Network Log analysis module

GECAD

2019 / 2020

1160779 Pedro Miguel Loureiro Ribeiro

Network Log analysis module

GECAD

2019 / 2020

1160779 Pedro Miguel Loureiro Ribeiro

Bachelors in Computer Science

September 2020

ISEP supervisor/ external supervisor: **Isabel Praça**

ISEP co-supervisors: **Orlando Sousa and Eva Maia**

Dedicated to my parents, Ana and José, and my sister Sara

Acknowledgements

Throughout the development of this capstone project, I have received a great deal of support and assistance.

I would like to express my deep and sincere gratitude to my capstone project supervisor, professor Isabel Praça and my co-supervisor, professor Orlando Sousa for their constant support and instrumental feedback throughout the project. Furthermore, I would like to thank them for their guidance and for believing in me, for being so comprehensive and giving me the freedom to move at my own pace.

I would also like to acknowledge researcher Eva Maia and her involvement in the initial phase of the project and thank her for her valuable input and other fellow colleagues also developing their PESTI projects in GECAD for their help and company.

I am extremely grateful to my parents for their love, encouragement and all the sacrifices they have endured to ensure my success and well-being. I am also very grateful for my younger sister, Sara, which always put me in a good mood whenever I was feeling down.

I want to give a heartfelt thanks to my good friend Luís for having taken the time to go over this report and give me sincere feedback.

A special thanks to all my friends which supported me throughout this journey and for their empathy.

Abstract

As computer networks grow larger and become more complex, it becomes increasingly necessary to ensure the security of both the infrastructure and the data assets they encompass. Adopted protection measures must be able to counter the erratic nature of today's attacks and dynamically adapt to its surrounding environment. However, data over which solutions need to be trained to perform intrusion detection are often obtained from wildly different sources in a network, can be of different natures and can even come in different formats.

The devised solution aims to consolidate network traffic data of disparate types and formats into one, more general, machine learning-ready format. To maximize the accuracy of the machine learning algorithm which will eventually be trained on the output produced by the devised solution, a set of features were defined to characterize the network traffic data. These features were chosen for their relevancy in describing network events and certain user behaviors and their capability of discriminating between attack traffic and normal traffic. This was achieved through an in-depth study of the existing types of network traffic data and the different formats which one type may come in was done.

The final results reveal that the devised solution complies with the established requirements and will be able to serve its role in an intrusion detection pipeline.

Keywords (Theme): Intrusion detection, KDD, feature selection, data preprocessing

Keywords (Technologies): Python3, Scapy, Wireshark

Resumo

À medida que as redes informáticas crescem e se tornam mais complexas, torna-se cada vez mais necessário garantir a segurança tanto das infra-estruturas como dos bens de dados que elas abrangem. As medidas de proteção adotadas devem ser capazes de dar resposta à natureza errática dos ataques de hoje em dia e adaptar-se dinamicamente ao ambiente em que se insere. No entanto, os dados sobre os quais é necessário treinar as soluções para efetuar a deteção de intrusões são frequentemente obtidos a partir de fontes extremamente diferentes numa rede, podendo estes ser de naturezas diferentes e até mesmo serem disponibilizados em formatos diferentes.

A solução concebida visa consolidar dados de tráfego de rede de tipos e formatos díspares num único formato, mais geral e pronto a ser dado como *input* a um algoritmo de *Machine Learning*. Para maximizar a precisão do algoritmo de ML que acabará por ser treinado com o *output* produzido pela solução concebida, foi definido um conjunto de características (*features*) para caraterizar os dados de tráfego da rede. Estas características foram escolhidas com base na sua relevância para descrever certos eventos e comportamentos dos utilizadores numa rede e a sua capacidade para discriminar entre tráfego de ataque e tráfego normal. Isto foi conseguido através de um estudo aprofundado dos tipos de dados de tráfego de rede existentes e os diferentes formatos em que um tipo pode entrar foi feito.

Os resultados finais revelam que a solução concebida cumpre os requisitos estabelecidos e será capaz de servir o seu papel num *pipeline* deteção de intrusões.

Palavras-chave (Tema): Deteção de intrusão, KDD, selecão de *features*, pré-processamento de dados

Palavras-chave (Tecnologias): Python3, Scapy, Wireshark

Index

Figure Index

Table Index

Notation and Glossary

AI	Artificial intelligence
ANIDS	Anomaly-based network-based intrusion detection system
BYOD	Bring Your Own Device
GECAD	Research Group on Intelligent Engineering and Computing for Advanced Innovation and Development
HIDS	Host-based intrusion detection system
IDS	Intrusion detection system
IPS	Intrusion prevention system
ISEP	Instituto Superior de Engenharia do Porto
IT	Information Technology
KDD	Knowledge discovery in data(bases)
ML	Machine Leaning
NIDS	Network-based Intrusion Detection Systems
PESTI	Projeto/Estágio
R&D	Research and Development
SAST	Static application security testing
UC	Use case
WFH	Work From Home

1 Introduction

Over the past century, humankind has observed an immense growth of the Information Technology (IT) field. It saw the way it communicated and shared information completely change, with an undeniable impact on all aspects of life, including the way business is conducted, the way humans express themselves, interact and communicate with each other and the way knowledge is shared and obtained.

Companies founded today, seeking to secure their place in the market and become globally recognized, are built over a digital foundation, with a strong emphasis on rapid delivery of services and scalability. However, a strong cyber-security foundation is equally as important and oftentimes, overlooked.

The technology of today is imperfect and serves many more purposes than just the one it was designed for. Hackers, individuals with advanced skills and vast knowledge in computer science, thrive on finding flaws and vulnerabilities in technology which they can then exploit for their gain, use to access systems and disrupt services, or sell online to the highest bidder.

Cyber-attacks taking advantage of these kinds of vulnerabilities in both software and hardware solutions, became prolific during the past decade, bringing attention to the urgent necessity to adopt a mindset of security by design. In 2011, Sony suffered a major security breach, revealing that hackers were able to access vital data of all its 70 million customers, resulting in losses of over 171 million dollars [1]. EasyJet, one of the largest airlines in the world, was also target of a cyberattack in May of 2020, which exposed personal info (email addresses, travel details) of 9 million of its customers. 2208 of those customers also had their banking information stolen [2]. In their 2019 midyear data breach report, RiskBased Security states that in the first quarter of 2019 alone, over 3813 breaches were reported, which resulted in 4.1 billion records being exposed to the Internet [3].

To ensure their well-standing and acceptance by the consumer market, enterprises need to heavily invest in the protection of their network infrastructure and data assets. They must be aware of the inherent risks which managing a digital business poses and strive to enforce cyber-security policies which ensure that the confidentiality, integrity and availability of their systems is not compromised. Failing to do so, can result in huge financial losses, damage to the brand's reputation and heavy legal repercussions.

To that end, several cyber security solutions are available, which operate on different facets of cybersecurity, including firewalls, antiviruses, intrusion detection systems (IDS), intrusion prevention systems (IPS) and cyber-physical solutions which relate physical events with software ones. Network-based Intrusion Detection Systems (NIDS) have been very sought after to integrate the line of defense against cyber-attacks, even more so during the COVID-19 pandemic as business shifted towards a Bring Your Own Device (BYOD) and Work From Home (WFH) working cultures [4]. NIDS can identify and flag cyber-attacks, by analyzing packet data which travelled over a network. In their primitive state, these systems were only able to detect attacks which were already known to it, based on a list of pre-compiled cyberattack signatures, offering very little protection against novel attacks.

However, the cybersecurity paradigm has changed. Attacks of today have become more sophisticated, leveraging vulnerabilities which are unknown to product vendors (zero-day attacks) and making use of increasingly complex methods to remain undetectable. This demands an even bigger effort by security experts to devise dynamic solutions, capable of adapting to the changes in the environment where they were deployed and perform detection of threats which might not even exist yet.

With the appearance of more capable hardware and with Artificial Intelligence (AI) technology becoming increasingly accessible, more recent IDS solutions are more autonomous and can detect a wider range of cyberattacks. Moreover, through the use of Machine Leaning (ML) technology, they actively learn from previously detected attacks, with minimal human intervention.

The success of these types of technologies, however, is highly dependent on adequate data preprocessing, which has a direct impact on the performance and the capacity of the algorithm to detect or not detect intrusions on a network.

1.1 Background

All work presented in this report was developed in the context of a project organized by Research Group on Intelligent Engineering and Computing for Advanced Innovation and Development (GECAD). This project, which will be addressed as “parent project” for readability-sake, is a Knowledge Discovery in Data(bases) (KDD) project, whose main goal is to extract “high-level knowledge from low-level data in the context of large data sets.” [5].

The proposed solution corresponds to a modular pipeline, where each major functionality is encapsulated in one, self-contained module. With functionality ranging from the collection of

raw network traffic to the presentation of statistics resulting from the detection, the solution stands as a robust and highly versatile means of detection of cyber-attacks, carried out over networks of high caliber.

The project being presented in this report, which will be referred to as “child project” throughout this section, aims to develop one of the modules which are part of the final solution of the parent project. This module lies between two others: one that handles the collection of raw network traffic data from a diverse set of sources (Module 1 in Figure 1) and the other which performs the detection of anomalous activity with resource to a machine learning algorithm (Module 3 in Figure 1). The developed module (Module 2) receives the raw network traffic from the first one, preprocesses that data and outputs it to the third one, in a format optimized for machine learning processing.

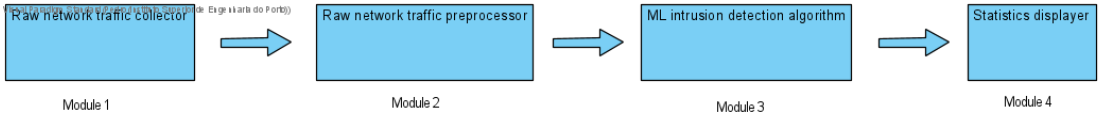


Figure 1 - Parent's project pipeline

The work covered in the present report was developed in the context of the curricular unit of Projeto/Estágio (PESTI). Within this curricular unit, students are expected to integrate a project either in a professional setting or not, analyze a problem and develop a solution that reflects all or part of the knowledge, competences, soft skills, and technical skills acquired throughout the Computer Science bachelor's degree, offered by Instituto Superior de Engenharia do Porto (ISEP).

1.1.1 Airbus

The parent project described in the previous section is being developed in conjunction with Airbus. Airbus [6] is an aerospace company, based in Europe, which designs and manufactures aircraft and helicopters. The three main areas in which Airbus’ operations focus on are Space and Defense, Commercial Aircraft and Helicopters.

With an annual investment of approximately 3 billion dollars into R&D [6], the project which counts with the participation of GECAD focuses on the study of ML-based solutions for the detection of intrusions on a network.

The application developed supports multiple file formats proposed by Airbus. To aid in the development, a set of files were provided which served as basis for the implementation and that should reflect the format to be supported.

1.1.2 Motivation

The accepted project presents a compelling challenge, in the cybersecurity field, requiring the application of knowledge spanning various areas of computer science, namely, networking, data mining and software engineering. For this reason, involvement in the project presented a very rich learning experience and a means to not only acquire new knowledge but also review and reinforce existing one. The project also required the use of new technologies, greatly benefiting from the use of a programming language unknown to the student, which just made it more interesting.

Furthermore, work for the project would be realized in a research environment, providing an opportunity to not only gain experience as a researcher but also interact with other people in the area and learn the types of challenges research work entails.

Additionally, the proponent organization is a research unit of renown, with many years of experience in the area of research and successful contributions to various fields of computer science. This made for a very rewarding experience to not only be able to actively contribute to its success and to learn from those who are a part of it but also be able to contribute to a project which will gain the recognition of researchers around the world and possibly integrate a real-life solution.

1.1.3 Organization presentation

GECAD [7] is the largest Research and Development (R&D) unit from the Polytechnic sub-system of Portugal, located inside ISEP's facilities. Its mission is to promote and develop scientific research a Knowledge and Decision Sciences domains, having Information Technologies as support. GECAD is most notably known for its involvement in the areas of Intelligent Systems and Power Energy Systems and has successfully contributed to important scientific journals.

It also has a strong presence in the research field of areas such as Affective Computing, Ambient Intelligence, Artificial Intelligence, Cyber-Physical Systems, Group Decision Support, and Internet-of-Things.

1.2 Problem description

Before an ML algorithm can perform intrusion detection it must first be trained on a large quantity of network traffic data, which might be labeled or not, depending on if learning is supervised or unsupervised, respectively. However, this data can be of different types, and each type can, in turn, be represented in different formats.

Nowadays, network traffic data will fall into one of two categories: “flows” or “packets”. Several format standards have been defined for both types of network traffic data.

The most popular packet-based network traffic data format is the pcap format, also called tcpdump format. This format is widely used and supported by numerous network monitoring-related tools [8].

Several standards exist for unidirectional and bidirectional flow-based network traffic data formats, including the family of Netflow versions, with Netflow v5 and Netflow v9 being the most popular, IPFIX, sFlow, among others [9].

Besides the standard formats previously presented, both flow-based data and packet-based data might be stored in a custom format like is the case of the Suricata IDS [10] that has the option to export flow-based network data into a JSON log file.

Additionally, datasets, which are flow-based network traffic data in CSV format, can also be used to train an algorithm. A good dataset should contain both normal and malicious activity, be publicly available and be labeled [11]. Each record of the dataset is related to one flow, either unidirectional or bidirectional, and is characterized by a number of features, arbitrarily defined by the authors [12].

Adding to the problem of the large number of formats in which network traffic data may come in, the data itself might contain unexpected artifacts that can negatively impact the performance and the success of the ML algorithm, often leading to an inaccurate model.

To account for the fact that all data on which an ML algorithm is trained, tested or analyzed on must follow a consistent and well-defined format and that data might contain errors which might have been induced during its creation or capture, this raw network traffic data must first be subject to preprocessing and be transformed into an appropriate form for subsequent analysis [13].

Preprocessing will consist on the conversion of raw network traffic into a series of observations, where each observation is represented as a feature vector. Optionally, each feature vector can be labeled using either domain knowledge or an automated tool such as an IDS. These feature vectors are then suitable as input to data mining or machine learning algorithms [12]. The steps involved in the preprocessing include the consolidation of data of different formats into a single, well-defined format, removal of any duplicate, inconsistent or irrelevant records, and defining a strategy for handling missing data fields [5].

Moreover, the process of definition of a general format which all training datasets produced by the proposed solution will adhere to, demanded an extensive study of the set of features most relevant to the detection of the widest range of attacks as possible and that best highlight the contrast between normal user behavior and attack behavior. These are the features that will be extracted and calculated from the raw data and later sent to the output file.

1.2.1 Objectives

The main goal of the proposed project is to develop a solution that is capable of receiving network traffic data in different formats and consolidate them into one single, ML-ready format for the training and testing of ML algorithms. The format of training sets produced by the solution should be characterized by a set of relevant features that reflect high-level information regarding the events that occurred on a network.

With these requirements in mind, the following objectives were set:

- Understand the different types of network flow data and the different formats in which each type of network data may exist in and how they relate to each other.
- Understand what is an IDS, what kinds of IDS exist, and what types of attacks are anomaly-based IDS capable of detecting.
- Study what kind of features are used in publicly available datasets and identify the top 20 most used features.
- Develop a solution that can receive, as input, files of multiple formats and produces, as output, a training set, characterized by a number of pre-defined features, with the same format, regardless of the file received as input. The solution must also be flexible enough to support other formats or features in the future, without requiring major changes to the code base.
- Employ feature selection in order to reduce the number of features present in the produced training set, while still maintaining data integrity.

1.2.2 Approach

The approach taken to solve the proposed problem was similar to that taken in research projects: initially, a large period of time was allocated to the study of the concepts related to the problem, namely, intrusion detection, IDS types of IDS, IDS solutions available in the market, types of network traffic data and formats each type might be found in. This was followed by an investigation of the features to calculate/extract from the input data, which would be saved to the output file. This was a crucial task as the information on which a model

is trained will directly impact the performance of the intrusion detection algorithm. To that end, articles relating to the most influential intrusion detection datasets, published in the last decade, were analyzed, with special focus on what features were chosen and what was the author's reasoning for choosing those particular features. Other articles with a focus towards the theoretical study of traffic classification and of relevant features were also analyzed.

Once all the required knowledge was gathered and a deeper understanding of the field and the different concepts it involves was attained, the analysis phase could start. Based on the artifacts resulting from the analysis phase, the design of the application was idealized and then came the application implementation, terminated by the testing phase.

Since the parent project in which the developed is very KDD-oriented, the application development will match the "Preprocessing" and "Transformation" phases of the KDD process, as seen in Figure 2.

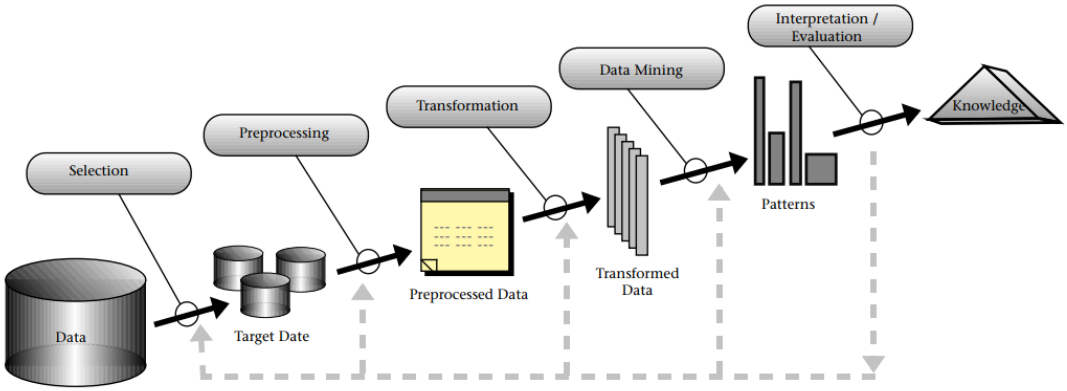


Figure 2 - An overview of the steps that compose the KDD process [5]

KDD is the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data [5]. It is an interactive and iterative process, involving the steps as depicted in Figure 2. Besides the steps mentioned, it requires a prior understanding of the application domain and of any relevant knowledge. One or more steps might correspond to one module in the project's pipeline, as illustrated in Figure 1.

1.2.3 Contributions

Based on the objectives defined in the "Objectives" section, the following are a set of contributes which the proposed solution stands to offer both the proposing organization and the field of cybersecurity:

- Versatile and lightweight application, written in Python, for the preprocessing of network data for ML-based intrusion detection.

- Comprehensive analysis of over 20 most publicly available intrusion detection datasets, alongside a table containing all features present in each of the analyzed datasets.
- Thorough explanation of the software solution, alongside code snippets and a textual descriptions of design decisions and justification for taking them, based on coding good practices and design principles which can serve as reference for future developers.

The application, on its own, does not bring much value as the functionality it entails is rather simplistic. Its only when it is integrated with the remaining modules of the parent project that it contributes towards a market-ready solution for the detection of intrusions on a network which, in turn, will contribute to more safer networks.

1.2.4 Work planning

The work plan for the project was highly influenced by the chosen development methodology. Since all requirements were settled at the beginning of the project and the probability of those requirements changing over time was slim, the Waterfall methodology was chosen to guide and orient the project development.

The applicability of other methodologies to the project, such as the Agile-based Scrum or Kanban frameworks, was studied. However, they did not fit the linear and non-repetitive nature of the project and their usage could not be justified.

The Waterfall methodology [14] proposes a linear mode of operation, where work is split into sequential stages, where each stage must be completed before advancing to the next one. Further, work is developed in a unidirectional fashion meaning that if changes are introduced into a phase which has already been completed, the whole process must be restarted.

This work model is particularly suited for projects whose requirements and scope are all known in advance and which are not likely to change, over time. The Waterfall methodology comprises six phases, as presented in Figure 3.

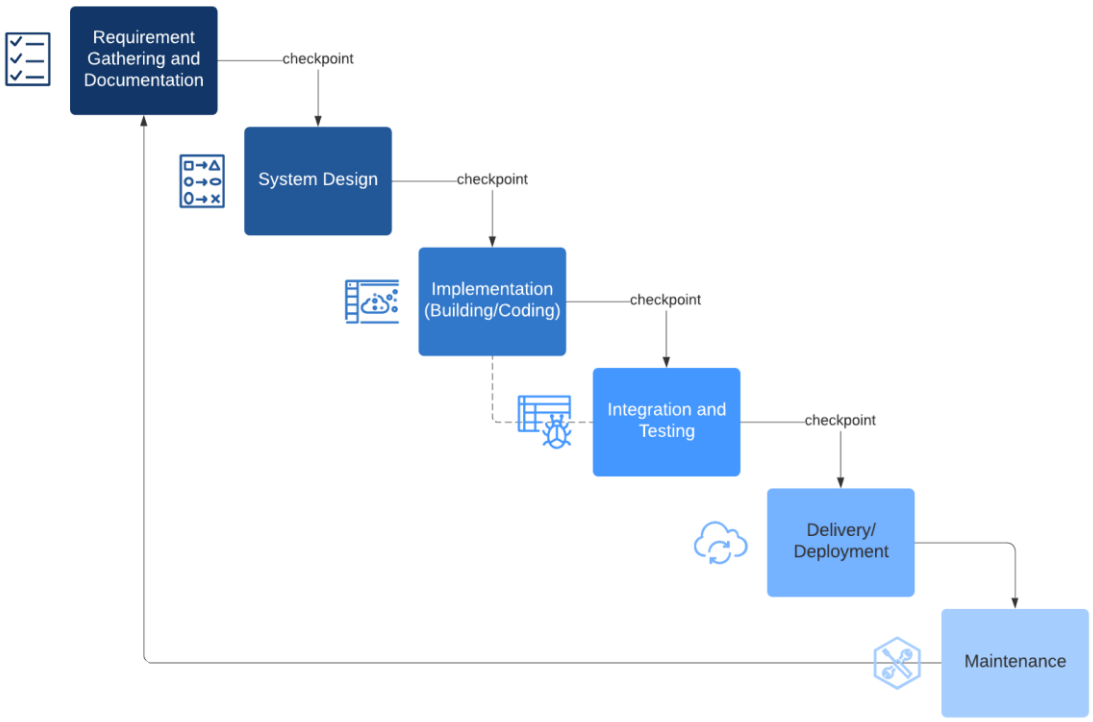


Figure 3 - Phases of the Waterfall process [15]

The first phase (Requirement gathering and documentation) involves the gathering of comprehensive information about what this project requires. This includes the comprehension and documentation of all requirements and objectives to complete, to achieve the required level of functionality by the application.

Next up is the second phase (System design) where the requirements specifications from the first phase are studied, enabling the definition of the overall system architecture and the different components which make up the final solution.

The third phase (Implementation) is characterized by the implementation of the desired system, based on the design decisions made in the previous step. The implementation process consists in the construction of small, easily testable units that, once integrated, make up the complete system.

Once all units are finished being implemented, the project advances to the fourth phase (Testing and integrating) where each unit is tested against some expected results, to assess if the implemented functionalities are working as intended and if both functional and non-functional requirements imposed by the client are being fulfilled. Once testing is done, all units are integrated into one system which will correspond to the final solution.

In the fifth and sixth phases of the Waterfall process, the product is deployed in the customer environment and then monitored and maintained by the development team, which will give support to the client, fix errors and flaws detected, and alter attributes or improve performance, as needed [14].

Once the most appropriate methodology was found for the realization of the project, an in-depth work plan was elaborated. This plan was elaborated between the student and the proponent organization and reflects all major phases of the project's lifecycle. Over time, the work planning was modified to account for the student's personal situation and external events such as the COVID-19 pandemic. The work plan, in the form of a Gantt Chart, found in Annex A, corresponds to the last revision made and should accurately represent the workflow of the project.

1.3 Report structure

The remaining report is structured as follows:

Chapter 2 establishes a technological context, introducing the reader that might not be so knowledgeable in the cybersecurity field, to some fundamental concepts used in this report.

Chapter 3 presents a study of the other relevant work conducted to the project developed and existing technologies used in the development of the final solution.

Chapter 4 explains the design approach followed by the student to solve the problem, including not only how the different methods, techniques, algorithms, and technologies were used but also the problem comprehension process.

Chapter 5 describes, in detail, how the solution proposed in the previous chapter was achieved and what kind of technologies and methodologies were employed to that end. The chapter also presents conducted testing and a brief evaluation of the application.

Chapter 6 concludes the report, reflecting on the developed work, how the goals were or were not successfully achieved and limitations and future work.

2 Technological context

In this section, a contextualization of crucial concepts related to the problem at hand is offered. T to give more context into the problem and the kind of technology which is involved. This section main intention is to give the less informed reader, a body of knowledge containing essential knowledge for a better understanding of the problem and act as a reference which the reader should refer to for a more detailed explanation of the business domain's concepts.

At the end of this section, the reader should feel more comfortable with the technological aspects of the project and be able to enact a more informed judgment of the proposed solution.

2.1 PCAP format

PCAP, an abbreviation of "packet capture", is a file format, also commonly called the "libpcap format" or the "tcpdump format", originally created by V. Jacobson, C. Leres and S. McCanne [16] for use with the tcpdump tool [17]. This file format has been registered with IANA as the MIME type application/vnd.tcpdump.pcap [16]. It is supported by the libpcap library [17], developed for POSIX-like systems and maintained by the development team of the tcpdump tool and the npcap library [18], which is a port of the libpcap library for Windows systems. These two libraries support the capture of network traffic at lower levels and essentially provide an API for developers who wish to imbue their applications with packet capture capabilities [19].

Names such as "PCAP file" or "libpcap capture file" refer to the same concept and are used interchangeably throughout this section.

A PCAP file [20] [21] is composed of a global header, which contains information regarding the whole file and individual headers for each of the packets it holds, as illustrated in Table 1:

Table 1 - Overview of pcap format

Global header	Packet header #1	Packet data #1	Packet header #2	Packet data #2	Packet header #3	Packet data #3	…

2.1.1 Global header

A global header is composed of the following fields:

Table 2 – Fields in the global header of pcap format

32 bit unsigned int	16 bit unsigned int	16 bit unsigned int	32 bit signed int	32 bit unsigned int	32 bit unsigned int	32 bit unsigned int
magic_number	version_major	version_minor	thiszone	sigfigs	snaplen	network

magic_number Specifies the byte order of the host that wrote the file. It is also used to detect the file format. Allows software reading the file to determine whether the byte order of the host that wrote the file is the same as the byte order of the host on which the file is being read, and thus whether the values in the per-file and per-packet headers need to be byte-swapped. Possible values are specified below.

Table 3 – Possible values for the magic number field in the global header of pcap format

Value	Description
0xa1b2c3d4	Reading application will either read 0xa1b2c3d4 (identical) or 0xd4c3b2a1 (swapped).
0xa1b23c4d	Used in files where nanosecond resolution is considered. Reading application will either read 0xa1b23c4d (identical) or 0x4d3cb2a1 (swapped).

version_major / version_minor Version number of the file format of the file being read (current version 2.4). The 2 being the major version number and the 4 being the minor version number.

thiszone Time, in seconds, that must be added or subtracted from the packet's timestamps to convert them to the GMT timezone.

For instance, if a packet's timestamp was labeled with the CET timezone which is GMT +1:00, then this field would be -3600 meaning that 3600 seconds would have to be subtracted from the timestamp for it to be in GMT timezone.

sigfigs In theory, it should be a value representative of the accuracy of the timestamp's in the capture. In practice, all tools set it to 0.

snaplen A libpcap capture file only contains the first N bytes (called the “snapshot length”) of each packet. In order to avoid having incomplete packets in a capture file, a value of 65535 or more is typically defined for N.

network Link-layer header type, specifying the type of headers at the beginning of the packet. Supported data link types available at [22].

2.1.2 Packet header

For each packet present in a PCAP file there is an accompanying header, with the format as depicted in

Table 4 - Fields in the packet header of pcap format

32 bit unsigned int	32 bit unsigned int	32 bit unsigned int	32 bit unsigned int
ts_sec	ts_usec	incl_len	orig_len

ts_sec Unix epoch time of when the packet was captured (will only resolve to date, hour, minute and second).

ts_usec Depending on the value of the magic_number field in the global header, it will either contain the microsecond when the packet was captured as an offset of ts_sec field or contain both the microsecond and the nanosecond when the packet was captured as an offset of the ts_sec field.

incl_len The number of bytes of packet data actually captured and saved in the file. Should never be larger than the value specified in the orig_len or the snaplen field in the global header.

orig_len The length of the packet as it appeared on the network when it was captured. If the values present in the incl_len and orig_len fields differ, the actually saved packet size was limited by the amount specified in the field snaplen of the global header.

The actual packet data will immediately follow the packet header as a data blob of incl_len bytes without a specific byte alignment.

2.2 Cyber attacks

In this section, the set of cyberattacks which anomaly-based IDS (ANIDS) are capable of detecting [23] is presented. This not only helps contextualize the problem better but also

reveals that even though an ANIDS stands as a great intrusion detection solution, it cannot protect against all threats.

Initially, ANIDS would rely on both header and payload information for the detection of intrusions [12]. However, as communications in the Internet became increasingly encrypted, payload analysis was rendered useless, making it difficult for ANIDS to detect attacks such as SQL injection, cross-site scripting (XSS), heartbleed, backdoors.

Other attacks such as phishing, insider threat or advanced persistent threats, which are carried out over a network, are not so easily detectable by an ANIDS, due to their more subtle nature, often ending up being misinterpreted as normal traffic [24].

This limits the scope of attacks which are effectively detected by an ANIDS to the ones presented in this section.

While an ANIDS alone might not be capable of detecting such attacks, network administrators might find success in mitigating such attacks by coupling ANIDS with individual host protection through the application of host-based IDS (HIDS) and antivirus software and the employment of additional security measures such as well-configured firewalls.

2.2.1 Botnet

A botnet [25] corresponds to a group of internet-connected machines, infected with malware which allows them to be remotely controlled and ordered to perform malicious actions against one or more targets, without the consent of the machine owner. Botnets will usually perform DDoS attacks, email spam campaigns, scanning, sniffing traffic and spread new malware [24]. The attacker communicates and delegates actions onto the bots through C&C servers, which use HTTP or IRC protocols [25].

Even though a botnet on its own does not pose a threat to a network, the misuse of a network's resources is reason enough to detect such illicit activity [25].

2.2.2 Denial of Service (DoS)

Denial of Service (DoS) [24] [26] is an attack on a computer or network that aims to reduce, restrict, or prevent legitimate use of its resources. A DoS attack will usually target a network's bandwidth, by overflowing it with a high volume of traffic in an attempt, impeding normal users from accessing its resources or will target a system's connectivity, by flooding it with a large number of non-legitimate requests, consuming all its available resources and rendering it incapable of processing the requests from legitimate users.

A DoS is executed over a single Internet connection to either exploit vulnerabilities in the implementation of TCP/IP protocols or a flaws in a specific OS or flood a target with fake requests.

2.2.3 Distributed Denial of Service (DDoS)

A DDoS attack [25] is a large-scale, coordinated attack launched against one or more targets. By using a collection of infected machines (botnet) and the employment of server/client technologies such as IRC, the attacker leverages the aggregate computing power of all machines and becomes able to conduct much more powerful attacks against the availability of services on a victim's system or network resources.

A DDoS attack has 2 possible outcomes: the targeted system remains operational although with significant less capability to respond to legitimate user requests, with an overall slower performance reduced throughput and higher latency. The alternative outcome is a complete unavailability of the system, which crashed due to the high computational exertion placed on its resources by a huge number of superfluous requests.

DDoS attacks can be classified by the type of resource it exhausts [24]:

Volumetric attacks Designed to exhaust a network or system's bandwidth and further hinder its capability to respond to legitimate user requests. These types of attack often target stateless protocols which do not offer built-in congestion avoidance which makes it easier to consume all bandwidth of target network.

Common volumetric attacks include UDP flood attack, ICMP flood attack, Ping of Death attack, Smurf attack, malformed IP packet flood attack and spoofed IP packet flood attack.

Protocol attacks Designed to disrupt services by filling connection state tables with connections which remain open for the duration of the attack, preventing new connections from being made until the existing ones are closed or expire. This attack can either target the victim directly or a network infrastructure device such as load-balancers, firewalls and application servers, between the victim and the internet.

Examples of protocol attacks include SYN flood, ACK flood, TCP connection flood, TCP state exhaustion, Fragmentation attack, DNS amplification, NTP amplification and RST attack.

Application attacks Exploit the vulnerabilities in either an application layer protocol or in the application itself, often web applications, with the goal of overwhelming the target with an abnormal number of requests, causing the application to crash, ultimately resulting in an inability on the part of legitimate users from accessing it. One form this attack can take is the

repeated invalid login attempts to lock the user out of his own account or the inability to respond to legitimate requests due to an overwhelming amount of GET or POST requests received by a REST web service. The magnitude of these types of attacks is measured in terms of requests per second.

Buffer overflow, HTTP flood and slowloris attacks are two attacks that fall into this category.

2.2.4 Brute force

Brute force attack [27] is a means of gaining unauthorized access to a system or service by repeatedly trying different combinations of username-password until a valid one is found.

These types of attacks can either be executed manually, with resource to scripts or application which automate the process and can try numerous combinations in parallel or they can be carried out with minimal human intervention, through the use of botnets.

Brute force attack can take various forms [27]:

Simple brute force attack Method where an attacker tries all possible combinations of characters, using all available characters, for a certain password length. This method, while more exhaustive than others, will only succeed if the password has a lower or equal number of characters than those defined as the limit by the attacker. It cannot succeed if the password of the username which the attacker is trying to guess has a high number of characters.

Dictionary brute force attack Method which relies on a predefined list of strings which the attacker then uses in combination with the desired username to guess the correct credentials. This list usually consists of passwords which have been leaked in one or more security breaches or a compilation of the passwords most commonly found in various security breaches.

Reverse brute force attack A technique where the attacker, instead of trying various passwords for the same username, performs the inverse operation and tries different usernames for the same password. By choosing a password which is very commonly used (such as “123456”, “123456789”, “qwerty” or “password”) [28], the attacker might find more success.

Common targets of brute force attacks are SSH and web services with login pages. Brute force attacks will usually produce a very clear fingerprint on the network, making them very easy to identify.

Brute force attacks can also by performed offline, to crack password hashes or API keys, by reversing the encryption to gain access to the raw data.

2.2.5 Scanning

Scanning [24] is the process of gathering more detailed information on a target, through the execution of complex and aggressive reconnaissance techniques. This process is particularly important for an attacker as it helps him profile its victim, learn more about possible entry points into the victim's system and any configuration lapses and vulnerabilities it might present. While the process of scanning is not an attack per se, it is the step which precedes an attack.

There are three types of scanning [24]:

Port scanning Results in a list of open ports and the services running on each respective port. It involves connecting to or probing TCP and UDP ports on the target system to determine if the services are running or are in a listening state. A service in a listening state often responds back to requests with information about the system's OS and the application (including its version) active on that port, effectively revealing important information about the system's architecture. Types of port scanning include TCP Connect/Full Open Scan, where port scanning is achieved through an attempt to establish the three-way handshake with an active TCP port which in case of success, is immediately followed by a RST packet to end the connection. Ports where the three-way handshake is successfully established correspond to open ports; SYN scan attempts to determine if a TCP port is open by partially opening a connection but stopping halfway through. The attacker sends a single SYN packet and depending on the response from the target system, the attacker can infer if the port is opened or closed; Xmas Scan; FIN Scan; NULL Scan; ACK Scan; IDLE scan; UDP Scan.

Network scanning Results in a list of IP addresses. Network scanning is a procedure for identifying active hosts on a network. Types of network scanning include ICMP scanning which involves sending an ICMP ECHO request to a host. If the host is alive, it will return an ICMP ECHO reply. This scan is useful for locating active devices or determining if ICMP is passing through a firewall; Ping sweep involves sending multiple ICMP ECHO requests to a range of IP addresses in simultaneous, to identify which ones respond and signal an active host on the network and which ones do not. A ping sweep can either be accomplished by sending individual ICMP ECHO requests to various IP addresses or by sending a single ICMP ECHO request to the broadcast address.

Vulnerability scanning Identifies weaknesses and vulnerabilities on a system, to assess if it can be exploited or not. This process will usually be conducted by a vulnerability scanner, equipped with a scanning engine and a catalog. The catalog includes a list of common

files with known vulnerabilities and common exploits for a range of servers. The scanning engine maintains logic for reading the exploit list.

2.3 Intrusion Detection System

In order to better understand the problem being tackled, it is important to define an Intrusion Detection System and clarify what it represents.

An IDS [29] [30] is a software or hardware system which continuously monitors a network entity for any anomalous activity which might infringe its confidentiality, integrity or availability. Rather than operating on its own, it complements other security systems such as firewalls or honeypots in the detection of attacks and anomalous behavior.

An IDS is generally categorized into on of two groups: signature-based IDS or anomaly-based IDS [11] [29] [30].

A signature-based IDS [12] [29] [30] (also commonly referred to as misuse-based or knowledge-based) detection is heavily reliant on pattern. During monitoring, the SIDS attempts to match the signature obtained from the current activity with signatures of known attacks. If a match is established, the system promptly raises an alarm. While SIDS excel at the detection of previously known intrusions, often reporting very high detection rates, they present some issues which make them an increasingly less viable option to integrate a computer system's line of defense. Firstly, due to the heavy reliance on previously collected knowledge of attacks to perform intrusion detection, a SIDS cannot detect novel attacks until its database is updated with the attack's signature. Secondly, SIDS are particularly weak to polymorphic and metamorphic malware and to encryption. Even small modifications to an intrusion which had been previously detected by a SIDS, such as the frequency of the space character, will force the creation of a new signature [29]. On top of that, SIDS require a constant update of its signature database to reflect current attack trends. This is a very laborious task, involving the analysis of the intrusion itself and the creation of rules by a security expert.

An anomaly-based IDS [12] [29] [30], on the other hand, is mainly built around a normal model using statistical-based, knowledge-based and machine learning techniques [29], which attempts to encapsulate the typical/normal behavior of the infrastructure element in question. Then, during monitoring, any behavior which significantly deviates from this model, is flagged as an anomaly. This alternative is much more suited to detect zero days attacks and has the advantage of being able to also detect known attacks with equally high accuracy.

However, it presents some challenges. Since no model can fully encompass all normal or valid behavior, AIDS often experience high false positive rate when confronted with new but non-malicious behavior.

An IDS can be further categorized into host-based or network-based IDS [11] [29] [30] depending on the environment it monitors and the kind of data it analyzes for the detection of abnormal behavior.

Host-based IDS (HIDS) [29] [30] monitor the activity of individual hosts and perform intrusion detection by analyzing its system calls logs such as those generated by the OS, anti-virus or firewall. Unlike a NIDS who only has access to streams of data, a HIDS has more context on the data being transferred between two hosts, it is not affected by end-to-end encryption communications and has access to reassembled packets, effectively mitigating evasion attempts in the form of fragmentation. HIDS, however, can only provide intrusion detection functionalities to the host where it is installed and since it runs as a daemon, it consumes part of the resources available to the host, potentially having a negative impact on its performance.

Network-based IDS (NIDS) [29] [30], on the other hand, monitor the traffic of a network, receiving as input packets, NetFlow records or other variants of session data such as IPFIX, sFlow. A NIDS operates on a network level so it can protect multiple hosts at the same time, regardless of their OS and without the need for additional configuration on their part. However, they present some limitations: NIDS were purposely designed to defend networks and threats against it, with the need for additional security measures to protect against other types of attacks. In the high bandwidth networks of today, where high-speed communication is the norm, NIDS are not able to inspect all data passing through, oftentimes having to resort to sampling.

A traditional anomaly-based NIDS architecture consists of the components as presented in Figure 4.

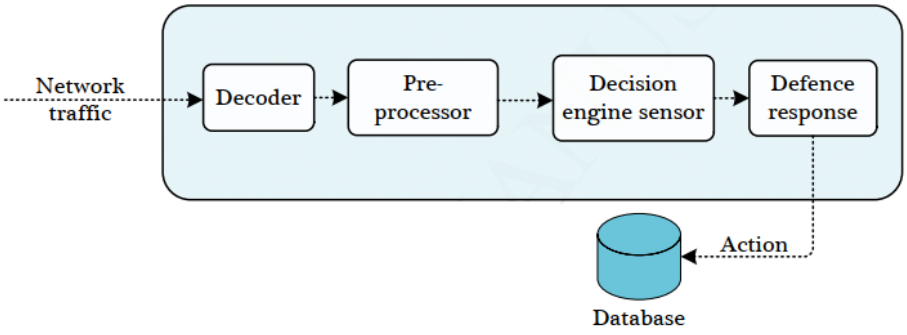


Figure 4 - Architecture of a NIDS [30]

First, there is the decoder, which will either collect packets on the wire or receive previous packet captures or flow records as input. This data will then be transferred to the pre-processor which will take the data in its original state and extract a set of the most relevant features (either directly or by calculating them) for intrusion detection. The decision engine receives the feature-rich data and performs intrusion detection based on a previously constructed model depicting normal behavior. Depending on the result of the analysis, the defense response component will then raise an alert and send it to a security administrator for manual approval.

2.4 Network traffic

Network traffic is usually captured in either a packet-based or flow-based form. In order to define what features will be contemplated in the final output file, produced by the application, a good understanding of what kind of information is available is important.

2.4.1 Packets

The packet is the most basic unit of network traffic data[31]. Every communication made over the Internet will involve the exchange of at least one packet [31], between any two IP nodes and thus constitutes one way of representing network traffic data. A set of packets is then able of reflecting the full communications between nodes on a network, for the timeframe in which packet collection took place.

Each packet is further divided into several datagrams, each one related to one protocol of one layer of the OSI model [31].

For this work, we will only be concerned with traffic sent over Ethernet technologies [31]. Under the perspective of the OSI model [31], traffic sent over any other type of layer 2 technology will be discarded as it does not fall under the scope of the proposed work.

Due to the close nature of operation between the physical and the data-link layers, the imposed restrictions on the layer 1 technologies will reflect on the layer 2 technologies to be considered for analysis as well. When analyzing packets at the layer 2 level, only Ethernet II frames will be considered. Other types of IEEE standard layer 2 frames such as the Novell raw IEEE 802.3 non-standard variation frame, the IEEE 802.2 Logical Link Control (LLC) frame or the IEEE 802.2 Subnetwork Access Protocol (SNAP) frame [31] will not be considered due to a lack of acceptance by the vendors, resulting in a very low number of packets featuring them and general lack of support of some layer 3 protocols and operations [32].

The following are the specifications of the protocols belonging to the data-link layer of the OSI model which are supported in the final application.

Ethernet 802.3 packet w/ Ethernet II frame [33] Since only traffic transported over an Ethernet physical layer will be considered, the traffic analysis starts with the decapsulation of the structure known as “ethernet packet” . This packet is the overall transmission unit of the “ethernet frame” (also known as the data link layer protocol data unit). The “ethernet packet” has the structure as represented in Table 5.

Table 5 - "Ethernet packet" format

7 bytes	1 byte	64 – 1522 bytes	12 bytes
Preamble	Start of frame delimiter	Ethernet II frame	Interpacket gap

As mentioned above, only Ethernet II type of frames will be analyzed.

An ethernet II frame consists of the fields as presented in Table 6.

Table 6 - Ethernet II frame format

6 bytes	6 bytes	2 bytes	46-1500 bytes	4 bytes
MAC destination	MAC source	Ethertype	Payload	Frame check sequence (32-bit CRC)

The Ethertype field identifies the layer 3 protocol featured in the payload. All possible values for this field can be found in [34].

