A MONITORING AND THREAT DETECTION SYSTEM USING STREAM PROCESSING AS A VIRTUAL FUNCTION FOR BIG DATA

Martin Esteban Andreoni Lopez

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia Elétrica.

Orientadores: Otto Carlos Muniz Bandeira Duarte
Guy Pujolle

Rio de Janeiro
Junho de 2018

A MONITORING AND THREAT DETECTION SYSTEM USING STREAM PROCESSING AS A VIRTUAL FUNCTION FOR BIG DATA

Martin Esteban Andreoni Lopez

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

____
Prof. Otto Carlos Muniz Bandeira Duarte, Dr.Ing.

____
Prof. Guy Pujolle, Dr.

____
Prof. Khaldoun AL Agha, Dr.

____
Prof. Thi-Mai-Trang Nguyen, Dr.

____
Prof. Mauro Sergio Pereira Fonseca, Dr.

____
Prof. Daniel Macêdo Batista, Dr.

____
Prof. Luís Henrique Maciel Kosmalski Costa, Dr.

RIO DE JANEIRO, RJ – BRASIL
JUNHO DE 2018

Andreoni Lopez, Martin Esteban

A Monitoring and Threat Detection System Using Stream Processing as a Virtual Function for Big Data/Martin Esteban Andreoni Lopez. – Rio de Janeiro: UFRJ/COPPE, 2018.

XII, 119 p.: il.; 29, 7cm.

Orientadores: Otto Carlos Muniz Bandeira Duarte
Guy Pujolle

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2018.

Referências Bibliográficas: p. 106 – 119.

1. Threat Detection. 2. Network Function Virtualization. 3. Stream Processing. I. Duarte, Otto Carlos Muniz Bandeira *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Acknowledgments

I thank Mama, Papa, Dani, Mora and Ana who have always been by my side, for all their love and understanding. In particular, I thank my parents for the support they give me at all times and for always motivating me to move on.

I thank my friends, in particular Lyno Ferraz, Diogo Menezes, Govinda Mohini, Igor Sanz for their friendship. Thanks also to all the friends I made in the Grupo de Teleinformática e Automação (GTA), since they have always contributed positively to the conclusion of this work. A special mention of thanks to Antonio Gonzalez Pastana Lobato for the help and discussions and in obtaining results of this doctorate.

Thanks also to all the teachers who participated in obtaining this degree. In particular, I thank my advisor, Professor Otto Carlos Duarte, for all the advice, dedication and especially patience during the orientation. Also, a special mention to Professor Guy Pujolle, for the discussion and contribution to this work and to personal life. I would also like to thank Professors Luís Henrique Maciel Kosmalski Costa, Miguel Elias Mitre Campista, Aloysio de Castro Pinto Pedroza and Pedro Braconnot Velloso, for making our GTA/UFRJ laboratory a pleasant working environment. Also, a big thank to professors, staff and students from Laboratoire d'Informatique de Paris (LIP6) and the PHARE team for the time we spent together and for their help.

I thank Professors Thi-Mai-Trang Nguyen, Daniel Macêdo Batista, Mauro Pereira Fonseca, and Khaldoun Al Agha for their participation in the examining jury.

I thank all the people who directly or indirectly collaborated with this stage of my life. Finally, a special thanks to Brazil and its people, who have received and treated me as one of their own and have supported, economically, with their taxes, most of this achievement.

Finally, I thank CAPES, CNPq, FAPERJ and FAPESP (2015/24514-9, 2015/24485-9, and 2014/50937-1) for the funding of this work.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

UM SISTEMA DE MONITORAMENTO E DETECÇÃO DE AMEAÇAS USANDO PROCESSAMENTO DE FLUXO COMO UMA FUNÇÃO VIRTUAL PARA BIG DATA

Martin Esteban Andreoni Lopez

Junho/2018

Orientadores: Otto Carlos Muniz Bandeira Duarte
Guy Pujolle

Programa: Engenharia Elétrica

A detecção tardia de ameaças de segurança causa um significante aumento no risco de danos irreparáveis, impossibilitando qualquer tentativa de defesa. Como consequência, a detecção rápida de ameaças em tempo real é essencial para a administração de segurança. Além disso, A tecnologia de virtualização de funções de rede (*Network Function Virtualization* - NFV) oferece novas oportunidades para soluções de segurança eficazes e de baixo custo. Propomos um sistema de detecção de ameaças rápido e eficiente, baseado em algoritmos de processamento de fluxo e de aprendizado de máquina. As principais contribuições deste trabalho são: i) um novo sistema de monitoramento e detecção de ameaças baseado no processamento de fluxo; ii) dois conjuntos de dados, o primeiro é um conjunto de dados sintético de segurança contendo tráfego suspeito e malicioso, e o segundo corresponde a uma semana de tráfego real de um operador de telecomunicações no Rio de Janeiro, Brasil; iii) um algoritmo de pré-processamento de dados composto por um algoritmo de normalização e um algoritmo para seleção rápida de características com base na correlação entre variáveis; iv) uma função de rede virtualizada em uma plataforma de código aberto para fornecer um serviço de detecção de ameaças em tempo real; v) posicionamento quase perfeito de sensores através de uma heurística proposta para posicionamento estratégico de sensores na infraestrutura de rede, com um número mínimo de sensores; e, finalmente, vi) um algoritmo guloso que aloca sob demanda uma sequência de funções de rede virtual.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

A MONITORING AND THREAT DETECTION SYSTEM USING STREAM PROCESSING AS A VIRTUAL FUNCTION FOR BIG DATA

Martin Esteban Andreoni Lopez

June/2018

Advisors: Otto Carlos Muniz Bandeira Duarte
Guy Pujolle

Department: Electrical Engineering

The late detection of security threats causes a significant increase in the risk of irreparable damages, disabling any defense attempt. As a consequence, fast real-time threat detection is mandatory for security guarantees. In addition, Network Function Virtualization (NFV) provides new opportunities for efficient and low-cost security solutions. We propose a fast and efficient threat detection system based on stream processing and machine learning algorithms. The main contributions of this work are i) a novel monitoring threat detection system based on stream processing; ii) two datasets, first a dataset of synthetic security data containing both legitimate and malicious traffic, and the second, a week of real traffic of a telecommunications operator in Rio de Janeiro, Brazil; iii) a data pre-processing algorithm, a normalizing algorithm and an algorithm for fast feature selection based on the correlation between variables; iv) a virtualized network function in an open-source platform for providing a real-time threat detection service; v) near-optimal placement of sensors through a proposed heuristic for strategically positioning sensors in the network infrastructure, with a minimum number of sensors; and, finally, vi) a greedy algorithm that allocates on demand a sequence of virtual network functions.

Contents

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Objectives 5

1.2 Text Organization 9

2 Related Work 10

2.1 Stream Processing Platforms Comparison 10

2.2 Real-Time Threat Detection 11

2.3 Virtual Network Function 13

2.4 Service Chaining 14

3 Threat Detection using Stream Processing 15

3.1 Methods of Data Processing 15

3.2 The Stream Processing 17

3.3 Stream Processing Platforms 19

3.3.1 Apache Storm 20

3.3.2 Apache Flink 22

3.3.3 Apache Spark Streaming 24

3.3.4 Fault Tolerance Schemes on Distributed Processing Platforms 27

3.4 Performance Evaluation of the Platforms 28

3.4.1 Experiment Results 29

3.5 The CATRACA Tool 32

3.5.1 CATRACA Architecture 36

4 Dataset and Data Preprocessing 42

4.1 Security Dataset Creation 42

4.2 Data Preprocessing 54

4.2.1 Feature Selection and Dimensionality Reduction 55

4.3 The Proposed Preprocessing Method 62

4.3.1 The proposed Normalization Algorithm 62
4.3.2 The proposed Correlation Based Feature Selection 65
4.3.3 Evaluation 66
4.3.4 Classification Results 69
4.3.5 Related Work 82

5 The Virtual Network Function 86
5.1 The Network Function Virtualization 86
5.1.1 The Open source Platform for Network Function Virtualization (OPNFV) 88
5.1.2 Threat-Detection Prototype Performance Evaluation 89
5.2 Modeling and Optimization Strategy for VNF Sensor Location 92
5.2.1 Optimal VNF Sensor Placement 94
5.3 The Virtual Network Function Chaining Problem 95
5.3.1 The Proposed VNF Chaining Scheme 97
5.4 The Evaluation of the Proposal 98

6 Conclusion 103
6.1 Future Work 105

Bibliography 106

List of Figures

1.1 Example of virtual network functions 4

3.1 The three-layered lambda architecture. 17
3.2 stream processing architecture. 18
3.3 Example of Storm topologies. 21
3.4 Storm architecture. 21
3.5 At-least-once delivery semantic used in Apache Storm. 22
3.6 Flink Topology. 23
3.7 Architecture of Flink system. 24
3.8 Checkpoint recovery used in Flink. 25
3.9 Micro-batch processing used in Spark streaming 26
3.10 Throughput comparison for GTA/UFRJ Dataset. 30
3.11 Throughput comparison for Twitter Dataset. 30
3.12 Storm and Flink behavior during a node failure. 31
3.13 Spark stream behavior during a node failure and message losses comparison. 31
3.14 CATRACA as a Virtual Network Function. 33
3.15 CATRACA running in offline mode. 34
3.16 CATRACA running in online mode. 35
3.17 Architecture of the CATRACA tool. 36
3.18 Flow Diagram of the Capture Module. 38
3.19 An example of Decision Tree for threat classifications. 39
3.20 Dashboard panel view of CATRACA. 41

4.1 Correlation matrix of the 24 features available in the dataset. 45
4.2 Classes distribution in the dataset. 46
4.3 Typical topology of broadband access network. 47
4.4 DSLAM Topology 48
4.5 NetOp dataset processing steps. 50
4.6 Number of Alerts and Normal flows. 50
4.7 Dataset NetOp Port Distribution. 50

4.8 Flow Distribution Protocols 51
4.9 Packets per Flow 52
4.10 Flow Size NetOp 2017 52
4.11 subflows Size NetOp 2017 53
4.12 Header Size NetOp 2017 53
4.13 Distribution of the main types of alerts in the analyzed traffic. 54
4.14 Preprocessing Steps. 55
4.15 The “curse of dimensionality” problem. 56
4.16 Eigenvalue for each of the 24 flow features. 58
4.17 Example of non-linear class separation with PCA. 59
4.18 Strategies for separating non-linear data distribution classes 59
4.19 Feature Selection Methods. 60
4.20 Representation of the feature divided in histogram. 64
4.21 Shapiro-Wilk hypothesis test. 67
4.22 Information gain sum for feature selection algorithms. 69
4.23 Accuracy comparison of features Selection methods. 70
4.24 Sensitivity of detection. 71
4.25 Performance of features selection algorithms. 72
4.26 Evaluation of Feature Selection in NSL-KDD dataset. 74
4.27 Metrics in NSL-KDD dataset with no feature selection. 76
4.28 Metrics reducing only 10% of the initial features in NSL-KDD. 76
4.29 Classification and training time in NSL-KDD Dataset. 77
4.30 Classification and training time in NSL-KDD Dataset 10% reduction. 77
4.31 Evaluation of Feature Selection varying the selected features in NetOp dataset. 78
4.32 Metrics with no future selection NetOp dataset. 78
4.33 Metrics with 90% of future selection NetOp dataset. 79
4.34 Classification and training time in NetOp Dataset. 80
4.35 Classification and training time in NetOp Dataset 90% reduction. . . 80
4.36 Evaluation of group features with different machine learning algorithms. 81
4.37 Flow diagram used for proposal evaluation. 82
4.38 Concept Drift detection. 83

5.1 ETSI NFV MANO framework. 87
5.2 OPNFV architecture. 88
5.3 CATRACA configuration as a Virtual Network Function. 90
5.4 VNF throughput. 90
5.5 Virtual Machine migration. 91
5.6 RNP topology. 94

5.7 Proposal Heuristic evaluation. 95
5.8 Example of Network Function forwarding graph. 96
5.9 Probability density function of the number of VNFs in a request. . . 99
5.10 Probability density function of the `volume` of resources used by each VNF. 99
5.11 Number of accepted requests for each proposed heuristics. 101
5.12 Simulation results. 101

List of Tables

3.1 Comparison summary between batch processing and stream processing. 16
3.2 Overview of the comparison between Stream Processing Systems. . . 29
3.3 Evaluation Metrics of Decision Tree for GTA/UFRJ Dataset. 40
3.4 Evaluation Metrics of Decision Tree for NetOp Dataset. 40

4.1 The 24 features obtained for each flow from TCP/IP headers. 45
4.2 Hypothesis comparison for a normal distribution approach. 67
4.3 Features Groups 81
4.4 Features description of NetOp Dataset. 85

Chapter 1

Introduction

In order to maintain the stability, reliability, and security of computer networks, it is fundamental to monitor the traffic to understand the type, volume, and intrinsic features of each flow that compose the whole traffic. Therefore, efficient network monitoring allows the administrator to achieve a better understanding of the network [1]. Network monitoring may vary from a simple long-term collection of link utilization statistics, to a complex upper-layer protocol traffic analysis to achieve network intrusion detection, network performance tuning, and protocol debugging. Current network monitoring tools, such as Tcpdump[1], wireshark, NetFlow, Bro [2], Multi Router Traffic Grapher (MRTG), are inadequate for current speed and management needs of large network domains. In addition, many tools generate a massive collection of files that needs post-processing by another kind of tools.

In network monitoring, data arrives in the form of streams from different sources. Monitoring data arises from several heterogeneous distributed sources, such as network packets from different nodes or multiple kinds of logging systems [3]. These stream applications are characterized by an unbounded sequence of events, or tuples, that arrive continuously [4]. One of the main problems of these type of applications is the big amount of data generated. Even moderate speed networks generate huge amounts of data. For example, monitoring a single gigabit Ethernet link running at 50% utilization generates a terabyte of data in a couple of hours. Furthermore, the advent of the Internet of Things (IoT) increases the need of real-time monitoring. The estimated number of sensors networked by 2025 is around 80 billion [5].

This scenario displays a high monitoring and protection complexity with several challenges in security and data privacy. The billions of devices generate a big amount of data streams, which need to be managed, processed, transferred, and stored in a secure real-time way. Besides, the big data characteristics of velocity, volume, and variety increase the number of vulnerabilities.

[1]Tcpdump packet analyzer www.tcpdump.org Accessed June 2018.

Consequently, detection time is essential to maintain security in communication systems [6]. If detection takes too long, irreparable damages will occur. Denial of Service (DoS) [7] attacks, for example, need to be mitigated as fast as possible in order to maintain a proper Quality of Service (QoS). The effective threat detection demands monitoring, processing, and management of data, in order to extract useful information from network traffic [1]. Current security systems, such as Security Information and Event Management (SIEM) [8], designed to gather data and analyze it at a single point, are not effective, since 85% of network intrusions are detected weeks after they happened [5]. Moreover, in a study conducted by IBM/ponemon surveying 350 companies the mean time to detect data leak was 206 days, with a range of between 20 and 582 days being reported [9]. Therefore, the long threat detection time makes unfeasible any kind of defense.

The impact of Distributed Denial of Service (DDoS) attacks is increasing, reaching attack rates of the order of 1 Tb/s [10]. The access tools to perform attacks are so popular that it is possible to buy an hour of a DDoS attack for $10 per hour on the deep web [11]. Nowadays, the attacks are no longer merely motivated by economic ends and became politically motivated [10]. Currently attacks aim to manipulate election results, such as the suspicion of Russian manipulation over the United States elections in 2016 [12].

One way to attain data processing optimization is to employ machine learning methods. These methods are well suited for big data, since with more samples to train, methods tend to have higher effectiveness [13]. However, with a high feature number, machine learning methods perform results with high latency due to computational resources consumption. This high latency is a drawback for machine learning methods that must analyze data as fast as possible in order to have fast responses. Features Selection is one way to resolve this problem, reducing the number of features to smaller subsets of the original. The main method to analyze big data in a distributed fashion is the MapReduce [14] technique with Hadoop [15] open-source implementation. Nevertheless, the platforms based on this technique are inappropriate to process real-time streaming applications. Applications processed by Hadoop correspond to queries or transactions performed in a static database without real-time requirements.Real-time monitoring applications require distributed stream processing that substantially differs from current conventional applications processed by distributed platforms. Network Monitoring normally requires to analyze multiple external stream sources, generating alerts in abnormal conditions. The real-time feature is intrinsic to stream processing applications and a big number of alerts is normally expected. The stream data are unbounded and arrive asynchronously. Besides, the stream analysis requires historical data rather than just the latest arrived data [16]. In cases of high incoming rates, flow and packet sampling techniques

are commonly adopted. Even if these techniques have been studied to achieve high data rates [17],[18], sampling entails an inherent loss of information [19]. Hence, distributed processing models have been proposed to meet real-time application requirements, receiving attention from researchers.

Real time analytic are essential for Future Internet and Smart Cities [20, 21]. Moreover, real-time stream processing enables the immediate analysis of different kinds of data and, consequently, they empower threat detection. Real-time distributed stream processing models can benefit traffic monitoring applications for cyber-security threats detection [22]. Sensor monitoring in the Internet of Things (IoT), network traffic analysis, cloud management [23], smart grids [24] are applications that generate large amounts of data. In these critical applications, data need to be processed in real time in order to detect security threats.

To meet these needs, Distributed Stream Processing Systems (DSPS) have been proposed to perform distributed processing with minimal latency. Besides, open-source general-purpose stream processing platforms meet the need of processing data continuously. Available implementation of these platforms are Apache Storm [25], Apache Spark [26] and Apache Flink [27]. These open-source platforms are able to define custom stream processing applications for specific cases. These general-purpose platforms offer an Application Programming Interface (API), fault tolerance, and scalability for stream processing.

Current enterprise networks rely on middleboxes. Middleboxes are intermediary devices that add new functionalities to the network. An example of network middleboxes are firewalls that establish a barrier for network attacks; load-balancers which improve performance distributing workload over network nodes; or proxies that reduce bandwidth consumption [28]. Middleboxes are usually dedicated hardware nodes, which perform a specific network function. Hence, middlebox platforms come with high Capital Expenditures (CAPEX) and Operational Expenditures (OPEX) [29]. In this way, the Network Function Virtualization (NFV) comes to leverage standard virtualization technology into the network core, and to consolidate network equipment into commodity server hardware [30]. In NFV, the network functions are deployed into virtualized environment and, thus, called Virtual Network Functions (VNF).

We aim to use NFV technology and its cluster infrastructure to combine virtualization, cloud computing and distributed stream processing to monitor network traffic. The objective is to provide an accurate, scalable, and real-time threat detection facility capable to attend usage peeks. The traffic monitoring and threat detection as a virtualized network function presents two main advantages: capacity self-adaptation to different traffic network load and high localization flexibility to place or move network sensors reducing latency.

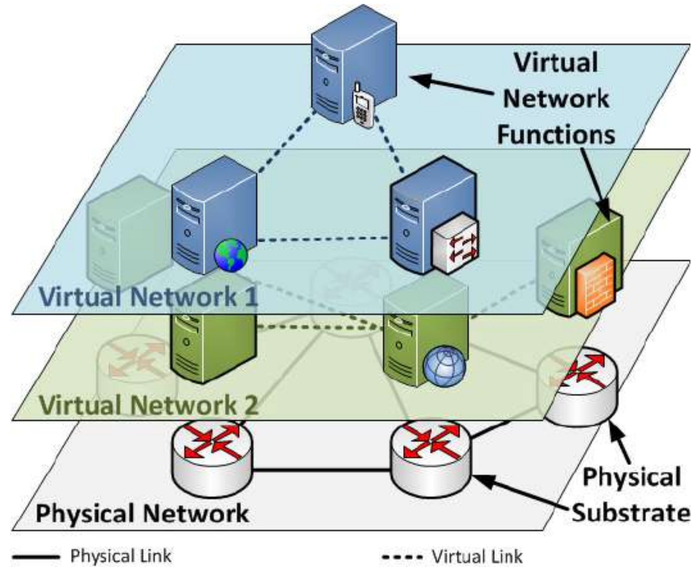


Figure 1.1: Example of virtual network functions. Two separated plans of virtual network functions decoupled from the underlying physical substrate.

When considering the deployment of middleboxes as Virtual Network Functions, a key challenge is the Service Function Chaining (SFC) [31]. The SFC problem stands for the requirement of traffic to pass through multiple middleboxes for packet processing in a previously defined order. It becomes harder when considering the NFV environment because allocating virtual network function over the physical nodes have to consider the packet-processing chaining order among all VNFs in the traffic path, as shown in Figure 1.1. Therefore, two main constraints for VNF chaining are to reduce the delay introduced by placing the VNFs on the network topology, and to allocate VNFs on physical nodes that can provide enough resources to run the hosted functions.

Chaining virtual network functions is essentially an optimization problem that recalls the facility location problem [31]. Previous works propose linear programming models to search for a solution for VNF chaining, considering resource allocation and the routing between virtual instances [31–33]. Other works propose to outsource network functions to the cloud [28, 34], but do not specify an algorithm for placing VNFs on the network. Moreover, there are also works that place specific-purpose VNFs on the network, such as traffic sensors or network controllers [35, 36]. Our proposal uses a greedy algorithm to place general-purpose VNFs over a network topology and compares different heuristics. Our scheme estimates the resources at each physical node on the network, and then, places the VNFs according to the available resources of the physical nodes and the requested resources for the VNF.

1.1 Objectives

The goal of this manuscript is to present the research work and the obtained results achieved so far. The main idea is to show the state of the art and the developed research work. The research topics assessed are Stream Processing, Real-Time Threat Detection System, Dataset and Feature Selection, Virtual Network Function, and Virtual Network Function Chaining. Next, we briefly describe these research topics.

Stream Processing

We analyze and compare two native distributed real-time and stream-processing systems, the Apache Storm [25] and the Apache Flink [27], and one micro-batch system, the Apache Spark Streaming [26]. The architecture of each analyzed system is discussed in depth and a conceptual comparison is presented showing the differences between these open-source platforms. Furthermore, we evaluate the data processing performance and the behavior of systems when a worker node fails.

Real-Time Threat Detection System

We propose and implement an accurate real-time threat detection system, the CATRACA[2] tool [37]. The integrated system allows big data analysis in a stream processing manner. The proposed system uses machine learning for both attack classification and threat detection. Moreover, the system has a friendly graphical interface that provides a real-time visualization of the parameters and the attacks that occur in the network.

Dataset and Data Preprocessing

We created two datasets, the first one is a synthetic security dataset to perform traffic classification and the second one is a real traffic from a network operator in Rio de Janeiro, Brazil. Furthermore, we present a fast preprocessing method for network traffic classification based on feature correlation and feature normalization. Our proposed method couples a normalization and a feature selection algorithms. We evaluate the proposed algorithms against three different datasets for eight different machine learning classification algorithms. Our proposed normalization algorithm reduces the classification error rate when compared with traditional methods. Our Feature Selection algorithm chooses an optimized subset of features improving accuracy by more than 11% within a 100-fold reduction in processing time when compared to traditional feature selection and feature reduction algorithms.

Virtual Network Function

We evaluate CATRACA as a Virtual Network Function (VNF). Consequently, we propose a virtualized network function in the Open Source Platform for Network Functions Virtualization (OPNFV) that provides an accurate real-time threat

[2]documentation available at `http://catraca.gta.ufrj.br/` Accessed June 2018.

detection service. To the best of our knowledge, this is the first threat detection function using stream processing implemented in the OPNFV platform. The service provided is able to scale the number of processing cores by adding virtual machines to the processing cluster that executes the detection in a parallel-distributed way, processing up to 15 Millions samples per minute. Besides, the Network Virtualization Platform enables the easy deployment of traffic capture sensor elements that can be placed and moved to several points in the network, offering customization and adaptability to network monitoring. The results show the potential for scalability, as we increase the number of processing cores in the distributed cluster. Another important feature of our proposal is the migration of processing machines. The experiments show that our system can migrate the processing elements without stopping the threat detection. The live migration enables the organization of the physical machines in the processing cluster, which results in several advantages, such as shutting down machines for maintenance or for reduction of energy consumption or allocating resources in a smart way to meet the demand.

Virtual Network Function Chaining

We propose a scheme for placing and chaining Virtual Network Functions over a network topology according to four different heuristics. The first heuristic places the VNF nodes into physical nodes that introduce the minimum delay between the traffic source and destination. The second heuristic searches for the best placement of VNF nodes considering the nodes that have the biggest amount of available resources and, thus, places the VNF over the most available node. This approach increases the number of accepted requests of VNFs in a network. The third heuristic places the VNF nodes according to the betweenness-centrality of the topology nodes. In the betweenness-centrality approach, the requests are primarily responded by allocating the most central nodes on the topology, which reduces the introduced delay. However, as the resources of the most central nodes are used, the following requests are allocated into peripheral network nodes, introducing a greater delay on the VNF chaining. The fourth heuristic weights the available resources and the introduced delay for each physical node. This approach allocates the VNFs on the nodes that present the greatest probability of supplying enough resources and the lowest delay. We deploy a greedy algorithm for all four approaches and we simulate the allocation of VNFs over a real network topology.

I have published as first author

- GTA-17-27 Andreoni Lopez, M., Sanz, I. J., Mattos, D. M. F., Duarte, O. C. M. B., Pujolle, G. - "CATRACA: uma Ferramenta para Classificação e Análise Tráfego Escalável Baseada em Processamento por Fluxo", in XVII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais - SBSeg'2017 -**Best Tool Award** - Brasília, DF, Brazil, November 2017. `http://www.gta.ufrj.br/ftp/gta/TechReports/ASM17.pdf`

- GTA-17-21 Andreoni Lopez, M., Silva Souza, R., Alvarenga, I. D., Rebello, G. A. F., Sanz, I. J., Lobato, A. P., Mattos, D. M. F., Duarte, O. C. M. B. and Pujolle, Guy - "Collecting and Characterizing a Real Broadband Access Network Traffic Dataset", in 1st Cyber Security in Networking Conference (CSNet'17) - **Best Paper Award** - Rio de Janeiro, Brazil, October 2017. `http://www.gta.ufrj.br/ftp/gta/TechReports/ASAR17.pdf`

- GTA-17-06 Andreoni Lopez, M., Silva, R. S., Alvarenga, I. D., Mattos, D. M. F., Duarte, O. C. M. B. - "Coleta e Caracterização de um Conjunto de Dados de Tráfego Real de Redes de Acesso em Banda Larga", in XXII Workshop de Gerência e Operação de Redes e Serviços (WGRS'2017) - SBRC'2017, Belém- Pará, PA, Brazil, May 2017. `http://www.gta.ufrj.br/ftp/gta/TechReports/ASA17.pdf`

- GTA-17-04 Andreoni Lopez, M., Lobato, A. G. P., Mattos, D. M. F., Alvarenga, I. D., Duarte, O. C. M. B., Pujolle, G. - "Um Algoritmo Não Supervisionado e Rápido para Seleção de Características em Classificação de Tráfego", in XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2017, Belém- Pará, PA, Brazil, May 2017. `http://www.gta.ufrj.br/ftp/gta/TechReports/ALM17.pdf`

- GTA-16-34 Andreoni Lopez, M., Lobato, A. G. P., Duarte, O. C. M. B., and Pujolle, G. - "Design and Performance Evaluation of a Virtualized Network Function for Real-Time Threat Detection using Stream Processing", Fourth Conference On Mobile And Secure Services (MobiSecServ), Miami, MI, USA, February 2018. `http://www.gta.ufrj.br/ftp/gta/TechReports/ALDP16.pdf`

- GTA-16-31 Andreoni Lopez, M., Mattos, D. M. F., and Duarte, O. C. M. B. - "Evaluating Allocation Heuristics for an Efficient Virtual Network Function Chaining", in IEEE 7th International Conference Network of the Future - NoF'2016, Búzios-RJ, Brazil, November 2016. `http://www.gta.ufrj.br/ftp/gta/TechReports/AMD16b.pdf`

- GTA-16-24 Andreoni Lopez, M., Lobato, A. G. P., and Duarte, O. C. M. B. - “A Performance Comparison of Open-Source Stream Processing Platforms”, in IEEE Global Communications Conference - GLOBECOM’2016, Washington, DC USA, December 2016. `http://www.gta.ufrj.br/ftp/gta/TechReports/ALD16b.pdf`

- GTA-16-11 Andreoni Lopez, M., Lobato, A. G. P., and Duarte, O. C. M. B. - “Monitoramento de Tráfego e Detecção de Ameaças por Sistemas Distribuídos de Processamento de Fluxos: uma Análise de Desempenho”, in XXI Workshop de Gerência e Operação de Redes e Serviços (WGRS’2016) - SBRC’2016, Salvador-Bahia, BA, Brazil, May 2016. `http://www.gta.ufrj.br/ftp/gta/TechReports/ALD16b.pdf`

Also, as co-author I have published

- GTA-18-08 Sanz, I. J., Andreoni Lopez, M., Rebello, G. A. F. and Duarte, O. C. M. B.- “Um Sistema de Detecção de Ameaças Distribuídos de Rede baseado em Aprendizagem por Grafo”, in XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC’2018. Campos de Jordão, SP, Brazil, May 2018.

- GTA-18-02 Lobato, A. G. P., Andreoni Lopez, M., Sanz, I. J., Cardenas, A. A., Duarte, O. C. M. B. and Pujolle, Guy- “An Adaptive Real-Time Architecture for Zero-Day Threat Detection”, to be published in IEEE International Conference on Communications - ICC’2018, May 2018. `http://www.gta.ufrj.br/ftp/gta/TechReports/LASC18.pdf`

- GTA-17-22 Sanz, I. J., Andreoni Lopez, M., Mattos, D. M. F., and Duarte, O. C. M. B. - “A Cooperation-Aware Virtual Network Function for Proactive Detection of Distributed Port Scanning”, in 1st Cyber Security in Networking Conference (CSNet’17), Rio de Janeiro, Brazil, October 2017. `http://www.gta.ufrj.br/ftp/gta/TechReports/SAMD17.pdf`

- GTA-17-16 Lobato, A. P., Andreoni Lopez, M. E., Rebello, G. A. F., and Duarte, O. C. M. B. - “Um Sistema Adaptativo de Detecção e Reação a Ameaças”, to be published in Anais do XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg’17, Brasília, DF, Brazil, November 2017. `http://www.gta.ufrj.br/ftp/gta/TechReports/LLD17.pdf`

- GTA-17-15 Sanz, I. J., Alvarenga, I. D., Andreoni Lopez, M. E., Mauricio, L. A. F., Mattos, D. M. F., Rubistein, M. G. and Duarte, O. C. M. B. - "Uma Avaliação de Desempenho de Segurança Definida por Software através de Cadeias de Funções de Rede", to be published in Anais do XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg'17, Brasília, DF, Brazil, November 2017. http://www.gta.ufrj.br/ftp/gta/TechReports/SAA17.pdf

- GTA-16-08 Lobato, A. G. P., Andreoni Lopez, M. and Duarte, O. C. M. B. - "An Accurate Threat Detection System through Real-Time Stream Processing", Technical Report, Electrical Engineering Program, COPPE/UFRJ, April 2016. http://www.gta.ufrj.br/ftp/gta/TechReports/LAD16.pdf

- GTA-16-10 Lobato, A. G. P., Andreoni Lopez, M. and Duarte, O. C. M. B. - "Um Sistema Acurado de Detecção de Ameaças em Tempo Real por Processamento de Fluxos", in XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2016, Salvador- Bahia, BA, Brazil, May 2016. http://www.gta.ufrj.br/ftp/gta/TechReports/LAD16b.pdf

As journal publications

- Andreoni Lopez, M., Mattos, D. M. F., Duarte, O. C. M. B., and Pujolle G. - "A Fast Unsupervised Preprocessing Method for Network Monitoring" , submitted to Annals of Telecommunications, Springer.

- GTA-16-05 Andreoni Lopez, M., Mattos, D. M. F., and Duarte, O. C. M. B. - "An elastic intrusion detection system for software networks", in Annals of Telecommunications, Springer, ISSN 0003-4347, DOI 10.1007/s12243-016-0506-y, 2016. http://www.gta.ufrj.br/ftp/gta/TechReports/AMD16.pdf

1.2 Text Organization

The rest of the paper is organized in six chapters. Chapter 2 discusses related work. Then, we introduce the concept of stream processing, we compare stream processing platforms in Chapter 3 and we present the CATRACA tool for network monitoring and real-time threat detection system. In Chapter 4, we introduce the security dataset and the network operator dataset, we also discuss data preprocessing methods, then, we propose a normalization and a feature selection algorithm. Chapter 5 presents the Virtual Network Function for threat classification and an algorithm for VNF chaining. Finally, Chapter 6 concludes the work and discusses the future work.

Chapter 2

Related Work

This chapter discusses the state of the art and presents the related work of the topics addressed in this work. We divide this chapter in four main topics. Section 2.1 describes the related work concerning Stream Processing Platforms. Section 2.2 addresses Real-time threat detection. Virtual Network Functions are presented in Section 2.3 and service chaining is introduced in Section 2.4.

2.1 Stream Processing Platforms Comparison

Distributed real-time stream processing systems is a recent topic that is gaining a lot of attention from researchers. Hence, performance evaluations and comparisons between stream processing systems are fairly unexplored in the scientific literature.

Hesse and Lorenz compare the Apache Storm, Flink, Spark Streaming, and Samza platforms [38]. The comparison is restricted to description of the architecture and its main elements. Gradvohl *et al.* analyze and compare Millwheel, S4, Spark Streaming, and Storm systems, focusing on the fault tolerance aspect in processing systems [39]. Actually, these two above cited papers are restricted to conceptual discussions without experimental performance evaluation. Landset *et al.* perform a summary of the tools used for process big data [40], which shows the architecture of the stream processing systems. However, the major focus is in batch processing tools, which use the techniques of MapReduce. Roberto Colucci *et al.* show the practical feasibility and good performance of distributed stream processing systems for monitoring Signaling System number 7 (SS7) in a Global System for Mobile communications (GSM) machine-to-machine (M2M) application [41]. They analyze and compare the performance of two stream processing systems: Storm and Quasit, a prototype of University of Bologna. The main result is to prove Storm practicability to process in real time a large amount of data from a mobile application.

Nabi *et al.* compare Apache Storm with IBM InfoSphere Streams platform in an e-mail message processing application [42]. The results show a better performance

of InfoSphere compared to Apache Storm in relation to throughput and CPU utilization. However, InfoSphere is an IBM proprietary system and the source code is unavailable. Lu *et al.* propose a benchmark [43] creating a first step in the experimental comparison of stream processing platforms. They measure the latency and throughput of Apache Spark and Apache Storm. The paper does not provide results in relation to Apache Flink and the behavior of the systems under failure.

Dayarathna e Suzumura [44] compare the throughput, CPU and memory consumption, and network usage for the stream processing systems S, S4, and the Event Stream Processor Esper. These systems differ in their architecture. The S system follows the manager/workers model, S4 has a decentralized symmetric actor model, and finally Esper is software running on the top of Stream Processor. Although the analysis using benchmarks is interesting, almost all evaluated systems are already discontinued or not currently have significant popularity.

Unlike most of above-mentioned papers, we focus on open-source stream processing systems that are currently available such as Apache Storm, Flink, and Spark Streaming [45, 46]. We aim at describing the architectural differences of these systems and providing experimental performance results focusing on the throughput and parallelism in a threat detection application on a dataset created by the authors.

2.2 Real-Time Threat Detection

Some proposals use Apache Storm stream processing tool to perform real-time anomaly detection. Du *et al.* use the Flume and Storm tool for traffic monitoring to detect anomalies. The proposal is to make the detection through the k-NN algorithm [47]. The article presents some performance results, but it lacks evaluation of the accuracy of detection and the tool only receives data from a centralized node, ignoring data from distributed sources. The work of Zhao *et al.* uses the Kafka and Storm, as well as the previous work, for the detection of network anomalies [48], characterizing flows in the NetFlow format. He *et al.* propose a combination of the distributed processing platforms Hadoop and Storm, in real time, for the detection of anomalies. In this proposal, a variant of the k-NN algorithm is used as the anomaly detection algorithm [49]. The results show a good performance in real time, however without using any process of reaction and prevention of the threats. Mylavarapu *et al.* propose to use Storm as a stream processing platform for intrusion detection [50].

Dos Santos *et al.* uses a combination of Snort IDS and OpenFlow to create a tool called OpenFlow Intrusion Detection Prevention System (Of-IDPS). Snort IDS is used as a detection tool, while OpenFlow actions perform the mitigation or prevention of detected attacks [51]. An evolution of Of-IDPS was proposed to develop an Autonomous Computation (AC) system to automatically create security

rules in Software Defined Network (SDN) switches [6]. Rules are created applying a machine learning algorithm to Snort IDS alerts and OpenFlow logs. The machine learning method used in this work is the FP-Growth to find frequent item sets, also called association rules. Schuartz *et al.* propose a distributed system for threat detection in Big Data traffic [52]. Apache Storm and Weka machine learning tool are used to analyze KDD-99 dataset. The system is based in lambda big data architecture that combines batch and stream processing.

Stream processing platforms have been used for security initiatives. Apache Metron[1] is a security analysis framework based on big data processing. Metron architecture consists of acquisition, consumption, distributed processing, enrichment, storage and visualization of the data layers. The key idea of this framework is to allow the correlation of security events from different sources, such as application logs and network packets. To this end, the framework employs distributed data sources such as sensors in the network, action logs of active network security elements and enriched data called telemetry sources. The framework also relies on a historical foundation of network threats from Cisco. Apache Spot[2] is a project similar to Apache Metron still in incubation. Apache Spot uses telemetry and machine learning techniques for packet analysis to detect threats. The creators say that the big difference with Apache Metron is the ability to use common open data models for networking. Stream4Flow[3] uses Apache Spark with the ElasticStack stack for network monitoring. The prototype serves as a visualization of network parameters. Stream4Flow [53], however, has no intelligence to perform anomaly detection. Hogzilla[4] is an intrusion detection system (IDS) with support for Snort, SFlows, GrayLog, Apache Spark, HBase and libnDPI, which provides network anomaly detection. Hogzilla also allows to realize the visualization of the traffic of the network.

The proposed CATRACA tool, like Metron, aims to monitor large volumes of data using flow processing. The CATRACA tool is implemented as a virtualized network function (VNF) in the Open Platform for Network Function Virtualization (OPNFV) environment. CATRACA focuses on real-time packet capture, feature selection and machine learning. CATRACA can be combined with a mechanism of action for immediate blocking of malicious flows. Thus, the CATRACA tool acts as a virtualized network intrusion detection and prevention function that reports flow summaries and can be linked to other network virtualized functions [54] as defined in the network function chain patterns (Service Function Chaining - SFC) and network service headers (Network Service Header - NSH). Network Service Header is a data-plane protocol that enables the encapsulation of SFCs. NSH is added to the

[1] http://metron.apache.org/, Accessed April 2018.
[2] http://spot.incubator.apache.org, Accessed April 2018.
[3] https://github.com/CSIRT-MU/Stream4Flow, Accessed April 2018.
[4] http://ids-hogzilla.org/, Accessed April 2018.

packet header to create a specific service place that is independent of the underlying transport protocol [55]. The main idea of NSH is to characterize a sequence of service nodes that must be routed before arrive to the destination.

2.3 Virtual Network Function

Machine learning is used for attack detection in virtualized environments [56, 57]. Azmandian *et al.* present an application based on machine learning to automatically detect malicious attacks on typical server workloads running on virtual machines. The key idea is to obtain the feature selection by Sequential Floating Forward Selection (SFFS) algorithm, also known as Floating Forward Search, and, then, classify the attacks with the K-Nearest Neighbor (KNN) and the Local Outlier Factor (LOF) machine learning algorithms. The system runs in one physical machine under VirtualBox environment. Li *et al.* present cloudmon [57], a Network Intrusion Detection System Virtual Appliance (NIDS-VA), or virtualized NIDS. Cloudmon enables dynamic resource provisioning and live placement for NIDS-VAs in Infrastructure as a Service (IaaS) cloud environments. The work uses Snort IDS and Xen hypervisor for virtual machine deployment. Moreover, Cloudmon uses fuzzy model and global resource scheduling to avoid idle resources in a cloud environment. The proposal employs the conventional Snort IDS, based on signature method, to detect misuse and focuses on the resource allocation. BroFlow covers the detection and mitigation of Denial of Service (DoS) attacks [22]. Sensors run in virtual machine under Xen hypervisor, and thus, include a mechanism for optimal sensor distribution in the network [22]. An attack mitigation solution, based on Software Defined Networking, complements the proposal, focusing on DoS attacks detection based on an anomaly algorithm implemented in the Bro IDS. CATRACA is proposed as a virtualized network function on Open Source Platform for Network Function Virtualization (OPNFV) that provides a threat detection facility. The function employs open source tools to detect threats in real time using flow processing and machine learning techniques.

