][1]))
    post_graphite("traffic.A->C_pkts %s %s\n" %(route_counter[1],
        results[0][1]))
    post_graphite("traffic.B->A_pkts %s %s\n" %(route_counter[2],
        results[2][1]))
    post_graphite("traffic.B->C_pkts %s %s\n" %(route_counter[3],
        results[2][1]))
    post_graphite("traffic.C->A_pkts %s %s\n" %(route_counter[4],
        results[3][1]))
```

```
    post_graphite("traffic.C->B_pkts %s %s\n" %(route_counter[5],
       results[3][1]))
    post_graphite("traffic.A->B_bytes %s %s\n" %(route_counter[6],
        results[0][1]))
    post_graphite("traffic.A->C_bytes %s %s\n" %(route_counter[7],
        results[0][1]))
    post_graphite("traffic.B->A_bytes %s %s\n" %(route_counter[8],
        results[2][1]))
    post_graphite("traffic.B->C_bytes %s %s\n" %(route_counter[9],
        results[2][1]))
    post_graphite("traffic.C->A_bytes %s %s\n" %(route_counter
       [10], results[3][1]))
    post_graphite("traffic.C->B_bytes %s %s\n" %(route_counter
       [11], results[3][1]))

#    while (int(time.strftime("%S")) != 29) and (int(time.strftime
   ("%S")) != 59):
    while int(time.strftime("%S")) != 59:
        time.sleep(0.3)

    print ("Ready for new collecting at: %s" %(time.strftime("%H:%
       M:%S")))

def main():

    connections_setup()

    start_time = time.strftime("%H:") + str(int(time.strftime("%M"
       ))+1)

    print ("Job will start at %s" %(start_time))

    schedule.every().day.at(start_time).do(sched_job)

    while True:
        schedule.run_pending()
        time.sleep(1)


if __name__ == '__main__':
    main()
```

A.4 NSF 92 Python framework

Python framework used to simulate the network model based on NSF 1992, running the CAIDA traffic and generating BitMatrices and traffic packet statistics based on

BitMatrices, as describe in chapter 6 Tests and Results, section 6.3 NSF92 Framework in Python.

Listing A.5 – Python framework for NSF 92 network simulation

```
#!/usr/bin/python2.7

from scapy.all import *
import csv
import crcmod
import os
import ipaddress
import datetime
import time
import sys

def crc16_comp(str_):
        str_ = bytearray(str_)
        crc16 = crcmod.mkCrcFun(0x18005, rev=False, initCrc=0xFFFF
            , xorOut=0x0000)
        answer = crc16(str(str_))
        return answer

def hashing(pkt):
        p=12
        hashlst=[pkt[p+0],                                  #version,
            ihl
                        pkt[p+2],pkt[p+3],                  #
                            totallenght
                        pkt[p+4],pkt[p+5],                  #
                            identification
                        pkt[p+6],pkt[p+7],                  #
                            flag,fragOffset
                        pkt[p+9],                           #
                            protocol
                        pkt[p+12],pkt[p+13],pkt[p+14],pkt[p+15], #
                            srcAddr
                        pkt[p+16],pkt[p+17],pkt[p+18],pkt[p+19], #
                            dstAddr
                        pkt[p+20],pkt[p+21],pkt[p+22],pkt[p+23], #
                            payld bytes 1-4
                        pkt[p+24],pkt[p+25],pkt[p+26],pkt[p+27]] #
                            payld bytes 5-8
        hash_crc16 = crc16_comp(hashlst)
        return hash_crc16