The frame check sequence field is for checking the integrity of the frame data and to confirm that its content was not tampered with while being transmitted or that the data did not become corrupt.

Up next are the specifications of the protocols belonging to the network layer of the OSI model which are supported in the final application.

Internet Protocol v4 The Internet Protocol v4 [31] [35] is a layer 3 protocol which defines the requirements and necessary specifications for the routing of packets between different networks, allowing for internetwork communication. Even though IPv4 still makes up for a large part of daily communications happening over the Internet, it poses a few limitations and so, a new version of the IP protocol, IPv6, was devised to address its shortcomings and is set to replace it over the coming years. In IPv4, network hosts are identified by a unique 32-bit address. The IPv4 packet consists of two sections: a header, which

is visually represented in Table 7 and a payload. This protocol does not implement payload checksum verification, relaying that responsibility to the transport layers.

Table 7 - IPv4 packet header format

	1 byte								2 bytes								3 bytes								4 bytes							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
4 bytes	Version				IHL				DSCP						ECN		Total Length															
8 bytes	Identification																Flags			Fragment Offset											
12 bytes	Time To Live								Protocol								Header Checksum															
16 bytes	Source IP Address																															
20 bytes	Destination IP Address																															
24 bytes	Options (if IHL > 5)																															
28 bytes																																
... bytes																																

The “version” field in IPv4 is always “4”.

The “Internet Header Length (IHL)” field specifies the header length in terms of 32-bit words. The minimum value this field can have is 5 (“Options” field not defined). Given that the maximum decimal value this field can hold is 15, an IPv4 packet header has, at most, 15 * 32 bits or 15 * 4 bytes which is equal to 60 bytes.

The “Differentiated Services Code Point (DSCP)” field defines the accompanying traffic priority, by classifying traffic into classes. This traffic management mechanism was defined in RFC 2474. One example of a service which is directly influenced by this field is VoIP.

The “Explicit Congestion Notification (ECN)” is a field which allows for the communication of network congestion between two communicating nodes, without the need to drop of packets. Must be supported by both nodes’ networks.

The “Total Length” specifies the entire packet size in bytes. Minimum packet size is 20 bytes (only the header with no options field and no payload) and the maximum packet size is 65,535 bytes.

The “Identification” field is used to identify fragments of the same split IP packet.

The “Flags” field is related to packet fragmentation process. Bit #1 is reserved and must always be 0. When set, bit #2 represents the “Don’t Fragment (DF)” flag and bit #3 represents the “More fragments (MF)” flag.

The “Fragment Offset” field measures blocks of 8 bytes. It indicates the offset of a fragment relatively to the first fragment of the complete IPv4 packet. If each fragment has the size of

24 bytes, the offset of the first fragment will be 0, the fragment of the second fragment will be 24 bytes, which would be equivalent to the decimal number “3” and the binary “10”. The maximum fragment offset is (213 -1) * 8 bytes = 65,528 bytes.

In theory, the ”Time To Live (TTL)” field indicates the maximum time, in seconds, a packet can travel over an internet. In practice, this field indicates the maximum number of hops the packet can make between routers, as it is routed to the destination. Every time the packet passes through a router, it decrements this field by one. Once it reaches 0, the router discards the packet and sends an ICMP Time Exceed message to the sender.

The” Protocol” field determines the protocol present in the payload. The numbers pertaining to each protocol defined by IANA can be found in [34].

The “Header checksum” field is calculated before the packet is sent. Routers which receive an IPv4 packet will calculate the header checksum to verify the integrity of the data. If the checksums do not match, the packet gets discarded. To note that there is no payload checksum.

The “Source address” field holds the sender IPv4 address.

The “Destination address” field holds the destination IPv4 address.

The last field is the “Options” field which specifies additional packet information which could not be specified anywhere else. This field is optional and is always terminated by an “End of Option List” option if it has at least 1 bit set. The “Options” field is further divided into 3 subfields:

- “Option value” subfield (1 byte), according to the IPv4 protocol definition, is further composed of 3 other fields. However, over time, these 3 fields have been interpreted as one single field. Values which this subfield can hold can be found in [36].
- “Option length” subfield (1 byte) specifies the length of the whole “Options” field (including itself).
- “Option data” subfield (variable size) holds data specific to a certain value of the “Option value” subfield. Padding might be introduced in this field so that the packet size obeys the restriction imposed by the IHL field which defines the header size in terms of 32 bit words.

Now come the specifications of the protocols belonging to the transport layer of the OSI model which are supported in the final application.

TCP TCP [31] [37] is a connection-oriented transport layer protocol which allows for the transmission of data between two nodes operating over IP in a reliable and secure way through the establishment of a communication channel prior to the trade of information, featuring error-checking mechanisms. A TCP header follows the format presented in Table 8.

Table 8 - TCP segment header format

	1 byte								2 bytes								3 bytes								4 bytes							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
4 bytes	Source port																Destination port															
8 bytes	Sequence number																															
12 bytes	Acknowledgment number (if ACK bit set)																															
16 bytes	Data offset				Reserved				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window size														
20 bytes	Checksum																								Urgent pointer (if URG bit set)							
24 bytes	Options (if "data offset" > 5)																															
28 bytes																																
... bytes																																

The “Source port” field indicates the port from where the packet was sent.

The “Destination port” field indicates the port to where the packet is being sent to.

The “Sequence number” and “Acknowledgement number” fields are numbers set by TCP-enabled nodes which ensure a connection’s security and reliability.

The “Data offset” field specifies the size of the TCP header in terms of 32-bit words. The minimum value this field can have is 5 (“Options” field not defined) which means that the minimum length a TCP header can have is 5 * 32 bits or 5 * 4 bytes which is equal to 20 bytes. Given that the maximum decimal value this field can hold is 15, a TCP segment header has, at most, 15 * 32 bits or 15 * 4 bytes which is equal to 60 bytes.

The ”Reserved” field is always set to 0.

The “Flags” fields is a 9 bit field where each bit is associated to a different flag.

- “NS” flag: ECN-nonce
- “CWR” flag: Acronym for “Congestion Window Reduced”; Flag which is set by a sending node if it previously received a TCP segment with the ECE flag.
- “ECE” flag: If the SYN flag is set, it indicates that the sending node is ECN capable. If the SYN flag is not set, it indicates that a packet with the ECN field in the IP header was received during normal transmission.
- “URG” flag: If set, it indicates that the data present in the “Urgent Pointer” field is relevant.

- “ACK” flag: If set, it indicates that the data present in the “Acknowledgement number” field is relevant.
- “PSH” flag: If set, sender is requesting that buffered data gets pushed to the receiving application.
- “RST” flag: If set, sender requests a connection reset.
- “SYN” flag: Synchronize sequence numbers. The first TCP segment sent between each of the two nodes attempting a TCP connection must have this flag set.
- “FIN” flag: If set, signalizes the end of a connection.

The “Window size” field specifies the “receive window” size of the sender, which indicates the number of bytes the sender can receive from the other end, before acknowledging the received data.

The “Checksum” field, which holds a checksum value, is used to match against the calculated checksum on the receiver end for detecting tampering and unintended modification of the sent data.

The “Urgent pointer” field, indicates, via a byte offset, where the urgent data ends in a TCP segment.

The “Options” field is divided into 3 subfields, much like the IPv4 protocol:

- “Option kind” subfield can hold different values, which can be consulted in [38].
- “Option length” field represents the length of one option (includes the size of the previous, current and the next subfield)
- “Option data” field is optional, and it holds a value relative to whatever property the “Option kind” subfield refers to. Padding might be introduced in this field so that the packet size obeys the restriction imposed by the “Data offset” field which defines the header size in terms of 32-bit words.

UDP UDP [31] [39] is a stateless, message-oriented transport layer protocol which means that UDP cannot assure the packets have arrived their destination or that they were received in the desired order, since no connection is established between two nodes. An UDP segment is composed of the field, as presented in Table 9.

Table 9 - UDP segment header format

	1 byte								2 bytes								3 bytes								4 bytes							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
4 bytes	Source port																Destination port															
8 bytes	Length																Checksum															

The “Source port” field indicates the port from where the packet was sent from.

The “Destination field” indicates the port where the packet is being sent to

The “Length” field specifies the total length of the UDP segment (header and payload), in bytes.

The “Checksum” field, which holds a checksum value, is used to match against the calculated checksum on the receiver end for detecting tampering and unintended modification of the sent data.

No protocols from the OSI layers 5 or above are considered since the information present in the headers no longer reflect a network state and do not carry any particularly useful information for network-based intrusion detection. Adding to this, as network traffic gets increasingly encrypted, no meaningful information for intrusion detection can be obtained from the payload [12] and so, there are no reasons to consider these kinds of protocols.

2.4.2 Flows

A flow is defined as a sequence of packets with some common properties that pass through a network device [40]. These common properties may include packet header fields, such as source and destination IP addresses and port numbers, packet contents, and metainformation [41] which all together form the “flow key”. Traditionally, a flow is uniquely identified as the combination of the following 5 IP packet attributes: source IP address, source port, destination IP address, destination port and layer 3 protocol [42]. Additionally, another two attributes can also be considered, namely, IP Type of Service and router or switch interface ID on which the traffic was entering the device [42].

Flows are usually characterized by other attributes, usually statistics such as the number of packets sent in forward direction, packet inter arrival time, bytes per second sent in backward direction, among others. With that said, a flow’s main objective is to summarize communications between sources and destinations on a network, while providing valuable information in the form of network traffic metadata [43].

Network flows can be sampled, meaning that not all observed traffic will be evaluated, just every n^{th} packet. The higher the sample rate is, the lesser the processing burden placed on the exporter and the lesser bandwidth is consumed to the send the collected data to the collector and, consequently, the lesser storage space is required [43]. Sampling can have a negative impact on intrusion detection and can lead to entire activities conducted by the attacker will not be detected and so must be done with caution [43].

A flow can be classified as either unidirectional or bidirectional.

A unidirectional flow represents the unilateral interaction between two nodes, aggregating only the packets sent from one node to the other. When attempting to characterize network traffic data this way, two unidirectional flows will be constructed for each two communicating nodes.

Bidirectional flows represent the full interaction between two nodes, aggregating all packets exchanged between them, in a certain timeframe. When attempting to characterize network traffic data this way, one bidirectional flow will be created for each two communicating nodes.

Flow creation process [42] starts in a network device, either a router or a switch with flow monitoring technology, usually called a probe or flow exporter. When a packet arrives at the device, the flow key is extracted and one of two outcomes can follow: if the packet's tuple matches that of any existing flow, the packet is grouped with that flow, followed by a recalculation of the features pertaining to the respective flow record, in order to reflect the new packet's data. If no flow exists with a matching tuple, then one is created.

Initially, flows will reside in the device's flow cache. Each entry in the flow cache corresponds to a flow record, with one or more fields (e.g. no. bytes sent, no. packets sent) other than the field pertaining to the flow key. Flows can be in one of three states: active, terminated or expired. A flow containing only layer 3 TCP packets is terminated once a packet matching the flow's tuple with the FIN or RST flag set is received. Additionally, a flow is considered to have expired if no packets matching the flow's tuple have been received in the past 15 seconds or if the flow has been active for longer than 30 minutes (e.g. large FTP download). These are the default time values defined by many vendors though they can be tuned to the network's administrator preference.

The flow cache is periodically checked for any flows which have terminated or have expired and if any are found, they are promptly bundled together and exported to a flow collector, located in the network, usually over UDP. The flow collector is then responsible for assembling and understanding the exported flows, filtering and aggregating them to produce the valuable

information regarding the network's users and application behavior in a user-friendly way. The network metrics obtained this way can then be applied towards traffic accounting, network planning, and attack monitoring purposes.

The flow creation process is summarized in Figure 5.

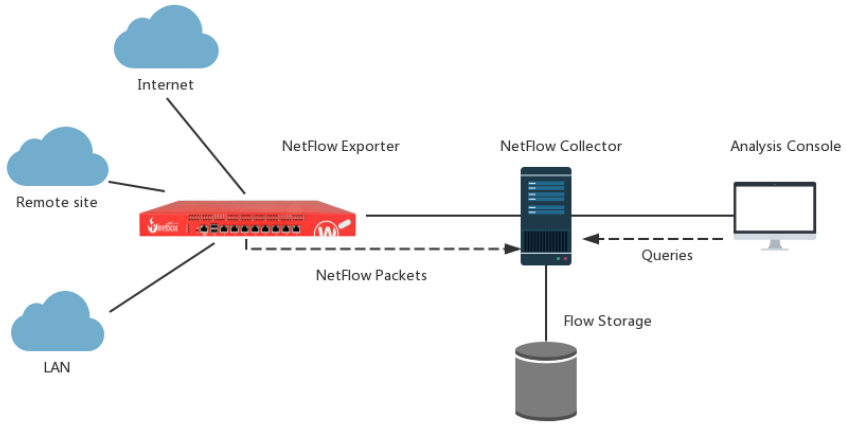


Figure 5 - Topology of the flow creation process [44]

This flow creation process and respective architecture is supported and employed by various network device vendors [41] such as Cisco, D-Link, Fortinet, Juniper Technologies. Each vendor might, however, support different flow export formats such as NetFlow export version family of formats, IPFIX or sFlow. These are the formats in which the flow records get sent from the flow cache to the flow collector. A discussion of three different formats is found further ahead in this section.

Vendors do not usually offer flow collector solutions so customers have to resort to various commercial and open-source flow collector available in the market. Popular commercial tools include ManageEngine's NetFlow analyzer, SolarWinds' NetFlow traffic analyzer, Caligare's Flow Inspector, Cisco's Stealthwatch, Plixer's Scrutinizer and Paessler's PRTG Network Monitor. Other open source alternatives are nfdump alongside nfsen and ntopng. These tools will often support one or more flow export formats, so that they are compatible with the flow creation solutions offered by the different network device vendors.

Besides the hardware-based architecture for flow gathering, there are various software alternatives for producing flow data from packet data (e.g. pcap files). While some tools produce flow data form packet data in an offline manner, meaning that packet data must be first captured, other tools can be integrated with anomaly-based IDS to generate flows from packet data, as it travels the wire. While this could be possible to implement, it would not be

very feasible, as having a router generate the flow data considerably lowers the strain that would be placed on the IDS, especially important in the high velocity, high bandwidth networks of today [12]. These tools are further discussed in another section ahead. Some of these tools will also offer functionalities for constructing bidirectional flows from packet data or from unidirectional flows, which aggregates packets sent in either direction of a connection session into a single bidirectional flow record.

Popular flow formats include NetFlow family of versions, IPFIX, sFlow. No further specification about these formats is done as their integration with the application was out of this scope, though future work could improve on this and support them.

2.4.3 Comparison

While a collection of packets exchanged between two nodes represents the full interaction between the two, for a certain timeframe, flows support better traffic characterization, giving a more granular understanding of how bandwidth is being used. They give a more in-depth insight into the communicative behavior of two hosts on a network through high-level features, which are not readily available at a packet level. Flows are also more storage efficient. Since they only contain statistical information and no payloads, they will span over a larger period of time and still use less storage space than packet data.

Packet inspection techniques might be too resource intensive for real time operation, a problem which is aggravated as networks become faster and more bandwidth is used. Packet header processing coupled with aggregation into flows is a more lightweight alternative to full packet analysis, while maintaining the same level of detection quality.

Since flows do not take payloads into consideration, they are less susceptible to privacy concerns and they require less efforts to anonymize, making them more readily available for manipulation than the packets counterpart. One piece of personally identifiable information which can be common to both types of network traffic data are the IPs of source and destination.

The context offered by flows cannot be understated. Unlike traffic in packet format where each packet exists on its own, flows are more dynamic and able to characterize traffic over time, providing a wider insight into the communication between two nodes.

Adding to that, traffic over the last years has become increasingly encrypted, rendering payload analysis useless, making flows a more appealing alternative since the information which characterizes them is provided by packet headers which are not encrypted.

Overall, flow format is more suited for intrusion detection than packet format.

A summary of the comparison between the packet and flow types of network traffic data can be found in Table 1.

Table 10 - Packets and flows comparasion summary

	Packets	**Flows**
Where does data reside	Key-value pairs (according to the protocol's specification)	Feature vectors
Data level of abstraction	Low-level	High-level
Storage requirements	Very high	Low
Processing power requirements	Very high (packet inspection is very resource intensive)	Low (packet header analysis + aggregation into flows)
Privacy concerns	Very high (packets featuring unencrypted protocols such as HTTP + IPs)	Minimal (IPs)
Encryption concerns	Very high (renders payload analysis useless)	None (analysis based on traffic statistics)

3 State of the art

The following section depicts the current state of the existing knowledge directly related to the problem domain. The section can be divided into two parts.

In the first part, a summary of related works is presented. These are works which have been published over the past decade and represent, as much as possible, the existing knowledge on the different subjects relevant to the problem. In its complete form, it represents the knowledge base over which the solution will be built.

The second part focuses on the study of existing technologies and a comparative analysis between different candidates to integrate the final solution. It provided an in-depth comparison between technologies and a justification of why one was preferred over the others.

3.1 Related work

Initially, a study of the most relevant features is performed. IDS base themselves on feature data to perform the actual intrusion detection so it is of highest importance that meaningful, representative features which are able to accurately reflect the presence of either malicious or benign behavior, are chosen to represent each flow record in a IDS's input data. To that end, numerous works which have studied the same problem are presented and several benchmark datasets, used in the training and evaluation of intrusion detection, have had their features analyzed.

3.1.1 Feature study

Many authors have expressed their concerns over the importance of feature study and how a proper identification of the most relevant features for intrusion detection can have a positive impact on both the efficacy and the time efficiency of a detection algorithm [12] [45] [46].

A difficulty by intrusion detection methods of today in accurately discriminating between normal and malicious behavior, gives more strength to the study of relevant features which clearly reflect the presence of normal behavior or attack behavior in the network.

While there are many tools capable of extracting features from network traffic data, it is important to understand why such features are extracted and what makes them so relevant for the detection of certain attacks, through the application of domain knowledge.

Moustafa et al [30] note that one of the most significant challenges when designing a NADS is determining the threshold which intrusion detection methods use to classify observed network behavior as either benign or malicious. This challenge rises from the overlapping patterns in the two behaviors, caused by a selection of poor network features which cannot reflect any variations between normal and abnormal patterns. The authors also mention the use of Snort IDS, Bro IDS, Argus and netmate as ways of extracting features from network packet data. They also highlight the advantages of defining additional features involving transactional flow identifiers and transactional connection times, the latter having been successful in the detection of attacks which stretch over time. All in all, the authors believe it is "very essential to select only significant features that can discriminate between normal and abnormal observations" though they themselves do not perform any study to that end.

Davis and Clark [12] present a comprehensive study of anomaly-based NIDS solutions, publicized between 1999 and 2010. The study mainly focused on the preprocessing phase of anomaly detection, motivated by the large impact data it has on the accuracy and capability of anomaly-based NIDS.

The authors address the following concerns: what preprocessing techniques were used (e.g. parsing individual network packet headers, organizing packets into flows with Netflow or tcptrace, calculating statistics for header values over a time window, parsing application protocols, or analyzing application content for fields of interest), which aspects of the network traffic were analyzed for feature construction, what actual feature construction and feature selection methods were employed to create additional, more discriminative features from the basic traffic data and to reduce the dimensionality of the dataset, respectively, and what actual features were derived from the network traffic to be later analyzed by the anomaly-based NIDS.

From their study, 5 types of feature sets could be identified, mainly, features obtained from network packet headers, network protocol information, the KDD Cup 99 dataset, network packet contents (payloads), and NIDS alerts. The authors noticed that NIDS sharing the same feature type, would have similar detection capabilities, indicating that the choice of feature types has an influence in the capabilities and coverage of the detector, i.e. what kind of attacks it can detect. Depending on the detection requirements,

Network packet headers feature types were further subdivided into:

Packet header basic features, which correspond to features directly taken from individual packet headers with no additional feature construction. However, some authors have advised against their usage for anomaly detection as it can lead to an inaccurate classifier.

Single connection derived (SCD) features are not obtained from individual packets but rather complete network flows, which represent a unidirectional connection between two hosts. The authors note that "the most common and important SCD features are time based statistical measures". This feature type is particularly useful for finding abnormal activity within a single session such as an unexpected protocol, unusual data sizes, unusual packet timing, or unusual TCP flag sequences. Attacks which can be discovered using these types of features include "backdoors, HTTP tunnels, stepping stones, BIND attacks, and command and control channels".

Multiple connection derived (MCD) features are constructed through the monitorization of packet header basic features or SCD over multiple flows or connections. This type of features is more suited for the detection of intrusions which produce unusual patterns of network traffic such as network probes, worms, DoS and DDoS attacks.

The authors presented the other feature types in detail, though they fall out of the scope of the project.

While the preprocessing stage of many NIDS is achieved by entirely relying on expert domain knowledge, it can be further enhanced with automated data reduction techniques such as PCA, leading to noticeable reduction in the computational requirements of the NIDS.

Even though the authors are very adamant about the analysis of payloads and consequent construction of relevant content-based features for the detection of attacks against web servers, web application exploits, and attacks targeting web clients such as drive-by-downloads and indicate it as an interesting field to follow, contemporary network traffic trends point to an increasingly encrypted traffic, rendering payload analysis useless and an uneventful task.

A compilation of all the features, organized by feature type, found by the authors in the papers which they reviewed, can be consulted in Annex B.1.

Stevanovic and Pedersen [47] study the application of network traffic classification for the detection of botnet network activity. The authors propose the construction of 3 detection methods, each one in charge of analyzing network traffic depicting one of the 3 most used transport layer protocols for conducting botnet C&C communication and attack: TCP, UDP and

DNS. Due to the different natures and mode of operation of each protocol, botnet activity will manifest itself slightly different over each protocol thus, having a separate approach for each protocol improves classification accuracy. Each detection method relies on a Random Forest classifier, a supervised machine learning algorithm, for the classification of traffic as botnet or benign. Before analysis can take place, network traffic in packet format is preprocessed to reflect bidirectional conversations between two hosts with "source IP address, source port, destination IP address, destination port, protocol" as each data instance's key. Further, a set of features are extracted from the conversations, based on a time window-basis, for each of the 3 transport layer protocols. The same set of features is shared by TCP traffic and UDP traffic. DNS traffic is characterized by a different set of features due to is different nature of operation.

Features extracted for UDP and TCP conversations, can be found in Annex B.2.

The authors refrained from using IP addresses as a feature in order to avoid a violation of privacy and "over optimistic classification performances".

Different types of features will help identify different types of attacks conducted by a botnet. Basic conversation features identify unusual ports associated with P2P and brute force attacks through the number of packets and their size. Time-based features will help detect brute-force attacks and periodicity of botnet traffic. Bidirectional features highlight one-sided conversations, which might be a sign of attacks being executed by bots (on a botnet) and communication between botmaster and bots. TCP specific features are able to characterize botnet communication and detect TCP-based brute force attacks such as SYN floods, SYN ACK floods, ACK floods and ACK PUSH floods.

Features extracted for DNS traffic, which comprises DNS queries/responses for queried FQDNs, can also be found in Annex B.2.

FQDN-based features can detect situations where an "unusual" FQDN, such as pseudorandom domains that characterize the Domain-Flux (DGA), have been queried. Query-based features describe the process of querying a FQDN. Response-based features identify any irregularities in the characteristics of query responses. Geographical location features are related to the IPs resolved for FQDNs and can indicate if the IPs are hosted over a high number of countries or Autonomous Systems (ASs).

The evaluation of the 3 detection approaches was achieved using several publicly available, non-malicious and malicious datasets in packet format.

The evaluation concluded that, for a time window of 300 s and for a number of processed packets per conversation of 1000, the TCP detection approach was able to classify malicious traffic with a precision and recall higher than 0.99 and 0.98, respectively, and classify benign traffic with a precision and recall values higher than 0.995. The Random Forest algorithm used for classification took approximately 220 seconds to be trained and to evaluate the traffic. The number of processed packets per conversation was highlighted as the most influential aspect on the classifier's performance though the authors also saw a boost in performance by increasing the length of the time window, albeit of lesser magnitude.

For the UDP detection approach, a time window of 3600s and for any of the considered number of processed packets per conversation (10, 100, 1000, 10000 packets), resulted in a recall and a precision values of approximately 0.99 when classifying attack traffic and a recall value higher than 0.995 and a precision value higher than 0.999 when classifying benign traffic. The classifier took less than 40 s to train and classify traffic, which can be attributed to the lower number of UDP traffic traces in the datasets used. Time window was found to be the most influential factor in the detection method's performance while it remained constant for a varying number of packets per conversation.

The DNS classifier was able to achieve a recall and a precision higher than 0.98 for both the classification of malicious and non-malicious traffic. These results remained the approximately the same for any of the values being considered for time window (300, 600, 1800, 3600 s). The classifier required less than 50 s to train and classify the traffic.

These results prove the efficacy of the chosen features and the impact a proper preprocessing of the data can have on a detection approach's performance.

Beer and Bühler [48] studied the use of network flow data for scalable intrusion detection and the viability of NetFlow/IPFIX-derived features for the detection of a broad range of attacks, including classic attacks such as spoofing and man-in-the-middle and more recent threats such as brute-force and botnet. Initially, the authors defined 42 features using expert domain knowledge, which can be found in Annex B.3.

There are 3 types of features: intrinsic (ID 1 to 14), content (ID 15 to 25) and traffic features (ID 26 to 41). Features with ID 18 through 25 are unique as they are related to information obtained from log events produced by hosts on the network where the dataset might be constructed.

Two datasets were constructed from packet data which was converted to bidirectional flow data using YAF. DS-1 dataset was custom-built and contains both benign traffic obtained from

the backend of a German enterprise and malicious traffic which reflect probing and spoofing attempts, SQL injections, dictionary attacks, cross site scripting, malware infections, variants of distributed denial-of-service attacks executed by the “Citadel” botnet and other vulnerabilities such as CVE-2015-2509 or CVE-2015-5122. Attacks without specific context were also conducted using DHCP starvations, DNS amplifications, and UDP floods. DS-2 was constructed from a mix of network traces from the ISCX and the CTU-13 dataset, containing floodings, bruteforce, injection attacks and botnet behavior. The datasets were respectively converted from packet-format into bidirectional flows. DS-2 is never characterized by features with ID 18 through 25 as they relate to information obtained from host log files which was not available for any of the 2 preexisting datasets used to build it.

DS-1 dataset was subject to a feature selection process based of Rough Set Theory. From this process, resulted 5 features subsets, each one containing a group of the most significant features, from a mathematical standpoint, for the detection of the attacks depicted in DS-1 alone. The process also revealed a set of features which are common to all 5 subsets, called “core” features due to their repeated presence in each one of them. Features 18-25 were not considered for the process of feature selection and thus are not represented in the analysis performed by the authors. These features can also be found in Annex B.3.

Both DS-1 and DS-2 datasets were then used to study the applicability of the features for the detection of attacks in different network traces, obtained in different conditions and depicting different attack vectors.

The authors preprocessed the 2 datasets, discretizing continuous features.

The 5 feature subsets, also called reducts, and their respective capability to significantly represent attack and benign behavior, were tested against 3 ML algorithms, namely, Hidden Naïve Bayes (HNB), decision tree ID3 and k-nearest neighbor (KNN) with k = 3. The Mathews Correlation Coefficient was used to evaluate the performance of each algorithm. This coefficient can take values between -1 and 1, which express no agreement or perfect agreement between prediction and observation obtained by ML model and selected features, respectively.

The results are the following: When performing classification on DS-1 without considering the features 18 through 25, KNN and ID3 classifiers obtained, on average, an MCC value of 0.981, for all 5 reducts and HNB averaged an MCC value of 0.975 in the same conditions.

The evaluation of the five feature sets against DS-2 gave slightly different results with respect to the performance for each learner. k-NN and ID3 managed to obtain an averaged MCC of 0.981 and 0.987, respectively, while HNB averaged MCC value was 0.973.

To better understand the impact that log event data can have on the classification of traffic, the 5 reducts were reworked to accommodate the 7 features which had been left out. Performance-wise, on average, the MCC for all three learners increased by more than 0.002. This indicates that the correlation of log information and flow data can further improve classification.

All in all, the reduced feature sets affected the overall detection quality only marginally in terms of MCC. Algorithms later trained with all 33 features (not including the log ones) saw an increase in the MCC of 0.003 for DS-1 and 0.015 for DS-2. The negligible increase in performance is outweighed by the fact that very similar performances can be achieved with half the number of features. Reduct #4 lead to lower MCC values in all 3 ML classifiers, when compared to the other 4 reducts and HNB algorithm performed the worst out of all the algorithms being considered. Nonetheless, all 5 feature subsets show great promise in the detection of a broad range of attacks.

Najafabadi et al [49] study the detection of brute force attacks against SSH service, at the network level. Moreover, the authors focus on the use of network flow data and the employment of machine learning solutions to perform the detection.

A custom dataset was built from real-world data, collected at a production computer network composed of over 300 hosts, spread across 4 subnets. Several of the network's hosts correspond to servers of various types including domain controller, Web, FTP, Email, and DNS servers. The dataset was initially captured in packet form with consequent extraction of flows using the SiLK tool. Flows were later labeled by network experts based on Snort alerts.

The features which will characterize the dataset can be found in Annex B.4.

The authors do not consider IP addresses as predictive features as to "avoid making out analysis special/customized to just the particular network" of their case study.

In order to evaluate the quality of the chosen features and to assert to what degree are machine learning solutions capable of detecting brute force attacks on a network level, the authors trained and tested four classification learners: k-Nearest Neighbor (k-NN) with k = 5, two forms of C4.5 Decision Trees (C4.5D and C4.5N), and Naive Bayes (NB).

Two experiments are performed. One does not consider any source or destination port as features when building the detection models and the other experiment does. This was done since it is common for network admins to change the port where a service is running to one different from the standard port.

To evaluate each algorithm, 4 runs of 5 cross-validation were carried out and AUC was the evaluation metric of choice. For the experiment where port-related features are not present in the feature set, k-NN, C4.5D, C4.5N and NB classifiers obtained an average AUC value of 0.9902, 0.9880, 0.9892 and 0.9707, respectively. Average AUC values are good indicatives of each algorithm's performance due to the low standard deviation of the 20 AUC values obtained for each classifier.

For the experiment where source and destination port are considered as features, k-NN, C4.5D, C4.5N and NB classifiers obtained an average AUC value of 0.9988, 0.9979, 0.9893 and 0.9975, respectively.

All "average AUC" mentions correspond to the average of all the AUC values calculated in each cross validation, for each run.

The two experiments show that for k-NN and C4.5N, the addition of the ports to the feature set did not translate into a such a big impact on their classification performance as it did for C4.5D and NB. This reveals that "using ports as features has different effects on the built classification models based on their nature".

Focusing now on the C4.5D algorithm, it is a tree-based algorithm and its classification process is supported by a decision tree structure. The hierarchy of features established through a normalized Information Gain is as follows: when ports are not considered as features, "number of bytes", "flow flags" and "number of packets" appear as the first, second and third levels of the trees. "Duration" will occasionally appear at the fourth level.

When ports are considered as features, "initial flags", "number of packets" and "destination port" appear as the first, second and third levels of the trees. Sometimes, "Source port" comes after the "destination port", occupying the fourth level of the hierarchy, ending on the labeled leaf.

Overall, all algorithms have shown very good performances in detecting brute force attacks.

Biglar Beigi et al [50] study and evaluate a group of flow-based features used by various machine learning-based botnet detection solutions, for the detection of general botnet activity and attempt to determinate what features, from the ones being considered, allow for

the most accurate detection, using stepwise selection. The authors perform a comprehensive analysis of all unique flow-based features found in the analyzed literature and give a brief theoretic explanation of how each feature is relevant for botnet detection. The main aim of this work was to assess whether the features employed by various existing botnet detection solutions reporting very high success rates and oftentimes performing detection in datasets with small number of botnet traces, could be applied to the detection of a broad range of botnet behavior and even novel botnet behavior.

The features on which the study based itself can be found in Annex B.5.

Authors reported an 11% increase in the botnet detection rate of the C4.5 machine learning algorithm from using all features to using the following 4 features: APL (average payload packet length), Duration (flow duration), BS (average bits-per-second) and PSP (percentage of small packets exchanged). Detection was performed on a custom built dataset which features botnet traces of over 16 different botnet solutions and benign traffic, from various publicly available datasets.

The authors conclude saying that statistical flow features alone are not enough to achieve the desired levels of detection required for its real world application and recommend using multi-dimensional snapshots of network traffic, i.e. combining flow-level features with either pair-level features or conversation-level features.

3.1.2 Datasets

Datasets are representations of the activity of a network, its hosts and the interactions between them and the internet. Over the years, various research efforts have been observed towards the construction of datasets, given their crucial role in the training, testing and evaluation of intrusion detection methods.

They can either follow a packet-based format or a flow-based format. Packet-based datasets are built from the packets travelling over a network, generated by its hosts, and can easily be constructed through port mirroring on a network device. The result will be a collection of packets which passed through the port being monitored.

Flow-based datasets consist of a collection of flows, which represent the interaction between two hosts, either in one direction or in both directions. These are built by either converting packet data into flows by means of a third party application or by setting up the network with flow-capable devices, namely a flow exporter and a flow collector which will, respectively,

construct the session data as it analyzes the packets travelling over the network and store them for later manipulation.

The following are the most prominently mentioned flow-based datasets in the literature [11] [51] and the ones which the feature study relied on. No packet-based datasets are considered as they fall out of the scope of the proposed problem.

CIC IDS 2017 The CIC IDS 2017 dataset [52] [53] was created within an infrastructure composed of 2 networks: one network, promptly named "attack-network", is set up as realistically as possible, containing one router, one switch, 1 host with Kali Linux OS and 3 hosts with Windows 8.1 OS. The other network named "victim-network" consists of three servers, with 2 of them being accessible through the Internet, one firewall, two switches and 10 hosts of varying OSs (Windows, Ubuntu and Mac).

The dataset's normal traffic was generated from profiles which contain abstract representations of events and behaviors seen on the network. By subjecting the traffic of 25 users on the same network, performing normal activities to a series of machine learning and statistical analysis techniques, it's possible to encapsulate its characteristics such as distributions of packet sizes of a protocol, number of packets per flow, certain patterns in the payload and others into a profile for later reproduction. In the case of CIC IDS 2017, these profiles were interpreted by Java-based agents, resulting in normal network activity in the form of packets.

Several attack scenarios were also defined, specifying the attacking and victim machines, the time of the day when they will be carried out and the tools to execute said attacks. Unlike the benign traffic profile, these scenarios require human intervention for executing them. Unique scenarios and respective conditions for execution were defined for the following attack categories: brute force, heartbleed, botnet, DoS, DDoS, Web attack (XSS, SQL injection), infiltration and port scans.

Network traffic was initially collected in packet form, by mirroring one of the ports of the main switch of the "victim-network". Once all attacks have been executed and the respective packet traffic has been collected, the traffic is then converted into bidirectional flows by CICFlowMeter software, developed by the same authors, which will also extract 80 features from the previously mentioned flows, with the final result being the dataset as it is publicly available.

Network data present in the dataset spans a period of 5 days.

CIC IDS 2018 The CIC IDS 2018 dataset [52] [54] was constructed in very similar conditions to the CIC IDS 2017, albeit with some slight improvements.

Firstly, the infrastructure supporting the dataset creation was moved to the AWS platform. It saw changes to the topology of its "attacking-network", which now has 50 machines, and a scale-up of the "victim-network" which is now composed of 5 subnetworks, representing the different departments of a company, and more than 420 hosts and 30 server featuring different OSs, different versions of the same OSs and architectures (32 bits and 64 bits).

Attack scenarios suffered slight changes in terms of host performing attacks, hosts which are victim of attacks and frequency of attacks, to accommodate for the changes made to the network topology.

Network data present in the dataset spans over 16 days, a larger period than the CIC IDS 2017 dataset.

CIDDS-001 CIDDS-001 dataset [55] [56] stands as a proof of concept for a new approach based on OpenStack technologies, for generating up-to-date, realistic and labeled flow-based datasets, depicting recent and relevant attacks as well as normal behavior.

The dataset, made up of generated network traffic in the form of unidirectional NetFlow records, spans over 4 weeks and contains approximately 32 million flows, each one labeled to one of five categories: normal, attacker, victim, suspicious and unknown. To collect the flows, the OpenStack platform was configured to use neutron with OpenVSwitch module which allowed for the capture of all network traffic within the virtual environment.

The environment set up for the creation of the CIDDS-001 dataset, in particular, was built to resemble a small business environment and consisted of 4 subnetworks, which hosted a total of four servers, three printers, four Windows clients and fifteen Linux clients. An external server, running two services and accessible through the Internet, was also configured to capture real malicious traffic, from attackers online.

Normal traffic was generated through scripts execution, which, based on a configuration file present in each host, would perform typical actions carried out by workers such as internet browsing, email writing, document printing and take into consideration working hours, breaks and other aspects.

Attacks were manually executed in specific machines to facilitate the labelling process later on. Attacks executed included DoS, brute force, port scans and ping scans.

CIDDS-002 CIDDS-002 [57] [58] was created in very similar conditions to CIDDS-001 though a few changes were made to the environment. First of all, the network was scaled down to 3 subnets which now host a total of four servers, two printers, 19 Linux clients and 4 Windows clients. No external servers were considered so no traffic from the Internet was collected. The dataset only has port scan attack activity and, unlike CIDDS-001 where all attacks were manually executed, scripts were updated to be able to autonomously execute port scan attacks, besides the normal activity, in a select number of hosts.

This dataset's time window was also reduced and only contains flows which span 2 weeks.

ISCX ISCX dataset is the result of an innovative approach devised by Shiravi et al [59] which aims to give researchers and practitioners the possibility of dynamically generating up-to-date and reliable datasets which reflect the intrusions of that time alongside the means for the dataset's modification, expansion and reproduction. This is made possible through the employment of "profiles".

Two types of profiles are considered: a(lpha) profiles describe an attack scenario and all the conditions that need to be met for it to take place. Each attack scenario is defined in an unambiguous way, with means of an attack description language. These profiles can then be interpreted and executed by human operators in order to produce malicious traffic in the network.

b(eta) profiles contain an abstract representation of events and behaviors seen on the network, that can range from a simple statistical distribution to a more complex algorithm. Through the analysis of real network traces, a b(eta) profile is able to encapsulate the characteristics of certain entities such as a protocol or an application, so that traffic relating to that entity can later be reproduced, with a similar fingerprint. In this case, these profiles allow the recreation of specific events on a network, essential for the generation of datasets that reflect different network realities.

For the ISCX dataset, 4 weeks of user activity on a network was recorded and further cleaned for any suspicious activity, for the creation of b(eta) profiles for various application protocols which include: HTTP, SMTP, POP3, IMAP, SSH, and FTP.

Several a(lpha) profiles were also crafted, featuring complex and multi-stage attacks scenarios to be executed during the capturing period. Attacks scenarios featured in the dataset include infiltration of the network from the inside, HTTP denial of service and brute force SSH

The physical testbed that supported the profile execution and consequent dataset creation consisted of 7 subnetworks, which harbored a total of 21 workstations (20 with Windows XP and 1 with Windows 7), 2 servers running Ubuntu 10.0.4, 1 Windows Server 2003 and 1 switch.

The dataset comprises network traffic captured over a period of 7 days. Network traffic was initially captured in packet format, by mirroring the port where traffic passed through.

From the packets, bidirectional flows were generated using IBM QRadar, an IDS management system.

TUIDS (version 1) In an attempt to produce more realistic and reliable datasets for intrusion detection method evaluation, Gogoi et al [60] created two datasets: the flow level TUIDS dataset and the packet level TUIDS dataset.

Both datasets were generated in a custom-built network testbed composed of one router, one L3 switch, two L2 switches, one server, two workstations and forty nodes, the last two being split over 6 VLANs.

As their respective names indicate, the two datasets were captured in both packet and flow form, by two separate port mirroring hosts. They contain normal behavior and attack behavior, which resulted from the manual execution of attacks during the capturing period. The dataset is home to the following attacks: bonk, jolt, land, nestea, newtear, syndrop, teardrop, winnuke, 1234, saihyousen, oshare, window, syn, xmas, fraggle and smurf.

Both types of captured traffic were preprocessed and filtered to extract various types of features, with support of a distributed feature extraction architecture. This architecture is mainly composed of 2 servers and 2 workstations, each one in charge of preprocessing the packet traffic data and the flow traffic data, respectively. To make the preprocessing stage as efficient as possible, each workstation manages various nodes, which are the pieces of hardware that effectively extract the features from the data and create additional ones, using various C routines, purposely designed for that end.

Focusing on the flow level TUIDS dataset, each data instance corresponds to a unidirectional NetFlow record. All NetFlow records were captured by a nfcapd daemon, included in the nfdump suite. The collected flows span a period of 4 weeks.

TUIDS (version 2) In an attempt to generate an unbiased real-life intrusion dataset which incorporated a large number of real world attacks, Bhuyan et al [61] created the TUIDS (Tezpur University Intrusion Detection System) intrusion dataset in both packet and flow

format, alongside two other datasets which focused on more specific attacks (the TUIDS coordinated scan dataset and the TUIDS DDoS dataset).

This dataset was generated in a custom-built network testbed which consisted of 250 hosts, 15 L2 switches, 8 L3 switches, 3 wireless controllers, and 4 routers that composed 5 different networks. Several VLANs were stablished alongside 1 DMZ where all the servers (main server, network file server, Telnet server, FTP Server, Windows Server 2003 and MySQL server) resided.

The normal traffic portion of the dataset was obtained from traffic generated by the TU faculty members, students, office staff and sys admins, which reflected their interactions with internal servers and the Internet.

The attack traffic, on the other hand, was introduced in the dataset by manually executing several attacks which included: bonk, jolt, land, saihyousen, teardrop, newtear, 1234, winnuke, oshare, nestea, syndrop, smurf, opentear, fraggle, linux-icmp, syn-flood, window-scan, syn-scan, xmasstree-scan, fin-scan, null-scan and udp-scan. All these attacks were integrated into 6 attack scenarios which contextualize the previously mentioned attacks, by defining additional steps that must be taken such as reconnaissance and information gathering in order to approximate the executed attacks to the type of behavior expected from real attack network activity.

Focusing on the flow-based version of the dataset in question, it consists of NetFlow records which were collected from different locations on the network, over 7 consequent days.

The tool which supported the collection and storage of the NetFlow records was nfcapd, which belongs to the nfdump suite. Once the collection period terminated, the authors applied several custom C routines to the NetFlow records, filtering any unwanted data and extracting additional features.

TWENTE Twente is the first labeled, flow-based dataset made publicly available for tuning, training and evaluating ID systems. Due to a lack of representative flow-based datasets and a growing interest for flow-based IDS, Sperotto et al [62] saw an opportunity to construct a dataset composed of flows, which was realistic, complete in terms of labelling and presented both an acceptable labelling time and adequate trace size.

This dataset comprises traffic originating from and directed to a computer with enhanced logging capabilities known as a honeypot, which the authors deliberately made available to

the Internet, so that malicious traffic could be generated from the attacks it would be exposed to. Due to the nature of honeypots, the majority of the collected traffic is malicious.