Sensor placement is studied by Chen *et al.*, that propose a technique based on Genetic Algorithms (GA) [58] for sensor placement. The algorithm has as a heuristic the minimization in the sensor number and the maximization of the detection rate. Bouet *et al.* also use GA as optimization technique for the deployment of Deep Packet Inspection (DPI) virtual sensors [59]. Bouet's proposal minimizes the sensor number and the load analyzed by each sensor, however, this proposal based on GA requires high processing time to obtain the results without warranting the solution convergence [60]. We model and propose a heuristic for optimization in VNF sensor placement, reducing the number of sensor and maximizing the network coverage [61].

2.4 Service Chaining

Virtual Network Function chaining is currently a trend topic in research. Several researches deal with the optimization problem to place a set of VNFs [31–33]. Addis *et al.* propose a mixed integer linear programming formulation to solve the VNF placement optimization from the Internet Service Providers (ISPs) point of view [32]. In a similar way, Bari *et al.* use an Integer Linear Programming in order to optimize the cost of deploying a new VNF, the energy cost for running a VNF, and the cost of forwarding traffic to and from a VNF [31]. A Pareto optimization is used for placing chained VNFs in an operator's network with multiple sites, based on requirements of the tenants and of the operator [33].

Other works propose the optimization placement of specific VNF [22, 36, 62]. A virtual Deep Packet Inspection (vDPI) placement is proposed by Bouet *et. al.* to minimize the cost that the operator faces [62]. In a previous work [22], we proposed the placement of an Intrusion Detection and Prevention System (IDPS) by a heuristic that maximizes the traffic passing through each node. In another previous work [36], we proposed a heuristic to optimize the placement of distributed network controllers in a Software Defined Network environment. Nevertheless, none of these works considers the trade-off of the costumer requests and infrastructure provider availability.

Optimizing resource allocation has been proposed in many other contexts. Sandpiper [63] is a resource management tool for datacenters. It focuses on managing the allocation of virtual machines over a physical infrastructure. Other proposal that estimates the resource usage for allocating virtual machines in a datacenter is Voltaic [64]. Voltaic is a management system focused on cloud computing which aims to ensure compliance with service level agreements (SLAs) and optimize the use of computing resources.

In Section 5.3, we propose four heuristics in order to minimize the delay between source and destination nodes for the best Quality of Experience (QoE). Another heuristic is proposed to minimize the resource usage on the network nodes to increase Infrastructure Provider (IP) benefits. Finally, a heuristic for using the most central nodes first to improve costumer QoE and IP benefit. We compare the four proposed heuristics with a greedy algorithm and we tested over a real Service Provider topology [65].

Chapter 3

Threat Detection using Stream Processing

In this chapter, we present a threat detection prototype using stream processing. First, we present the main data processing techniques. Then, we introduce the stream processing paradigm. Next, we describe and compare the main Open-Source stream processing platforms in order to select the most suitable for our Network Analytics tool. Finally, we present the CATRACA tool, a network monitoring and threat detection tool using stream processing and machine learning techniques.

3.1 Methods of Data Processing

Stream processing makes it possible to extract values on moving data, as batch processing does for static data. Static data remain unchanged after being recorded. On the other hand, moving or dynamic data change after recorded and have to be continually updated. The purpose of stream processing is to enable real-time or near-real-time decision making by providing the ability to inspect, correlate, and analyze stream data as data flows through the processing system. Examples of scenarios that require stream processing are: traffic monitoring applications for computer network security; social networking applications such as Twitter or Facebook; financial analysis applications that monitor stock data flows reported on stock exchanges; detection of credit card frauds; inventory control; military applications that monitor sensor readings used by soldiers, such as blood pressure, heart rate, and position; manufacturing processes; energy management; among others. Many scenarios require processing capabilities of millions or hundreds of millions of events per second, making traditional databases, called Data Base Management System (DBMS), inappropriate to analyze stream data [66]. Data Base Management Systems store and index data records before making them available to the query activity, which makes

them unsuitable for real-time applications or responses in the sub-second order [67]. Static databases were not designed for fast and continuous data loading. Therefore, they do not directly support the continuous processing that is typical of data stream applications. Also, traditional databases assume that the process is strictly stationary, differing from real-world applications, in which the output could gradually change over time. Security Threats in TCP/IP networks are a typical example of moving data, in which the output to the same query changes over time.

Data processing is divided in three main processing approaches: batch, micro-batch, and stream. The analysis of large sets of static data, which are collected over previous periods, is done with batch processing. A famous technique that uses batch processing is the MapReduce [14], with the popular open-source implementation Hadoop [15]. In this scheme, data is collected, stored in files, and then processed, ignoring the timely nature of the data production. However, this technique presents large latency, with responses in the order of seconds, while several applications require real-time processing, with responses in microsecond order [68]. Also, this technique can perform near real-time processing by doing micro-batch processing. Micro-batch treats the stream as a sequence of smaller data blocks. Data input is grouped into smaller data blocks and delivered to the batch system to be processed. On the other hand, the third approach, stream processing, is able to analyze massive sequences data that are continuously generated with responses of real time [69].

Stream Processing differs from the conventional batch model in: i) the data elements in the stream arrive online; ii) the system has no control over the order in which the data elements arrive to be processed; iii) stream data are potentially unlimited in size; iv) once an element of a data stream has been processed, it is discarded or archived and cannot be easily retrieved, unless it is explicitly stored in memory, which is usually small relative to the size of the data streams. Further, latency of stream processing is better than micro-batch, since messages are processed immediately after arrival. Stream processing performs better for real time; however, fault tolerance is costlier, considering that it must be performed for each processed message. Table 3.1 summarizes the main differences between static batch processing and moving data stream processing.

Table 3.1: Comparison summary between batch processing and stream processing.

	Batch	**Stream**
Num. times it can process data	Multiple times	Once
Processing Time	Unlimited	Restricted
Memory usage	Unlimited	Restricted
Result type	Accurate	Approximate
Processing topology	Centralized./Distrib.	Distributed
Fault Tolerance	High	Moderate

Batch and stream processing paradigms, are combined in the lambda architecture to analyze big data in a real-time [70]. Lambda architecture is a big data concept that combines batch and stream processing in a single architecture. Stream processing is used as a fast path for timely approximate results, and a batch offline path for late accurate results. In the lambda architecture, stream data is used to update batch processing parameters of an off-line training. The architecture combines traditional batch processing over a historical database with real-time stream processing analysis.

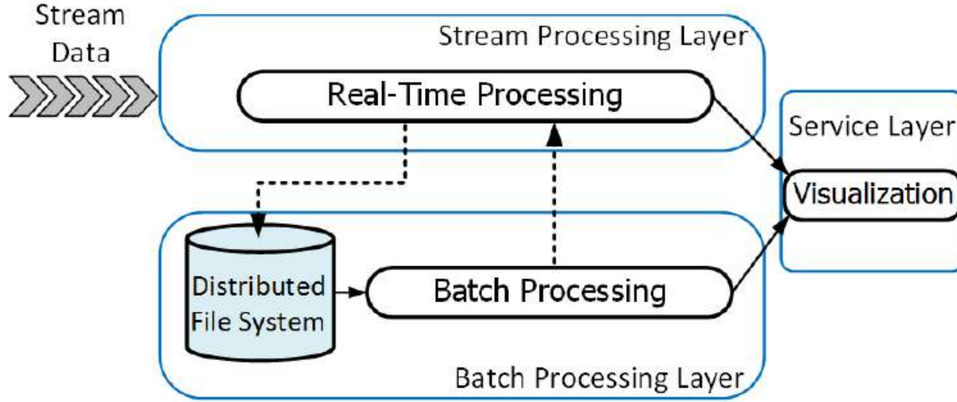


Figure 3.1: The three-layered lambda architecture, which combines stream with batch processing: stream processing, batch processing, and service layers.

As shown in Figure 3.1, the lambda architecture has three layers: the stream processing layer, the batch-processing layer, and the service layer. The stream processing layer deals with the incoming data in real-time. The batch-processing layer analyzes a huge amount of stored data in a distributed way through techniques such as map-reduce. Finally, the service layer combines the obtained information of the two previous layers to provide an output composed by analytic data to the user. Therefore, the lambda architecture goal is to analyze stream data accurately and in real-time, even with an ever-changing incoming rate to obtain results in real-time based on historical data.

3.2 The Stream Processing

The stream processing is modeled through a Directed Acyclic Graph (DAG). The graph is composed by source data node which continuously emit samples and interconnected processing nodes. A data stream ψ is an unbounded set of data, $\psi = \{D_t | t > 0\}$ where a point D_t is a set of attributes with a timestamp. Formally, one data point is $D_t = (\mathbf{V}, \tau_t)$, where $\mathbf{V}$ is a n-tuple, in which each value corresponds to an attribute, and τ_t is the time stamp for the t-th sample. Source nodes emit tuples

or messages that are received by Processing Elements (PE). Each PE receives data on its input queues, performs computation using local state and, finally, produces an output to its output queue. Figure 3.2 shows the conceptual stream processing system architecture.

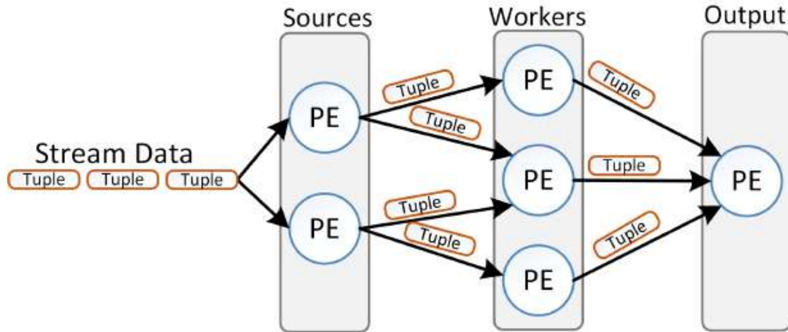


Figure 3.2: stream processing architecture. Processors Elements (PE) interconnected to create a directed acyclic graph. The data stream is received by the PE sources, they are immediately processed by workers and finally are aggregated at output. The output is a processing element that performs a specific function on the data, for example to visualize the data.

A number of requirements must be met on distributed stream processing platforms, Stonebraker *et al.* highlight the most important [4]. The ability to process data online without the need to store it for operations is critical to maintaining low latency, since storage operations such as writing and reading on disk add unacceptable processing delays. In addition, ideally the processing system should be active, depending only in its own policies to operate on the data without relying on external instructions. Due to the large volume, data must be partitioned to be treat by them in parallel. High availability and fault recovery are also critical in stream processing systems. In low latency applications, recovery must be fast and efficient, providing processing guarantees. Thus, stream processing platforms must provide resiliency mechanisms against imperfections or failures, such as delays, data loss or out-of-order samples, which are common in distributed stream processing in computational clusters. Besides, processing systems must have a highly optimized execution mechanism to provide real-time response for applications with high data rates. Therefore, the ability to process millions of messages per second with low latency, within microseconds, is essential. To achieve this performance, platforms must minimize the communication overhead between distributed processes.

Active backup, passive backup, and upstream backup algorithms are used by Distributed Stream Processors (DSP) to provide data processing guarantees upon failures. Furthermore, platforms must store data efficiently, access and modify status information, and combine them with data streams in real time. This combination allows the adjustment and verification of algorithms for better performance.

3.3 Stream Processing Platforms

Stream processing platforms have been researched since the 90s, presenting an evolution in three generations. First-generation platforms were based on database systems that evaluate rules expressed as condition-action pairs when new events arrive. These systems were limited in functionality and also were not designed for large volumes of stream data. Examples of this generation systems include Starburst [71], Postgres [72] and NiagaraCQ [73]. The company Apama[1], founded in 1999, was the first real-time, event-driven analysis application company focused on business. The technology provided by the Apama platform allowed to monitor events, analyze them and perform actions in milliseconds.

Second-generation systems focus on extending the Structured Query Language (SQL) to process stream data by exploring the similarities between a stream and an online query. In May 2003, at Stanford University, the STanford stREam datA Manager (STREAM) [74] project was created. The STREAM project is considered to be one of the first general-purpose Data Stream Management Systems (DSMS). The STREAM project prompted the foundation in 2003 of Coral8[2]. In 2007, Coral8 launched a commercial platform, based on Microsoft technologies, capable of processing and analyzing thousands of SQL requests per second. The Aurora [66] project was launched in 2002 in a collaboration with Brandeis University, Brown University and MIT. The main drawback of Aurora is that the project was designed as a single, centralized stream processing mechanism. A new distributed version was released in 2003, called Aurora*. One last version was officially released under the name Borealis [67], with some other improvements, such as fault tolerance. The Medusa [75] project used the Borealis distribution to create a federated stream processing system, in which the distributed nodes are controlled by a single entity. Borealis and Medusa became obsolete in 2008. The Aurora/Borealis projects boosted in 2003 the founding of the company StreamBase System[3], which commercially launched the StreamBase platform for Complex Event Processing (CEP) for decision-making purposes. The University of Berkeley, in 2003, created a language for continuously executing SQL queries based on the Postgres database system called TelegraphCQ [76]. Based on the TelegraphCQ, the Truviso[4] was created in 2009, and in 2012 Truviso was acquired by Cisco. In 2006, Cornell University created Cayuga [77], a state-of-the-art publish/subscribe system, which developed a simple

[1]The first real-time event analysis company, Apama Stream Analytics, was sold in 2005 to Progress Software Coorporation for $ 25 million. https://www.softwareag.com/corporate/products/apama_webmethods/analytics/default.html Accessed April 2018.

[2]Sold to Aleri in 2009.

[3]Sold to TIBCO Software in 2013.

[4]Truviso Analytic http://jtonedm.com/2009/03/03/first-look-truviso/ Accessed April 2018

query language to perform processing over data that scales both with arrival rate of events as well as number of queries. Cayuga was replaced by Cougar[5] and is still an active research.

Third-generation systems have emerged to address the need for Internet companies to process large volumes of data produced at high speed. The main focus of this generation is the scalable distributed processing of data in computational clusters. Google revolutionizes distributed processing by proposing the MapReduce [14] programming model for scalable parallel processing of large volumes of data in clusters. The key idea to spread-process-combine is used to scalably perform different tasks in parallel on commodity servers in a cluster. The Hadoop [15] platform is the Open Source implementation of MapReduce to perform analytics on big data. However, due to the high latency that MapReduce produces, some projects have been proposed to perform real-time data stream analysis. The Spark project [26] replaces Hadoop's MapReduce to perform memory operations that Hadoop performs on data retrieved from the disk. The open source platforms Storm [25] and Flink [27] are proposed for stream processing. The Spark platform proposes an extension for stream processing in micro-batches, the Spark streaming. Next, Apache Storm, Apache Flink and Apache Spark streaming are described as third-generation open source scalable platforms.

3.3.1 Apache Storm

Apache Storm[6] [25] is a real-time stream processor, written in Java and Clojure. Stream data abstraction is called tuples, composed by the data and an identifier. In Storm, applications consist of topologies forming a directed acyclic graph composed of inputs nodes, called Spouts, and processing nodes, called Bolts, and edges represent data stream. Spouts are responsible for abstracting incoming data into tuples that flow through the graph. Each Bolt executes a user-defined function considered as atomic. A topology works as a data graph in which nodes process the data as the data stream advance in the graph. A topology is analog to a MapReduce Job in Hadoop. Both Bolts and Spouts are parallelizable and can be defined with a degree of parallelism that indicates the number of competing tasks present in each node. An example of a topology with two Spouts and three Bolts is shown in Figure 3.3. The grouping type used defines the link between two nodes in the processing graph. The grouping type allows the designer to set how the data should flow in topology.

[5]Cougar processing http://www.cs.cornell.edu/database/cougar/, Accessed April 2018

[6]Nathan Marz, PhD from Stanford University, working on BackType, developed Storm in 2011, a framework for distributed stream processing, to handle in real time the large number of messages (tweets) received by Twitter company. The BackType company is acquired by Twitter and Storm becomes open source, migrating to the Apache Foundation in 2013.

Storm has eight data grouping types that represent how data is sent to the next graph-processing node, and their parallel instances, which perform the same processing logic. The main grouping types are: *shuffle*, *field*, and *all* grouping. In *shuffle* grouping, the stream is randomly sent across the Bolt instances. In *field* grouping, each Bolt instance is responsible for all samples with the same key specified in the tuple. Thus, a Bolt will be responsible for all samples of a certain type and be able to concentrate the information regarding such group. Finally, in *all* grouping, samples are sent to all parallel instances.

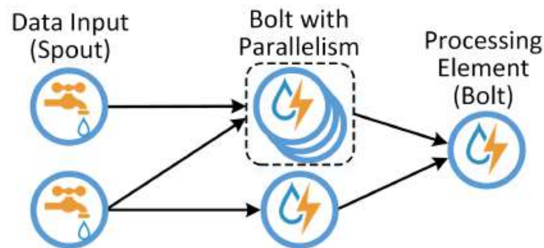


Figure 3.3: Storm topology with element processors *Spouts* and *Bolts*. *Spouts* are input nodes while *Bolts* are nodes that produce a processing in the tuples.

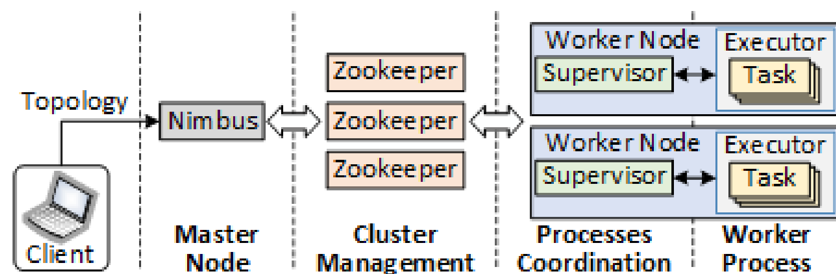


Figure 3.4: Storm architecture. Nimbus receives topologies and communicates to Supervisors that coordinate process in workers. All the coordination between Nimbus and Supervisors is made by *Zookeeper* who stores the cluster state.

Figure 3.4 shows the coordination processes in a Storm cluster. The master node, Nimbus, receives a user-defined topology. In addition, Nimbus coordinates each process considering the topology specification, i.e., it coordinates Spouts and Bolts instantiation and their parallel instances. The Zookeeper is responsible for managing the worker nodes and for storing state of all elements of the system. At each worker node, a supervisor monitors the executors, which are processes responsible for performing one or more tasks. Supervisors report the status and availability of the executors through a periodic heartbeat mechanism, allowing Nimbus to identify system failures. Executor failures are handled by the supervisors themselves, who restart the corresponding processes on the worker node. A supervisor failure is handled by the Nimbus, which can relocate all tasks from the failing supervisor to

another worker node. If Nimbus fails, the system is still capable of running all ongoing topologies, but the user is not able to submit new topologies. After recovery, the Nimbus and the supervisors can resume the last state stored in the Zookeeper.

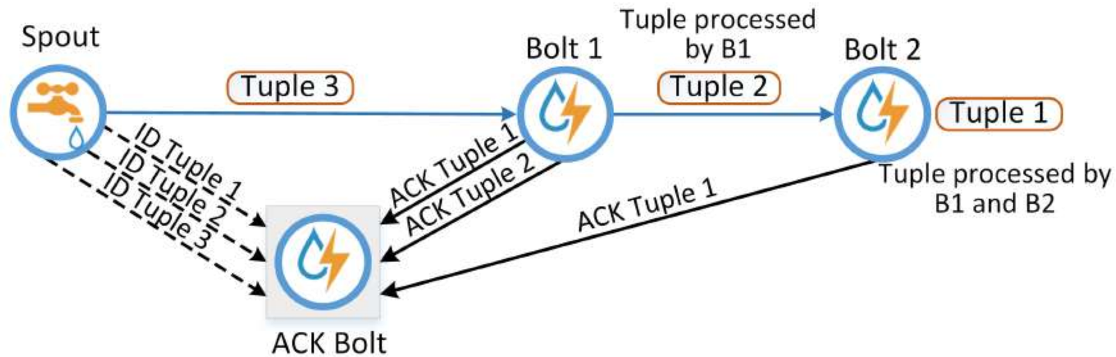


Figure 3.5: At-least-once delivery semantic used in Apache Storm. Every time a tuple is emitted by a spout, a record is saved in the acknowledge bolt. After the tuple leaves the topology, the ack bolt acknowledges all operators.

Apache Storm uses storage and acknowledgment (ACK) mechanisms to ensure tuples are processed even after a failure. For this, all tuples are identified by the spouts and their identifiers are sent to a special Bolt, which stores the state of each tuple. An example of topology with ACK Bolt is shown in Figure 3.5. For each processed tuple, a Bolt should send a positive acknowledgment (ACK) to the ACK Bolt. If all the tuples receive an ACK for each Bolt, the acknowledgment Bolt ignores the IDs and informs the Spout that the processing has been successful. Otherwise, the acknowledgment Bolt asks the Spout to resubmit all tuples and the system goes back to the point of failure. Non-receipt of an ACK is recognized by the expiration of a timer timeout defined in the acknowledgment Bolt. This ensures message delivery semantics "at least once", where each tuple is processed one or more times in the case of reprocessing. It is still possible to disable ACK Bolt for applications that do not require processing guarantees.

3.3.2 Apache Flink

The Apache Flink[7] [78] is a hybrid processing platform that supports stream and batch processing. The core of Flink is stream processing, making batch processing a special case. Figure 3.6 shows how the analytical tasks of Flink are abstracted in Directed Acyclic Graphs (DAG). The graph is formed by four components: sources, operators, exit taps, and records that run through the graph. The abstraction of

[7]Flink was born in 2010 from a European research project called "Stratosphere: Information Management in the Cloud" developed in collaboration with the Technical University of Berlin, Humboldt-Universität zu Berlin and Hasso-Plattner-Institut Potsdam. In 2014, Stratosphere renames the project to Flink and opens its code at the Apache Foundation.

the topology is performed through programming in Java or Scala. As in Storm, the task division is based on a master-worker model. Figure 3.7 shows the Apache Flink architecture.

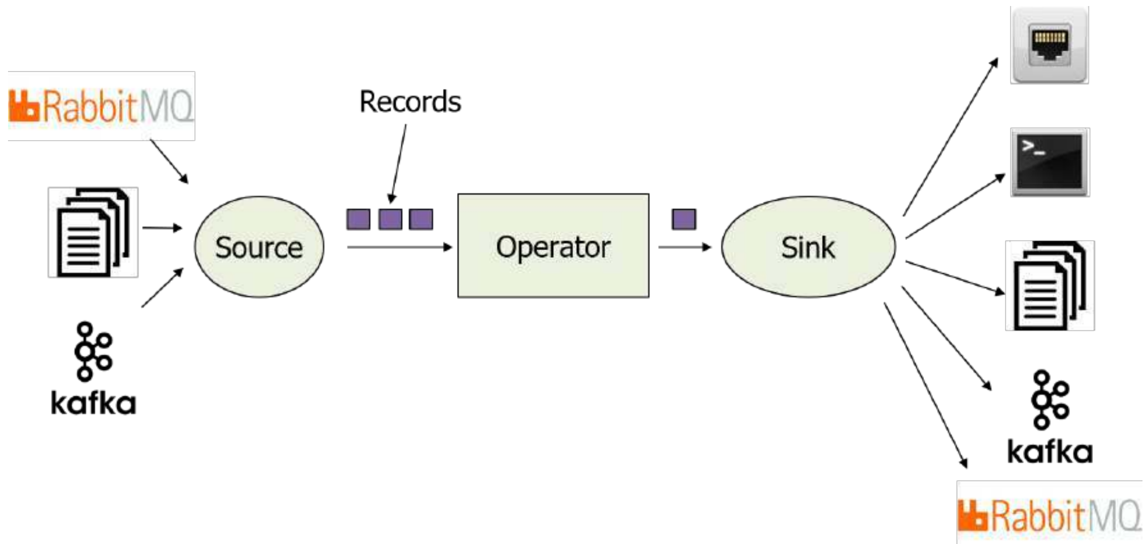


Figure 3.6: Flink Topology composed by our elements: sources, operators, records and sinks.

The Flink master node is called the job manager and interacts with client applications with responsibilities similar to the Storm master node (Nimbus). The job manager receives client applications, organizes the tasks and sends them to the worker nodes, which are called task managers. In addition, the work manager maintains the status of all executions and each worker. The states of workers are informed through a periodic heartbeat signal mechanism. The task manager has a function similar to the worker node in the Storm. Task managers perform tasks assigned by the job manager and exchange information with other task managers when needed. Each task manager provides slots of processing to the computational cluster, which are used to execute tasks in parallel.

The abstraction of the data stream in Flink is called DataStream and is defined as a sequence of partially ordered records. Partially because there is no guarantee of order if an operator element receives more than one data stream as input. DataStreams are similar to Storm tuples and receive stream data from external sources, such as message queues, sockets. DataStream programming supports several native functions for operating on data streams, such as *map*, *filtering*, *reduction*, *join*, which are applied incrementally to each entry, generating a new DataStream. Each of these operations can be parallelized by configuring a parallelism parameter. Thus, parallel instances of the operations are assigned to the available processing slots of the task managers to simultaneously handle DataStream partitions.

Figure 3.8 shows the checkpoint recovery method used by Apache Flink. Flink

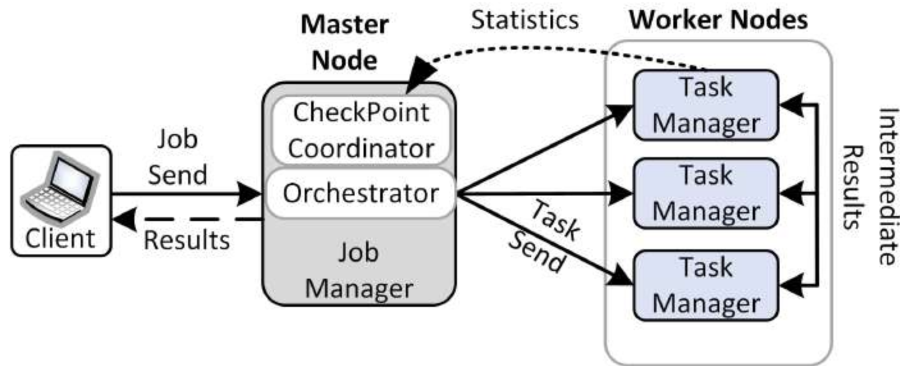


Figure 3.7: Architecture of Flink system. The Job manager receives jobs from clients, divides the jobs into tasks, and sends the tasks to the workers. Workers communicate statistics and results.

has a reliable delivery semantics of *exactly-once* messages. This semantics is based on the fault tolerance scheme with checkpoints, or checkpoint barriers, so that the system can return on failure. Barriers work as control registers and are regularly injected into the data stream by the source elements to flow through the graph along with the sample records. When a barrier passes through an operator element, it performs a snapshot of the state of the system. The snapshot consists of storing the state of the operator, for example the contents of a sliding window or a custom structure of data, and its position in the data stream. After an alignment phase between operators to make sure the barrier has crossed all the operators that handle that stream, operators write the snapshot in a file system, such as Hadoop Distributed File System (HDFS). In case of software, node or network failure, Flink interrupts the DataStream. The system immediately resets the operators and restarts from the last successful snapshot capture. As in Storm, Flink fault tolerance is guaranteed based on the premise that the system is preceded by a persistent forwarding message system, such as Apache Kafka. A persistent forwarding message system store the information in distributed nodes to ensure fault-tolerance. In the case of Apache Kafka, messages are abstracted in topics. Each topic has a partitioned log replicated in the cluster. Logs are a persistent ordered data structure that only supports append. The logs in Apache Kafka are stored in disk. In the special case of batch processing, there is no fault tolerance scheme, and if a fault occurs, the entire operation must be restarted from scratch.

3.3.3 Apache Spark Streaming

Spark is a project initiated by UC Berkeley and is a platform for distributed data processing, written in Java and Scala. Spark has different libraries running on the top of the Spark Engine, including Spark Streaming [26] for stream processing.

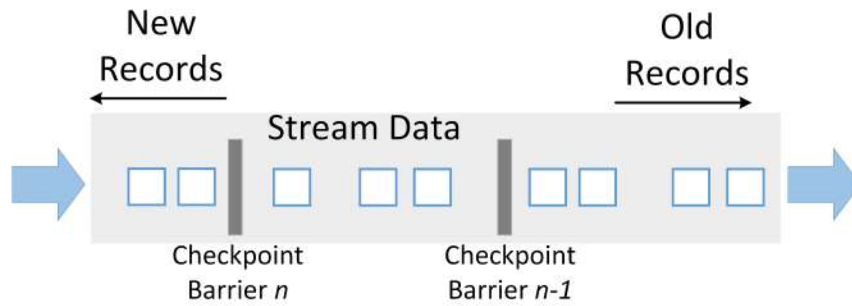


Figure 3.8: Barriers are injected in source elements and flow through the graph together with the records, flow downstream and trigger state snapshots when they pass through operators. When an operator receives a barrier from every incoming stream, it checkpoints its state to storage.

Apache Spark is a parallel engine, which executes the MapReduce technique. Apache Spark is optimized to execute MapReduce jobs into the main memory, improving performance while compared with Hadoop [26].

The stream abstraction is called Discrete Stream (D-Stream) defined as a set of short, stateless, deterministic tasks. In Spark, stream computation is treated as a series of deterministic batch computations on smaller intervals. Similar to MapReduce, a job in Spark is defined as a parallel computation that consists of multiple tasks, and a task is a unit of work that is sent to the Task Manager. As Figure 3.9 shows, when a stream enters Spark, it divides data into micro-batches, which are the input data of the Resilient Distributed Dataset (RDD), the main class in Spark Engine, stored in memory. A DStream is a potentially infinite sequence of Resilient Distributed Dataset (RDD). Then the Spark Engine executes by generating jobs to process the micro-batches. RDD are the basic elements in the Spark Engine and are partitioned across all the nodes in the cluster. RDD are by definition immutable, when an operation, called transformation, such as `map()`, `join()`, `filter()` is applied, it creates a new RDD. When a transformation is applied to the RDD, Spark does not execute it immediately, instead it creates a lineage. A lineages, also known as RDD operator graph, keeps track of all transformations that been applied on a RDD, including from where it has to read the data and all the transformations applied to that RDD. All transformations are computed when an action such as `save()/display()` is called in the driver program. The driver program is the interaction between the spark cluster and the client. The driver program receives a program that declares the transformations and actions on that must be executed on the RDDs, submitting those requests to the worker nodes.

Linage allows Spark to recover the last operation performed in case of failure.

Figure **??** shows the layout of a Spark cluster. Applications or jobs within the Spark run as independent processes in the cluster which is coordinated by the

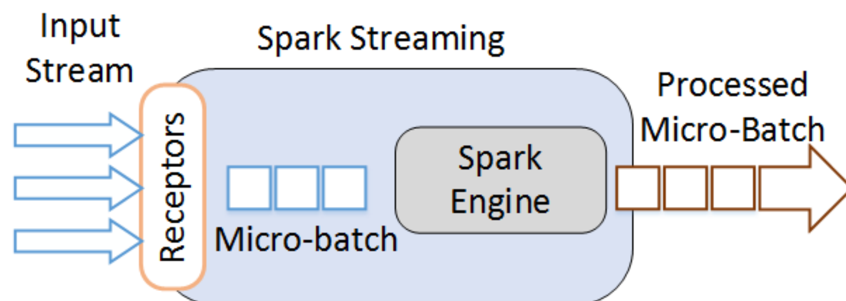


Figure 3.9: Micro-batch processing used in Spark stream. The input streams are received by receptors and data are transformed in micro-batch. Micro-batches are executed in a traditional Map-Reduce Spark Engine.

master or Driver Program, responsible for scheduling tasks and creating the `Spark Context`. The `Spark Context` connects to various types of cluster managers, such as the Spark StandAlone, Mesos or Hadoop YARN (*Yet Another Resource Negotiator*). These cluster managers are responsible for resource allocation between applications. Once connected, Spark executes tasks within the task managers, which perform processing and data storage, equivalent to Storm workers, and results are then communicated to the `Spark Context`. The mechanism described in Storm, in which each worker process runs within a topology, can be applied to Spark, where applications or jobs are equivalent to topologies. A disadvantage of this concept in Spark is the message exchange between different programs, which is only done indirectly for example writing data to a file. This concept worse the latency that could be around seconds in applications of several operations.

Because Spark operates on data stored in volatile memory, there is a need to provide fault tolerance for data while it is being processed, not just after saving to disk as done on Hadoop. Spark has "*exactly-once*" message delivery semantics. The idea is to process a task on several distinct working nodes and, in the event of a failure, the processing of the micro-batch can be redistributed and recalculated. The state of the RDDs is periodically replicated to other working nodes. Tasks are then discretized into smaller tasks performed on any node, without affecting execution. Thus, failing tasks can be thrown in parallel, evenly distributing the task, without affecting performance. This procedure is called parallel recovery. The semantics of "*exactly-once*" reduce the overhead compared to *upstream backup*, where all tuples must be positively recognized, as in Storm. However, micro-batch processing has disadvantages. The configuration and distribution of each micro-batch may take longer than the arrival rate of the native stream. Consequently, micro-batches are stored in a processing queue affecting latency.

3.3.4 Fault Tolerance Schemes on Distributed Processing Platforms

A robust fault tolerance scheme is essential for distributed processing platforms running on cluster, which are sensitive to node failures in network and software. It should be noted that a datacenter has a structure in computational clusters, in which nodes are low-cost commercial off-the-shelf (COTS) servers. In batch processing systems, latency is acceptable and as a result, the system does not need to recover quickly from a failure. However, in real-time systems since the data is not stored, failures can mean data loss. Therefore, fast and efficient recovery is important to avoid loss of information [79].

The most common form of failure recovery is storage and forwarding, also known as *upstream backup*. Considering a processing topology, the algorithm uses the parent nodes to act as backups, storing and temporarily preserving the tuples in their output queues until their downstream neighbors process them and send positive acknowledgment (ACK). Every tuple must be individually positively recognized with an ACK. If any of these downstream neighbors fails, an ACK will not be sent, and by timer overflow, the parent node reproduces the tuples on another node. Another form of positive recognition is by group tuples. Identifying that a tuple is missing, the entire group of tuples is reproduced.

A disadvantage of this approach is the long recovery time since the system must wait until the protected node takes over. To address this problem, in [69] the *parallel recovery* algorithm is proposed. In this algorithm, the system periodically checks the states replicating asynchronously to other nodes. When a node fails, the system detects the missing partitions and launches tasks to retrieve them from the last checkpoint. Many tasks can be launched at the same time to calculate different partitions on different nodes. Thus, *parallel recovery* is faster than the *upstream backup.*

Another solution is proposed in [27] based on the *Asynchronous Barrier Snapshot* (ABS) algorithm. The main idea is to mark the overall state of a distributed system. In ABS algorithm, a snapshot is the overall state of the rendering distributed system represented as a graph. A snapshot captures all the information necessary to restart the calculation of that particular execution state. A barrier separates the record set in two sides. From one side, the records that accompany the current snapshot, and from the other side the records that are inserted into the next snapshot. Barriers do not interrupt data flow. Several different snapshot barriers may be in the stream at the same time, which means that multiple snapshots may occur simultaneously. When a source receives a barrier, the source takes a snapshot from its current state and then transmits the barrier to all the outputs. When a non-source task receives a

barrier from one of its inputs, it blocks that input until it receives a barrier of all the inputs. When the barriers were received from all entries, the task takes a snapshot from its current state and transmits the barrier to its outputs. The task then unlocks its input channels to continue its computation. Thus, disaster recovery reverts all states of the operator to their states taken from the last successful snapshot and restarts the inbound streams from the last barrier for which there is a snapshot.

The delivery assurance semantics that a system offers to process a sample can be divided into three types: "Exactly once", "at least once", and "at most once". The simplest semantics is "at most once", also known as "may be once" where there is no error recovery, that is, the samples are processed or lost. When a failure occurs, data can be routed to another processing element without losing information. The "at most once" semantic should be used in applications where the occasional loss of a message does not leave the system in an inconsistent state. In the semantics "exactly once" positive recognitions are individual by tuple. Also, in the "exactly once", the system guarantees that a source that crashes will eventually restart. The system must keep track of calls on sinks that have crashed, and allows them to be later adopted by a new sink. In the semantics "at least once", also known as "once or more", the error correction is done together for a group of samples, so if an error occurs within any of the samples, the entire group is reprocessed and therefore it is possible that some samples are processed one more time. In "at least once" semantic, the source continues to send tuples to the sink until it gets an acknowledgement. If one or more acknowledgements are lost, the server may execute the call multiple times. The semantics "at least once" is less costly than "exactly once", which requires individual confirmation for each processed tuple.

Table 3.2 presents a summary of features underlined in the comparison of the stream processing systems. The programming model can be classified as compositional and declarative. The compositional approach provides basic building blocks, such as Spouts and Bolts on Storm and must be connected together in order to create a topology. On the other hand, operators in the declarative model are defined as higher-order functions, that allow writing functional code with abstract types and the system will automatically create the topology.

3.4 Performance Evaluation of the Platforms

This section evaluates the processing rate and behavior during the node failure of the three stream processing platforms presented: Apache Storm version 0.9.4, Apache Flink version 0.10.2 and Apache Spark stream version 1.6.1, with microbatch size set to 0.5 seconds. The evaluated application is a threat detection system with a neural network classifier programmed in Java. The experiments were performed

Table 3.2: Overview of the comparison between Stream Processing Systems.

	Storm	**Flink**	**Spark stream**
Stream Abstraction	Tuple	DataStream	DStream
Build Language	Java/Closure	Java/Scala	Java/Scala
Messages Semantic	At least once	Exactly one	Exactly one
Failure-Tolerance Mechanism	Upstream Backup	Check-point	Parallel Recovery
API	Compositional	Declarative	Declarative
Failures Subsystem	Nimbus, Zookeeper	No	No

in an environment with eight virtual machines running on a server with the Intel Xeon processor E5-2650 at 2.00 GHz and 64 GB of RAM. The experiment topology configuration is a master and seven working nodes for the three systems evaluated. The results are presented with a confidence interval of 95%.

Apache Kafka in version 0.8.2.1, which operates as a publish/subscribe service, was used to enter data at high rates in stream processing systems. In Kafka, the samples, or events, are called messages, name used from now on. Kafka abstracts the flow of messages into threads that act as buffers or queues, adjusting different rates of production and consumption. Therefore, producers record the data in topics and consumers read the data from those topics. The dataset used is a security dataset created by the authors [80], which was replicated to obtain enough data to evaluate the maximum processing the system can operate on.

3.4.1 Experiment Results

The first experiment evaluates the performance of platforms in terms of processing [81]. The full content of the dataset is injected into the system and then it is replicated as many times as necessary to create a large volume of data. The experiment calculates the rate of consumption and processing of each platform. Also, the parallelism parameter was varied, which represents the total number of cores available for the cluster to process samples in parallel. Figure 3.10 shows the results of the experiment. Apache Storm has the highest throughput. For a single core, unparalleled, Storm already shows better performance with a flow rate at least 50% higher when compared to Flink and Spark streaming. Flink has a linear growth, but with values always inferior to those of Apache Storm. The processing rate of Apache Spark streaming, when compared to Storm and Flink, is much lower and

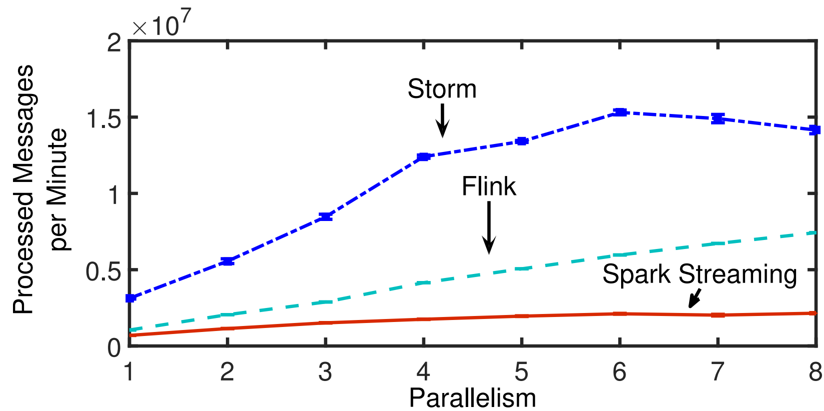


Figure 3.10: Throughput results of the platforms in terms of number of messages processed per minute as function of the task parallelism for GTA/UFRJ Dataset.

this is due to the use of a microbatch. Each microbatch is pooled prior to processing, generating a delay in each processed sample. Apache Storm behavior is linear up to four-core parallelism. Then, the processing rate grows until the parallelism of six, in which the system saturates. This behavior was also observed in Apache Spark streaming with the same parallelism of six cores.