def setup():
```

```
#define global variable to:
global  rtable, \                # load src, dst tenants
   routing table in memory
                   topo, \                   # to load tenant's
                        src ip in memory
                   pkt_counter, \  # number of packet
                       processed
                   bm_len, \                 # bitmatrix lenght
                   bm_pkt_size               # number of packet
                        to process per bitmatrix
                   bm_rtr_1, \
                   bm_rtr_2, \
                   bm_rtr_3, \
                   bm_rtr_4, \
                   bm_rtr_5, \
                   bm_rtr_6, \
                   bm_rtr_7, \
                   bm_rtr_8, \
                   bm_rtr_9, \
                   bm_rtr_10, \
                   bm_rtr_11, \
                   bm_rtr_12, \
                   colision_rtr_0, \
                   colision_rtr_1, \
                   colision_rtr_2, \
                   colision_rtr_3, \
                   colision_rtr_4, \
                   colision_rtr_5, \
                   colision_rtr_6, \
                   colision_rtr_7, \
                   colision_rtr_8, \
                   colision_rtr_9, \
                   colision_rtr_10, \
                   colision_rtr_11, \
                   colision_rtr_12, \
                   pktcounter_rtr_1, \
                   pktcounter_rtr_2, \
                   pktcounter_rtr_3, \
                   pktcounter_rtr_4, \
                   pktcounter_rtr_5, \
                   pktcounter_rtr_6, \
                   pktcounter_rtr_7, \
                   pktcounter_rtr_8, \
                   pktcounter_rtr_9, \
                   pktcounter_rtr_10, \
                   pktcounter_rtr_11, \
```

pktcounter_rtr_12

```
pkt_counter = 0
bm_len = 65536 #65536
num_of_tenants = 17
bm_rtr_1 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_2 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_3 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_4 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_5 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_6 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_7 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_8 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_9 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_10 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_11 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
bm_rtr_12 = [[0 for x in range(bm_len)] for y in range(
    num_of_tenants)]
colision_rtr_1 = [0] * num_of_tenants
colision_rtr_2 = [0] * num_of_tenants
colision_rtr_3 = [0] * num_of_tenants
colision_rtr_4 = [0] * num_of_tenants
colision_rtr_5 = [0] * num_of_tenants
colision_rtr_6 = [0] * num_of_tenants
colision_rtr_7 = [0] * num_of_tenants
colision_rtr_8 = [0] * num_of_tenants
colision_rtr_9 = [0] * num_of_tenants
colision_rtr_10 = [0] * num_of_tenants
colision_rtr_11 = [0] * num_of_tenants
colision_rtr_12 = [0] * num_of_tenants
pktcounter_rtr_1 = [0] * num_of_tenants
pktcounter_rtr_2 = [0] * num_of_tenants
pktcounter_rtr_3 = [0] * num_of_tenants
pktcounter_rtr_4 = [0] * num_of_tenants
```

```
        pktcounter_rtr_5 = [0] * num_of_tenants
        pktcounter_rtr_6 = [0] * num_of_tenants
        pktcounter_rtr_7 = [0] * num_of_tenants
        pktcounter_rtr_8 = [0] * num_of_tenants
        pktcounter_rtr_9 = [0] * num_of_tenants
        pktcounter_rtr_10 = [0] * num_of_tenants
        pktcounter_rtr_11 = [0] * num_of_tenants
        pktcounter_rtr_12 = [0] * num_of_tenants


        # bm_rtr [tenant] [position]
        # load src, dst tenants routing table in memory
        with open('routing.csv', 'rb') as csvfile:
                reader = csv.reader(csvfile, delimiter=',')
                rtable = [[int(row[0]), int(row[1]), int(row[2]),
                    int(row[3]), \
                                int(row[4]), int(row[5]), int(row
                                    [6]), int(row[7])] \
                                for row in reader]