Instead of collecting the traffic at a flow-enabled network device, the authors opted for collecting only the traffic originating from and arriving at the honeypot in order to avoid split TCP sessions and timing delays between the logs created by the honeypot and the network packet's timestamps. The traffic was originally collected in packet format using tcpdump . Once the capturing period terminated, all network packet traffic was converted to unidirectional NetFlow flows using softflowd.

The honeypot was set up as a virtual machine powered by Debian Etch 4.0, a Linux distribution, running 3 services: an OpenSSH service, an Apache Web Server, and a proftp service, which offers ftp functionalities.

The traffic capture period lasted 6 days, resulting in a total of 14.2 million flows.

CTU-13 In an attempt to make the comparison of botnet detection methods a more simple and accessible process and to allow newer contributions to be rapidly improved, García et al [63] have created a comparison methodology, a novel error metric which will support the comparison of the results of different botnet detection methods and the CTU-13 dataset.

The dataset was created from traffic collected from the network of the university of one of the authors. Adding to the already existing infrastructure, a set of virtual machines running Windows XP SP2 were created. Each virtual machine was running on top of a Linux Debian host. Each VM was bridged to the respective network to make the capturing environment as realistic as possible. No further information regarding the network's topology was presented.

Network traffic was collected at two locations in the network: at the router, so that both normal and background traffic generated by the users in the network could be collected, alongside the attack traffic, generated by the infected host and at the respective infected hosts, so that only attack traffic would be captured. By doing this, the authors were able to easily discern between attack traffic and normal traffic and label it accordingly. Traffic was captured at both ends using tcpdump and later converted to bidirectional NetFlow records using Argus.

To make a dataset as comprehensive as possible and representative of different botnet malware behavior, the authors present 13 scenarios, each one featuring a different botnet malware threat and a network traffic capture which reflects its behavior on the network. Further, the datasets pertaining to scenarios 1, 2, 6, 8 and 9 were designed to be used for

algorithm testing and the datasets of scenarios 3, 4, 5, 7, 10, 11, 12 and 13 were specifically designed for the training and cross-validations of algorithms.

Each respective scenario's dataset totaled an average of 4 301 754 flows.

UGR'16 UGR'16 [64] is the first dataset purposely built for the training and evaluation of IDSs that consider the cyclostationary evolution of traffic, that is, the differences between daytime/nighttime and weekdays/weekends traffic. To that end, the authors ensured that the dataset featured long duration traffic traces, so that a representative study of the network's patterns over several time cycles could be achieved. Despite the clear emphasis on long capture time, the authors also strived to comply with the other dataset requirements such as have a number of relevant features, feature real background traffic as well as updated attack traffic, be labeled and be publicly available.

The dataset consists of anonymized unidirectional NetFlow v9 traces, captured over a period of 4 months, in a tier 3 ISP network, featuring recent attack strategies such as low-rate DoS, port scanning and botnet traffic.

The network itself is virtual and its topology consists of 2 redundant routers, 2 firewalls and 3 subnetworks: the "core" and "inner" networks, which are separated by two firewalls, already existed in the ISP network before the traffic collection process took place. They were fitted with 5 victim machines and further divided into 3 networks, each one connecting 5 victim machines, respectively. The final subnetwork, promptly called "attacker" network, was purpose-built for the creation of the dataset and consisted of 5 attacker machines. This network was directly connected to one of the routers so that the traffic coming from it resembles that of attackers on the Internet.

The dataset is split into two captures: a calibration capture, with over 10 billion flows, which only contains background traffic and that should be used to train the normality model; testing capture, characterized by 3.9 billion flow records, which is a mix of background and synthetically generated attack traffic, intended for the evaluation of the previously trained intrusion detection approach. The captures span over 100 days and 33 days, respectively.

Further, traffic was captured using the nfdump tool, more precisely, the nfcapd daemon service.

SSHCure SSHCure is an IDS specialized in the detection of SSH brute-force attacks. Its detection algorithm was developed around the idea that each SSH brute-force attack comprises 3 phases: scan phase, when the attackers scan for hosts with active SSH daemons;

brute-force phase, where the attacker has found a victim and performs a large number of authentication attempts, in a very short period of time, using a wide range of username/password combinations; compromise phase, which is the phase that starts as soon as the attacker successfully gained access to the victim host.

With the goal of improving the performance of the "compromise" detection of their IDS, Hofstede et al [65] identified 4 compromise scenarios by characterizing network traffic produced by over 10 attack tools. In order to evaluate the improvements made to the SSHCure IDS, they also present two new datasets, crafted from the traffic generated by over 25 thousand hosts, belonging to a segment of the network of the University of Twente which is exposed to the Internet.

Both dataset's feature network traffic from the campus network of the University of Twente, obtained from the edge routers in NetFlow format, over a period of 1 month.

Dataset 1, as the authors named, consists of traffic from various low-, medium- and high-interaction honeypots.

Dataset 2, on the other hand, exclusively contains traffic originating and sent to servers and workstations.

The datasets are not directly labeled and information regarding the success of compromises can be inferred from the accompanying honeypot, workstations and server logs.

Kent2016 The Kent2016 dataset [66] reflects various network events such as: Windows-based authentication events from both individual computers and centralized Active Directory domain controller servers, process start and stop events from individual Windows computers, DNS lookups as collected on internal DNS servers, network flow data as collected at several key router locations and a set of well-defined red teaming events that present bad behavior.

Focusing on the network flow data (which will be referred throughout the rest of the subsection as "dataset") , it contains traffic captured from central routers within the Los Alamos National Laboratory's (LANL) corporate, internal computer network, in NetFlow format, spanning a period of 29 days.

The dataset is not labeled and minimal information is provided for other researchers who wish to do so. Due to privacy concerns, some aspects including IP addresses, timestamps and some lesser known ports have been anonymized.

Unified Host and Network Dataset Following similar motivations of other authors, Turcotte et al [67] created the UHAN dataset to stimulate new research endeavors in the field

of cybersecurity and motivate other organizations to follow suit and publicize new, representative datasets.

The dataset consists of network traffic traces in the form of bidirectional flows and computer events which were collected from the LANL enterprise network. The collection period lasted approximately 90 days.

Focusing solely on the traffic flow data, it was collected from various internal routers in the form of unidirectional NetFlow v9 records which were then subject to biflowing, which consists on the conversion of unidirectional flows to bidirectional flows, using a custom heuristic based on the port numbers to establish which flows correspond to the source and destination.

Much like the Kent2016 dataset, which was also constructed from the same network, UHAN is not labeled and any sensitive information regarding the LANL's operational IT environment or its users such as IP addresses, timestamps and some lesser known ports, was anonymized.

The network itself is mostly composed of Windows hosts. No additional information regarding its topology is given.

UNSW-NB15 UNSW-NB15 dataset [68] [69] aims to be a comprehensive network dataset, which accurately reflects the modern network traffic behavior and the patterns induced in the traffic of a network by low footprint intrusions. Even though firewalls, a very common and reliable form of network defense, can deter many threats from the outside, IDSs offer greater defensive capabilities against contemporary network threats that firewalls cannot fully mitigate due to an inability to analyze packets. For this reason and a general dissatisfaction in the level of quality and comprehensiveness of existing datasets, led the authors to the creation of this dataset.

The dataset, which features a hybrid of real, modern, normal behaviors and artificial attack activities, was constructed using the IXIA PerfectStorm tool, which generated both the normal traffic as well as attack traffic which reflects the patterns of 9 categories of attacks: fuzzing, analysis, backdoor, DoS, exploit, generic, reconnaissance, shellcode and worm attacks.

It comprises two captures. The first one features one attack per second and contains traffic lasting 15 hours. The second capture, which took place a month later, features 10 attacks per second over 16 hours. Once features were properly extracted and created from the collected packet network traffic, the first captured resulted in a total of 987627 flows and the second one resulted in 976882 flows.

The network traffic featured in the dataset was initially captured in packet form by the tcpdump tool. The dataset features were then extracted from the raw network traffic packets using Argus and BRO IDS. Once this process was complete, the derived features from each tool were matched, originating one single feature base. Additional features were later were created from 12 custom-built C# algorithms.

Labelling was performed based on ground-truth information provided by an automated report generated by the IXIA PerfectStorm tool, which contains several details regarding the execution of the different attacks such as time of execution, duration, among others.

The testbed which allowed for the creation of the dataset was set up in the Cyber Range Lab of the Australian Centre for Cyber Security (ACCS) and was composed of 3 servers, one connected to one router called “Router 1” and the other two connected to the other router called “Router 2”. Both routers were separated by a firewall and had several other clients connected to them though no more information was given. Traffic was collected from Router 1.

CID DDoS 2019 The CIC DDoS 2019 dataset [46] , unlike other datasets, only contains DDoS attack behavior. The authors were compelled to create it due to a lack of comprehensive and reliable datasets to test and validate DDoS attack detection systems. In parallel to the release of the dataset, the authors also strived to define a new taxonomy on which to categorize DDoS attacks, which consists on the following categories: reflection-based attacks where a legitimate network component, not necessarily compromised, is used by an attacker to carry out malicious actions towards the victim, effectively hiding his identity; and exploitation-based attacks where the attacker exploits the protocol’s design flaws to disrupt the victim’s normal behavior.

The dataset features 12 types of DDoS attack techniques: WebDDoS (ARME), DNS, LDAP, MSSQL, NTP, NetBIOS, SNMP, SSDP, SYN, TFTP, UDP and UDP-Lag.

The testbed network where the dataset’s traffic was collected consisted of two networks: the “victim network” was composed of one web server, one firewall, two switches and four PCs. Traffic was collected in the form of packets by mirroring one port in the main switch; and the “attack network”, a third-party network of unknown topology from where the attacks were executed, directed towards the “victim network”.

Normal traffic was artificially generated through the B-profile approach, already employed in the CIC IDS 2018 and CIC IDS 2017 datasets.

The capture period lasted, approximately, 15 hours. The dataset was initially captured in packet format with a unknown tool, and fed as input to the CICFlowMeter v3 tool which promptly created bidirectional flows from the packet data, characterized by over 80 features.

Much like the CIC IDS 2017 and the CIC IDS 2018 datasets, the authors also used the RandomForestRegressor class of scikit-learn to study the relevance of each feature in identifying each one of the 12 DDoS attacks present in the dataset.

To finalize, the authors trained and tested 4 ML algorithms, namely, ID3, Random Forest, Naïve Bayes and Multinomial Logistic Regression using the generated dataset and a five-fold cross validation, having obtained lackluster results.

LITNET 2020 Like many other datasets which have been released in the past decade, LITNET 2020 [45] was built with the purpose of giving researchers and practitioners a realistic, labeled, publicly available dataset, representative of normal user behavior and multiple types of attack behavior, against which they could compare and validate the capabilities of their own intrusion detection implementations with existing ones.

To overcome the shortcomings of its counterparts, the LITNET 2020 dataset features a wide variety of contemporary network attacks such as Smurf, ICMP Flood, UDP Flood, SYN flood, HTTP Flood, LAND, W32.Blaster, Code Red, SPAM, Reaper Worm, Scan and Fragmentation. It encompasses over 45 million network traffic traces which were collected from 06/03/2019 to 31/01/2020, spanning over 11 months. No information regarding how the attacks were carried out

Network flow traffic was captured using the nfcapd module of the nfdump tool, in bidirectional NetFlow v9 record format. It contains a mix of real attack traffic executed by the authors and real background traffic

Traffic collection took place at the Lithuanian Research and Education Network (LITNET), a vast network which connects 6 universities located at 5 different Lithuanian cities and various end users from schools, municipalities and other organizations. LITNET is subdivided into 5 networks, each one serving the researchers and students of one of the 5 cities. The LITNET dataset contains network traffic collected at 5 distinct points of this wide network, known as NetFlow exporters. More precisely, NetFlow data was collected from the router of the CITY1 network (also known as the Kaunas city network) and from 2 CISCO routers and a Fortige firewall hosted at the Kaunas University of Technology (KTU) subnetwork which is part of the CITY1 network. Additional NetFlow data was also retrieved from the CAPACITY network's

router (also known as the Vilnius city network), which comprises the Vilnius Gediminas Technical University network.

All the collected data was routed to a NetFlow collector running Debian Linux OS, responsible for receiving, storing and filtering the data.

The dataset is composed of 85 features, defined based on the recommendations given by the authors of the UGR'16 dataset: 64 features are directly obtained from the NetFlow v9 records and 15 features were custom-built by the authors for attack type recognition using custom Python scripts. The last 2 features correspond to the "attack label" feature which indicates if the flow corresponds to attack traffic or normal traffic and the "attack type" feature which further specifies the type of attack the flow is related to (only when "attack label" is set to 1).

To conclude the dataset study, a comparative table was constructed. Each row corresponds to a feature and each column to a dataset. This way, it is possible to visually compare what features are present in which dataset and which ones are not. The table is available in Annex C.

For more information on any of the datasets regarding the labelling process, advanced testbed configuration, statistics regarding the collected traffic, attack schedule and the files pertaining to each dataset, the reader is directed to the reference which accompanies it.

3.1.3 Tools

The following are a set of tools which preprocess raw network data in packet format much like the proposed solution. The tools convert the data into a series of observations, where each observation is represented as a feature vector. These feature vectors are then suitable as input to data mining or machine learning algorithms [12].

Among all tools which have been developed to that end, 5 stood out for their acceptance in the community, overall utility and numerous mentions in the literature.

CICFlowMeter CICFlowMeter [54] [70] is an open source tool, written in Java, maintained by the Canadian Institute for Cybersecurity (CIC) of the University of New Brunswick (UNB). The tool's main functionality is to generate bidirectional flows from network packet data and for each flow record, extract 83 features. The tool has two input methods: offline and online. In offline mode, the application will read .pcap files, captured prior to the application execution. In online mode, the application performs live packet capture from an interface. The tool will then output the results to a .csv file, where each column represents a feature and each row a flow record.

The tool supports the definition of custom values for flow timeout and activity timeout.

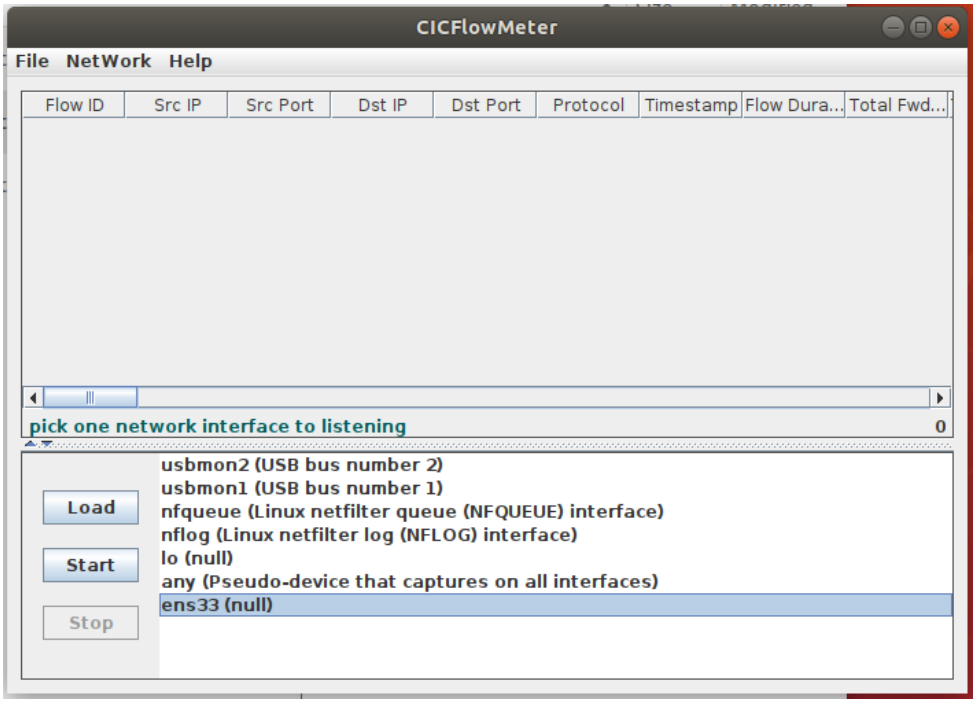


Figure 6 - CICFlowMeter tool GUI

YAF Yet Another Flowmeter (YAF) [71] [72] is a suite of tools intended for performing flow metering in a network. YAF, which was created by CERT NetSA group, is often deployed as a sensor to capture flow information from a network and export that information in IPFIX format. YAF reads packet data either from pcap dumpfiles as generated by tcpdump or from a network interface via live capture. As individual packets are analyzed, they get aggregated into bidirectional flows, based on the popular flow key “Source IP, source port, destination IP, destination port, protocol”. In terms of output, the created flow records are either serialized into IPFIX message streams (IPFIX files) on the local file system or are exported to a collector over SCTP, TCP or UDP, Spread where they will be further analyzed.

YAF supports native bidirectional flow creation, capture of both IPv4 and IPv6 packets within the same capture period, application labelling of individual flows based on deep packet inspection techniques and high-speed collection from dedicated capture cards such as Endace DAG and Napatech devices.

YAF also offers a very comprehensive packet filtering engine which can filter, group and store packets by flow, in different pcap files, based on a certain flow key [73].

Users of the application have access to a wide range of Information Elements, or features, including, features which have been defined in IPFIX-related RFCs and additional features

which are derived from packet-level or packet payload information, such as TCP initial sequence numbers or payload Shannon entropy.

Much like Netflow, YAF is used as a sensor to capture flow information on a network and export that information in IPFIX format. YAF output can be analyzed in a human-readable format using yafscii, which comes with YAF, the SiLK suite of tools or nafalize tool, both available from the CERT NetSA group.

An illustrative example of the YAF and yafscii tools working in conjunction can be observed in Figure 7.

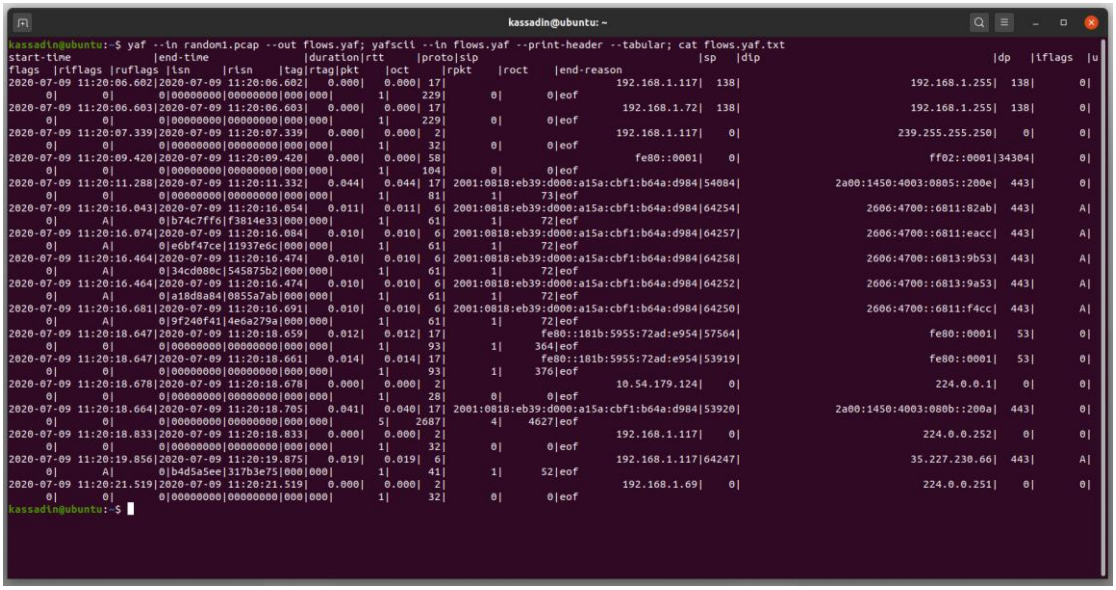


Figure 7 - YAF suite of tools in execution

nProbe nProbe [74] [75] is a software probe, developed and maintained by the nTop team, capable of collecting, analyzing and exporting network traffic reports using the standard Cisco NetFlow v5/v9 and IPFIX formats. It is a very lightweight and fast It features 3 distinct modes of operation: probe, collector and proxy, making it a very versatile solution, able to fill all the roles in a flow creation process. In proxy mode, it can be fitted to convert from NetFlow v5, NetFlow v9, NetFlow Lite, sFlow, IPFIX jFlow to NetFlow v5, v9 or IPFIX in order to smoothly upgrade to newer NetFlow protocol versions while capitalizing older ones. Regardless of the mode of operation, it supports the export and the collection of NetFlow v5, NetFlow v9 and IPFIX flows. In terms of performance, nProbe is very resourceful, using only 2Mb of memory regardless of network size, and natively supports PF_RING technologies and the newest PF_RING Zero Copy (ZC) kernel-bypass which allows for very high packet capture speeds. Adding to that, nProbe also features a plugin engine that extends the core engine with additional capabilities. nProbe is not open source.

nfdump nfdump [76] is an open source suite of tools designed for the purpose of flow collection, currently compatible with flow formats NetFlow v1, v5, v7, v9, IPFIX and sFlow. Each tool of the suite performs a specific action and when complemented with each other, give origin to a robust and comprehensive flow collector. Nonetheless, each tool can be used in a standalone way, without restrictions. There are, in total, 5 tools:

The nfcapd tool performs the actual collection of flow data sent from flow exporters and stores them in files.

The nfdump tool processes the collected flow data. It offers a filtering syntax similar to that of tcpdump with small adaptations for flow data. It can also perform calculation of certain statistics on the available data and flow aggregation functionalities, including unidirectional flow aggregation into bidirectional ones. It supports the export into binary and CSV format.

The nfanon tool anonymizes flow records, namely the IP addresses, using the CryptoPAn method.

The nfexpire tool manages data expiration and set appropriate limits.

Lastly, the nfreplay tool turns the built pipeline into a proxy, effectively reading the flow data stored in the files by the nfcapd tool (or not) and sends it over the network to another host.

Of the remaining optional tools, the nfpcapd tool (note the extra “p” when compared to nfcapd) is particularly relevant as it allows the live capture of packets from an interface or from a previously captured packet trace in a pcap file.

Nfdump is commonly paired with nfsen, a graphical web-based frontend for the nfdump suite of tools, for graphical flow analysis.

Table 11 - Comparative table between existing solutions

	Supported input formats	Supported output formats	OS compatibility	Data labelling	Data anonymization
CICFlowMeter [70]	Packets (pcap file and live capture)	Bidirectional feature vectors (80+ custom features in CSV format)	Windows, Linux/UNIX	✗	✓
YAF [71]	Packets (pcap file and live capture)	IPFIX (binary)	Linux/UNIX	✗	✗
nProbe [74] [75]	Packets (pcap and live capture) in probe mode; unidirectional feature vectors (NetFlow v5, v9, NetFlow Lite, sFlow, IPFIX and jFlow in binary) in collector and proxy mode	Unidirectional feature vectors (NetFlow v5, v9, NetFlow Lite, sFlow, IPFIX and jFlow in binary) in probe and proxy mode	Linux, Windows, and embedded environments ARM and MIPS/MIPSEL	✗	✗
nfdump [76]	Packets (pcap file and live capture) and flows (NetFlow v1, v5, v7, v9, IPFIX and SFLOW in binary)	Unidirectional or bidirectional feature vectors (NetFlow v1, v5, v7, v9, IPFIX and sFlow in binary or CSV)	Linux/UNIX	✗	✓
Proposed solution	Packets (pcap file) and flows (CIC IDS 2018 dataset, Suricata log files)	Bidirectional and unidirectional feature vectors (variable number of custom features in CSV format)	Any Python-supported OS (Windows, Linux/UNIX, Mac OS)	✓	✓

Besides the tools previously presented, there are documentations in the literature of other tools which were used for similar purposes. Such tools are Argus, Netmate, SiLK and softflowd. Some tools of these tools, which are no longer being maintained, include flow-tools, softflowd, tcptrace, cflowd and flowd.

Some IDS have also been used to convert packet data into flow data, including the Snort and Bro IDS, the former having been used to construct the KDDCup, NSLKDD and the Kyoto 2006 datasets which were not addressed in this report.

Although some of the analyzed tools also double as exporters or collectors in a flow creation process, as described in Section 2.4.2, the developed application developed does not perform any kind of packet collection, relying exclusively on prior captures of traffic traces.

3.2 Existing technologies

In this section, a revision of existing technologies is made, comprising a critical comparison between the chosen technologies and other candidate technologies which were considered to be part of the proposed solution. The revision is divided into three part, one for each type of technology which the application required.

Following, the patterns and good practices applied throughout the development cycle are presented.

3.2.1 Programming languages

The problem proposed demands the construction of a script. The purpose of the application is to preprocess data in a quick and efficient manner so that it can then be given as input in a ML algorithm. Also, the final application will be mainly used in a network administration and monitoring context where scripting and the piping of scripts is very common, to allow for the automation of processes. Due to this, the application does not require a GUI meaning that the chosen language does not need to support GUI creation natively and the existence of GUI-building frameworks should not be a priority when choosing one.

With that said, the Python and the Java languages were considered for the development of the application.

Python Python [77] is an interpreted, high-level, general-purpose programming language, created by Guido van Rossum and first released in 1991. It supports multiple programming paradigms including procedural, object-oriented and functional programming. When used for object-oriented programming, it supports multiple inheritance. It is equipped

with automatic garbage collection mechanism, based on reference counting and cycle detection.

Python's core philosophy is summarized in the Zen of Python manifesto, which comprises 19 aphorisms [78].

In order to comply with the 7th aphorism of the Zen of Python “Readability counts.”, Python strives to be an easily readable language, with various efforts having been made to keep its syntax simple and accessible while giving the developers the freedom to choose their own coding methodology. As an example, Python prefers the use of English expressions in favor of punctuation in its syntax. Code blocks are defined solely through whitespace-based indentation unlike other languages which use a bracket nomenclature. It is clear the emphasis given to the visual aspect of the language which, in the end, reflects in less cluttered and easily understandable code.

Python's design philosophy also prioritizes extensibility which is reflected in a small, more narrow scoped core library and a comprehensive, large standard library and easily extensible interpreter, so that is easy for the community to contribute with new functionality and make it as wide-ranging and versatile as possible. This paradigm made possible the construction of a large community-driven code repository, the Python Package Index (PyPI), which has amassed a total of 256,380 projects as of August 2020 [79].

Python, like any other programming language, is divided into two parts: an interface and an implementation. The interface defines the syntactic and semantic rules which describe the language's structure and meaning, respectively. The implementation will compile the code into some form of bytecode which is later executed by a virtual machine. With that said, Python has various implementations: CPython, PyPy, IronPython, Jython, among others. As an example, the CPython implementation, is written in C and compiles the Python code to CPython bytecode which is then run on the CPython Virtual Machine. Jython, on the other hand, is written in Java and compiles the Python code to Java bytecode which is then run on the JVM. Depending on the design behind each implementation, the resulting bytecode from the compilation from Python code will either get interpreted, like what happens in the CPython implementation, or get further compiled like what happens in the JIT-based implementation PyPy [80].

It is a dynamically typed language meaning that type-checking[1] occurs at runtime. This means that when a type error occurs, normal program execution will halt and if the error goes unchecked, the program will crash. Further, in a dynamically typed language the developer does not need to explicitly declare the type of the expressions.

As a consequence of being a dynamically typed language, Python is commonly described as a “duck typed” language. Duck typing is a programming style where an object's interface (its methods and attributes) is more important than its class. By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution [81]. This complements very well with the "Easier to Ask for Forgiveness than Permission" coding practice, considered to be the pythonic way of developing code. It supports the idea that its preferable to let code error out and deal with any errors or exceptions once they occur, rather than attempt to verify if all conditions are met before executing a piece of code. This practice flourished, in part due to Python's dynamically typed nature and "we are all adults here" approach, which gives the developer more freedom in the way it codes.

It is strongly typed meaning that the type of an object remains invariable throughout its lifetime, regardless of when type checking actually occurs i.e. if the language is dynamically or statically typed. An example is how in Python the sum of a string with an integer will result in a type error but in PHP the same is not true and program execution carries on as usual.

Its own dynamic typed nature confers it dynamic name resolution or late binding characteristics. Late binding is a property of dynamically typed programming languages where binding, the process of matching function calls written by the programmer to the actual code (internal or external) that implements the function [82], is only done at runtime. This means that the code to be executed when a certain method/function is invoked, is only searched during run-time. If no method with the textual name of the invoked method exists, the program halts execution and raises an exception or crashes, if the exception is not resolved.

Lastly, it supports introspection, the ability of a program to examine the type or properties of an object at runtime, and reflection, which goes a step further and gives a computer program the ability to examine and modify its structure and behavior (specifically the values, meta-data, properties and functions) at runtime [83].

[1] The mechanism which ensures that the application is type-safe and that no operations between incompatible types happen

Java Java is a compiled, high-level, general-purpose programming language. It was created by James Gosling while working at Sun Microsystems and released in 1995 as a core component of the Sun Microsystem' Java platform. Since then, Sun Microsystems has been acquired by Oracle and all components of Java have been made open source. It supports object-oriented programming and single inheritance, with the exception for interfaces.

It is equipped with an automatic garbage collector for memory management.

Java's design philosophy emphasizes portability, with the goal of being able to run on any platform and with as few dependencies as possible [84]. Java successfully implements the "write once, run anywhere" paradigm which addresses the need for identical code to behave as closely as possible on any kind of hardware/OS combination. The devised solution was to separate the hardware from the code compilation. Source code is first compiled to an intermediary form called Java bytecode which is then executed by a virtual machine (JVM), purposefully built for the computer architecture in question. Moreover, any computer architecture which supports Java will be able to run the code, without the need for recompilation.

It features a comprehensive standard library which supports the application development.

It is statically typed, which means that type-checking takes place during compile-time. As a result, any type error, which is not introduced during run-time due to reflection, is detected before the program is executed. This improves the language's performance by minimizing the occurrence of type errors during execution. In terms of how code is written, developers must explicitly declare the type of the expressions.

It is strongly typed, and it is characterized by early binding where the compiler can work out where the called function will be at compile time. It can thus guarantee that each function invoked exists and is callable at runtime and ensures that the number and type of arguments being supplied matches that of the function definition. It also checks that the return value is of the correct type [85].

It has both introspection and reflection capabilities.

Table 12 - Comparison between Java and Python

	Python	Java
Type of language	Interpreted (depends on the implementation) [80]	Compiled

Supported programming paradigms	Procedural programming, functional programming and object-oriented programming.	Object-oriented programming.
Language type (dynamic or static)	Dynamically typed	Statically typed
Language type (strong or weak)	Strongly typed	Strongly typed
Binding	Late binding	Early binding (for normal method calls and overloaded method calls) and late binding (reflection and method overriding (run time polymorphism)) [86]
Inheritance	Single and multiple inheritance	Multiple inheritance is partially achieved through interfaces
Cross-platform support	✔	✔
Support for reflection and introspection	✔	✔
Implementations	CPython, PyPy (just-in-time compiler), Jython, IronPython	Oracle implementation, OpenJDK
AI libraries	TensorFlow, Keras, Pytorch and Scikit-learn	Weka, Mallet, Deeplearning4j, MOA
GitHub's Annual Octoverse language ranking [87]	2	3
TIOBE ranking (August 2020) [88]	3	1

Both languages have an "(almost) everything is an object" approach and offer extensive standard libraries for application development though Python has a more powerful string manipulation library.

Java is a more verbose language when compared to Python, requiring more boilerplate code when compared to Python, for the same functionality to be achieved [89].

In terms of performance, the overhead of interpreting bytecode into machine instructions makes interpreted programs almost always run more slowly than their native executables counterparts resulting from a compilation process. While compilation of large applications takes more time, the tradeoff of long compilation times is better performance during execution.

Most JVMs perform just-in-time compilation to all or part of programs to native code, which significantly improves performance [90][89]. The mainstream Python implementation, CPython, does not have support for JIT compilation but PyPy, another open source implementation, does.

Static-typing makes code run faster. A compiler working on statically typed code can optimize better for the target platform and has more control over memory management [89].

Since all information needed to call a method is available before run time, early binding, supported by Java, results in faster execution of a program. While for later binding, in this case supported by Python, a method call is not resolved until run time and this results in somewhat slower execution of code [86].

All in all, Java will usually have better performance than Python, for the appointed reasons which are further corroborated by this study [91] which shows that a simple binary tree test runs ten times faster in Java than in Python 3.

In terms of portability, Python is less portable as it needs an interpreter to be executed while Java can generate executables which can easily be ported and executed in Java-supported systems, though it is dependent on the JVM to also be executed. This, however, will not be a problem as the developed solution will be mainly used in Linux distributions, which usually come pre-installed with Python.

Since the project benefits from the application of artificial intelligence for feature selection, the opportunities offered by each language must be studied. Java is considered a good option when it comes to machine learning, it is easy to debug and use and it is already being used for large-scale and enterprise-level applications. Among the libraries, you could use in that area are Weka, Mallet, DeepLearning4, and MOA [84]. Python, however, has a stronger presence in the Artificial Intelligence app development scene, offering the highly powerful and popular TensorFlow, Keras, Pytorch and Scikit-learn libraries. Its highly accessible syntax means that interested parties can easily start prototyping which is not usually the case with Java. Adding to that, Python’s high versatility and huge community involvement made it a popular option for people from different disciplines who wished to experiment with machine learning and

bring the power of AI into their respective fields. That is why a lot of the development in AI and machine learning is done with Python with a huge ecosystem and libraries [84].

To conclude, Python was the chosen language for the development of the application. Although the bachelors focused heavily on Java, Python's affinity for AI projects and powerful libraries make it the best alternative due to the nature of the project.

3.2.2 CLI building libraries

Several libraries exist for the creation of CLI for applications whose main form of interaction with the user is through the keyboard rather than the mouse. Several libraries were found in the literature Among several libraries found in the literature, 3 stood out for their acceptance in the community and overall utility, enabling the fast and easy creation of intuitive CLIs.

argparse Argparse [92] is a library available in Python's standard library for building intuitive CLIs. It features built-in argument and option type-checking. It has support for subcommands, automatic help page generation, parser hierarchy which allows the definition of common options or arguments which are expected to be received by multiple subcommands, the definition of types and default values for the options or arguments, multi-valued options or arguments, mutually exclusive options or arguments, the definition of both single and double-dashed names for options, the definition of allowable values for an option or argument and the grouping of arguments in the help page based on a custom category rather than just if they are "optional" or "positional".

It does not allow proper nesting of commands by design and has some deficiencies when it comes to POSIX compliant argument handling [93]. Further, it has built-in behavior to guess if something is an argument or an option. This can lead to unreliable application behavior when dealing with incomplete command lines [93].

Argparse operates in the following way: a program which features the Argparse library for CLI building, must first parse the received arguments from the command line through a manual call to the parse_args() instance method on a argparse.ArgumentParser instance. This instance reflects all the arguments, options and commands which are supported by the application, as well as any type restrictions or default values in case a value for an optional argument is not specified. Once all arguments, options and commands are properly parsed and validated, they are further accumulated into a Namespace object over which application logic is performed.

click Click [94] is an open source, third party library for command-line argument parsing. Initially based on optparse, it proposes a decorator-based approach for the creation

of CLIs, where commands are mapped to functions and each function is, in turn, wrapped in one or more decorators. It is more complex than argparse and offers a more manageable alternative for the creation of more large-scale and comprehensive CLIs. It is lazily composable without restrictions and fully nestable. It is able to parse and dispatch to the appropriate code. It has a strong concept of an invocation context that allows subcommands to respond to data from the parent command. It makes strong information available for all parameters and commands, so that it is possible to generate unified help pages for the full CLI and assist the user in converting the input data as necessary. It features a strong understanding of what types are, and it can give the user consistent error messages if something goes wrong. It has enough meta information available for its whole program to evolve over time and improve the user experience without forcing developers to adjust their programs.

It has support for colorized output, progress bars, user input prompt including clear and password-type text, easy integration with setuptools for module distribution, implementation of Unix/POSIX command line conventions, native loading of values from environment variables, native file handling, “wait for key press” functionality and launching of external applications.

Click’s design philosophy prioritizes a consistent command line experience across multiple systems and reliability of results over configurability. This is reflected in the large number of hardcoded behaviors which ensure the testability of the code and ease the documentation process of a new interface. Furthermore, it aims to create a level of composability of applications that goes beyond what the system itself supports [93].

Click operates in the following mode: argument parsing is achieved through decorated functions. The decorators specify the kind of arguments, options or subcommands supported. When the function is invoked, it reads the arguments from the command-line which are passed to it as function parameters, which must have the same name as the one specified in the decorators and executes business logic.

docopt Docopt [95] is a third-party, open source CLI building library which is not only supported in Python but in many other programming languages, which makes it very versatile. A docopt implementation will extract all required information to generate a command-line argument parser from a textual description in docstring format, which contains wildcard symbols such as brackets "[ ]", parentheses "( )", pipes "|" and ellipsis "..." which describe optional, required, mutually exclusive, and repeating elements, respectively. In essence, docopt parses the help page and then parses received command line arguments according to

those rules. The text of the interface description also doubles as the help message shown when the program is invoked with the -h or --help options.

Docopt's design philosophy puts more importance on interface configurability and aesthetics in favor of composability [93].

Docopt will operate in the following way: a parser is automatically generated from a formatted docstring written by the developer. Command line arguments are then parsed into a dict-like object, which maps parameters to strings. Business logic is executed next, over the info available in the dict.

All libraries presented seem to offer the same core functionality expected from a CLI: parsing of the received arguments, options and commands and error issuing when users give the program invalid arguments. From the analysis, there does not seem to exist a single, perfect library for CLI construction but rather, each one has its own advantages and disadvantages and each one will be better suited for projects of different sizes and complexity.

Argparse has the advantage of being part of the standard library which means that in terms of distribution, the app will have any dependencies.

Docopt, in comparison to Argparse and Click, in special [93], is quite rigid in how it handles the command line interface as the entire CLI must be defined in one string. On one hand, it gives the developer more control over the help page. On the other hand, due to this it cannot rewrap the output for the current terminal width, and it makes translations hard. Docopt is also restricted to basic parsing. It does not address issues such as argument dispatching and callback invocation or types. This ultimately results in more code having to be written in addition to the basic help page to handle the parsing results. While docopt does support dispatching to subcommands, it does natively support any kind of automatic subcommand enumeration based on what is available and will not enforce subcommands to work in a consistent way.

Click is the most powerful of all libraries analyzed and while it is quite accessible for simple building simple and small CLIs, it has a steeper learning curve and the process to achieve a certain functionality in click is not as straightforward as in Argparse for example.

In the end, Argparse was chosen to integrate the final solution as it provides enough functionality to fulfill the proposed requirements. This library presented itself as a very appealing alternative as it did not introduce any additional dependencies into the application, making it more portable. Adding to this, the official documentation is very accessible and

properly displays the flexibility and various functionalities of the library which contributed to its election.

Other libraries, featuring similar CLI building capabilities, found in the literature which are worth mentioning include fire, python-nubia, knack, Plac, Cliff, Cement and PyInquirer.

3.2.3 Packet parsing libraries

In order to extract meaningful information from packets for flow creation, these must be parsed and have their respective fields be easily accessible. Preferably, packet parsing must be a process as optimized as possible, to account for large traffic traces. Multiple open source packet manipulation libraries are available. The following were the libraries considered to be a part of the final solution, to aid in the parsing of pcap files and the crafting of custom packets which would later be sent on the wire, for testing purposes.

scapy Scapy [96] is an open source, cross-platform project for packet manipulation purposes. Available both as a shell and a library, Scapy's main reason for its popularity is a robust system for crafting, sending and receiving packets. Adding to this, it is capable of matching requests to the appropriate replies, returning a list of packet couples (request, answer) and other unmatched packets. It also provides functionality for live capture, offline packet manipulation from pcap files, forging and the decoding of network packets from a wide number of protocols. Moreover, it can perform tasks which other libraries are not capable such as sending invalid frames onto a network, injecting custom 802.11 frames and combining different techniques like VLAN hopping and simultaneous ARP cache poisoning.

Scapy design philosophy is one centered around flexibility and giving developers just enough functionality so that they are free to build any tool featuring any kind of functionality or mix of functionalities imaginable in a packet manipulation context. It aims to remain low-level to not overfit the functionalities it offers to a very specific end which could end up restricting the scope of applications that can be developed.

Scapy makes a clear distinction between "decoding" and "interpreting". Decoding is about translating information from a machine level to a human readable level, based on protocol specifications. At the end, the results are facts and correspond to raw information as is. Interpretation corresponds to the illations made over that information. Oftentimes, port scanning utilities will say that "port X is open" instead of "received a SYN-ACK from port X". The first is an interpretation of what is actually going on, at a protocol level, which might not always be reliable and actually end up introducing a bias while the second one is a product of decoding the information received from a host on the network, sent in binary form.

Scapy also aims to make all accessible and decodable information from a packet available to the user. For instance, a reduced number of tools give information on the Ethernet padding.

"Probe once, interpret many times" is an inherent methodology for designing Scapy. While many tools are built around one point of view and end up discarding data which might not be directly related to that point of view, Scapy takes a more broad approach and limits itself to making available the complete raw data, so that it can be interpreted many times, under different points of view. This way, the complete packet data remains available, no matter what the point of view is being applied and can always be accessed even when a different point of view is needed.

dpkt dpkt [97] is an open-source library for the crafting and parsing of packets. It supports the basic TCP/IP protocols. Initially created by Dug Song, it is now being maintained and improved by an extended set of contributors and developers.

PyPcapKit PyPcapKit [98] is an open-source project, available as both a standalone app and as a library. It focuses exclusively on the parsing of pcap files. Unlike other libraries, PyPcapKit uses a streaming strategy for reading input files i.e. packets are read frame by frame, using less memory. Format of file output is supported by dictdumper, an utility which receives dicts and that is able to output different formats like JSON, PLIST, XML and others.

Pcap parsing is performed by a custom extraction engine, built for PyPcapKit. PyPcapKit also supports extraction engines of other third-party packet manipulation tools like scapy, dpkt and pyshark.

PyShark PyShark [99] is an opensource Python wrapper for tshark[2], allowing python packet parsing using wireshark dissectors. This library does not actually perform any packet parsing on its own but rather uses tshark's ability to export packet information to XML and just parses the result. Even though this might introduce an unjustifiable overhead at the light of other available solutions, one advantage of using tshark is the extensive Wireshark dissector library to which it has access, supporting many more protocols than any other tool.

It supports reading from a capture file, reading from a live interface (with or without ring buffer) and reading from a live remote interface. It also supports the use of BPF filters and display filters.

PyPacker PyPacker [100] is an open source packet parsing library, based off the dpkt tool. Unlike other tools, its more low-level, forcing the developer to manually open

[2] TShark is a terminal-based version of Wireshark

sockets for performing the capture and sending of packets. It supports creation of custom packets via keywords or from raw bytes, read and write packets from and to a pcap file, merge of multiple pcap files into one file, send and receive layer 2 packets and interception (and also modification) of packets for Man In The Middle attacks.

Table 13 - Comparison between packet parsing libraries

	Scapy [101]	Dpkt [102]	PyPcapKit [98]	PyShark [99]	PyPacker [100]
pcap file parsing	✓	✓	✓	✓	✓
Packet crafting	✓	✓	✗	✗	✓
Support for BPF filter	✓	✗	✗	✓	✗
Support of basic TCP/IP stack protocols[3]	✓	✓	✓	✓	✓
Ability to not retain read packets in memory	✓	✗	✓	✓	✗
Performance (seconds per packet) [98]	0.000360	0.000173	0.000953	0.042009	-
GitHub stars	5.5k	737	83	1.2k	14
GitHub forks	1.3k	212	12	268	-

All libraries presented offer more or less the same functionality and while all of them could fulfill the required purpose which is to read pcap file, some are more oriented towards live capture from an interface and offer a more simplistic pcap parsing functionality than other libraries which are more focused on pcap parsing.