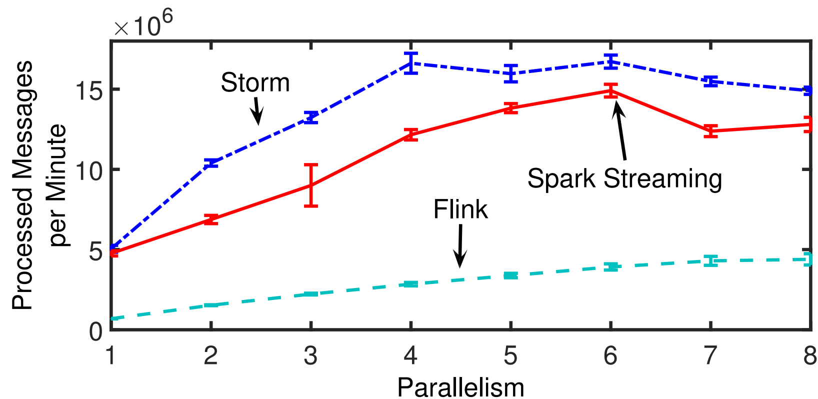
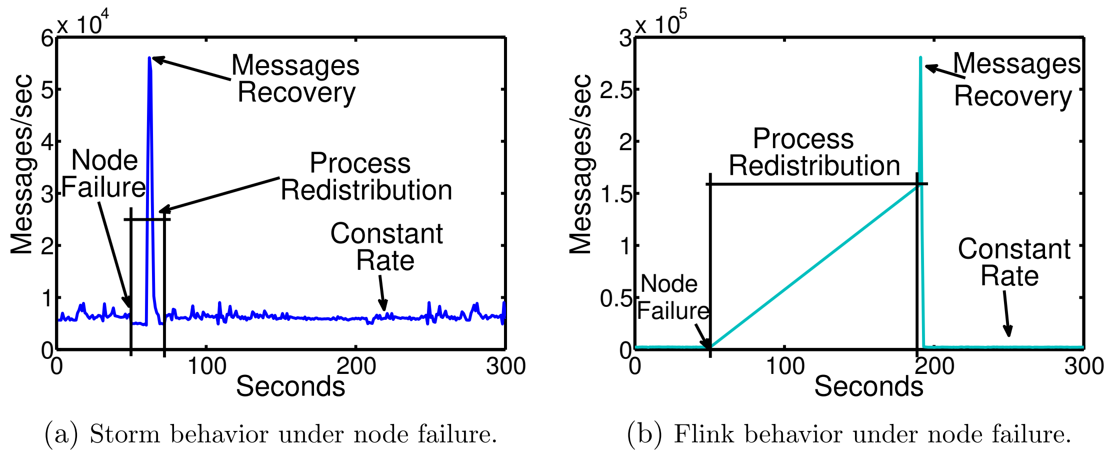


Figure 3.11: Throughput results of the platforms in terms of number of messages processed per minute as function of the task parallelism. Evaluation of the word-count performance in Twitter Dataset.

All three platforms offer the word-count application as examples of tutorials, so we show this result for an unbiased comparison that is not affected by code implementation on each platform. This experiment counts the number of times each word appears in a text, using a dataset containing more than 5,000,000 tweets [82]. Figure 3.11 shows the performance of the three systems in the `wordcount` program. This experiment shows a similar result to that shown previously. However, in this case Spark streaming outperforms Apache Flink.

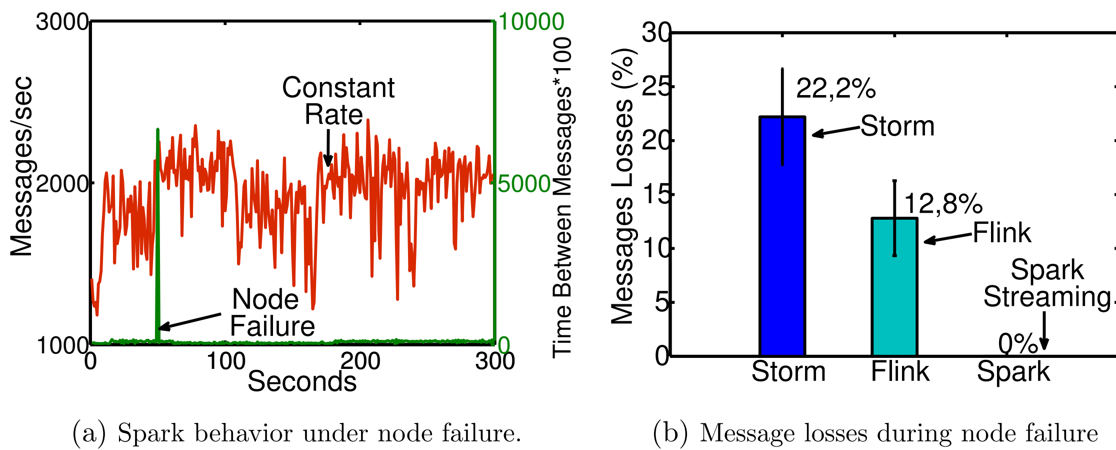
The second experiment shows the system behavior when a node fails. Messages are sent at a constant rate to analyze the system behavior during the crash. The node



(a) Storm behavior under node failure.

(b) Flink behavior under node failure.

Figure 3.12: Storm and Flink behavior during a node failure. A failure is produced at 50 seconds. a) Storm and b) Flink system behavior after detecting the failure and consisting of process redistribution and message recovery procedures.



(a) Spark behavior under node failure.

(b) Message losses during node failure

Figure 3.13: Spark stream behavior during a node failure and message losses comparison. a) The Spark system behavior under failure, indicating that it keeps stable and does not lose messages. b) Percentage of message losses.

failure is simulated by turning off a virtual machine. Figures 3.12 and 3.13 show the behavior of the three systems before and after a worker node failure at 50 seconds. Apache Storm takes some time in the redistribution processes after the fault was detected. This time is due to communication with the Zookeeper. The Zookeeper has an overview of the cluster and reports the state for Nimbus in Storm, which reallocates the processes on other nodes. Soon after this redistribution, the system retrieves Kafka messages at approximately 75 seconds. Although the system can quickly recover from node failure, during the process there is a significant message loss. A similar behavior is observed in Apache Flink. After detecting the failure at approximately 50 seconds, the system redistributes the processes for active nodes. Flink does this process internally without the help of any subsystem, unlike Apache Storm that uses the Zookeeper.

Figure 3.12b shows that the time period in which Flink redistributes processes is much greater than the time spent in Apache Storm. However, message recovery is also higher, losing some messages during the redistribution process. Figure 3.13a shows Spark streaming behavior during a failure. When a failure occurs at approximately 50 seconds, the system behavior is basically the same as before. This is due to the use of tasks with microbatch that are quickly distributed without affecting performance. Spark stream shows no message loss during fail. Thus, despite the low performance of Spark stream, it could be a good choice in applications where resilience and processing all messages are necessary.

Figure 3.13b shows the comparison of lost messages between Storm, Flink and Spark. The result shows that Spark had no loss during the fault. The measure shows the percentage of lost messages by systems, calculated by the difference of messages sent by Apache Kafka and messages analyzed by the systems. Thus, Apache Flink has a smaller loss of messages during a fault with about a 12.8% compared to 22.2% in Storm. We obtain the result with a 95% confidence interval.

We can conclude that if we want to prioritize throughput, Apache Storm is the best choice. On the other hand, if fault tolerance and "exactly once" message processing is required, Apache Spark must be employed.

3.5 The CATRACA Tool

CATRACA tool uses Network Function Virtualization (NFV) technology and the Network Function Virtualization Infrastructure (NFVI) to combine virtualization, cloud computing, and distributed stream processing to monitor network traffic and detect threats. The goal is to provide an accurate, scalable and real-time threat detection tool capable of meeting peaks of use, providing a high Quality of Service. Traffic monitoring and threat detection as a virtualized network function have two main advantages: the ability to self-adapt to different traffic volumes and the flexibility of installation and migration of sensors in the network to reduce the latency in monitoring [22]. Self-adaptation is reached with an elastic behavior, adapting to different traffic and processing rates. The system creates and destroy virtual machines when necessary. The system has flexibility of installation because runs on virtual machine that are hardware agnostic. Sensors are migrated using virtualization features. Thus, the tool analyzes large volumes of data, the Machine Learning techniques classify the traffic into normal or threat, and, finally, the knowledge extracted from the flows is presented in a user interface.[8] As shown in Figure 3.14 CATRACA is deployed as a Virtual Network Function (VNF). CATRACA sensors

[8]The tool, as well as its documentation and complementary information can be accessed at http://gta.ufrj.br/catraca .

are deployed in virtual networks. The goal of the sensors is to mirror traffic to the CATRACA cloud. CATRACA cloud is composed by Apache Kafka that receives the mirrored traffic and send it to Apache Spark, responsible for data processing. Apache Spark creates machine learning model that are stored in the Hadoop Distribute File System (HDFS) and finally, results are display in the ElasticStack that contains the Elastic Search and Kibana for data visualization.

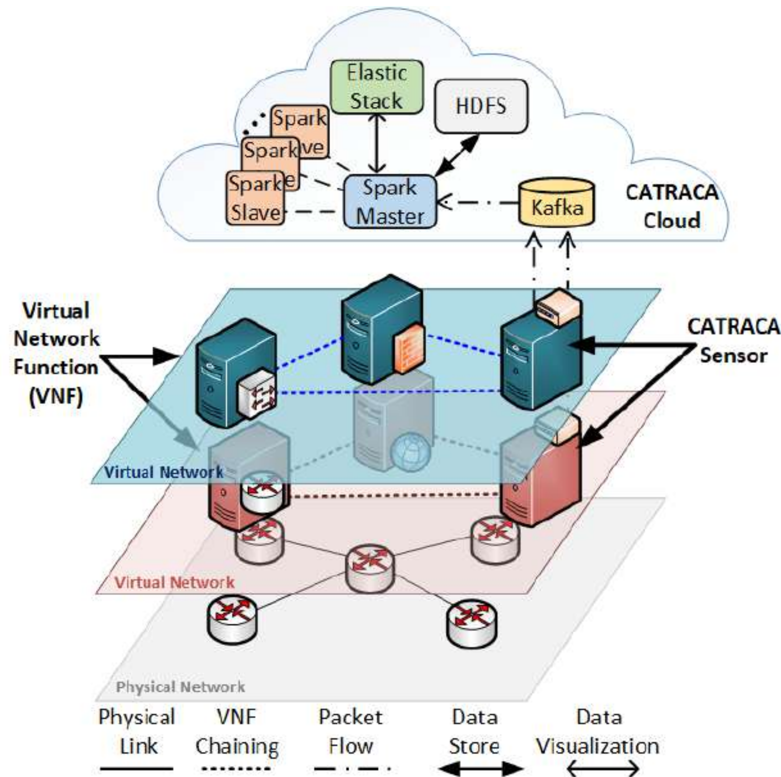


Figure 3.14: CATRACA as a Virtual Network Function. CATRACA sensors mirror traffic to Apache Kafka. Flows are then send to analyze in Apache Spark. Machine Learning models are obtained and stored in the Hadoop Distribute File System (HDFS) and results are display in the ElasticStack.

In CATRACA several sensors are distributed in different network locations. Then, the data are grouped to be processed in a centralized point. With sensors distribution, our tool is able to detect attacks in different network location and have a global view of the attack. As a consequence, a Distributed Denial of Service (DDoS), for example, is easier to detect.

CATRACA runs in two modes: offline and online modes. Offline mode is used to analyzes big security datasets stored and invariables with the time. Figure 3.15 shows the flowchart of the offline program in CATRACA. Boxes are the name of the functions and the arrows show the communication between the functions. Firstly, a file in `csv` format is uploaded from Hadoop Distributed File System (HDFS). The `readFile` function reads the file and transforms it in a Resilient Distributed Dataset (RDD) in the spark context. The file is passed to the next function abstracted

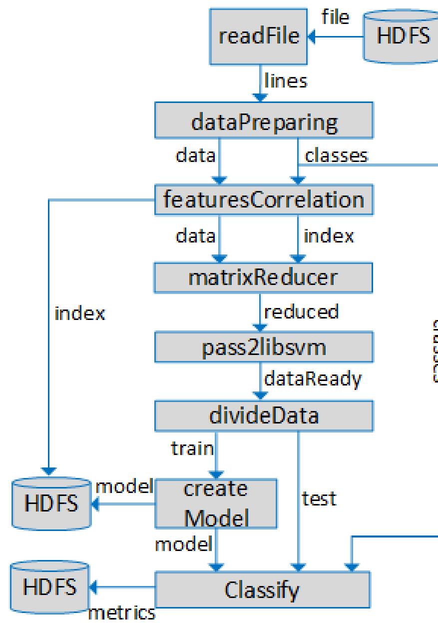


Figure 3.15: CATRACA running in offline mode.

by lines where each line represents a network flow. The `dataPreparing` function processes the lines, separating each feature by comma, converting the values to float, and also removing the label or classes for each line. Data and the classes are send separately. The `featureCorrelation` function uses our feature selection algorithm to select the most important feature for each flow. Each feature is represented by an index, from 1 to 45. Once selected, the indexes are stored in the HDFS. Next, the data and the indexes are send to the `matrixReducer` function, where the original matrix of lines is reduced in the index selected before. The `pass2libsvm` transforms the reduced matrix in a libsvm format. LibSVM is a library used for support vector machine classification, however, CATRACA only use the data format of this library as input for machine learning algorithms in the spark context. The format of the libsvm library is $< label >< index1 >:< value1 >< index2 >:< value2 >$ where *label* is the class of the flow, *index* are the features and *value* are the numerical values of the features. Once the data is ready, it passed to the `divideData` function, where data is divided in train and test set in a proportion of 70% for training and 30% for test. The train set is passed to the `createModel` function that creates the machine learning model. In CATRACA we use decision tree as machine learning model. The model is then stored in the HDFS for further use. Finally, the `Classify` function obtain the model and evaluate it with the test set. This function also compares the predicted values with the original dataset classes and the metrics such as accuracy, precision, and F1-score are obtained. The metrics are finally stored also in Hadoop File Distributed System.

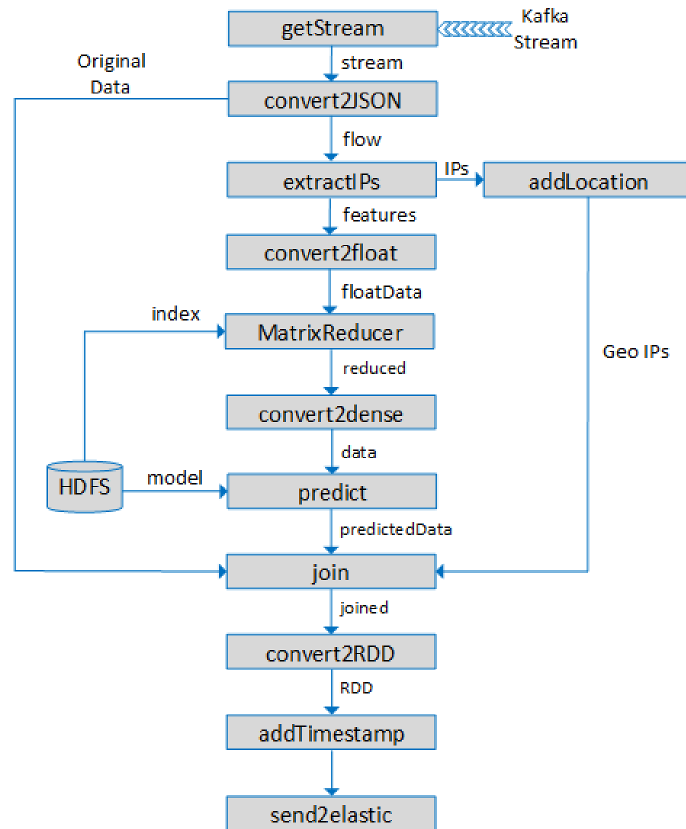


Figure 3.16: CATRACA running in online mode.

CATRACA online mode is presented in Figure 3.16. This mode works in a similar way that offline mode. However, in contrast to offline mode that analyses static data, online mode uses stream processing for dynamic data. As a consequence, the stream data arrives without any class because is generated in real-time. First, the `getStream` function get the streaming flows that came from Apache Kafka. In this function are defined the parameters of the Apache Kafka receiver inside the Apache Spark. Then, stream data is passed to `convert2JSON` function. In this function stream data is parsed to the JavaScript Object Notation (JSON), that is easier to handle. `ExtractIPs` function get the IPs source and destination address from each flow; these IPs are passed to the `addLocation` where the geographical coordinates of each IP are added. On the other side, the features without the IPs are send to the `convert2float` function. This function transforms all data into float values. Next, the `MatrixReducer` function is inherited from the offline mode. This function takes the stored indexes from the HDFS that were calculated in the offline mode and apply a reduction on the data. The reduced data is then converted to dense data. This step is similar to pass2libsvm in the offline mode, however the libsvm format cannot be used in a stream environment. The `predict` function load the machine learning model obtained during the offline model and classify the flows

in 0 as normal flow or 1 to alert. The classification is combined with the original data and with the geographical location in the `join` function, merging in single vector by flow. The merged data is converted to RDD. A timestamp when a flow is processed is added to enrich the data. Finally, the `send2elastic` function sets the parameters and adapts the data to send to elasticsearch.

3.5.1 CATRACA Architecture

The CATRACA architecture is composed of three layers: Visualization Layer, Processing Layer and Capture Layer, as shown in Figure 3.17.

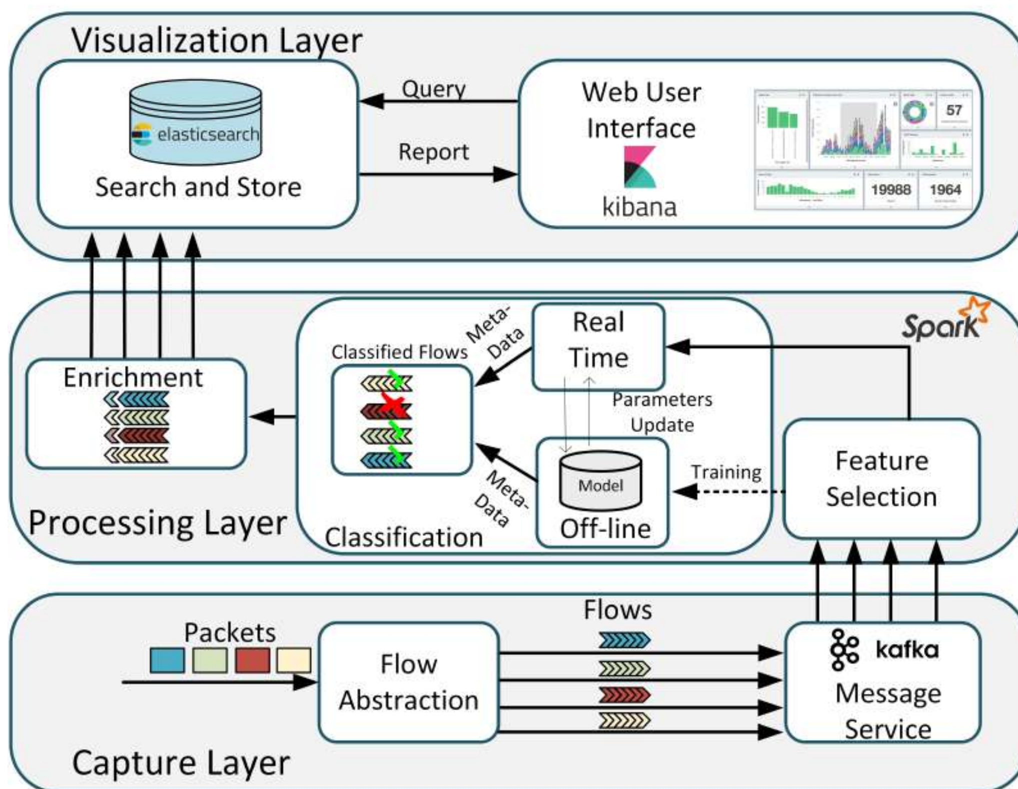


Figure 3.17: The layered architecture of the CATRACA tool: the capture layer, the processing layer, and the visualization layer.

The first layer, the Capture Layer, is responsible for capturing the packets. The packets are captured, through traffic mirroring, by the libpcap library. A python application based on flowtbag abstracts the packets into flows. Many open-source software exist to abstract packets into flow features such as tcptrace[9], flowtbag[10], Traffic Identification Engine (TIE)[11], flowcalc[12], Audit Record Generation and Uti-

[9]Tcptrace http://www.tcptrace.org, Accessed April 2018.
[10]flowtbag: https://github.com/DanielArndt/flowtbag, Accessed April 2018.
[11]Traffic Identification Engine http://tie.comics.unina.it/doku.php, Accessed April 2018.
[12]flowcalc http://mutrics.iitis.pl/flowcalc, Accessed April 2018.

lization System (ARGUS)[13], among others. We choose flowtbag, because it abstracts more packet features than the others. Flowtbag get 40 features: (5) flow tuple information (IP/Ports/protocol), (4) packets/byte in forward/backward direction, (8) packets statistics forward/backward direction, (8) time between packets forward/backward direction, (4) flow time statistics, (4) subflow packets/bytes forward/backward direction, (4) TCP flags, (2) Bytes used in headers, (1) type of service. An online version of flowtbag was implemented to handle streaming data.

We define a flow is defined as a sequence of packets with the same quintuple source IP, destination IP, source port, destination port, and protocol, during a time window. In all, 46 flow features are extracted and published in a producer/consumer service of Apache Kafka. This service operates as a low latency queue and data flow manipulation system, where queue features are consumed by the Processing Layer.

Figure 3.18 shows a simplified flow chart of the capture module program. First, the apache kafka producer is configured in `ConfigureKafka` function. In this step, the Apache Kafka server IP is configured as well as the name of the topic, queues offsets, etc. Then, `getPackets` function obtain the raw packets from the Network Interface Card (NIC). The program creates virtual queues for each flow. If the packets belongs to the same quintuple, a flow is created and the statistics are obtained in `getFlowStatistics` function. This function obtain statistics about packets and the flow it self, abstracting them in features. If the packets do not belong to a known flow, a new flow is created. Flow statistics are updated during a two second time window. Once this time finished, the flow with 45 features is published to kafka. In addition, a file is created and continuously updated with all the flows seen so far. The the dash line shows the original flowtbag program, and outside the box are the modules implemented in CATRACA to run in real-time.

The Processing Layer is instantiated in a dedicated cloud for classification and its core is the Apache Spark. In our case, the dedicated cloud is the Open Platform for Network Functions Virtualization (OPNFV). We choose OPNFV cloud due to the simplicity to manage Virtual Network Functions (VNF). The Spark framework has been chosen among the different flow-processing platforms because it presents the best fault tolerance performance [45], making CATRACA more robust in case of failure. Spark is implemented in a cluster following the master/slave model, where slaves have the capacity to expand and reduce resources, making the system scalable. Once the flow arrives in the Processing Layer, the feature selection algorithm [80], presented in Section 4.3.2, selects the most important characteristics for threat classification. In the processing step, the processed metadata is enriched through different information such as the geographical location of the analyzed IPs. Then, the flows are classified as malicious or benign through decision trees.

[13]ARGUS http://www.qosient.com/argus Accessed April 2018.

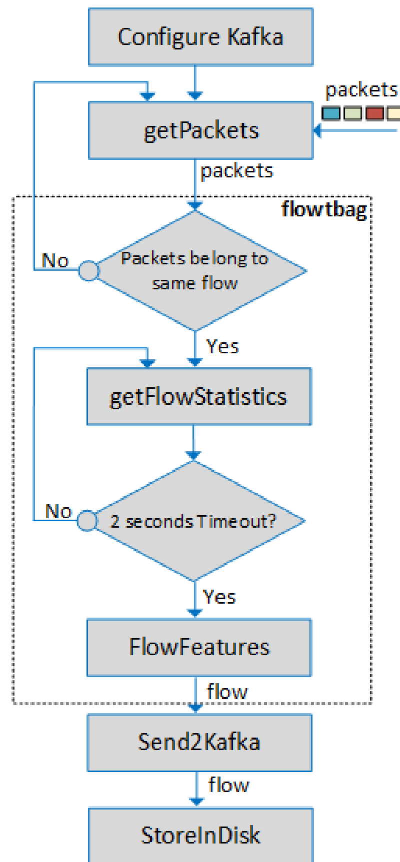


Figure 3.18: Flow Diagram of the Capture Module. An implementation of flowtbag in real time was used to handle streaming data.

Finally, the Visualization Layer is implemented using the Elastic Stack. The Elastic Stack allows custom event viewing in real time. Thus, the output of the Processing Layer is sent to the elasticsearch which provides a fast search and store service. The Elasticsearch[14] communicates through queries with the user interface that runs in the Kibana environment in which the results are represented to be viewed by the user.

Use Case: CATRACA for Traffic Classification

The classification begins with the preprocessing in selection of the most important characteristics of the flows using our feature selection algorithm [80]. The tool operates in either real time or offline mode. The offline traffic classification consists of processing of the mini-batches Spark platform. In this mode, large volume datasets are loaded in a distributed file system, such as the Hadoop Distributed File System (HDFS). The dataset is separated into a set of training and a test set at a ratio of 70 % to training and 30 % to the test.

[14] ElasticSearch and Kibana are open-source code and belong to the ElasticStack. https://www.elastic.co/products, Accessed April 2018.

A Decision Tree (DT) algorithm is trained to obtain the classification model. The DT classification algorithm is implemented in core of CATRACA, due to its training speed allied to its high accuracy and precision [83]. The decision tree is a greedy algorithm that performs a recursive binary partitioning of the resource space. Each sheet, in the case of CATRACA a feature or a combination of them, is chosen by selecting the best separation from a set of possible divisions, to maximize the gain of information in a tree node. The division into each node of the tree is chosen from the $argmax_d GI(CD, d)$, where $argmax$ is the point where function gets its maximum value, $GI(CD, d)$ is the information gain when a division d is applied to a set of CD data. The idea of the algorithm is to find the best division between features to classify threats. For that we use the heuristic of Information Gain. The gain of information GI of the tool CATRACA is the impurity of Gini,$\sum_{i=1}^{C} f_i(1-f_i)$, which indicates how separated the classes are, where f_i is the frequency of class i in a node and C is the number of classes. Once it is obtained, the model is stored in the file system and loaded in to be used in real-time traffic classification mode online. Thus, it is also possible to validate the model with the 30% test set obtained earlier.

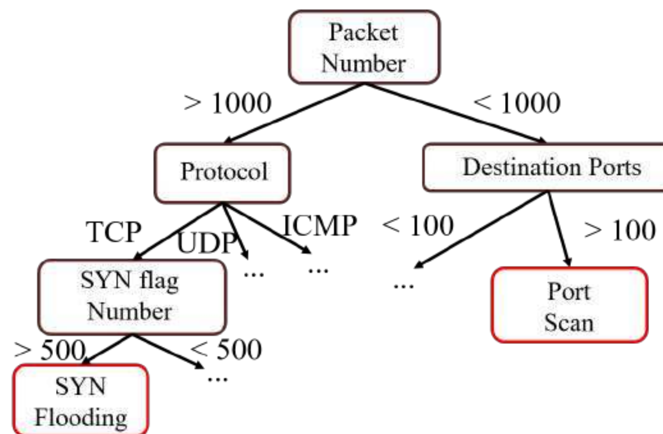


Figure 3.19: An example of Decision Tree for threat classifications. Features are split by an heuristic in order to classify traffic threats.

The Table 3.3 shows the confusion matrix of the security dataset evaluation [83]. The confusion matrix clearly specifies the rate of false positives and other metrics of each class in the test data set. The rows represent the elements that actually belong to the real class and the columns represent the elements that were classified as belonging to the class. Therefore, the prominent diagonal elements of this array represent the number of elements that are correctly sorted. In addition, Table 3.3 shows metrics complementary to the confusion matrix. By observing the values of Accuracy and Precision it is possible to see the good performance of the decision tree algorithm in off-line classification. The table verified that the algorithm presented

a high accuracy in almost all classes, with a low false positive rate. Another way to see the false positive rate is to observe the values that are outside the main diagonal. A similar result is shown in Table 3.4 where NetOp dataset was used evaluate the decision tree. This dataset has two classes, threat and normal. We can see that the false positives, the values outside the main diagonal also increase, however, the overall accuracy has increase. The results were obtained with 10-cross validation, in four virtual machines, one master and three salves, using Ubuntu 16.04 with 4GB of RAM and 2 cores.

Table 3.3: Confusion Matrix and Evaluation Metrics of Decision Tree for GTA/UFRJ Dataset.

	Normal	**DoS**	**PortScan**	**Precision**	**Overall Accuracy**
Normal	29126	1	0	99.97%	95.99%
DoS	60	5845	0	98.94%	
PortScan	8	1782	9434	84.05%	
Recall	99.76%	76.62%	100%		

Table 3.4: Confusion Matrix and Evaluation Metrics of Decision Tree for NetOp Dataset.

	Normal	**Threat**	**Precision**	**Overall Accuracy**
Normal	3713600	30140	99.19%	98.74%
Threat	22350	416100	94.90%	
Recall	99.40%	93.24%		

After obtaining the classification model from the historical base, one can evaluate the accuracy of the tool with data arriving in real time. The operation of the CATRACA tool in real time uses the stream module of the Spark platform. Thus, abstracted packets in streams, captured on different virtual machines in the cloud, are processed as they reach the Spark platform. In CATRACA we consider a flow as a stream. When a stream arrives at the detection tool, it is summarized in characteristics using the selection algorithm [80], in order to reduce processing time. Thus, the vector of selected characteristics is evaluated in the model obtained in the off-line processing. After extracting the analytical data from the flows, the results are stored in a database for further analysis. The stored data has the information collected during the detection of threats and can be reprocessed offline to calculate the parameters to be used in the real-time model. To make the system more accurate, when a new threat is detected, offline parameters are updated, obtained a feedback between online and offline detection.

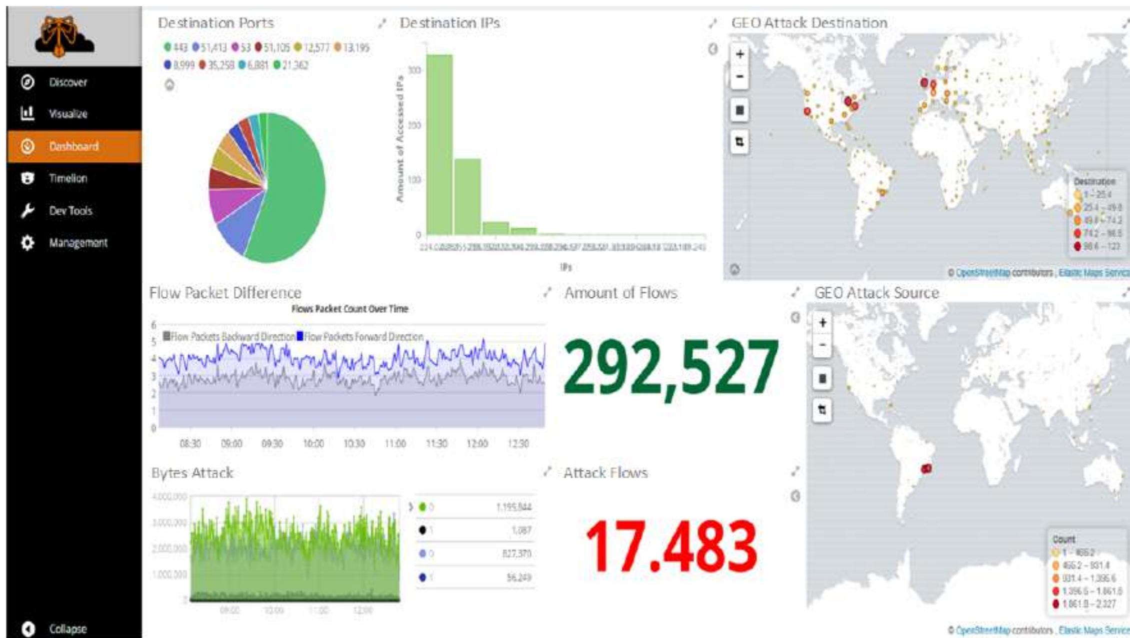


Figure 3.20: Dashboard panel view of CATRACA.

Real-Time Visualization of Enriched Data

The visualization of the enriched data occurs through a simple and friendly web interface to allow the user to monitor the different parameters of the network in real time. The open source viewer Kibana, a component of the Elastic stack, was used for the development of the web interface, as it allows the visualization of the data in a simple and fast way allied to the performance of processing of queries with large volumes of data with low latency.

Figure 3.20 shows some of the different scenarios that can be viewed in the control panel, such as the most accessed destination/source ports,the most commonly used destination IP addresses, the average size of the flows in the round-trip directions, the number of analyzed flows, among others. It is worth emphasizing the visualization of the attacks in progress through a map that portrays the origin, the destination and the number of occurrences. This is possible due to the enrichment of the data through the correlation with geolocation metadata in the processing module[15]. Thus, both data and threats are viewed in real time. In addition, all data is stored in time stamp, allowing the processing of the data through time series.

[15]for geolocation metadata we use geoip library https://pythonhosted.org/python-geoip/, Accessed April 2018.

Chapter 4

Dataset and Data Preprocessing

This chapter presents and analyzes two datasets. First, a security dataset that we have created in the laboratory GTA/UFRJ. The Second dataset is composed of more than ten days, in two different periods of real traffic for a network operator in Rio de Janeiro, Brazil. Then, we introduce the concepts of Feature Selection and Dimensionality Reduction, and, finally, we propose a new algorithm for feature selection.

4.1 Security Dataset Creation

In order to evaluate the defense mechanisms against network attacks, the first challenge is to obtain a suitable dataset for the evaluations. The availability of datasets in the literature is limited as there is concern about privacy and the fear of leakage of sensitive information contained in packet payload [84]. One of the main datasets available is the **DARPA** [85], consisting of raw TCP/IP traffic and UNIX operating system data of a simulated network obtained over seven weeks of collection totaling 9.87 GB of data. Because DARPA 98 consists of raw files, it is necessary to extract the features of these files to use them in machine learning algorithms. A greater amount of background traffic and different types of attacks were added to build the DARPA 99. The first two weeks were attack free, so it is suitable for training anomaly detection algorithms. In the next two weeks, several simulated attacks were used against the base. New attacks were introduced in DARPA 99 [86], mainly attacks on different operating systems like SunOS, Solaris, Linux, and Windows NT. Currently these operating systems are obsolete.

Most research uses a mixture of the two datasets referring to the DARPA dataset. The KDD99 dataset, in turn, was created from the files of the DARPA 98 set for an intrusion detection competition in the year 1999 [87] and consists of samples defined by 41 features and an output class. The dataset is composed of two weeks of attacks. The classes are divided into five categories that contain 24 types of training

attacks and 14 types of attacks in the training set, totaling 38 attacks. The training set consists of 494,021 flows and the test set 311,029 flows. Classes include Denial of Service (DoS), Probe, Root2Local (R2L), User2Root (U2R), and Normal Traffic. One of the problems of KDD99 is imbalance. Approximately 80% of the samples are considered attacks, which differs widely from reality. The dataset contains few types of U2R and R2L attacks and many of these attacks are duplicates. By duplicating samples, classifiers become biased to denial of service (DoS) attacks and to normal traffic, which are the most abundant in KDD99.

The NSL-KDD is a modification of the original KDD-99 set and has the same 41 features and the same five categories as the KDD 99. The improvements of the NSL-KDD over KDD 99 are the elimination of redundant and duplicate samples to avoid a biased classification and overfitting, and a better cross-class balancing to avoid random selection. Despite the reduction in size, the NSL-KDD maintains the proportions of attacks as in KDD 99. NSL-KDD contains 125,973 training samples and 22,544 test samples. However, DARPA, KDD, as well as NLS-KDD are criticized because their traffics are synthetic and therefore do not faithfully represent real computer network scenarios [88]. These datasets contain redundant data, which affect the performance of classification algorithms. Another important critic is datasets outdated, since they have been simulated more than 15 years ago [89] and many applications, as well as attacks, have since appeared.

Since the creation of KDD99, other datasets have been published and made available introducing advantages and disadvantages. In this way, there is no dataset that suits all cases and the choice of which one to use depends on the scenario and the application. Some examples of dataset often found in the literature are a simulation traffic of a real scenario, the **UNB ISCX IDS 2012** [90] dataset, a simulated botnet traffic from **CTU-13** [91] dataset, traffic of a real DDoS attack in **CAIDA DDoS**[1] dataset, real traffic of a backbone between US and Japan in **MAWI** [92], real honeypots traffic in **Kyoto** [93], and a set of 11GB of packet header traces from October 2004 through January 2005 from **LBNL**[2] edge routers.

Many enterprise unpublished their network trace due to privacy concerns. Anonymization techniques are normally adopted to preserve data privacy. Techniques such as k-anonymous [94], in which a sample is published only if it is undistinguishable from $k-1$ other samples in the data, and data perturbation [95] in which noise is added to change probability distribution of the data. Nevertheless, due to complexity of the used algorithms, those techniques are designed to work with small data, presenting poor performance with big data [96].

[1]The Cooperative Analysis for Internet Data Analysis, http://www.caida.org accessed April 2018

[2]Lawrence Berkeley National Laboratory - LBNL/ICSI Enterprise Tracing Project, http://www.icir.org/enterprise-tracing/download.html, Accessed April 2018.

A contribution of this work is the creation of two dataset with synthetic and real network traffic to evaluate network security tools. Firstly, we elaborate a dataset through the packet capture in computers from our lab, GTA at Federal University of Rio de Janeiro. Then we analyze real network traffic captured for more than ten days of a real network operator in Rio de Janeiro, Brazil.

GTA/UFRJ dataset

The traffic contains both normal traffic and real network threats. After the packets are captured, the data are gathered from packet header and grouped in a time window, generating flow data. We define a flow as a sequence of packets from the same IP source to the same IP destination.

Each flow has 24 features, generated by TCP/IP header data such as TCP, UDP and ICMP packet rate, number of source and destination ports, number of each TCP flag, among others. Table 4.1 shows the full list of features. The analysis of packet header information detects two threat classes: Denial of Service (DoS) attacks and Probe. Therefore, we elaborate the dataset with several attacks from both these classes. Altogether, the dataset contains seven types of DoS and nine types of Probe. The DoS attacks are *ICMP flood*, *land*, *nestea*, *smurf*, *SYN flood*, *teardrop*, and *UDP flood*. The different types of probe in the dataset are *TCP SYN scan*, *TCP connect scan*, *SCTP INIT scan*, *Null scan*, *FIN scan*, *Xmas scan*, *TCP ACK scan*, *TCP Window scan*, and *TCP Maimon scan*. We perform the threats using tools from the *Kali Linux* distribution, which aims to test computer system security. These attacks were labeled in the dataset by origin and destination IP filters, separating the traffic belonging the attack machines from the rest.

In Figure 4.1, we plot the correlation between the 24 features in our created dataset. This matrix calculates the Pearson correlation, also known as Pearson product-moment correlation coefficient (PPMCC). Pearson correlation is a measure of the linear dependence between two variables X and Y. Values in PPMCC vary from +1 to -1, where one is total positive linear correlation, zero is no linear correlation, and -1 is total negative linear correlation. In the figure, total correlation is represented by the dark red color and the no linear correlation is represented by dark blue color. Features 21 and 22 indicate the "Amount of ECE Flags" and the "Amount of CWR Flags" respectively. The Explicit Congestion Notification (ECN) Echo (ECE) and the Congestion Window Reduced (CWR) flags are used to warn senders of congestion in the network thereby avoiding packet drops and retransmissions. In the correlation matrix, these two features are represented in a dark blue color, indicating the lowest possible correlation. In the case of our dataset, these two variables are empty. This is due to the fact that we create our dataset in a

Table 4.1: The 24 features obtained for each flow from TCP/IP headers.