        # load tenant's src ip in memory
        with open('topology.csv', 'rb') as csvfile:
                reader = csv.reader(csvfile, delimiter=',')
                topo = [[ipaddress.ip_network(unicode(row[0]),
                    strict=False), int(row[1])] \
                                for row in reader]

def routing_table(ten_a, ten_b):
        for row in rtable:
                if (row[0] == ten_a and row[1] == ten_b):
                        while 0 in row:
                                row.remove(0)
                        return row[2:]

def lookup_tenant(tenant_ip):
        for row in topo:
                if (ipaddress.ip_address(unicode(tenant_ip)) in \
                        ipaddress.ip_network(unicode(row[0]))):
                        return  row[1]

def bitmatrix(router, ten, hash):
        comm = 'pktcounter_rtr_' + str(router) + \
                        '[' + str(ten) + ']'+ \
                    ' += 1'
        exec(comm)
        load = 0
        comm = 'load = bm_rtr_' + str(router) + \
```

```
                    '[' + str(ten) + ']'+ \
                    '[' + str(hash%bm_len) + ']'
        exec(comm)
        if load == 1:
                comm = 'colision_rtr_' + str(router) + \
                                '[' + str(ten) + ']'+ \
                            ' += 1'
                exec(comm)
        comm = 'bm_rtr_' + str(router) + \
                    '[' + str(ten) + ']'+ \
                    '[' + str(hash%bm_len) + '] = 1'
        exec(comm)

def print_counters_bitmatrix():
        global recordtime
        recordtime = recordtime + datetime.timedelta(seconds=10)
        file = open("bitmatrix_stats_4sics.csv","a")
        for i in range(1,17):
                for j in xrange(1,13):
                        pkt_cnt = 0
                        comm0 = 'pkt_cnt = pktcounter_rtr_'+ str(j
                           ) + '[' + str(i) +']'
                        colli = 0
                        comm1 = 'colli = colision_rtr_'+ str(j) +
                           '[' + str(i) +']'
                        sumbm = 0
                        comm2 = 'sumbm = sum(bm_rtr_' + str(j) + '
                           [' + str(i) +'])'
                        exec(comm0)
                        exec(comm1)
                        exec(comm2)
                        file.write(str(recordtime) + ","+
                                           str(i) + ","+
                                           str(j) + ","+
                                           str(pkt_cnt) + ","+
                                           str(colli) + ","+
                                           str(sumbm) + ","+
                                           str("{:3.2f}".format((
                                              pkt_cnt/float(bm_len
                                              ))*100)) + '%' + ","
                                              +
                                           str("{:3.2f}".format((
                                              colli/float(bm_len))
                                              *100)) + '%' + "\n")
        file.close()
```

```
def ppu(pkts):
        for i in xrange(len(pkts)):
                try:
                        global pkt_counter, pkt_counter_master
                        # 43,000 pkts corresponds to 1 second of
                           traffic
                        if pkt_counter > 430000:
                                print_counters_bitmatrix()
                                setup()
                        pkt = [ord(c) for c in raw(pkts[i])]
                        for f in range(28-len(pkt)):
                                pkt.append(0)
                        hash = hashing(pkt)
                        ten_A = lookup_tenant(pkts[i][IP].src)
                        ten_B = lookup_tenant(pkts[i][IP].dst)
                        routers = routing_table(ten_A, ten_B)
                        for router in routers:
                                bitmatrix(router, ten_A, hash)
                        pkt_counter_master += 1
                        pkt_counter += 1

                except Exception as e:
                        sys.stdout.write("\r pkt %d does not
                           exists or cant be processed"  % i)
                        sys.stdout.flush()



def loader():
        cap_files = []
        for (dirpath, dirnames, filenames) in os.walk("../4sics"):
                cap_files.extend(filenames)
                print cap_files
                return cap_files

def main():
        setup()
        global pkt_counter_master, recordtime
        recordtime = datetime.datetime(2020, 1, 1, 0, 0, 0)
        pkt_counter_master = 0
        cap_files = loader()
        for cap_file in cap_files:
                startload = time.time()
                print 'loading capture file "' + cap_file + '"
                   please, be patient...'
                pkts=rdpcap("../4sics/" + cap_file)
                endload = time.time()
```