[3] Ethernet, IPv4, IPv6, TCP, UDP

Overall, from the analysis of Table 13, Scapy seems to be more in tune with what is needed. Scapy is of particular interest because it not only offers the ability to not retain packets in memory when parsing a pcap, considerably lowering the memory requirements of the application, but also has a very simple engine for creating packets which is ideal for testing. Adding to this, it is the most popular of all libraries which usually translates into a better documented solution and more help available by the community to solve problems.

Scapy also offers very good performance for large packet captures, as evidenced by the results from a test ran by the author of the PyPcapKit library, depicted in Table 13.

Scapy ended up being chosen to integrate the final solution.

3.2.4 Patterns and techniques

In this subsection are presented all patterns and techniques employed during the development of the solution to ensure that the solution has a long lifecycle and that it can be expanded with functionality with as little change as possible to the codebase.

Abstraction Abstraction, in OOP, is about hiding the complexity of operations behind a readily accessible and simple interface. Objects, classes and variables represent more complex underlying code and data [103]. Through abstraction, the user can make use of provided functionality without having to worry about all the hidden complexity and what kind of knowledge or dependencies were employed to make it possible. This allows for code reusability.

Inheritance Inheritance is an intrinsic property of OOP in which a hierarchy of classes can be established. Classes can extend other classes and doing so allows a child class to inherit all functionality pertaining to the parent's class public interface, including methods and attributes. A class which extends another is called often called a subclass or child class and the class which is extended is called a superclass or parent class. Once a class extends another, it is said to gain a "is-a" relationship with its parent class [104] which implicates other behavior described further ahead in the "Polymorphism" entry. Inheritance not only helps in modeling real life more realistically but also contributes to the reusability of code and its maintainability.

Encapsulation and information hiding Encapsulation and data hiding [105] are two terms which represent two distinct concepts. The concept of encapsulation is provided by the programming language while the concept of information hiding is provided by applied design decisions. Both of these concepts, however, serve a same purpose: allowing the creation of classes that are not only more resilient to change, but also more easily used by client objects.

Encapsulation refers to the bundling of data with the methods that operate on that data. Encapsulation, on its own, does not guarantee either protection of internal data or information hiding nor does it ensure cohesive class design. Encapsulation is achieved by applying a responsibility-driven design to determine how should data and the operations that perform on that data be organized.

Information hiding is a design principle that strives to shield client classes from the internal workings of a class. It is more concerned with the isolation of clients from requiring intimate knowledge of the design to use a module, and from the effects of changing those decisions. Without data hiding, a class's data items will be exposed to any of its clients, which leaves it with complete freedom to directly access and modify internal data, with no intervention from the class.

Polymorphism Polymorphism is the ability of an object to take many forms and another core concept of OOP. In the context of the class hierarchy established by inheritance, objects of different types can be accessed through the same interface. Further, each type can provide its own, independent implementation of this interface. Polymorphism can be static and take the form of method overloading which enables the creation of more than one method, within the same scope, with the same name but a different signature or dynamic, which is a property only reflected at runtime and which refers to the ability of a subclass to override a method of its superclass, meaning that on a client's perspective, nothing changes as the interface remains the same [106]. What does change is its internal functionality. At runtime, it is the responsibility of the dispatcher to figure out the type of the object and invoke the correct method.

Reflection Reflection is the integral ability for a program to observe or change its own code as well as all aspects of its programming language (syntax, semantics, or implementation), during runtime [107]. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability of a program to observe and therefore reason about its own state and structure. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data. providing such an encoding is called reification [108].

Reflection allows a program access to an object's complete interface, including a list of its methods and fields and the ability to change their implementation and values, respectively, overriding any data hiding or defensive programming in place. Reflection also gives the

possibility to invoke methods by name, query the value of a named instance variable within a specified object and define new methods on individual objects [109].

MVC As an architectural pattern, MVC aims to give a general solution to a commonly occurring problem when building software architecture [110, p. 73].

It divides an application's architecture into three components: Model, View, Controller. The advantages of this pattern is that it not only increases code readability and manageability, it also makes it easier to code, debug and test, since it can achieve high separation of concerns. The modularity conferred by this architecture to the application further simplifies its maintainability and makes expandability an operation less prone to errors.

Views are responsible for presenting content through the user interface. They receive any user input, interpret it and relay the received information to the respective controller component.

Controllers are the components that handle user interaction in conjunction with the model. They essentially establish the bridge between the views and the model and do not perform any logic of their own. Instead, they "control" the flow of the operation and delegating responsibilities to the respective model classes.

The model represents the state of the application and any business logic or operations that it requires to perform. It encapsulates business logic alongside implementation logic for persisting the state of the application [111].

Template method Template method is a behavioral design pattern introduced. Behavioral patterns are mainly concerned with algorithms and the assignment of responsibilities between objects. They describe not just patterns of objects or classes but also the patterns of communication between them [112].

The template method, in specific, suggests that the algorithm which characterizes the desired operation is to be broken down into multiple steps. Each of these steps should, in turn, be implemented into separate methods and each one should then be called, in the desired order, inside a single template method. The steps may either be abstract or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself) [113].

Factory method Factory method is a creation design pattern. Creational design patterns are concerned with establishing different mechanisms to achieve object creation, promoting flexibility and reusability of code [114].

The factory method defines an interface for the creation of objects whose types are subclasses of the same class. The class to be instantiated is defined through a unique identifier which is passed to the factory method. This contributes to making a system more independent of how its objects are created.

The application of all these patterns will make it easier to maintain and extend the code, as time goes on.

4 Solution analysis and design

This chapter comprises the requirements gathering, analysis and design phases which supported the implementation of the application. Initially, the functional and non-functional requirements obtained from discussions with the product owner are documented. Following this, the concepts of the domain identified during the analysis phase are presented, alongside the domain model and the glossary which further explains the meaning of each concept. Lastly, the design phase of the application development is described, with resource to several artifacts.

4.1 Functional and non-functional requirements

In this subsection are found the application's functional requirements, embodied in the form of Use Cases (UC), followed by the supplementary specification which describes the remaining functional requirements, not bound to any specific UC, and non-functional requirements. These requirements are the result of the discussion between the student and the project owner.

4.1.1 Use Cases

The following are a set of Use Cases in brief format which represent the functional attributes the application must fulfill.

UC1 – Obtain features from a PCAP file

The user starts the application. The application requests the path to a pcap file to be served as input to the application and the path for the output CSV file and other desirable information. The user enters the required information and confirms. The application calculates and extracts the necessary information from the input file and stores it in the provided output file path. The application confirms the success of the operation.

UC2 – Obtain features from Airbus' "Retour de Flamme" project log file

The user starts the application. The application requests the path to a "Retour de Flamme" project log file to be served as input to the application and the path for the output CSV file and other desirable information. The user enters the required information and confirms. The application calculates and extracts the necessary information from the input file and stores it in the provided output file path. The application confirms the success of the operation.

UC3 – Obtain features from an Airbus-provided Graylog log file

The user starts the application. The application requests the path to a Graylog log file to be served as input to the application and the path for the output CSV file and other desirable information. The user enters the required information and confirms. The application calculates and extracts the necessary information from the input file and stores it in the provided output file path. The application confirms the success of the operation.

UC4 – Obtain features from a CICIDS 2018 dataset file

The user starts the application. The application requests the path to a CICIDS 2018 dataset file to be served as input to the application and the path for the output CSV file. The user enters the required information and confirms. The application calculates and extracts the necessary information from the input file and stores it in the provided output file path. The application confirms the success of the operation.

In Figure 8 a visual summary of all the uses cases, the functionality they represent, and the intervening actors can be consulted. The diagram depicted in the picture is called Use Case Model.

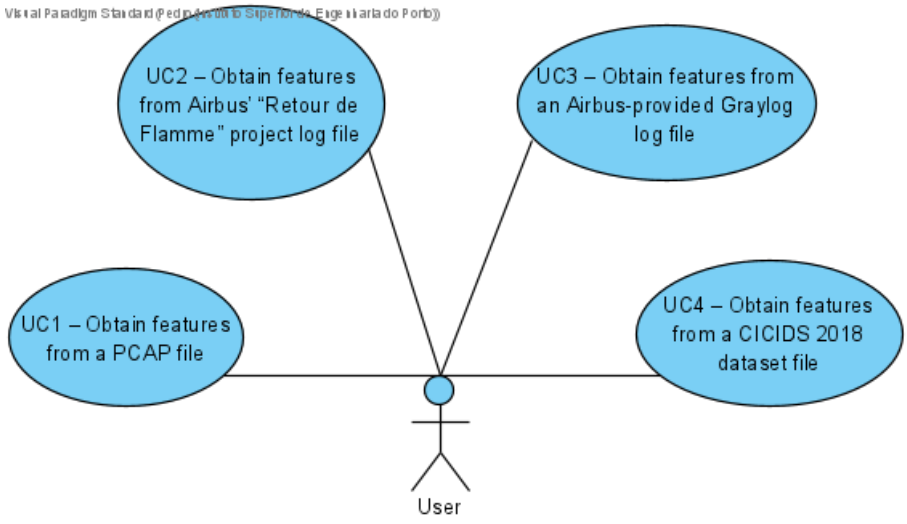


Figure 8 - Use case model

4.1.2 Supplementary specification

The remaining functional and all non-functional requirements are presented with resource to the FURPS+ [115, p. 107] model. The term "user" refers to any person using the application for its intended purpose.

Functionality

- "Help" functionalities must be available to the user.
- The user must be informed of the progress of the application's execution.
- Depending on the nature of the input field, both unidirectional and bidirectional output must be supported.
- Mechanism for labelling the received network traffic data as "malicious traffic" or "normal traffic" must be supported.
- The application must be secure and thus appropriate measures must be taken to ensure that no vulnerabilities are present.

Usability

- The application must feature a CLI.

Reliability

-

Performance

-

Supportability

- The application must run in both Windows OS and Linux OS.
- The application must support the addition of new input file formats.
- The application must support the addition of new output features.

Design constraints

- The application must reflect good design practices, including but not limited to, GRASP patterns and SOLID guidelines.

Implementation restrictions

- The application must account for large input files and thus the application is required to have a design, mindful of the time and storage complexity issues.

4.2 Problem domain

In this subsection are presented the concepts or conceptual classes identified during the analysis phase of the project. From this phase resulted two artifacts: the domain model, illustrated in Figure 9, and the glossary represented by Table 14.

The domain model is a visual representation of the concepts (or conceptual classes) and real objects pertaining to the business domain, the attributes which characterize them and the

relationships between themselves. These conceptual classes might not necessarily correspond to software classes.

The glossary gives a brief description of each of the concepts, in the context of the business domain.

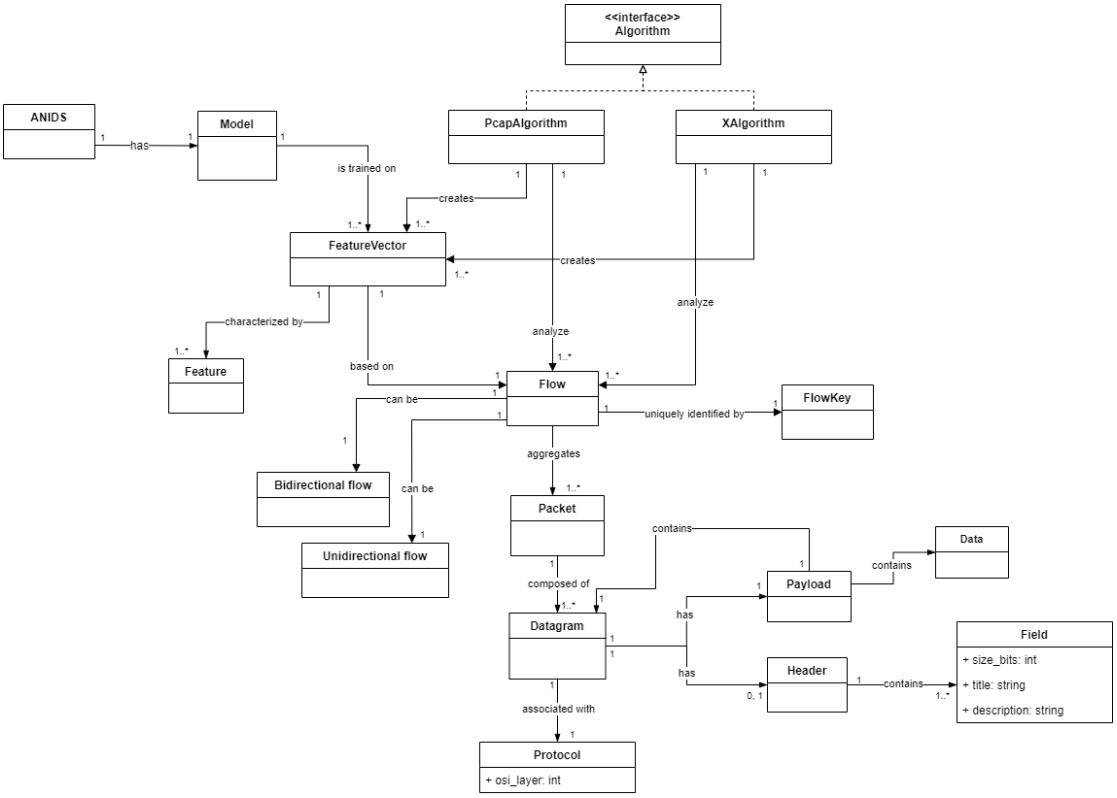


Figure 9 - Domain model

Firstly, it is important to understand the dynamic of the **Packet** concept. This concept is referring to the network packet which was defined in Section 2.4.1. Just like described in the domain model presented in Figure 9, a **Packet** is composed of one **Datagram** or more, each one usually composed of a **Header**, which will include one **Field** or more and a **Payload**. **Packets** are built based on the principle of encapsulation where **Datagrams** coming from upper layers are successively encapsulated into the payload of another **Datagram** successively lower layers of the OSI model, until it reaches the physical layer of the OSI model, becoming ready to be sent on the wire. Naturally, a **Datagram** of the most upper layer (application layer) of the OSI model will have no **Header** and just contain **Data**. Unfortunately, the domain model is not capable of properly reflecting this behavior though there was an effort to do so.

One **Feature Vector** is related to one **Flow** which, in turn, is just an aggregation of one **Packet** or more, which share the same **Flow Key**. **Flows** can either be **Bidirectional Flow** or **Unidirectional Flow**.

Preprocessing converts network traffic into a series of observations, where each observation is represented as a **Feature Vector** [12], characterized by one **Feature** or more.

As explained before, the solution developed will be bundled into a module and later fitted into a pipeline which will perform all steps of Intrusion Detection. The module which comes immediately after the one represented by the proposed solution is composed of a **Model** which is to be trained on its output i.e. one **Feature Vector** or more.

There are different **Algorithms** for each of the supported input files, namely, **PcapAlgorithm**.

The glossary defines the identified concepts in a clear way, so there are no ambiguities in what each concept means and how it related to others. The glossary can be viewed in Table 14.

Table 14 - Glossary

Concept	Description
ANIDS	Anomaly-based Intrusion Detection System corresponds to the complete system and should be capable of obtaining network traffic data, preprocess it and extract any relevant features and contain a ML model which will be trained on said network traffic so that it can later perform intrusion detection, in an independent fashion.
Model	A model is a structure which reflect a set of patterns, identified by an algorithm. A model is trained on a set of data, via an ML-based algorithm which will reason over and learn from those data [116].
Algorithm	An algorithm, in its general sense, will read or construct flows from the file received as input and calculate or extract a series of features, which will be aggregated into a feature vector.
PCAP algorithm	A concrete algorithm which will construct flows from packet traffic traces provided in pcap format.
X algorithm	Refers to any other algorithm which the program might support. Meant to represent the similar behavior between all algorithms, even those that have not yet been implemented.
Feature vector	A feature vector reflects one network flow, characterized by several features which reflect important metrics of the network traffic.
Feature	Any piece of information which can be derived from a set of network traffic data, preferably relevant for the detection of intrusions.
Flow	Flow represent an aggregation of packets which share a same key flow.

Unidirectional flow	Unidirectional flows are characterized by packets sent in one direction in a two-way communication, as is usual on networks.
Bidirectional flow	Bidirectional flows are characterized by packets sent in both directions in a two-way communication.
Flow key	A set of attributes obtainable from a packet which uniquely identify a flow. Will be differently calculated for a unidirectional and a bidirectional flow.
Packet	Basic unit of packet-switched network communication. It is made up of datagrams which are successively encapsulated inside one another.
Datagram	A datagram is composed of a header and a payload. Reflects the structure of a protocol.
Payload	Will either contain another datagram or just raw data.
Header	Contains metadata regarding the datagram to which it belongs, in the form of field/value tuples.
Field	Represents a property of a protocol whose value will heavily influence how the protocol behaves.
Protocol	A network protocol is “an established set of rules that determine how data is transmitted between different devices in the same network”. Essentially, a protocol “will allow connected devices to communicate with each other, regardless of any differences in their internal processes, structure or design” [117] .

4.3 Design

In this section, the design process, and the decisions it entailed are discussed and justified, based on the guidelines and good practices. This analysis is supported by class diagrams of different aspects of the application.

The design process followed a responsibility-driven approach, with emphasis on identification of the responsibilities, roles and collaborations to be established between objects. The design process was also heavily influenced by the GRASP and SOLID guidelines and their applicability will be brought up throughout the section, to justify certain design decisions.

However, the SOLID guidelines “Open-closed principle” and “Liskov Substitution Principle” clash with the design philosophy of Python. As encapsulation is achieved through convention in Python, software entities are never truly closed for modification. Adding to this, the Liskov Substitution Principle states that objects of a superclass might be replaced with objects of its

subclasses, without affecting application execution. However, Python is characterized by duck typing, which was explained back in Section 3.2.1 which is not compatible with that principle.

Application's architectural design is strongly based on the MVC pattern, which is explained in Section 3.2.4. In order to take full advantage of the object-oriented programming paradigm, the MVC pattern was used which not only increases the application's modularity but also improves its readability and expandability factor. The MVC pattern is easily identified by how the dependences between packages is expressed, as displayed in Figure 10.

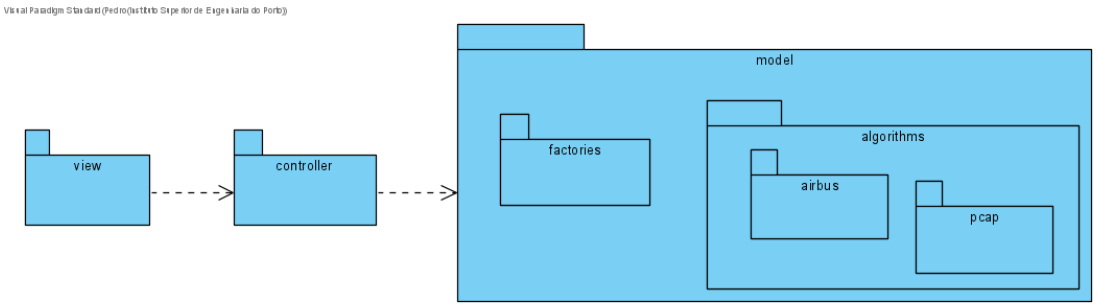


Figure 10 - Package diagram

Due to the size of the application, several class diagrams will be presented, depicting different perspectives of the application.

Starting off with the View and Controller components, due to the close nature of the functionality reflected of the defined UCs, an abstract ExtractionUI and abstract ExtractionController classes were defined, which all concrete UI and Controller classes, respectively, must extend.

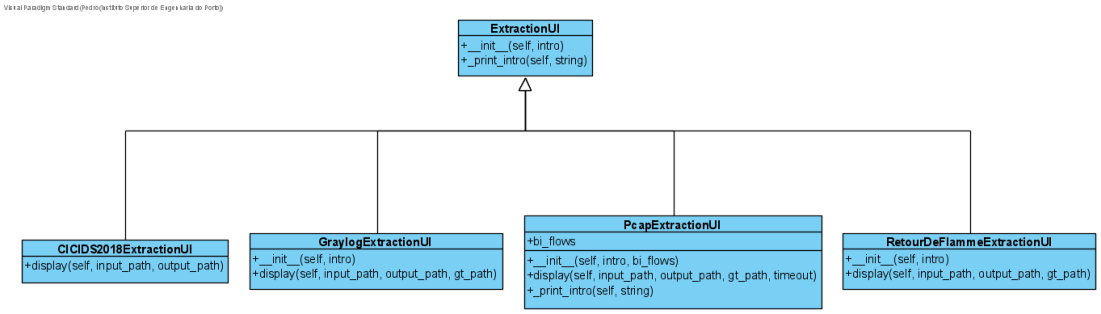


Figure 11 - Class diagram of View component

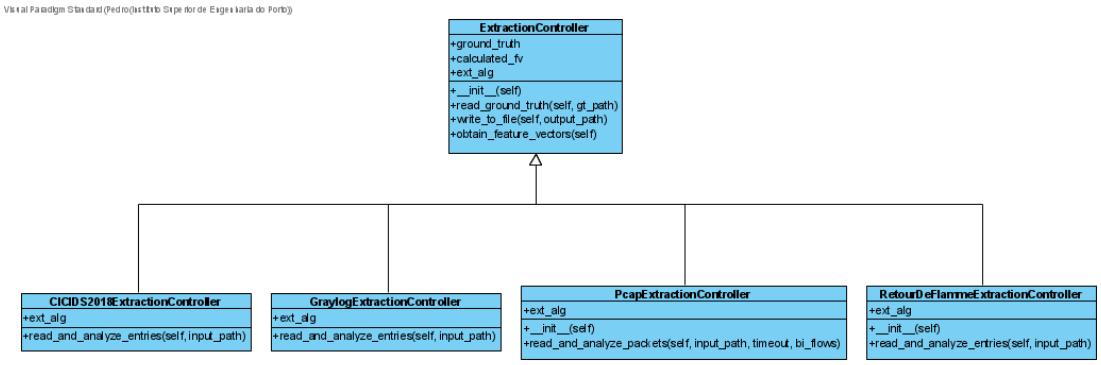


Figure 12 - Class diagram of Controller component

Even though it may seem that all UI and Controller classes could be merged into one single UI and Controller class, each file given as input will require different arguments to be indicated. For instance, when analyzing a pcap file, the user might want to indicate a timeout value which influences the flow creation process. This argument, however, is not available when attempting to analyze the CICIDS2018 dataset for example.

The MVC pattern inherently follows the “Controller” and “Pure fabrication” guidelines specified in GRASP.

Focusing now on the Model component of the established MVC architecture, it contains several conceptual classes which were featured in the domain model and that were further promoted to code classes, presented in Section 4.2, and other classes which were not envisioned during the analysis phase.

The classes which are part of the Model component are presented in the context of the class diagram for UC1 (Obtain features from a PCAP file), in particular, the scenario where the user chooses to obtain bidirectional feature vectors.

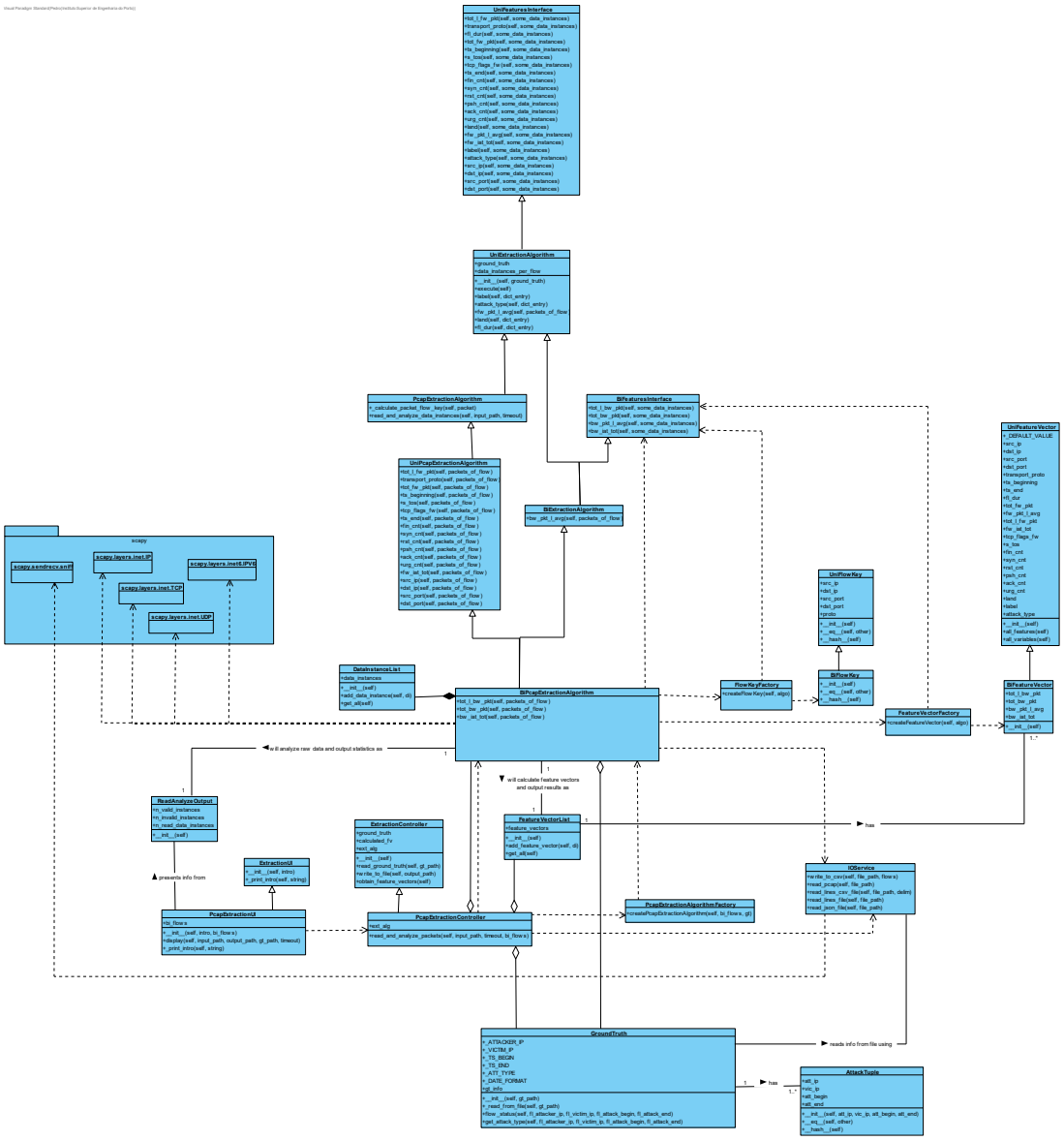


Figure 13 - Class diagram for UC1

One thing to note about the diagram is that many of the dependencies are tied to the BiPcapExtractionAlgorithm though this class gets most of its functionality from parent classes. This was strictly for aesthetic purposes and to better represent the functionality which revolves around the algorithm.

To conclude, application design tried to adhere as much as possible to the GRASP and SOLID guidelines, with a great emphasis having been placed in achieving a balance between high cohesion and low coupling, as well as designing classes with only one responsibility.

5 Solution implementation

In this section, the solution's implementation is broken down and analyzed in detail. It describes the tools that aided in the implementation process and presents, in detail, the implementation of UC1, supported by relevant code snippets and its sequence diagram. This section ends with the evaluation of the solution, with an investigation of how and if the functional and nonfunctional requirements were fulfilled and an evaluation of the application's security, through the static analysis of its source code.

5.1 Tools

The following are the set of tools which used throughout the development cycle. The chosen tools greatly improve the coding experience and provide useful features which aim to further simplify the development phase and increase productivity.

5.1.1 PyCharm IDE

PyCharm is an IDE developed by JetBrains, initially released in 2010. Being an IDE, it proposes a set of features which greatly improve the coding experience over other non-IDE alternatives such as Visual Studio Code. PyCharm's debug engine is very accessible and powerful, with support for thread debugging. Adding to this, it features code refactoring, support for different execution profiles, native unit test development, among others. PyCharm is also continually updated, having observed 3 major releases in 2020 alone. Figure 14 depicts the UI of the IDE.

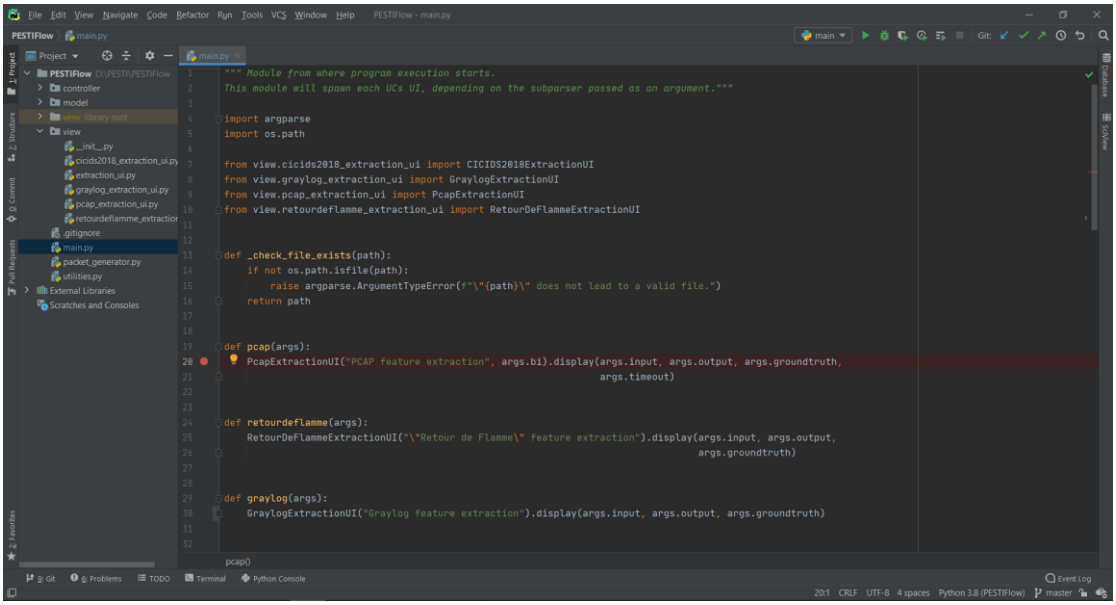


Figure 14 - PyCharm IDE UI

5.1.2 GitKraken

GitKraken is a multiplatform graphical user interface (GUI) Git client developed by Axosoft. Its UI is intuitive and simplifies all Git related functionality by presenting repository info in a clear and illustrative way. Staged and unstaged files can very easily be found with the added functionality of being able to compare the file to be committed with the latest version in the repository. The application's UI can be seen in Figure 15.

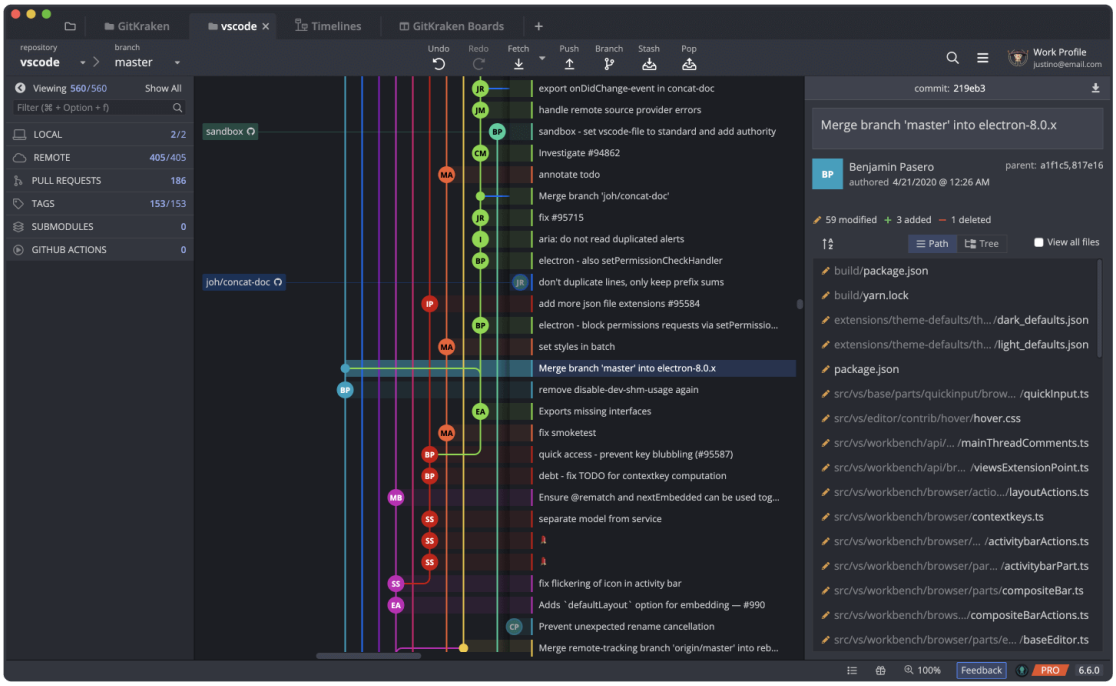


Figure 15 - GitKraken UI [118]

The application was primarily developed in a Windows 10 Education environment, in a computer with the following specifications: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 8 GB of RAM, NVIDIA GeForce GTX 1050 Ti.

5.2 Supported file formats

Although there are several formats in which network traffic data may come in, the application has support for 4 categories of file formats: pcap files, Airbus' "Retour de Flamme" scenario log files, Airbus' Graylog log files and the CICIDS2018 dataset.

Depending on the type and network traffic data they hold, bidirectional or unidirectional feature vectors will be created from them, depending on if the "-bi" flag was passed as an argument for the file types that support it.

No attempt was made to obtain unidirectional feature vectors from bidirectional network traffic data of flow type as it would be too cumbersome and a lot of information would simply not be able to be obtained due to the different nature of the data.

5.2.1 PCAP

The PCAP file format has already been addressed in Section 2.1 and both unidirectional as well as bidirectional feature vectors can be obtained from them.

5.2.2 Airbus' "Retour de Flamme" scenario log files

The Airbus' "Retour de Flamme" scenario log files were produced during the execution of the scenario. This scenario's main objective was to demonstrate the correlation of events between them by HuMa in the context of an infection of a website and a compromise of the credit card management system. No further information about this scenario was provided.

The resulting log files are believed to have been produced by a node on the network, at the time of the execution of the scenario. The log files seem to follow an arbitrarily-defined format.

In Figure 16 is presented an excerpt of the file, containing only lines with the "Suricata" string. The JSON-formatted string, which comes immediately after the "Suricata" string, is assumed to have been produced, at some point, by a Suricata IDS instance, possibly running on the same node which created the log files. Figure 17 shows other lines present in the file, which follow a different format and seem to be related to processes running on the capturing node.

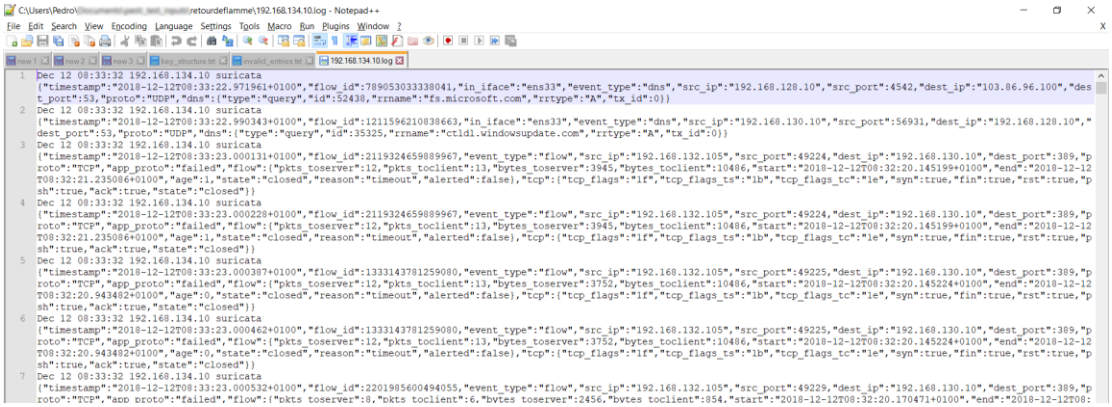


Figure 16 – Exceprt of the "Retour de flamme" log file - Suricata lines

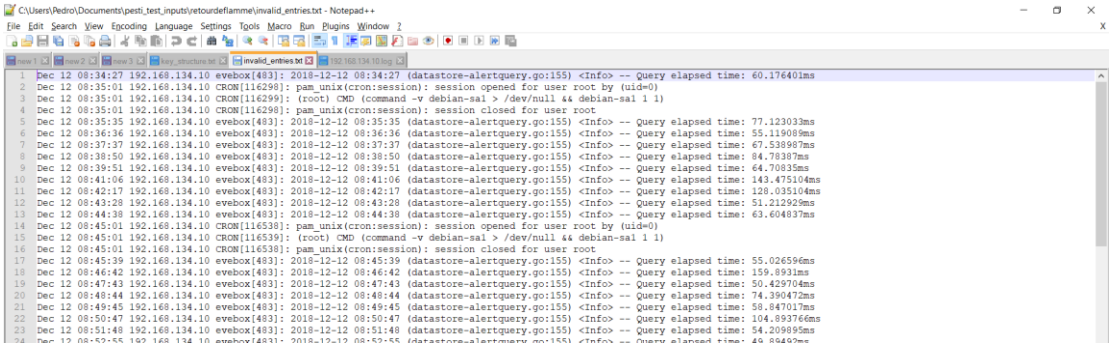


Figure 17 – Exceprt from the "Retour de flamme" log file - Other lines

The application is only concerned with the JSON-formatted string which seems to be present in every line containing the “Suricata” string. From the analysis conducted, the JSON-formatted string contains bidirectional flow information. Only one file of this category was provided, of about 26535 KB in size. Based on that file, the general key structure was obtained and can be found in Annex D, more precisely, in the D.1 subsection. The code to obtain this structure can be found in subsection D.2.

Network data information present in the "Retour de flamme" log file corresponds to bidirectional feature vectors.

5.2.3 Airbus’s Graylog files

Files from this category are obtained from Graylog, a log management tool, and follow a JSON format.

Only one file was provided, with 679 KB in size. Other than its filename which contains the string “graylog”, no more information was given regarding its origin, its format, in what context they were obtained or how they were crafted. Its general key structure can be found in Annex E, more precisely in the annex’s subsection E.1. The code to obtain this structure can be found in the annex’s subsection E.2.

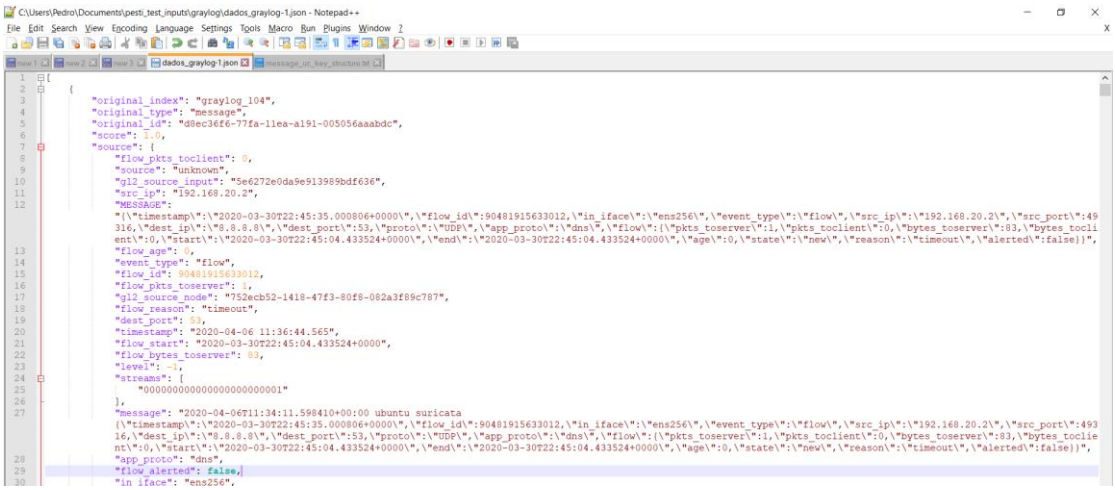


Figure 18 - Excerpt from the Graylog file

Further analysis of the file revealed that each JSON entry in the file contains duplicate information. In fact, information present in the “MESSAGE” and “message” keys (depicted in lines 12 and 27 respectively) of the top-level “source” key (depicted in line 7 of Figure 18) contain the same exact information. Moreover, keys of the aforementioned “source” key will often reflect the data present in the “MESSAGE” and “message” keys, in a more individual manner. As an example, the key “flow_bytes_toserver” (depicted in line 22 of Figure 18) which has value 83 is also present inside the value of both the “MESSAGE” and “message” keys.

Another particular property which was noticed during development was that both the JSON-formatted string present in the lines containing the “Suricata” string of the provided file belonging to the file category described in section 755.2.2 and the values (which are also JSON-formatted strings) of the “MESSAGE” and “message” keys of the file format being presented in the current section, follow very similar formats. This influenced the development of the application, as it will be seen further ahead.

Network data information present in Airbus’s Graylog files corresponds to bidirectional feature vectors.

5.2.4 CICIDS 2018

The CICIDS 2018 dataset was one of the datasets which was thoroughly studied in Section 3.1.2.

Focusing on the format of the files available in the dataset, all files follow a CSV format, with the comma being used as a separator. The first line is reserved for headers. Each column corresponds to one feature, making each line correspond to one feature vector. The order of the columns can be consulted in Annex F.

Network data information present in files of the CICIDS 2018 dataset corresponds to bidirectional feature vectors.

5.3 Supported features

Two sources of knowledge were considered for the study and identification of relevant features for ML-based intrusion detection: feature studies and datasets. Both these sources were extensively analyzed in Section 3.

The final set of features chosen can be found in Table 15 .

Table 15 - Features supported by the app

Feature	Description	# of datasets where it is present
dst_port	Destination port	16
tot_l_fw_pkt	Total size of packets (in bytes) sent in forward direction	16
transport_proto	Transport layer protocol	15
src_ip	Source IP address	14
src_port	Source port	14
dst_ip	Destination IP address	14
fl_dur	Flow duration	14
tot_fw_pkt	Total # of packets sent in the forward direction	13
label	Class label (attack vs normal) (different format for all datasets)	13
tot_l_bw_pkt	Total size of packets (in bytes) sent in backward direction	9
ts_beginning	Timestamp (beginning of the flow)	8
tot_bw_pkt	Total # of packets sent in the backward direction	7
sTos	Source TOS byte value	7

tcp_flags_fw	TCP flags which were set on packets sent in forward direction (ex. "FSA" for the flags FIN, SYN and ACK)	6
ts_end	Timestamp (end of the flow)	5
fin_cnt	Number of packets with FIN	5
syn_cnt	Number of packets with SYN	5
rst_cnt	Number of packets with RST	5
psh_cnt	Number of packets with PUSH	5
ack_cnt	Number of packets with ACK	5
urg_cnt	Number of packets with URG	5
rel_time	Relative time since capture start	4
land	If source (1) and destination (3)IP addresses equal and port numbers (2)(4) equal then, this variable takes value 1 else 0	4
fw_pkt_l_avg	Average size of packet in forward direction	4
Bw_pkt_l_avg	Mean size of packet in backward direction	4
fw_iat_tot	Total time between two packets sent in the forward direction	4
bw_iat_tot	Total time between two packets sent in the backward direction	4
attack_type	Type of Attack (portScan, dos, bruteForce, -)	4

Justification for using these features is based on two central criteria: number of datasets which use the same feature and if a certain feature is present or not in a feature study.