Number	**Abbreviation**	**Feature**
1	qtd_pkt_tcp	Amount of TCP Packets
2	qtd_src_port	Amount of Source Ports
3	qtd_dst_port	Amount of Destination Ports
4	qtd_fin_flag	Amount of FIN Flags
5	qtd_syn_flag	Amount of SYN Flags
6	qtd_psh_flag	Amount of PSH Flags
7	qtd_ack_flag	Amount of ACK Flags
8	qtd_urg_flag	Amount of URG Flags
9	qtd_pkt_udp	Amount of UDP Packets
10	qtd_pkt_icmp	Amount of ICMP Packets
11	qtd_pkt_ip	Amount of IP Packets
12	qtd_tos	Amount of IP Service Type
13	ttl_m	Average TTL
14	header_len_m	Average Header Size
15	packet_len_m	Average Packet Size
16	qtd_do_not_frag	Amount of "Do Not Frag" Flags
17	qtd_more_frag	Amount of "More Frag" Flags
18	fragment_offset_m	Average Fragment Offset
19	qtd_rst_flag	Amount of RST Flags
20	qtd_ece_flag	Amount of ECE Flags
21	qtd_cwr_flag	Amount of CWR Flags
22	offset_m	Average Offset
23	qtd_t_icmp	Amount of ICMP Types
24	qtd_cdg_icmp	Amount of ICMP Codes

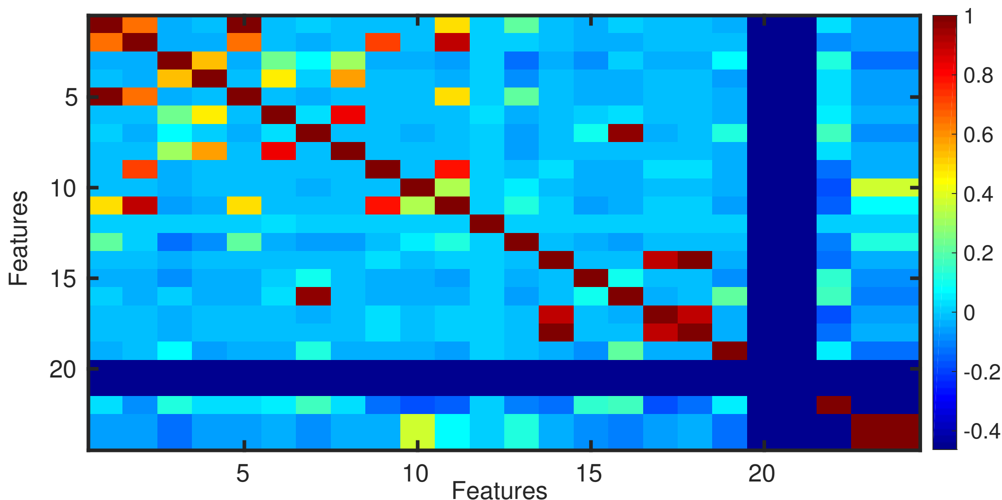


Figure 4.1: Correlation matrix of the 24 features available in the dataset. The dark red points indicate the maximum correlation and the blue points indicate the minimal correlation.

simple network that is free of congestion. On the other hand, features 23, "Amount of ICMP Types" and 24 "Amount of ICMP Codes", are represented with a dark red. This means, that the two features are highly correlated, as a consequence, only one feature is sufficient to describe the data. Because of the synthetic nature of this dataset, the ICMP type are unchanged and always coincides with the ICMP codes.

Altogether, around 95 GB of packet capture data were collected, resulting in 214,200 flows composing normal and malicious traffic[3]. To evaluate the machine learning algorithms, we perform holdout validation. In holdout validation, the data is split into two different datasets labeled as a training and a testing dataset. Holdout validation is the simplest method and shows good performance [97]. This method results in statistics that are determined with new data, not analyzed in the training phase, as usually employed in scientific works of the area. For the anomaly detection, the training is performed with 70% of legitimate flow data to determine normal behavior. The other 30% are used to determine false-positive rate and the attack data are used to calculate the attack-detection rate. Figure 4.2 show the relation of classes used in the dataset. The Normal class is around 70% of the dataset with 106.955 samples. The Denial of Service (DoS) class is 10% of the total dataset with 16.741 samples, and, finally, Probe class represents almost the 20% of the dataset with 30.491 samples.

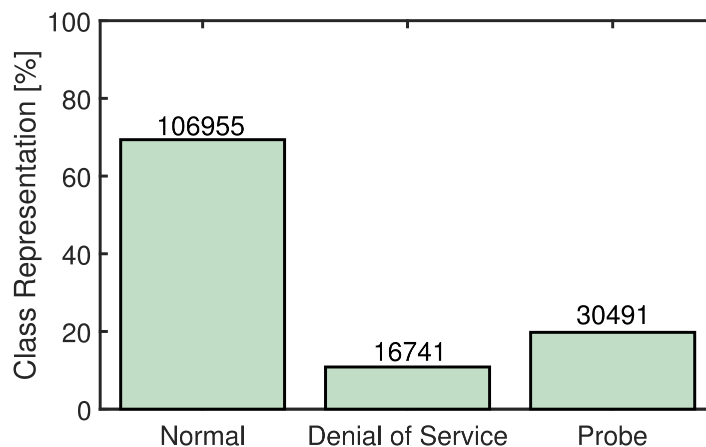


Figure 4.2: Classes Distribution in the Dataset. The main class is the Normal with almost 70% of the dataset, DoS is around 10% and Probe correspond to 20% of the Dataset.

[3]Data can be consulted through email contact with authors.

Network Operator (NetOp) Dataset

In the NetOp dataset we collected real and anonymized data from a major telecommunications operator[4]. The dataset is created by capturing 9 TB of access data of 373 residential broadband users in the city of Rio de Janeiro, Brazil. Capture is performed from 16th of February until 3rd of March of 2017. The dataset contains legitimate traffic, attacks and other security threats. An Intrusion Detection System (IDS) inspects the traffic and then summarizes a set of flow features associated with either an IDS alert or a legitimate traffic class.

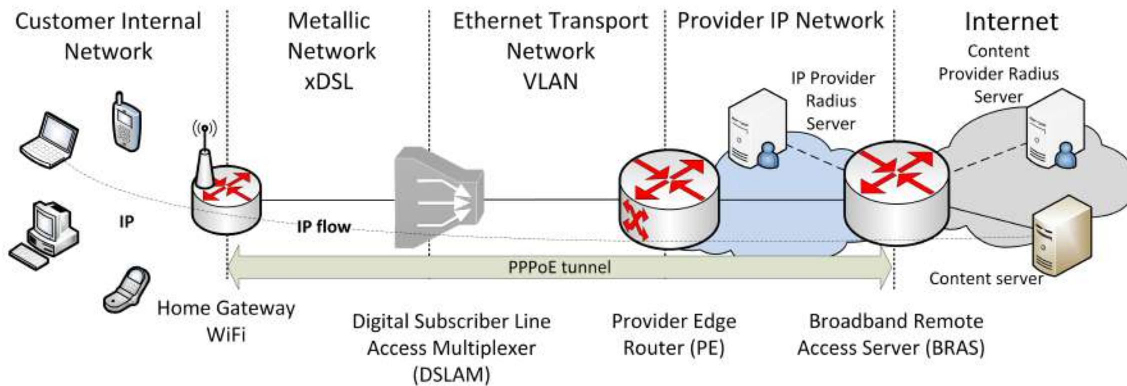


Figure 4.3: Typical topology of broadband access network. The connection between the Home Gateway and the Internet is authenticated and registered by the Radius server. The traffic is encapsulated in Point-to-Point Protocol over Ethernet (PPPoE) sessions between the user's home and the Broadband Remote Access Server (BRAS). Traffic inspection and collection occurs after BRAS.

Figure 4.3 shows a typical access topology for the broadband service composed of a Customer Premises Equipment (CPE) connected to a Digital Subscriber Line Access Multiplexer (DSLAM), a transport network, such as Multiprotocol Label Switching (MPLS) network, and a section aggregator Broadband Remote Access Server (BRAS) that authenticates the session of the users through a RADIUS server, also responsible for auditing the network usage. Thus, in an access network for fixed broadband users, the monitoring is performed only after the aggregation of the traffic, since there are no nodes that allow data monitoring in the users' premises or in the perimeter closest to the users.

The analyzed traffic is composed of the aggregated traffic coming from the high capillarity, last mile, of different users with a wide variety of service profiles accessed by each user and generating a large data volume.

The analyzed dataset was created from the capture of raw packets containing real Internet Protocol (IP) traffic information of the residential users. Traffic was collected and recorded uninterruptedly for one week through the `tcpdump`[5] software.

[4]Anonymized data can be consulted through email contact with authors.
[5]Available at http://www.tcpdump.org, Accessed April 2018.

The processes of collecting and writing file were packet unfiltered, therefore, all packets on the network were raw and recorded directly in the dataset. The physical collection structure has been configured by mirroring the aggregate traffic of one DSLAM to another port of the transport network metro Ethernet switch. The mirroring of the DSLAM port on the switch allows all traffic originated or destined to the DSLAM to be cloned to a computer running an Ubuntu Linux OS.

To ensure high-speed storage and to allow easy data transport, the dataset was written to an external hard drive with a USB 3.0 interface. Figure 4.4 shows the basic topology and the assembled structure for data collection. It is worth to mention that analyzing all traffic from operator is out of scope, thus data consumption samples satisfy the needs for the proposed characterization.

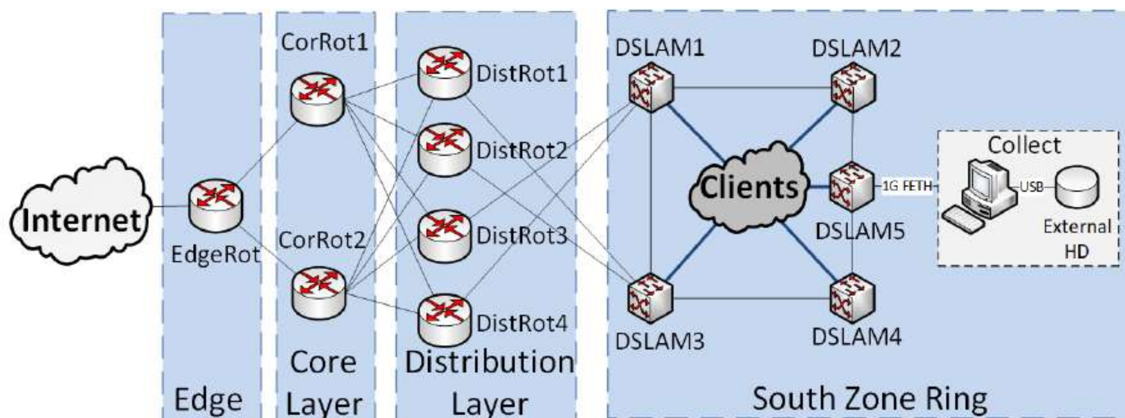


Figure 4.4: Topology of the data collection structure of the main DSLAM port with 373 broadband clients.

The data capture procedure ensures no loss in port mirroring at 1 Gb/s. Thus, 100% of the traffic generated by the 373 customers was collected and recorded in the dataset, totaling 5 TB of information. Although the average available speed at each port of the DSLAM is approximately 12 Mb/s, generating a hypothetical aggregate traffic of more than 4 Gb/s, it was verified that during the entire capture process, aggregate real traffic did not exceed 800 Mb/s. Aggregate traffic is composed by round-trip, uplink and downlink traffic. It is worth noting that all the captured traffic comes from fixed broadband sessions.

The Data Analysis

The analysis of captured data from the telecommunications operator's network was divided into three stages. The first stage handles the raw data capture files through a network intrusion detection system (IDS) and then generates a summary of the data in the form of flows. We use the flow definition based on RFC 7011 [98].

A flow is defined as the set of packets collected during a window time in a monitoring point sharing common features. These features include information and statistics of the packets and packets header. We abstract the flow in 44 features as shown Table 4.4.

The first stage is shown in Figure 4.5. Data analysis was based on the features extraction of flows represented by the captured packets, as well as the verification of possible alerts through an IDS. Since the packets come from residential clients with Asymmetric Digital Subscriber Line (ADSL) access, the captured traffic is encapsulated in Point-to-Point Protocol over Ethernet (PPPoE) sessions which make the analysis of packets impossible for some IDS that do not perform the inspection of this type of protocol, such as Snort [99]. Therefore, to perform traffic classification on different types of alerts, the Suricata IDS[6], Version 3.2, was used with its most recent signature database.

The classification between normal traffic and alert was performed based on Suricata signatures since there was no previous knowledge about threat information. Because the data is real and hence untagged, it is impossible to ensure that all flows are legitimate or, even after IDS classification, that all alert flows are malicious.

Parallel to packet classification by the IDS, the captured packets were decapsulated from the PPPoE session using the tool `stripe`[7] and were summarized in flows through the `flowtbag`[8] tool. In addition, a Python application was developed to process the output of the Suricata IDS, and the summarized flow features in order to correlate which flow was reported as an alert by the IDS.

Thus, it was possible to obtain a flow dataset with the respective class labels. As we remove payload packets and source and destination IP addresses features from the dataset to ensure the data anonymization, the dataset presents 43 features of each flow plus the class to which each flow belongs. The output class, feature 44, is given by the type of alert generated by the IDS or 0 in the case of a normal flow.

Figure 4.6 shows the number of threats and normal flow in each day of the dataset in 2017. We can see that almost all days contains around 30% of alerts. Only day 17/2 contains less number of alerts. The maximum alerts number was during the Saturday 25/2 with 1.8 Million alerts.

Figure 4.7 shows the source and destination ports of the flows. The figure focuses on the 1024 first ports (from 0 to 1023), as they are the operating-system restricted ports. Usually, these ports are used by daemons that execute services with system administrator privileges. Our flow definition assumes that the source port initiates the TCP connection. Because the dataset portrays home users, it is expected that

[6]Available at https://suricata-ids.org, Accessed April 2018.
[7]Available at https://github.com/theclam/stripe, Accessed April 2018.
[8]Available at https://github.com/DanielArndt/flowtbag, Accessed April 2018.

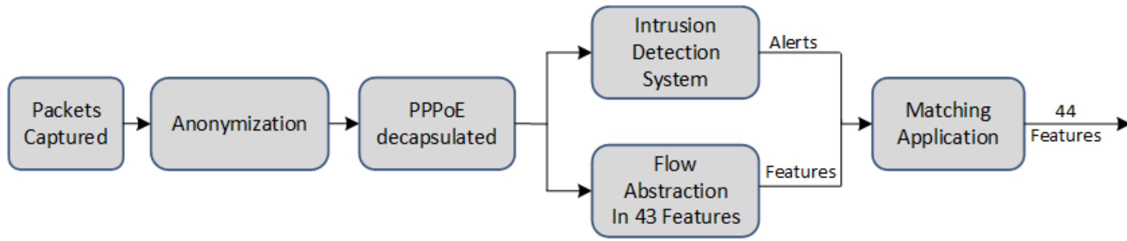


Figure 4.5: Network Operator (NetOp) dataset processing steps. Packets are first anonymized, then PPPoE encapsulation is removed. An Intrusion Detection System (IDS) is used to classify alerts, in parallel, packets are abstracted in 43 flow features. Finally, an application is used to match traffic flows with IDS alerts, generating a flow with 44 features corresponding to 1 if alert and 0 to normal traffic.

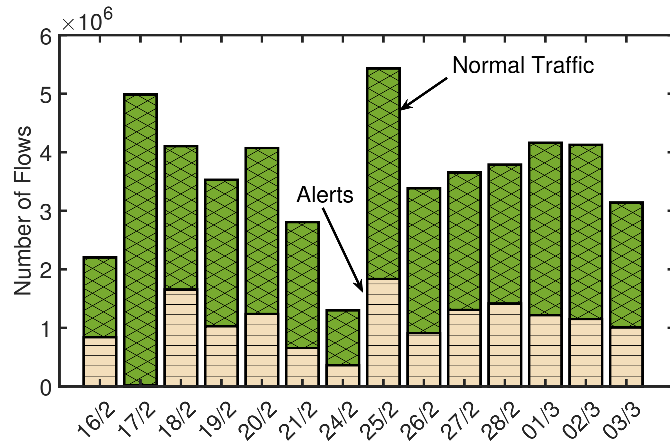
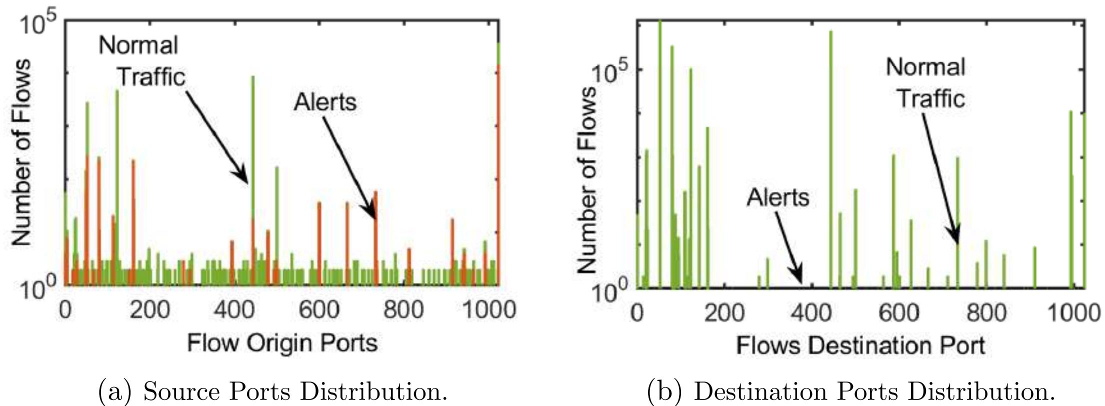


Figure 4.6: Number of Alerts and Normal Traffic flows in Network Operator dataset.



(a) Source Ports Distribution.

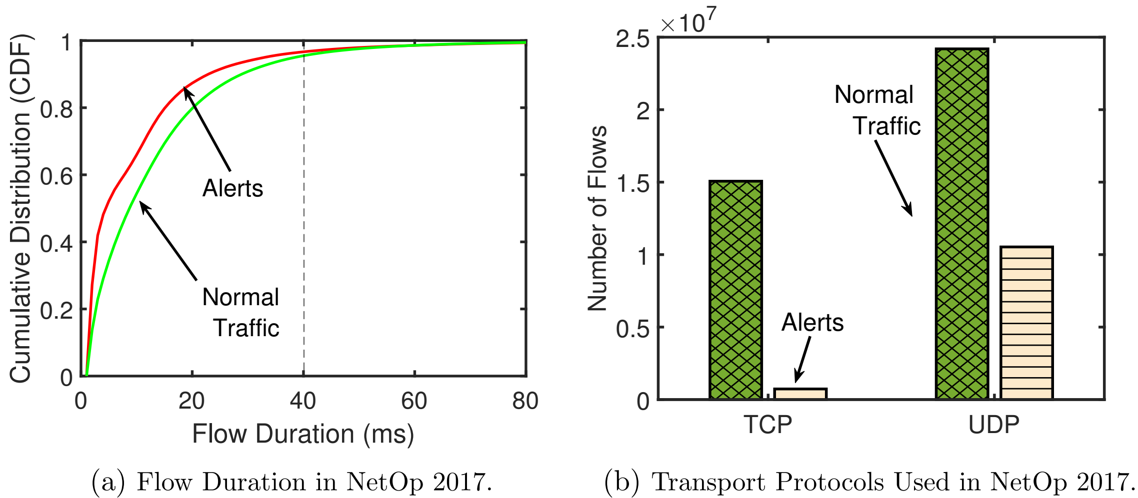
(b) Destination Ports Distribution.

Figure 4.7: Ports used in flows. Comparison of the use of the lowest 1024 ports (restricted ports) in the evaluated flows. Because they are home users, the largest number of flows originating from these ports are flows that generate alerts.

most connections will be destined to restricted and dynamic ports. Thus, it is remarked that the number of alerts coming from connections that the destination port is in the range of restricted ports is relatively low to the total number of connections on these ports, Figure 4.7b. When considering the flows, in which the source port is in the range of restricted ports, almost all flows are labeled as alert by

the IDS, shown in Figure 4.7a. Another important fact is that most of the analyzed flows reflect the use of the DNS service (UDP 53) and HTTPS and HTTP services (TCP 443 and 80). The prevalence of HTTPS services over HTTP reflects the shift that major Internet content providers, such as Google and Facebook, have done to use encrypted service by default to ensure users' privacy and security.

The relation between the most accessed services and flow duration is shown in Figures 4.8a and 4.8b. The duration of analyzed flows is mostly less than 40 ms, characterizing the use of DNS, HTTP and HTTPS services. Regarding the protocols used, the prevalence of UDP flows is evident and refers to DNS queries. It is worth mentioning that the number of alerts generated by UDP flows is more than 10 times greater than the number of alerts generated by TCP flows. Another important point is that the number of flows that generate alerts is approximately 26% of total flows.



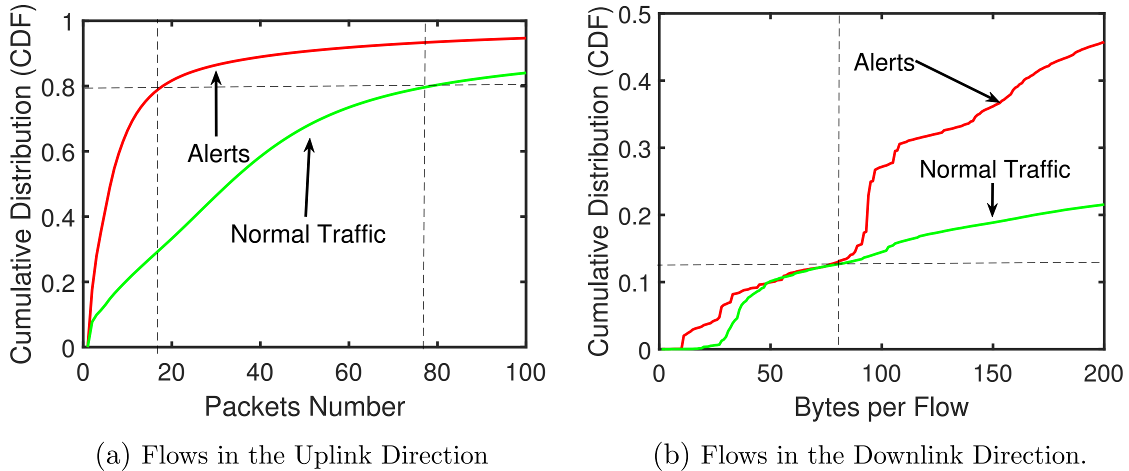
(a) Flow Duration in NetOp 2017.

(b) Transport Protocols Used in NetOp 2017.

Figure 4.8: Cumulative Probability Density Function (CDF) for the distribution of the duration of flows in milliseconds and number of flows per transport protocols. A) The flows that generate alerts are shorter in duration than the average flow. B) The legitimate flows with UDP are numerous due to DNS (port 53 UDP). The number of alerts in UDP is more than 10 times greater than in TCP flows.

Figure 4.9 shows the characterization of the number of packets per flow in uplink and bytes per packet in downlink direction. In uplink direction, Figure 4.9a, 80% of alerts starts with 20 packets or less while normal traffic starts with almost 80 packets. This behavior is typical from probe or scans attacks that send small amounts of packet to discover target vulnerabilities. In Figure 4.9b alerts and normal traffic show a similar pattern of 11% of flows, however, alerts use more than 100 Bytes in more than 30% of flows.

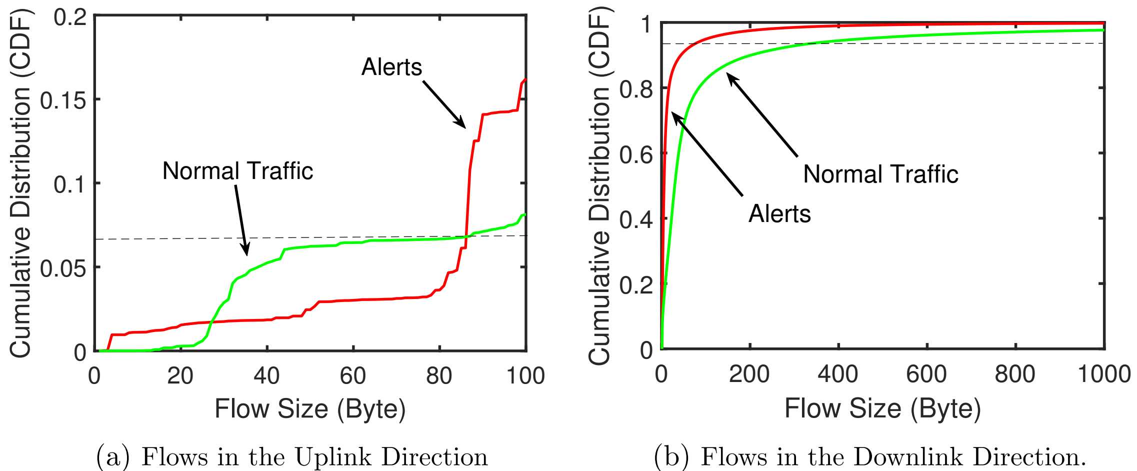
Considering the amount of data transferred in each flow, Figure 4.10 compares the round-trip flows in relation to volume in bytes. The disparity of the traffic volume in both directions of the communication is evident. While in one way 95% of traffic has a maximum volume of 100 B, in the other way, the same traffic share



(a) Flows in the Uplink Direction (b) Flows in the Downlink Direction.

Figure 4.9: Cumulative Probability Density (CDF) function for the distribution of the number of packets per flow. Flows that generate alerts tend to have fewer packets.

presents up more than 500 B. This result demonstrates that the residential broadband user profile is a content consumer. Another interesting point is that the flows that generate alerts have a similar traffic volume profile in both directions. Asymmetric traffic is more typical of the legitimate users.

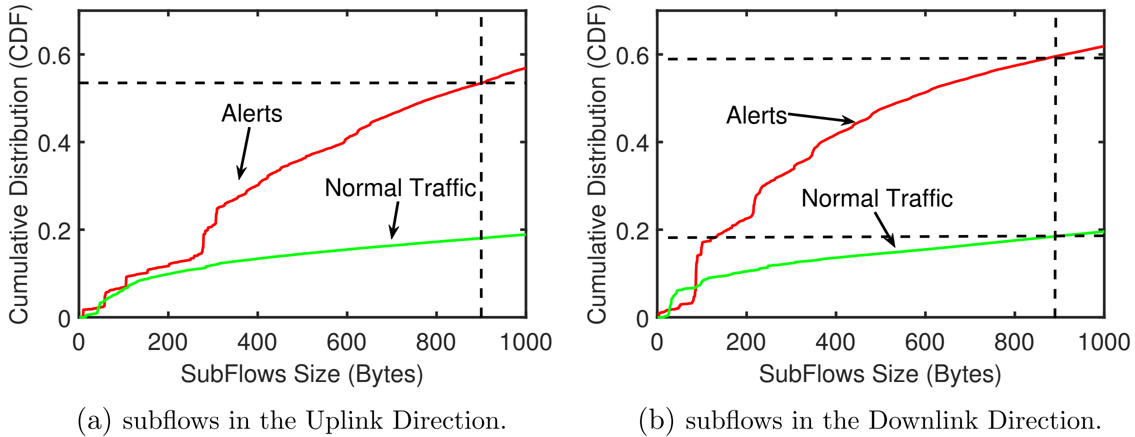


(a) Flows in the Uplink Direction (b) Flows in the Downlink Direction.

Figure 4.10: Cumulative Probability Density (CDF) function for volume distribution in bytes by flow. Flows that generated alerts tend to have smaller volumes in transferred bytes.

Figure 4.11 shows the behavior of the subflows generated in each connection. A subflow is considered a flow in one direction. Both Figures 4.11a, 4.11b, subflows size in uplink and downlink, shows a very similar behavior. More than 20% of Normal traffic flows, reach 900 B, but this value is reached in almost 60% of the flows. Values of subflows are almost ten times bigger than values represented in Figures 4.10 where Bytes flows size are shown. This is because the flows are mostly of short duration,

evidenced in Figure 4.8a, and thus do not generate subflows. Data analysis showed that the flows do not pass to the idle state, when the flow is inactive.

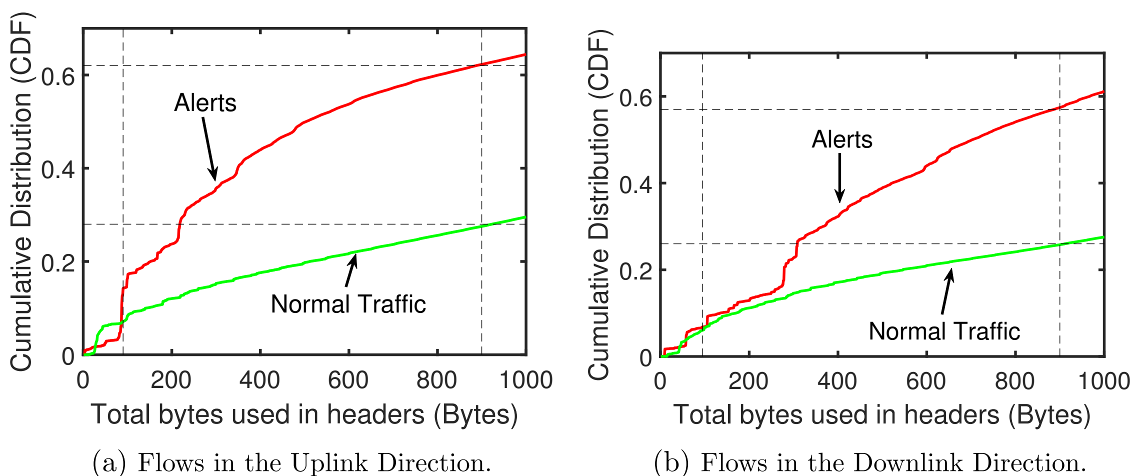


(a) subflows in the Uplink Direction.

(b) subflows in the Downlink Direction.

Figure 4.11: Cumulative Probability Density (CDF) function for volume distribution in bytes by subflow in each flow. Flows that generate alerts tend to have smaller volumes in bytes that are transferred in subflows.

Another important feature is the total amount of data in the packet headers. Figure 4.12 shows that in both flows directions, alert and total traffic have the same behavior. In particular, there is symmetry in the round-trip traffic in terms of the volume of data in the headers. It highlights that malicious traffic does not rely on the usage of header options. Also, in both senses, uplink and downlink show a similar behavior. Until 90 Bytes per header alerts and normal traffic are similar, however, with 900 Bytes are represented by almost 30% of normal traffic and close to 60% of alerts flows.



(a) Flows in the Uplink Direction.

(b) Flows in the Downlink Direction.

Figure 4.12: Cumulative Probability Density Function (CDF) for volume distribution in bytes of the data in packet headers. The behavior of traffic that generates alerts is very similar to total traffic.

Figure 4.13 shows which are the main classes of alerts triggered by the IDS.

Alerts for attacks against HTTP are the most frequent. This class of alerts includes SQL injection attacks through HTTP calls and XSS attacks (cross-site scripting). Home users can execute these attacks, as they use the parameters of HTTP calls to insert some malicious code into the servers and, therefore, are not filtered by access rules. Other important alerts are port scanning and execution of malicious applications (trojan and malware). The scans are generally intended to identify open ports and vulnerabilities in user premises such as the home gateway. Alerts for trojan and malware identify activities typical of known malicious applications that aim to create and exploit vulnerabilities in the devices of the home users. Other alerts refer to information theft and to Byzantine-attack signatures on common protocols, such as IMAP and Telnet[9].

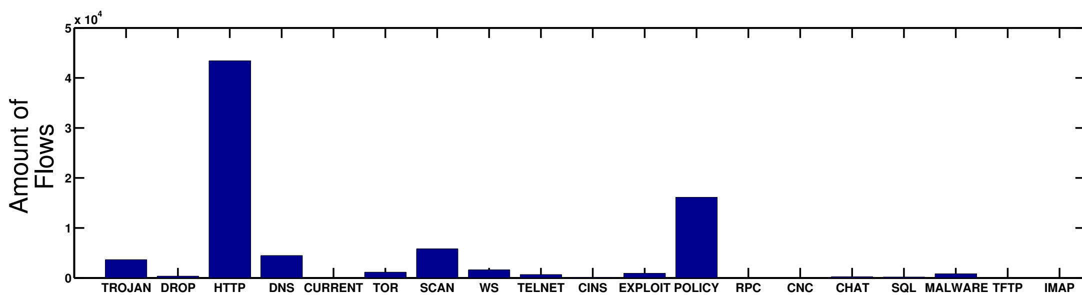


Figure 4.13: Distribution of the main types of alerts in the analyzed traffic.

4.2 Data Preprocessing

Data preprocessing is the most time-consuming task in machine learning [100]. As shown in Figure 4.14 Data preprocessing is composed by four main steps [101]. The first step, Data Consolidation, data is collected from several sources and data are interpreted for better understanding. The second step, Data Cleaning, all samples are analyzed and it is verified if there are values that are empty or missing and is an anomaly in the dataset, also this step check if there are some inconsistencies. In the third step, Data Transformation, different functions are applied to data to improve the machine learning process. Data Normalization, conversion of variables from categorical into numerical values are done in this step. In the last step, Data Reduction, techniques such as feature selection are applied to reduce data in order to improve and fast machine learning process. When the entire process is finished, data is ready for input in any Machine Learning algorithm. In this work, we focus in the last two steps Data Transformation and Data Reduction which are the most time-consuming steps.

[9]Mainly used for remote configuration of network equipment.

Furthermore, all Feature Selection algorithms assume that data arrive preprocessed. Normalization, also known as feature scaling, is an important method for proper use of classification algorithms, because normalization bounds the domain of each feature to a known interval. If the dataset features have different scales, they may impact in different ways on the performance of the classification algorithm. Ensuring normalized feature values, usually in $[-1, 1]$; implicitly weights all features equally in their representation. Classifier algorithms that calculate distance between two points, e.g., KNN and K-Means, suffer from the weighted feature effect [102]. If one of the feature has a bigger range of values, the distance calculation will be highly influenced by this feature. Therefore, the range of all features should be normalized, and each feature contributes approximately proportionally to the final distance. In addition, many preprocessing algorithms consider that data are statically available before the beginning of the learning process [103].

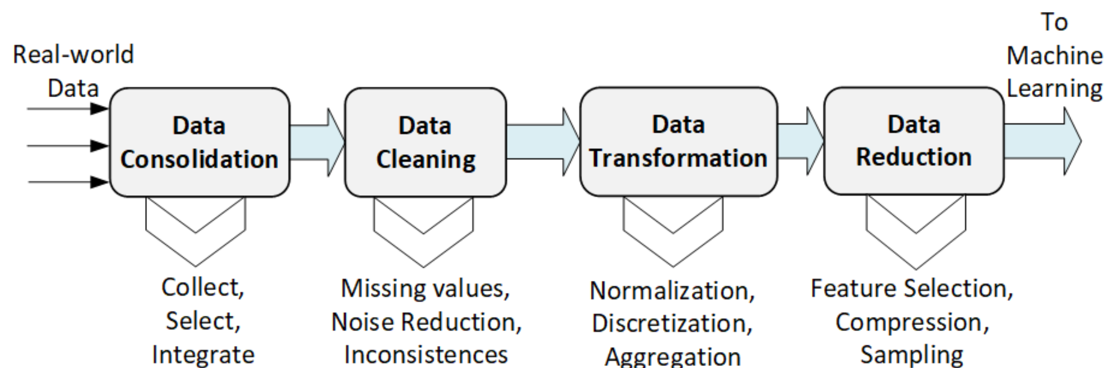


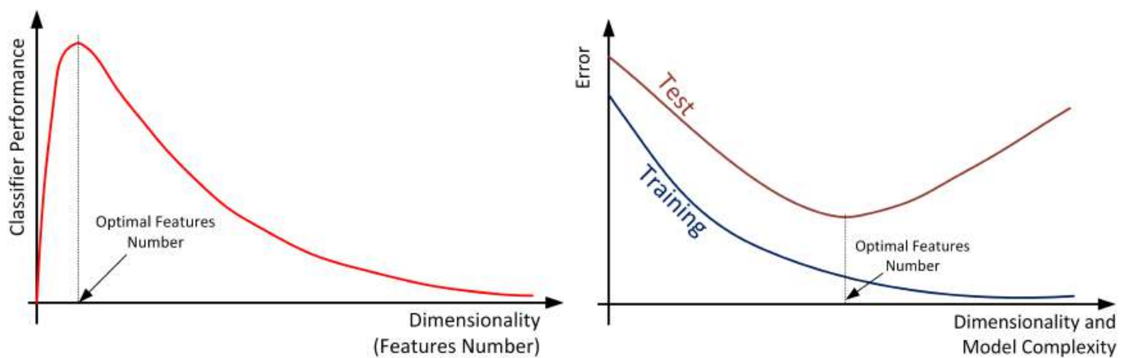
Figure 4.14: Preprocessing steps composed of Data Consolidation, Data Cleaning, Data Transformation and Data Reduction. Data Transformation and Data Reduction are the most time-consuming steps.

4.2.1 Feature Selection and Dimensionality Reduction

An information can be represented or classified by its features or attributes. The number of features or attributes used to represent information varies greatly. A relevant problem is that increasing the number of attributes does not always improve the accuracy of the information classification. This problem is known as the "curse of dimensionality" [104] which states that there is an optimal number of features that can be selected in relation to the size of the sample to maximize the performance of the classifier.

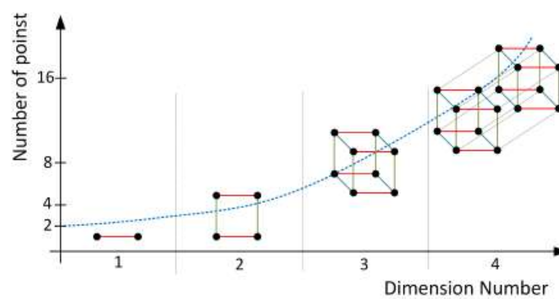
Figure 4.15a shows that when dimensionality increases, the performance of the classifier increases until the optimal number of features is reached. From this optimal value, the performance falls. Therefore, from the optimal value of feature number, increasing dimensionality without increasing the number of training samples results

in a decrease in classifier performance. Figure 4.15b shows the performance degradation of the algorithms. The increase in dimensionality is directly proportional to the complexity of the model, obtaining a low error rate during the training. However, these models present the overfitting problem during the training step, in which the model perform poorly in the test step. This mean that the model is specialized for the training data and misclassifying samples in the test set. Figure 4.15c shows that the number of dots increases exponentially with increasing dimensionality. In spaces with many dimensions, the points become sparse and not very similar, with points very distant from each other and approximately equidistant, which leads to a classifier prone to errors. In addition, other problems arise when using a high number of dimensions with machine learning algorithms. Some traditional distance metrics, such as Euclidean distance, are no longer meaningful in high dimensions, requiring the use of other types of metrics, such as the cosine distance, which has a higher computational cost. Therefore, it is common to reduce the number of features or dimensionality prior to the application of a machine learning algorithm.



(a) Performance of the classifier with increasing dimensionality.

(b) Overfitting effect on training data, which reduces test performance.



(c) Exponential growth of the number of points with an increase in dimensionality.

Figure 4.15: The "curse of dimensionality" problem. This problem asserts that there are an optimum number of features that can be selected in relation to the size of the sample to maximize the performance of the classifier.

Feature selection or dimensionality reduction techniques are used to improve the learning performance. Feature selection maintains the most relevant features of

the original dataset, creating a smaller subset of the features. On the other hand, the dimensionality reduction takes advantage of the redundancy of the input data, calculating a smaller set of new dimensions or synthetic features. The new synthetic features are a linear or non-linear combination of the input features. The main idea of the methods is to remove all redundant information, keeping only the necessary information, which is sufficient to represent the original information.