```
                print 'capture file "' + str(cap_file) + \
                          '" were loadede in ' + str("{:10.2f}".
                             format(float(endload-startload))) + '
                              seconds'
                print  str(len(pkts)) + ' packets were loaded from
                    ' + str(cap_file)
                ppu(pkts)


main ()
```

A.5 Routing table for NSF 92 Python framework

Tenant pair	Routers in path
1,2	1,2
1,3	1,2,3
1,4	1,4
1,5	1,4
1,6	1,2,3,5
1,7	1,4,6
1,8	1,4,6
1,9	1,2,7
1,10	1,2,7,8
1,11	1,2,7,8
1,12	1,4,6,5,9
1,13	1,2,7,8,11,10
1,14	1,2,7,8,11
1,15	1,4,6,5,9,12
1,16	1,4,6,5,9,12
2,3	2,3
2,4	2,1,4
2,5	2,1,4
2,6	2,3,5
2,7	2,7,6
2,8	2,7,6
2,9	2,7
2,10	2,7,8
2,11	2,7,8
2,12	2,3,5,9
2,13	2,7,8,11,10

Tenant pair	Routers in path
6,10	5,6,7,8
6,11	5,6,7,8
6,12	5,9
6,13	5,9,10
6,14	5,9,10,11
6,15	5,9,12
6,16	5,9,12
7,8	6
7,9	6,7
7,10	6,7,8
7,11	6,7,8
7,12	6,5,9
7,13	6,5,9,10
7,14	6,7,8,11
7,15	6,5,9,12
7,16	6,5,9,12
8,9	6,7
8,10	6,7,8
8,11	6,7,8
8,12	6,5,9
8,13	6,5,9,10
8,14	6,7,8,11
8,15	6,5,9,12
8,16	6,5,9,12
9,10	7,8
9,11	7,8

Tenant pair	Routers in path
2,14	2,7,8,11
2,15	2,3,5,9,12
2,16	2,3,5,9,12
3,4	3,5,6,4
3,5	3,5,6,4
3,6	3,5
3,7	3,5,6
3,8	3,5,6
3,9	3,2,7
3,10	3,2,7,8
3,11	3,2,7,8
3,12	3,5,9
3,13	3,5,9,10
3,14	3,2,7,8,11
3,15	3,5,9,12
3,16	3,5,9,12
4,5	4
4,6	4,6,5
4,7	4,6
4,8	4,6
4,9	4,6,7
4,10	4,6,7,8
4,11	4,6,7,8
4,12	4,6,5,9
4,13	4,6,5,9,10
4,14	4,6,7,8,11
4,15	4,6,5,9,12
4,16	4,6,5,9,12
5,6	4,6,5
5,7	4,6
5,8	4,6
5,9	4,6,7
5,10	4,6,7,8
5,11	4,6,7,8
5,12	4,6,5,9
5,13	4,6,5,9,10
5,14	4,6,7,8,11
5,15	4,6,5,9,12

Tenant pair	Routers in path
9,12	7,6,5,9
9,13	7,8,11,10
9,14	7,8,11
9,15	7,8,11,12
9,16	7,8,11,12
10,11	8
10,12	8,11,10,9
10,13	8,11,10
10,14	8,11
10,15	8,11,12
10,16	8,11,12
11,12	8,11,10,9
11,13	8,11,10
11,14	8,11
11,15	8,11,12
11,16	8,11,12
12,13	9,10
12,14	9,10,11
12,15	9,12
12,16	9,12
13,14	10,11
13,15	10,11,12
13,16	10,11,12
14,15	11,12
14,16	11,12
15,16	12
1,1	1
2,2	2
3,3	3
4,4	4
5,5	4
6,6	5
7,7	6
8,8	6
9,9	7
10,10	8
11,11	8
12,12	9

Tenant pair	Routers in path
5,16	4,6,5,9,12
6,7	5,6
6,8	5,6
6,9	5,6,7

Tenant pair	Routers in path
13,13	10
14,14	11
15,15	12
16,16	12

Table 9 – Path used for traffic between tenants