5.4 Supported protocols

In terms of supported protocols, all of the attacks which are in fact detectable by an ANIDS, as presented in Section 2.2, are carried out using the IPv4 or IPv6 network layer protocols,

over TCP or UDP transport protocols and so, only these protocols were considered in the development of the application. In terms of layer 2 protocols, only Ethernet II frames were considered.

5.5 Implementation overview

This subsection describes all aspects related to the implementation of the application, including how the UI was constructed and how the functionality of UC1 was achieved.

5.5.1 UI

Starting of with the UI, the developed application establishes interaction with the user through a CLI. The users specify arguments which can either be positional or optional. Optional arguments will always be preceded with either a single dash (-) or a double dash (--).

In order to easily parse the arguments, the argparse library, presented in Section 3.2.2 was used to define a series of parsers. In total, 7 parsers were defined: 1 base parser, 2 parent parsers and 4 parsers (which represent commands), one for each supported file format.

Firstly, the base parser is defined. This parser will aggregate all positional and optional arguments which are intended to be available, regardless of whatever file format is presented as an argument and the commands.

In the base parser, two positional arguments (which are mandatory) were defined, one for the input file path and another for the output file path.

Thirdly, 2 parent parsers are constructed. These contain arguments which give will be used in conjunction with other commands (which are explained in the next paragraph), and will trigger different functionality. In this case, one of the parent parsers provides the optional argument for indicating the path to a ground truth file and the other parent parser provides the optional argument for indicating if the output should reflect unidirectional flow data or bidirectional flow data. Parsers which inherit these, will “inherit” their arguments.

Fourthly, parsers are defined for each of the supported file formats. This will allow the user to pass in arguments which are exclusive to that file format, further improving the app’s modularity. These parsers originate what are called “commands” or “subcommands”.

The code for each of the previous steps is presented in Code Snippet 1, Code Snippet 2, Code Snippet 3 and Code Snippet 4 respectively.

```
parser = argparse.ArgumentParser(description="PESTIFlow app created by Pedro
Ribeiro (1160779@isep.ipp.pt)")
```

Code Snippet 1 - Instantiation of base parser

```
# Positional argument 1
parser.add_argument("input", help="Path to the file from where features will
be extracted.", type=_check_file_exists)

# Positional argument 2
parser.add_argument("output", help="Path to the file where calculated flows
will be stored. File will be in CSV format.", type=_check_file_exists)
```

Code Snippet 2 - Definition of positional arguments in base parser

```
# Parent parser of all file formats which support ground truth being fed into
them (or not)
gt_parser = argparse.ArgumentParser(add_help=False)

gt_parser.add_argument("-gt", "--groundtruth", help="Path to the file
containing ground truth information used to label the flows. File must be in
CSV format.", type=_check_file_exists)

# Parent parser of all file formats which support obtaining bidirectional
flows from them
bi_parser = argparse.ArgumentParser(add_help=False)

bi_parser.add_argument("-bi", action="store_true", default=False,
help="Indicates that bidirectional flows should be constructed from the input
data, instead of the default unidirectional flows.")
```

Code Snippet 3 – Definition of parent parsers

```
subparsers = parser.add_subparsers(title="Supported file formats",
help="Supported file formats for feature extraction.")

# PCAP subparser
pcap_parser = subparsers.add_parser("pcap", help="Input file follows the
tcpdump format, also known as the \"pcap\" format.", parents=[gt_parser,
bi_parser])
pcap_parser.add_argument("--timeout", help="Defines the maximum time
interval, in seconds, between two packets of the same flow. If the time
arrival time difference between the current packet and the last received
packet of the flow to which the current packet belongs to, is superior to
this value, a new flow is created from the received packet. Default value is
15s.", type=int, default=15)

pcap_parser.set_defaults(func=pcap)

# retourdeflame subparser
retourdeflame_parser = subparsers.add_parser("retourdeflamme", help="Input
file follows the format of the \"RetourDeFlamme\" files provided by Airbus.",
parents=[gt_parser])

retourdeflame_parser.set_defaults(func=retourdeflamme)

# graylog subparser
graylog_parser = subparsers.add_parser("graylog", help="Input file follows
the format of the Graylog files provided by Airbus.", parents=[gt_parser])

graylog_parser.set_defaults(func=graylog)

# CICIDS 2018 subparser
cicids2018_parser = subparsers.add_parser("cicids2018", help="Input file
corresponds to one of the files available in the CICIDS 2018 dataset.")
cicids2018_parser.set_defaults(func=cicids2018)
```

Code Snippet 4 - Definitions of the subcommands

All arguments which involve a file path are properly validated, to ensure that the file exists.

The following is an example of the usage of the application.

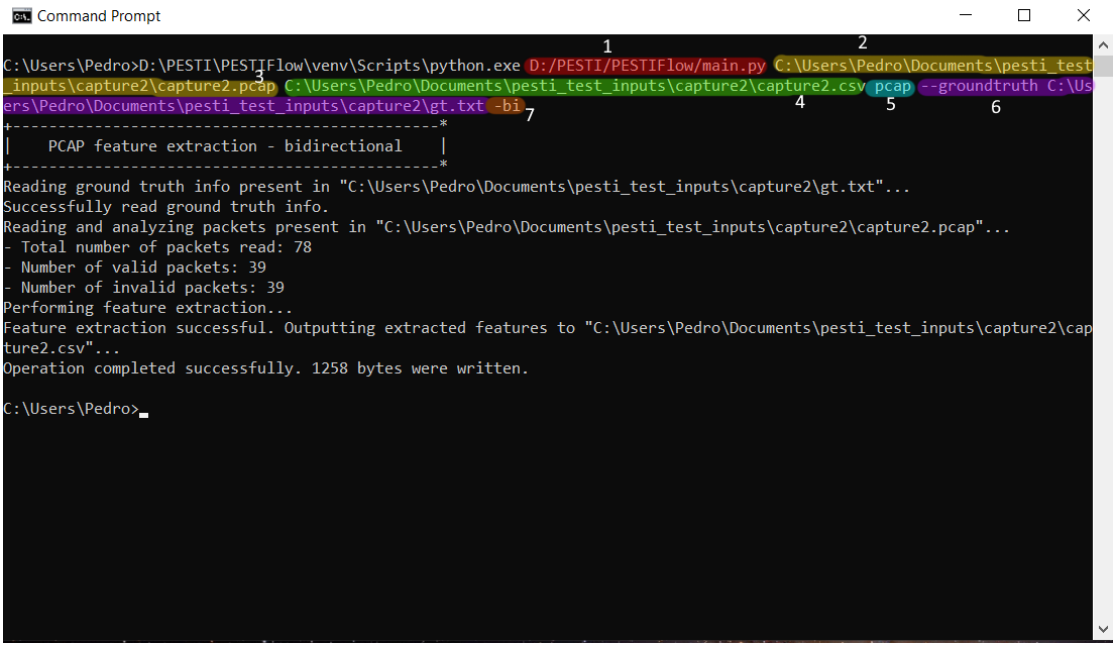


Figure 19 - Application UI

Legend:

1 – Python executable from the virtual environment established.
2 – The application's entry point, located at the root of the PyCharm project.
3 – Input file to be analyzed by the application.
4 – Output file where the results will be stored.
5 – Command corresponding to the type of file to be analyzed
6 and 7 – Optional argument obtained from parent parsers.

As it can be seen in Figure 19, the application will inform the user as it gets executed, displaying information regarding the step which is being executed.

The application also has a "-h" argument which displays the help page for a parser of the user's choosing, not only showing what arguments can be passed but also what each argument represents.

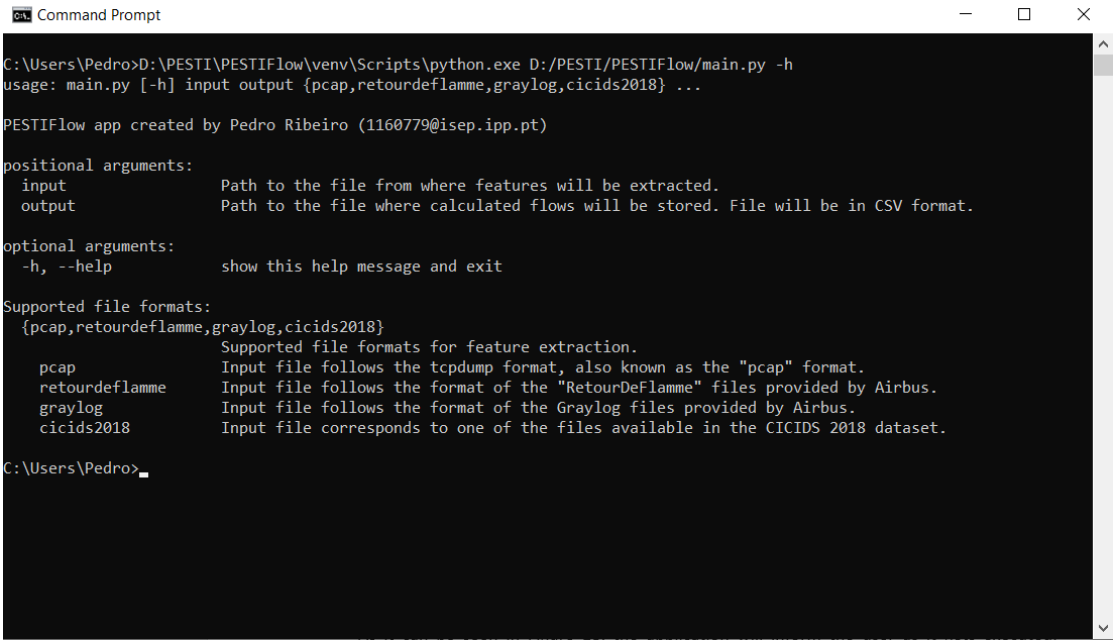


Figure 20 - Application's help page

5.5.2 Remaining implementation

Explanation of the decisions taken to implement the application will be done with resource to a sequence diagram, presented in Figure 21, representing the scenario of UC1, where the user passed the bidirectional flag as an argument. This will make it possible to explain all the steps that the application will take not only for that UC but for all the others, as they are very closely related and will operate in largely the same way.

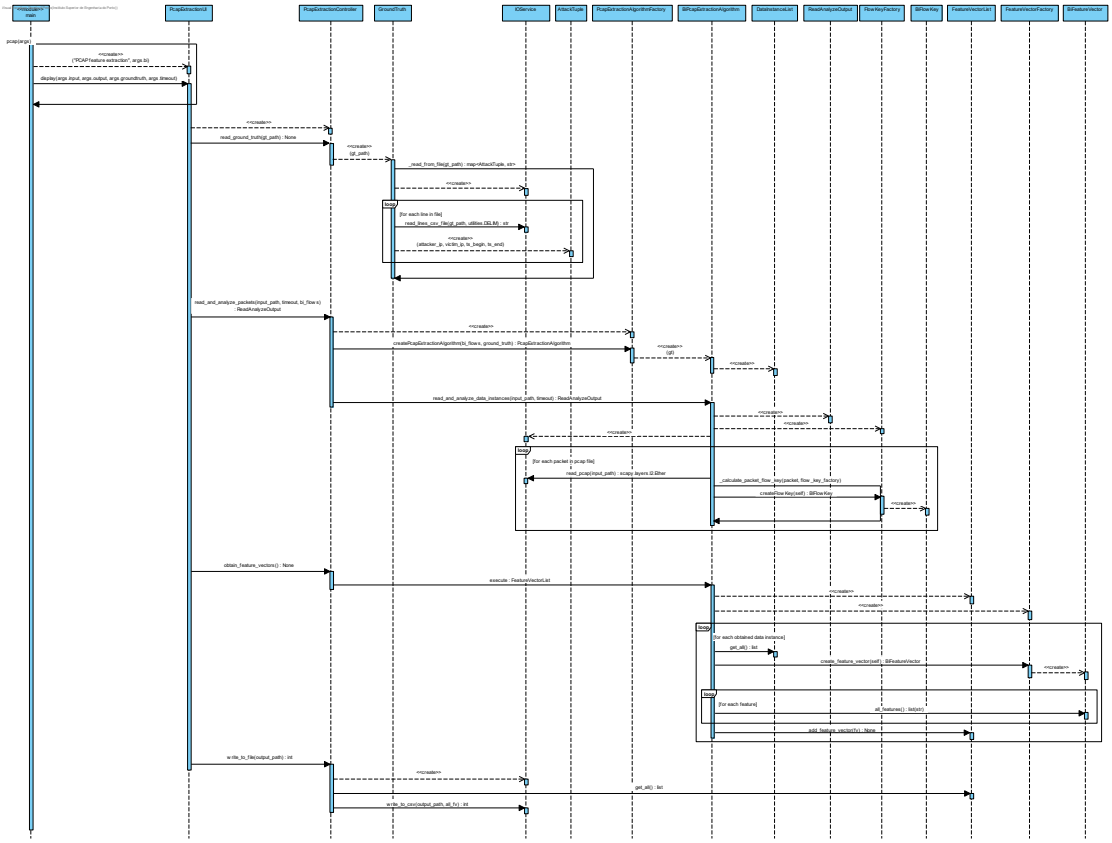


Figure 21 - Sequence diagram for UC1, with bidirectional flag passed as an argument

In the main module, depending on the subcommand passed as an argument, a function will be called which instantiates the respective UI class. In the scenario being considered, PcapExtractionUI class is instantiated and the method “display” is called on that instance.

Inside this method, the PcapExtractionController class is instantiated and several method calls are invoked on it. The functionality was spread out across various functions to allow for the return of timely info to the user, as the application gets executed. As different steps of the application functionality get executed, the user is informed that they are being executed and if they terminated successfully or not, as shown in Figure 19.

Firstly, if the user passed the “-gt” flag as an argument, a GroundTruth instance is created from the data present in the indicated ground truth file path. The instance is then stored in the controller instance.

The file must follow a CSV format and should have the fields as displayed in Figure 22.

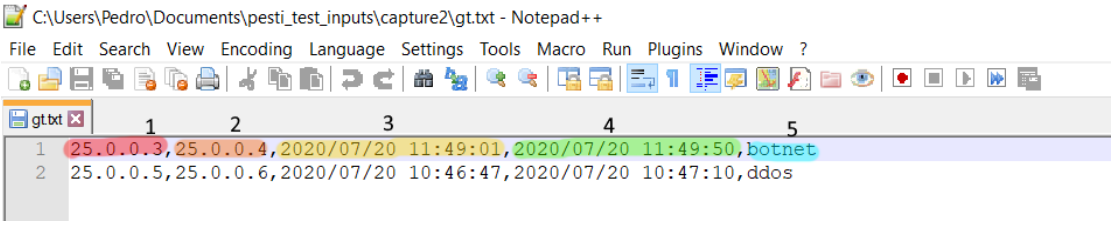


Figure 22 - Ground truth file format

Legend:

1 – Attacker IP
2 – Victim IP
3 – Timestamp, in the format YYYY/mm/dd HH:MM:SS, of the beginning of the attack
4 – Timestamp, in the format YYYY/mm/dd HH:MM:SS, of the end of the attack
5 – Attack type (optional)

By specifying a ground truth file, when flows are created or acquired from the data instances present in the input file, the output file will reflect the attacks described in the ground truth file and promptly label the corresponding feature vectors as either “Attack” or “Normal” and, if the user specified field 5 in the ground truth file, the appropriate attack type.

A GroundTruth instance will hold a dict{AttackTuple, str}. An AttackTuple instance will have 4 attributes, each one corresponding to fields 1-4, as depicted in Figure 22. The str value corresponds to field 5. The AttackTuple overrides the __eq__ and the __hash__ methods since its instances will be used as keys in a dictionary.

This, not only simplifies the process of asserting if a certain flow corresponds to an attack or not and what type of attack is reflected in that flow but also leverages the dictionary data structure and key lookup mechanism, considerably increasing performance.

In the **second** step, the packets in the pcap file whose path was passed as input are read and analyzed. The way they are analyzed, however, will depend if the user passed the “-bi” flag as an argument. To that end, PcapExtractionAlgorithmFactory class is used to produce the appropriate algorithm instance, either BiPcapExtractionAlgorithm or UniPcapExtractionAlgorithm.

The code present in Code Snippet 5 represents the PcapExtractionAlgorithmFactory class.

```
class PcapExtractionAlgorithmFactory:
    def createPcapExtractionAlgorithm(self, bi_flows, gt):
        if bi_flows:
            return BiPcapExtractionAlgorithm(gt)
        else:
            return UniPcapExtractionAlgorithm(gt)
```

Code Snippet 5 - PcapExtractionAlgorithmFactory class

Once the proper algorithm instance is obtained, it is time to read and analyze the packets.

To return the necessary information to be displayed in the UI, the ReadAnalyzeOutput class was devised. This class only contains three attributes: “n_valid_instances”, “n_invalid_instances” and “n_read_data_instances”, which represent the number of valid data instances, number of invalid data instances and the sum of the other two, respectively. In this case in specific, “n_valid_instances” represent the number of detected packets which feature the supported protocols and “n_invalid_instances” represent the number of packets of other protocols, not supported by the app.

During packet analysis, each packet’s flow key is calculated. A flow key corresponds to the “source IP – destination IP - protocol – source port – destination port” tuple and defines the flow to which a certain packet belongs. This will make it possible to group the packets by their flow keys, effectively defining flows over which feature vectors will be constructed.

There are two classes, one for each of the type of flows which can constructed: BiFlowKey which group packets under a bidirectional flow perspective and UniFlowKey which groups packets under a unidirectional flow perspective. BiFlowKey class extends the UniFlowKey class and both have the same 5 attributes, corresponding to each of the elements of the “source IP – destination IP - protocol – source port – destination port” tuple. The only difference is in their __eq__ and __hash__ methods.

This is important because instances of both these classes will be used as dictionary keys.

```
class UniFlowKey:
    def __init__(self):
        self.src_ip = None
        self.dst_ip = None
        self.src_port = None
        self.dst_port = None
        self.proto = None

    def __eq__(self, other):
        if type(other) is type(self):
            return self.__dict__ == other.__dict__
        return NotImplemented

    def __hash__(self):
        return hash((self.src_ip, self.dst_ip, self.src_port, self.dst_port,
self.proto))
```

Code Snippet 6 - UniFlowKey class

```
class BiFlowKey(UniFlowKey):

    def __init__(self):
        super().__init__()

    def __eq__(self, other):
        if type(other) is type(self):
            return self.__dict__ == other.__dict__ or \
                   (self.src_ip == other.dst_ip and
                    self.dst_ip == other.src_ip and
                    self.src_port == other.dst_port and
                    self.dst_port == other.src_port and
                    self.proto == other.proto)
        return NotImplemented

    def __hash__(self):
        return hash((self.src_ip, self.src_port)) | hash((self.dst_ip,
self.dst_port))
```

Code Snippet 7 - BiFlowKey class

Analyzing the __eq__ and the __hash__ method implementations of class UniFlowKey, as presented in Code Snippet 6 , one can conclude that UniFlowKey instances with the same value in each of the attributes will be equal (==) and will hash to the same value.

On the other hand, BiFlowKey's implementation of the __eq__ and __hash__ methods are wildly different. Analyzing their implementations, as presented in Code Snippet 7, it can be concluded that two BiFlowKey instances are equal (==) and will hash to the same value in the conditions presented in Figure 23.

Destination IP	=	Destination IP		Destination IP	=	Source IP
Source port	=	Source port		Source port	=	Destination port
Destination port	=	Destination port		Destination port	=	Source port
Protocol	=	Protocol		Protocol	=	Protocol

Figure 23 - Conditions of equality for BiFlowKey instances

This effectively results in the grouping of packets in a bidirectional way.

By defining these two classes this way, the same codebase can be used to perform the packet analysis, considerably improving code readability.

Continuing the packet analysis, once the current packet's flow key is calculated, the application will then attempt to organize the packet aggregations (which are also called "flows" throughout this section) and validate if a flow has already terminated or not.

Two flow ending conditions are checked:

1. Time-related ending condition

   A flow, independently of the protocol it represents, might be terminated after a set period of time passes after the last packet is received for that flow. This value is defined by the user in the "--timeout" command line argument. What it means is that X seconds after the last packet is received, the next packet received with the same flow key, will originate a new flow.
2. TCP exclusive condition

   A TCP flow is terminated when the connection termination handshake is established. A connection termination handshake is depicted in Figure 24 . If packets with the same flags are detected, the flow terminates and the next packet with the same flow key will originate a new flow.

   A TCP flow might also be terminated when a RST flag is detected.

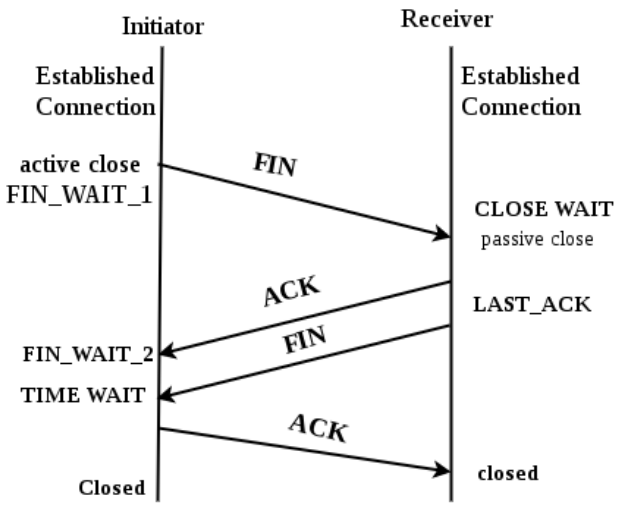


Figure 24 - TCP connection termination[119]

The packet analysis results in a DataInstanceList object which has a list of whatever objects are required for the feature vector construction which follows. This DataInstanceList object is stored inside the algorithm instance so that it can later be used for feature vector construction. The actual method returns a ReadAnalyzeOutput object.

While for pcap input file, the resulting DataInstanceList object contains a list made up of other lists containing packets, when receiving a CICIDS 2018 dataset file, the resulting DataInstanceList will contain a list of several other lists which contain strings, with each list of strings corresponding to a line of the file.

The **third** step is the construction of the feature vectors. Each feature vector will correspond to one line in the output file. Based on the ground truth information read and the DataInstanceList object obtained from the previous step, the application will now proceed to calculate all supported features.

The “obtain_feature_vectors” method of the controller is called which, in turn, calls the “execute” method of the algorithm. The code for this method is presented in Code Snippet 8.

```
def execute(self):
    all_fv = FeatureVectorList()
    factory = FeatureVectorFactory()
    for current_data_instances in self.data_instances_per_flow.get_all():
        fv = factory.create_feature_vector(self)

        for feature in fv.all_features():
            try:
                calc_current_feature = getattr(self, feature)
                if (value := calc_current_feature(current_data_instances)) is not None:
                    setattr(fv, feature, value)
            except AttributeError:
                pass
        all_fv.add_feature_vector(fv)
    return all_fv
```

Code Snippet 8 - "execute" method of the UniExtractionAlgorithm class

```
class UniFeatureVector:
    _DEFAULT_VALUE = ""

    def __init__(self) -> None:
        # Source IP
        self.src_ip = self._DEFAULT_VALUE
        # Destination IP
        self.dst_ip = self._DEFAULT_VALUE
        # Layer 3 source port
        self.src_port = self._DEFAULT_VALUE
        # Layer 3 destination port
        self.dst_port = self._DEFAULT_VALUE
        # Transport layer protocol
        self.transport_proto = self._DEFAULT_VALUE
        # Timestamp (beginning of the flow)
        self.ts_beginning = self._DEFAULT_VALUE

        ...

        # Class label (attack vs normal)
        self.label = self._DEFAULT_VALUE
        # Type of Attack (portScan, dos, bruteForce, -)
        self.attack_type = self._DEFAULT_VALUE

    def all_features(self):
        return self.__dict__.keys()

    def all_variables(self):
        return self.__dict__
```

Code Snippet 9 - UniFeatureVector class (adapted)

```
class BiFeatureVector(UniFeatureVector):

    def __init__(self):
        super().__init__()
        # Total size of packets (in bytes) sent in backward direction
        self.tot_l_bw_pkt = self._DEFAULT_VALUE
        # Total # of packets sent in the backward direction
        self.tot_bw_pkt = self._DEFAULT_VALUE
        # Mean size of packet in backward direction
        self.bw_pkt_l_avg = self._DEFAULT_VALUE
        # Total time between two packets sent in the backward direction
        self.bw_iat_tot = self._DEFAULT_VALUE
```

Code Snippet 10 - BiFeatureVector class

The method starts by instantiating the FeatureVectorList class. This object will hold all created feature vectors and is the object which will be returned at the end.

Right after, a FeatureVectorFactory object is also created. Like before, the type of feature vectors to be created will depend if the user passed the “-bi” flag as an argument or not and so, a factory class was built for the creation of either UniFeatureVector or BiFeatureVector objects. These two classes are represented in Code Snippet 9 and Code Snippet 10, respectively.

For each data instance in the DataInstanceList object obtained in the last step, there will be a BiFeatureVector object.

Each feature is calculated in a rather interesting way. The method devised, presented in Code Snippet 8, is based on reflection and introspection. All features which can be calculated in a unidirectional fashion are described as class attributes, in class UniFeatureVector. Features which are exclusive to bidirectional analysis are present in BiFeatureVector class. This class will, in turn, extend UniFeatureVector and will inherit all its attributes (or its features).

Using the "all_features" method present in UniFeatureVector class, it is possible to obtain a list of str containing the name of each supported feature.

Then, each feature name will be used to invoke a method with that same name in the algorithm object where the "execute" method was called.

To show the versatability of this architecture, the following figure shows how a method with the same name, is implemented by two different algorithms.

```python
def tot_fw_pkt(self, packets_of_flow):
    session_sender_ip = packets_of_flow[0][1].src

    ret = 0
    for packet in packets_of_flow:
        if packet[1].src == session_sender_ip:
            ret += 1
    return ret

def ts_beginning(self, packets_of_flow):
    return datetime.fromtimestamp(float(packets_of_flow[0].time))

def s_tos(self, packets_of_flow):
    first_packet = packets_of_flow[0]
    if IP in first_packet:
        return first_packet[IP].tos
    elif IPv6 in first_packet:
        return utilities.NOT_AVAILABLE
```

From UniPcapExtractionAlgorithm class

```python
def tot_fw_pkt(self, entry):
    return entry[self._TOT_FWD_PKTS]

def ts_beginning(self, entry):
    return datetime.strptime(entry[self._TIMESTAMP], self._DATE_FORMAT)

def s_tos(self, entry):
    pass
```

From CICIDS2018ExtractionAlgorithm class

Figure 25 - Comparasion of three feature method implementations

As one can see, the same type of information is returned. However, due to the different nature of the data types, it will require more or less code to calculate it or simply retrieve it from the file, as is. This functionality makes use of the Template design pattern, explained in more detail in Section 3.2.4.

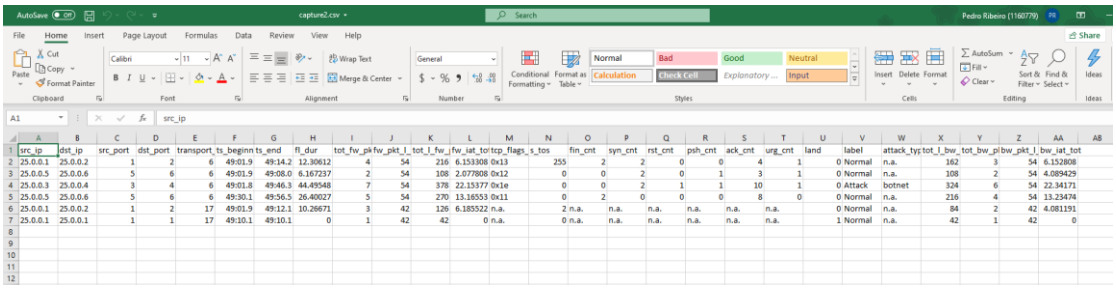
This means that when adding a new feature, only three changes are required:

1. Add the desired feature to the UniFeatureVector or BiFeatureVector (depending on the nature of the feature) as an attribute with the default value "self._DEFAULT_VALUE".

2. Add the desired feature to UniFeaturesInterface or BiFeaturesInterface (depending on the nature of the feature) as a method. This will ensure that the method to calculate said feature is implemented by the child classes.
3. As a consequence of point 2, implement the interface methods in child algorithm classes.

This makes the solution much more accessible without the need for external files to keep track of what features are supported.

The **fourth** and last step, involves the writing of the FeatureVectorList returned by the last step to a CSV file of the path passed as an argument by the user, delimited by commas. Figure 26 shows an example of expected output from the application.



src_ip	dst_ip	src_port	dst_port	transport	ts_beginn	ts_end	fl_dur	tot_fw_pk	fw_pkt_l	tot_l_fw	fw_iat_tot	tcp_flags	s_tos	fin_cnt	syn_cnt	rst_cnt	psh_cnt	ack_cnt	urg_cnt	land	label	attack_typ	tot_l_bw	tot_bw_pl	bw_pkt_l	bw_iat_tot
25.0.0.1	25.0.0.2	1	2	6	49:01.9	49:14.2	12.30612	4	54	216	6.153308	0x13	255	2	2	0	0	4	1	0	Normal	n.a.	162	3	54	6.152808
25.0.0.5	25.0.0.6	5	6	6	49:01.9	49:08.0	6.167237	2	54	108	2.077808	0x12	0	0	2	0	1	3	1	0	Normal	n.a.	108	2	54	4.089429
25.0.0.3	25.0.0.4	3	4	6	49:01.8	49:46.3	44.49548	7	54	378	22.15377	0x1e	0	0	2	1	1	10	1	0	Attack	botnet	324	6	54	22.34171
25.0.0.5	25.0.0.6	5	6	6	49:30.1	49:56.5	26.40027	5	54	270	13.16553	0x11	0	2	0	0	0	8	0	0	Normal	n.a.	216	4	54	13.23474
25.0.0.1	25.0.0.2	1	2	17	49:01.9	49:12.1	10.26671	3	42	126	6.185522	n.a.	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	0	Normal	n.a.	84	2	42	4.081191
25.0.0.1	25.0.0.1	1	1	17	49:10.1	49:10.1	0	1	42	42	0	n.a.	0	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	1	Normal	n.a.	42	1	42	0

Figure 26 - Example of application output file

The output file will have a header, with the name of each feature in each column.

All other UCs (UC2, UC3 and UC4) play out the same way. Notable changes include the UI and Controller classes, the algorithm class and the method present in the algorithm which will read and analyze the data instances offered by the file, usually following the name “read_and_analyze_*”.

5.6 Tests

In order to ensure that the both functional and non-functional requirements are fulfilled, unit tests and integrations tests were developed.

5.6.1 Unit tests

Unit tests are very low level, in terms of application testing, working over the source code. Each unit test is focused on testing the functionality of individual methods and functions of the classes, components or modules used by the software [120].

Unit tests were developed for all Model component classes of the MVC architecture. For each method in the class, several unit tests were developed, depicting both normal and edge cases.

Each class was tested in the order of their appearance when executing the application as many parts of the application depend on others.

In Figure 27, it I possible to see the structure of the modules which holds unit tests and in Figure 28, the output from running those unit tests. In total, 50 unit tests were built.

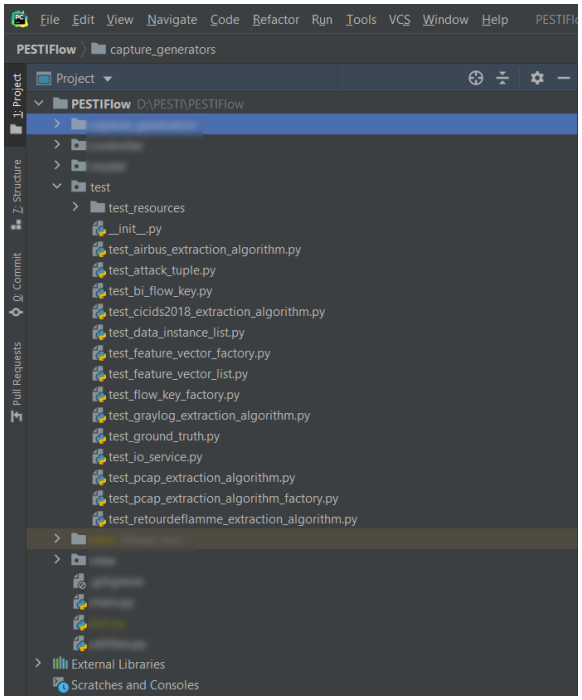


Figure 27 - Unit tests modules structure

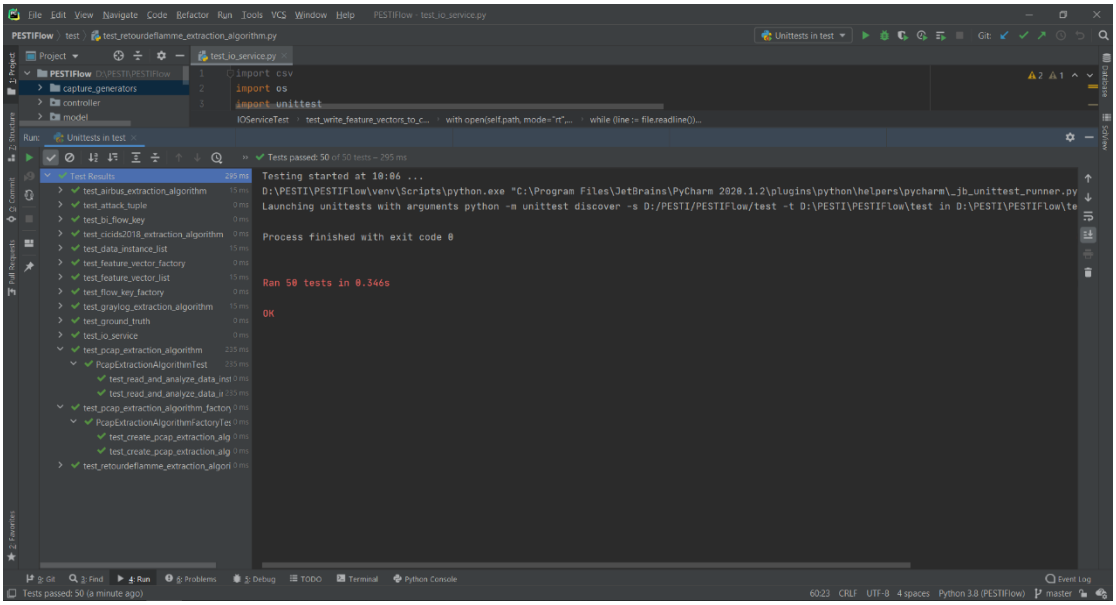
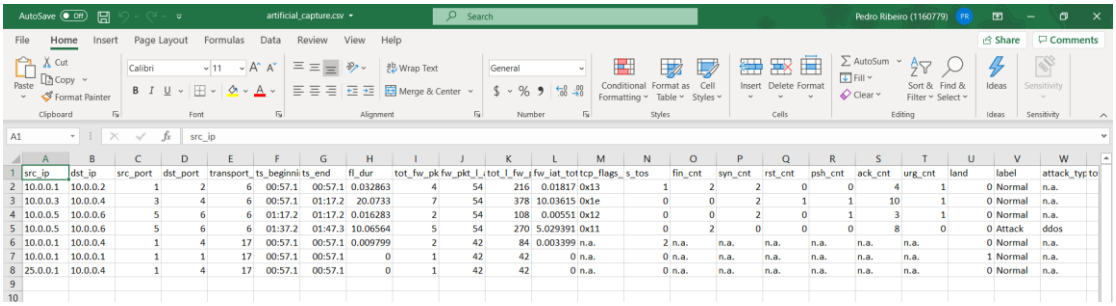


Figure 28 - Unit tests execution output

5.6.2 Functional tests

Functional tests focus on the business requirements of an application and address only the output of the application and verify if the requirements are met. It a black box-type of testing, oriented towards the functional requirements of an application where the internal implementation is not taken in consideration when performing the tests [120] [121].

To that end, the output generated by the application when fed different file formats was studied. Figure 29 through Figure 30 show the output file for when a pcap file (with the "-bi" flag set), a "Retour de Flamme" file, a Graylog file and a CICIDS2018 file, and are given as input and the arguments passed to the application, respectively.

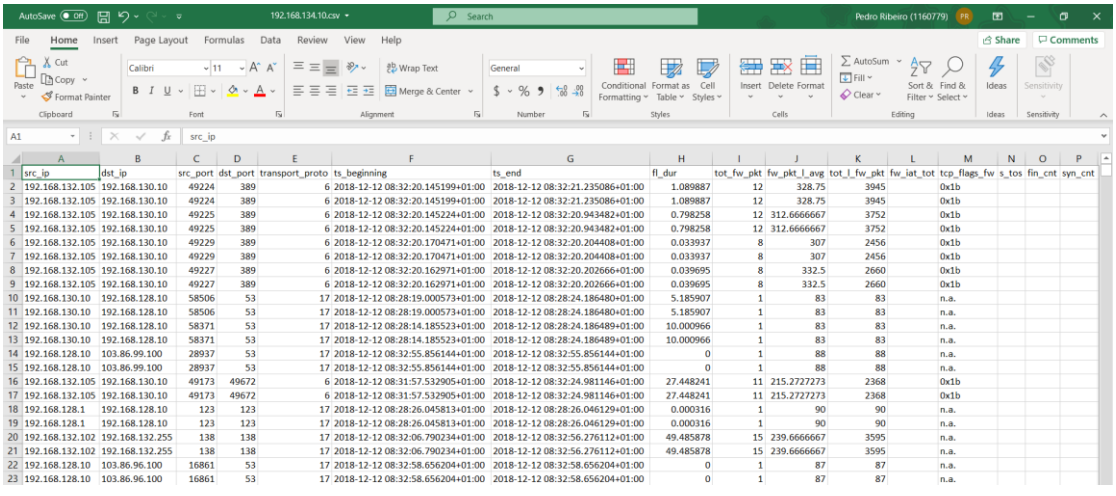


src_ip	dst_ip	src_port	dst_port	transport_	ts_beginni	ts_end	fl_dur	tot_fw_pk	fw_pkt_l_	tot_l_fw_	fw_iat_tot	tcp_flags_	s_tos	fin_cnt	syn_cnt	rst_cnt	psh_cnt	ack_cnt	urg_cnt	land	label	attack_typ	to
10.0.0.1	10.0.0.2	1	2	6	00:57.1	00:57.1	0.032863	4	54	216	0.01817	0x13	1	2	2	0	0	4	1	0	Normal	n.a.	
10.0.0.3	10.0.0.4	3	4	6	00:57.1	01:17.2	20.0733	7	54	378	10.03615	0x1e	0	0	2	1	1	10	1	0	Normal	n.a.	
10.0.0.5	10.0.0.6	5	6	6	01:17.2	01:17.2	0.016283	2	54	108	0.00551	0x12	0	0	2	0	1	3	1	0	Normal	n.a.	
10.0.0.5	10.0.0.6	5	6	6	01:37.2	01:47.3	10.06564	5	54	270	5.029391	0x11	0	2	0	0	0	8	0	0	Attack	ddos	
10.0.0.1	10.0.0.4	1	4	17	00:57.1	00:57.1	0.009799	2	42	84	0.003399	n.a.	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	0	Normal	n.a.	
10.0.0.1	10.0.0.1	1	1	17	00:57.1	00:57.1	0	1	42	42	0	n.a.	0	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	1	Normal	n.a.	
25.0.0.1	10.0.0.4	1	4	17	00:57.1	00:57.1	0	1	42	42	0	n.a.	0	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	0	Normal	n.a.	

Figure 29 - Application output when given pcap file as input

Arguments of the command line to achieve this output:

"C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\artificial_capture.pcap C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\artificial_capture.csv pcap -gt C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\gt.txt -bi"



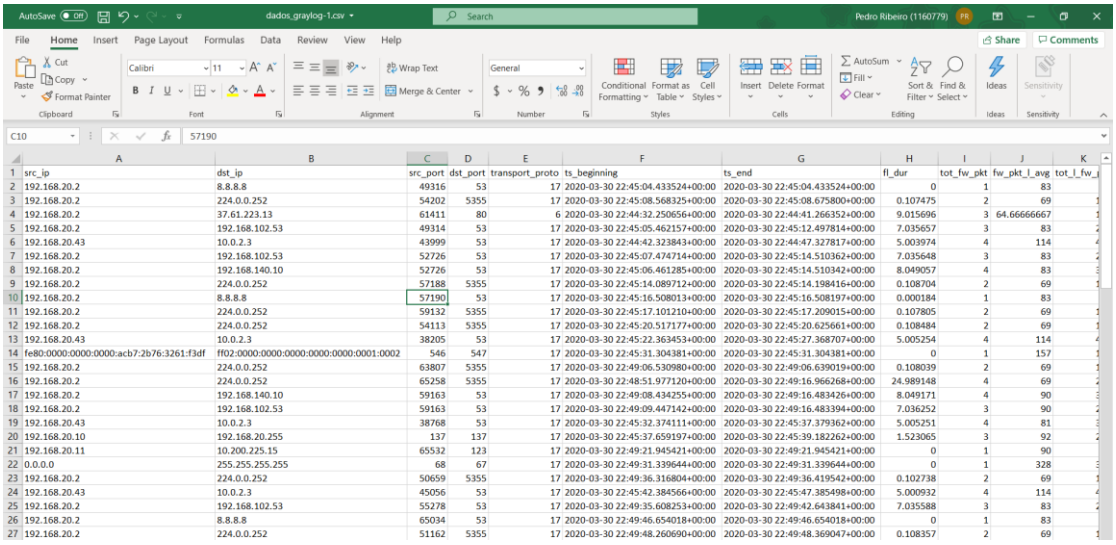
src_ip	dst_ip	src_port	dst_port	transport_proto	ts_beginning	ts_end	fl_dur	tot_fw_pkt	fw_pkt_l_avg	tot_l_fw_pkt	fw_iat_tot	tcp_flags_fw	s_tos	fin_cnt	syn_cnt
192.168.132.105	192.168.130.10	49224	389	6	2018-12-12 08:32:20.145199+01:00	2018-12-12 08:32:21.235086+01:00	1.089887	12	328.75	3945		0x1b			
192.168.132.105	192.168.130.10	49224	389	6	2018-12-12 08:32:20.145199+01:00	2018-12-12 08:32:21.235086+01:00	1.089887	12	328.75	3945		0x1b			
192.168.132.105	192.168.130.10	49225	389	6	2018-12-12 08:32:20.145224+01:00	2018-12-12 08:32:20.943482+01:00	0.798258	12	312.6666667	3752		0x1b			
192.168.132.105	192.168.130.10	49225	389	6	2018-12-12 08:32:20.145224+01:00	2018-12-12 08:32:20.943482+01:00	0.798258	12	312.6666667	3752		0x1b			
192.168.132.105	192.168.130.10	49229	389	6	2018-12-12 08:32:20.170471+01:00	2018-12-12 08:32:20.204408+01:00	0.033937	8	307	2456		0x1b			
192.168.132.105	192.168.130.10	49229	389	6	2018-12-12 08:32:20.170471+01:00	2018-12-12 08:32:20.204408+01:00	0.033937	8	307	2456		0x1b			
192.168.132.105	192.168.130.10	49227	389	6	2018-12-12 08:32:20.162971+01:00	2018-12-12 08:32:20.202666+01:00	0.039695	8	332.5	2660		0x1b			
192.168.132.105	192.168.130.10	49227	389	6	2018-12-12 08:32:20.162971+01:00	2018-12-12 08:32:20.202666+01:00	0.039695	8	332.5	2660		0x1b			
192.168.130.10	192.168.128.10	58506	53	17	2018-12-12 08:28:19.000573+01:00	2018-12-12 08:28:24.186480+01:00	5.185907	1	83	83		n.a.			
192.168.130.10	192.168.128.10	58506	53	17	2018-12-12 08:28:19.000573+01:00	2018-12-12 08:28:24.186480+01:00	5.185907	1	83	83		n.a.			
192.168.130.10	192.168.128.10	58371	53	17	2018-12-12 08:28:14.185523+01:00	2018-12-12 08:28:24.186489+01:00	10.000966	1	83	83		n.a.			
192.168.130.10	192.168.128.10	58371	53	17	2018-12-12 08:28:14.185523+01:00	2018-12-12 08:28:24.186489+01:00	10.000966	1	83	83		n.a.			
192.168.128.10	103.86.99.100	28937	53	17	2018-12-12 08:32:55.856144+01:00	2018-12-12 08:32:55.856144+01:00	0	1	88	88		n.a.			
192.168.128.10	103.86.99.100	28937	53	17	2018-12-12 08:32:55.856144+01:00	2018-12-12 08:32:55.856144+01:00	0	1	88	88		n.a.			
192.168.132.105	192.168.130.10	49173	49672	6	2018-12-12 08:31:57.532905+01:00	2018-12-12 08:32:24.981146+01:00	27.448241	11	215.2727273	2368		0x1b			
192.168.132.105	192.168.130.10	49173	49672	6	2018-12-12 08:31:57.532905+01:00	2018-12-12 08:32:24.981146+01:00	27.448241	11	215.2727273	2368		0x1b			
192.168.128.1	192.168.128.10	123	123	17	2018-12-12 08:28:26.045813+01:00	2018-12-12 08:28:26.046129+01:00	0.000316	1	90	90		n.a.			
192.168.128.1	192.168.128.10	123	123	17	2018-12-12 08:28:26.045813+01:00	2018-12-12 08:28:26.046129+01:00	0.000316	1	90	90		n.a.			
192.168.132.102	192.168.132.255	138	138	17	2018-12-12 08:32:06.790234+01:00	2018-12-12 08:32:56.276112+01:00	49.485878	15	239.6666667	3595		n.a.			
192.168.132.102	192.168.132.255	138	138	17	2018-12-12 08:32:06.790234+01:00	2018-12-12 08:32:56.276112+01:00	49.485878	15	239.6666667	3595		n.a.			
192.168.128.10	103.86.96.100	16861	53	17	2018-12-12 08:32:58.656204+01:00	2018-12-12 08:32:58.656204+01:00	0	1	87	87		n.a.			
192.168.128.10	103.86.96.100	16861	53	17	2018-12-12 08:32:58.656204+01:00	2018-12-12 08:32:58.656204+01:00	0	1	87	87		n.a.			