The purpose of feature selection and dimensionality reduction is to produce a minimum set of features so that maintains the most similar performance of the generating set. Therefore, feature selection and dimensionality reduction improve classification performance by providing faster and therefore economically viable classifiers. The feature selection has an additional property, because it allows a better understanding of the process that generates the data. In the dimensionality reduction the selected features are synthetic and composed of a combination of the original features, which makes it difficult to understand the process.

The dimensionality reduction can also be seen as the process of deriving a set of degrees of freedom, which are used to reproduce most of the variability of a dataset [105]. Ideally, the reduced representation must have a dimensionality that corresponds to the intrinsic dimensionality of the data. The intrinsic dimensionality of the data is the minimum number of dimensions required to meet the observed properties of the data. Generally, in dimensionality reduction a new feature space is created through some kind of transformation of the original feature space.

Thus, in the reduction of dimensionality, given the random variable of n-dimensions $\mathrm{x} = (x_1, x_2, \ldots, x_n)$, it is possible to find a lower dimensional representation of it, $s = (s_1, s_2, \ldots, s_k)$ with $k \leq n$. Many algorithms with different approaches have been developed to reduce dimensionality that can be classified into two groups: linear and non-linear. The linear reduction of dimensionality is a linear projection, in which n-dimensional data are reduced in k-dimensional data using k linear combinations of n variables. Two important examples of linear dimension reduction algorithms are Principal Component Analysis (PCA) and Independent Component Analysis (ICA). The objective of the PCA is to find an orthogonal linear transformation that maximizes the feature variance. The first PCA base vector, called the main direction, describes better the variability of the data. The second vector is the second-best description and must be orthogonal to the first and so on in order of importance. On the other hand, the goal of ICA is to find a linear transformation in which the base vectors are statistically independent and not Gaussian, that is, the mutual information between two variables of the new vector space is equal to zero. In signal processing, ICA is used to separate two mixed signals. A common example is the cocktail party [106], in which the objective is to listen one-person speech in a noisy room. Unlike PCA, the base vectors in ICA are neither

orthogonal nor ranked in order, all vectors are equally important. PCA is normally used when we want to find a reduced representation of the data. On the other hand, the ICA is normally used to obtain features extraction, identifying and selecting the features that best suit the application. Figure 4.16 shows the eigenvalues associated to the synthetic dataset (GTA/UFRJ). The first fourth components calculated by the PCA linear transformation represent 80% of the total variance. Therefore, these four components are selected and the others, that represent less than 20% of the total data variance, are discarded, improving the processing time, which is critical in real-time applications.

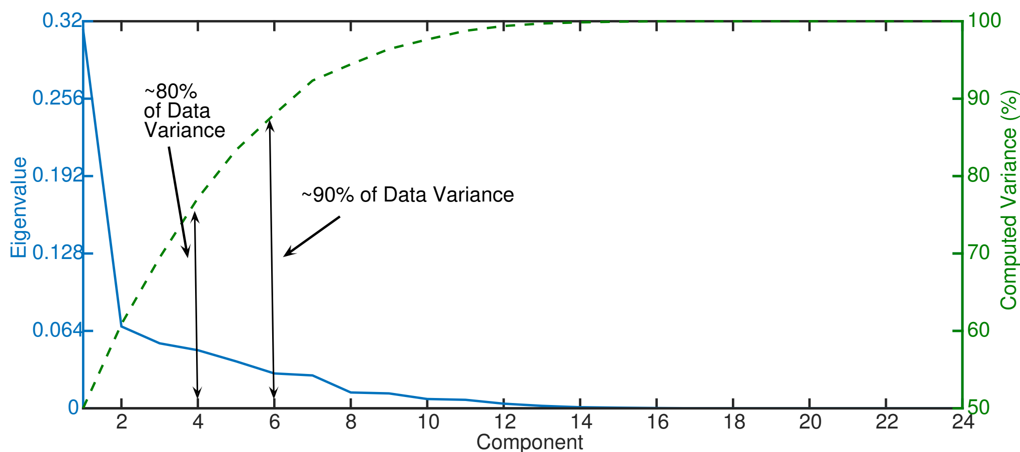
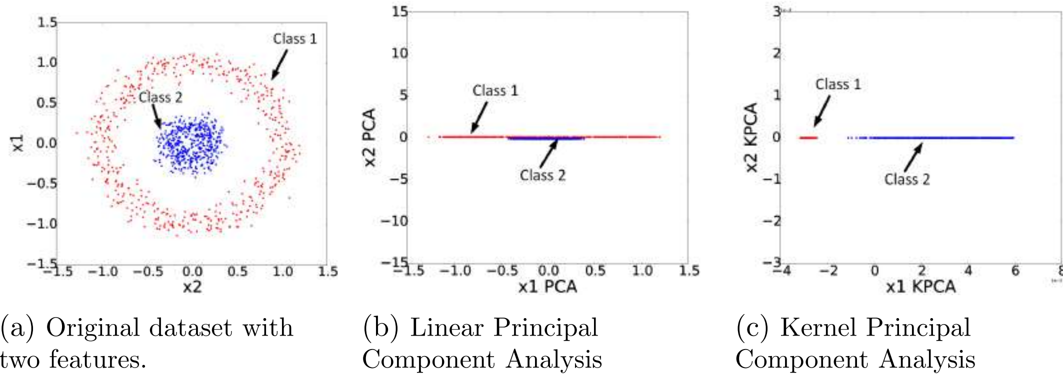


Figure 4.16: Eigenvalue for each of the 24 flow features in GTA/UFRJ dataset. The eigenvalue associated to each of the transformed features is proportional to the data variance. 80% and 90% of the total data variance is represented between the fourth and the sixth highest principal components.

In cases where high-dimensional data have a non-linear nature, linear algorithms do not perform well. This means that the relationship between classes is not described in a linear subspace, as shown in Figure 4.17a. For these cases, it is possible to use the PCA technique with kernels functions. A Kernel function transforms the input vectors of the original space into a larger dimensional space, in which the problem becomes linearly separable [107]. Figure 4.17 shows a comparison of linear and non-linear reduction methods. The original dataset, shown in Figure 4.17a, is a case of two concentric circles, each circle is a class. The goal is to reduce a 2-dimensions ($\mathbb{R}^2$) space into a 1-dimension space ($\mathbb{R}^1$). After applying a linear reduction, Figure 4.17b, the Principal Components cannot achieve a subspace where the classes are linearly separated into space ($\mathbb{R}^1$). This is because the two concentric circles are two separable nonlinear classes. After applying a non-linear method, such as a Gaussian Kernel PCA, shown in Figure 4.17c, the method gets a subspace where the classes are separated properly.

There are two approaches to class separation in data that cannot be separated



(a) Original dataset with two features.

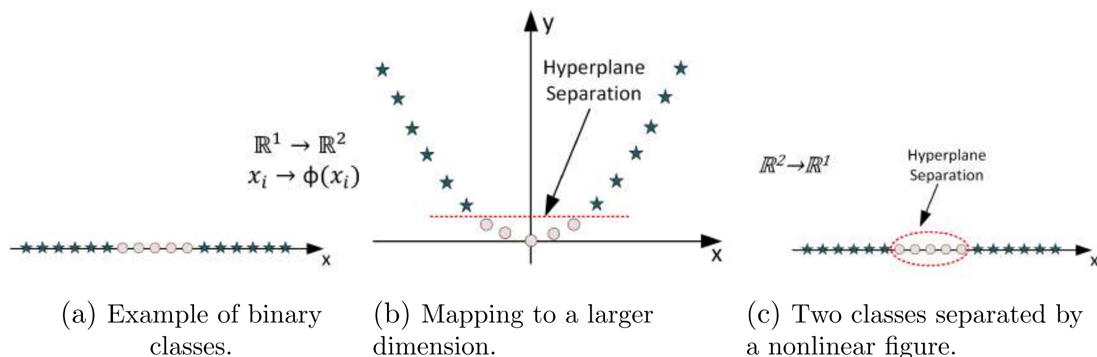
(b) Linear Principal Component Analysis

(c) Kernel Principal Component Analysis

Figure 4.17: Example of non-linear class separation. a) two classes in a concentric circle manifold; b) The linear PCA is not able to separate properly the two classes; c) A better result is achieved when the Kernel PCA is used.

linearly, as shown in Figure 4.17a. The first mapping the data into a larger space, in which the classes can be separated linearly by straight lines as shown in Figure 4.18b. The example of Figure 4.18 is a binary example, in the plane of $\mathbb{R}^1$ there is no linear hyperplane that is able to separate the two classes.

If the problem is taken to a plane with a higher dimensionality, $\mathbb{R}^1 \rightarrow \mathbb{R}^2$, it is possible to find a hyperplane, dash line in Figure 4.18b that separates classes. The dataset can be mapped into a space of a larger dimension, $\mathbb{R}^1 \rightarrow \mathbb{R}^2$, and thereby enable a linear hyperplane to separate classes, as shown in Figure 4.18b.



(a) Example of binary classes.

(b) Mapping to a larger dimension.

(c) Two classes separated by a nonlinear figure.

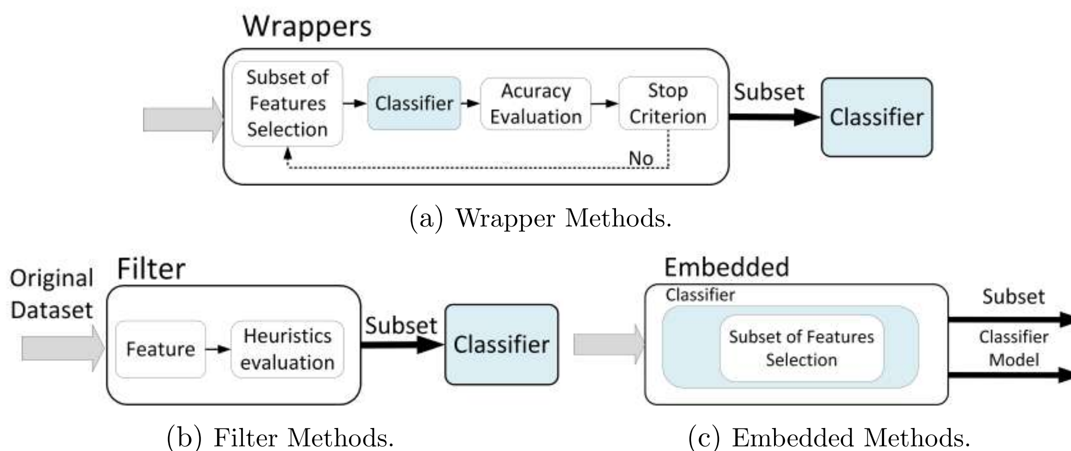
Figure 4.18: Strategies for separating non-linear data distribution classes: a) original dataset of two classes with non-linear distribution; b) data mapping with increase of size that allows the separation with linear algorithms, such as PCA; c) use of nonlinear hyperplane, as in the PCA nonlinear kernel algorithm.

This hyperplane, when brought down to a smaller dimension, corresponds to a nonlinear figure as shown in Figure 4.18c $\mathbb{R}^2 \rightarrow \mathbb{R}^1$. However, when mapping to larger dimensions, the "curse of dimensionality" explained above is incurred, which generates a high computational cost. To solve this problem the "kernel trick" is applied. A kernel function is a similarity function that corresponds to the dot product in an expanded vector space. The idea is to find a nonlinear function

in which dimension mapping is not necessary and computation is independent of the number of features. If there exists a nonlinear transformation $\Phi(x)$ from the original D-dimensional feature space to an M-dimensional feature space, where $M \gg D$. Then each data point x_i is projected to a point $\Phi(x_i)$. If the $K()$ matrix containing the scalar products among all pairs of data points is now calculated as $K(x_i, x_j) = \Phi(x_i)^T \Phi(x_j)$.Then each data point x_i is projected to a point $\Phi(x_i)$. The most commonly used kernels are the polynomial, Gaussian and tangent nucleus (hyperbolic tangent). If a used kernel is linear, we can see the standard PCA as a case of the Kernel PCA.

Feature Selection

The feature selection produces a subset of the original features, which are the best representatives of the data. As opposed to dimensionality reduction there is no loss of information. Feature selection techniques can be divided into three types of algorithms [108]: wrappers, filter and embedded.



(a) Wrapper Methods.

(b) Filter Methods.

(c) Embedded Methods.

Figure 4.19: Feature Selection Methods. a) Wrappers methods use a classifier to evaluate the subset of features. b) Filter methods use heuristics to evaluate a feature or a subset. c) The embedded methods use a specific classification algorithm to make the selection naturally.

The wrapper algorithms, as shown in Figure 4.19a, use machine learning techniques such as Support Vector Machine (SVM), decision tree, among others, to measure the quality of features subsets without incorporating knowledge about the specific structure of the classification function. Thus, the method will evaluate each subset within the internal classifier. The method will select the subset with the highest accuracy of the classifier. In high dimensional dataset this search is a NP-hard problem. Wrapper methods tend to be more accurate than the filter methods, at a higher computational cost [109]. One popular Wrapper algorithm is the Sequential Forward Selection (SFS). Starting with an empty set S and the full set of all features

X, the SFS algorithm makes a bottom-up search and gradually adds features selected by an evaluation function to S, that minimizes the mean square error (MSE). At each iteration, the feature to be included in S is selected among the remaining available features of X. One problem of this method is that SFS is prone to select redundant features. Once a feature is selected, it cannot be evaluated again. Thus, the redundant selected feature could not be the best.

Embedded methods are a sub-class of wrapper methods. In this case, the subset of features is obtained as well as the model where they were selected. Embedded methods perform the feature selection process during the learning phase of a classifier. These methods select features based on criteria generated during the learning process of a specific classifier. In contrast to wrapper methods, embedded methods do not separate learning from feature selection. As in wrapper methods, embedded methods use the accuracy of a classifier to evaluate the relevance of the feature. However, embedded methods need to modify the classification algorithm in the learning process. The SVM-RFE feature selection method was proposed in the elimination of recursive features in the selection of genes for cancer classification [110]. The algorithm selects the features according to a classification problem based on the training of a linear Support Vector Machine (SVM). The features with the lowest ranking are removed according to a w criterion, sequentially backward. The criterion w is the value of the hyperplane decision in the SVM.

To reduce the high computational cost of evaluating the various subsets of classifier-based features, the filter methods were proposed. Filter methods are called open-loop methods because they do not interact with the classifier. Heuristics are used instead of classifiers to evaluate the relevance of the feature [111]. As its name implies, features that satisfy the heuristic criterion are filtered. Gaining information, distance, consistency and similarity between features as well as statistical measures are some of the most used heuristics in filter evaluation. Since the selection process is done in a step prior to classification, only after the best features are found, classification algorithms can use them. While this method is fast to select features, having no interaction with the classifier subset of feature may imply lower accuracy. One of the most popular filter methods is Relief, in which the feature score is calculated as the difference between the distance of the closest sample from the same class and the closest sample of the different class. The main disadvantage of this method is that data classes must be labeled in advance. Relief is limited to problems with only two classes, but ReliefF [112] is an enhancement to the Relief method that handles multiple classes using the nearest k-neighbors technique. ReliefF is a supervised method in which classes labeling must be known prior to the application of the method. In applications such as network monitoring and threat detection, network flows reach unclassified classifiers. Therefore, unsupervised algorithms must be applied.

4.3 The Proposed Preprocessing Method

Our preprocessing method comprises two algorithms. First, a normalization algorithm enforces data to a normal distribution which values are -1 and 1 interval. The proposal is parametric-less. Then, we propose a feature selection algorithm based in correlation between pairwise features. The proposed algorithm is inspired by the Correlation Features Selection (CFS) [113]. CFS scores the features through the correlation between the feature and the target class. The CFS algorithm calculates the correlation between pairwaise features and the target class to get the importance of each feature. Thus, the CFS depends on target class information *a priori*, so it is a supervised algorithm. The proposed algorithm performs an unsupervised feature selection. The correlation and variance between the features measures the amount of information that each feature represents in relation to the others. Thus, the presented algorithm demands less computational time independently of class labeling *a priori*.

4.3.1 The proposed Normalization Algorithm

In our normalization algorithm 1, a histogram of a feature f_i is represented as a vector $b_1, b_2, ..., b_n$, such that b_k represents the number of samples that falls in the bin k. In practice, it is not possible to know in advance the min and max for any feature. As a consequence, we use a sliding window approach, where the dataset X are the s last seen samples. For every sliding window we obtain the min and max values of each feature. Then, data values are grouped in a set b of intervals called bins. The idea is to divide the feature f_i in a histogram composed by bins $b_1, b_2, ..., b_m$, where $m = \sqrt{n} + 1$, being n the number of features, as shown in line 3 in Algorithm 1.

Each bin consists of thresholds k, for example the feature f_i is grouped in $b_1 = [min_i, k_1)$, $b_2 = [k_1, k_2)$, ... , $b_m = [k_m - 1, max_i]$. The step between threshold k is called *pivot* and it is determined as $(max_i - min_i)/m$, as it is show in Algorithm 2. If the min or max values of the previous sliding window are smaller or bigger than min or max of the current window, that is, $min_{i-1} < min_i$ or $max_{i-1} > max_i$, new bins are created until the new values of min or max are reached. With the creation of new bins, the proposal is able to detect changes in the concept drift but it cannot follow the change in the distribution.

The frequency of each bin is obtained by rate between the number of observed samples in a bin and the total number of samples that were added to the histogram. Comparing the sample x_i against the thresholds k of the bins, line 4 Algorithm 3, we define in which bin we have to increment the number of observed samples. If the value of the sample x_i is in-between the thresholds of the bin_j, then the hit number

Algorithm 1: Stream Normalization Algorithm

Input : X: Sliding window of Features, w: Window Number
Output: **H**: Normalized Features, f_r: relative frequency

1 **if** $w == 1$ **then**
2 **for** *feature f in X* **do**
3   b_n=$\sqrt{n}+1$;`/* n: number of features */`
4   H=`CreateHistogram`(X,b_n);
5 **end**
6 **else if** $w > 1$ **then**
7   **for** *sample s in f* **do**
8    $[H, f_r]$=`UpdateHistogram`(X,b);
9   **end**

Algorithm 2: `CreateHistogram()` Function

Input : X: Sliding window of Features, b_n: number of bins
Output: **H**: Histogram

1 $[max, min]$=CalculateMaxMin;
2 k=$(max - min)/(b_n)$; `/* k: threshold */`
3 **for** *bin b in* b_n **do**
4 b=$[min_i, k)$;
5 k+=min;
6 **end**

Algorithm 3: `UpdateHistogram()` Function

Input : X: Sliding window of Features, b_n: number of bins
Output: **H**: Histogram, f_r: relative frequency

1 **for** *sample s in X* **do**
2 **for** *b in bin* **do**
3   **if** *s in b* **then**
4    $b+=1$;`/* getting frequency */`
5   **else if then**
6   *add bin to b until s in b*
7 **end**
8 f_r=*Calculate using Equation* 4.1;
9 H=*map s to NormalDistribution*;
10 **end**

of observed samples fq_j of the bin_j is increased by one. Moreover, we calculate the relative frequency of each bin as the relation between the bin hit number and the total number of samples, $fr_j = fq_j/N$. Finally, the relative frequency values fr are mapped into a normal distribution by:

$$Z > P\left(x = \sum_{0}^{m} fr_j\right). \quad (4.1)$$

with Equation 4.1 is it possible to see that all values are now mapped into a normal probability distribution with $\mu = 0$ and $\sigma = 1$, line 8 in algorithm 3. As a consequence, all samples are normalized between $-1 \leq x_i \leq 1$.

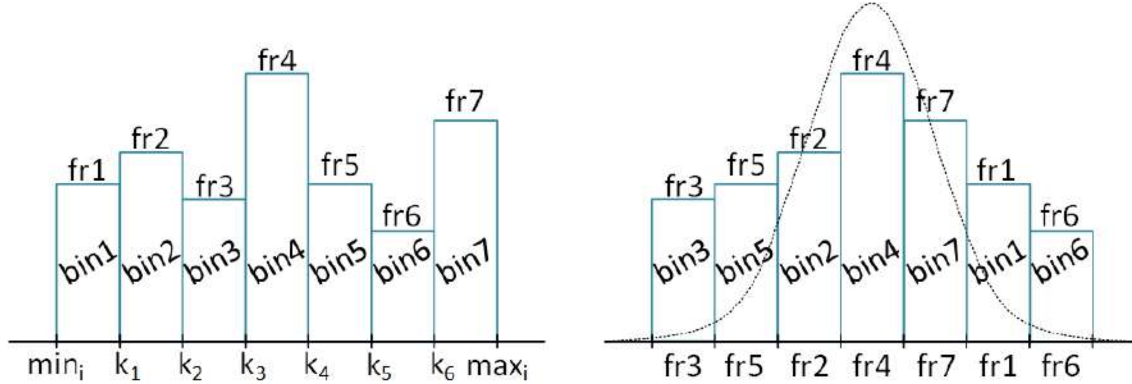


Figure 4.20: Representation of the feature divided in histogram. Each feature is divided in bins that represent the relative frequency of the samples comprised between the thresholds k. The second step of the algorithm approximates the histogram to a normal distribution.

If we consider that the process that generate the stream is non-stationary, it implies a possible concept drift. Haim and Tov affirm that the histogram must be dynamic when dealing with streaming data [114]. As a consequence, intervals do not have fixed value and the bins adapts to concept drift. However, if the bins remain static it reflects the evolution of the change during time [115]. In our application, feature normalization for network monitoring, we follow the former approach. Maintaining fixed intervals allow us to see how a feature evolves over time. In addition, as our histogram algorithm creates new bins when a value does not enter in any of the current intervals, it enables to dynamically detect outliers. In streaming data, it is not possible to maintain all the samples x_i, because it is computationally inefficient and, in case of unlimited data, it does not fit in memory. Our algorithm only efficiently keeps the frequency of each bins.

The most complex function in the normalization process is to update the bins. If the *max* and *min* reference values of the window change, the bins update functions takes the complexity $\mathcal{O}(n)$ on time. The creation of the histogram is only done in the first window and takes constant time. The histogram update uses a binary search to fill the bin value in $\mathcal{O}(\log n)$ time.

4.3.2 The proposed Correlation Based Feature Selection

We propose the Correlation Based Feature Selection, a simple unsupervised filter method for feature selection. Our method is based in the correlation between features. The Pearson correlation of two variables is a measure of their linear dependence. The key idea of the method is to weight each feature based on the correlation of the feature against all other features that describe the dataset. We adopt the Pearson's coefficient as the correlation metric. Pearson's coefficient value is between $-1 \leq \rho \leq 1$, where 1 means that the two variables are directly correlated, linear relationship, and -1 in the case of inverse linear relationship, also called anticorrelation.

The Pearson Coefficient ρ, can be calculated in terms of mean μ and standard deviation σ,

$$\rho(A,B) = \frac{1}{N-1}\sum_{i=1}^{N}\left(\frac{A_i-\mu_A}{\sigma_A}\right)\left(\frac{B_i-\mu_B}{\sigma_B}\right), \tag{4.2}$$

or in terms of the covariance

$$\rho(A,B) = \frac{cov(A,B)}{\sigma_A \sigma_B}, \tag{4.3}$$

then we calculate the weight vector,

$$\mathbf{w_i} = \frac{\sigma_i^2}{\sum_{j=0}^{j=N} |\rho_{ij}|}. \tag{4.4}$$

Firstly, we need to obtain the correlation matrix, calculated by Equation 4.3, line 1 algorithm 4. The correlation matrix is the pairwise covariance calculations between features. Then, applying the Equation 4.4, we establish a weight w that is a measure of the importance of the feature. In order to calculate w, we sum the absolute values of the correlation features, lines 5-6 algorithm 4. This absolute value sum is due to Pearson's coefficient, ρ, may assume negative values. Then we calculate the variance $\hat{V}$ of each feature that privilege the feature that has greater variance and lower covariance, line 8 algorithm 4. The idea is to establish which feature represent the most information, giving the correlation between two features. Furthermore, the weights give us an indication of the amount of information the feature has independently from the others. The weight w has values between $0 \leq N$, where N is the number of features, and 0 means that the features is totally independent of the other. The higher the w value is, the higher is the variance of the feature and lesser correlation with other features, thus more information is the aggregated by this feature.

Algorithm 4: Correlation Based Feature Selection

Input : X: Matrix of Features and Data

Output: $\mathbf{r}$: Vector of Ranked Features, $\mathbf{w}$: Vector of weights

```
1  ρ = Corr(X) /* Correlation Matrix                    */
2  for 0 ≤ i < len(ρ) do
3  |   W_i = 0
4  |   for 0 ≤ j < len(ρ_i) do
5  |   |   k_i = abs(ρ_ij) /* Absolute Values           */
6  |   |   aux_i += k_i /* Sample Addition              */
7  |   end
8  |   w_i = V̂(i)/aux_i /* Calculate Weights            */
9  end
10 r = sort(w, byhighervalues)
```

4.3.3 Evaluation

To evaluate the proposed algorithm, we perform traffic classification to detect threats. We chose the traffic classification application, because it is time sensitive and our algorithm can significantly reduce the processing time, enabling prompt defense mechanisms. We implemented traffic classification using machine learning algorithms against three different datasets, NSL-KDD, GTA/UFRJ and NetOp datasets. The measurements are performed in Intel Xeon processors with clock frequency of 2.6 GHz and 256 GB of RAM.

In the first experiment, we use one day from NetOp dataset to evaluate our normalization method. Shapiro–Wilk test was used to verify that our proposal enforces a normal distribution for the normalized features. Table 4.2 show Shapiro-Wilk test, we considered $\alpha = 0.05$. We evaluate the hypothesis that our proposal normalization method follows a normal distribution. Checking the results, the proposal method has a p-value of $0.24 > 0.05$, and W is closer to one, W=0.93, then we assume that samples are not significantly different than a normal population. In the case of Max-Min normalization [116], p-value is very smaller than the α, and W indicates that it is far from 1. As a consequence, we refuse the hypothesis assuming that sampling data are significantly different than a normal population. Figure 4.21 shows a graphical interpretation of the Shapiro–Wilk test, it represent a sample after being normalized. As our proposal follows the normal distribution, the blue points follow the dashed line, while the max-min approach follows a right skewed distribution.

In the following experiments we verify our preprocessing method in a use case of traffic classification. Thus, we implement the Decision Tree (DT), with C4.5 algorithm, Artificial Neural Networks (ANN), and Support Vector Machine (SVM) as classification algorithms to evaluate the proposed feature selection algorithm. We selected these algorithms because they are the most used ones for network se-

Table 4.2: Hypothesis comparison for a normal distribution approach. In Shapiro-Wilk test p-value is $0.24 > 0.05$, and W is closer to one, W=0.93, confirming that values follow a normal distribution.

	Shapiro-Wilk	
	Mean W	Mean p
proposal	0.93	0.24
max-min	0.65	9.28e-07

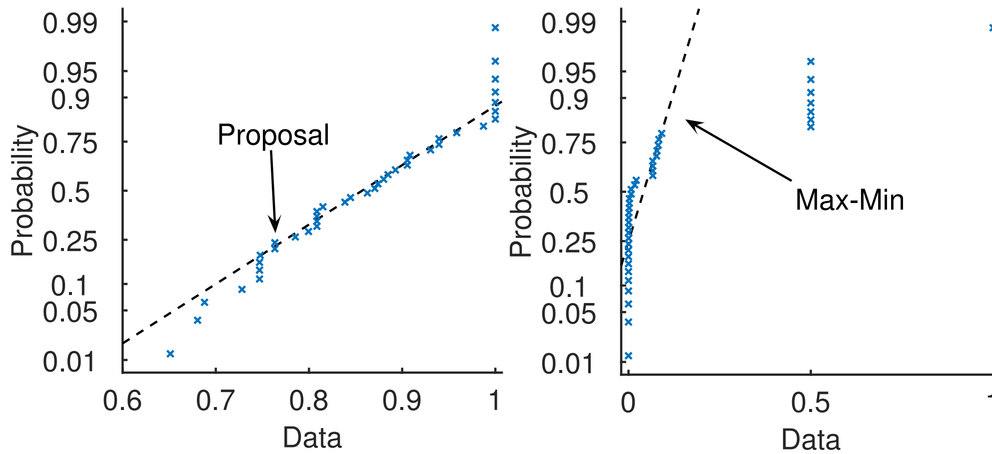


Figure 4.21: Shapiro-Wilk hypothesis test. Our proposal shows a better approximation to a normal distribution when compared with $Max - Min$ proposal

curity [117]. In all methods, the training is performed in a 70% partition of the dataset and the test run over the remaining 30%. During the training phase, we perform a tenfold cross validation to avoid overfitting. In cross validation, parts of the dataset are divided and not used in model parameters estimation. They are further used to check whether the model is general enough to adapt to new data, avoiding overfitting to training data.

The Decision Tree Algorithm

In decision tree, leaves represent the final class and branches represent conditions based on the value of one of the input variables. During the training part, the C4.5 algorithm determines a tree-like classification structure. The real-time implementation of the decision tree consists in if-then-else rules that generate the tree-like structure previously calculated. The results are presented in the Section 4.3.4, along with the ones from the other algorithms.

The Artificial Neural Network Algorithm

The artificial neural networks are inspired on the human brain, in which each neuron performs a small part of the processing and transfers the result to the next

neuron. In artificial neural networks, the output represents a degree of membership for each class, and the highest degree is selected. The weight vectors Θ are calculated during the training. These vectors determine the weight of each neuron connection. In the training, there are input and output sample spaces and the errors, caused by each parameter. Errors are minimized through the back-propagation algorithm.

In order to determine to which class a sample belongs each neural network layer computes the following equations:

$$z_{(i+1)} = \Theta_{(i)} a_{(i)} \quad (4.5) \qquad a_{(i+1)} = g(z_{(i+1)}) \quad (4.6) \qquad g(z) = \frac{1}{1+e^{-z}} \quad (4.7)$$

where a is the vector that determines the output of layer i, $\Theta_{(i)}$ is the weight vector that leads layer i to layer $i+1$, and $a_{(i+1)}$ is the output of layer $i+1$. The function $g(z)$ is the activation function, represented by *Sigmoid* function, which plays an important role in the classification. For high values of z, $g(z)$ returns one and for low values $g(z)$ returns zero. Therefore, the output layer gives the degree of membership of each class, between zero and one, classifying the sample as the highest one. The activation function enables and disables the contribution of a certain neuron to the final result.

The Support Vector Machine Algorithm

The Support Vector Machine (SVM) is a binary classifier, based on the concept of a decision plane that defines the decision thresholds. SVM algorithm classifies through the construction of a hyper-plane in a multidimensional space that split different classes. An iterative algorithm minimizes an error function, finding the best hyper-plane separation. A kernel function defines this hyper-plane. In this way, SVM finds the hyper-plane with a maximum margin, that is, the hyper-plane with the biggest distance possible between both classes.

The real-time detection is performed by the classification to each class pairs: normal and non-normal; DoS and non-DoS; and probe and non-probe. Once SVM calculates the output, the chosen class is the one with the highest score. The classifier score of a sample x is the distance from x to the decision boundaries, that goes from $-\infty$ to $+\infty$. The classifier score is given by:

$$f(x) = \sum_{j=1}^{n} \alpha_j y_j G(x_j, x) + b, \tag{4.8}$$

where $(\alpha_1, ..., \alpha_n.b)$ are the estimated parameters of SVM, and $G(x_j, x)$ is the used kernel. In this work, the kernel is linear, that is, $G(x_j, x) = x_j' x$, which presents a good performance with the minimum quantity of input parameters.

4.3.4 Classification Results

This experiment shows the efficiency of our feature selection algorithm when compared with literature methods. We try a linear Principal Component Analysis (PCA), The ReliefF algorithm, the Sequential Forward Selection (SFS), and the Support Vector Machine Recursive Feature Elimination (SVM-RFE). For all methods, we analyze their version with four and six output features. For the sake of fairness, we tested all the algorithms with the classification methods presented before. We use a decision tree with the with a minimum of 4096 leaves, a binary support vector machine (SVM) with linear kernel, and finally a neural network with one hidden layer with 10 neurons. We use ten-fold cross validation for our experiments.

Figure 4.22 presents information gain (IG) sum of the selected feature for each evaluated algorithm. Information gain measures the amount of information, in bits, that a feature adds in relation to the class prediction. Thus, it is computed as the difference of target class entropy and the conditional entropy of target class given the feature value as known. When employing six features, the results show our algorithm has information retention capability between SFS and ReliefF, and greater than SVM-RFE. The information retention capability of PCA, is greater than feature selection methods, as each feature is a linear combination of the original features and is computed to retain most of dataset variance.

Figure 4.23 shows the accuracy of the three classification methods, Decision Tree, Neural Network and Support Vector Machine (SVM), when the input variables are chosen by different dimensionality reduction methods. In the first group, our proposal with six features reaches 97.4% accuracy, which is the best results for the decision tree classifier. The following result is PCA with four and six features in

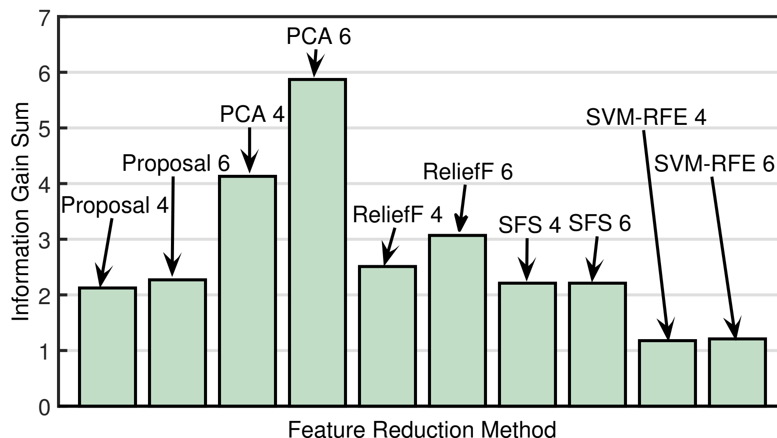


Figure 4.22: Information gain sum for feature selection algorithms. The selected features by our algorithm keeps an information retention capability between SFS and ReliefF.

96% and 97.2%. The Sequential Forward Selection (SFS) presents the same result with four and six features with 95.5%. The ReliefF algorithm has the same results in both four and six features as 91.2%. Finally, the lowest result is show by SVM-RFE algorithm with four and six features. As the decision tree algorithm creates the decision nodes based on the variables with greater entropy, the proposed feature selection algorithm better performs because it keeps most of the variance of the dataset.

The second classifier, the neural network, the best result is shown by the PCA with six features in 97.6% of accuracy, however the PCA with four features present a lower performance with 85.5%. ReliefF presents the same results for both features in 90.2%. Our proposal shows a result with 83.9% and 85.0% for four and six features. On the other hand, the SFS presents the worst results of all classifiers, 78.4% with four features and 79.2% with six features. One impressive result is the SVM-RFE, with four features presents a very low result of 73.6% that is one of the worst for all classifiers, however, with six features present almost the same best second result with 90.1%.

In the Support Vector Machine (SVM) classifier, the PCA presents a similar behavior compared with the neural networks. For six features presents the highest accuracy of all classifiers with 98.3%, but just 87.8% for four features. ReliefF again presents the same result for both cases in 91.4%. Our proposal has 84% for four features and 85% for six features. SFS present the same result for both features in 79.5%. The lowest accuracy of this classifier is the SVM-RFE with 73.6% for both cases. As our proposal maximizes the variance on the resulting features, the resulting reduced dataset is spread into the new space. For a linear classifier, such as SVM, it is hard to define a classification surface for a spread data. Thus, the resulting accuracy is not among the highest. However, as the selected set of features still being significant for defining the data, the resulting accuracy is not the worst one.

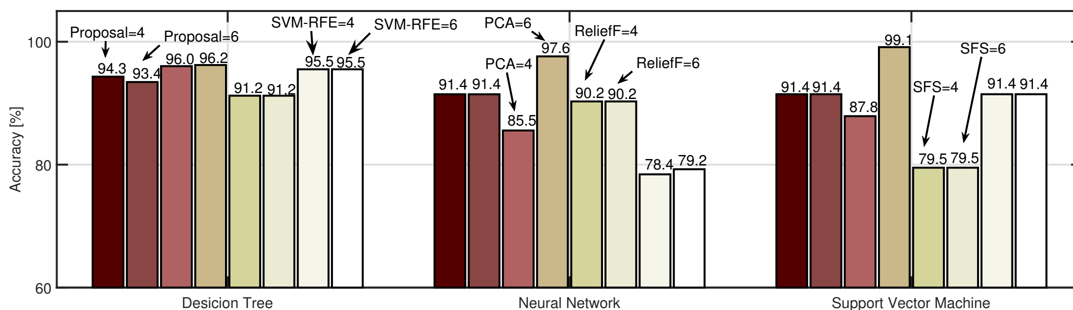


Figure 4.23: Accuracy comparison of features selection methods. Our Proposal, Linear PCA, ReliefF, SFS and SVM-RFE compared in decision tree, SVM, and neural network algorithms.

The Sensitivity metric shows the rate of correctly classified samples. It is a good a metric to evaluate the success of a classifier, when using a dataset in which a class has much more samples than others. In our problem, we use sensitivity as metric to evaluate our detection success. For this, we consider the detection problem as a binary classification, i.e., we consider two classes: a normal and an abnormal traffic. In this way, the Denial of Service (DoS) and Port Scanning threat classes were joined into a common attack class. Similar to Accuracy representation in Figure 4.23, Figure 4.24 represents the sensitivity of the classifiers applying the different methods of feature selection. In the first group, the classification with Decision Three, PCA shows the best sensitivity with 99% of correct classification, our algorithm achieves a performance of almost 95% of sensitivity, with four and six features. Neural Networks, represented in the second group, have the best sensitivity with PCA using six features with 97.7%, then our results show a good performance with both four and six features in 89%. In this group the worst sensitivity of all classifiers is reached by the SVM-RFE with four and six features in 69.3%. Finally, the last group shows the Sensitivity for Support Vector Machine (SVM) classifier. Again, showing a similar behavior as the previous group PCA with six features shows the best sensitivity with 97.8%. Then, the second-best result is reached by our algorithm, as well as with ReliefF, with 89% of sensitivity with both features. It is worthy to note that our algorithm presents a stable behavior in Accuracy as well as in Sensitivity. We highlight that our algorithm performs nearly equal to PCA. PCA creates artificial features that are a composition of all real features, while our algorithm just selects some features from the complete set of features. In this way, our algorithm was the best feature-selection method that was evaluated, and it also introduce less computing overhead when compared with PCA.

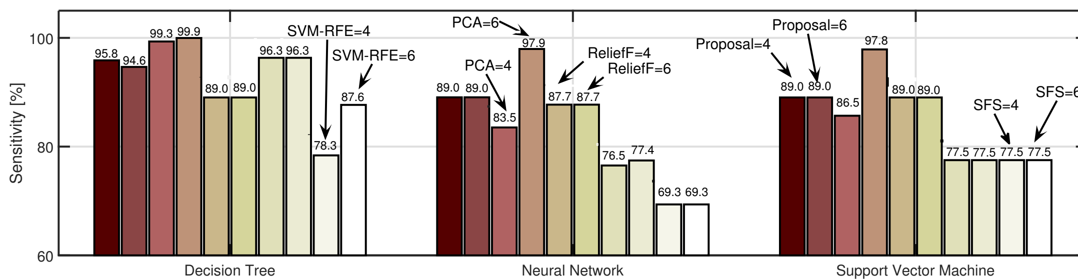


Figure 4.24: Sensitivity of detection in decision tree, SVM, and neural network algorithms for feature selection methods.

When analyzed the features each method chooses, it is possible to see none of the methods selects the set of same features. Nevertheless, ReliefF and SFS select as the second-best *amount of IP packets*. One surprising result from the SFS is the election of *Amount of ECE Flags* and *Amount of CWR Flags*. In a correlation test, these two features show that any information is added because they are empty

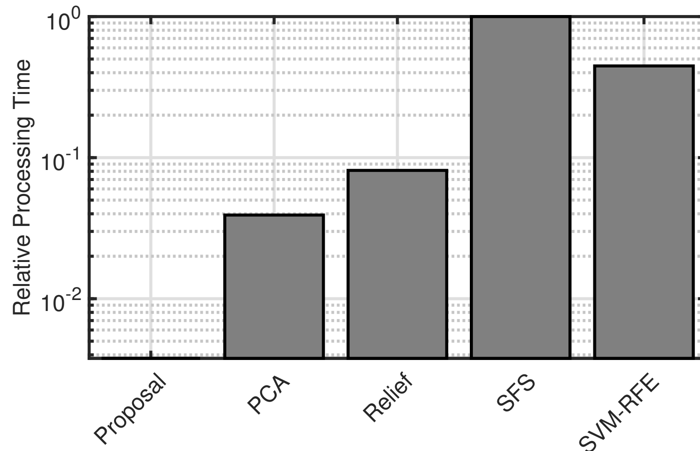


Figure 4.25: Performance of features selection algorithms according to processing time. The proposal and the PCA show the best processing time.

variables. However, we realized that one of the main features is *Average Packet Size*. In this dataset, the average packet size is fundamental to classify attacks. One possible reason is that during the creation of the dataset the Denial of Service (DoS) and Probe attacks were performed for an automated tool. Mainly this automated tool produces attacks without altering the length of the packet.

Figure 4.25 shows a comparison of processing time of all implemented feature selection and dimensionality reduction methods. All measures are in relative value. We can see that SFS show the worst performance. The SFS algorithm performs multiple iteration in order to minimize the mean square error (MSE). Consequently, all these iterations increase the processing time. Our proposal shows the best processing time together with PCA, because both implementations perform a matrix multiplication. Matrix multiplication is a simple computation function.

The next experiment is to evaluate our proposal in different dataset. We use the NSL-KDD dataset and the NetOp dataset. Besides linear SVM, Neural Network and Decision Tree, we also evaluate K-Nearest Neighbors (KNN), Random Forest, two kernels, linear and Radial Basis Function (RBF) kernel in Support Vector Machine (SVM), Gaussian Naive Bayes, and Stochastic Gradient Descendant. Adding these algorithms, we cover the full range of the most important algorithms for supervised machine learning.

The **Random Forests** (RF) algorithm avoids overfitting when compared to the simple decision tree because it constructs several decision trees that are trained in different parts of the same dataset. This procedure decreases variance of classification and improves the performance regarding the classification of a single tree. The prediction of the class in the RF classifier consists of applying the sample as input to all the trees, obtaining the classification of each one of them and, then, a voting system decides the resulting class. The construction of each tree must follow the

rules: (i) for each node d, select k input variables of total m input variables, such that $k \ll m$; to calculate the best binary division of the k input variables for the node d, using an objective function; repeat the previous steps until each tree reaches l number of nodes or until its maximum extension.

The simple Bayesian classifier (**Naive Bayes** - NB) takes the strong premise of independence between the input variables to simplify the classification prediction, that is, given the value of each input variable, it does not influence the value of the other input variables. From this, the method calculates the probabilities *a priori* of each input variable, or a set of them, to set up a given class. As a new sample arrives, the algorithm calculates for each input variables the probability of being a sample of each class. The output of all probabilities of each input variable result in *a posterior* probability of this sample belonging to each class. The algorithm, then, returns the classification that contains the highest estimated probability.

In the **k-Nearest Neighbors** (k-NN) the class definition of an unknown sample is based on the k-neighbors classes closest to the sample. The value k is a positive integer and usually small. If $k = 1$, then the sample class is assigned to the class of its nearest neighbor. If $k > 1$, the sample class is obtained by starting a resultant function, such as a simple voting or weighted voting, of the k-neighbors classes. The neighborhood definition is based on a measure of similarity between samples in the feature space. Euclidean distance is commonly used in the threat detection literature, however, other distances have good results and the best choice for similarity measure will depend on the type of dataset used [118]. The Euclidean distance of two samples $\mathbf{p}$ and $\mathbf{q}$ in the space of n features is given by

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i}^{n} (p_i - q_i)^2}. \tag{4.9}$$

Stochastic Gradient Descent with Momentum: This scheme relies on the Stochastic Gradient Descent (SGD) [119] algorithm, is a stochastic approximation of Gradient Descent, in which the gradient is approximated by a single sample. In our application, we consider two classes, normal and threat. Therefore, we use the Sigmoid Function, expressed by

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^\intercal x}}, \tag{4.10}$$

to perform logistic regression. In the Sigmoid function, low product values of the parameters $\theta^\intercal$ times the sample feature vector x return zero, whereas high values return one. When a new sample $x_{(i)}$ arrives, the SGD evaluates the Sigmoid function and returns one for $h_\theta(x_{(i)})$ greater than 0.5 and zero otherwise. This decision presents an associated cost, based on the real class of the sample $y_{(i)}$. The cost

function is defined in Equation 4.11. This function is convex and the goal of SGD algorithm is to find its minimum, expressed by

$$J_{(i)}(\theta) = y_{(i)}log(h_\theta(x_{(i)})) + (1 - y_{(i)})log(1 - h_\theta(x_{(i)})). \tag{4.11}$$

On each new sample, the algorithm takes a step toward the cost function minimum based on the gradient of the cost function.

Validation in NSL-KDD and NetOp Datasets

The first experiment evaluates the performance of the feature selection in both datasets. In this experiment we vary the number of selected features to evaluate the impact in the accuracy. We analyze the performance with no feature selection (No FS), and then we reduce features from 10% to 90% of the original set of features. All the experiments were performed using a K-fold cross-validation. The K-fold cross-validation performs K training iterations in the partitions of the data and, at each iteration, in the remaining $K-1$ partition, the K-fold cross-validation performs the test in a mutually exclusive manner. We use $K = 10$, which is commonly used. Figure 4.26 shows the effect of feature selection. No Feature Selection performs well for almost all algorithms. Reducing the number of features in 10%, however, improve accuracy in all algorithms, except for Random Forest. In contrast, a bigger reduction of feature deteriorates the accuracy for all classifiers.

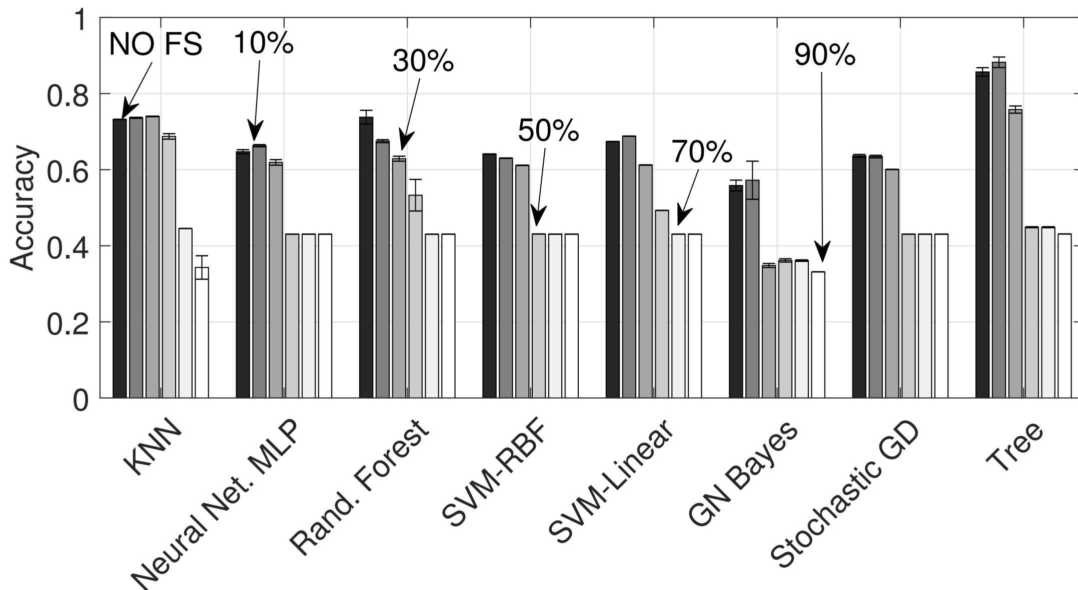


Figure 4.26: Evaluation of Feature Selection varying the selected features in NSL-KDD dataset.

Besides Accuracy, we also measure other metrics, such as Sensitivity, Precision, F1-Score, training and classification time. The **accuracy** of a method is the ratio of

the total of correctly sorted samples (True Positives (TP) + True Negatives (TN)) divided by the total number of samples. The **sensitivity**, also known as **recall** or **true positive rate**, is the ratio of the number of correctly classified samples to the positive class (TP), divided by the total of samples belonging to this class (TP+ FN). The analog for the negative class is called **specificity** or **true negative rate**. the **precision** is the ratio between the number of samples correctly classified for the positive class (TP), divided by the total of samples classified for this class (TP + FP). There is a natural compromise between the sensitivity and precision of an algorithm and balancing good accuracy with good sensitivity is a difficult task. **F1-score** is the harmonic mean of the precision with the sensitivity, expressed by

$$F1\ score = \frac{2}{\frac{1}{Precision} + \frac{1}{Sensitivity}}. \tag{4.12}$$

We compare the effect of 10% reduction in all these metrics. Figure 4.274.28 show accuracy, precision, sensitivity and F1 - score for dataset with no feature selection Figure 4.27 and with 10% of reduction Figure 4.28. For KNN, SVM with Radial Basis Function (RBF) kernel, and Gaussian Naive Bayes metrics remain the same. For the Neural Network MLP and for SVM with linear kernel an improvement between 2-3% in all metrics is reached with a 10% of features reduction. Random Forest present the worst performance when features are reduced, all metrics worsen their values between 8-9%. Stochastic Gradient Descendant (SGD) also suffer a small reduction of 1% in their metrics. Decision tree are the most benefited improving between 3-4% their metrics, which shows the capability of reducing overfitting when applying feature selection.

Figure 4.29 shows training and classification time with no features selection, while Figure 4.30 shows results with 10% of reduced features. KNN algorithm augmented considerably its training time, passing from 0.63 seconds to 5.03, while classification time also suffer an increase passing from 1.89 seconds to 2.88. Neural Network reduced 9% of the training time, from 22.99 seconds to 20.92, classification time got 0.01 second increased. Random Forest training time increased 0.02 second, and classification time remained the same, which is negligible because of the intrinsic error of the cross validation. SVM with Radial Basis Function (RBF) and SVM with linear kernel are the most benefited from features selection. SVM-RBF training time reduced 11% while the classification time, 16%. SVM-Linear classification time reduced 46%, from 654 seconds to 349 seconds, and training time, 40%, from 54.86 to 32.88 seconds. Feature selection in Gaussian Naive Bayes, Stochastic Gradient Descendant and Decision Tree strongly impacts in training time with an approximate reduction of 30%, while the classification time was reduced one-time unit in three algorithms.

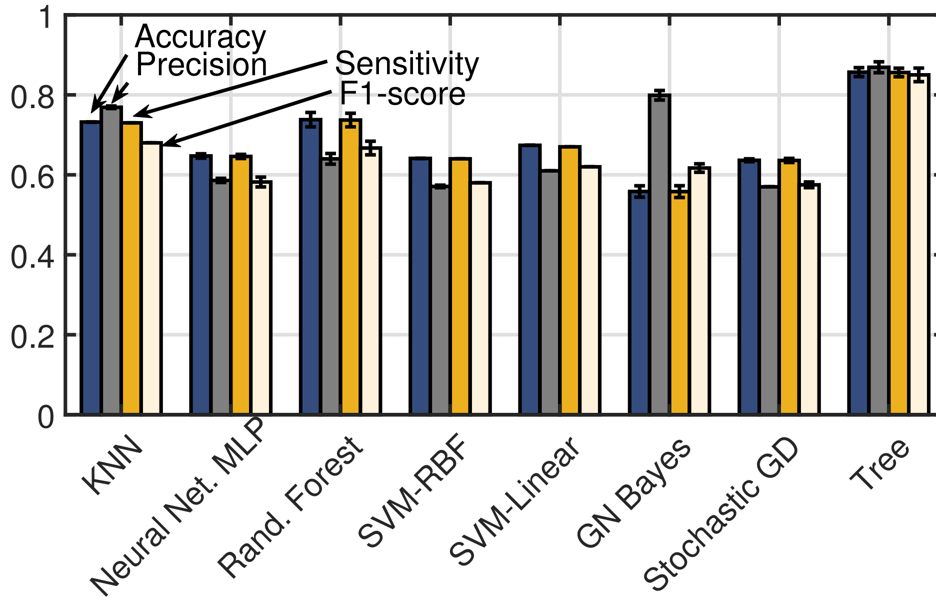


Figure 4.27: Accuracy, precision, sensitivity and F1-Score for NSL-KDD. Metrics with no future selection.

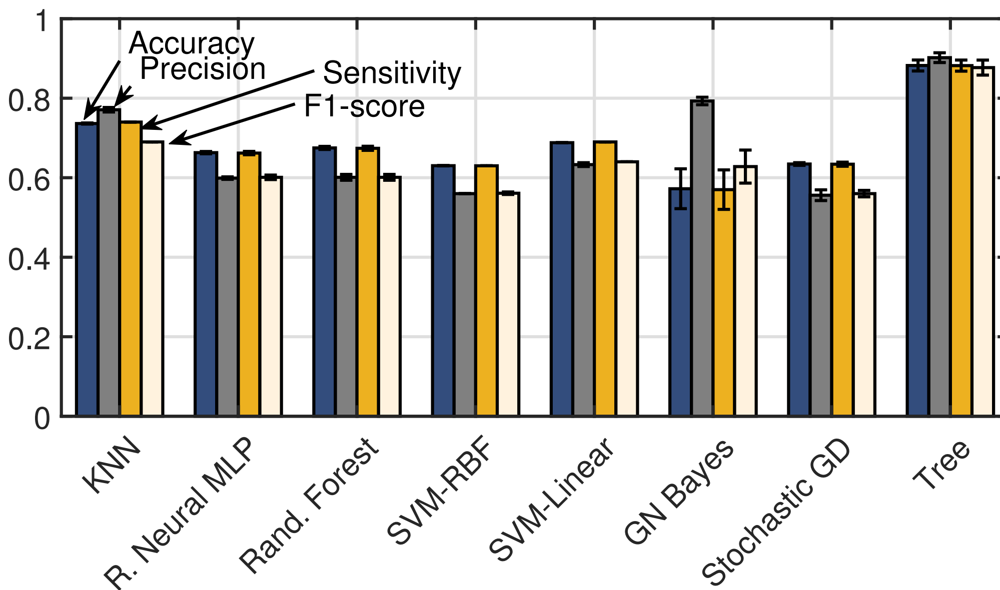


Figure 4.28: Metrics reducing only 10% of the initial features in NSL-KDD.

We performed the same experiment in the NetOp Dataset. Figure 4.31 shows the accuracy of different classifiers while reducing from 10% to 90% of the features. Using the NetOp dataset, applying feature selection keeps unaffected classifier accuracy unaffected. In the case of KNN, the accuracy variation is less than 0.02%. A similar case occurs with Neural Networks, SVN with linear and with RBF kernels, Stochastic Gradient Descendant and Decision Tree. In Random Forest, the best accuracy is found with a reduction of 30% of the original set of features of the dataset.

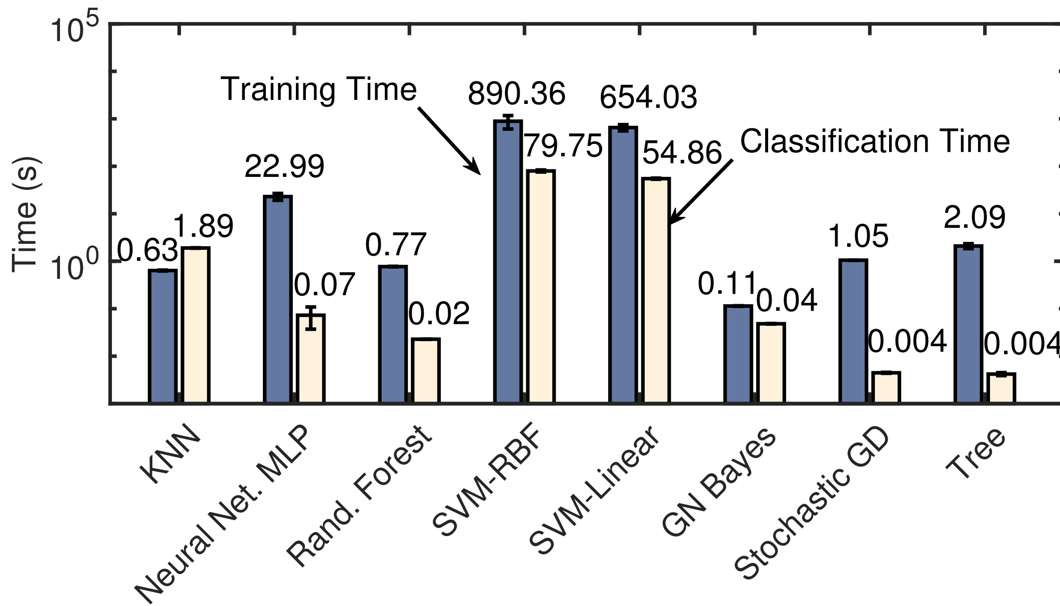


Figure 4.29: Classification and training time in NSL-KDD Dataset with no feature Selection.

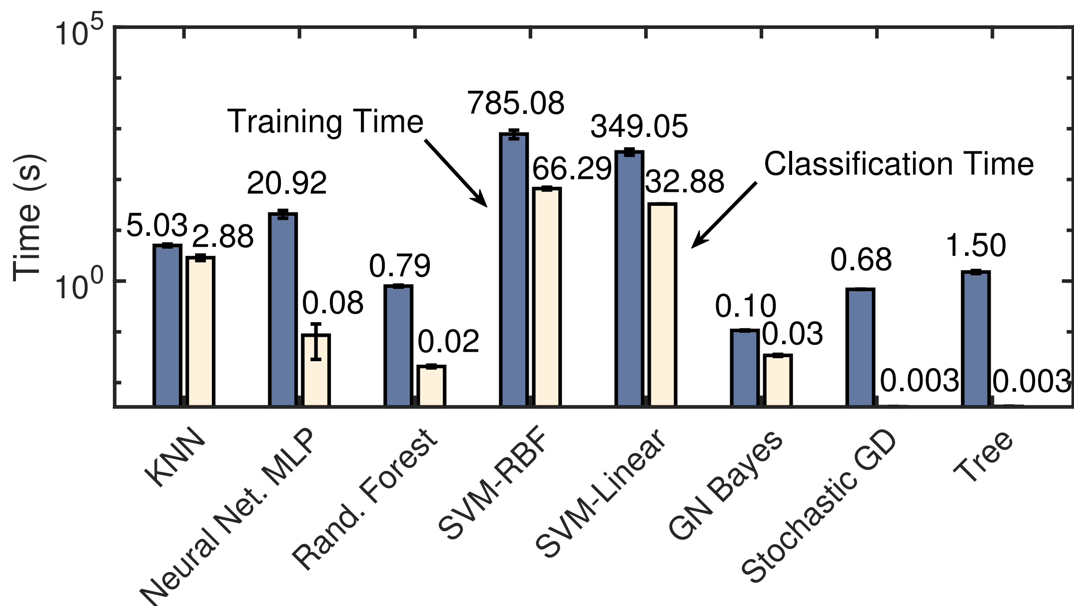


Figure 4.30: Classification and training time in NSL-KDD Dataset with only 10% of the initial features.

The best result is reached in Gaussian Naive Bayes, in which 90% of reduction in the selected features increases the accuracy from 57% to 78%, using only five features.

Reducing 90% of selected features, we analyze other metrics, such as Precision, Sensitivity and F1-Score, for all classifiers. We compare the results with no feature selection, Figure 4.32, and with only five features, Figure 4.33. All metrics remains almost equal. A slight positive variation is produced in Gaussian Naive Bayes and

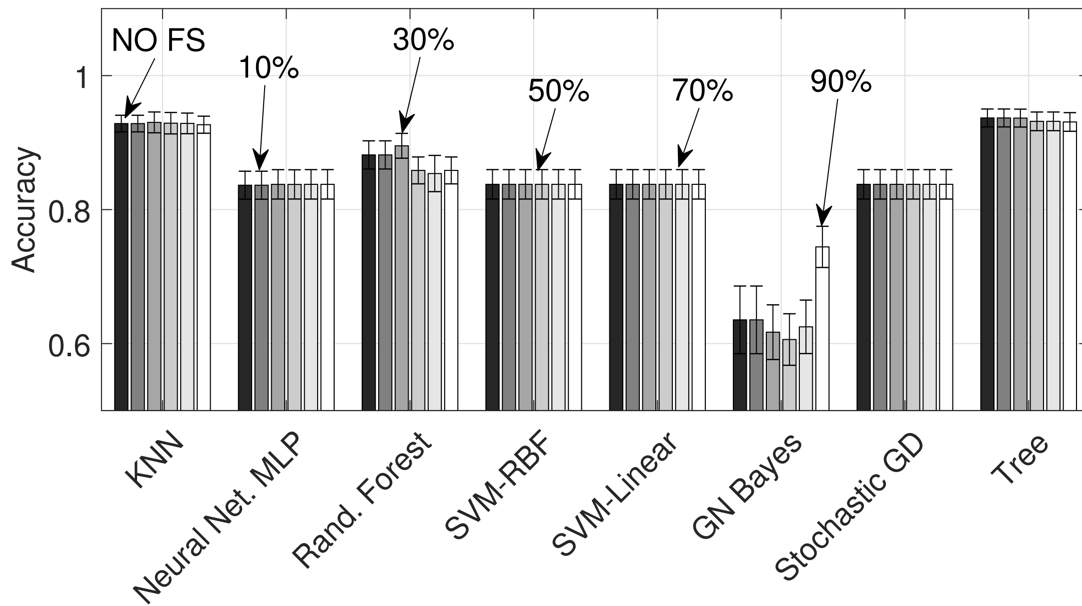


Figure 4.31: Evaluation of Feature Selection varying the selected features in NetOp dataset.

in Random Forest. We conclude that, for this dataset, our Feature Selection method maintains the metrics unaltered or increase classifier performance, because our proposal keeps the most of independent features in dataset.

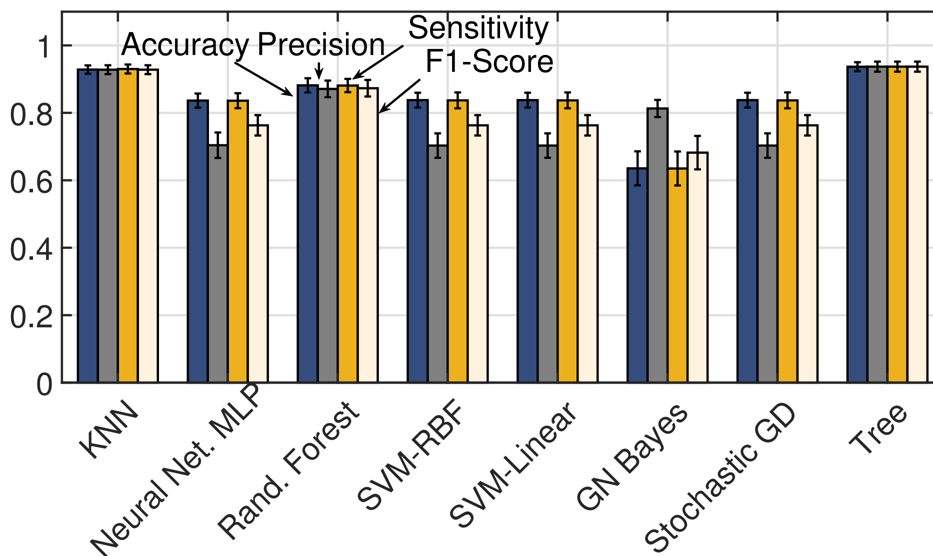


Figure 4.32: Accuracy, precision, sensitivity and F1-Score for NetOp dataset. Metrics with no future selection.

Figure 4.34 shows the training and classification times with no feature selection, while Figure 4.34 shows the training and classification times for the dataset with 90% of feature reduction. All the classifiers reduced their times. KNN training time

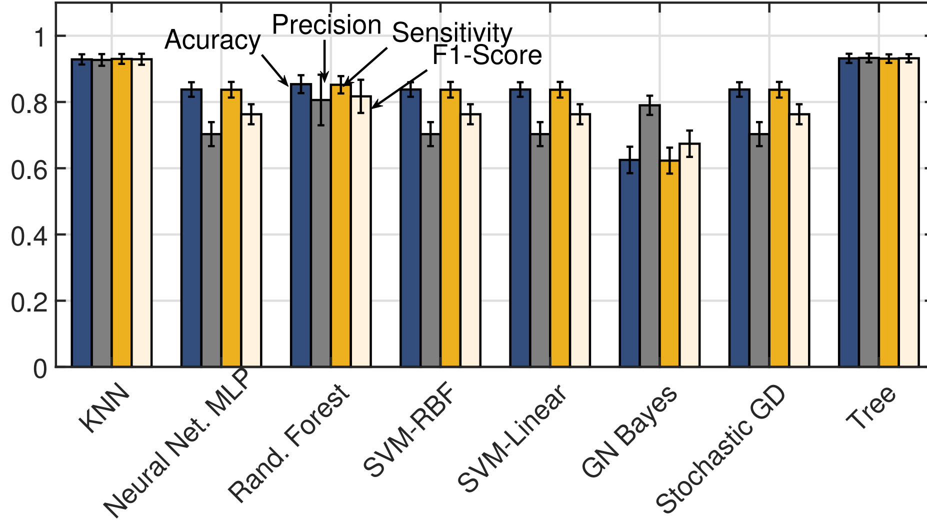


Figure 4.33: Metrics reducing only 90% of the initial features.

is reduced in 71%, while classification time is reduced in 84%. For Neural Networks reduced the training time in 25% and classification time is reduced in 0.02 seconds. Random Forest reduced their training time in 38% while their classification time remains the same. SVM with RBF kernel training time is reduced in 78% and training time is reduced in 54%. SVM with linear kernel received the biggest improvement. Training time was reduced in 88% while classification time was reduced in 81%. Gaussian Naives Bayes reduced their training time in 80% while classification time was reduced in 76%. Stochastic Gradient Descendant also show a reduction of 61% in training and 66% for classification time. Finally, Decision Tree reduced training time in 79% and classification time got faster, being reduced in 28%. As a consequence, a feature reduction of 90% impacts directly in the training and classification time of the machine learning classifiers. Therefore, our Feature Selection method improves training and classification times in all classifiers.

In this experiment we show what is the most important group of features. Thus, we group features into eight groups in the NetOp dataset. Flow tuple information features are removed because our algorithm works on numerical features and tuple information features are categorical. Table 4.3 describes the groups. We established the window size at 1000 samples. Figure 4.36 shows accuracy for each seven algorithms for classification. In Decision Tree all groups show a similar behavior and present high accuracy. Gaussian Naive Bayes and SVM with linear kernel for group 3, Time Between Packets, and for group 5, SubFlow information, present the lowest accuracy. For the rest of the groups, these classifiers also reach high accuracy. K-Nearest Neighbors (KNN) shows a special case, besides group 2, which is the highest accuracy, all the other groups show a different behavior. In Neural Networks, groups 2 and 3, Packet Statistics and Time between Packets, show the highest accuracy,

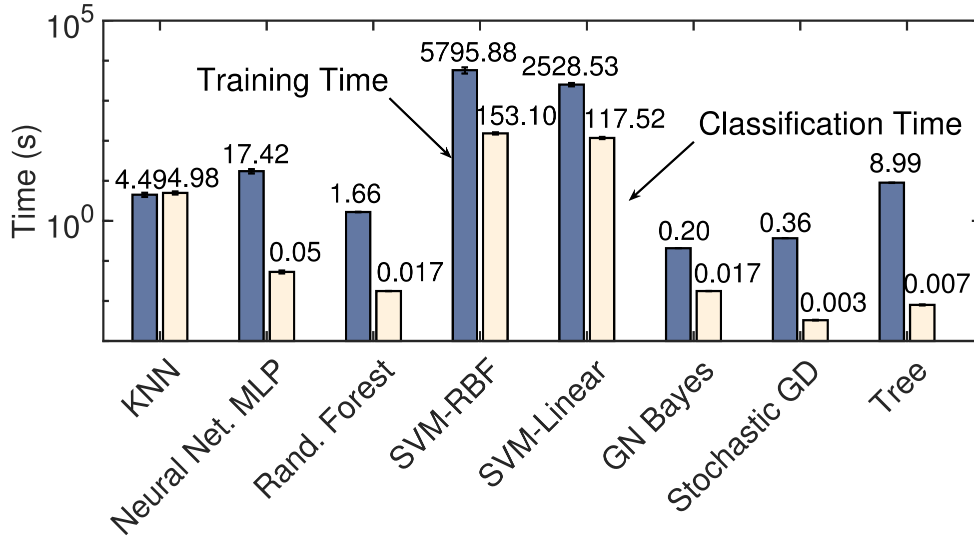


Figure 4.34: Classification and training time in NetOp Dataset with no feature Selection.

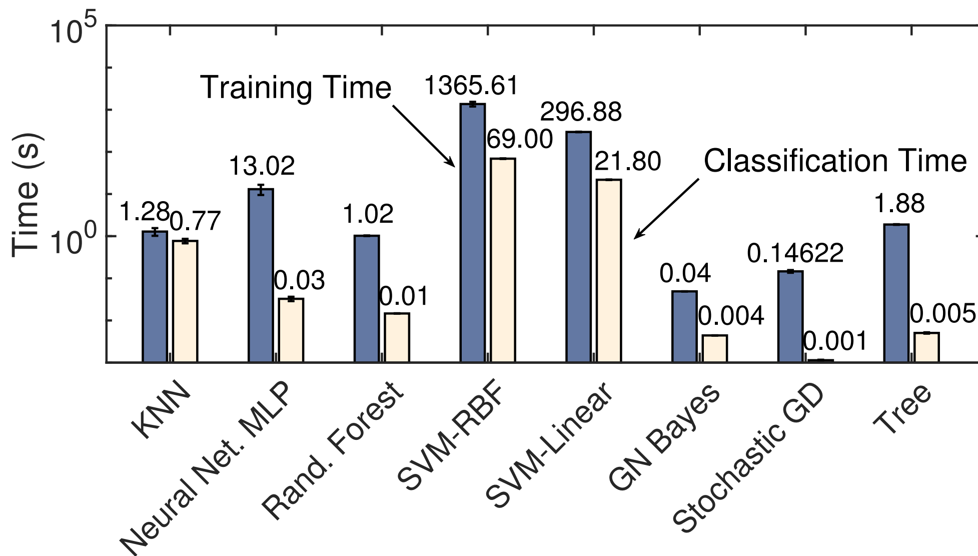


Figure 4.35: Classification and training time in NetOp Dataset with only 90% of the initial features.

while the reminding groups maintain in 50%. Random Forest show a similar behavior than Decision Tree, with high accuracy in all their groups. Nevertheless, the group 5, SubFlow information, present the lowest accuracy. Stochastic Descendant Gradient show the highest accuracy in group 2,6 and 7. We conclude that group 2, Packet Statistics, is the most important for the accuracy calculation for all the classifiers.

Finally, this experiment shows how our preprocessing method when executing with machine learning classifiers in stream data, is able to detect concept drift. This experiment also demonstrate that the proposed preprocess method is able to run

Table 4.3: Features Groups

Group	Description	Number of Features
G1	Packet Volume	4
G2	Packet Statistics	8
G3	Time Between Packets	8
G4	Flow Time Statistics	9
G5	SubFlow Information	4
G6	TCP Flags	4
G7	Bytes in headers + ToS	3

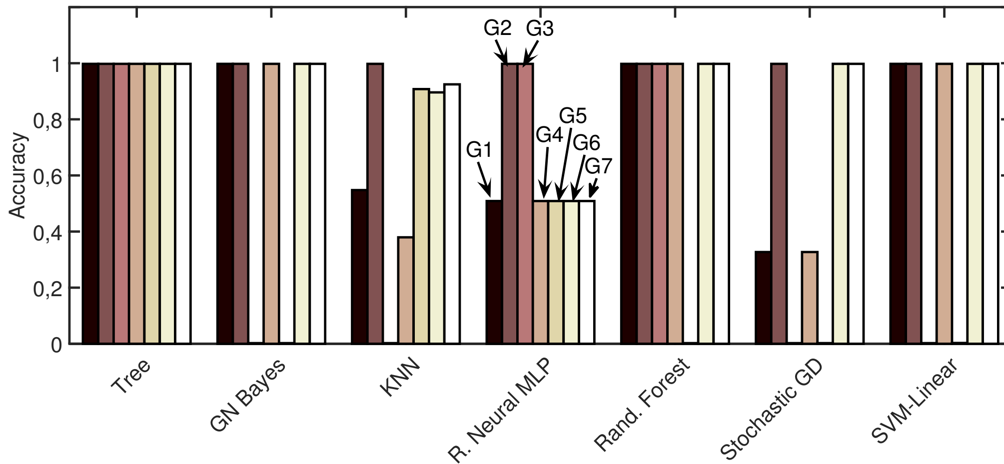


Figure 4.36: Evaluation of group features with different machine learning algorithms.

under batch and stream data. We use the flow diagram of the Figure 4.37. We force traditional learning methods to become adaptive learners in order to detect the concept drift. Adaptive learners dynamically adapt to new training data when learned concept is contradict. Once a concept drift is detected, a new model is created.

We validate the proposal with the NetOp dataset. The dataset is labeled in threats and normal traffic, a binary classification problem. We divide the dataset in training set and test set, in a relation of 70% for training and 30% for test. We consider the training set as static, in which T consecutive sample windows have been presented for training. We have used the Synthetic Minority class Oversampling TEchnique (SMOTE) [120] approach to oversampling the minority class, only in the training set, initial window. When a number of samples in a class is predominant in the dataset, it is called class imbalance. Class imbalance is typical in our kind of threat detection application when attacks are rare events in relation to normal traffic. The test set is streaming data arriving with the same frequency. Data is grouped in a sliding window N samples.

Figure 4.38 show the accuracy when we analyze one day from NetOp dataset. In the experiment, we measure the impact of the concept-drift in the final accuracy.

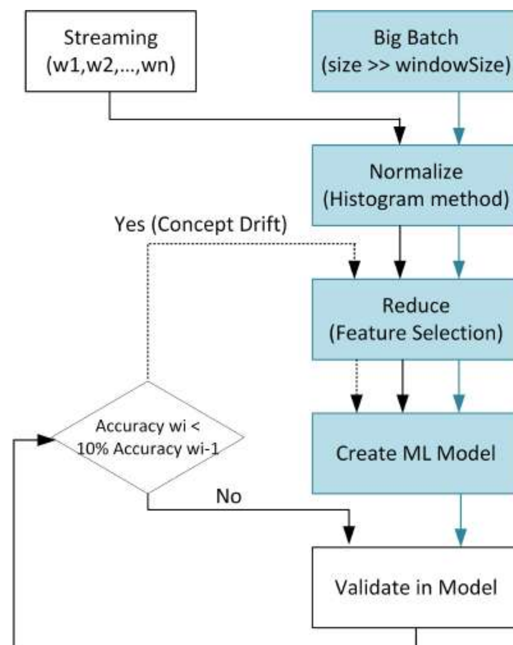


Figure 4.37: Flow diagram used for proposal evaluation.

We train different static algorithm with 30% of the dataset. We use 1000 samples as window size. The trained static algorithms are the Support Vector Machine (SVM) with linear kernel, and with Radial Basis Function (RBF) kernel, Gaussian Naive Bayes, Decision Tree, and Stochastic Gradient Descent (SGD). The decision tree has the worst accuracy when compared with the other algorithms. Decision tree shows a low accuracy in the second window. This behavior means that the created model during the training step do not fully represent the model of the entire dataset. Stochastic Gradient Descendant shows a similar behavior of decision tree, having a concept drift in the second window. The SVM with linear kernel presents a concept drift in the seventh window. SVM with RBF shows a lower accuracy during all experiment and a concept drift at the last window. Finally, due to the implementation of the Gaussian Naive Bayes, it follows the same probability distribution as our normalization method, as consequence does not present any concept drift.

4.3.5 Related Work

State-of-art proposals focus on algorithms for online feature selection. Perkins and Theiler Grafting algorithm based on a stage wise gradient descent approach. Gradient Feature Testing (grafting) [121] treats feature selection as an integral part of learning a predictor within a regularized framework. The objective function is a binomial negative log-likelihood loss. The grafting method incrementally add a feature set while a predictor model is trained in a iterative way. In each step of

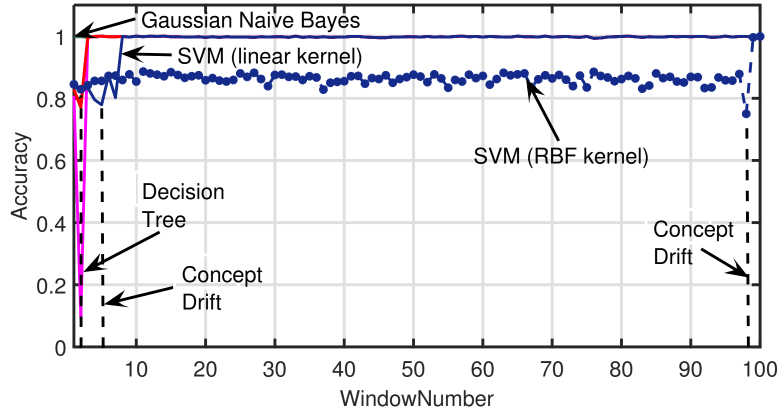


Figure 4.38: Concept Drift detection. Evaluation of preprocessing methods for concept drift detection. Our proposal was able to detect early concept drift in SGD and in SVM with linear kernel. Gaussian Naive Bayes shows a very high performance with no concept-drift.

the iteration, a heuristic based on gradient descendant verify if the selected feature most likely to improve the existing model. Grafting use a λ parameter to regularize the overfitting of the model when a new feature is added. However, the λ parameter must be determined by empirical test or with previous information about the global feature set. Therefore, Grafting is ineffective in dealing with streaming features with an unknown feature size.

The Alpha-investing method [122] considers that new features arrive in a stream manner generated sequentially for a predictive model. The main advantage of Alpha-investing is the possibility to handle candidate feature sets of unknown or even infinite sizes. Every time a feature arrives, alpha-investing uses a linear regression to dynamically reduce the threshold of error to be included in the model. As a drawback, alpha-investing only consider the addition of new features without evaluating the redundancy after the feature is added.

Wu *et al.* presented the OSFS (Online Streaming Feature Selection) algorithm and its faster version, the Fast-OSFS algorithm, in order to avoid the redundancy of added features [123]. The OSFS algorithm uses a Markov blanket of a feature to determine the relevance of the feature in relation with their neighbors. The Markov blanket for a node A in a Bayesian network is the set of nodes A composed of A's parents, its children, and its children's other parents. In a Markov random field, the Markov blanket of a node is its set of neighboring nodes. A Markov blanket may also be denoted by $MB(A)$. Facing the scalability and online processing challenges in big data analytics, the computational cost inherent in those three algorithms may still be prohibitive when the dimensionality is extremely high, likewise millions or more features.

Smart Preprocessing for Streaming Data (SPSD) is an approach that use min-max normalization of numerical features [116]. The authors use two metrics to avoid unnecessary renormalization. SPSD only renormalizes when a threshold exceeds some threshold value of the metrics. Streaming data is grouped in equal size chunk where all operations are produced. The first data chunk is used to take the references min-max values and to send the normalized data for the training model. The metric 1 represents the amount of sample falling outside the min-max reference values, the metric 2 is the relation between new sample values in each dimension, and the referenced min-max value for that dimension. Similar to our proposal, the algorithm works with numerical data.

Incremental Discretization Algorithm (IDA) uses a quantile approach to discretize data stream [115]. The algorithm discretizes data stream in m equal frequency bins. A sliding window version of the algorithm is proposed to follow the evolution of the data stream. To follow the distribution drift, the algorithm maintains the data into bins with fixed quantiles of the distribution, rather than fixed absolute values.

In our proposal, we propose an unsupervised preprocessing method. Our method include normalization, and feature selection all together. The proposal is parametric-less. Our algorithm follows an active approach for concept drift detection. The active approach monitors the concept, the label, to determine when a drift occurs before take any action. Passive approach, in contrast, updates the model every time new data arrives, wasting resources. We adapted our proposed Feature Selection algorithm to calculate correlation between features in a sliding window. Also, a normalization algorithm is proposed to handle data stream.