Figure 30 - Application output when given a "Retour de Flamme" file as input

Arguments of the command line to achieve this output:

"C:\Users\Pedro\Documents\pesti_test_inputs\retourdeflamme\192.168.134.10.log

C:\Users\Pedro\Documents\pesti_test_inputs\retourdeflamme\192.168.134.10.csv retourdeflamme”

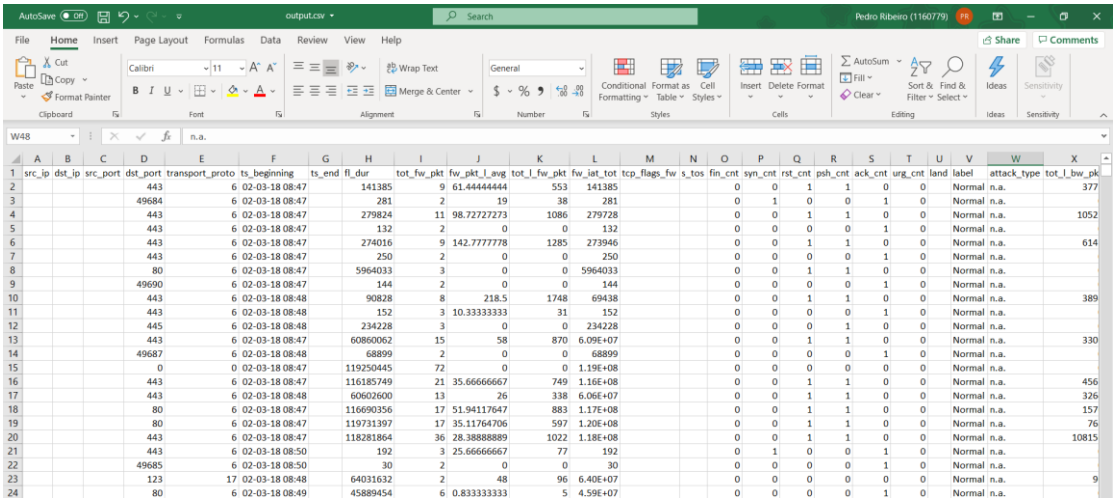


	A	B	C	D	E	F	G	H	I	J	K
1	src_ip	dst_ip	src_port	dst_port	transport_proto	ts_beginning	ts_end	fl_dur	tot_fw_pkt	fw_pkt_l_avg	tot_l_fw_
2	192.168.20.2	8.8.8.8	49316	53	17	2020-03-30 22:45:04.433524+00:00	2020-03-30 22:45:04.433524+00:00	0	1	83	
3	192.168.20.2	224.0.0.252	54202	5355	17	2020-03-30 22:45:08.568325+00:00	2020-03-30 22:45:08.675800+00:00	0.107475	2	69	
4	192.168.20.2	37.61.223.13	61411	80	6	2020-03-30 22:44:32.250656+00:00	2020-03-30 22:44:41.266352+00:00	9.015696	3	64.66666667	
5	192.168.20.2	192.168.102.53	49314	53	17	2020-03-30 22:45:05.462157+00:00	2020-03-30 22:45:12.497814+00:00	7.035657	3	83	
6	192.168.20.43	10.0.2.3	43999	53	17	2020-03-30 22:44:42.323843+00:00	2020-03-30 22:44:47.327817+00:00	5.003974	4	114	
7	192.168.20.2	192.168.102.53	52726	53	17	2020-03-30 22:45:07.474714+00:00	2020-03-30 22:45:14.510362+00:00	7.035648	3	83	
8	192.168.20.2	192.168.140.10	52726	53	17	2020-03-30 22:45:06.461285+00:00	2020-03-30 22:45:14.510342+00:00	8.049057	4	83	
9	192.168.20.2	224.0.0.252	57188	5355	17	2020-03-30 22:45:14.089712+00:00	2020-03-30 22:45:14.198416+00:00	0.108704	2	69	
10	192.168.20.2	8.8.8.8	57190	53	17	2020-03-30 22:45:16.508013+00:00	2020-03-30 22:45:16.508197+00:00	0.000184	1	83	
11	192.168.20.2	224.0.0.252	59132	5355	17	2020-03-30 22:45:17.101210+00:00	2020-03-30 22:45:17.209015+00:00	0.107805	2	69	
12	192.168.20.2	224.0.0.252	54113	5355	17	2020-03-30 22:45:20.517177+00:00	2020-03-30 22:45:20.625661+00:00	0.108484	2	69	
13	192.168.20.43	10.0.2.3	38205	53	17	2020-03-30 22:45:22.363453+00:00	2020-03-30 22:45:27.368707+00:00	5.005254	4	114	
14	fe80:0000:0000:0000:acb7:2b76:3261:f3df	ff02:0000:0000:0000:0000:0000:0001:0002	546	547	17	2020-03-30 22:45:31.304381+00:00	2020-03-30 22:45:31.304381+00:00	0	1	157	
15	192.168.20.2	224.0.0.252	63807	5355	17	2020-03-30 22:49:06.530980+00:00	2020-03-30 22:49:06.639019+00:00	0.108039	2	69	
16	192.168.20.2	224.0.0.252	65258	5355	17	2020-03-30 22:48:51.977120+00:00	2020-03-30 22:49:16.966268+00:00	24.989148	4	69	
17	192.168.20.2	192.168.140.10	59163	53	17	2020-03-30 22:49:08.434255+00:00	2020-03-30 22:49:16.483426+00:00	8.049171	4	90	
18	192.168.20.2	192.168.102.53	59163	53	17	2020-03-30 22:49:09.447142+00:00	2020-03-30 22:49:16.483394+00:00	7.036252	3	90	
19	192.168.20.43	10.0.2.3	38768	53	17	2020-03-30 22:45:32.374111+00:00	2020-03-30 22:45:37.379362+00:00	5.005251	4	81	
20	192.168.20.10	192.168.20.255	137	137	17	2020-03-30 22:45:37.659197+00:00	2020-03-30 22:45:39.182262+00:00	1.523065	3	92	
21	192.168.20.11	10.200.225.15	65532	123	17	2020-03-30 22:49:21.945421+00:00	2020-03-30 22:49:21.945421+00:00	0	1	90	
22	0.0.0.0	255.255.255.255	68	67	17	2020-03-30 22:49:31.339644+00:00	2020-03-30 22:49:31.339644+00:00	0	1	328	
23	192.168.20.2	224.0.0.252	50659	5355	17	2020-03-30 22:49:36.316804+00:00	2020-03-30 22:49:36.419542+00:00	0.102738	2	69	
24	192.168.20.43	10.0.2.3	45056	53	17	2020-03-30 22:45:42.384566+00:00	2020-03-30 22:45:47.385498+00:00	5.000932	4	114	
25	192.168.20.2	192.168.102.53	55278	53	17	2020-03-30 22:49:35.608253+00:00	2020-03-30 22:49:42.643841+00:00	7.035588	3	83	
26	192.168.20.2	8.8.8.8	65034	53	17	2020-03-30 22:49:46.654018+00:00	2020-03-30 22:49:46.654018+00:00	0	1	83	
27	192.168.20.2	224.0.0.252	51162	5355	17	2020-03-30 22:49:48.260690+00:00	2020-03-30 22:49:48.369047+00:00	0.108357	2	69	

Figure 31 - Application output when given a Graylog file as input

Arguments of the command line to achieve this output:

“C:\Users\Pedro\Documents\pesti_test_inputs\graylog\dados_graylog-1.json C:\Users\Pedro\Documents\pesti_test_inputs\graylog\dados_graylog-1.csv graylog”



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
1	src_ip	dst_ip	src_port	dst_port	transport_proto	ts_beginning	ts_end	fl_dur	tot_fw_pkt	fw_pkt_l_avg	tot_l_fw_pkt	fw_iat_tot	tcp_flags_fw	s_tos	fin_cnt	syn_cnt	rst_cnt	psh_cnt	ack_cnt	urg_cnt	land	label	attack_type	tot_l_bw_pk
2				443	6	02-03-18 08:47		141385	9	61.44444444	553	141385			0	0	1	1	0	0		Normal	n.a.	377
3				49684	6	02-03-18 08:47		281	2	19	38	281			0	1	0	0	1	0		Normal	n.a.	
4				443	6	02-03-18 08:47		279824	11	98.72727273	1086	279728			0	0	1	1	0	0		Normal	n.a.	1052
5				443	6	02-03-18 08:47		132	2	0	0	132			0	0	0	0	1	0		Normal	n.a.	
6				443	6	02-03-18 08:47		274016	9	142.7777778	1285	273946			0	0	1	1	0	0		Normal	n.a.	614
7				443	6	02-03-18 08:47		250	2	0	0	250			0	0	0	0	1	0		Normal	n.a.	
8				80	6	02-03-18 08:47		5964033	3	0	0	5964033			0	0	1	1	0	0		Normal	n.a.	
9				49690	6	02-03-18 08:47		144	2	0	0	144			0	0	0	0	1	0		Normal	n.a.	
10				443	6	02-03-18 08:48		90828	8	218.5	1748	69438			0	0	1	1	0	0		Normal	n.a.	389
11				443	6	02-03-18 08:48		152	3	10.33333333	31	152			0	0	0	0	1	0		Normal	n.a.	
12				445	6	02-03-18 08:48		234228	3	0	0	234228			0	0	0	1	0	0		Normal	n.a.	
13				443	6	02-03-18 08:47		60860062	15	58	870	6.09E+07			0	0	1	1	0	0		Normal	n.a.	330
14				49687	6	02-03-18 08:48		68899	2	0	0	68899			0	0	0	0	1	0		Normal	n.a.	
15				0	0	02-03-18 08:47		119250445	72	0	0	1.19E+08			0	0	0	0	0	0		Normal	n.a.	
16				443	6	02-03-18 08:47		116185749	21	35.66666667	749	1.16E+08			0	0	1	1	0	0		Normal	n.a.	456
17				443	6	02-03-18 08:48		60602600	13	26	338	6.06E+07			0	0	1	1	0	0		Normal	n.a.	326
18				80	6	02-03-18 08:47		116690356	17	51.94117647	883	1.17E+08			0	0	1	1	0	0		Normal	n.a.	157
19				80	6	02-03-18 08:47		119731397	17	35.11764706	597	1.20E+08			0	0	1	1	0	0		Normal	n.a.	76
20				443	6	02-03-18 08:47		118281864	36	28.38888889	1022	1.18E+08			0	0	1	1	0	0		Normal	n.a.	10815
21				443	6	02-03-18 08:50		192	3	25.66666667	77	192			0	1	0	0	1	0		Normal	n.a.	
22				49685	6	02-03-18 08:50		30	2	0	0	30			0	0	0	0	1	0		Normal	n.a.	
23				123	17	02-03-18 08:48		64031632	2	48	96	6.40E+07			0	0	0	0	0	0		Normal	n.a.	9
24				80	6	02-03-18 08:49		45889454	6	0.833333333	5	4.59E+07			0	0	0	0	1	0		Normal	n.a.	

Figure 32 - Application output when given a CICIDS2018 file as input

Arguments of the command line to achieve this output:

“C:\Users\Pedro\Documents\pesti_test_inputs\cicids2018\Friday-02-03-2018_TrafficForML_CICFlowMeter.csv C:\Users\Pedro\Documents\pesti_test_inputs\cicids2018\output.csv cicids2018”

The output provided by each one was manually checked against expected results, to ensure that the obtained results are the correct ones.

For ensuring that the expected results were being obtained when giving a pcap file as input to the application, an artificial capture, with customized packets, was built using the Scapy library, the same library used for accessing packets in a pcap file, and captured with Wireshark. The code used to create the packets can be seen in Annex G. The output presented in Figure 29 was obtained from this artificial capture.

5.6.3 Integration tests

Integration tests verify that the different units, which compose the application, work well together [120].

To that end, each UC was executed and the values displayed by the application's UI, after having finished execution, were checked against the expected values, described in Table 16 . The information displayed by the application's UI, for each UC being executed, is also checked for any errors which might indicate a failure to integrate the different code units. Figures Figure 33 - Figure 37 show the application's UI for UC1 through UC4, respectively. There are two figures relating to UC1, one where the "-bi" flag is passed as an argument and one where it is not.

The same artificial capture crafted for the functional test where the application receives a pcap file as input, presented in the previous section, was also used to for UC1's integration test.

Table 16 - Integration tests summary

UC	Arguments	Expected results	Obtained results
UC1	C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\artificial_capture.pcap C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\artificial_capture.csv pcap -gt C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\gt.txt -bi	Total number of packets read: 39; Number of valid entries: 39; Number of invalid entries: 0; Bytes written: ~1k	Equal (see Figure 33)
	C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\artificial_capture.pcap C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\artificial_capture.csv pcap -gt C:\Users\Pedro\Documents\pesti_test_inputs\artificial_capture\gt.txt	Total number of packets read: 39; Number of valid entries: 39; Number of invalid entries: 0; Bytes written: ~2k	Equal (see Figure 34)
UC2	C:\Users\Pedro\Documents\pesti_test_inputs\retourdeflamme\192.168.134.10.log C:\Users\Pedro\Documents\pesti_test_inputs\retourdeflamme\192.168.134.10.csv retourdeflamme	Total number of packets read: 35286; Number of valid entries: 10250; Number of invalid entries: 25036; Bytes written: ~1.6M	Equal (see Figure 35)
UC3	C:\Users\Pedro\Documents\pesti_test_inputs\graylog\dados_graylog-1.json C:\Users\Pedro\Documents\pesti_test_inputs\graylog\dados_graylog-1.csv graylog	Total number of packets read: 232; Number of valid entries: 90; Number of invalid entries: 142; Bytes written: ~14k	Equal (see Figure 36)

UC4	C:\Users\Pedro\Documents\pesti_test_inputs\cicids2018\Friday-02-03-2018_TrafficForML_CICFlowMeter.csv C:\Users\Pedro\Documents\pesti_test_inputs\cicids2018\output.csv cicids2018	Total number of packets read: 1048576; Number of valid entries: 1048575; Number of invalid entries: 1; Bytes written: ~100M	Equal (see Figure 37)

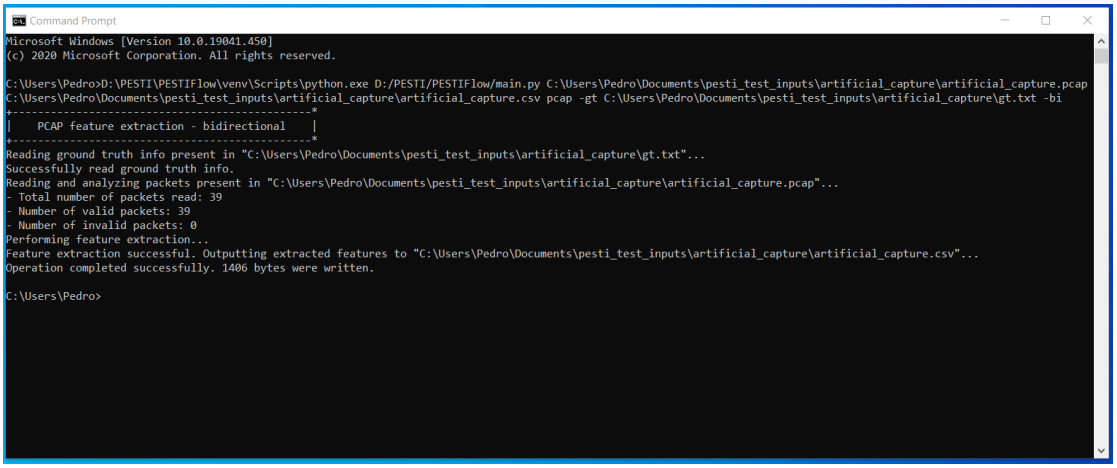


Figure 33 - Application's UI when given pcap file as input (with -bi flag) (UC1)

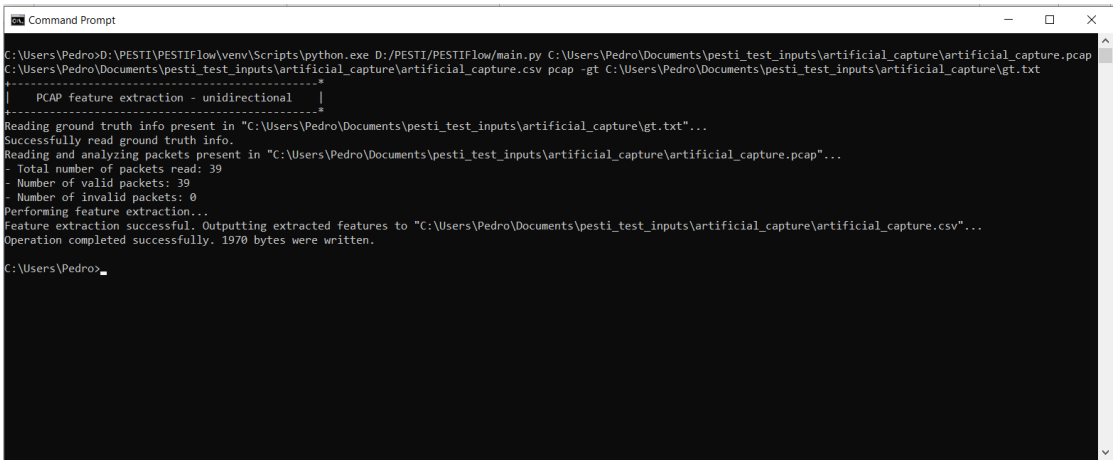


Figure 34 - Application's UI when given pcap file as input (without -bi flag) (UC1)

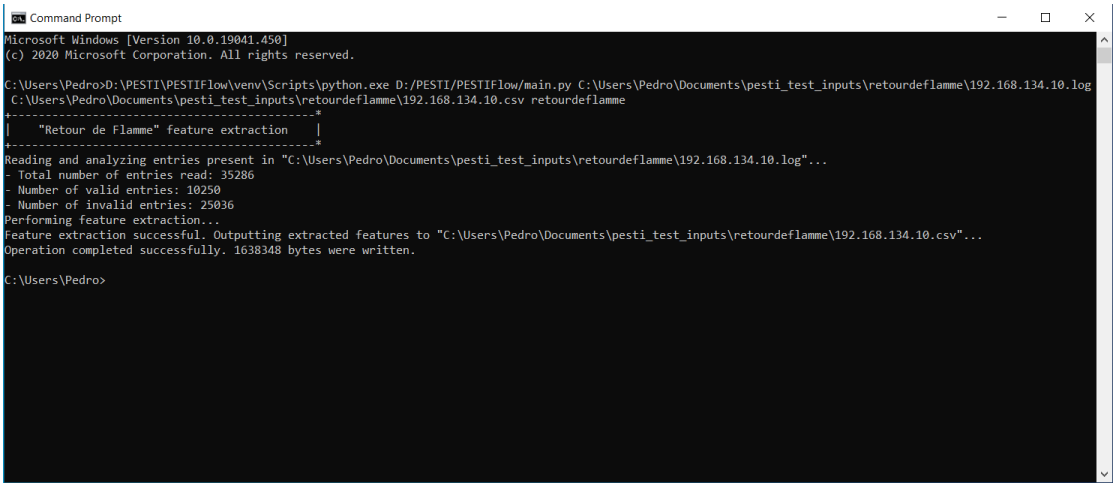


Figure 35 - Application's UI when given "Retour de flamme" scenario file as input (UC2)

Figure 36 - Application's UI when given Graylog file as input (UC3)

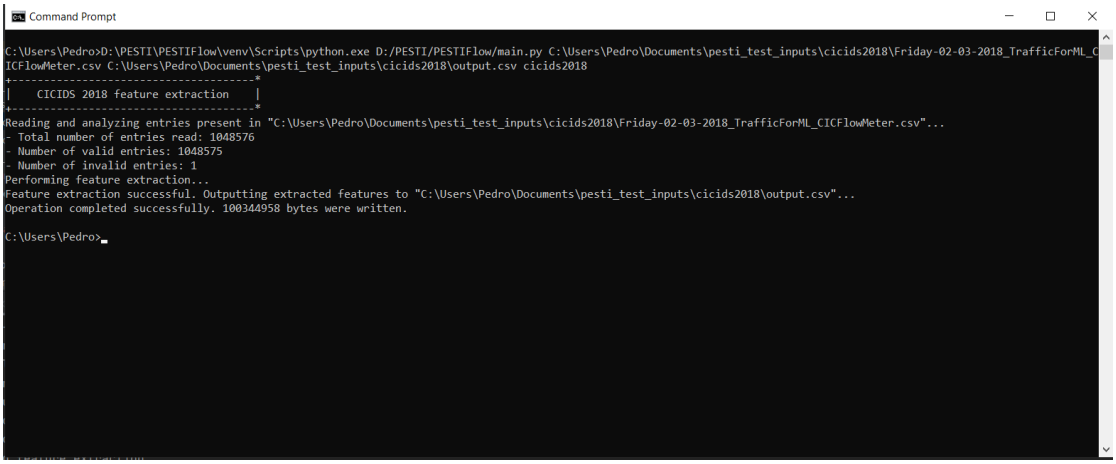


Figure 37 - Application's UI when given CICIDS 2018 file as input (UC4)

5.7 Solution evaluation

In this section, the evaluation of the application is performed and is divided into two parts: first, all functional and non-functional requirements are individually assessed and the appropriate verification measures are taken to determine if the requirement was fulfilled, partially fulfilled or not fulfilled. This evaluation is summarized in Table 17 .

Secondly, the application's level of security is determined using a Static Application Security Testing (SAST) tool which analysis the application's source code and identifies known vulnerabilities. In the end, a report is generated which includes all the findings.

From table Table 17, one concludes that all functional and non-functional requirements were fulfilled.

Table 17 - Application evaluation summary

		Requirement	Evaluation
FUNCTIONAL		UC1 - Obtain features from a PCAP file	Fulfilled (confirmed by the integration test summarized in Table 16.)
		UC2 - Obtain features from Airbus' "Retour de Flamme" project log file	Fulfilled (confirmed by the integration test summarized in Table 16.)
		UC3 - Obtain features from an Airbus-provided Graylog log file	Fulfilled (confirmed by the integration test summarized in Table 16.)
		UC4 – Obtain features from a CICIDS 2018 dataset file	Fulfilled (confirmed by the integration test summarized in Table 16.)
NON-FUNCTIONAL	**F**	"Help" functionalities must be available to the user.	Fulfilled (see Figure 20)
		The user must be informed of the progress of the application's execution.	Fulfilled (see any of the integration tests presented in Section 5.6.3)
		Depending on the nature of the input field, both unidirectional and bidirectional output must be supported.	Fulfilled
		Mechanism for labelling the received network traffic data as "malicious traffic" or "normal traffic" must be supported.	Fulfilled
		The application must be secure and thus appropriate measures must be taken to ensure that no vulnerabilities are present.	Fulfilled (see the next subsection for detailed security overview)
	U	The application must feature a CLI.	Fulfilled (see Section 5.5.1)
	R	-	-
	P	-	-
	S	The application must run in both Windows OS and Linux OS.	Fulfilled (implemented in Python)

		The application must support the addition of new input file formats.	Fulfilled
		The application must support the addition of new output features.	Fulfilled
	+	The application must reflect good design practices, including but not limited to, GRASP patterns and SOLID guidelines.	Fulfilled
		The application must account for large input files and thus the application is required to have a design, mindful of the time and storage complexity issues.	Fulfilled

The application's security requirement was assessed using Bandit [122], an open source tool designed to find common security issues in Python code. To achieve this, Bandit will analyze each file, build an AST from the respective file, and run appropriate plugins against the created AST nodes. Once Bandit has finished its operation, it generates a report. For more information on the type of vulnerabilities the tool attempts to detect, refer to [122]. The command executed was

```
bandit . -r -x ./venv -o ./security-analysis.txt -f txt
```

and the output obtained is represented in Figure 38.

```
1  Run started:2020-09-05 21:21:32.767431
2
3  Test results:
4      No issues identified.
5
6  Code scanned:
7      Total lines of code: 1732
8      Total lines skipped (#nosec): 0
9
10 Run metrics:
11     Total issues (by severity):
12         Undefined: 0.0
13         Low: 0.0
14         Medium: 0.0
15         High: 0.0
16     Total issues (by confidence):
17         Undefined: 0.0
18         Low: 0.0
19         Medium: 0.0
20         High: 0.0
21 Files skipped (0):
```

Figure 38 - Application's security analysis

As it can form the reported from the Bandit tool, the application is secure and does not contain any type of vulnerabilities.

6 Conclusions

The solution developed stands as a very lightweight CLI-based application for the consolidation of different formatted files, containing different network traffic information. The application was written in Python, which is often referred in the community as a "glue language", allowing the interoperation of code of disparate languages. This means that the solution developed can be easily fitted to work within a pipeline totally defined in C or any other language. A solution built with Python also means that it will be able to run in any Python-supported environments, further contributing to its versatility.

With a design centered around the supportability of multiple file formats and features, the solution developed is able to contribute greatly to the cybersecurity field, specially the intrusion detection community which will benefit from a centralized solution capable of consolidating different formats into a single, well-defined one.

6.1 Achieved goals

Throughout the project, the objectives initially defined, as described in Section 1.2.1 did not suffer anu changes. One of the defined objectives (the last one presented in both Section 1.2.1 and Table 18), however, ended up being discarded due to time constraints and the wish of the student to focus more on the software development side of the project, rather than the application of artificial intelligence.

Table 18 revisits the initially defined objectives for the project, as they are defined in Section 1.2.1, and demonstrates their level of completeness, in the form of a percentage.

Table 18 - Objective rate of completion

Goal	Completion rate
Understand the different types of network flow data and the different formats in which each type of network data may exist in and how they relate to each other.	100%
Understand what is an IDS, what kinds of IDS exist, and what types of attacks are anomaly-based IDS capable of detecting.	100%

Study what kind of features are used in publicly available datasets and identify the top 20 most used features.	100%
Develop a solution that can receive, as input, files of multiple formats and produces, as output, a training set, characterized by a number of pre-defined features, with the same format, regardless of the file received as input. The solution must also be flexible enough to support other formats or features in the future, without requiring major changes to the code base.	100%
Employ feature selection in order to reduce the number of features present in the produced training set, while still maintaining data integrity.	0% (discarded)

6.2 Constraints and future work

One of the biggest constraints identified during the development of the solution was a lack of documentation of the files provided by Airbus. For the files related to the “Retour de Flamme” scenario, no documentation about their format was given. Only one file was provided, and very little context was given regarding how the provided files were captured.

Files of the Graylog category were in a similar situation. Besides the fact that only one file of this type was provided, it was only known that it was a file originating from Graylog because it was present in the filename. No more information was given regarding its format.

Another constraint imposed to the development of the application was the COVID-19 pandemic. This pandemic, which led to the shutdown of the installations of ISEP where the PESTI was taking place, forced the student to have to work. While this in itself was not cause for problem, the mandatory confinement took a toll on his mental health and, as a consequence, his performance and well-being suffered. Nonetheless, the student was able to recuperate from that and carry on with the development of the project.

As for future work, several things might be done in the future which can not only improve the application's performance but also its functionality.

Feature normalization "avoids features with large numerical values from dominating other features" [12]. Feature selection and feature extraction have also proved, in the past, to be able to improve the quality of the models trained on data which was subject to dimensionality reduction tasks [12] [45, p. 2] . This results in higher accuracy when classifying traffic as normal or attack.

In terms of performance, the application can benefit from feature normalization, which "avoids features with large numerical values from dominating other features" [12]. Feature selection and feature extraction have also proved, in the past, to be able to improve the quality of the models trained on data which was subject to dimensionality reduction tasks [12] [45, p. 2] . This results in higher accuracy when classifying traffic as normal or attack. Moreover, different Python implementations will bring different benefits. As presented in Section 3.2.1, Python is made up of an interface and an implementation. CPython, Python's reference implementation, was used to develop and test the application. However, PyPy, another implementation, is able to, on average, speeds up Python code interpretation by about 7.6 times when compared to CPython, with some tasks accelerated 50 times or more [123].

In terms of functionality, future work might include adding support for more file formats and adding more features. Not only add more features but consider new type of features. David & Clark [12] identified several multiple connection derived features which instead of dealing with flows (packet aggregations), they operate at a higher level, over aggregates of flows, giving a different insight into behaviors on the network. These types of features have been used in two of the datasets presented in Section 3.1.2 (UNSW-NB15 and TUIDS) and the paper previously mentioned indicates it as a promising vector of data for the construction of ML models for intrusion detection.

6.3 Final appreciation

This project presented a very rich learning experience, requiring the application of knowledge from multiple areas including networking, software development, cybersecurity and machine learning. Having taken place in a research environment, it not only gave the student the opportunity to deepen its knowledge in a specific area, it provided him with invaluable experience on how to better search and locate good and reliable sources of knowledge.

Throughout the project, the student was given the opportunity to apply all knowledge learnt in the bachelors towards a solution which will have a real and positive impact in the world. The underlying problem was challenging and allowed the student to not only gain more experience with already known technologies as well as learn new ones, including a new programming language.

Participation in this project also greatly improved the student's soft skills, from not only interacting with other researchers but also with new colleagues developing similar projects.

References

[1] J. Schreier, “Sony Estimates $171 Million Loss From PSN Hack,” *Wired*, May 23, 2011.

[2] J. Porter, “EasyJet hack hits 9 million customers,” *The Verge*, May 19, 2020.

[3] RiskBased Security, Inc., “2019 MidYear QuickView Data Breach Report,” Aug. 2019.

[4] MarketsandMarkets Research, “Intrusion Detection and Prevention Systems Market by Component (Solutions and Services), Type, Deployment Type (Cloud and On-Premises), Organization Size (SMEs and Large Enterprises), Vertical, and Region - Global Forecast to 2025,” *Intrusion Detection and Prevention Systems Market*. https://www.marketsandmarkets.com/Market-Reports/intrusion-detection-prevention-system-market-199381457.html.

[5] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, “From Data Mining to Knowledge Discovery in Databases,” *AI Magazine*, vol. 17, no. 3, p. 37, Mar. 1996, doi: 10.1609/aimag.v17i3.1230.

[6] “We are Airbus,” *Airbus*. https://www.airbus.com/company/we-are-airbus.html (accessed Sep. 05, 2020).

[7] “Gecad - Home.” http://www.gecad.isep.ipp.pt/GECAD/Pages/Presentation/Home.aspx (accessed Sep. 04, 2020).

[8] “Related Projects | TCPDUMP/LIBPCAP public repository.” https://www.tcpdump.org/related.html (accessed Sep. 04, 2020).

[9] B. Li, J. Springer, G. Bebis, and M. Hadi Gunes, “A survey of network flow applications,” *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 567–581, Mar. 2013, doi: 10.1016/j.jnca.2012.12.020.

[10] “Suricata,” *Suricata*. https://suricata-ids.org/.

[11] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, “A survey of network-based intrusion detection data sets,” *Computers & Security*, vol. 86, pp. 147–167, Sep. 2019, doi: 10.1016/j.cose.2019.06.005.

[12] J. J. Davis and A. J. Clark, “Data preprocessing for anomaly based network intrusion detection: A review,” *Computers & Security*, vol. 30, no. 6, pp. 353–375, Sep. 2011, doi: 10.1016/j.cose.2011.05.008.

[13] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, 1st ed. Boston: Pearson, 2005.

[14] L. Sharma, “WaterFall Model in Software Developement Life Cycle | SDLC,” *TOOLSQA*, Apr. 17, 2016. https://www.toolsqa.com/software-testing/waterfall-model/.

[15] Lucidchart Content Team, “Agile vs. Waterfall vs. Kanban vs. Scrum,” Oct. 23, 2017. https://www.lucidchart.com/blog/agile-vs-waterfall-vs-kanban-vs-scrum.

[16] Glen Turner, “application/vnd.tcpdump.pcap MIME media type,” Mar. 30, 2011. https://www.iana.org/assignments/media-types/application/vnd.tcpdump.pcap.

[17] “TCPDUMP/LIBPCAP public repository.” https://www.tcpdump.org/ (accessed Sep. 05, 2020).

[18] “Npcap: Windows Packet Capture Library & Driver.” https://nmap.org/npcap/ (accessed Sep. 05, 2020).

[19] The WinPcap Team, “WinPcap: WinPcap internals.” https://www.winpcap.org/docs/docs_412/html/group__internals.html (accessed Sep. 05, 2020).

[20] “Manpage of PCAP-SAVEFILE.” https://www.tcpdump.org/manpages/pcap-savefile.5.html.

[21] “Development/LibpcapFileFormat - The Wireshark Wiki.” https://wiki.wireshark.org/Development/LibpcapFileFormat.

[22] The Tcpdump Group, “LINK-LAYER HEADER TYPES,” *Tcpdump and libpcap*. https://www.tcpdump.org.

[23] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, “An Overview of IP Flow-Based Intrusion Detection,” *IEEE Communications Surveys Tutorials*, vol. 12, no. 3, pp. 343–356, Third 2010, doi: 10.1109/SURV.2010.032210.00054.

[24] EC-Council, *Ethical Hacking and Countermeasures - Version 10*. EC-Council, 2018.

[25] imperva, “What is a DDoS Botnet | Common Botnets and Botnet Tools | Imperva.” https://www.imperva.com/learn/application-security/botnet-ddos/.

[26] imperva, “What does DDoS Mean? | Distributed Denial of Service Explained | Imperva.” https://www.imperva.com/learn/application-security/denial-of-service/.

[27] imperva, “What is a Brute Force | Common Tools & Attack Prevention | Imperva.” https://www.imperva.com/learn/application-security/brute-force-attack/.

[28] Rob Picheta, “The most commonly hacked passwords, revealed,” *CNN*, Apr. 23, 2019. https://www.cnn.com/2019/04/22/uk/most-common-passwords-scli-gbr-intl/index.html.

[29] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, “Survey of intrusion detection systems: techniques, datasets and challenges,” *Cybersecur*, vol. 2, no. 1, p. 20, Jul. 2019, doi: 10.1186/s42400-019-0038-7.

[30] N. Moustafa, J. Hu, and J. Slay, “A holistic review of Network Anomaly Detection Systems: A comprehensive survey,” *Journal of Network and Computer Applications*, vol. 128, pp. 33–55, Feb. 2019, doi: 10.1016/j.jnca.2018.12.006.

[31] M. Meyers, *CompTIA Network+ Certification All-in-One Exam Guide, 5th Edition*, 5th Edition. New York: McGraw-Hill Education, 2012.

[32] M. Farooq-i-Azam, “A Comparison of Existing Ethernet Frame Specifications,” *arXiv:1610.00635 [cs]*, Oct. 2016, [Online]. Available: http://arxiv.org/abs/1610.00635.

[33] IEEE Computer Society, *802.3-2018 - IEEE Standard for Ethernet*. New York, USA: IEEE, 2018.

[34] IANA, “IEEE 802 Numbers,” Dec. 23, 2019. https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[35] “Internet Protocol,” RFC Editor, RFC 791, Sep. 1981. [Online]. Available: https://tools.ietf.org/html/rfc791.

[36] IANA, “Internet Protocol Version 4 (IPv4) Parameters,” May 03, 2018. https://www.iana.org/assignments/ip-parameters/ip-parameters.xhtml.

[37] “Transmission Control Protocol,” RFC Editor, RFC 793, Sep. 1981. [Online]. Available: https://tools.ietf.org/html/rfc793.

[38] IANA, “Transmission Control Protocol (TCP) Parameters,” Apr. 03, 2020. https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml.

[39] “User Datagram Protocol,” RFC Editor, RFC 768, Aug. 1980. [Online]. Available: https://tools.ietf.org/html/rfc768.

[40] B. Claise, “Cisco Systems NetFlow Services Export Version 9.” https://tools.ietf.org/html/rfc3954.

[41] R. Hofstede *et al.*, “Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX,” *IEEE Communications Surveys Tutorials*, vol. 16, no. 4, pp. 2037–2064, Fourthquarter 2014, doi: 10.1109/COMST.2014.2321898.

[42] CISCO, “Introduction to Cisco IOS NetFlow - A Technical Overview,” May 29, 2012. https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html.

[43] European Union Agency For Cybersecurity, *Introduction to Network Forensics*. 2019.

[44] “Firebox NetFlow and PRTG Integration Guide.” https://www.watchguard.com/help/docs/help-center/en-US/Content/Integration-Guides/Netflow/NetFlow_PRTG_Integration_Guide.html (accessed Aug. 10, 2020).

[45] R. Damasevicius *et al.*, “LITNET-2020: An Annotated Real-World Network Flow Dataset for Network Intrusion Detection,” *Electronics*, vol. 9, no. 5, Art. no. 5, May 2020, doi: 10.3390/electronics9050800.

[46] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, “Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy,” in *2019 International Carnahan Conference on Security Technology (ICCST)*, Oct. 2019, pp. 1–8, doi: 10.1109/CCST.2019.8888419.

[47] M. Stevanovic and J. M. Pedersen, “An analysis of network traffic classification for botnet detection,” in *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, Jun. 2015, pp. 1–8, doi: 10.1109/CyberSA.2015.7361120.

[48] F. Beer and U. Bühler, “Feature selection for flow-based intrusion detection using Rough Set Theory,” in *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*, May 2017, pp. 617–624, doi: 10.1109/ICNSC.2017.8000162.

[49] M. M. Najafabadi, T. M. Khoshgoftaar, C. Kemp, N. Seliya, and R. Zuech, “Machine Learning for Detecting Brute Force Attacks at the Network Level,” in *2014 IEEE International Conference on Bioinformatics and Bioengineering*, Nov. 2014, pp. 379–385, doi: 10.1109/BIBE.2014.73.

[50] E. Biglar Beigi, H. Hadian Jazi, N. Stakhanova, and A. A. Ghorbani, “Towards effective feature selection in machine learning-based botnet detection approaches,” in *2014 IEEE Conference on Communications and Network Security*, Oct. 2014, pp. 247–255, doi: 10.1109/CNS.2014.6997492.

[51] A. Thakkar and R. Lohiya, “A Review of the Advancement in Intrusion Detection Datasets,” *Procedia Computer Science*, vol. 167, pp. 636–645, Jan. 2020, doi: 10.1016/j.procs.2020.03.330.

[52] I. Sharafaldin, A. Habibi Lashkari, and A. Ghorbani, “Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization,” Jan. 2018, pp. 108–116, doi: 10.5220/0006639801080116.

[53] University of New Brunswick - Canadian Institute for Cybersecurity, “Intrusion Detection Evaluation Dataset (CICIDS2017),” *IDS 2017 | Datasets | Research | Canadian Institute for Cybersecurity | UNB*. https://www.unb.ca/cic/datasets/ids-2017.html.

[54] University of New Brunswick - Canadian Institute for Cybersecurity, “CSE-CIC-IDS2018 on AWS,” *IDS 2018 | Datasets | Research | Canadian Institute for Cybersecurity | UNB*. https://www.unb.ca/cic/datasets/ids-2018.html.

[55] M. Ring, S. Wunderlich, D. Grüdl, D. Landes, and A. Hotho, “Flow-based benchmark data sets for intrusion detection,” in *Proceedings of the 16th European Conference on Cyber Warfare and Security (ECCWS)*, 2017, pp. 361–369.

[56] Markus Ring, Sarah Wunderlich, and Dominik Grüdl, “Technical Report CIDDS-001 data set,” Apr. 2017.

[57] M. Ring, S. Wunderlich, D. Grüdl, D. Landes, and A. Hotho, “Creation of Flow-Based Data Sets for Intrusion Detection,” *Journal of Information Warfare*, vol. 16, no. 4, pp. 40–53, 2017.

[58] Markus Ring and Sarah Wunderlich, “Technical Report CIDDS-002 data set,” Oct. 2017.

[59] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, “Toward developing a systematic approach to generate benchmark datasets for intrusion detection,” *Computers & Security*, vol. 31, no. 3, pp. 357–374, May 2012, doi: 10.1016/j.cose.2011.12.012.

[60] P. Gogoi, M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, “Packet and Flow Based Network Intrusion Dataset,” in *Contemporary Computing*, Berlin, Heidelberg, 2012, pp. 322–334, doi: 10.1007/978-3-642-32129-0_34.

[61] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, “Towards Generating Real-life Datasets for Network Intrusion Detection,” *I. J. Network Security*, vol. 17, pp. 683–701, 2015.

[62] A. Sperotto, R. Sadre, F. van Vliet, and A. Pras, “A Labeled Data Set for Flow-Based Intrusion Detection,” in *IP Operations and Management*, Berlin, Heidelberg, 2009, pp. 39–50, doi: 10.1007/978-3-642-04968-2_4.

[63] S. García, M. Grill, J. Stiborek, and A. Zunino, “An empirical comparison of botnet detection methods,” *Computers & Security*, vol. 45, pp. 100–123, Sep. 2014, doi: 10.1016/j.cose.2014.05.011.

[64] G. Maciá-Fernández, J. Camacho, R. Magán-Carrión, P. García-Teodoro, and R. Therón, “UGR‘16: A new dataset for the evaluation of cyclostationarity-based network IDSs,” *Computers & Security*, vol. 73, pp. 411–424, Mar. 2018, doi: 10.1016/j.cose.2017.11.004.

[65] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, “SSH Compromise Detection Using NetFlow/IPFIX,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 20–26, Oct. 2014, doi: 10.1145/2677046.2677050.

[66] A. D. Kent, “Cyber security data sources for dynamic network research,” in *Dynamic Networks and Cyber-Security*, pp. 37–65.

[67] M. J. M. Turcotte, A. D. Kent, and C. Hash, *Unified Host and Network Data Set*. 2017.

[68] N. Moustafa and J. Slay, “UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set),” in *2015 Military Communications and Information Systems Conference (MilCIS)*, Nov. 2015, pp. 1–6, doi: 10.1109/MilCIS.2015.7348942.

[69] Nour Moustafa, “The UNSW-NB15 Dataset Description,” *The UNSW-NB15 data set description*, Nov. 14, 2018. https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-NB15-Datasets/.

[70] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani, “Characterization of Encrypted and VPN Traffic using Time-related Features,” in *Proceedings of the 2nd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,* 2016, pp. 407–414, doi: 10.5220/0005740704070414.

[71] C. Inacio and B. Trammell, “YAF: Yet Another Flowmeter.” Aug. 2010, [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=57044.

[72] CERT NetSA group, “YAF - Documentation.” https://tools.netsa.cert.org/yaf/yaf.html (accessed Aug. 10, 2020).

[73] CERT NetSA group, “YAF - Indexing PCAP Files with YAF.” https://tools.netsa.cert.org/yaf/libyaf/yaf_pcap.html (accessed Aug. 10, 2020).

[74] ntop team, “nProbe,” Aug. 04, 2011. https://www.ntop.org/products/netflow/nprobe/ (accessed Aug. 10, 2020).

[75] “nProbe Modes — nProbe 9.1 documentation.” https://www.ntop.org/guides/nprobe/configurations.html (accessed Aug. 10, 2020).

[76] P. Haag, *phaag/nfdump*. 2020.

[77] “General Python FAQ — Python 3.8.5 documentation.” https://docs.python.org/3/faq/general.html#what-is-python (accessed Aug. 11, 2020).

[78] Tim Peters, “PEP 20 -- The Zen of Python,” *Python.org*, Aug. 19, 2004. https://www.python.org/dev/peps/pep-0020/ (accessed Aug. 12, 2020).

[79] PyPi, “PyPI · The Python Package Index,” *PyPI*. https://pypi.org/ (accessed Aug. 14, 2020).

[80] “Is Python interpreted, or compiled, or both?,” *Net-informations.com*. http://net-informations.com/python/iq/interpreted.htm (accessed Aug. 14, 2020).

[81] The Python Software Foundation, “Glossary - Python 3.8.5 documentation (duck-typing).” https://docs.python.org/3/glossary.html#term-duck-typing (accessed Aug. 14, 2020).

[82] Microsoft, “Use early binding and late binding in Automation - Office,” Jan. 24, 2020. https://docs.microsoft.com/en-us/previous-versions/office/troubleshoot/office-developer/binding-type-available-to-automation-clients (accessed Aug. 12, 2020).

[83] Aaron Krauss, “Programming Concepts: Type Introspection and Reflection | Aaron Krauss,” Feb. 12, 2016. https://thecodeboss.dev/2016/02/programming-concepts-type-introspection-and-reflection/ (accessed Aug. 14, 2020).

[84] Youssef Nader, “Python vs Java in 2020: Comparison, Features & Applications,” *Hackr.io*, Aug. 06, 2020. https://hackr.io/blog/python-vs-java (accessed Aug. 14, 2020).

[85] Scott Langham, “c# - Early and late binding (Top answer),” *Stack Overflow*, Jan. 27, 2009. https://stackoverflow.com/questions/484214/early-and-late-binding (accessed Aug. 14, 2020).

[86] “Difference between Early and Late Binding in Java,” *Techie Delight*, Aug. 11, 2017. https://www.techiedelight.com/difference-between-early-late-binding-java/ (accessed Aug. 12, 2020).

[87] GitHub, “The State of the Octoverse,” *The State of the Octoverse*. https://octoverse.github.com/ (accessed Aug. 12, 2020).

[88] TIOBE, “index | TIOBE - The Software Quality Company.” https://www.tiobe.com/tiobe-index/ (accessed Aug. 12, 2020).

[89] Eric Goebelbecker, “Python vs Java: Which is best? Code examples and comparison for 2019,” *Raygun Blog*, Dec. 13, 2018. https://raygun.com/blog/java-vs-python/ (accessed Aug. 15, 2020).

[90] T. Radcliffe, “Python vs. Java: Duck Typing, Parsing on Whitespace and Other Cool Differences,” *ActiveState*, Jan. 26, 2016. https://www.activestate.com/blog/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences/ (accessed Aug. 15, 2020).

[91] “Python 3 vs Java - Which programs are fastest? | Computer Language Benchmarks Game.” https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-java.html (accessed Aug. 15, 2020).

[92] The Python Software Foundation, “argparse — Parser for command-line options, arguments and sub-commands — Python 3.8.5 documentation,” *argparse — Parser for command-line options, arguments and sub-command*. https://docs.python.org/3/library/argparse.html (accessed Aug. 16, 2020).

[93] “Why Click? — Click Documentation (7.x).” https://click.palletsprojects.com/en/7.x/why/ (accessed Aug. 16, 2020).

[94] “Welcome to Click — Click Documentation (7.x),” *Click*. https://click.palletsprojects.com/en/7.x/#documentation (accessed Aug. 16, 2020).

[95] “docopt—language for description of command-line interfaces.” http://docopt.org/ (accessed Aug. 16, 2020).

[96] “Introduction — Scapy 2.4.4 documentation.” https://scapy.readthedocs.io/en/latest/introduction.html (accessed Aug. 18, 2020).

[97] “dpkt — dpkt 1.9.2 documentation.” https://dpkt.readthedocs.io/en/latest/ (accessed Aug. 18, 2020).

[98] J. Shaw, *JarryShaw/PyPCAPKit*. 2020.

[99] D. Green, *KimiNewt/pyshark*. 2020.

[100] “Projects · Michael Stahn / pypacker,” *GitLab*. https://gitlab.com/mike01/pypacker (accessed Aug. 19, 2020).

[101] *secdev/scapy*. SecDev, 2020.

[102] K. Bandla, *kbandla/dpkt*. 2020.

[103] STACKIFY, “What Are OOP Concepts in Java? 4 Primary Concepts,” *Stackify*, Apr. 05, 2017. https://stackify.com/oops-concepts-in-java/ (accessed Aug. 21, 2020).

[104] J. Paul, “What is Inheritance in Java with example - Object Oriented Programming Tutorial.” https://www.java67.com/2012/08/what-is-inheritance-in-java-oops-programming-example.html (accessed Aug. 21, 2020).

[105] Wm. P. Rogers, “Encapsulation is not information hiding,” *InfoWorld*, May 18, 2001. https://www.infoworld.com/article/2075271/encapsulation-is-not-information-hiding.html (accessed Aug. 21, 2020).

[106] THORBEN JANSSEN, “OOP Concepts for Beginners: What is Polymorphism,” *Stackify*, Dec. 22, 2017. https://stackify.com/oop-concept-polymorphism/ (accessed Aug. 21, 2020).

[107] J. Malenfant, M. Jacques, and F. Demers, “A Tutorial on Behavioral Reflection and its Implementation,” 1996.

[108] Hyunyoung Lee, “CSCE 314 Programming Languages Reflection,” [Online]. Available: https://people.engr.tamu.edu/hlee42/csce314/lec16-java-reflection.pdf.

[109] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language: Everything You Need to Know*, 1 edition. Beijing ; Sebastopol, CA: O’Reilly Media, 2008.

[110] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, 1 edition. Hoboken, NJ: Wiley, 2009.

[111] Steve Smith, “Overview of ASP.NET Core MVC,” Feb. 12, 2020. https://docs.microsoft.com/en-us/aspnet/core/mvc/overview (accessed Aug. 21, 2020).

[112] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition. Reading, Mass: Addison-Wesley Professional, 1994.

[113] Refactoring Guru, “Template Method,” *Template Method*. https://refactoring.guru/design-patterns/template-method (accessed Aug. 21, 2020).

[114] Refactoring Guru, “Creational Design Patterns.” https://refactoring.guru/design-patterns/creational-patterns (accessed Aug. 21, 2020).

[115] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3 edition. Upper Saddle River, N.J: Prentice Hall, 2004.

[116] Quinn Radich and Eliot Cowley, “What is a machine learning model?” https://docs.microsoft.com/en-us/windows/ai/windows-ml/what-is-a-machine-learning-model (accessed Aug. 23, 2020).

[117] “Network Protocol Definition | Computer Protocol | Computer Networks | CompTIA.” https://www.comptia.org/content/guides/what-is-a-network-protocol (accessed Aug. 23, 2020).

[118] “Best Git Client - Features | GitKraken Git GUI,” *GitKraken.com*. https://www.gitkraken.com/git-client (accessed Aug. 27, 2020).

[119] Kadam Patel, “TCP Connection Termination,” *GeeksforGeeks*, Dec. 08, 2019. https://www.geeksforgeeks.org/tcp-connection-termination/ (accessed Aug. 29, 2020).

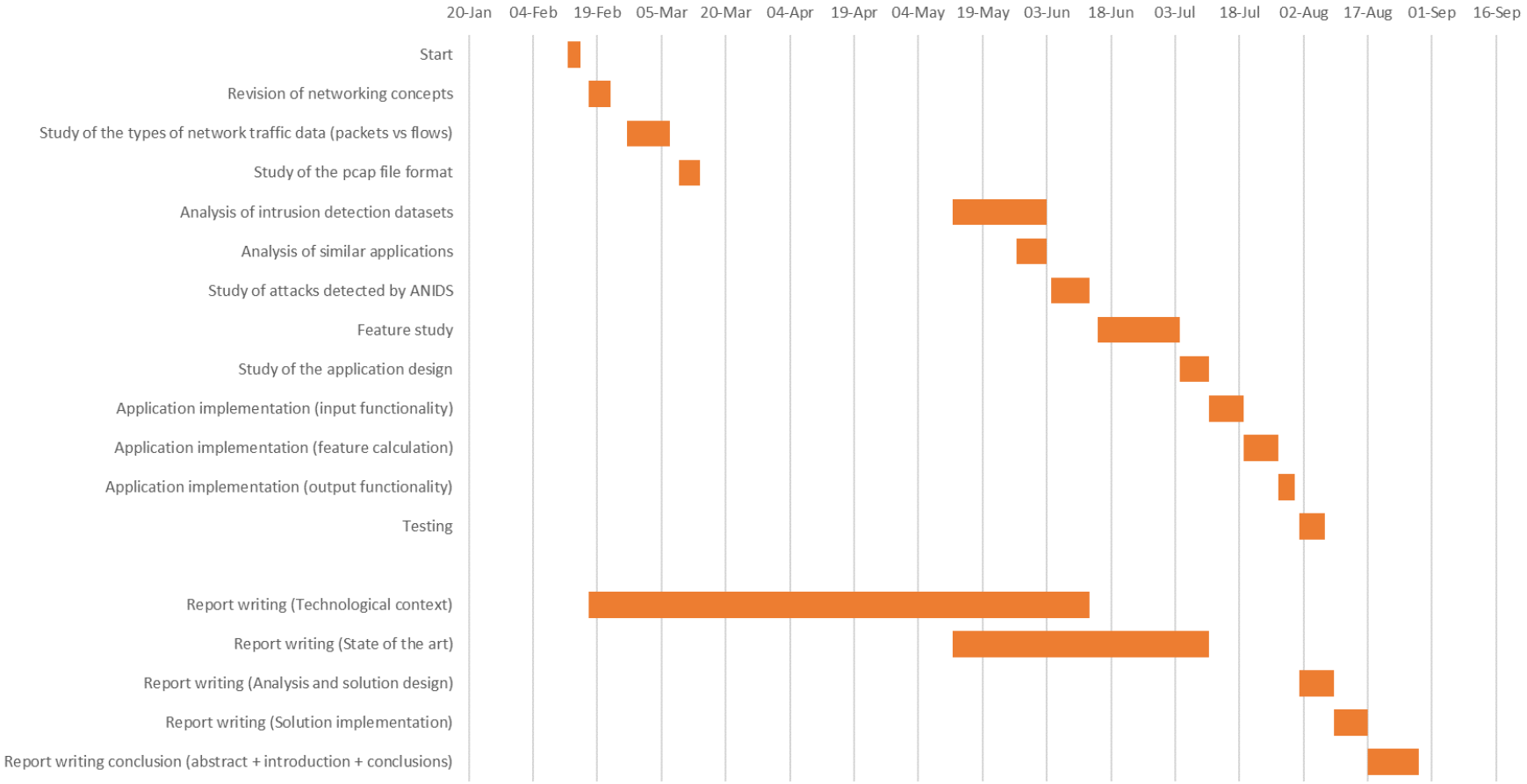
[120] STEN PITTET, “The different types of testing in Software,” *Atlassian*. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing (accessed Aug. 31, 2020).

[121] “Types of Software Testing: Different Testing Types with Details,” Aug. 01, 2020. https://www.softwaretestinghelp.com/types-of-software-testing/ (accessed Aug. 31, 2020).

[122] *PyCQA/bandit*. Python Code Quality Authority, 2020.

[123] S. Yegulalp, “What is PyPy? Faster Python without pain,” *InfoWorld*, May 01, 2019. https://www.infoworld.com/article/3385127/what-is-pypy-faster-python-without-pain.html (accessed Aug. 31, 2020).

Annex A Work planning



Annex B Features from other papers

In this annex are presented the features found in the different papers addressed in the State of the art section. The features are presented in a table form, accompanied by the name given to the table by the original authors.

B.1 Features from [12]

Protocol fields	Fields used as basic features
Ethernet headers	Size, dest hi, dest lo, src hi, src lo, protocol
IP headers	Header length, TOS, Frag ID, Frag Ptr, TTL, Protocol, Checksum, Src ip, Dest ip
TCP headers	Src Port, Dest Port, Seq, Ack, Header Len, Flag UAPRSF, Window Sz, Checksum, URG Ptr, Option
UDP headers	Src Port, Dest Port, Len, Checksum
ICMP headers	Type, Code Checksum

"Table A1 - Summary of basic features"

Feature	Description
Contextual	Quad: combination of src ip, src port, dst ip, dst port define a single connection Service type (TCP, UDP or ICMP) and application protocol (HTTP, SMTP, SSH or FTP etc.) to group similar traffic
Duration	Start time, end time, and the duration of the connection
Status	Normal or error status of connection e.g. valid TCP 3-way handshake, and FIN to end session
SCD timing	Number of questions per second
	Average size of questions
	Average size of answers

	Question answer idle time
	Answer question idle time
RTT	The round trip time (RTT) of a TCP packet sent through a connection chain is calculated from timestamps of TCP send and echo packets
Fingerprint	Percentage of packets with each of the TCP flags set
	Mean packet inter-arrival time
	Mean packet length

"Table A2 - Summary of SCD features."

Feature	**Description**
Entropy measures	Entropy of basic features over dataset: src ip, src port, dst ip, dst port
Association rules	Mine rules from connection records containing: start time, quad, and connection status
Flow concentration	Count of TCP flows with same src ip, dst ip and dst port in this time slice
Data points per cluster	A cluster is a frequently occurring value for a feature, e.g. a common IP address
%Control, %Data	Percentage of control/data packets
Av duration	Average flow duration over all flows
Av duration dest	Average flow duration per destination
Max flows	Maximum number of flows to a particular service
%_same_service_host	Percent of traffic from a particular src port to a particular dst ip
%_same_host_service	Percent of traffic from a particular src ip to a particular dst port

Count variance	Variance measure for the count of packets for each src-dest pair
Wrong resent rate	Count of bytes sent even after being acknowledged
Duplicate ACK rate	Count of duplicate acknowledgment packets
Data bytes	Count of data bytes exchanged per flow
sdp statistics	Source-destination pairs (sdps) are unique combinations of src ip, dest ip and dest port
	Number of unique sdps in collection interval
	Number of new sdps in this data collection interval
	Number of new sdps which were not seen in last month
	Number of well known ports used in interval
	Variance of the count of packets seen against sdps
	Count of sdps which include hosts outside local network domain
	Number of successfully established TCP connections in time interval

"Table A4 - Summary of miscellaneous MCD features." (adapted)

Volume-based feature	**Description**
count-dest	Flow count to unique dest IP in the last T seconds from the same src
count-src	Flow count from unique src IP in the last T seconds to the same dest
count-serv-src	Flow count from the src IP to the same dest port in the last T seconds
count-serv-dest	Flow count to the dest IP using same src port in the last T seconds

count-dest-conn	Flow count to unique dest IP in the last N flows from the same src IP
count-src-conn	Flow count from unique src IP in the last N flows to the same dest IP
count-serv-src-conn	Flow count from the src IP to the same dest port in the last N flows
count-serv-dest-conn	Flow count to the dest IP using same source port in the last N flows
num_packets_src_dst/dst_src	Count of packets flowing in each direction
num_acks_src_dst/dst_src	Count of acknowledgment packets flowing in each direction
num_bytes_src_dst/dst_src	Count of data bytes flowing in each direction
num_retransmit_src_dst/dst_src	Count of retransmitted packets flowing in each direction
num_pushed_src_dst/dst_src	Count of pushed packets flowing in each direction
num_SYNs(FINs)_src_dst/dst_src	Count of SYN/FYN packets flowing in each direction
connection_status	Status of the connection (0 e Completed; 1 - Not completed; 2 e Reset)
count_src'	Connection count from same source as the current record
count_serv_src	Count of different services from the same source as the current record
count_serv_dest	Count of different services to the same destination IP as the current record
count_src_conn	Connection count from this src IP in the last 100 connections
count_dest_conn	Connection count to this dest IP in the last 100 connections

count_serv_src_conn	Connection count with same dst port and src IP in the last 100 connections
count_serv_dst_conn	Connection count with same dst port and dst IP in the last 100 connections

"Table A5 - Summary of volume-based MCD features similar to KDD99"

B.2 Features from [47]

Feature	Type	Number[4]
Basic conversation features		
Port number	Numerical	2
Layer 7 protocol	Categorical	1
Duration (last pkt - first pkt)	Numerical	1
Total number of packets	Numerical	2
Total number of Bytes	Numerical	2
Mean of the number of Bytes per packet	Numerical	2
Std of the number of Bytes per packet	Numerical	2
Time-based features		
Number of packets per second	Numerical	2
Number of Bytes per second	Numerical	2
Mean of packets inter-arrival time	Numerical	2
Std of packets inter-arrival time	Numerical	2
Bidirectional features		
Ratio of number of packets OUT/IN	Numerical	1
Ratio of number of Bytes OUT/IN	Numerical	1
Ratio of inter-arrival times OUT/IN	Numerical	1
TCP specific features		
Number of three way handshakes	Numerical	1
Number of connection tear downs	Numerical	1
Number of complete conversation	Numerical	1
Average conversation duration	Numerical	1

[4] Some features are calculated for both directions of the conversation while others are unique for the conversation.

Percentage of TCP SYN packets	Numerical	2
Percentage of TCP SYN ACK packets	Numerical	2
Percentage of TCP ACK packets	Numerical	2
Percentage of TCP ACK PUSH packets	Numerical	2

"TABLE I: TCP/UDP traffic analysis: the list of features extracted for TCP and UDP conversations."

Feature	Type
FQDN-based features	
Number of tokens	Numerical
Avg length of token	Numerical
Length of SLD (Second Level Domain)	Numerical
Number of dictionary words in SLD	Numerical
Number of numerical characters in SLD	Numerical
Ratio between number of numerical characters and alphabetical characters in SLD	Numerical
Query-based features	
Type of query	Categorical
Number of DNS servers contacted	Numerical
Number of queries	Numerical
Mean of query length	Numerical
Std of query length	Numerical
Mean of queries inter-arrival time	Numerical
Std of queries inter-arrival time	Numerical
Response-based features	
Number of query responses	Numerical

Mean of query response length	Numerical
Std of query response length	Numerical
Mean of query responses inter-arrival time	Numerical
Std of query response inter-arrival time	Numerical
Number of NOERROR responses	Numerical
Number of NXDOMAIN responses	Numerical
Avg number of answers	Numerical
Avg number of authority answers	Numerical
Avg number of additional answers	Numerical
Avg number of resolved IPs	Numerical
Mean of the value of TTL (Time To Live) field	Numerical
Std of the value of TTL field	Numerical
Geographical location features	
Number of countries resolved IPs belong to	Numerical
Number of ASs resolved IPs belong to	Numerical

"TABLE II: DNS traffic analysis: the list of features extracted for every queried FQDN"

B.3 Features from [48]

ID	Description
1	Duration of flow in seconds
2	Unidirectional or bidirectional flow
3	1 if connection is from/to the same host/port, 0 otherwise
4	Type of protocol (ARP, ICMP, TCP, UDP)
5	Service of the flows based on destination port (e.g HTTP, FTP); NA if not extractable
6	Number of packets sent from initiator to responder
7	Number of packets sent from responder to initiator
8	Bytes sent by initiator
9	Bytes sent by responder
10	Distinct TCP flags (FIN, SYN, RST, PUSH, ...) in the initial segment (forward)
11	Distinct TCP flags (FIN, SYN, RST, PUSH, ...) in the initial segment (backward)
12	Distinct TCP flags (FIN, SYN, RST, PUSH, ...) in all subsequent segments (forward)
13	Distinct TCP flags (FIN, SYN, RST, PUSH, ...) in all subsequent segments (backward)
14	Status of the flow (S0, S1, S2, S3, SF, REJ, RSTO, RSTR) based on BRO semantics; NA otherwise
15	Shannon-Fano entropy of the first 20 KBytes of payload (forward)
16	Shannon-Fano entropy of the first 20 KBytes of payload (backward)
17	Service based on DPI (e.g. HTTP, FTP); NA if not extractable
18	Number of all login attempts (SSH, Kerberos, RDP)
19	% of failed login attempts (related to ID 18)
20	Number of login attempts w higher privileges (related to ID 18)
21	Number of all request attempts (web server and network share)

22	% of failed request attempts (related to ID 21)
23	1 if a frequent number of flip flops are identified in the ARP cache, 0 otherwise
24	Number of modified files and events with high severity at the responder
25	% of security related file modifications and suspicious events (related to ID 24)
26	Number of flows to same destination as current flow (window of past 200 adjacent flows)
27	% of flows to same destination and service as current flow (related to ID 26)
28	% number of flows to same destination as current flow w state S0, S1, S2, S3 (related to ID 26)
29	% of flows to same destination as current flow w state REJ, RSTO, RSTR (related to ID 26)
30	Number of flows to same service as current flow (window of past 200 adjacent flows)
31	% of flows to same service and source port as current flow (related to ID 30)
32	% of flows to same service as current flow w state S0, S1, S2, S3 (related to ID 30)
33	% of flows to same service as current flow w state REJ, RSTO, RSTR (related to ID 30)
34	Number of flows to same destination as current flow (window of past 10 sec)
35	% of flows to same destination and service as current flow (related to ID 34)
36	% of flows to same destination as current flow w state S0, S1, S2, S3 (related to ID 34)
37	% of flows to same destination as current flow w state REJ, RSTO, RSTR (related to ID 34)
38	Number of flows to same service as current flow (window of past 10 sec)
39	% of flows to same service and source port as current flow (related to ID 38)
40	% of flows to same service as current flow w state S0, S1, S2, S3 (related to ID 38)

41	% of flows to same service as current flow w state REJ, RSTO, RSTR (related to ID 38)
42	Decision class: 0 (normal) or 1 (attack)

"TABLE 1 - COMPLETE SET OF CONSIDERED FEATURES"

Subset name	**Feature IDs**
Core attributes	5, 8, 10, 26, 27, 29-31, 34, 35, 38
Reduct R1	1, 4, 5, 8-10, 12, 14, 26, 27, 29-31, 34, 35, 38
Reduct R2	1, 4, 5, 8-12, 26, 27, 29-31, 34, 35, 38
Reduct R3	1, 4, 5, 6, 8-10, 12, 26, 27, 29-31, 34, 35, 38
Reduct R4	4, 5, 6, 8-12, 26, 27, 29-31, 34, 35, 38
Reduct R5	1, 5, 6, 8-12, 14, 26, 27, 29-31, 34, 35, 38

"TABLE II - EXPOSED CORE AND REDUCTS USING DS-1"

B.4 Features from [49]

Feature	Description
Source Port	Source port seen in the flow
Destination Port	Destination port seen in the flow
Number of Packets	Total number of packets seen in the flow
Number of Bytes	Total number of bytes seen in the flow
Duration	Flow duration time
Flow Flags	Cumulative OR of all the TCP flags seen in the flow
Initial Flags	The flags of the first packet seen in the flow
Session Flags	Cumulative OR of all the TCP flags seen in the flow except the TCP flags of the first packet

"Table I: Features used for flow analysis"

B.5 Features from [50]

Feature	Description	Type
Source IP	-	Independent
Destination IP	-	Independent
Source port	-	Independent
Destination port	-	Independent
Protocol	-	Independent
PX (total number of packets exchanged)	Used to separate normal and P2P traffic and group hosts with similar activities	Independent
NNP (number of null packets exchanged)	-	Independent
NSP (number of small packets exchanged)	-	Decentralized
PSP (percentage of small packets exchanged)	-	Decentralized
IOPR (ratio between the number of incoming packets over the number of outgoing packets)	Used to identify similar communications	Independent
Reconnect (number of reconnects)	To capture bots that perform frequent reconnections to evade detection	Independent
Duration (flow duration)	Used to classifying traffic into IRC chat and non IRC. In some botnets e.g. weasel that establish brief connections, this feature can also be used	Centralized

	to identify similar malicious communications	
FPS (length of the first packet)	Many protocols show identical behavior when exchanging the first packet	Independent
TBT (total number of bytes)	This feature is used to extract similarity in botnets traffic, e.g. fixed length commands	Independent
APL (average payload packet length)	This feature is used to extract similarity in botnets traffic	Independent
DPL (total number of packets with the same length over the total number of packets)	This feature is used to extract similarity in botnets traffic	Independent
PV (Standard deviation of payload packet length)	This feature is used to extract similarity in botnets traffic and also classify IRC traffic from non IRC traffic	Independent
BS (average bits-per-second)	This feature is used to extract similarity in botnets traffic and also to classify IRC traffic from non IRC traffic	Independent
PS (average packets-per-second in a time window)	This feature is used to extract similarity in botnet traffic and also to classify IRC traffic and non IRC traffic	Independent

AIT (average inter arrival time of packets)	-	Independent
PPS (average packets-per-second)	-	Independent

"TABLE I Summary of flow features used in machine learning based detection approaches"

Annex C Features from all datasets

In this annex a table containing all the features found in all analyzed datasets can be found.

#	Feature	Description	Count	CIC IDS 2018	CIC IDS 2017	CIDDS-001	CIDDS-002	ISCX 2012	TUIDS v1	TUIDS v2	TWENTE	CTU-13	UGR'16	SSHCure	Kent 2016	Unified Host and Network	CIC DDoS 2019	LITNET 2020	UNSW-NB 15
1	src_ip	Source IP address	14			X	X	X	X	X	X	X	X	X	X	X	X	X	X
2	src_port	Source port	14			X	X	X	X	X	X	X	X	X	X	X	X	X	X
3	dst_ip	Destination IP address	14			X	X	X	X	X	X	X	X	X	X	X	X	X	X
4	dst_port	Destination port	16	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
5	src_mask	Source mask	1															X	
6	dst_mask	Destination mask	1															X	
7	fwd_status	Forwarding status is encoded on 1 byte with the 2 left bits giving the status and the 6 remaining bits giving the reason code.	2										X					X	
8	network_proto	Network layer protocol	1					X											
9	transport_proto	Transport layer protocol	15	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X
10	application_proto	Application layer protocol	2					X											X
11	rel_time	Relative time since capture start	4			X	X								X	X			
12	ts_beginning	Timestamp (beginning of the flow)	8	X				X			X	X		X			X	X	X
13	ts_end	Timestamp (end of the flow)	5					X			X		X					X	X
14	fl_dur	Flow duration	14	X	X	X	X		X	X		X	X	X	X	X	X	X	X
15	tot_pkt	Total # of packets sent in either direction	1									X							
16	tot_l_pkt	Total size of packet (in bytes) sent in either direction	1									X							
17	tot_fw_pkt	Total # of packets sent in the forward direction	13	X	X	X	X	X			X		X	X	X	X	X	X	X
18	tot_bw_pkt	Total # of packets sent in the backward direction	7	X	X			X								X	X	X	X
19	tot_l_fw_pkt	Total size of packets (in bytes) sent in forward direction	16	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
20	tot_l_bw_pkt	Total size of packets (in bytes) sent in backward direction	9	X	X			X	X	X						X	X	X	X
21	fw_payload_b64	Packet payload sent in forward direction in base64	1					X											
22	fw_payload_utf	Packet payload sent in forward direction in UTF	1					X											
23	bw_payload_b64	Packet payload sent in backward direction in base64	1					X											
24	bw_payload_utf	Packet payload sent in backward direction in UTF	1					X											
25	sTos	Source TOS byte value	7			X	X		X	X		X	X					X	
26	dTos	Destination TOS byte value	2									X						X	
27	land	If source (1) and destination (3)IP addresses equal and port numbers (2)(4) equal then, this variable takes value 1 else 0	4						X	X								X	X
28	conn_status	Status of the connection (e.g., '1' for complete, '0' for reset)	1							X									
29	state	Indicates the principle state for the transaction report, and is protocol dependent	2									X							X
30	sttl	Source to destination time to live value	1																X
31	dttl	Destination to source time to live value	1																X
32	ct_state_ttl	No. for each state (27) according to specific range of values for source/destination time to live (28) (29).	1																X
33	sloss	Source packets retransmitted or dropped	1																X
34	dloss	Destination packets retransmitted or dropped	1																X
35	fw_pkt_l_max	Maximum size of packet in forward direction	3	X	X												X		
36	fw_pkt_l_min	Minimum size of packet in forward direction	3	X	X												X		
37	fw_pkt_l_avg	Average size of packet in forward direction	4	X	X												X		X
38	fw_pkt_l_std	Standard deviation size of packet in forward direction	3	X	X												X		
39	Bw_pkt_l_max	Maximum size of packet in backward direction	3	X	X												X		
40	Bw_pkt_l_min	Minimum size of packet in backward direction	3	X	X												X		

(continued on next page)

#	Feature	Description	Count	CIC IDS 2018	CIC IDS 2017	CIDDS-001	CIDDS-002	ISCX 2012	TUIDS v1	TUIDS v2	TWENTE	CTU-13	UGR'16	SSHCure	Kent 2016	Unified Host and Network	CIC DDoS 2019	LITNET 2020	UNSW-NB 15
41	Bw_pkt_l_avg	Mean size of packet in backward direction	4	X	X												X		X
42	Bw_pkt_l_std	Standard deviation size of packet in backward direction	3	X	X												X		
43	fl_byt_s	flow byte rate that is number of bytes transferred per second	3	X	X												X		
44	fl_pkt_s	flow packets rate that is number of packets transferred per second	3	X	X												X		
45	sload	Source bits per second	1																X
46	dload	Destination bits per second	1																X
47	fl_iat_avg	Average time between two flows	3	X	X												X		
48	fl_iat_std	Standard deviation time two flows	3	X	X												X		
49	fl_iat_max	Maximum time between two flows	3	X	X												X		
50	fl_iat_min	Minimum time between two flows	3	X	X												X		
51	fw_iat_tot	Total time between two packets sent in the forward direction	4	X	X												X		X
52	fw_iat_avg	Mean time between two packets sent in the forward direction	3	X	X												X		
53	fw_iat_std	Standard deviation time between two packets sent in the forward direction	3	X	X												X		
54	fw_iat_max	Maximum time between two packets sent in the forward direction	3	X	X												X		
55	fw_iat_min	Minimum time between two packets sent in the forward direction	3	X	X												X		
56	bw_iat_tot	Total time between two packets sent in the backward direction	4	X	X												X		X
57	bw_iat_avg	Mean time between two packets sent in the backward direction	3	X	X												X		
58	bw_iat_std	Standard deviation time between two packets sent in the backward direction	3	X	X												X		
59	bw_iat_max	Maximum time between two packets sent in the backward direction	3	X	X												X		
60	bw_iat_min	Minimum time between two packets sent in the backward direction	3	X	X												X		
61	fw_psh_flag	Number of times the PSH flag was set in packets travelling in the forward direction (0 for UDP)	3	X	X												X		
62	bw_psh_flag	Number of times the PSH flag was set in packets travelling in the backward direction (0 for UDP)	3	X	X												X		
63	fw_urg_flag	Number of times the URG flag was set in packets travelling in the forward direction (0 for UDP)	3	X	X												X		
64	bw_urg_flag	Number of times the URG flag was set in packets travelling in the backward direction (0 for UDP)	3	X	X												X		
65	fw_hdr_len	Total bytes used for headers in the forward direction	3	X	X												X		
66	bw_hdr_len	Total bytes used for headers in the forward direction	3	X	X												X		
67	fw_pkt_s	Number of forward packets per second	3	X	X												X		
68	bw_pkt_s	Number of backward packets per second	3	X	X												X		
69	pkt_len_min	Minimum length of a flow	3	X	X												X		
70	pkt_len_max	Maximum length of a flow	3	X	X												X		
71	pkt_len_avg	Mean length of a flow	3	X	X												X		
72	pkt_len_std	Standard deviation length of a flow	3	X	X												X		
73	pkt_len_va	Minimum inter-arrival time of packet	3	X	X												X		
74	fin_cnt	Number of packets with FIN	5	X	X				X	X							X		
75	syn_cnt	Number of packets with SYN	5	X	X				X	X							X		
76	rst_cnt	Number of packets with RST	5	X	X				X	X							X		
77	psh_cnt	Number of packets with PUSH	5	X	X				X	X							X		
78	ack_cnt	Number of packets with ACK	5	X	X				X	X							X		
79	urg_cnt	Number of packets with URG	5	X	X				X	X							X		
80	cwe_cnt	Number of packets with CWE	3	X	X												X		
81	ece_cnt	Number of packets with ECE	3	X	X												X		
82	tcp_flags	TCP flags which were set on packets sent in either direction (ex. "FSA" for the flags FIN, SYN and ACK)	1															X	
83	tcp_flags_fw	TCP flags which were set on packets sent in forward direction (ex. "FSA" for the flags FIN, SYN and ACK)	6			X	X	X			X		X					X	
84	tcp_flags_bw	TCP flags which were set on packets sent in backward direction (ex. "FSA" for the flags FIN, SYN and ACK)	1					X											
85	synack	TCP connection setup time, the time between the SYN and the SYN_ACK packets.	1																X

(continued on next page)

#	Feature	Description	Count	CIC IDS 2018	CIC IDS 2017	CIDDS-001	CIDDS-002	ISCX 2012	TUIDS v1	TUIDS v2	TWENTE	CTU-13	UGR'16	SSHCure	Kent 2016	Unified Host and Network	CIC DDoS 2019	LITNET 2020	UNSW-NB 15
86	ackdat	TCP connection setup time, the time between the SYN_ACK and the ACK packets.	1																X
87	tcprtt	TCP connection setup round-trip time, the sum of 'synack' and 'ackdat'.	1																X
88	down_up_ratio	Download and upload ratio	3	X	X												X		
89	pkt_size_avg	Average size of packet	3	X	X												X		
90	fw_seg_avg	Average size observed in the forward direction	3	X	X												X		
91	bw_seg_avg	Average size observed in the backward direction	3	X	X												X		
92	fw_byt_blk_avg	Average number of bytes bulk rate in the forward direction	3	X	X												X		
93	fw_pkt_blk_avg	Average number of packets bulk rate in the forward direction	3	X	X												X		
94	fw_blk_rate_avg	Average number of bulk rate in the forward direction	3	X	X												X		
95	bw_byt_blk_avg	Average number of bytes bulk rate in the backward direction	3	X	X												X		
96	bw_pkt_blk_avg	Average number of packets bulk rate in the backward direction	3	X	X												X		
97	bw_blk_rate_avg	Average number of bulk rate in the backward direction	3	X	X												X		
98	subfl_fw_pk	The average number of packets in a sub flow in the forward direction	3	X	X												X		
99	subfl_fw_byt	The average number of bytes in a sub flow in the forward direction	3	X	X												X		
100	subfl_bw_pkt	The average number of packets in a sub flow in the backward direction	3	X	X												X		
101	subfl_bw_byt	The average number of bytes in a sub flow in the backward direction	3	X	X												X		
102	fw_win_byt	Number of bytes sent in initial window in the forward direction	3	X	X												X		
103	bw_win_byt	# of bytes sent in initial window in the backward direction	3	X	X												X		
104	swin	Source TCP advertisement value	1																X
105	dwin	Destination TCP advertisement value	1																X
106	stcpb	Source TCP base sequence number	1																X
107	dtcpb	Destination TCP base sequence number	1																X
108	Fw_act_pkt	# of packets with at least 1 byte of TCP data payload in the forward direction	3	X	X												X		
109	fw_seg_min	Minimum segment size observed in the forward direction	3	X	X												X		
110	atv_avg	Mean time a flow was active before becoming idle	3	X	X												X		
111	atv_std	Standard deviation time a flow was active before becoming idle	3	X	X												X		
112	atv_max	Maximum time a flow was active before becoming idle	3	X	X												X		
113	atv_min	Minimum time a flow was active before becoming idle	3	X	X												X		
114	idl_avg	Mean time a flow was idle before becoming active	3	X	X												X		
115	idl_std	Standard deviation time a flow was idle before becoming active	3	X	X												X		
116	idl_max	Maximum time a flow was idle before becoming active	3	X	X												X		
117	idl_min	Minimum time a flow was idle before becoming active	3	X	X												X		
118	netw_dir	4 unique values, consisting of 'L2L', 'L2R', 'R2L', and 'R2R' which stand for local-to-local, local-to-remote, remote-to-local, and remote-to-remote respectively.	1					X											
119	host_dir	Direction of the transaction, as can be best determined from the datum, and is used to indicate which hosts are transmitting	1									X							
120	similar_http	?	1														X		
121	inbound	?	1														X		
122	trans_depth	Represents the pipelined depth into the connection of http request/response transaction	1																X
123	res_bdy_len	Actual uncompressed content size of the data transferred from the server's http service.	1																X
124	ct_flw_http_mthd	No. of flows that has methods such as Get and Post in http service.	1																X
125	Sjit	Source jitter (mSec)	1																X
126	Djit	Destination jitter (mSec)	1																X
127	ct_ftp_cmd	No of flows that has a command in ftp session.	1																X
128	is_ftp_login	If the ftp session is accessed by user and password then 1 else 0.	1																X
129	in	Input interface num	1															X	
130	out	Output interface num	1															X	

(continued on next page)

#	Feature	Description	Count	CIC IDS 2018	CIC IDS 2017	CIDDS-001	CIDDS-002	ISCX 2012	TUIDS v1	TUIDS v2	TWENTE	CTU-13	UGR'16	SSHCure	Kent 2016	Unified Host and Network	CIC DDoS 2019	LITNET 2020	UNSW-NB 15
131	sas	Source AS	1															X	
132	das	Destination AS	1															X	
133	nf_c_dir	Indicates if the packets were collected when arriving or leaving the NetFlow collector	1															X	
134	nh	Next-hop IP address	1															X	
135	nhb	BGP Next-hop IP Address	1															X	
136	svln	Src vlan label	1															X	
137	dvln	Dst vlan label	1															X	
138	ismc	Input Src Mac Addr	1															X	
139	odmc	Output Dst Mac Addr	1															X	
140	idmc	Input Dst Mac Addr	1															X	
141	osmc	Output Src Mac Addr	1															X	
142	mpls1	MPLS label 1	1															X	
143	mpls2	MPLS label 2	1															X	
144	mpls3	MPLS label 3	1															X	
145	mpls4	MPLS label 4	1															X	
146	mpls5	MPLS label 5	1															X	
147	mpls6	MPLS label 6	1															X	
148	mpls7	MPLS label 7	1															X	
149	mpls8	MPLS label 8	1															X	
150	mpls9	MPLS label 9	1															X	
151	mpls10	MPLS label 10	1															X	
152	cl	Client latency	1															X	
153	sl	Server latency	1															X	
154	al	Application latency	1															X	
155	ra	Router IP Address	1															X	
156	eng	Engine Type/ID	1															X	
157	exid	??	1															X	
158	tr	Time the flow was received by the collector	1															X	
159	icmp_dst_ip_b	Broadcast requests to the network on behalf of the victim computer	1															X	
160	icmp_src_ip	Large traffic of ICMP packets	1															X	
161	udp_dst_p	Large traffic flow to DNS	1															X	
162	tcp_f_s	Large traffic of TCP traffic with SYN attack	1															X	
163	tcp_f_n_a	Large traffic of TCP traffic with SYN attack	1															X	
164	tcp_f_n_f	Large traffic of TCP traffic with SYN attack	1															X	
165	tcp_f_n_r	Large traffic of TCP traffic with SYN attack	1															X	
166	tcp_f_n_p	Large traffic of TCP traffic with SYN attack	1															X	
167	tcp_f_n_u	Large traffic of TCP traffic with SYN attack	1															X	
168	tcp_dst_p	High traffic with HTTP protocol	1															X	
169	tcp_src_tftp	Large volume of traffic towards TFTP port	1															X	
170	tcp_src_kerb	Large volume of traffic towards Kerberos authentication port	1															X	
171	tcp_src_rpc	Large volume of traffic towards Remote Procedure Call (RPC) port	1															X	
172	tcp_dst_p_src	Uses a vulnerability in a HTTP server	1															X	
173	smtp_dst	The presence of an excessively large number of SMTP connections	1															X	
174	udp_p_r_range	Reaper is a botnet that uses the HTTP-based exploits of known vulnerabilities in IoT. Scans on UDP ports 80, 8080, 81, 88, 8081, 82, 83, 84, 1080, 3000, 3749, 8001, 8060, 8090, 8443, 8880, and 10,000.	1															X	
175	p_range_dst	An abnormal number of connections from one host to one or more other hosts	1															X	
176	udp_src_p_0	Denial of service attacks are based on the use of many fragmented packets	1															X	
177	label	Class label (attack vs normal) (different format for all datasets)	13	X	X	X	X	X	X	X	X	X	X				X	X	X
178	attack_type	Type of Attack (portScan, dos, bruteForce, -)	4			X	X											X	X
179	attack_id	Unique attack id. All flows which belong to the same attack carry the same attack id.	2			X	X												
180	attack_description	Provides additional information about the set attack parameters (e.g. the number of attempted password guesses for SSH-Brute-Force attacks)	2			X	X												