This chapter presented two security datasets. The first dataset was created at laboratory GTA/UFRJ and presents three classes, DoS, Probe and Normal, composed by 25 features. The second dataset, NetOp, was more than ten days of real traffic for a network operator in Brazil. NetOp dataset is divided in two classes, normal traffic and alerts and it is composed of 45 features. Also, we present an unsupervised pre-processing algorithm. The evaluation of the algorithms shows a good behavior in batch mode and in stream mode, being able to reach a good performance in batch mode and capable of detecting concept drift in stream mode.

Table 4.4: Features description of NetOp Dataset. FD: Forward Direction; BD: Backward Direction.

Number	**Name**	**Description**
1	srcport	Source port number
2	dstport	Destination port number
3	proto	Protocol
4	total_fpackets	Total packets in FD
5	total_fvolume	Total bytes in FD
6	total_bpackets	Total packets in BD
7	total_bvolume	Total bytes in the BD
8	min_fpktl	Size of the smallest packet sent in FD
9	mean_fpktl	Mean size of packets in FD
10	max_fpktl	Size of the largest packet in FD
11	std_fpktl	Std. deviation from the mean of the packets in FD
12	min_bpktl	Size of the smallest packet in BD
13	mean_bpktl	Mean size of packets in BD
14	max_bpktl	Size of the largest packet in BD
15	std_bpktl	Std. deviation from the mean of the packets in BD
16	min_fiat	Minimum amount of time between two packets in FD
17	mean_fiat	Mean amount of time between two packets in FD
18	max_fiat	Maximum amount of time between two packets in FD
19	std_fiat	Std. deviation of time between two packets in FD
20	min_biat	Minimum amount of time between two packets in BD
21	mean_biat	Mean amount of time between two packets in BD
22	max_biat	Maximum amount of time between two packets in BD
23	std_biat	Std. deviation of time between two packets sent in the BD
24	duration	Duration of the flow
25	min_active	Minimum amount of time that the flow was active
26	mean_active	Mean amount of time that the flow was active
27	max_active	Maximum amount of time that the flow was active
28	std_active	Std. deviation of time that the flow was active
29	min_idle	Minimum time a flow was idle before becoming active
30	mean_idle	Mean time a flow was idle
31	max_idle	Maximum time a flow was idle
32	std_idle	Std. deviation from the mean time a flow was idle
33	sflow_fpackets	Average number of packets in a sub flow in FD
34	sflow_fbytes	Average number of bytes in a sub flow in FD
35	sflow_bpackets	Average number of packets in a sub flow in BD
36	sflow_bbytes	Average number of packets in a sub flow in BD
37	fpsh_cnt	Number of times PSH flag was set in packets in FD
38	bpsh_cnt	Number of times PSH flag was set in packets in BD
39	furg_cnt	Number of times URG flag was set in packets in FD
40	burg_cnt	Number of times URG flag was set in packets in BD
41	total_fhlen	Total bytes used for headers in FD
42	total_bhlen	Total bytes used for headers in BD
43	DSCP	Differentiated Services Code Point
44	class	Class label

Chapter 5

The Virtual Network Function

This chapter introduces the concept of Network Function Virtualization and the Open Network Function Virtualization Platform (OPNFV). We present our proposed threat detection system as a Virtual Network Function running on OPNFV that will provide flexibility, agility, and cost reduction to monitor network traffic. Moreover, we propose a heuristic for virtual sensor placement. Finally, we propose a greedy based algorithm for service chaining.

5.1 The Network Function Virtualization

Network Function Virtualization (NFV) technology intends to offer software virtualized network services using costumer off the shelf (COTS) hardware in order to lower Operating Expenditures (OPEX) and Capital Expenditures (CAPEX) costs, and greatly reducing the time to the market (TTM) of innovations [124]. The key idea is to offer communication, processing, and storing services for big data [125]. Thus, Virtual Network Functions (VNF) are implemented in software running on different physical servers, usually on a cluster environment. Therefore, network services such as firewall and threat detection can be executed as a set of VNF allowing a bigger flexibility, scalability, and easier deployment when compared to traditional services. The main goal of the NFV technology is to optimize network services. This concept is complemented with the idea of Software Defined Networking (SDN) that provides a greater programmability for network management due to its global network view in the network controller. Specially, SDN acts in the control and in the implementation of packet forwarding and processing, while NFV acts in the provision of network services, such as firewall, Intrusion Detection System (IDS), Network Address Translation (NAT), or even higher layer services, such as Web servers, email servers, among others.

Instead of using expensive proprietary network equipment, a strong tendency is rising to provide services with open-source trusted platforms that integrate the

processing, storing, and communication of data. This means that the main concept of NFV is to decouple the Network Functions (NFs) from the physical infrastructure on which they run [126]. Trying to accelerate the implantation of Virtualized Network Functions, the Linux Foundation develops a collaborative project called Open source Platform for Network Functions Virtualization (OPNFV)[1]. The main idea behind OPNFV is to use open-source software to provide a platform compatible with the European Telecommunications Standards Institute (ETSI) standards.

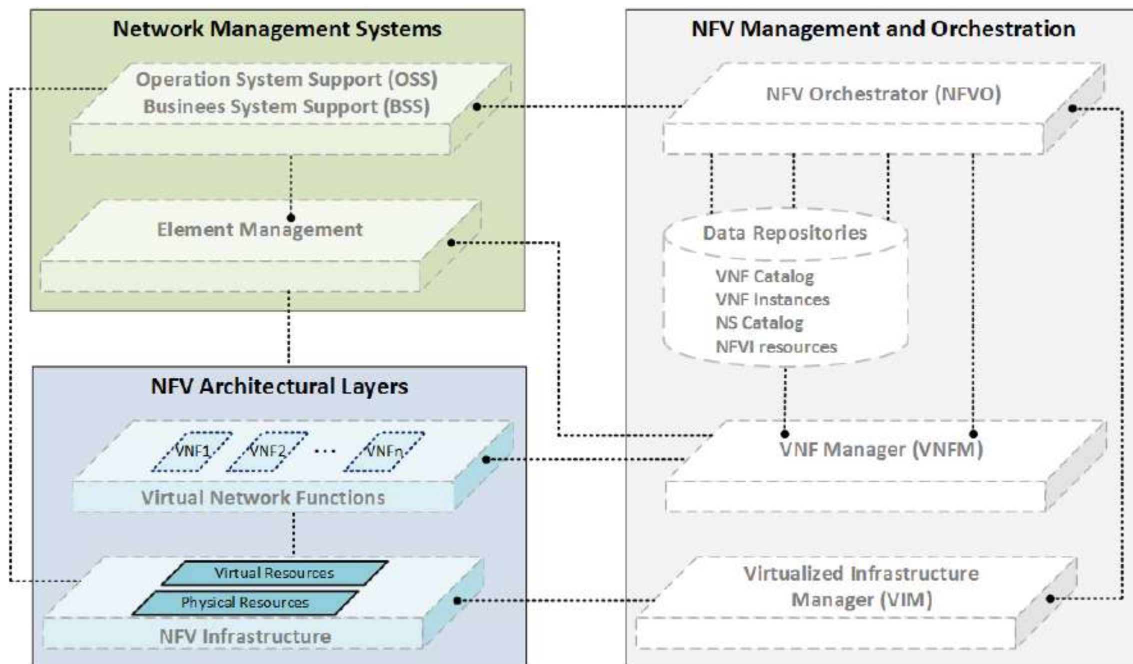


Figure 5.1: ETSI NFV MANO framework, composed by three main components, Network Management System, NFV Architectural Layers, and NFV Management and Orchestration.

Figure 5.1 presents the MANO framework. According with the ETSI definition, the MANO framework [127] consists of three functional blocks: The Virtualized Infrastructure manager (VIM), which is responsible for managing and controlling the physical and virtual infrastructure in a single domain. One NFV architecture could contain more than one VIM specialized to manage one or more certain types of NFVI resources such as compute, store, among others. Second is the Virtual Network Function Manager (VNFM) that is responsible for the management of the life cycle of one or more VNFs. Finally, the Network Function Virtualized Orchestration (NFVO) combines one or more functions to create the end-to-end service. The NFVO could be divided into resource orchestration and service orchestration. Resource orchestration ensures there are adequate compute, storage, and network resources available to provide a network service. This orchestrator can work with the VIM or directly with the NFVI, and it has the ability to coordinate, autho-

[1]The Open source Platform for Network Functions Virtualization https://www.opnfv.org/

rize, release, and engage NFVI resources independently of any specific VIM. It also provides governance of VNF instances sharing resources of the NFVI. On the other hand, the Service Orchestration is responsible for the creation of end-to-end services composed by different VNFs, also known as service chaining, and to manage the network topology for the different instances. Furthermore, the MANO contains data repositories that keep different types of information. The first is the Network Service (NS) catalog, which contains templates for services in terms of VNFs can be deployed and implemented, as well as their connectivity in the virtual links. This catalog can contain additional information such as NFV instances lifetime. Another catalog is the NFVI resources that contains the information about available/allocated resources in the NFVI.

5.1.1 The Open source Platform for Network Function Virtualization (OPNFV)

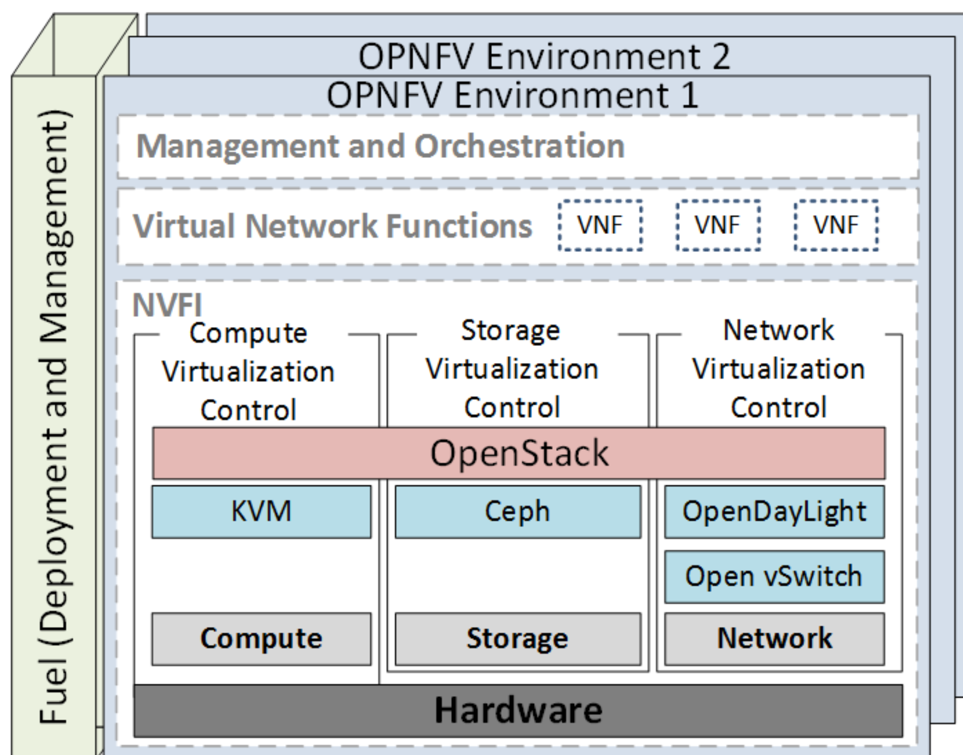


Figure 5.2: The OPNFV architecture: Network Virtual Function Infrastructure (NFVI), Virtual Network Functions (VNFs), and Management and Orchestration. The deployment and management of the OPNFV environments is coordinated by Fuel.

Figure 5.2 shows the architecture of the Open source Platform for Network Function Virtualization (OPNFV). As it can be seen, OPNFV implements only the NFV Architectural Layer component from the ETSI NFV MANO framework. Fuel deploys and manages the OPNFV environment. Three main modules compose this

environment, the Network Function Virtual Infrastructure (NFVI), the Virtual Network Functions (VNFs) and the Management and Orchestration (MANO). MANO provides the functionality required for the provisioning of VNFs, and related operations such as the configuration of VNFs and the infrastructure on which these functions run.

When the ETSI MANO concept is mapping into the OPNFV platform, the NFVI contains the compute, storage and network module. The compute module administrates the virtual machines through the KVM hypervisor. The Storage module uses the Ceph tool that is a distributed object store and file system. The network module uses the Software Defined Networking (SDN) paradigm through the OpenDayLight controller that manages the Open vSwitches virtual switches. Network services are deployed in middleboxes or network appliances called virtualized network functions (VNF). VNFs consist of one or more virtual machines that run specific network functions such as firewall, IDS, NAT, among others. The Virtual Network Functions (VNFs) can be combined together, chaining in deliver full-scale networking communication services [128]. We implemented our proposed Threat Detection System as a virtual network function (VNF). Finally, the Management and Orchestration layer provides the logic and functionality required for the provision of resources, configuring the VNFs and the infrastructure.

5.1.2 Threat-Detection Prototype Performance Evaluation

CATRACA [37] configuration is shown in Figure 5.3 as a virtual network function. To evaluate the performance of the prototype, we analyze latency requirements and speedup factor for real-time stream processing. Speedup factor is a number that measures the relative performance of two systems processing the same problem. We perform the experiments in the OPNFV Brahmaputra 3.0 environment and we use a module developed by the Sahara project to provide an Apache Spark cluster. Our OPNFV environment is composed by 96 GB of RAM, 700 TB of storage and 128 cores of Intel Xeon processors with clock frequency of 2.6 GHz. We calculate all the results with 95% of confidence interval.

CATRACA first uses a machine learning model trained offline. A decision tree algorithm is trained offline from dataset within a combination of normal traffic and threats stored in a historical database. We distribute sensor machines to capture traffic over the network. Sensor machines are simple devices that mirror captured traffic to the spark cloud, specially to Kafka. Kafka abstracts message stream into topics that act as buffers or queues, adjusting different production to consumption rates. To avoid latency overhead, sensor machine must be as much simple as possible. Finally, the offline model is loaded in the spark cloud, and the master VM will apply

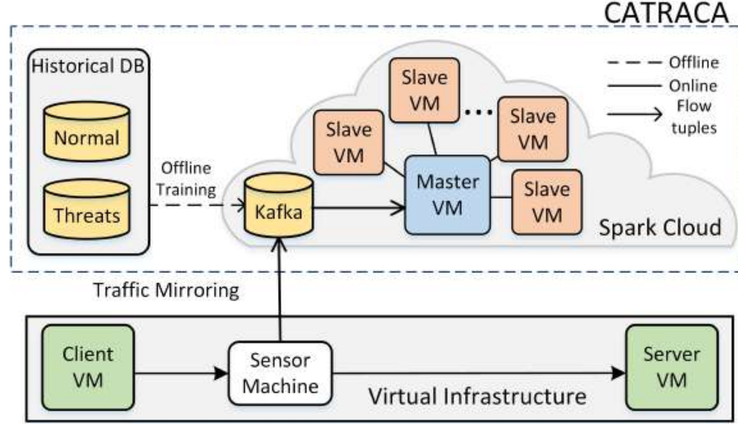


Figure 5.3: CATRACA configuration as a Virtual Network Function. A sensor machine is deployed to mirror the traffic. Mirrored traffic is send to CATRACA as a Virtual Network Funtion (VNF) that runs on the cloud.

the model to classify network traffic in real time.

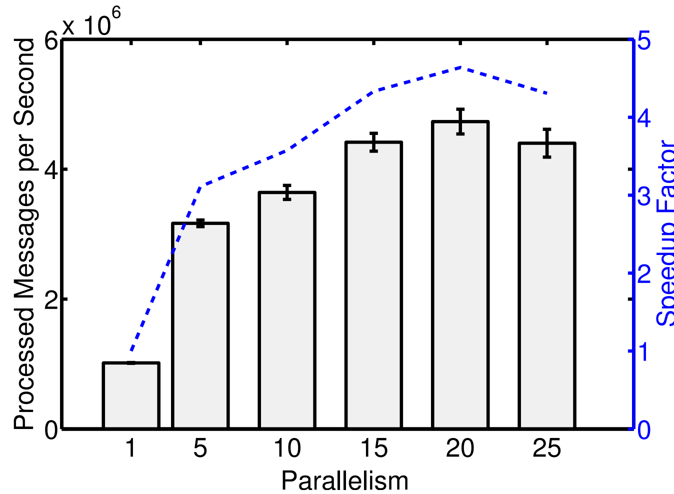


Figure 5.4: Messages processed per second (left axis) and speedup factor (right axis) in function of the task parallelism for the Apache Spark Streaming Engine.

The first experiment measures the performance of CATRACA in terms of processing throughput and latency. The dataset is fully injected into the system and also replicated as many times as necessary to obtain a huge amount of data to submit into our system. The experiment calculates the consumption of messages by our stream processing engine and its processing rate. We also vary the parallelism parameter, which represents the total number of cores available for the cluster to process samples in parallel. Each slave VM is configured with only one core, and we create as many Virtual Machines as possible. Figure 5.4 shows the results of the experiment. The left y axis shows the system throughput given by the number of messages processed per second, and the right y axis indicates the speedup factor. The speedup factor is calculated as follows: $S_{latency} = \frac{La1}{La2}$; where $La1$ is the latency

of the system when parallelism is equal to one, and $La2$ is the latency of the system with the variation of the parallelism parameter. In other words, it represents the decrease factor in latency or the speed up factor.

The proposed Virtual Network Function is able to improve the processing capacity up to twenty cores in parallel. The system shows an improvement for latency as well as for throughput. Considering throughput, the system is able to handle more than five million of messages per second. Moreover, the speed factor reaches around 4.65 for a parallelism of twenty cores. This value indicates that the system can parallelize and improve the speed of almost five times with twenty virtual machines running one core each.

The second experiment aims to show the operation efficiency of the implemented function under live migration. The live migration offers a great flexibility for the user and it is possible thanks to the virtualization, achieved through the OPNFV platform. In our proposed threat-detection virtualized network function, live migration provides several advantages. A security advantage is the possibility to place and rearrange dynamically sensor machines to better protect the network and reduce the threat detection time. We can place sensor machines where more attack traffic enters or near a sensitive server to be protected. A general advantage concerning the processing cluster is the ability to migrate machines, allowing a smart distribution among the physical servers and enabling the optimization of the number of running servers, avoiding the waste of resources.

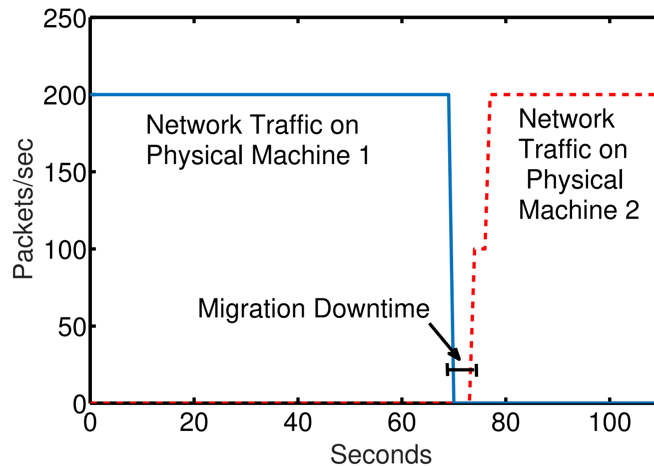


Figure 5.5: A Virtual Machine migration from the Physical Machine 1 to Physical Machine 2. The constant flow rate applied to the virtual machine at Physical Machine 1, after 60 seconds approximately of migration, goes to Physical Machine 2.

Figure 5.5 shows the behavior of a network flow under live migration. In this experiment we send a constant-rate flow of 200 packets per second from one virtual machine to another. Both virtual machines are hosted in the same physical machine.

Then, approximately at 60 seconds, the migration process is started, in order to migrate the virtual machine that receives the flow to another physical server. The Figure 5.5 shows the low migration downtime, making the flow unaffected under the migration. therefore, the migration feature allows our threat detection application to set monitoring sensors as close to the client as possible, avoiding latency problems.

5.2 Modeling and Optimization Strategy for VNF Sensor Location

Our system is able to distribute sensors along the whole network. Nevertheless, because of cost or performance reasons, we can establish a reduced number of sensors instead of placing sensors in every switch. Hence, then sensor placement results into an optimization problem. This section models formally the problem of the sensor network placement. A heuristic is proposed to minimize the number of sensors and maximize the network coverage reached by each sensor. We define network coverage as the portion of traffic that a sensor is able to analyze. We assume that each node owns enough processing power to analyze all incoming network traffic, as it is for example, in a datacenter.

Let G be a finite connected graph $G = (V, E)$, where V is the non-empty set of all available nodes in the network and E is the set of non-directed edges of G. Let $v \in V$ be a vertex, and $s \in S \subseteq V$, where S is the subset of nodes which are implemented as a threat detection sensor. We use x^v as the variable that determines if vertex v holds a sensor s

$$x^v = \begin{cases} 1, & \text{if } v \in S \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

Each vertex sends a traffic unit to each other vertices. Therefore, the total amount of traffic T in the network equals $T = \frac{|V|(|V|-1)}{2}$. The traffic from the source i to destination j uses a single path, and we represent whether vertex v is in the path of i, j traffic by $t^v_{ij} = 1$, otherwise $t^v_{ij} = 0$. Thus, the total amount of traffic passing through vertex v is $t^v = \sum_{i \neq j} t^v_{ij}$. We also define an auxiliary variable $y_{i,j}$ to determine whether there is a sensor in the path from the source i to destination j. The variable $y_{i,j}$ is given by

$$y_{i,j} = \begin{cases} 1, & \exists\, v \in V \mid x^v \wedge t^v_{i,j} \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

Thus, the total amount of traffic analyzed by the sensors is

$$T_x = \sum_{i \neq j \in V} y_{ij}, \tag{5.3}$$

The objective function $F(x)$ represents the global cost to minimize, composed by two objective functions: the number of sensors in the network and the total network traffic coverage that is analyzed by all sensors. Then, we want to minimize the global cost

$$\min F(x) = F_{sensor}(x) - F_{traf}(x), \tag{5.4}$$

where $F_{sensor}(x)$ is the relation between the sensor nodes and the total of vertices in the network, expressed by

$$F_{sensor}(x) = \frac{1}{|V|} \sum_{v \in V} x^v, \tag{5.5}$$

and $F_{traf}(x)$ is the traffic percentage in relation to the total amount of network traffic that each sensor node analyzes, given by

$$F_{traf}(x) = \frac{T_x}{T} = \frac{1}{T} \sum_{i \neq j \in V} y_{ij} \tag{5.6}$$

such that $0 \leq F_{sensor}(x) \leq 1$ e $0 < F_{traf}(x) \leq 1$. Besides, $\sum_{v \in V} x^v \leq |V|$, and $x^v \in [0, 1]$. This problem can be reduced to a Set Covering Problem (SCP). A problem that cannot be resolved in deterministic polynomial time, also known as NP-hard problem. To solve this problem, we use a greedy algorithm, which sorts the vertex list according to t^v, the amount of traffic that passes through the vertices. Thus, the algorithm chooses first central nodes that concentrate the largest amount of the traffic. We use as metric the amount of traffic $H(x) = T_x/T$ covered by the current sensors. Hence, after the selection of a sensor node, it is verified if the solution $H(x)$ reaches a target amount of traffic and stops. Otherwise, the algorithm selects another node. The process is repeated until the sensor set covers the target amount of traffic, or when it is not possible to add more sensors. Greedy algorithms make locally optimal choices that eventually reach a global optimal. Nevertheless, the computational cost execution is low compared to other solutions.

We also used the Simulated Annealing (SA) optimization method, which guarantees convergence to a global minimal in an undetermined time. Every iteration selects a number of sensors and their positions at random and generates a new candidate solution that could be accepted in case it has a lower objective function $F(x)$ than the previous iteration. If the solution is worse than the previous one, the SA accepts the new solutions by a decreasing probability according to iteration. Thus,

the solution may be accepted even if the objective function is higher than the previous one. This behavior is necessary to avoid the solution convergence to a local minimal. The perturbation used to select the number of sensors and the positions follow a Cauchy distribution. Hence, we use the greedy algorithm to obtain the fastest result and the Simulated Annealing to evaluate how far from the minimal the result is.

5.2.1 Optimal VNF Sensor Placement

To evaluate the proposed heuristic we use a real topology from *topology zoo* [2]. The analyzed topology is the Brazilian Internet backbone network, *Rede Nacional de Ensino e Pesquisa* (RNP), that has 31 vertex with 34 edges distributed geographically in the Brazilian states. The real topology can be seen in Figure 5.6.

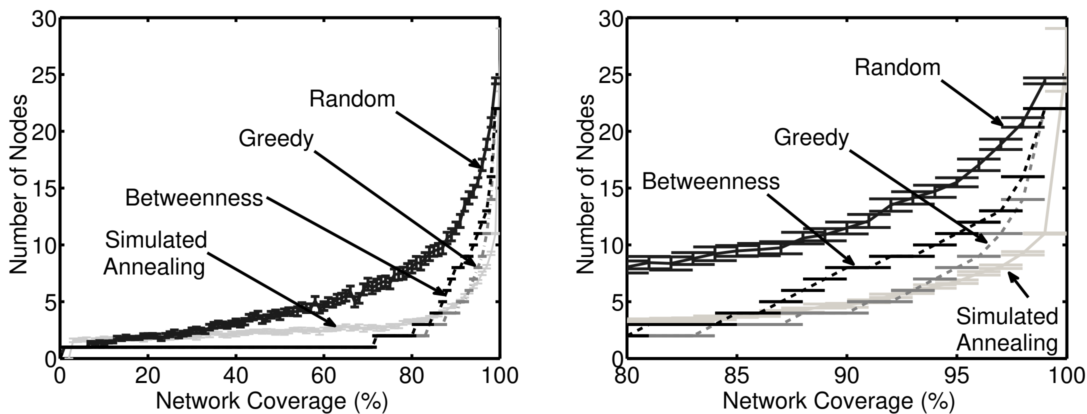
Figure 5.6: The Brazilian *Rede Nacional de Ensino e Pesquisa* (RNP) real topology, with 31 vertex and 34 edges.

We have evaluated our metric with two meta-heuristic solutions, greedy and simulated annealing, and with a random choice. In addition, we implemented the node selection by the betweenness centrality heuristic. Betweenness centrality is shown in Equation 5.7, where σ_{ij} is the total number of shortest paths from node i to node j and $\sigma_{ij}(v)$ is the number of those paths that pass through v. The betweenness centrality expresses the influence that a specific node could have on other nodes in the network [129]. The betweenness centrality considers all the nodes and paths in the network, while our proposal is relative to each node.

$$g(v) = \sum_{i \neq v \neq j} \frac{\sigma_{ij}(v)}{\sigma_{ij}} \tag{5.7}$$

[2] www.topology-zoo.org, Accessed April 2018.

Figure 5.7a shows in percentage the covered traffic in relation with the number of VNF sensors used. The system determines the best position that each node should be located warranting the maximal network coverage. With low network coverage the behavior of all approaches is similar. The random choice shows the worst behavior followed by the simulated annealing. Then, the simulated annealing improved its results over the random choice. The betweenness and the greedy proposal present a good result until approximately 80%. Figure 5.7b shows in higher detail the results. The random choice is still being the worst approach followed by the betweenness. Both approaches with our heuristic show the best efficiency for high values of network coverage, minimizing the number of sensors nodes used. The simulated annealing presents better behavior despite of its high computational cost. With 95% of network coverage the random solution uses 15 nodes and the greedy and simulated annealing reduce it by two times using 7 nodes. Moreover, with 99%, the simulated annealing improves the result by more than two times when compared with the random choice, placing 11 nodes instead of 21 nodes required by the random choice.



(a) Heuristic evaluation in a topology of 31 nodes.

(b) Network coverage vs. Number of sensors in a zoomed region.

Figure 5.7: Efficiency results of different placement methods in function of the number of sensor nodes required to cover all network traffic.

5.3 The Virtual Network Function Chaining Problem

Service chaining simply consists of a set of network services interconnected through the network infrastructure to support an application requested for the costumer. Traditionally, Service Function Chaining (SFC) was built in the early years of high-performance computing being rigid and static installed at fixed locations in

the core or at the edge of the carrier network [130]. The SFC is enhanced with the advent of NFV that enables operators to configure network services dynamically in software without having to make changes to the network at the hardware level. Therefore, virtualized NFs (VNFs) can be placed when and where needed. This implies an optimization problem that uses VNFs or services as a graph to address the requirement for a better utilization of resources, for latency decrease and for network optimization [32]. Typically, network flows go through several network functions as shown in Figure 5.8. When a NF or a set of NFs are specified the flows traverse these NFs in a specific order so that the required functions are applied to the flows. Usually, the NFs demand certain dependency among them that should be chaining to the traffic in a network in a specific order. Depending on the way each network function is set in the chain, it impacts in network traffic, application performance, and latency.

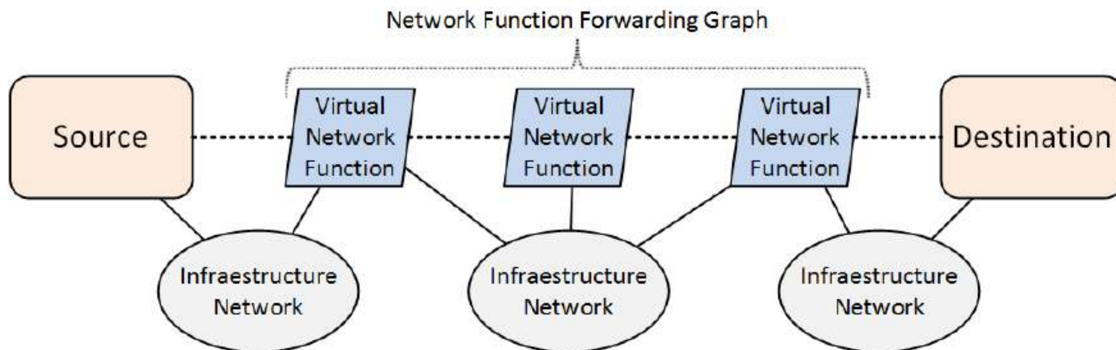


Figure 5.8: Example of Network Function forwarding graph. Three Virtualized Network Functions (VNF) are chained from the source to the destination to establish a service. The virtual network functions are executed over the physical network infrastructure.

The VNF chaining problem consists, thus, of two sub-problems. The first subproblem is the placement problem, in which the VNF instances are allocated onto physical nodes. This problem consists into finding a physical node that has enough resources to host the VNF, serving it with the requested resources. The second sub-problem consists into a routing challenge, because mapping a set of VNFs over a physical topology should consider the iteration among all VNFs. The routing problem should ensure that the traffic between VNFs would always suffer a limited delay, and the network path presents enough bandwidth. If any of these constraints are not satisfied by the chaining scheme, the VNF request may not be accepted. Therefore, deciding for accepting VNF requests is also part of the VNF chaining problem.

5.3.1 The Proposed VNF Chaining Scheme

Our proposal considers a scenario in which the requests for a new Virtual Network Function arrive to a network manager, who has to allocate them into the available nodes. We consider as a request a sorted list of VNFs that describe the order in which traffic has to be processed. Therefore, the allocation of the request on the network has to consider the order of the VNFs as well as the source and the destination of the traffic handled by the set of VNFs in the request. We also consider that, when allocating a VNF over a physical node, the physical node has to provide enough resources to answer the needs of all hosted VNFs. Our proposed scheme is composed of two main phases. The first phase is to estimate the resources available on the physical nodes and the resources requested by the VNFs. The second phase is to run a greedy algorithm that takes as input the VNF requests as they arrive, and then it places each VNF on a physical node that has enough resources. Our greedy algorithm considers four different heuristics to place the VNFs on the network.

Estimating the available resources on physical and virtual nodes is challenging because there are three main resources which should be considered: CPU, memory, and network. In order to summarize all resources into one single variable, we consider the `Volume` metric introduced by Wood *et al.* [63]. We consider that the `volume` of a physical server is 1, and the `volume` of each VNF is given by

$$Volume_{VNF} = \frac{1}{1-cpu} * \frac{1}{1-mem} * \frac{1}{1-net}, \tag{5.8}$$

where cpu stands for the normalized CPU usage of the VNF, mem for memory, and net for network. Thus, for each VNF the `volume` metric is the ratio of the resources on physical node that the VNF is requesting. The VNF `volume` ranges from 0 to 1, where 1 means that a VNF is requesting an entirely available physical node to be installed.

Following, on the second phase, we run a greedy algorithm that allocates a VNF request as it arrives. Our algorithm adopts one of the four heuristics:

- **minimum latency**, in which the algorithm chooses the node that introduces a minimum delay to the path, in comparison to the previous selected nodes to host the other VNFs, or the source of the traffic;
- **maximum usage of resources**, in which the algorithm chooses the node that has the biggest amount of available resources to host a VNF, without considering the routing constrains between the already placed VNFs;
- **most central nodes**, in which the algorithm chooses to place the VNF into the most central node, i.e. the node that presents the greatest betweenness-centrality value, and has enough resources to host the VNF;

- **weighted latency and resource**, in which the probability of choosing each a node for hosting a VNF is weighted based on the latency that it introduces to the path and the available resources that it has. The weight of the node i is given by

$$w_i = \left(1 - \frac{lat_i}{max_{j \in N}(lat_j)}\right) \times \left(\frac{rec_i}{max_{j \in N}(rec_j)}\right),$$

  where lat_i stand for the latency introduced by node i, rec_i is the available resources in node i, and N is the set of all nodes in the network. The greedy algorithm searches for hosting VNFs on the nodes that have the biggest w_i value first.

Our proposal works as follows. First, the network manager receives a sorted list with the requested VNFs, the source and the destination of the traffic, and the requested resources of each VNF. Then, our algorithm selects the first VNF on the request and search for a node in which the requested resources meet the available resources on the physical node. To verify if the physical node has enough resources, the algorithm compares the VNF `volume` with the available `volume` of the physical node. If the available `volume` is greater than the requested, the VNF is installed on this candidate physical node. Otherwise, the algorithm selects the next physical node until finding an available node. If there is no available physical node that meets the requested VNF `volume`, the VNF request is entirely rejected and no VNF is allocated. After mapping all VNFs over the physical nodes, the VNFs are installed and the `volume` of each physical node that receives a VNF is decremented by the `volume` of the VNF that it hosts. It is worth noting that a VNF request should be entirely accepted or rejected. If the algorithm realizes that there is not enough resource in any node in the network to complete the VNF request allocation, the request is completely rejected and no node is allocated on the network. We adopt the all or nothing approach, because a partially allocated VNF request does not implement all packet-processing functions that it supposed to deploy, thus it is not a feasible solution.

5.4 The Evaluation of the Proposal

We evaluate the proposed greedy algorithm through simulation. We implemented a simulator[3], written in Python language, in which the VNF requests arrive at each simulation step. At a simulation step, the proposed scheme evaluates the used

[3] available at `https://github.com/tinchoa/VNFsimulator`, Accessed April 2018.

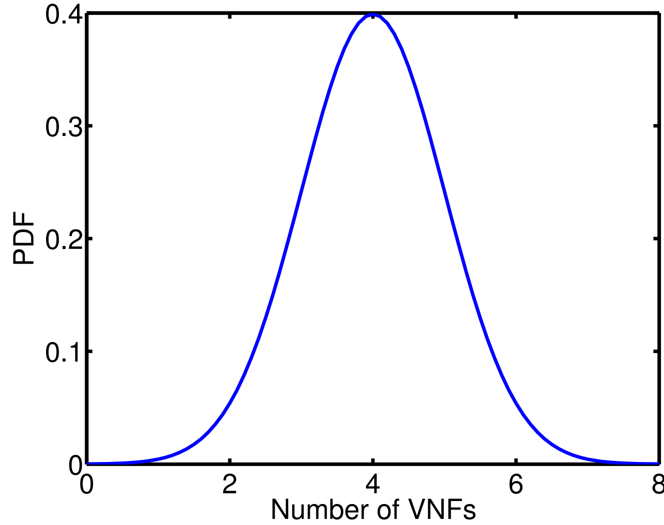


Figure 5.9: Probability density function of the number of VNFs in a request. The modeled number of VNFs in each request follows a normal distribution, with mean equals to 4, and standard deviation equals to 1 ($\mu = 4$ and $\sigma = 1$).

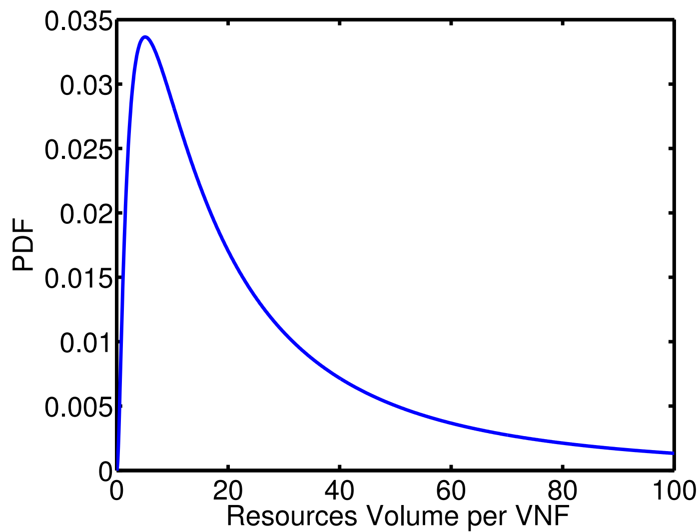


Figure 5.10: Probability density function of the `volume` of resources used by each VNF. The modeled resource consumption of each VNF follows a lognormal distribution, with mean equals to 3, and standard deviation equals to 1.17 ($\mu = 3$ and $\sigma = 1.17$). The lognormal distribution is truncated at 100, which represents the maximum `volume` usage ($volume = 1$).

resources of each physical node on the network topology and generates the available *volume* metric for each physical node. Then, our scheme gets the next request and allocates the VNFs on the network according to one of the four proposed heuristics.

We establish the proposal evaluation in two steps. The first step is to simulate the costumer Virtual Network Functions (VNFs) requests. The requests are generated based on a normal distribution with $\mu = 4$ and $\sigma = 1$, as it is shown in Figure 5.9. In this way, the customer requests are generated randomly associating different

number of VNF for each request. We model the number of VNFs in each request based on the studies carried out by Sekar *et al.* [29]. We consider that a Virtual Machine (VM) deploys each VNF. Then, we simulate the resource consumption of each VNF. We model the resource consumption based in a lognormal distribution, with $\mu = 3$ and $\sigma = 1.17$, as it is shown in Figure 5.10. The resource consumption of each VNF is modeled as a truncated lognormal distribution because it should reflect the behavior of the middleboxes, in which it usually uses a small amount of resources. The distribution is truncated at 100, because it is the maximum *volume* that a VNF can assume ($volume_{VNF} = 1$). As result, we obtain the costumer VNF request with different resource *volume* and a selected order of chaining. We highlight that the resources of all VNFs over a single physical host is never higher than 100%. An example of a VNF costumer request is $[VNF_1 = 15\%, VNF_2 = 26\%, VNF_3 = 45\%]; src, dst$, where the number of VNF are randomly chosen, and the *src* and *dst* are the source and destination of each chaining request. The source and the destination are uniformly chosen on the network topology. Our model does not consider that VNFs quit the network after being allocated.

The second step of the proposal evaluation asserts the optimization heuristics. Our experiments evaluate the placement heuristics on the RNP (Rede Nacional de Pesquisa) topology. Using a greedy algorithm, we place the VNF in different nodes and we evaluate the amount of VNF requested for each heuristic. We consider only the propagation delay between the nodes to estimate the latency between the nodes. The propagation delay is estimated according to the distance between nodes. We consider the propagation speed of 2×10^8 m/s, which is commonly used in other works [131]. The distance between each node is calculated based on the geographic location of each node.

The results in Figure 5.11 show that the maximum resource allocation heuristic is the one that accepts more requests around 53% more than the betweenness-centrality heuristic. The betweenness-centrality heuristic is the simplest to calculate, as it only depends on the topology characteristics. Nevertheless, it is the one that rejects the greatest number of requests. In addition, the latency heuristic presents a better performance when compared with the betweenness-centrality, however, this heuristic shows the worst complexity when executed. It is worth noting that, although the maximum resource allocation, heuristic optimizes the acceptation rate of VNFs on the network; it does not consider the routing constraints between VNFs. In this way, it increases the delay introduced by the deployment of network functions as VNFs because the packets may pass through distant nodes in order to follow the entire packet-processing path.

We also compare the dispersion of the latency distribution of the allocated VNFs for each heuristic. As shown in Figure 5.12a, the minimum latency heuristics in-

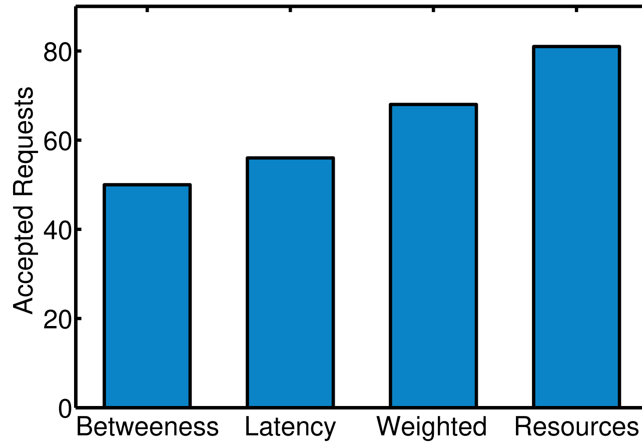
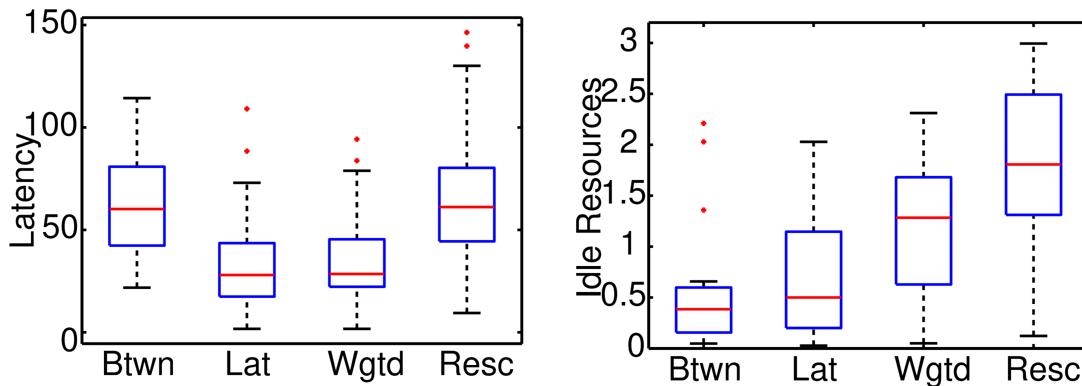


Figure 5.11: Number of accepted requests for each proposed heuristics. The maximum resource allocation heuristic is the one that accepts more requests. The betweenness-centrality heuristic is the simplest to calculate, as it only depends on the topology characteristics, but it is the one that rejects the greatest number of requests.



(a) Dispersion of the latency distribution into the allocated VNFs.

(b) Dispersion of the remaining idle resource distribution after allocating all VNFs.

Figure 5.12: `Btw` stands for the betweeness-centrality; `Lat`, for latency; `Wgtd`, for weighted latency and resource; `Resc`, for maximum resource. a) The minimum latency heuristics introduces the lower average delay on the packet-processing path. The maximum resource usage heuristic is the one that presents the greatest dispersion into the latency distribution thanks to ignoring the latency concerns when placing the VNFs. b) The Maximum Resource heuristic presents the most distributed remaining resources.

troduces the lowest average delay on the packet-processing path. The maximum resource usage heuristic is the one that presents the greatest dispersion into the latency distribution thanks of ignoring the latency concerns when placing the VNFs. This result shows that the Latency heuristic reduces 52% of the average delay when compared with the betweeness-centrality heuristic. Moreover, the latency heuristic also achieves the greatest number of accepted VNF requests, which have the mini-

mum latency, even when compared with maximum resource allocation that achieves to allocate more requests than all others. Figure 5.12a also reveals that the latency is limited in all heuristics and, even in the highest delay scenario, it is still limited to 150 ms. Figure 5.12b shows the remaining resources after all VNF allocation. Although the maximum resource heuristic instantiates more VNFs, it presents the biggest amount of idle resources. Nevertheless, it is also the most distributed idle resource pattern, which implies a load distribution between all physical nodes.

Comparing Figures 5.11 and 5.12a, we emphasize that the greater dispersion of the latency achieved by the maximum resource allocation heuristic is a reflect of the greater number of accepted requests, when compared with the minimum latency heuristic. It is worth noting that all heuristics are compliant with the resource constraints. Therefore, choosing among the four heuristics, when designing a NFV environment, should consider the goals of the network manager. In case the main goal is to maximize the number of accepted VNFs, the results show that we should select the maximum resource allocation heuristic, in order to present good results of latency between nodes. Nevertheless, if the main goal is to achieve the maximum performance of VNFs, the minimum latency heuristics is the best choice. Intermediary solutions are the weighted latency and resource solution, which keep the bounded latency, and it increases up to 22% the acceptance rate when compared to the Latency heuristic.

In this chapter, we analyze our Virtual Network Function performance. First, we describe the Network Function Virtualization and the Open Network Function Virtualization Platform (OPNFV) and the benefits it brings to CATRACA. Then we propose a heuristic for virtual sensor placement maximizing the traffic analyzed with the minimum number of sensors in the network. Finally, we propose a greedy based algorithm for service chaining. We evaluate four heuristics. Our simulation and results show that using a heuristic for placing VNFs on nodes with the biggest amount of available resources increases the acceptation rate of VNF requests by 53%.

Chapter 6

Conclusion

This work proposed a fast and efficient network monitoring and threat detection system. We used a combination of machine learning and stream processing for real time threat detection. A new generation of tool is needed in order to real-time monitor and secure computer networks. We described and compared the three-major open source distributed stream processing systems: Apache Storm, Apache Flink, and Apache Spark Streaming. We performed throughput analysis, allocating more processing cores to achieve higher processing rates, Apache Storm was able to process up to 15 million samples per minute. Moreover, we performed fault tolerance test to compare these three most popular open-source Distribute Stream Processors (DSP). In this case, we showed that Spark streaming, using micro-batch processing model, can recover the failure without losing any messages. Spark Streaming stores the full processing state of the micro-batches and distributes the interrupted processing homogeneously among other worker nodes.

In order to increase the analysis speed and improve the efficiency of big data analysis, it is mandatory to implement pre-processing methods. This work presented and compared different methods for dimensionality reduction and feature selection. Furthermore, we proposed a new fast-unsupervised algorithm for pre-process stream data. The algorithm includes feature selection and data normalization. Our feature selection algorithm calculates the correlation of the features in a network traffic data and selects the best features in an unsupervised way. We selected the features with the higher absolute correlation in comparison with the others methods. This procedure tends to gather the features with most of the information of the dataset in a new set of reduced features. The reduced new set of features is used, thus, to train the machine learning methods that classify and characterize the network traffic. Our algorithm showed a performance up to 10 times faster than literature algorithms for feature selection. We evaluated the feature selection algorithm in two different datasets achieving a good performance. Moreover, our pre-processing algorithm is able to detect concept-drift in stream data. We showed that our nor-

malizer adapts the data to a normal distribution reducing the error of machine learning classifiers. To evaluate the proposed algorithms, we implemented eight machine learning algorithms, decision tree, neural networks, k-nearest neighbors, support vector machine with linear and Radial Basis Function (RBF) kernel, Gaussian Naive Bayes and Stochastic Gradient Descendant. We evaluated machine learning algorithm performance considering six metrics, accuracy, precision, sensitivity, F-1 score, classification and training time.

We created two datasets publicly available. First a synthetic security dataset obtaining real network traffic along with network threats composed of 24 features. The GTA/UFRJ dataset contains more than 16 different types of attacks, as well as normal network traffic. The second dataset, NetOp, is composed by more than ten days of real traffic from a real telecommunication operator network located in the city of Rio de Janeiro, Brazil. The dataset represents the use of the fixed-line access service of 373 home users. The analysis of the data allows identifying that the main services accessed are those of DNS and web services. We use this information to create more than 5 TB of data combined in 45 flows features. Each flow is previously classified as normal traffic or alert by using an Intrusion Detection System.

Network Function Virtualization (NFV) is a promising technique that enables to decouple the network function from its physical realization by virtualizing the network equipment. Thus, network functions are deployed within virtual environment and, thus, called Virtual Network Functions (VNF). Therefore, we deploy our threat detection system as a virtualized network function. The threat detection function is implemented in the Open-source Platform for Network Function Virtualization (OPNFV) and shows high throughput and a low latency and live migration features.

Combining stream processing, machine learning and feature selection we the CATRACA tool. The proposed tool is executed in an OPNFV environment allowing the system to perform migration as close as possible to the source attack. Moreover, the tool combines batch and stream processing in a big data architecture that allows to perform threat analysis on incoming traffic in real time on a historical database. Our tool displays the knowledge extracted from the enriched data through a graphical user interface for visualizing different analyzes and the geographical location of the source and destination of the threats in real time. CATRACA code is publicly available.

Finally, we proposed a method for the strategic location of traffic capture sensors. For this, a mathematical modeling was developed, obtaining a heuristic that considers the minimum number of sensors reached the maximum coverage of the network. The evaluation of the heuristic was analyzed in two different real topologies. The results show that with a high network coverage the proposed system has a

great gain in relation to random choice. In addition, we modeled and discussed the VNF sensor placement as an NP-hard problem. Our heuristic reduces the number of sensor and maximizes the network coverage.

We proposed a VNF chaining scheme, in which a greedy algorithm places the VNFs on the network according to four different criteria. Our simulation and results showed that using a heuristic for placing VNFs on nodes with the biggest amount of available resources increases the acceptation rate of VNF requests by 53%. Moreover, we also showed that using a heuristic for introducing minimum delay on the path, we are able to reduce the average packet-processing delay by 52%.

6.1 Future Work

New methods of anomaly detection and traffic classification need to be experimented in CATRACA to avoid the use of the batch layer. Algorithms such as deep learning or Hoeffding Trees should be implemented in the tool. Due to the distributed nature of our monitoring and threat detection tool, we need to implement an algorithm for event correlation. A future work foresees to detect intrusion symptoms by collecting diverse information at several architectural levels, from raw packet until system logs, using distributed security probes, as well as performing complex event analysis based on a complex event processing engine.

Bibliography

[1] HU, P., LI, H., FU, H., et al. "Dynamic defense strategy against advanced persistent threat with insiders". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 747–755. IEEE, 4 2015. ISBN: 978-1-4799-8381-0. doi: 10.1109/INFOCOM.2015.7218444.

[2] PAXSON, V. "Bro: a system for detecting network intruders in real-time", *Computer Networks*, v. 31, n. 23-24, pp. 2435–2463, 12 1999. ISSN: 13891286. doi: 10.1016/S1389-1286(99)00112-7.

[3] BAR, A., FINAMORE, A., CASAS, P., et al. "Large-scale network traffic monitoring with DBStream, a system for rolling big data analysis". In: *2014 IEEE International Conference on Big Data (Big Data)*, pp. 165–170. IEEE, 10 2014. ISBN: 978-1-4799-5666-1. doi: 10.1109/BigData.2014.7004227.

[4] STONEBRAKER, M., ÇETINTEMEL, U., ZDONIK, S. "The 8 requirements of real-time stream processing", *ACM SIGMOD Record*, v. 34, n. 4, pp. 42–47, 12 2005. ISSN: 01635808. doi: 10.1145/1107499.1107504.

[5] CLAY, P. "A modern threat response framework", *Network Security*, v. 2015, n. 4, pp. 5–10, 2015.

[6] DOS SANTOS, L. A. F., CAMPIOLO, R., MONTEVERDE, W. A., et al. "Abordagem autonômica para mitigar ciberataques em LANs", *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2016*, 2016.

[7] NASSERALA, A., MORAES, I. M. "Analyzing the producer-consumer collusion attack in Content-Centric Networks". In: *2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pp. 849–852. IEEE, 2016.

[8] KOTENKO, I., CHECHULIN, A. "Attack modeling and security evaluation in SIEM systems", *International Transactions on Systems Science and Applications*, v. 8, pp. 129–147, 2012.

[9] PONEMON, I., IBM. “2015 Cost of Data Breach Study: Global Analysis”. 5 2015. Available in: <www.ibm.com/security/data-breach/>.

[10] CHANDRASEKAR, K., CLEARY, G., COX, O., et al. “Internet Security Threat Report-Symantec Corporation,V22”. Accessed April 2018, 2017. Available in: <https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf>.

[11] ARMOR. “The Black Market Report 2018-03”. Accessed April 2018, 2018. Available in: <https://www.armor.com/app/uploads/2018/03/2018-Q1-Reports-BlackMarket-DIGITAL.pdf>.

[12] ICA, I. C. A. *Assessing Russian Activities and Intentions in Recent US Elections 2017-01D*. Technical report, Office of the director of national Intelligence, 2017. Available in: <https://www.dni.gov/files/documents/ICA_2017_01.pdf>.

[13] MAYHEW, M., ATIGHETCHI, M., ADLER, A., et al. “Use of machine learning in big data analytics for insider threat detection”. In: *IEEE Military Communications Conference, MILCOM*, pp. 915–922, 10 2015.

[14] JIANG, W., RAVI, V. T., AGRAWAL, G. “A Map-Reduce system with an alternate API for multi-core environments”. In: *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 84–93. IEEE Computer Society, 2010.

[15] KALA KARUN, A., CHITHARANJAN, K. “A review on Hadoop—HDFS infrastructure extensions”. In: *IEEE Conference on Information & Communication Technologies (ICT)*, pp. 132–137. IEEE, 2013.

[16] GAMA, J., RODRIGUES, P. P. “Data stream processing”. In: *Learning from Data Streams*, Springer, pp. 25–39, 2007.

[17] CANINI, M., FAY, D., MILLER, D. J., et al. “Per flow packet sampling for high-speed network monitoring”. In: *2009 First International Communication Systems and Networks and Workshops*, pp. 1–10, 1 2009. doi: 10.1109/COMSNETS.2009.4808888.

[18] DUFFIELD, N. “Sampling for passive internet measurement: A review”, *Statistical Science*, pp. 472–498, 2004.

[19] CHENG, G., GONG, J., TANG, Y. “A hybrid sampling approach for network flow monitoring”. In: *Workshop on End-to-End Monitoring Techniques and Services (E2EMon’07)*, pp. 1–7. IEEE, 2007.

[20] BATISTA, D. M., GOLDMAN, A., HIRATA, R., et al. "InterSCity: Addressing Future Internet research challenges for Smart Cities". In: *2016 7th International Conference on the Network of the Future (NOF)*, pp. 1–6. IEEE, 11 2016. ISBN: 978-1-5090-4671-3. doi: 10.1109/NOF.2016.7810114.

[21] CRUZ, P., COUTO, R. S., COSTA, L. H. M. "An algorithm for sink positioning in bus-assisted smart city sensing", *Future Generation Computer Systems*, 10 2017. ISSN: 0167739X. doi: 10.1016/j.future.2017.09.018.

[22] ANDREONI LOPEZ, M., FERRAZANI MATTOS, D., DUARTE, O. C. M. B. "An elastic intrusion detection system for software networks", *Annales des Telecommunications/Annals of Telecommunications*, v. 71, n. 11-12, pp. 595–605, 12 2016. ISSN: 0003-4347. doi: 10.1007/s12243-016-0506-y.

[23] DAB, B., FAJJARI, I., AITSAADI, N., et al. "VNR-GA: Elastic virtual network reconfiguration algorithm based on Genetic metaheuristic". In: *IEEE GLOBECOM*, pp. 2300–2306, 12 2013.

[24] GUIMARÃES, P. H. V., MURILLO P., A. F., ANDREONI LOPEZ, M. E., et al. "Comunicação em Redes Elétricas Inteligentes: eficiência, confiabilidade, segurança e escalabilidade". In: *SBRC 2013 - Minicursos*, pp. 101–164, 5 2013.

[25] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., et al. "Storm@Twitter". In: *ACM SIGMOD International Conference on Management of Data*, pp. 147–156. ACM, 2014.

[26] FRANKLIN, M. "The Berkeley Data Analytics Stack: Present and future". In: *IEEE International Conference on Big Data*, pp. 2–3. IEEE, 2013.

[27] CARBONE, P., FÓRA, G., EWEN, S., et al. "Lightweight Asynchronous Snapshots for Distributed Dataflows", *Computing Research Repository (CoRR)*, v. abs/1506.0, 2015.

[28] SHERRY, J., HASAN, S., SCOTT, C., et al. "Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service", *SIGCOMM Comput. Commun. Rev.*, v. 42, n. 4, pp. 13–24, 8 2012. ISSN: 0146-4833.

[29] SEKAR, V., EGI, N., RATNASAMY, S., et al. "Design and Implementation of a Consolidated Middlebox Architecture". In: *9th Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 323–336, San Jose, CA, 2012. USENIX. ISBN: 978-931971-92-8.

[30] JEON, H., LEE, B. "Network service chaining challenges for VNF outsourcing in network function virtualization". In: *International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 819–821, 10 2015.

[31] BARI, M. F., CHOWDHURY, S. R., AHMED, R., et al. "On orchestrating virtual network functions". In: *11th International Conference on Network and Service Management (CNSM)*, pp. 50–56, 11 2015.

[32] ADDIS, B., BELABED, D., BOUET, M., et al. "Virtual network functions placement and routing optimization". In: *IEEE 4th International Conference on Cloud Networking (CloudNet)*, pp. 171–177, 10 2015.

[33] MEHRAGHDAM, S., KELLER, M., KARL, H. "Specifying and Placing Chains of Virtual Network Functions". In: *IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pp. 7–13, 10 2014.

[34] LAUFER, R., GALLO, M., PERINO, D., et al. "CliMB: Enabling Network Function Composition with Click Middleboxes". In: *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pp. 50–55, New York, NY, USA, 2016. ACM. ISBN: 978-1-4503-4424-1.

[35] ANDREONI LOPEZ, M., DUARTE, O. C. M. B. "Providing elasticity to intrusion detection systems in virtualized Software Defined Networks". In: *2015 IEEE International Conference on Communications (ICC)*, v. 2015-Septe, pp. 7120–7125, London, United Kingdom, 6 2015. IEEE. ISBN: 978-1-4673-6432-4. doi: 10.1109/ICC.2015.7249462.

[36] MATTOS, D. M. F., DUARTE, O. C. M. B., PUJOLLE, G. "A Resilient Distributed Controller for Software Defined Networking". In: *IEEE ICC 2016 - Next Generation Networking and Internet Symposium (ICC'16 - NGN)*, Kuala Lumpur, Malaysia, 5 2016.

[37] ANDREONI LOPEZ, M., SANZ, I. J., FERRAZANI MATTOS, D. M., et al. "CATRACA: uma Ferramenta para Classificação e Análise Tráfego Escalável Baseada em Processamento por Fluxo". In: *Salão de Ferramentas do XVII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais - SBSeg'2017*, pp. 788–795, 2017.

[38] HESSE, G., LORENZ, M. "Conceptual Survey on Data Stream Processing Systems". In: *IEEE 21st International Conference on Parallel and Distributed Systems*, pp. 797–802, 2015.

[39] GRADVOHL, A. L. S., SENGER, H., ARANTES, L., et al. "Comparing distributed online stream processing systems considering fault tolerance issues", *Journal of Emerging Technologies in Web Intelligence*, v. 6, n. 2, pp. 174–179, 2014.

[40] LANDSET, S., KHOSHGOFTAAR, T. M., RICHTER, A. N., et al. "A survey of open source tools for machine learning with big data in the Hadoop ecosystem", *Journal of Big Data*, v. 2, n. 1, pp. 1–36, 2015.

[41] COLUCCIO, R., GHIDINI, G., REALE, A., et al. "Online stream processing of machine-to-machine communications traffic: A platform comparison". In: *IEEE Symposium on Computers and Communication (ISCC)*, pp. 1–7, 6 2014. doi: 10.1109/ISCC.2014.6912528.

[42] NABI, Z., BOUILLET, E., BAINBRIDGE, A., et al. "Of Streams and Storms", *IBM White Paper*, 2014.

[43] LU, R., WU, G., XIE, B., et al. "Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks". In: *IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pp. 69–78, 2014.

[44] DAYARATHNA, M., SUZUMURA, T. "A performance analysis of System S, S4, and Esper via two level benchmarking". In: *Quantitative Evaluation of Systems*, Springer, pp. 225–240, 2013.

[45] ANDREONI LOPEZ, M., LOBATO, A. G. P., DUARTE, O. C. M. B. "A Performance Comparison of Open-Source Stream Processing Platforms". In: *IEEE GLOBECOM*, pp. 1–6, Washington, USA, 12 2016. IEEE. ISBN: 9781509013289. doi: 10.1109/GLOCOM.2016.7841533.

[46] ANDREONI LOPEZ, M., LOBATO, A. G. P., DUARTE, O. C. M. B. "Monitoramento de Tráfego e Detecção de Ameaças por Sistemas Distribuídos de Processamento de Fluxos: uma Análise de Desempenho", *XXI Workshop de Gerência e Operação de Redes e Serviços (WGRS) do SBRC'2016*, pp. 103–116, 2016.

[47] DU, Y., LIU, J., LIU, F., et al. "A real-time anomalies detection system based on streaming technology". In: *Sixth International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, v. 2, pp. 275–279. IEEE, 2014.

[48] ZHAO, S., CHANDRASHEKAR, M., LEE, Y., et al. "Real-time network anomaly detection system using machine learning". In: *11th International Conference on the Design of Reliable Communication Networks (DRCN)*, pp. 267–270. IEEE, 2015.

[49] HE, G., TAN, C., YU, D., et al. "A real-time network traffic anomaly detection system based on storm". In: *Proceedings - 2015 7th International Conference on Intelligent Human-Machine Systems and Cybernetics, IHMSC 2015*, v. 1, pp. 153–156, 2015. ISBN: 9781479986460. doi: 10.1109/IHMSC.2015.152.

[50] MYLAVARAPU, G., THOMAS, J., TK, A. K. "Real-Time Hybrid Intrusion Detection System Using Apache Storm". In: *17th International Conference on High Performance Computing and Communications*, pp. 1436–1441. IEEE, 8 2015. ISBN: 978-1-4799-8937-9. doi: 10.1109/HPCC-CSS-ICESS.2015.241.

[51] SANTOS, L. A. F., CAMPIOLO, R., BATISTA, D. M. "Uma Arquitetura Autonômica para Detecçao e Reação a Ameaças de Segurança em Redes de Computadores". In: *III WoSiDA'14*, pp. 1–4, 2014.

[52] SCHUARTZ, F. C., MUNARETTO, A., FONSECA, M. "Sistema Distribuído para Detecção de Ameaças em Tempo Real Utilizando Big Data". In: *XXXV Simpósio Brasileiro de Telecomunicações e Processamento de Sinais (SBrT)*, 2017.

[53] JIRSIK, T., CERMAK, M., TOVARNAK, D., et al. "Toward Stream-Based IP Flow Analysis", *IEEE Communications Magazine*, v. 55, n. 7, pp. 70–76, 2017. ISSN: 0163-6804. doi: 10.1109/MCOM.2017.1600972.

[54] SANZ, I. J., ALVARENGA, I. D., ANDREONI LOPEZ, M., et al. "Uma Avaliação de Desempenho de Segurança Definida por Software através de Cadeias de Funções de Rede". In: *XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2017*, 2017.

[55] QUINN, P., ELZUR, U., PIGNATARO, C. *Network Service Header (NSH)*. Technical report, 2018.

[56] AZMANDIAN, F., KAELI, D. R., DY, J. G., et al. "Securing virtual execution environments through machine learning-based intrusion detection". In: *25th International Workshop on Machine Learning for Signal Processing (MLSP)*, pp. 1–6, 2015.

[57] LI, B., LI, J., LIU, L. "CloudMon: a resource-efficient IaaS cloud monitoring system based on networked intrusion detection system virtual appliances", *Concurrency and Computation: Practice and Experience*, v. 27, n. 8, pp. 1861–1885, 2015.

[58] CHEN, H., CLARK, J. A., TAPIADOR, J. E., et al. "A multi-objective optimisation approach to IDS sensor placement". In: *Computational Intelligence in Security for Information Systems*, Springer, pp. 101–108, 2009.

[59] BOUET, M., LEGUAY, J., CONAN, V. "Cost-based placement of virtualized Deep Packet Inspection functions in SDN". In: *IEEE Military Communications Conference, MILCOM*, pp. 992–997. IEEE, 2013.

[60] FERRAZ, L. H. G., MATTOS, D. M. F., DUARTE, O. C. M. B. "A two-phase multipathing scheme based on genetic algorithm for data center networking". In: *IEEE GLOBECOM 2014*, pp. 2270–2275, 12 2014.

[61] ANDREONI LOPEZ, M., MATTOS, D. M. F., FERRAZ, L. H. G., et al. "Localização Eficiente de Sensores Colaborativos para Detecção e Prevenção de Intrusão em Ambientes Virtualizados". In: *XX Workshop de Gerência e Operação de Redes e Serviços (WGRS 2015) do SBRC'2015*, 2015.

[62] BOUET, M., LEGUAY, J., COMBE, T., et al. "Cost-based placement of vDPI functions in NFV infrastructures", *International Journal of Network Management*, v. 25, n. 6, pp. 490–506, 2015.

[63] WOOD, T., SHENOY, P., VENKATARAMANI, A., et al. "Sandpiper: Black-box and gray-box resource management for virtual machines", *Computer Networks*, v. 53, n. 17, pp. 2923–2938, 2009. ISSN: 1389-1286.

[64] CARVALHO, H. E. T., DUARTE, O. C. M. B. "VOLTAIC: volume optimization layer to assign cloud resources". In: *Proceedings of the 3rd International Conference on Information and Communication Systems*, ICICS'12, pp. 3:1–3:7, 2012. ISBN: 978-1-4503-1327-8.

[65] ANDREONI LOPEZ, M., MATTOS, D. M. F. D., DUARTE, O. C. M. B. "Evaluating Allocation Heuristics for an Efficient Virtual Network Function Chaining". In: *7th International Conference Network of the Future (NoF'16)*. IEEE, 2017. ISBN: 9781509046713. doi: 10.1109/NOF.2016.7810141.

[66] CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., et al. "Monitoring Streams: A New Class of Data Management Applications". In: *28th International Conference on Very Large Data Bases*, pp. 215–226, 2002.

[67] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., et al. "The Design of the Borealis Stream Processing Engine." *Cidr*, pp. 277–289, 2005. doi: 10.1.1.118.7039.

[68] RYCHLY, M., KODA, P., SMRZ, P. "Scheduling Decisions in Stream Processing on Heterogeneous Clusters". In: *Eighth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pp. 614–619, 7 2014.

[69] ZAHARIA, M., DAS, T., LI, H., et al. "Discretized streams: Fault-tolerant streaming computation at scale". In: *XXIV ACM Symposium on Operating Systems Principles*, pp. 423–438. ACM, 2013.

[70] MARZ, N., WARREN, J. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems.* 1st ed. Greenwich, CT, USA, Manning Publications Co., 2013.

[71] WIDOM, J. "The Starburst rule system: Language design, implementation, and applications", *IEEE Data Engineering Bulletin*, 1992.

[72] STONEBRAKER, M., KEMNITZ, G. "The POSTGRES next generation database management system", *Communications of the ACM*, v. 34, n. 10, pp. 78–92, 1991.

[73] CHEN, J., DEWITT, D. J., TIAN, F., et al. "NiagaraCQ: A scalable continuous query system for internet databases". In: *ACM SIGMOD Record*, v. 29, pp. 379–390. ACM, 2000.

[74] ARASU, A., BABCOCK, B., BABU, S., et al. *STREAM: The Stanford Data Stream Management System.* Technical Report 2004-20, Stanford InfoLab, 2004.

[75] BALAZINSKA, M., BALAKRISHNAN, H., STONEBRAKER, M. "Load management and high availability in the Medusa distributed stream processing system". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04*, p. 929, New York, New York, USA, 2004. ACM Press. ISBN: 1581138598. doi: 10.1145/1007568.1007701.

[76] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., et al. "TelegraphCQ: continuous dataflow processing". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, p. 668. ACM, 2003.

[77] DEMERS, A. J., GEHRKE, J., PANDA, B., et al. "Cayuga: A General Purpose Event Monitoring System." In: *Proceedings of the Conference on Innovative Data Systems Research*, v. 7, pp. 412–422, 2007.

[78] CARBONE, P., EWEN, S., HARIDI, S., et al. "Apache Flink: Unified Stream and Batch Processing in a Single Engine", *Data Engineering*, pp. 28–38, 2015.

[79] KAMBURUGAMUVE, S., FOX, G., LEAKE, D., et al. "Survey of distributed stream processing for large stream sources". 2013.

[80] ANDREONI LOPEZ, M., LOBATO, A. G. P., MATTOS, D. M. F., et al. "Um Algoritmo Não Supervisionado e Rápido para Seleção de Características em Classificação de Tráfego". In: *XXXV SBRC'2017*, Belém- Pará, PA,, 2017.

[81] ANDREONI LOPEZ, M., LOBATO, A. G. P., DUARTE, O. C. M. B., et al. "An evaluation of a virtual network function for real-time threat detection using stream processing". In: *IEEE Fourth International Conference on Mobile and Secure Services (MobiSecServ)*, pp. 1–5, 2018. doi: 10.1109/MOBISECSERV.2018.8311440.

[82] CHENG, Z., CAVERLEE, J., LEE, K. "You Are Where You Tweet: A Content-based Approach to Geo-locating Twitter Users". In: *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pp. 759–768. ACM, 2010. ISBN: 978-1-4503-0099-5.

[83] LOBATO, A. G. P., ANDREONI LOPEZ, M., DUARTE, O. C. M. B. "Um Sistema Acurado de Detecção de Ameaças em Tempo Real por Processamento de Fluxos". In: *SBRC'2016*, pp. 572–585, Salvador, Bahia, 2016.

[84] HEIDEMANN, J., PAPDOPOULOS, C. "Uses and challenges for network datasets". In: *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*, pp. 73–82. IEEE, 2009.

[85] LIPPMANN, R. P., FRIED, D. J., GRAF, I., et al. "Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation". In: *Proceedings of DARPA Information Survivability Conference and Exposition. DISCEX'00.*, v. 2, pp. 12–26. IEEE, 2000.

[86] HAINES, J. W., LIPPMANN, R. P., FRIED, D. J., et al. *1999 DARPA intrusion detection evaluation: Design and procedures.* Technical report, Massachusetts Inst Of Tech Lexington Lincoln Lab, 2001.

[87] LEE, W., STOLFO, S. J., MOK, K. W. "Mining in a data-flow environment: Experience in network intrusion detection". In: *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 114–124. ACM, 1999.

[88] TAVALLAEE, M., BAGHERI, E., LU, W., et al. "A detailed analysis of the KDD CUP 99 data set". In: *Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications*. IEEE, 2009.

[89] SOMMER, R., PAXSON, V. "Outside the closed world: On using machine learning for network intrusion detection". In: *IEEE Symposium on Security and Privacy (SP)*, pp. 305–316. IEEE, 2010.

[90] SHIRAVI, A., SHIRAVI, H., TAVALLAEE, M., et al. "Toward developing a systematic approach to generate benchmark datasets for intrusion detection", *Computers and Security*, v. 31, n. 3, pp. 357–374, 2012. ISSN: 0167-4048.

[91] GARCIA, S., GRILL, M., STIBOREK, J., et al. "An empirical comparison of botnet detection methods", *Computers & Security*, v. 45, pp. 100–123, 2014.

[92] FONTUGNE, R., BORGNAT, P., ABRY, P., et al. "{MAWILab}: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking". In: *ACM CoNEXT '10*, Philadelphia, PA, 2010.

[93] JUNGSUK, S., TAKAKURA, H., OKABE, Y. *Description of Kyoto university benchmark data*. Technical Report 01, Academic Center for Computing and Media Studies (ACCMS), Kyoto University, 2006.

[94] SWEENEY, L. "Achieving k-anonymity privacy protection using generalization and suppression", *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, v. 10, n. 05, pp. 571–588, 2002.

[95] KARGUPTA, H., DATTA, S., WANG, Q., et al. "On the privacy preserving properties of random data perturbation techniques". In: *Third IEEE International Conference on Data Mining*, pp. 99–106, 11 2003. doi: 10.1109/ICDM.2003.1250908.

[96] MUNTÉS-MULERO, V., NIN, J. "Privacy and Anonymization for Very Large Datasets". In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pp. 2117–2118, New

York, NY, USA, 2009. ACM. ISBN: 978-1-60558-512-3. doi: 10.1145/1645953.1646333. Available in: <http://doi.acm.org/10.1145/1645953.1646333>.

[97] TANTITHAMTHAVORN, C., MCINTOSH, S., HASSAN, A. E., et al. "An Empirical Comparison of Model Validation Techniques for Defect Prediction Models", *IEEE Transactions on Software Engineering*, v. 43, n. 1, pp. 1–18, 1 2017. ISSN: 0098-5589. doi: 10.1109/TSE.2016.2584050.

[98] B. CLAISE, E., B. TRAMMELL, E., AITKEN, P. "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information". RFC 7011 (Informational), 2013.

[99] ROESCH, M. "Snort-Lightweight Intrusion Detection for Networks". In: *Proceedings of the 13th USENIX conference on System administration*, pp. 229–238. USENIX Association, 1999.

[100] GARCÍA, S., LUENGO, J., HERRERA, F. "Tutorial on practical tips of the most influential data preprocessing algorithms in data mining", *Knowledge-Based Systems*, v. 98, pp. 1–29, 4 2016. ISSN: 09507051. doi: 10.1016/j.knosys.2015.12.006.

[101] ZHANG, S., ZHANG, C., YANG, Q. "Data preparation for data mining", *Applied artificial intelligence*, v. 17, n. 5-6, pp. 375–381, 2003.

[102] TAN, S. "Neighbor-weighted k-nearest neighbor for unbalanced text corpus", *Expert Systems with Applications*, v. 28, n. 4, pp. 667–671, 2005.

[103] RAMÍREZ-GALLEGO, S., KRAWCZYK, B., GARCÍA, S., et al. "A survey on data preprocessing for data stream mining: Current status and future directions", *Neurocomputing*, 2017. ISSN: 18728286. doi: 10.1016/j.neucom.2017.01.078.

[104] ZHAI, Y., ONG, Y.-S., TSANG, I. W. "The Emerging Big Dimensionality", *Comp. Intell. Mag.*, v. 9, n. 3, pp. 14–26, 2014. ISSN: 1556-603X. doi: 10.1109/MCI.2014.2326099.

[105] VAN DER MAATEN, L., POSTMA, E., DEN HERIK, J. "Dimensionality reduction: a comparative", *Journal of Machine Learning Research*, v. 10, pp. 66–71, 2009.

[106] AINHOREN, Y., ENGELBERG, S., FRIEDMAN, S. "The cocktail party problem", *IEEE Instrumentation and Measurement Magazine*, 2008. ISSN: 10946969. doi: 10.1109/MIM.2008.4534378.

[107] SCHÖLKOPF, B., SMOLA, A. J., MÜLLER, K.-R. "Kernel principal component analysis". In: *Advances in kernel methods*, pp. 327–352. MIT Press, 1999.

[108] MLADENIĆ, D. "Feature Selection for Dimensionality Reduction". In: Saunders, C., Grobelnik, M., Gunn, S., et al. (Eds.), *Subspace, Latent Structure and Feature Selection (SLSFS): Statistical and Optimization Perspectives Workshop.*, Springer Berlin Heidelberg, pp. 84–102, Bohinj, Slovenia, 2006. ISBN: 978-3-540-34138-3. doi: 10.1007/11752790-5.

[109] ANG, J. C., MIRZAL, A., HARON, H., et al. "Supervised, Unsupervised, and Semi-Supervised Feature Selection: A Review on Gene Selection", *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, v. 13, n. 5, pp. 971–989, 9 2016. ISSN: 1545-5963. doi: 10.1109/TCBB.2015.2478454.

[110] GUYON, I., WESTON, J., BARNHILL, S., et al. "Gene selection for cancer classification using support vector machines", *Machine learning*, v. 46, n. 1-3, pp. 389–422, 2002.

[111] CHANDRASHEKAR, G., SAHIN, F. "A survey on feature selection methods", *Computers & Electrical Engineering*, v. 40, n. 1, pp. 16–28, 2014. ISSN: 0045-7906. doi: http://dx.doi.org/10.1016/j.compeleceng.2013.11.024.

[112] ROBNIK-ŠIKONJA, M., KONONENKO, I. "Theoretical and Empirical Analysis of ReliefF and RReliefF", *Machine Learning*, v. 53, n. 1/2, pp. 23–69, 2003. ISSN: 08856125. doi: 10.1023/A:1025667309714.

[113] HALL, M. A. *Correlation-based Feature Selection for Machine Learning.* Tese de Doutorado, The University of Waikato, 1999.

[114] BEN-HAIM, Y., TOM-TOV, E. "A streaming parallel decision tree algorithm", *Journal of Machine Learning Research*, v. 11, n. Feb, pp. 849–872, 2010.

[115] WEBB, G. I. "Contrary to popular belief incremental discretization can be sound, computationally efficient and extremely useful for streaming data". In: *IEEE International Conference on Data Mining (ICDM)*, pp. 1031–1036. IEEE, 2014.

[116] HU, H., KANTARDZIC, M. "Smart preprocessing improves data stream mining". In: *49th Hawaii International Conference on System Sciences (HICSS)*, pp. 1749–1757. IEEE, 2016.

[117] BUCZAK, A., GUVEN, E. "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection", *IEEE Communications Surveys Tutorials*, , n. 99, pp. 1–26, 2015.

[118] PRASATH, V. B. S., ALFEILAT, H. A. A., LASASSMEH, O., et al. "Distance and Similarity Measures Effect on the Performance of K-Nearest Neighbor Classifier - A Review", *CoRR*, v. abs/1708.0, 2017.

[119] ZHANG, T. "Solving large scale linear prediction problems using stochastic gradient descent algorithms". In: *Proceedings of the twenty-first international conference on Machine learning*, p. 116. ACM, 2004.

[120] CHAWLA, N. V., BOWYER, K. W., HALL, L. O., et al. "SMOTE: synthetic minority over-sampling technique", *Journal of artificial intelligence research*, v. 16, pp. 321–357, 2002.

[121] PERKINS, S., THEILER, J. "Online feature selection using grafting". In: *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 592–599, 2003.

[122] ZHOU, J., FOSTER, D. P., STINE, R. A., et al. "Streamwise feature selection", *Journal of Machine Learning Research*, v. 7, n. Sep, pp. 1861–1885, 2006.

[123] WU, X., YU, K., DING, W., et al. "Online feature selection with streaming features", *IEEE transactions on pattern analysis and machine intelligence*, v. 35, n. 5, pp. 1178–1192, 2013.

[124] YANG, W., FUNG, C. "A survey on security in network functions virtualization". In: *IEEE NetSoft Conference and Workshops*, pp. 15–19. IEEE, 2016.

[125] MIJUMBI, R., SERRAT, J., GORRICHO, J.-L., et al. "Network Function Virtualization: State-of-the-Art and Research Challenges", *IEEE Communications Surveys & Tutorials*, v. 18, n. 1, pp. 236–262, 2015.

[126] MIJUMBI, R., SERRAT, J., GORRICHO, J.-L., et al. "Management and orchestration challenges in network functions virtualization", *IEEE Communications Magazine*, v. 54, n. 1, pp. 98–105, 1 2016. ISSN: 0163-6804. doi: 10.1109/MCOM.2016.7378433.

[127] ETSI GS NFV-MAN. *Network Functions Virtualisation (NFV); Management and Orchestration*. Technical Report 001, European Telecommunications Standards Institute (ETSI), 12 2014.

[128] BARI, M. F., CHOWDHURY, S. R., AHMED, R., et al. "Orchestrating Virtualized Network Functions", *Transactions on Network and Service Management*, v. PP, n. 99, 5 2016.

[129] MEDEIROS, D. S. V., CAMPISTA, M. E., MITTON, N., et al. "Weighted Betweenness for Multipath Networks". In: *Global Information Infrastructure and Networking Symposium (GIIS)*, 2016.

[130] QUINN, P., NADEAU, T. *Problem Statement for Service Function Chaining.* Technical Report RFC 7498, Active Internet-Draft, TETF Secretariat, 2015.

[131] COUTO, D. R. S., SECCI, S., CAMPISTA, M. E. M., et al. "Reliability and Survivability Analysis of Data Center Network Topologies", *Journal of Network and Systems Management*, v. 24, n. 2, pp. 346–392, 2016.