Legend:
X - Feature in the dataset might represent something slightly different; needs manual check with the dataset features

To note:

- For unidirectional flows, "forward direction" is referring to the only direction that exists. For bidirectional flows, "forward direction" is the direction from the host which sent the first packet to the other one.
- For the feature "attack_type", it is important to note that other datasets might actually specify the type of attacks a data instance is related to. It is just that if they have no X for this feature, it just means that they don't have an extra feature for that purpose and actually specify it right in the "label" feature

Annex D “Retour de flamme” scenario log files

D.1 General key structure

timestamp
flow_id
in_iface
event_type
src_ip
src_port
dest_ip
dest_port
proto
dns
    type
    id
    rrname
    rrtype
    tx_id
    rcode
    flags
    qr
    aa
    rd
    ra
    ttl
    rdata
app_proto
flow
    pkts_toserver
    pkts_toclient
    bytes_toserver
    bytes_toclient
    start
    end
    age
    state
    reason
    alerted
tcp
    tcp_flags
    tcp_flags_ts
    tcp_flags_tc
    syn
    fin
    rst
    psh
    ack
    state
    ecn
    cwr

stats
- uptime
- capture
  - kernel_packets
  - kernel_packets_delta
  - kernel_drops
  - kernel_drops_delta
- decoder
  - pkts
  - pkts_delta
  - bytes
  - bytes_delta
  - invalid
  - invalid_delta
  - ipv4
  - ipv4_delta
  - ipv6
  - ipv6_delta
  - ethernet
  - ethernet_delta
  - raw
  - raw_delta
  - null
  - null_delta
  - sll
  - sll_delta
  - tcp
  - tcp_delta
  - udp
  - udp_delta
  - sctp
  - sctp_delta
  - icmpv4
  - icmpv4_delta
  - icmpv6
  - icmpv6_delta
  - ppp
  - ppp_delta
  - pppoe
  - pppoe_delta
  - gre
  - gre_delta
  - vlan
  - vlan_delta
  - vlan_qinq
  - vlan_qinq_delta
  - ieee8021ah
  - ieee8021ah_delta
  - teredo
  - teredo_delta
  - ipv4_in_ipv6
  - ipv4_in_ipv6_delta
  - ipv6_in_ipv6
  - ipv6_in_ipv6_delta
  - mpls
  - mpls_delta
  - avg_pkt_size

- avg_pkt_size_delta
- max_pkt_size
- max_pkt_size_delta
- erspan
- erspan_delta
- ipraw
  - invalid_ip_version
  - invalid_ip_version_delta
- ltnull
  - pkt_too_small
  - pkt_too_small_delta
  - unsupported_type
  - unsupported_type_delta
- dce
  - pkt_too_small
  - pkt_too_small_delta

flow
- memcap
- memcap_delta
- tcp
- tcp_delta
- udp
- udp_delta
- icmpv4
- icmpv4_delta
- icmpv6
- icmpv6_delta
- spare
- spare_delta
- emerg_mode_entered
- emerg_mode_entered_delta
- emerg_mode_over
- emerg_mode_over_delta
- tcp_reuse
- tcp_reuse_delta
- memuse
- memuse_delta

defrag
- ipv4
  - fragments
  - fragments_delta
  - reassembled
  - reassembled_delta
  - timeouts
  - timeouts_delta
- ipv6
  - fragments
  - fragments_delta
  - reassembled
  - reassembled_delta
  - timeouts
  - timeouts_delta
- max_frag_hits
- max_frag_hits_delta

tcp
- sessions
- sessions_delta

ssn_memcap_drop
ssn_memcap_drop_delta
pseudo
pseudo_delta
pseudo_failed
pseudo_failed_delta
invalid_checksum
invalid_checksum_delta
no_flow
no_flow_delta
syn
syn_delta
synack
synack_delta
rst
rst_delta
segment_memcap_drop
segment_memcap_drop_delta
stream_depth_reached
stream_depth_reached_delta
reassembly_gap
reassembly_gap_delta
overlap
overlap_delta
overlap_diff_data
overlap_diff_data_delta
insert_data_normal_fail
insert_data_normal_fail_delta
insert_data_overlap_fail
insert_data_overlap_fail_delta
insert_list_fail
insert_list_fail_delta
memuse
memuse_delta
reassembly_memuse
reassembly_memuse_delta
detect
engines
alert
alert_delta
app_layer
flow
http
http_delta
ftp
ftp_delta
smtp
smtp_delta
tls
tls_delta
ssh
ssh_delta
imap
imap_delta
msn
msn_delta
smb

```
                smb_delta
                dcerpc_tcp
                dcerpc_tcp_delta
                dns_tcp
                dns_tcp_delta
                ftp-data
                ftp-data_delta
                failed_tcp
                failed_tcp_delta
                dcerpc_udp
                dcerpc_udp_delta
                dns_udp
                dns_udp_delta
                failed_udp
                failed_udp_delta
        tx
                http
                http_delta
                ftp
                ftp_delta
                smtp
                smtp_delta
                tls
                tls_delta
                ssh
                ssh_delta
                smb
                smb_delta
                dcerpc_tcp
                dcerpc_tcp_delta
                dns_tcp
                dns_tcp_delta
                ftp-data
                ftp-data_delta
                dcerpc_udp
                dcerpc_udp_delta
                dns_udp
                dns_udp_delta
        expectations
        expectations_delta
flow_mgr
        closed_pruned
        closed_pruned_delta
        new_pruned
        new_pruned_delta
        est_pruned
        est_pruned_delta
        bypassed_pruned
        bypassed_pruned_delta
        flows_checked
        flows_checked_delta
        flows_notimeout
        flows_notimeout_delta
        flows_timeout
        flows_timeout_delta
        flows_timeout_inuse
        flows_timeout_inuse_delta
```

```
                flows_removed
                flows_removed_delta
                rows_checked
                rows_checked_delta
                rows_skipped
                rows_skipped_delta
                rows_empty
                rows_empty_delta
                rows_busy
                rows_busy_delta
                rows_maxlen
                rows_maxlen_delta
        dns
                memuse
                memuse_delta
                memcap_state
                memcap_state_delta
                memcap_global
                memcap_global_delta
        http
                memuse
                memuse_delta
                memcap
                memcap_delta
        ftp
                memuse
                memuse_delta
                memcap
                memcap_delta
app_proto_tc
icmp_type
icmp_code
tx_id
http
        hostname
        url
        http_method
        protocol
        length
        status
app_proto_orig
```

D.2 Code

```
1.  import json
2.  from json import JSONDecodeError
3.
4.  from model.services.io_service import IOService
5.
6.  def identify_unique_keys(entry, dictionary):
7.      for k in entry.keys():
8.          if k not in dictionary:
9.              dictionary[k] = {}
10.         if isinstance(entry[k], dict):
11.             dictionary[k] = identify_unique_keys(entry[k], dictionary[k])
12.     return dictionary
13.
```

```
14.
15. def show_key_structure(str_before, dictionary):
16.     for k in dictionary:
17.         print(str_before + k, flush=True)
18.         if isinstance(dictionary[k], dict) and dictionary[k]:
19.             show_key_structure(str_before + "\t", dictionary[k])
20.
21.
22. final = {}
23. for current_entry in IOService().read_lines_file(
24.         "C:\\Users\\Pedro\\Documents\\pesti_test_inputs\\retourdeflamme\\192
    .168.134.10.log"):
25.     current_entry_split = current_entry.split()
26.     try:
27.         dict_entry = json.loads(current_entry_split[-1])
28.         final = identify_unique_keys(dict_entry, final)
29.     except JSONDecodeError:
30.         continue
31.
32. show_key_structure("", final)
```

Annex E Graylog files

E.1 General key structure

original_index

original_type

original_id

score

source

- flow_pkts_toclient
- source
- gl2_source_input
- src_ip
- MESSAGE
  - timestamp
  - flow_id
  - in_iface
  - event_type
  - src_ip
  - src_port
  - dest_ip
  - dest_port
  - proto
  - app_proto
  - flow
    - pkts_toserver
    - pkts_toclient
    - bytes_toserver
    - bytes_toclient
    - start
    - end
    - age
    - state
    - reason
    - alerted
  - stats
    - uptime
    - capture

```
            kernel_packets
            kernel_drops
            errors
    decoder
            pkts
            bytes
            invalid
            ipv4
            ipv6
            ethernet
            raw
            null
            sll
            tcp
            udp
            sctp
            icmpv4
            icmpv6
            ppp
            pppoe
            gre
            vlan
            vlan_qinq
            vxlan
            ieee8021ah
            teredo
            ipv4_in_ipv6
            ipv6_in_ipv6
            mpls
            avg_pkt_size
            max_pkt_size
            erspan
            event
                    ipv4
                            pkt_too_small
                            hlen_too_small
                            iplen_smaller_than_hlen
                            trunc_pkt
```

opt_invalid
opt_invalid_len
opt_malformed
opt_pad_required
opt_eol_required
opt_duplicate
opt_unknown
wrong_ip_version
icmpv6
frag_pkt_too_large
frag_overlap
frag_ignored

icmpv4

pkt_too_small
unknown_type
unknown_code
ipv4_trunc_pkt
ipv4_unknown_ver

icmpv6

unknown_type
unknown_code
pkt_too_small
ipv6_unknown_version
ipv6_trunc_pkt
mld_message_with_invalid_hl
unassigned_type
experimentation_type

ipv6

pkt_too_small
trunc_pkt
trunc_exthdr
exthdr_dupl_fh
exthdr_useless_fh
exthdr_dupl_rh
exthdr_dupl_hh
exthdr_dupl_dh
exthdr_dupl_ah
exthdr_dupl_eh

exthdr_invalid_optlen
wrong_ip_version
exthdr_ah_res_not_null
hopopts_unknown_opt
hopopts_only_padding
dstopts_unknown_opt
dstopts_only_padding
rh_type_0
zero_len_padn
fh_non_zero_reserved_field
data_after_none_header
unknown_next_header
icmpv4
frag_pkt_too_large
frag_overlap
frag_ignored
ipv4_in_ipv6_too_small
ipv4_in_ipv6_wrong_version
ipv6_in_ipv6_too_small
ipv6_in_ipv6_wrong_version

tcp
pkt_too_small
hlen_too_small
invalid_optlen
opt_invalid_len
opt_duplicate

udp
pkt_too_small
hlen_too_small
hlen_invalid

sll
pkt_too_small

ethernet
pkt_too_small

ppp
pkt_too_small
vju_pkt_too_small
ip4_pkt_too_small

ip6_pkt_too_small
wrong_type
unsup_proto
pppoe
pkt_too_small
wrong_code
malformed_tags
gre
pkt_too_small
wrong_version
version0_recur
version0_flags
version0_hdr_too_big
version0_malformed_sre_hdr
version1_chksum
version1_route
version1_ssr
version1_recur
version1_flags
version1_no_key
version1_wrong_protocol
version1_malformed_sre_hdr
version1_hdr_too_big
vlan
header_too_small
unknown_type
too_many_layers
ieee8021ah
header_too_small
ipraw
invalid_ip_version
ltnull
pkt_too_small
unsupported_type
sctp
pkt_too_small
mpls
header_too_small

```
                                pkt_too_small
                                bad_label_router_alert
                                bad_label_implicit_null
                                bad_label_reserved
                                unknown_payload_type
                        erspan
                                header_too_small
                                unsupported_version
                                too_many_vlan_layers
                dce
                        pkt_too_small
        flow
                memcap
                tcp
                udp
                icmpv4
                icmpv6
                spare
                emerg_mode_entered
                emerg_mode_over
                tcp_reuse
                memuse
        defrag
                ipv4
                        fragments
                        reassembled
                        timeouts
                ipv6
                        fragments
                        reassembled
                        timeouts
                max_frag_hits
        flow_bypassed
                local_pkts
                local_bytes
                local_capture_pkts
                local_capture_bytes
                closed
```

pkts

bytes

tcp

sessions

ssn_memcap_drop

pseudo

pseudo_failed

invalid_checksum

no_flow

syn

synack

rst

midstream_pickups

pkt_on_wrong_thread

segment_memcap_drop

stream_depth_reached

reassembly_gap

overlap

overlap_diff_data

insert_data_normal_fail

insert_data_overlap_fail

insert_list_fail

memuse

reassembly_memuse

detect

engines

alert

file_store

open_files_max_hit

fs_errors

open_files

app_layer

flow

http

ftp

smtp

tls

ssh

imap

smb

dcerpc_tcp

dns_tcp

nfs_tcp

ntp

ftp-data

tftp

ikev2

krb5_tcp

dhcp

snmp

failed_tcp

dcerpc_udp

dns_udp

nfs_udp

krb5_udp

failed_udp

tx

http

ftp

smtp

tls

ssh

imap

smb

dcerpc_tcp

dns_tcp

nfs_tcp

ntp

ftp-data

tftp

ikev2

krb5_tcp

dhcp

snmp

dcerpc_udp

dns_udp

- nfs_udp
- krb5_udp
- expectations
- flow_mgr
  - closed_pruned
  - new_pruned
  - est_pruned
  - bypassed_pruned
  - flows_checked
  - flows_notimeout
  - flows_timeout
  - flows_timeout_inuse
  - flows_removed
  - rows_checked
  - rows_skipped
  - rows_empty
  - rows_busy
  - rows_maxlen
- http
  - memuse
  - memcap
- ftp
  - memuse
  - memcap

tcp

- tcp_flags
- tcp_flags_ts
- tcp_flags_tc
- syn
- state

dns

- type
- id
- rrname
- rrtype
- tx_id

icmp_type

icmp_code

smb

id

dialect

command

status

status_code

session_id

tree_id

ntlmssp

domain

user

host

flow_age

event_type

flow_id

flow_pkts_toserver

gl2_source_node

flow_reason

dest_port

timestamp

flow_start

flow_bytes_toserver

level

streams

message

timestamp

flow_id

in_iface

event_type

src_ip

src_port

dest_ip

dest_port

proto

app_proto

flow

pkts_toserver

pkts_toclient

bytes_toserver

bytes_toclient

start

end

age

state

reason

alerted

stats

uptime

capture

kernel_packets

kernel_drops

errors

decoder

pkts

bytes

invalid

ipv4

ipv6

ethernet

raw

null

sll

tcp

udp

sctp

icmpv4

icmpv6

ppp

pppoe

gre

vlan

vlan_qinq

vxlan

ieee8021ah

teredo

ipv4_in_ipv6

- ipv6_in_ipv6
- mpls
- avg_pkt_size
- max_pkt_size
- erspan
- event
  - ipv4
    - pkt_too_small
    - hlen_too_small
    - iplen_smaller_than_hlen
    - trunc_pkt
    - opt_invalid
    - opt_invalid_len
    - opt_malformed
    - opt_pad_required
    - opt_eol_required
    - opt_duplicate
    - opt_unknown
    - wrong_ip_version
    - icmpv6
    - frag_pkt_too_large
    - frag_overlap
    - frag_ignored
  - icmpv4
    - pkt_too_small
    - unknown_type
    - unknown_code
    - ipv4_trunc_pkt
    - ipv4_unknown_ver
  - icmpv6
    - unknown_type
    - unknown_code
    - pkt_too_small
    - ipv6_unknown_version
    - ipv6_trunc_pkt
    - mld_message_with_invalid_hl
    - unassigned_type
    - experimentation_type

ipv6

- pkt_too_small
- trunc_pkt
- trunc_exthdr
- exthdr_dupl_fh
- exthdr_useless_fh
- exthdr_dupl_rh
- exthdr_dupl_hh
- exthdr_dupl_dh
- exthdr_dupl_ah
- exthdr_dupl_eh
- exthdr_invalid_optlen
- wrong_ip_version
- exthdr_ah_res_not_null
- hopopts_unknown_opt
- hopopts_only_padding
- dstopts_unknown_opt
- dstopts_only_padding
- rh_type_0
- zero_len_padn
- fh_non_zero_reserved_field
- data_after_none_header
- unknown_next_header
- icmpv4
- frag_pkt_too_large
- frag_overlap
- frag_ignored
- ipv4_in_ipv6_too_small
- ipv4_in_ipv6_wrong_version
- ipv6_in_ipv6_too_small
- ipv6_in_ipv6_wrong_version

tcp

- pkt_too_small
- hlen_too_small
- invalid_optlen
- opt_invalid_len
- opt_duplicate

udp

pkt_too_small

hlen_too_small

hlen_invalid

sll

pkt_too_small

ethernet

pkt_too_small

ppp

pkt_too_small

vju_pkt_too_small

ip4_pkt_too_small

ip6_pkt_too_small

wrong_type

unsup_proto

pppoe

pkt_too_small

wrong_code

malformed_tags

gre

pkt_too_small

wrong_version

version0_recur

version0_flags

version0_hdr_too_big

version0_malformed_sre_hdr

version1_chksum

version1_route

version1_ssr

version1_recur

version1_flags

version1_no_key

version1_wrong_protocol

version1_malformed_sre_hdr

version1_hdr_too_big

vlan

header_too_small

unknown_type

too_many_layers

```
                ieee8021ah
                    header_too_small
                ipraw
                    invalid_ip_version
                ltnull
                    pkt_too_small
                    unsupported_type
                sctp
                    pkt_too_small
                mpls
                    header_too_small
                    pkt_too_small
                    bad_label_router_alert
                    bad_label_implicit_null
                    bad_label_reserved
                    unknown_payload_type
                erspan
                    header_too_small
                    unsupported_version
                    too_many_vlan_layers
            dce
                pkt_too_small
    flow
        memcap
        tcp
        udp
        icmpv4
        icmpv6
        spare
        emerg_mode_entered
        emerg_mode_over
        tcp_reuse
        memuse
    defrag
        ipv4
            fragments
            reassembled
            timeouts
```

- ipv6
  - fragments
  - reassembled
  - timeouts
- max_frag_hits

flow_bypassed
- local_pkts
- local_bytes
- local_capture_pkts
- local_capture_bytes
- closed
- pkts
- bytes

tcp
- sessions
- ssn_memcap_drop
- pseudo
- pseudo_failed
- invalid_checksum
- no_flow
- syn
- synack
- rst
- midstream_pickups
- pkt_on_wrong_thread
- segment_memcap_drop
- stream_depth_reached
- reassembly_gap
- overlap
- overlap_diff_data
- insert_data_normal_fail
- insert_data_overlap_fail
- insert_list_fail
- memuse
- reassembly_memuse

detect
- engines
- alert

file_store

open_files_max_hit

fs_errors

open_files

app_layer

flow

http

ftp

smtp

tls

ssh

imap

smb

dcerpc_tcp

dns_tcp

nfs_tcp

ntp

ftp-data

tftp

ikev2

krb5_tcp

dhcp

snmp

failed_tcp

dcerpc_udp

dns_udp

nfs_udp

krb5_udp

failed_udp

tx

http

ftp

smtp

tls

ssh

imap

smb

dcerpc_tcp

dns_tcp

nfs_tcp

ntp

ftp-data

tftp

ikev2

krb5_tcp

dhcp

snmp

dcerpc_udp

dns_udp

nfs_udp

krb5_udp

expectations

flow_mgr

closed_pruned

new_pruned

est_pruned

bypassed_pruned

flows_checked

flows_notimeout

flows_timeout

flows_timeout_inuse

flows_removed

rows_checked

rows_skipped

rows_empty

rows_busy

rows_maxlen

http

memuse

memcap

ftp

memuse

memcap

tcp

tcp_flags

tcp_flags_ts

- tcp_flags_tc
- syn
- state
- dns
- type
- id
- rrname
- rrtype
- tx_id
- icmp_type
- icmp_code
- smb
- id
- dialect
- command
- status
- status_code
- session_id
- tree_id
- ntlmssp
- domain
- user
- host
- app_proto
- flow_alerted
- in_iface
- src_port
- flow_state
- flow_end
- dest_ip
- proto
- flow_bytes_toclient
- facility
- stats_tcp_ssn_memcap_drop
- stats_app_layer_flow_dcerpc_udp
- stats_flow_mgr_closed_pruned
- stats_decoder_event_ppp_pkt_too_small
- stats_flow_mgr_flows_checked

stats_tcp_overlap

stats_decoder_ipv6_in_ipv6

stats_tcp_insert_data_normal_fail

stats_decoder_event_ipv4_opt_unknown

stats_decoder_event_erspan_too_many_vlan_layers

stats_decoder_event_mpls_bad_label_implicit_null

stats_decoder_event_ipv4_icmpv6

stats_decoder_event_ipv6_icmpv4

stats_decoder_event_ipv4_frag_pkt_too_large

stats_decoder_event_gre_version1_flags

stats_capture_errors

stats_flow_mgr_rows_checked

stats_decoder_event_ipv4_opt_invalid

stats_defrag_ipv4_fragments

stats_decoder_event_ltnull_pkt_too_small

stats_defrag_ipv4_reassembled

stats_tcp_midstream_pickups

stats_tcp_pseudo

stats_flow_mgr_est_pruned

stats_flow_mgr_rows_empty

stats_flow_mgr_flows_removed

stats_decoder_event_tcp_opt_invalid_len

stats_defrag_ipv6_reassembled

stats_tcp_memuse

stats_decoder_event_gre_version1_recur

stats_app_layer_flow_snmp

stats_app_layer_expectations

stats_tcp_stream_depth_reached

stats_http_memuse

stats_decoder_event_icmpv6_experimentation_type

stats_decoder_event_gre_version1_no_key

stats_decoder_event_mpls_unknown_payload_type

stats_flow_bypassed_local_bytes

stats_flow_mgr_flows_timeout_inuse

stats_decoder_event_ipv6_frag_overlap

stats_app_layer_flow_dns_udp

stats_decoder_gre

stats_app_layer_flow_ssh

stats_decoder_event_ipv4_iplen_smaller_than_hlen

stats_decoder_event_ipv4_opt_duplicate

stats_flow_spare

stats_decoder_event_ipv4_opt_eol_required

stats_decoder_event_ipv6_rh_type_0

stats_decoder_event_pppoe_pkt_too_small

stats_decoder_event_ppp_wrong_type

stats_decoder_event_ipv6_wrong_ip_version

stats_decoder_event_ieee8021ah_header_too_small

stats_defrag_max_frag_hits

stats_flow_bypassed_local_pkts

stats_app_layer_tx_ssh

stats_file_store_open_files

stats_app_layer_flow_dns_tcp

stats_app_layer_tx_snmp

stats_decoder_event_gre_version1_hdr_too_big

stats_decoder_event_mpls_pkt_too_small

stats_flow_bypassed_local_capture_bytes

stats_flow_emerg_mode_entered

stats_decoder_event_ipv6_trunc_exthdr

stats_decoder_sctp

stats_http_memcap

stats_decoder_event_gre_version0_hdr_too_big

stats_decoder_event_gre_pkt_too_small

stats_decoder_event_gre_version0_recur

stats_decoder_event_udp_hlen_invalid

stats_decoder_event_ipv6_trunc_pkt

stats_decoder_event_ipv6_frag_pkt_too_large

stats_flow_icmpv4

stats_app_layer_flow_nfs_udp

stats_app_layer_tx_imap

stats_flow_icmpv6

stats_decoder_event_icmpv6_pkt_too_small

stats_decoder_event_ipv6_exthdr_dupl_rh

stats_app_layer_tx_smtp

stats_app_layer_flow_krb5_tcp

stats_flow_mgr_rows_maxlen

stats_decoder_event_sctp_pkt_too_small

stats_decoder_event_mpls_bad_label_reserved

stats_app_layer_tx_tftp

stats_app_layer_flow_failed_udp

stats_decoder_event_ipv6_exthdr_dupl_ah

stats_decoder_event_icmpv4_ipv4_trunc_pkt

stats_app_layer_tx_nfs_udp

stats_flow_emerg_mode_over

stats_flow_mgr_flows_notimeout

stats_decoder_ppp

stats_decoder_event_ipv4_trunc_pkt

stats_decoder_event_icmpv4_unknown_code

stats_app_layer_flow_nfs_tcp

stats_tcp_pkt_on_wrong_thread

stats_decoder_event_ltnull_unsupported_type

stats_defrag_ipv6_timeouts

stats_decoder_event_ipv6_ipv4_in_ipv6_wrong_version

stats_app_layer_flow_krb5_udp

stats_tcp_synack

stats_ftp_memuse

stats_decoder_event_ipv6_exthdr_ah_res_not_null

stats_tcp_reassembly_gap

stats_decoder_dce_pkt_too_small

stats_app_layer_tx_http

stats_app_layer_flow_failed_tcp

stats_decoder_event_gre_version1_ssr

stats_decoder_event_ipv6_ipv6_in_ipv6_too_small

stats_decoder_event_ipv6_exthdr_invalid_optlen

stats_tcp_reassembly_memuse

stats_file_store_fs_errors

stats_flow_mgr_bypassed_pruned

stats_decoder_mpls

stats_flow_bypassed_local_capture_pkts

stats_decoder_event_ethernet_pkt_too_small

stats_decoder_event_ppp_vju_pkt_too_small

stats_decoder_ieee8021ah

stats_decoder_event_tcp_hlen_too_small

stats_flow_mgr_new_pruned

stats_decoder_event_mpls_bad_label_router_alert

stats_decoder_event_ipv6_hopopts_only_padding

stats_decoder_event_erspan_unsupported_version

stats_app_layer_flow_dcerpc_tcp

stats_decoder_event_icmpv6_ipv6_trunc_pkt

stats_tcp_overlap_diff_data

stats_decoder_event_ipv4_pkt_too_small

stats_decoder_erspan

stats_decoder_event_ipv4_opt_invalid_len

stats_tcp_insert_data_overlap_fail

stats_decoder_icmpv4

stats_decoder_event_icmpv6_ipv6_unknown_version

stats_decoder_event_icmpv6_unknown_type

stats_ftp_memcap

stats_decoder_event_vlan_too_many_layers

stats_app_layer_tx_nfs_tcp

stats_app_layer_tx_krb5_udp

stats_decoder_event_icmpv6_unassigned_type

stats_decoder_raw

stats_decoder_icmpv6

stats_app_layer_flow_tftp

stats_detect_alert

stats_app_layer_tx_ikev2

stats_decoder_event_ipv6_unknown_next_header

stats_flow_mgr_rows_busy

stats_app_layer_flow_smb

stats_decoder_event_gre_version1_route

stats_decoder_event_gre_version1_wrong_protocol

stats_flow_bypassed_bytes

stats_decoder_event_ipv4_opt_malformed

stats_app_layer_flow_ftp

stats_flow_memcap

stats_app_layer_tx_krb5_tcp

stats_decoder_event_icmpv6_unknown_code

stats_decoder_event_ipv6_exthdr_dupl_eh

stats_decoder_event_ipv4_wrong_ip_version

stats_tcp_invalid_checksum

stats_decoder_event_ipv6_hopopts_unknown_opt

stats_app_layer_tx_dhcp

stats_decoder_event_udp_pkt_too_small

stats_app_layer_flow_smtp

stats_decoder_event_ipv6_exthdr_useless_fh

stats_decoder_tcp

stats_decoder_event_ipv6_pkt_too_small

stats_decoder_event_ipv6_frag_ignored

stats_app_layer_flow_imap

stats_decoder_event_ipv6_exthdr_dupl_dh

stats_decoder_event_pppoe_wrong_code

stats_decoder_event_tcp_opt_duplicate

stats_decoder_event_erspan_header_too_small

stats_decoder_ipv4

stats_defrag_ipv4_timeouts

stats_decoder_ipv6

stats_decoder_event_sll_pkt_too_small

stats_decoder_bytes

stats_app_layer_tx_tls

stats_decoder_event_ipv6_ipv4_in_ipv6_too_small

stats_decoder_event_vlan_unknown_type

stats_decoder_event_ppp_unsup_proto

stats_decoder_vxlan

stats_decoder_event_icmpv4_pkt_too_small

stats_tcp_segment_memcap_drop

stats_app_layer_tx_dcerpc_tcp

stats_decoder_udp

stats_flow_memuse

stats_decoder_event_ipraw_invalid_ip_version

stats_decoder_vlan_qinq

stats_decoder_event_ipv6_data_after_none_header

stats_decoder_event_tcp_invalid_optlen

stats_decoder_event_gre_version0_malformed_sre_hdr

stats_decoder_event_icmpv6_mld_message_with_invalid_hl

stats_app_layer_flow_http

stats_tcp_rst

stats_decoder_invalid

stats_uptime

stats_decoder_event_ipv6_dstopts_only_padding

stats_app_layer_tx_dcerpc_udp

stats_decoder_event_udp_hlen_too_small

stats_tcp_sessions

stats_decoder_event_gre_wrong_version

stats_flow_bypassed_closed

stats_decoder_event_ipv6_exthdr_dupl_fh

stats_app_layer_flow_tls

stats_flow_udp

stats_decoder_ipv4_in_ipv6

stats_decoder_avg_pkt_size

stats_app_layer_tx_ftp-data

stats_decoder_teredo

stats_decoder_event_ipv6_ipv6_in_ipv6_wrong_version

stats_app_layer_flow_dhcp

stats_app_layer_tx_smb

stats_decoder_pkts

stats_decoder_event_pppoe_malformed_tags

stats_flow_tcp

stats_capture_kernel_packets

stats_decoder_event_ipv4_frag_overlap

stats_decoder_null

stats_decoder_ethernet

stats_decoder_event_ppp_ip4_pkt_too_small

stats_decoder_event_ipv6_zero_len_padn

stats_decoder_event_ipv4_frag_ignored

stats_decoder_event_gre_version0_flags

stats_tcp_pseudo_failed

stats_file_store_open_files_max_hit

stats_app_layer_flow_ikev2

stats_tcp_insert_list_fail

stats_decoder_event_tcp_pkt_too_small

stats_decoder_event_ipv6_exthdr_dupl_hh

stats_decoder_event_mpls_header_too_small

stats_decoder_max_pkt_size

stats_decoder_event_vlan_header_too_small

stats_flow_mgr_flows_timeout

stats_decoder_event_icmpv4_ipv4_unknown_ver

stats_decoder_event_ipv4_hlen_too_small

stats_decoder_event_ipv6_fh_non_zero_reserved_field

stats_decoder_event_gre_version1_chksum

stats_app_layer_tx_dns_tcp

stats_flow_tcp_reuse

stats_flow_mgr_rows_skipped

stats_capture_kernel_drops

stats_decoder_event_ppp_ip6_pkt_too_small

stats_defrag_ipv6_fragments

stats_decoder_event_gre_version1_malformed_sre_hdr

stats_app_layer_flow_ntp

stats_decoder_sll

stats_decoder_pppoe

stats_tcp_no_flow

stats_detect_engines

stats_decoder_vlan

stats_decoder_event_ipv6_dstopts_unknown_opt

stats_app_layer_tx_ftp

stats_decoder_event_icmpv4_unknown_type

stats_decoder_event_ipv4_opt_pad_required

stats_flow_bypassed_pkts

stats_tcp_syn

stats_app_layer_flow_ftp-data

stats_app_layer_tx_dns_udp

stats_app_layer_tx_ntp

tcp_state

tcp_tcp_flags

tcp_tcp_flags_ts

tcp_tcp_flags_tc

tcp_syn

dns_tx_id

dns_type

dns_rrname

dns_id

dns_rrtype

icmp_type

icmp_code

smb_dialect

smb_status_code

smb_id

smb_status

smb_session_id

smb_tree_id

smb_ntlmssp_host

smb_command

E.2 Code

```
import json

from model.services.io_service import IOService


def identify_unique_keys(entry, dictionary):
    for k in entry.keys():
        if k not in dictionary:
            dictionary[k] = {}
        if isinstance(entry[k], dict):
            dictionary[k] = identify_unique_keys(entry[k], dictionary[k])
    return dictionary


def show_key_structure(str_before, dictionary):
    for k in dictionary:
        print(str_before + k, flush=True)
        if isinstance(dictionary[k], dict) and dictionary[k]:
            show_key_structure(str_before + "\t", dictionary[k])


final = {}
for current_entry in IOService().read_json_file("C:\\Users\\Pedro\\Documents
\\pesti_test_inputs\\graylog\\dados_graylog-1.json"):
    current_entry["source"]["MESSAGE"] = json.loads(current_entry["source"][
"MESSAGE"])

    split = current_entry["source"]["message"].split()
    current_entry["source"]["message"] = json.loads(split[-1])

    final = identify_unique_keys(current_entry, final)

show_key_structure("", final)
```

Annex F CICIDS 2018 file format

The following are the features of the CICIDS 2018 dataset, in the order they appear as columns, in the files of the dataset.

Feature	**Description**
dst_port	Destination port
proto	Transport layer protocol
timestamp	Timestamp
fl_dur	Flow duration
tot_fw_pk	Total packets in the forward direction
tot_bw_pk	Total packets in the backward direction
tot_l_fw_pkt	Total size of packet in forward direction
tot_l_bw_pkt	Total size of packet in backward direction
fw_pkt_l_max	Maximum size of packet in forward direction
fw_pkt_l_min	Minimum size of packet in forward direction
fw_pkt_l_avg	Average size of packet in forward direction
fw_pkt_l_std	Standard deviation size of packet in forward direction
Bw_pkt_l_max	Maximum size of packet in backward direction
Bw_pkt_l_min	Minimum size of packet in backward direction
Bw_pkt_l_avg	Mean size of packet in backward direction
Bw_pkt_l_std	Standard deviation size of packet in backward direction
fl_byt_s	flow byte rate that is number of packets transferred per second
fl_pkt_s	flow packets rate that is number of packets transferred per second

fl_iat_avg	Average time between two flows
fl_iat_std	Standard deviation time two flows
fl_iat_max	Maximum time between two flows
fl_iat_min	Minimum time between two flows
fw_iat_tot	Total time between two packets sent in the forward direction
fw_iat_avg	Mean time between two packets sent in the forward direction
fw_iat_std	Standard deviation time between two packets sent in the forward direction
fw_iat_max	Maximum time between two packets sent in the forward direction
fw_iat_min	Minimum time between two packets sent in the forward direction
bw_iat_tot	Total time between two packets sent in the backward direction
bw_iat_avg	Mean time between two packets sent in the backward direction
bw_iat_std	Standard deviation time between two packets sent in the backward direction
bw_iat_max	Maximum time between two packets sent in the backward direction
bw_iat_min	Minimum time between two packets sent in the backward direction
fw_psh_flag	Number of times the PSH flag was set in packets travelling in the forward direction (0 for UDP)
bw_psh_flag	Number of times the PSH flag was set in packets travelling in the backward direction (0 for UDP)
fw_urg_flag	Number of times the URG flag was set in packets travelling in the forward direction (0 for UDP)
bw_urg_flag	Number of times the URG flag was set in packets travelling in the backward direction (0 for UDP)
fw_hdr_len	Total bytes used for headers in the forward direction

bw_hdr_len	Total bytes used for headers in the backward direction
fw_pkt_s	Number of forward packets per second
bw_pkt_s	Number of backward packets per second
pkt_len_min	Minimum length of a flow
pkt_len_max	Maximum length of a flow
pkt_len_avg	Mean length of a flow
pkt_len_std	Standard deviation length of a flow
pkt_len_va	Minimum inter-arrival time of packet
fin_cnt	Number of packets with FIN
syn_cnt	Number of packets with SYN
rst_cnt	Number of packets with RST
pst_cnt	Number of packets with PUSH
ack_cnt	Number of packets with ACK
urg_cnt	Number of packets with URG
cwe_cnt	Number of packets with CWE
ece_cnt	Number of packets with ECE
down_up_ratio	Download and upload ratio
pkt_size_avg	Average size of packet
fw_seg_avg	Average size observed in the forward direction
bw_seg_avg	Average size observed in the backward direction
fw_byt_blk_avg	Average number of bytes bulk rate in the forward direction
fw_pkt_blk_avg	Average number of packets bulk rate in the forward direction

fw_blk_rate_avg	Average number of bulk rate in the forward direction
bw_byt_blk_avg	Average number of bytes bulk rate in the backward direction
bw_pkt_blk_avg	Average number of packets bulk rate in the backward direction
bw_blk_rate_avg	Average number of bulk rate in the backward direction
subfl_fw_pk	The average number of packets in a sub flow in the forward direction
subfl_fw_byt	The average number of bytes in a sub flow in the forward direction
subfl_bw_pkt	The average number of packets in a sub flow in the backward direction
subfl_bw_byt	The average number of bytes in a sub flow in the backward direction
fw_win_byt	Number of bytes sent in initial window in the forward direction
bw_win_byt	# of bytes sent in initial window in the backward direction
Fw_act_pkt	# of packets with at least 1 byte of TCP data payload in the forward direction
fw_seg_min	Minimum segment size observed in the forward direction
atv_avg	Mean time a flow was active before becoming idle
atv_std	Standard deviation time a flow was active before becoming idle
atv_max	Maximum time a flow was active before becoming idle
atv_min	Minimum time a flow was active before becoming idle
idl_avg	Mean time a flow was idle before becoming active
idl_std	Standard deviation time a flow was idle before becoming active
idl_max	Maximum time a flow was idle before becoming active
idl_min	Minimum time a flow was idle before becoming active

label	Class label ("Benign" or type of attack)

Annex G Code for generating artificial capture

```
import time
from multiprocessing.context import Process

from scapy.layers.inet import IP, TCP, UDP
from scapy.sendrecv import send

# Each method corresponds to one or more feature vectors in the
output file.

def session1():
    """ Session #1 - This capture reflects the interaction between
node 10.0.0.1 and 10.0.0.2. It is a normal
    interaction between the two where a connection is established and
is then closed.
    TCP flags are set.
    Only one feature vector will result from this interaction.
    """
    send(IP(src="10.0.0.1", dst="10.0.0.2", tos=0b1) / TCP(sport=1,
dport=2, flags="S"), iface="Ethernet")
    send(IP(src="10.0.0.2", dst="10.0.0.1") / TCP(sport=2, dport=1,
flags="SA"), iface="Ethernet")
    send(IP(src="10.0.0.1", dst="10.0.0.2") / TCP(sport=1, dport=2,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.2", dst="10.0.0.1") / TCP(sport=2, dport=1,
flags="U"), iface="Ethernet")
    send(IP(src="10.0.0.1", dst="10.0.0.2") / TCP(sport=1, dport=2,
flags="F"), iface="Ethernet")
    send(IP(src="10.0.0.2", dst="10.0.0.1") / TCP(sport=2, dport=1,
flags="AF"), iface="Ethernet")
    send(IP(src="10.0.0.1", dst="10.0.0.2") / TCP(sport=1, dport=2,
flags="A"), iface="Ethernet")


def session2():
    """ Session #2 - This capture reflects the interaction between
node 10.0.0.1 and 10.0.0.4.
    It reflects a land attack where the src IP = dst IP and src port
= dst port.
    Two feature vectors will result from this interaction.
    """
    send(IP(src="10.0.0.1", dst="10.0.0.4", tos=0b00000010) /
UDP(sport=1, dport=4), iface="Ethernet")
    send(IP(src="10.0.0.4", dst="10.0.0.1") / UDP(sport=4, dport=1),
iface="Ethernet")
    send(IP(src="10.0.0.1", dst="10.0.0.4") / UDP(sport=1, dport=4),
iface="Ethernet")
    send(IP(src="10.0.0.4", dst="10.0.0.1") / UDP(sport=4, dport=1),
iface="Ethernet")
    send(IP(src="10.0.0.1", dst="10.0.0.1") / UDP(sport=1, dport=1),
iface="Ethernet")
    send(IP(src="25.0.0.1", dst="10.0.0.4") / UDP(sport=1, dport=4),
iface="Ethernet")
```

```
def session3():
    """ Session #3 - This capture reflects the interaction between
node 10.0.0.3 and 10.0.0.4.
    TCP flags are set.
    If the "--timeout" argument is set to 15s, this interaction
should result in one feature vector.
    """
    send(IP(src="10.0.0.3", dst="10.0.0.4") / TCP(sport=3, dport=4,
flags="S"), iface="Ethernet")
    send(IP(src="10.0.0.4", dst="10.0.0.3") / TCP(sport=4, dport=3,
flags="SA"), iface="Ethernet")
    send(IP(src="10.0.0.3", dst="10.0.0.4") / TCP(sport=3, dport=4,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.4", dst="10.0.0.3") / TCP(sport=4, dport=3,
flags="U"), iface="Ethernet")
    send(IP(src="10.0.0.3", dst="10.0.0.4") / TCP(sport=3, dport=4,
flags="AP"), iface="Ethernet")
    time.sleep(10)
    send(IP(src="10.0.0.4", dst="10.0.0.3") / TCP(sport=4, dport=3,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.3", dst="10.0.0.4") / TCP(sport=3, dport=4,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.4", dst="10.0.0.3") / TCP(sport=4, dport=3,
flags="A"), iface="Ethernet")
    time.sleep(10)
    send(IP(src="10.0.0.3", dst="10.0.0.4") / TCP(sport=3, dport=4,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.4", dst="10.0.0.3") / TCP(sport=4, dport=3,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.3", dst="10.0.0.4") / TCP(sport=3, dport=4,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.4", dst="10.0.0.3") / TCP(sport=4, dport=3,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.3", dst="10.0.0.4") / TCP(sport=3, dport=4,
flags="R"), iface="Ethernet")


def session4():
    """ Session #4 - This capture reflects the interaction between
node 10.0.0.5 and 10.0.0.6.
    TCP flags are set.
    If the "--timeout" argument is set to 15s, this interaction
should result in two feature vector.
    """
    send(IP(src="10.0.0.5", dst="10.0.0.6") / TCP(sport=5, dport=6,
flags="S"), iface="Ethernet")
    send(IP(src="10.0.0.6", dst="10.0.0.5") / TCP(sport=6, dport=5,
flags="SA"), iface="Ethernet")
    send(IP(src="10.0.0.5", dst="10.0.0.6") / TCP(sport=5, dport=6,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.6", dst="10.0.0.5") / TCP(sport=6, dport=5,
flags="APU"), iface="Ethernet")
    time.sleep(20)
    send(IP(src="10.0.0.5", dst="10.0.0.6") / TCP(sport=5, dport=6,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.6", dst="10.0.0.5") / TCP(sport=6, dport=5,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.5", dst="10.0.0.6") / TCP(sport=5, dport=6,
flags="A"), iface="Ethernet")
    time.sleep(5)
```

```
    send(IP(src="10.0.0.6", dst="10.0.0.5") / TCP(sport=6, dport=5,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.5", dst="10.0.0.6") / TCP(sport=5, dport=6,
flags="A"), iface="Ethernet")
    send(IP(src="10.0.0.6", dst="10.0.0.5") / TCP(sport=6, dport=5,
flags="A"), iface="Ethernet")
    time.sleep(5)
    send(IP(src="10.0.0.5", dst="10.0.0.6") / TCP(sport=5, dport=6,
flags="F"), iface="Ethernet")
    send(IP(src="10.0.0.6", dst="10.0.0.5") / TCP(sport=6, dport=5,
flags="AF"), iface="Ethernet")
    send(IP(src="10.0.0.5", dst="10.0.0.6") / TCP(sport=5, dport=6,
flags="A"), iface="Ethernet")


if __name__ == '__main__':
    p1 = Process(target=session1())
    p1.start()
    p2 = Process(target=session2())
    p2.start()
    p3 = Process(target=session3())
    p3.start()
    p4 = Process(target=session4())
    p4.start()
```